FULLY DYNAMIC GRAPH ALGORITHMS AND THEIR DATA STRUCTURES

Monika Rauch

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

January 1993

© Copyright by Monika Rauch 1993
All Rights Reserved

Abstract

We present four fully dynamic algorithms, i.e., algorithms that are able to answer queries about properties of a graph or a set during a sequence of modifications of the graph or set. Additionally, we give lower bounds for fully dynamic graph algorithms.

The first three algorithms maintain properties of graphs during insertions and deletions of edges and vertices. Let n denote the number of vertices in a graph and m the number of edges. We give the first sublinear time algorithm for maintaining biconnectivity in graphs. A query can be answered in constant time. The amortized time of an insertion or deletion is $O(m^{2/3})$. A slight variation of the algorithm gives an amortized running time of $O(\sqrt{n}\log n)$ per insertion or deletion in a plane graph. The worst-case running time per query is $O(\log^2 n)$ in a plane graph.

The second data structure maintains two-edge connectivity in plane graphs in a fully dynamic environment. It achieves a worst case running time of $O(\log^2 n)$ per insertion or deletion and $O(\log n)$ per query.

Finally, we give an algorithm for deciding whether adding an edge to a plane graph maintains the planarity of its embedding. If the edge can be added, the algorithm returns a face into which the edge can be embedded without destroying the planarity of the embedding. Any query and any insertion or deletion of an edge takes $O(\log^2 n)$ time. All of the algorithms use $O(n)$ space and isolated vertices may be inserted in constant time.

In addition, we give a lower bound in the cell probe model of $\Omega(\log n/\log\log n)$ per operation for fully-dynamic planarity testing and for fully-dynamic k-edge connectivity and k-vertex connectivity in plane graphs for constant k.

The fourth algorithm maintains predecessors in integer sets during insertions and

deletions of elements. The integers are chosen from the universe $\{0, 1, \ldots, u-1\}$. There is an upper bound of $O(\log \log u)$ for this problem [29] and a lower bound of $\Omega(\log \log u)$ per query operation (even for randomized algorithms) in the cell probe model [2]. If the set of numbers is uniformly distributed over the universe, we give an algorithm with worst case query time $O(1)$ and amortized expected time $O(1)$ for any insertion or deletion.

Acknowledgments

I am indebted to my advisor Bob Tarjan for the guidance that he provided during this work. He taught me how to do research by confronting me with challenging problems and supporting my attempts to solve them. I am also thankful for the time he spent reading this thesis.

I want to thank Bernard Chazelle for teaching me the art of writing. I am especially grateful for his comments and his patience while reading preliminary versions of this work.

Additionally I want to thank Andrew Yao for reading this thesis and making helpful comments.

The discussions with Brandon Dixon and his encouragement provided important support for this work.

I need to thank John Hershberger and Subhash Suri with whom I did the work in Chapter 3 and Giuseppe Italiano and Han LaPoutré with whom I collaborated on the work of Chapter 4.

The continuous support of my parents made it possible for me to study in Princeton.

I would also like to thank Rebecca Davis who helped me with all Latex questions and Melissa Lawson who was very helpful with all the paperwork.

This research was supported in part by a Fellowship of the German National Fellowship Foundation, Studienstiftung des deutschen Volkes, and by a Wallace Memorial Honorific Fellowship from Princeton University.

Contents

Abstract iii

Acknowledgments v

1 Introduction 1

2 Biconnectivity in Graphs 5

2.1 Introduction 5

2.2 Partitioning the Graph 7

2.3 The Internal Data Structure 12

2.3.1 Description of the Internal Data Structure 13

2.3.2 Updating the Internal Data Structure 22

2.4 The Shared Data Structure 26

2.4.1 The Local Data Structure 27

2.4.2 The Global Data Structure 39

2.4.3 The Look-up Data Structure 40

2.5 Updates 45

2.6 Queries 50

3 Two-Edge Connectivity in Plane Graphs 53

3.1 Introduction 53

3.2 Preliminaries 54

3.3 Topology trees 55

3.4 Edge Bundles 59

3.5 Supernodes and Coverage Graphs 61
3.6 Recipes 63
3.7 Queries 69
3.8 Updates 70
3.8.1 Inserting an Edge 70
3.8.2 Deleting a Non-Tree Edge 71
3.8.3 Deleting a Tree Edge 72
3.8.4 Merging the Topology Tree 72

4 Planarity Testing in Graphs **77**
4.1 Introduction 77
4.2 Preliminaries 78
4.3 Coverage Lists and Face Lists 80
4.4 Recipes 82
4.5 Queries 90
4.6 Updates 91
4.6.1 Merging the Topology Tree 92

5 Lower Bounds for Fully Dynamic Problems **95**
5.1 Introduction 95
5.2 The Lower Bounds 96

6 Finding Predecessors in Integer Sets **104**
6.1 Introduction 104
6.2 A Simple Solution 106
6.3 An Improved Solution 115

7 Conclusions **116**

Bibliography **118**

List of Figures

1 A graph G and the transformed graph G'. 8
2 The vertices u and v are biconnected in G, but not in G'. 9
3 A restricted partition of order 4 of a graph and the corresponding high-level graph H. Non-tree edges are dotted. 10
4 Two clusters and all possible edge bundles between them. 11
5 The graphs $G_a(C(x))$ and $G_b(C(x))$ of the graph in Figure 3. 13
6 A graph and its block tree. 17
7 The new high-level graph and the new graph $G_a(C(x))$ if the tree edge e of H is deleted and the non-tree edge e' becomes a new tree edge in the graph of Figure 3. 23
8 A graph G with a shared vertex s and the local data structure of s. . 28
9 The high-level graph H, a plane multigraph containing all neighboring clusters of s and the multigraph with the artificial edges in bold. . . . 31
10 The tree edge e is replaced by the non-tree edge e' in C. Thin edges are tree edges of C, thick edges connect outside vertices representing C. 32
11 The tree edge e of C is deleted. Thin edges are tree edges of C, thick edges connect outside vertices representing C. 34
12 The edge (u, v) is inserted. Thin edges are tree edges of $C(u)$, thick edges connect outside vertices representing $C(u)$. 35
13 The clusters on $\pi(u, v)$. If all coverage bits are one, the boxes indicate the blocks on $\pi(u, v)$. 48
14 Expanding the topology tree at vertex u. 57
15 Different targeting schemes for edge bundles. 60

16 Changes in the path covered by an edge. 61
17 Storing the coverage graph. 63
18 Merging a degree-3 cluster with a degree-1 cluster (Case 2). 64
19 Merging a degree-1 cluster with a degree-1 cluster (Case 3). 65
20 Merging a degree-2 cluster with a degree-1 cluster (Case 4). 65
21 Merging a degree-2 cluster with a degree-2 cluster (Case 5). 66
22 Before retargeting: All bundles in the cluster graph with target C. The number inside a bundle indicates its number of edges. 68
23 The cluster graph after the retargeting. 69
24 A cluster with is edge bundles and coverage lists. 81
25 Merging a degree-3 cluster with a degree-1 cluster (Case 2). 84
26 Merging a degree-1 cluster with a degree-1 cluster (Case 3). 84
27 Merging a degree-2 cluster with a degree-1 cluster (Case 4). 85
28 Merging with self–loops (Case 4). 86
29 Merging a degree-2 cluster with a degree-2 cluster (Case 5). 87
30 The plane graph for the array 8 3 2 4 0 6 7 2 1 3 4. 97
31 The plane graph of the array 8 3 2 5 0 6 7 2 1 3 4. 99
32 The plane graph constructed for the array 3 2 5 1 7 6 to show a lower bound for 3-edge connectivity. 101
33 The plane graph constructed for the array 3 2 5 1 7 6 to show a lower bound for 3-vertex connectivity. 102
34 An array a with bucket pointers and buckets. 106

Chapter 1

Introduction

Research on dynamic algorithms has been particularly active in recent years. Dynamic algorithms involve a sequence of intermixed updates and queries. For instance, in maintaining a graph dynamically, an algorithm must handle requests such as insertions or deletions of edges or vertices and queries asking about properties of the graph. If the algorithm maintains a set of integers, an update inserts or deletes an element of the set and a query asks about properties of the set. For all of the graph properties that we discuss linear-time algorithms already exist. Therefore, our goal is to build a data structure that can support updates and queries in sublinear time. If the update operations are restricted to only insertions or only deletions, the algorithm is called *semi-dynamic*. If both operations are allowed, the algorithm is called *fully dynamic*.

In this dissertation we present fully dynamic graph algorithms for maintaining biconnectivity, two-edge connectivity, and planarity. Additionally, we give lower bounds for fully dynamic graph algorithms in embedded planar (also called *plane*) graphs. Finally we describe a fully dynamic algorithm for finding predecessors in integer sets.

Connectivity in graphs has received considerable attention in the past. Using depth-first search, Hopcroft and Tarjan designed linear-time algorithms for computing the biconnected and two-edge connected components of a graph and for testing the planarity of a graph [17]. However, most applications of these algorithms —for example in interconnection networks, CAD/CAM, and distributed computing— use

graphs that are not static, but change over the course of the algorithm. The previous algorithms require that the computation is repeated from scratch when a change occurs. Dynamic algorithms whose running time is sublinear in the size of the graph are more efficient alternatives than frequent recomputation. Until recently, semi-dynamic algorithms that allow only insertions were studied. The solutions for biconnectivity, two-edge connectivity and planarity testing use a quasi-linear union-find data structure and achieve $O(\alpha(m,n))$ amortized time per operation, where n is the number of vertices in the graph, m is the number of edges, and α is a functional inverse of Ackermann's function [27, 19].

The problem becomes significantly harder if only deletions are allowed. Except for the special case of planar graphs [12], no faster algorithms for the semi-dynamic problem with deletions-only are known than those for the fully dynamic version. The best known result for fully dynamic connectivity and two-edge connectivity in general graphs is an algorithm requiring $O(\sqrt{n}\log(m/n))$ time per update and constant time per query [6].

Chapter 2 of this dissertation gives the first sublinear algorithm for fully dynamic biconnectivity in general graphs. It achieves $O(m^{2/3})$ amortized time per update and constant time per query. Our algorithm uses linear space and preprocessing time. Independently Eppstein et al. [7] have developed an algorithm for this problem whose running time per update is $O(n\log(m/n))$ and whose running time per query is $O(1)$. For a plane graph we achieve $O(\sqrt{n}\log n)$ time per update and $O(\log^2 n)$ time per query. The update time can be improved to $O(\sqrt{n\log n})$, but for ease of description the $O(\sqrt{n}\log n)$-time solution is presented. The best previously known solution in plane graphs requires time $O(n^{2/3})$ per update or query [12]. This running time was recently improved to $O(\sqrt{n})$ [18].

When restricted to plane graphs, fully dynamic connectivity can be maintained in time $O(\log n)$ per update or query [6]. In Chapter 3 we give the fastest known algorithm for fully-dynamic two-edge connectivity in plane graphs. It uses time $O(\log^2 n)$ per update and time $O(\log n)$ per query. It requires linear space and linear preprocessing time. This algorithm is joint work with Subhash Suri and John Hershberger and was developed while the author was working at Bell Communications Research.

The best previously known algorithm takes time $O(\log^3 n)$ per update and $O(\log n)$ per query [9].

Many fully dynamic connectivity algorithms for plane graphs expect to be given as part of the input the face of the embedding where a new edge is to be inserted. These algorithms do not test whether this insertion preserves the planar embedding. If the embedding is not maintained, the correctness of the algorithm is not guaranteed. This is the motivation for an algorithm that maintains a planar embedding of a graph under insertions and deletions and answers queries of the form: "Can an edge be inserted between vertex u and vertex v in a way that keeps the embedding planar?". If so, the algorithm returns a face in which the edge (u, v) can be embedded without destroying the planarity of the embedding. Given a face and two vertices as input, the algorithm can also determine whether an edge between the two vertices can be inserted in the given face. The running time is $O(\log^2 n)$ per operation. The algorithm uses linear space and preprocessing time. This algorithm represents joint work with Han LaPoutré and Giuseppe Italiano. A solution to a version of this problem that allows arbitrary insertions but only restricted deletions achieves running time $O(\log n)$ per operation [23].

In Chapter 5 we obtain a lower bound in the cell probe model for fully-dynamic planarity testing in plane graphs. We show that for each fully-dynamic planarity testing algorithm there exists a sequence of m operations for which the algorithm takes $\Omega(m \log n / \log \log n)$ time. We prove the bound by reduction from the modular prefix sum problem for which a lower bound was shown in [11]. We also present a similar reduction that shows a lower bound of $\Omega(\log n / \log \log n)$ on the amortized time per operation for fully-dynamic k-edge connectivity and k-vertex connectivity in plane graphs, where k is a constant.

Chapter 6 presents an algorithm for finding predecessors in an integer set S during a sequence of insertions and deletions of elements of S. The set S is a subset of the universe $\{0, 1, \ldots, u-1\}$. There exists a well-known algorithm that achieves time $O(\log \log u)$ per operation [30]. Ajtai [2] proves a lower bound of $\Omega(\log \log u)$ per query operation (even for randomized algorithms) in the cell probe model. We make the assumption that the integers in S are uniformly distributed over the universe and

give an algorithm in this case with worst-case query time $O(1)$ and amortized expected time $O(1)$ for any insertion or deletion. Our algorithm needs $O(u^\epsilon)$ preprocessing time to build a look-up table of size $O(u^\epsilon)$. Additionally it uses $O(n)$ space, where n is the size of S.

The last chapter contains concluding remarks and open problems. The results of Chapter 2 have appeared in [21] and the results of Chapter 3 have appeared in [16].

Chapter 2

Biconnectivity in Graphs

2.1 Introduction

An undirected graph $G = (V, E)$ is *biconnected* if there are at least two vertex–disjoint paths from each vertex to every other vertex. It is well known [1] that a graph G is not biconnected if and only if there is a vertex whose removal disconnects G. Such a vertex is called an *articulation point* of G. An articulation point s *separates* two vertices u and v if its removal disconnects u and v. A graph that is not biconnected is naturally partitioned into *biconnected components* or *blocks*. We say that two vertices are *biconnected* if they are in a common biconnected component.

Similarly a graph is *two-edge connected* it there are at least two edge-disjoint paths from each vertex to every other vertex. An edge whose removal disconnects G is called a *bridge*. Note that biconnectivity implies two-edge connectivity, but not the other way around. Any dynamic algorithm for biconnectivity can be transformed into an dynamic algorithm for two-edge connectivity with the same time and space bounds. This shows that fully dynamic maintenance of biconnectivity is as least as hard as fully dynamic maintenance of two-edge connectivity.

Maintaining blocks is complicated by the fact that, unlike two-edge connectivity, biconnectivity is not an equivalence relation on the vertices of a graph. Additionally standard techniques to transform G into a bounded–degree graph, which are used by

all the previous fully dynamic connectivity algorithms, preserve connectivity and two-edge connectivity, but destroy the biconnectivity properties of G. We present a data structure that maintains biconnected components under a sequence of the following operations in arbitrary order:

$Insert(u, v)$: Insert an edge between the two vertices u and v.

$Delete(u, v)$: Delete the edge (u, v).

$Query(u, v)$: Return *true* if vertices u and v are in the same biconnected component and an articulation point separating u and v otherwise.

We give an algorithm for general graphs with $O(m^{2/3})$ amortized update time and $O(1)$ query time, where m is the number of edges in the graph. For a plane graph with n vertices we achieve $O(\sqrt{n}\log n)$ time per update and $O(\log^2 n)$ time per query operation. The update time can be speeded up to $O(\sqrt{n\log n})$, but for ease of description we present the $O(\sqrt{n}\log n)$-time solution in this work. Our algorithm uses linear space and preprocessing time.

The new techniques for fully dynamic graph algorithms introduced in this chapter involve precomputation and "lazy" updating. We split the graph into connected subgraphs and build a data structure for each subgraph which allows us to answer queries in the subgraph efficiently. If an edge is inserted or deleted, the data structures of the subgraphs maintaining its endpoints are updated. To avoid reconfiguring the entire data structure at each update we adopt a lazy strategy that spreads the reconfiguration costs over time. With this approach we are able to show that during l insertions and deletions only l data structures have to be updated. We also create a new data structure to handle the problems created by the transformation of G into a bounded-degree graph.

In the next section we introduce our terminology and explain the partitioning of the graph. Sections 2.3 and 2.4 present the data structure. Sections 2.5 and 2.6 describe updates and queries, respectively.

2.2 Partitioning the Graph

Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Let $E_1, \ldots, E_h$ be a partition of E into equivalence classes such that two edges e and e' lie in the same class if and only if either $e = e'$ or there is a simple cycle containing both e and e'. For $1 \leq i \leq h$, let V_i be the set of vertices of the edges in E_i. Each subgraph $G_i = (V_i, E_i)$ is called a *biconnected component* or *block* of G. Every edge belongs to exactly one biconnected component. A vertex is an articulation point if and only if it is contained in more than one block.

For expository convenience we assume that G is always connected. If the graph is not connected, we maintain the data structures described in the following sections for each connected component. If an insertion connects two different connected components, then the data structure of the two connected components can be joined in the time bounds for an insertion. If a deletion disconnects a connected component, the algorithm described below splits the data structure of the connected component in two data structures, one for each new connected component.

Let T be a spanning tree of G. We denote by $\pi_G(u, v)$ the spanning tree path from u to v in G. If G is understood, we use $\pi(u, v)$. Note that the definition always refers to the same T. We will make use of certainly tree traversals like preorder and postorder traversal [24].

We call two adjacent tree edges (w, x) and (x, y) *covered* if there exists a path from w to y that does not contain x. Otherwise, the two tree edges are called *uncovered*. Obviously, if (w, x) and (x, y) are uncovered, then x is an articulation point separating w from y. We use this property of edges to test whether two vertices are biconnected.

Lemma 2.2.1 *Let u and v be vertices of G. Then u and v are biconnected if and only if every pair of adjacent edges on $\pi(u, v)$ is covered.*

Proof: Let (w, x) and (x, y) be edges on $\pi(u, v)$. The vertex x is an articulation point separating u and v if and only if there is no path from w to y that does not use x and thus if and only if (w, x) and (x, y) are not covered. It follows that u and v are biconnected if and only if there is not an articulation point

on $\pi(u,v)$ separating u and v, and thus, if and only if every pair of adjacent edges on $\pi(u,v)$ is covered. ∎

To split the graph into small connected subgraphs, we use a graph of bounded degree. We apply a well-known transformation [14] to map G to a graph G', whose vertices have degree no greater than 3. For a vertex x of G of degree $d \geq 4$, with $y_1, \ldots, y_d$ the vertices adjacent to x (in the clockwise order of the embedding in the plane case), we replace x with new vertices $x_1, \ldots, x_d$ and replace the edge (x, y_i) with a *solid* edge (x_i, y_i), $1 \leq i \leq d$. Furthermore, we add new *dashed* edges (x_i, x_{i+1}), $1 \leq i \leq d-1$. See Figure 1. We call this the *expansion* of x and we say that the

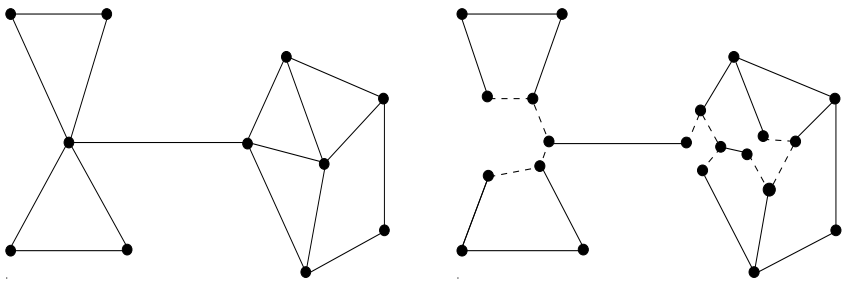

Figure 1: A graph G and the transformed graph G'.

edge (x_i, x_{i+1}) *belongs* to x. If the graph is embedded, we add the dashed edges so that $(x_i, x_{i-1}), (x_i, y_i), (x_i, x_{i+1})$ is the clockwise order of the edges at x_i. Thus each vertex in G' is incident to at most two dashed edges and each edge in G corresponds to a unique solid edge in G'. The number of vertices and edges of G' is $O(m+n)$. If G is a plane graph, then this transformation gives a planar embedding for G'.

To *contract* a dashed edge (x, y), we remove (x, y) and identify x with y. We need dashed edges for partitioning the graph. However, two vertices u and v that are not biconnected in G can be biconnected in G'. See Figure 2. Thus for answering a $query(u, v)$ operation, we have to contract dashed edges. The next lemma shows that contracting the dashed edges of all vertices on $\pi_{G'}(u, v)$ is sufficient for correctly answering queries.

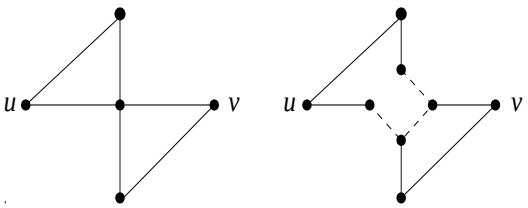

Figure 2: The vertices u and v are biconnected in G, but not in G'.

Lemma 2.2.2 *Let G'' be the graph that results from G' by contracting all dashed edges of every vertex on $\pi_{G'}(u,v)$. Two vertices u and v are biconnected in G if and only if they are biconnected in G''.*

Proof: The graph G'' can be created from G by expanding appropriate vertices. Thus if u and v are biconnected in G, they are biconnected in G''.

If u and v are not biconnected in G, the articulation point x separating u and v in G has to lie on $\pi_G(u,v)$ and thus x is represented by only one vertex in G''. If x is no articulation point in G'' separating u and v, then there exists a path P in G'' from u to v that does not use x. But P has a corresponding path in G from u to v that does not use x, which leads to a contradiction. This implies that x is also an articulation point in G'' separating u and v and hence, u and v are not biconnected in G''. ∎

We give each solid edge of G' cost 2 and each dashed edge cost 1. This guarantees that all the dashed edges are contained in any minimum spanning tree of G'. A standard linked list representation of G' can be generated in time $O(m+n)$ and a minimum spanning tree T of G' can be determined in time $O(m+n)$ by computing a spanning forest of all edges of cost 1 and then connecting the spanning forest into a spanning tree with cost 2 edges.

Next we find a partition of G' along the lines of [9]. A *cluster* with respect to T is a set of vertices such that the subgraph of T induced on the cluster is connected. The *external degree* of a cluster is the number of tree edges with precisely one endpoint in the cluster. A *restricted partition of order k* with respect to T is a partition of the set of vertices of T such that

1. Each set in the partition is a vertex cluster with external degree at most 3 and of cardinality at most k.
2. Each cluster with external degree 3 is of cardinality 1.
3. No two adjacent clusters can be combined and still satisfy 1 and 2.

The partition splits T into clusters of size at most k. If the external degree of a cluster is 1 or 2, it is easy to handle. If the external degree is 3, the definition implies that the cluster contains only one vertex x and that x is incident to three tree edges. Since every vertex in G' has at most degree 3, x is not incident to a non-tree edge. Thus clusters with external degree 3 are not incident to non-tree edges. Figure 3 depicts a restricted partition of order 4. The number of clusters in a restricted partition of order k is $\Theta(m/k)$. A restricted partition can be found in time $O(m+n)$ [9].

We extend this definition slightly in the following way: An edge with exactly one endpoint in a cluster C is called *incident* to C. We require that all dashed edges incident to a cluster belong to the same vertex of G. Since each dashed edge is a tree edge and vertices of degree 3 are incident to at most two dashed edges, each cluster is incident to at most two dashed edges. Thus, if the dashed edges of a cluster C belong to two different vertices of G, we split C into two clusters, each incident to one dashed edge. This increases the time to find a partition and the number of clusters in the partition only by a constant factor.

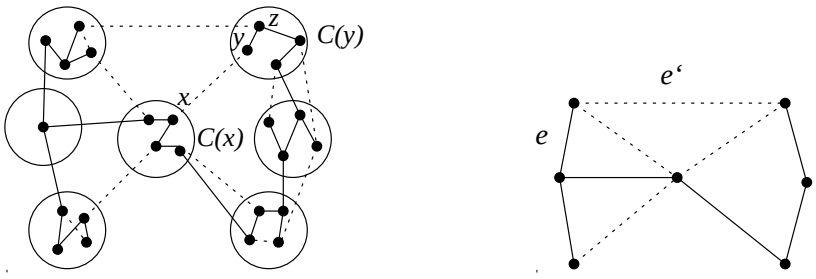

Figure 3: A restricted partition of order 4 of a graph and the corresponding high-level graph H. Non-tree edges are dotted.

The restricted partition defines the *high-level graph* H of G'. Its vertices are the clusters. If G is a general graph, there is an edge between two vertices C and C' of

H if and only if there exists an edge from a vertex in the cluster C to a vertex in the cluster C' (see Figure 3).

In a plane graph the edges incident to a cluster C are embedded according to a fixed order around C. We say that two edges e_1 and e_2 are *equivalent* if they are connecting the same two clusters C and C' and

1. $e_1 = e_2$ or
2. e_1 is a neighbor of e_2 in the embedding at C and C' or
3. there exists an edge e_3 that is equivalent to e_1 and to e_2.

An *edge bundle* is an equivalence class on the set of all edges according to this equivalence relation. Since each cluster that is incident to an edge bundle has external degree 3 and the graph is planar, it follows that there are at most 3 edge bundles between two clusters (see Figure 4). The high-level graph H of a plane graph contains as edges all edge bundles of G. Thus, H is a plane graph with $O(m/k)$ edges.

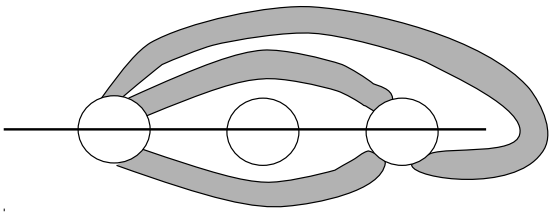

Figure 4: Two clusters and all possible edge bundles between them.

The cluster that contains the vertex u is denoted $C(u)$ and a pointer to it is stored at u. If (x, y) is a tree edge in G and $C(x) \neq C(y)$, then the edge in H representing (x, y) is a tree edge. If (x, y) is dashed, then the edge representing it in H is dashed. The tree edges define a spanning tree of H.

An edge with both endpoints in C is called an *internal edge* of C. The edge from x to y in Figure 3 is incident to $C(y)$, while the edge from y to z is an internal edge of $C(y)$. A tree neighbor of a cluster C in the spanning tree of H is called a *tree neighbor* of C. A *non-tree neighbor* of C is a neighbor of C in H that is not connected to C by a tree edge. The *high-level degree* of a cluster is the number of edges incident to the cluster in H. In Figure 3 the high-level degree of $C(y)$ is 3 and of $C(x)$ is 5.

For two clusters C and C' there is a vertex $b(C, C')$ of C such that the tree path from any vertex of C to any vertex of C' goes through $b(C, C')$. Note that $b(C, C') \neq b(C', C)$.

For $d > 3$, a vertex s in G of degree d is represented in G' by a chain of $d-1$ dashed edges and d vertices of degree 3. If the endpoints of l of these dashed edges, $0 < l < d$, are in different clusters (i.e. belong to a dashed edge bundle in the high-level graph), then s is *shared* by $l + 1$ clusters and we call s a *shared vertex*. Since all the dashed edges incident to a cluster belong to the same shared vertex, a cluster shares at most one vertex. This implies that there are $O(m/k)$ shared vertices. We call a cluster that does not share s, but that is a tree neighbor of a cluster sharing s a *tree-neighboring cluster of a shared vertex* s. A cluster that does not share s and that is connected to clusters sharing s only by non-tree edges is called a *non-tree-neighboring cluster of* s.

To maintain a graph whose vertices are of degree at most 3 during the insertion of an edge (u, v), we may have to add a constant number of vertices to G'. In a general graph this increases the number of vertices in a cluster. In a plane graph the new vertices have to be added consistently with the embedding. Thus, either the number of vertices in a cluster or the number of clusters (with external degree 3) increases by a constant. During a deletion these numbers are only increased if the vertices in a cluster become disconnected and the cluster has to be split into a constant number of clusters. We want to keep as an invariant that each cluster has at most $2k$ vertices and that there are $O(m/k)$ clusters. This implies that we rebuild the complete data structure every $\min\{k, m/k\}$ insertions and every m/k deletions. As we show later, the cost of a complete reconstruction is $O(m)$. Therefore, the reconstructions add an amortized cost of $O(k)$ to each of the m/k deletions and an amortized cost of $O(m/k + k)$ to each of the $\min\{k, m/k\}$ insertions.

2.3 The Internal Data Structure

The data structure described in this section is used to test in constant time whether two vertices that are in a cluster C or that are connected to C by a tre edge are biconnected in G. Since a cluster with external degree 3 consists of only one vertex

that is incident to three tree edges, it follows that we can answer the question above in constant time provided we know the blocks of H and whether the shared vertex of the cluster is an articulation point. The latter question can be answered with the data structure of Section 2.4. Thus we can assume in the following section that every cluster for which we build an internal data structure has external degree at most 2.

2.3.1 Description of the Internal Data Structure

We maintain a data structure at each cluster C, which is called the *internal data structure* of C. It consists of a graph $G(C)$ and a graph $G(c)_a$. The graph $G(C)$ comprises all the vertices and internal edges of C. The internal dashed edges of C are contracted in $G(C)$. Additionally $G(C)$ contains a *big vertex* for every cluster that is adjacent to C. If $u \in C$, then all edges of the form (u, v), where $v \in C'$, are represented as a single edge from u to C' in $G(C)$ and stored alongside (u, C'). There are no edges between big vertices in $G(C)$. The graph $G_a(C)$ contains all the vertices and edges of $G(C)$ and additional edges between big vertices (see below).

Figure 5: The graphs $G_a(C(x))$ and $G_b(C(x))$ of the graph in Figure 3.

After a delete operation it is possible that two or more clusters that are adjacent to C and biconnected in H are represented by the same big vertex in $G(C)$. We show, however, that this does not influence the correctness of the answers computed by using the internal data structure. Note that this data structure does not make use of the embedding in the planar case. Since we guarantee that C has at most $2k$ vertices of degree at most 3, $G(C)$ consists of $O(k)$ edges and vertices and the total size of all internal data structures is $O(m)$.

By adding *artificial* edges (i.e. edges that are not necessarily in G) between big

vertices of $G(C)$ we create the graph called $G_a(C)$. If C_1 is a tree neighbor of C, we add the artificial edge (C_1, B) for every non-tree neighbor B of C that is connected to C_1 by a tree path not containing C (see Figure 5). In the graphs $G(C)$ and $G_a(C)$ two vertices of C that are biconnected in either of them are also biconnected in G (under the additional condition that C does not share a vertex). Still they cannot be used for answering biconnectivity questions of vertices in C, since the converse of this statement does not always hold. The following graph $G_b(C)$, however, has the desired property that two vertices of C are biconnected in $G_b(C)$ if and only if they are biconnected in C (again under the additional condition that C does not share a vertex). Thus it is used for answering biconnectivity queries. We define $G_b(C)$ to be $G_a(C)$ if C has external degree 1. If C has external degree 2, let C_1 and C_2 be its tree neighbors. Then $G_b(C)$ is defined to be $G_a(C)$ if C_1 and C_2 are not biconnected in H, and $G_a(C)$ with the addition of edge (C_1, C_2) if C_2 and C_2 are biconnected in H.

Let C_T be the set of vertices of C with the vertices that are connected to vertices of C by a solid tree edge. In the following two lemmas we prove that the biconnectivity of two vertices of C_T can be decided using $G_b(C)$. The next lemma shows that $G_b(C)$ returns the same answer to biconnectivity queries, if two biconnected neighbors of C are represented by one big vertex in $G_b(C)$ instead of by two big vertices. Thus the relaxation of the data structure after edge deletions does not influence the correctness.

Lemma 2.3.1 *Let C be a cluster and let B_1 and B_2 be two neighbors of C that are biconnected in H. Let $G_b(C)'$ be the graph that is created from $G_b(C)$ by joining B_1 and B_2 to one vertex B and by removing parallel edges. The vertices u and v of C_T are biconnected in $G_b(C)'$ if and only if they are biconnected in $G_b(C)$.*

Proof: Since B_1 and B_2 are biconnected in H, they are connected by a path of artificial edges in $G_b(C)$. Let u and v be two vertices of C_T. They are connected by a tree path that does not contain B (except as possibly an endpoint). If u and v are biconnected in $G_b(C)'$, there is path P from u to v in $G_b(C)'$ that does not use any vertex on $\pi(u, v)$ (except u and v). If P does not contain B, then P is also a path from u to v in $G_b(C)$ that does not use a vertex on $\pi(u, v)$ and hence, u and v are biconnected in $G_b(C)$.

If P contains B, let (x, B) and (B, y) be the edges on P. Without loss of generality, let us assume that x is incident to B_1 in $G_b(C)$. If y is incident to B_1 are well, then p with B replaced by B_1 is a path from u to v in $G_b(C)$ that does not use a vertex of $\pi(u, v)$. If y is not incident to B_1, it is incident to B_2. Thus P with B replaced by B_1, B_2, and the path (of artificial edges) between B_1 and B_2 is a path in $G_b(C)$ that is vertex-disjoint from $\pi(u, v)$. Hence in both cases u and v are biconnected in $G_b(C)$.

To show the other direction let us assume that u and v are biconnected in $G_b(C)$. This implies that there is a path P from u to v in G that is vertex-disjoint from $\pi(u, v)$. If P does not contain B_1 and B_2, then P is also a path in $G_b(C)'$ from u to v that does not contain a vertex on $\pi(u, v)$. If P contains B_1 and/or B_2, let P' be the path that is created from P by substituting B_1, B_2, and the path between them by B. The path P' is a path from u to v in $G_b(C)'$ that is vertex-disjoint from $\pi(u, v)$. This implies in both cases that u and v are biconnected in $G_b(C)'$. ∎

Now we want to show that $G_b(C)$ can be used to test the biconnectivity of two vertices of C_T. If C does not share a vertex, two vertices of C_T are biconnected in $G_b(C)$ if and only if they are biconnected in G. If C shares a vertex, two vertices of C_T are biconnected in G if and only if they are biconnected in $G_b(C)$ and the shared vertex of C does not separate them.

Lemma 2.3.2 *Two vertices u and v of C_T are biconnected in G if and only if they are biconnected in $G_b(C)$ and the shared vertex of C is not an articulation point separating u and v.*

Proof: Let G'' be G' where all dashed edges of vertices on $\pi(u, v)$ that are internal to C are contracted. Note that the big vertices representing two neighbors of C are biconnected in $G_b(C)$ if and only if they are biconnected in H. We show next that two vertices of C_T are biconnected in $G_b(C)$ if and only if they are biconnected in G''.

A cluster C' is *contracted* if all the internal edges of C' are contracted. Note that $G_b(C)$ can be constructed from G'' by contracting every cluster other than

C to one vertex, by removing parallel edges, and by contracting all (remaining) internal dashed edges of C. Thus if there are two vertex-disjoint paths between u and v in $G_b(C)$, then there are two such paths in G''.

To prove the other direction, note that all vertices on $\pi_G(u,v)$ are in C (except possibly the endpoints). Thus if u and v are biconnected in G'', there exists a second path P that does not use any vertices on the tree path $\pi(u,v)$ in G''. Every subpath of P that consists of a sequence of vertices from a neighboring cluster C' of C certainly contains no vertex of $\pi(u,v)$. Let P' be the path that is created from P by contracting every such subpath to a vertex and by contracting dashed edges internal to C. It follows that P' is a path in $G_b(C)$ that is vertex-disjoint from $\pi(u,v)$. Thus u and v are biconnected in $G_b(C)$.

Let s be a vertex shared by C if it exists. Now we want to show that two vertices of C_T are biconnected in G if and only if they are biconnected in G'' and s is not an articulation point in G separating u and v. If s does not lie on $\pi_G(u,v)$, all dashed edges of every vertex on $\pi_{G'}(u,v)$ are contracted in G''. By Lemma 2.2.2, u and v are biconnected in G if and only if they are biconnected in G''. If s lies on $\pi(u,v)$, Lemma 2.2.2 shows that u and s are biconnected in G'' if and only if they are biconnected in G. The same is true for s and v. Thus u and v are biconnected in G if and only if u and s are biconnected in G, s and v are biconnected in G, and s is no articulation point in G separating u and v. But as we just have shown this is equivalent to u and s being biconnected in G'', s and v being biconnected in G'', and s not being an articulation point in G separating u and v. This is true if and only if u and v are biconnected in G'' and s is not an articulation point in G separating u and v. As shown in the previous paragraph this is true if and only if u and v are biconnected in $G_b(C)$ and the shared vertex of C is not an articulation point in G separating u and v. ∎

In our data structure artificial edges are not added to our representation of $G(C)$, but to a contracted version of $G(C)$. Our goal is to store $G(C)$ in a contracted graph

so that $d-2$ artificial edges between d big vertices can be added (or removed) in time $O(d)$ and that queries whether two vertices are biconnected in the resulting graph can be answered in time $O(1)$. Thus this graph should have size $O(d)$. To contract $G(C)$ we build the *block tree* of $G(C)$ [14] by making each block a round node and each vertex of $G(C)$ a square node in the block tree. There is an edge from the round node of a vertex x of $G(C)$ to the square node of every block that contains x in $G(C)$. See Figure 6. Note that every square node with degree larger than 1 represents an articulation point of $G(C)$.

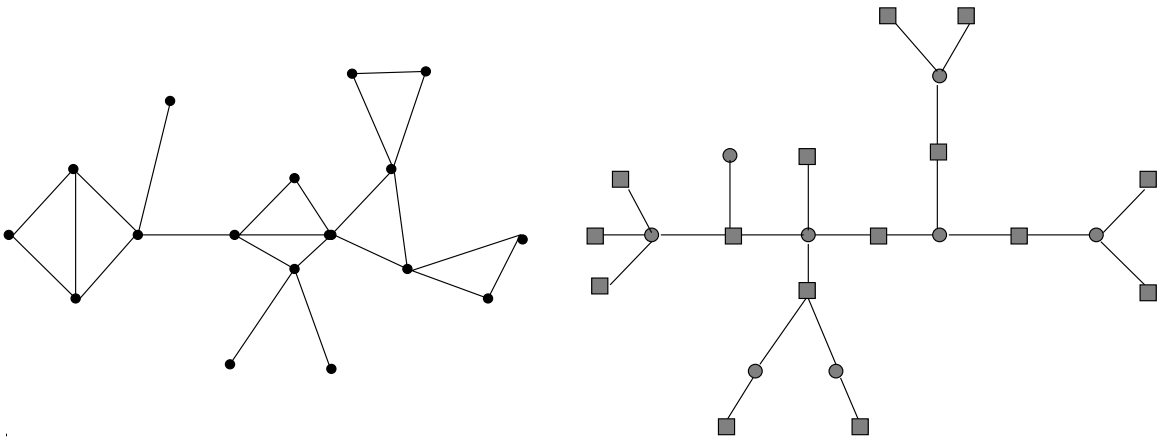

Figure 6: A graph and its block tree.

The contracted graph $Comp(C)$ is created from the block tree in the following way: We remove all edges and nodes of the block tree that are not contained on a simple path between big vertices of C. If we discard a vertex x and all its blocks of $G(C)$, then x is only biconnected with the vertices of its blocks. Let $x_0, x_1, \ldots, x_{l+1}$ for $l \geq 1$ be a path in the resulting block tree such that the degree of x_i for $1 \leq i \leq l$ is 2 and the degree of x_0 and x_{l+1} is not 2. We replace the chain $x_2, \ldots, x_{l-1}$ by one vertex r and we replace $(x_1, x_2) \ldots (x_{l-1}, x_l)$ by (x_1, r) and (r, x_l). We store a pointer to r at every node x_i, $(2 \leq i < l)$. Then we add all artificial edges to $Comp(C)$. (Note that every big vertex of $G(C)$ is represented by its own round node in $Comp(C)$, since it belongs to only one block of $G(C)$ and thus has degree 1 in the block tree.)

The graph $Comp(C)$ has size $O(d)$, where d is the high-level degree of C. Each node r of the contracted block tree represents either a single vertex x of $G(C)$ or one

block b of $G(C)$ or a collection of adjacent blocks and vertices of degree two. In the first case r is the *representative* of x, in the second case r is the representative of every vertex in b that does not have a representative on its own, and in the third case r is the representative for every vertex that does not have a representative on its own and that belongs to one of the blocks in the collection.

The data structure that we actually maintain for C consists of the graph $G(C)$ and the graph $Comp(C)$. We introduced $G_a(C)$ and $G_b(C)$ only for ease of description. At each edge of $G(C)$ we remember its block of $G(C)$ and at each vertex its representative in $Comp(C)$. We use $Comp(C)$ to test whether two vertices of $G_a(C)$ are biconnected in the following way.

Lemma 2.3.3 *Let x_1 and x_2 be two vertices of $G(C)$ and let r_1 and r_2 with $r_1 \neq r_2$ be their representatives in $Comp(C)$. The nodes x_1 and x_2 are biconnected in $G_a(C)$ if and only if every articulation point separating r_1 and r_2 in $Comp(C)$ is a node that represents a block of $G(C)$.*

Proof: Every square node of degree larger than 1 on the path from u to v in the block tree is an articulation point of $G(C)$ separating u and v. If there is an articulation point on $\pi_{Comp(C)}(r_1, r_2)$ that separates r_1 and r_2 and represents a single node x of $G(C)$, then x is also an articulation point separating x_1 and x_2 in $G_a(C)$. Similarly if there is an articulation point on $\pi_{Comp(C)}(r_1, r_2)$ that represents a collection of nodes and blocks, then each node in the collection is an articulation point separating x_1 and x_2 in $G_a(C)$.

To show the other direction let us consider the case where no node representing a vertex of $G(C)$ or a collection of vertices and blocks separates r_1 and r_2. Then the only articulation points separating r_1 and r_2 are nodes that represent one block of $G(C)$. Let $a_1, a_2, \ldots, a_l$ be these articulation points (that are blocks of $G(C)$) in increasing order of their distance from r_1. We show that x_1 and x_2 are biconnected in $G_a(C)$.

For every path in $Comp(C)$ there is a corresponding path in the block tree with the addition of the artificial edges and for every such path in the block tree there is a corresponding path in $G_a(C)$ that uses as vertices the square nodes

on the path in the block tree. Since there is no articulation point separating r_1 and a_1 in $Comp(C)$, there exist simple cycles in $G_a(C)$ that connect x_1 with the vertices of the block a_1. If all these simple cycles contained only one vertex of a_1, this vertex would then be an articulation point separating x_1 from a_1 in $Comp(C)$ leading to a contradiction. This implies that there is a simple cycle in $G_a(C)$ that contains x_1 and at least two vertices of a_1.

If there is only one articulation point separating r_1 and r_2, we use the same argument that there is a simple cycle in $G_a(C)$ that contains x_2 and at least two vertices of a_1. Since a_1 is a block in $G(C)$, there exists a simple cycle in $G_a(C)$ that contains x_1 and x_2. Otherwise we use induction on the number of articulation points separating r_1 and r_2 and we assume inductively that there exists a simple cycle between at least two vertices of a_2 and x_2. Together with the cycle between x_1 and two vertices of a_2 this shows that there is a simple cycle in $G_a(C)$ between x_1 and x_2 and thus that x_1 and x_2 are biconnected in $G_a(C)$. ∎

To record within each edge of $Comp(C)$ the block of $G_a(C)$ to which it belongs, we mark all edges of a block of $Comp(C)$ as being in the same block of $G_a(C)$. Additionally we mark all edges incident to a node that represents a block of $G(C)$ as belonging to the same block of $G_a(C)$. Now we can answer biconnectivity queries in $G(C)$ by using the following characterization.

Lemma 2.3.4 *Two vertices u and v are biconnected in $G_a(C)$ if and only if*

(i) there exists a block (consisting of more than 1 edge) in $G(C)$ that contains u and v, or

(ii) u and v do not have a common block in $G(C)$, but they are represented by the same node r in $Comp(C)$ and the edges (r_1, r) and (r, r_2) joining r to its two neighbors r_1 and r_2 are marked as being in the same block of $G_a(C)$, or

(iii) the representatives of u and v in $Comp(C)$ are not identical, but their representatives have a common block in $G_a(C)$.

Proof: Let $b(u)$ be the block in $G(C)$ that contains the edge on $\pi(u,v)$ incident to u and let $b(v)$ be the block that contains the edge on $\pi(u,v)$ incident to v. There exists a block in $G(C)$ that contains u and v if and only if $b(u) = b(v)$.

We show first that if (i) or (ii) or (iii) is true, then u and v are biconnected. Obviously if $b(u) = b(v)$ and this block consists of more than one edge, then u and v are biconnected in $G_a(C)$.

If $b(u) \neq b(v)$, the representatives of u and of v can only be identical, if both are represented by a node r that substitutes a chain of degree-2 nodes of the block tree that contains $b(u)$ and $b(v)$. Thus $b(u)$ and $b(v)$ both have degree 2 and are connected in the block tree by a chain of degree-2 nodes. Therefore, u and v are biconnected in $G_a(C)$ if and only if r_1 and r_2 are marked as belonging to the same block of $G_a(C)$.

If the representatives of u and v are not identical, then Lemma 2.3.4 shows that u and v are biconnected if and only if their representatives are marked as belonging to the same block of $G_a(C)$.

If none of (i) and (ii) and (iii) are true, then there is no block in $G(C)$ that contains u and v and either their representatives are identical, but then the neighbors of the representative have not the same biconnectivity properties regarding $G_a(C)$, or their representatives are not identical and do not belong to the same block of $G_a(C)$. We have shown above that in both cases u and v are not biconnected in $G_a(C)$. ∎

The following lemma shows that biconnectivity queries can be answered in constant time using $Comp(C)$ and $G(C)$.

Lemma 2.3.5 *We can test in constant time whether two vertices u and v of $G_b(C)$ are biconnected in $G_b(C)$ using $G(C)$ and $Comp(C)$.*

Proof: We show first that we can test in constant time whether two vertices u and v of C_T are biconnected in $G_a(C)$. Since we store at each vertex in $G(C)$ a pointer to its representative in $Comp(C)$, the representatives of u and v can be found and tested for equality in constant time. By Lemma 2.3.2 to test for

biconnectivity in $G_a(C)$ we additionally have to be able to test whether two vertices have a common block in $G(C)$ and in $G_a(C)$.

To test in constant time whether there is a block in $G(C)$ that contains u and v, we root the spanning tree of $G(C)$ at a fixed tree neighbor C_1 of $G(C)$. Whenever we rebuild $G(C)$, we store at each vertex u of $G(C)$ the closest ancestor of u that is an articulation point separating u from C_1. These articulation points can be found for all vertices of $G(C)$ during one depth–first traversal of the spanning tree of $G(C)$ starting at C_1. During this traversal we also label the vertices with preorder and postorder numbers. To test whether u and v have a common block in $G(C)$ we first determine with the help of the labels whether u is an ancestor of v or vice versa. If no, then u and v have a common block if and only if the edge from u to its parent and the edge from v to its parent are in the same block. Otherwise u and v have a common block if and only if the same articulation point is stored at both of them. All these tests can be answered in constant time. We proceed in the same way when we test whether two nodes of $Comp(C)$ have a common block in $G_a(C)$. Thus we can execute all tests of Lemma 2.3.4 and thus test for biconnectivity in $G_a(C)$ in constant time. Using these articulation points we can also find an articulation point that separates u and v in constant time.

Now we discuss how we can test in constant time in $G_b(C)$. If C has only one tree neighbor in H or two tree neighbors that are not biconnected, $G_b(C)$ is identical to $G_a(C)$. If, however, C_1 and C_2 are two tree neighbors of C that are biconnected, the edge (C_1, C_2) has to be added to $G_a(C)$ to create $G_b(C)$. We solve this problem by marking all the nodes on $\pi_{Comp(C)}(C_1, C_2)$. Adding (C_1, C_2) to $G_a(C)$ joins all the marked nodes into the same block of $Comp(C)$. Thus to test whether u and v are biconnected in $G_b(C)$, we first test whether they are biconnected in $G_a(C)$. If they are, we know they are biconnected in $G_b(C)$. Otherwise we additionally check whether the representatives of the two query vertices are marked. If so, u and v are biconnected in $G_b(C)$; otherwise, they are not biconnected. Since all these tests take constant time, we can test for biconnectivity in $G_b(C)$ in constant time. ∎

Since $Comp(C)$ has size $O(d)$, the artificial edges of C can be changed, the biconnected components of C can be computed, and the edges can be marked with the block of $G_a(C)$ they belong to in time $O(d)$. Note that $G(C)$ does not contain any artificial edges and thus does not have to be rebuilt whenever the artificial edges of C change. This shows that the data structure that consists of $G(C)$, $Comp(C)$, and all the information stored at the vertices and edges of these two graphs can be updated in time $O(d)$ when the artificial edges of C change. It answers biconnectivity queries in constant time. If an edge incident to a vertex of C is inserted or deleted, this data structure can be built in time $O(k)$. This shows the following lemma.

Lemma 2.3.6 *An internal data structure for a cluster C can be built in time $O(k)$. Whenever the artificial edges of C change, it can be recomputed in time $O(d)$ with d the high-level degree of C. Biconnectivity queries in $G_b(C)$ can be answered in constant time using the internal data structure.*

2.3.2 Updating the Internal Data Structure

Now we discuss how the internal data structure is maintained during a sequence of insertions and deletions. In this section rebuilding the internal data structure of a cluster C, i.e. $G(C)$ and $Comp(C)$, is referred to as simply *rebuilding* C.

To insert $e = (u, v)$, we add e to $C(u)$ and $C(v)$ and rebuild the clusters. We also recompute the blocks of H. This takes time proportional to the number of edges in H, which is $O((m/k)^2)$ in a general graph and $O(m/k)$ in a planar graph. Thus, the total time for an insertion is $O(k + (m/k)^2)$ in a general graph and $O(k + m/k)$ in a planar graph.

To delete the edge $e = (u, v)$, we remove e from $C(u)$ and $C(v)$ and rebuild them. If $C(u) = C(v)$ during a deletion, it is possible that the set of vertices of $C(u)$ becomes disconnected and $C(u)$ is split into two clusters, each consisting of a connected component. Then a new internal data structure for each new cluster is created. In this case and if a tree edge of H has been deleted, a suitable non-tree edge becomes a new tree edge and the clusters containing its endpoints have to be rebuilt. Additionally it is possible that the external degree of a cluster becomes 3,

even though it contains more than one vertex. To maintain a restricted partition of order k, C' has to be split into at most four new clusters, one with external degree 3 and up to three with external degree 2. Since the new tree edge is not dashed (it was a non-tree edge before) and since C' shared at most one vertex, at most one of the new clusters with external degree 2 shares two vertices and, in this case, has to be split again into two clusters. Thus C' is split into at most five new clusters, one with external degree 3 and at most four of tree degree 2 and each new cluster shares at most one vertex. This implies that a constant number of new internal data structures have to be created. All this can be accomplished in time $O(k)$. Then we recompute the blocks of H.

If the deleted edge (u, v) is a tree edge, two additional problems have to be considered. The first one occurs if a high-level non-tree edge became a new tree edge. Artificial edges in $G(C)$ of every cluster C on $\pi(C(u), C(v))$ in the new high-level

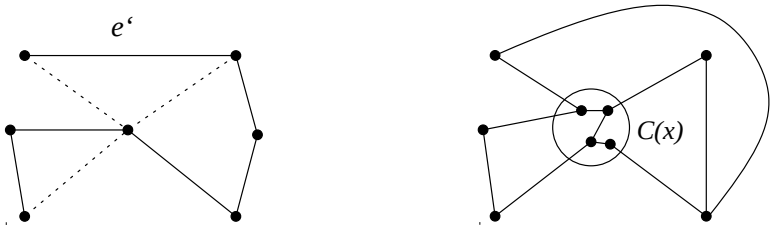

Figure 7: The new high-level graph and the new graph $G_a(C(x))$ if the tree edge e of H is deleted and the non-tree edge e' becomes a new tree edge in the graph of Figure 3.

graph might be invalid, since the tree path of $C(v)$ or $C(u)$ to C might have changed. See Figure 2.3.2. To fix this we recompute the artificial edges of each cluster C on $\pi(C(u), C(v))$ in the new high-level graph and recompute to which blocks of $G_a(C)$ the edges of $Comp(C)$ belong. Determining the new artificial edges for all clusters on $\pi(C(u), C(v))$ can be done with the postorder numbering of one depth–first traversal of the spanning tree of H starting at $C(v)$. The traversal takes time $O(m/k)$. Using the postorder number of the two neighbors of C we can determine which non-tree neighbor is connected to the tree neighbor in constant time. Thus determining all

artificial edges of C takes time proportional to the high-level degree of C. Recomputing the block information in $Comp(C)$ at a cluster C on $\pi(C(u), C(v))$ takes time proportional to the high-level degree of C. Thus the total time is $O(m/k)$ in a planar graph and $O((m/k)^2)$ in a general graph.

The second problem occurs if a cluster C is split into C_1 and C_2. If C_1 and C_2 are biconnected in H, the big vertex representing C in the reduced graph of every cluster B adjacent to C represents now C_1 and C_2. If, however, C_1 and C_2 are not biconnected in H, the big vertex of C has to be split into two big vertices. If C_1 and C_2 are not biconnected when they are created, there is at most one cluster B that is adjacent to both. In this case we rebuild B, which takes time $O(k)$. If C_1 and C_2 are biconnected when they are created, we postpone recomputing B for every neighbor B of C_1 and C_2 until C_1 and C_2 are no longer biconnected. According to Lemma 2.3.1 the internal data structure of B is still valid. If a cluster is split into l, $(l \leq 5)$ clusters, we treat this as $l-1$ splits of a cluster.

To test efficiently whether a cluster has to be rebuilt, we maintain the following data structure. A cluster C that has not been split since the last reconstruction of B is called an *original* cluster for B. The clusters that are created by splitting a cluster C into smaller and smaller clusters are called the *descendants* of C. To test efficiently for each cluster B whether $Comp(B)$ has to be rebuilt each cluster keeps a list for each of its original neighbors C. After a rebuilding of B there is a list for every neighbor C of B containing only C. Whenever a descendant C_1 of C is split into C_2 and C_3, and also C_2 and C_3 are both adjacent to B, then C_1 is replaced by C_2 and C_3 in the list of B for C. If only one of the new clusters, say C_2, is adjacent to B, then C_1 is replaced by C_2. Thus the list at B for the original neighbor C consists of all currently existing descendants of C that are adjacent to B. Whenever any two of them that are in the same list are no longer biconnected, B is recomputed.

As discussed above, we compute the blocks of H after each update operation. Thus we can use them to test at each cluster B whether every pair of clusters stored in the same list at B are biconnected. Every list can be checked in time proportional to its length, since all clusters in a list are adjacent to B. If one of the pairs at B is not biconnected, B is rebuilt, the old set of lists is discarded, and a new set of

lists is stored at B. The lists are updated in the same way, if B is rebuilt for some other reason. Since C is only in the list of B if there is an edge between C and B, the time spent to check all the lists of B is proportional to the high-level degree of B. Thus, all the testing takes time $O(m/k)$ in a planar graph and $O((m/k)^2)$ in a general graph. Since each rebuilding of a cluster takes time $O(k)$, we have to show that only an amortized constant number of clusters is rebuilt per operation.

Lemma 2.3.7 *During l deletions and insertions at most $8l$ clusters are rebuilt.*

Proof: After each deletion or insertion of (u, v), $C(u)$ and $C(v)$ have to be rebuilt. If a non-tree edge is added to the spanning tree, the clusters containing the endpoints of the new tree edge have to be rebuilt as well.

To determine the number of clusters B that are rebuilt because two neighbors of B are no longer biconnected, we use a bipartite graph $\tilde{G}$ connecting a set of blue nodes and a set of red nodes. The graph $\tilde{G}$ contains a blue node for every existing cluster that was created by a split after the last reconstruction of the complete data structure. For each cluster C, $\tilde{G}$ contains a red node for each list at C that consists of at least two clusters. For a red node r representing a list L, there are edges between r and all the blue nodes representing the clusters in L. (Note that each cluster in a list was created by a split after the last reconstruction of the complete data structure and thus has a blue node.) Whenever a cluster is split or rebuilt, $\tilde{G}$ is adapted accordingly.

There is an edge between a cluster and each of the clusters in its lists. Thus every edge in $\tilde{G}$ corresponds to an edge in H, and if two nodes are connected in $\tilde{G}$, they are connected in H (but the converse is not always true).

Whenever a test determines that a cluster B has to be rebuilt because its two neighbors C_1 and C_2 in list L are no longer biconnected in the high-level graph, B must be an articulation point in the high-level graph, and hence the red vertex of L must be an articulation point in $\tilde{G}$, separating the two blue vertices C_1 and C_2. Rebuilding B removes all red vertices of the lists of B and thus disconnects C_1 and C_2 in $\tilde{G}$. Hence the number of connected components

of $\tilde{G}$ increases by at least one. We assume first that there are no insertions of edges in G'. Then two connected components in $\tilde{G}$ that are created during a split cannot become connected again. For any $l' \leq l$, there are at most $4l'$ splits of clusters during l' deletions and therefore there are at most $4l'$ blue vertices in $\tilde{G}$. Every red vertex is always connected to a blue vertex. Thus there are at most $4l'$ connected components in $\tilde{G}$. Each rebuilding of an articulation point in G' increases the number of connected components by one. Thus there are at most $4l'$ rebuildings because two vertices are no longer biconnected.

Now consider also insertions of edges. Because of insertions two connected components of $\tilde{G}$ that were disconnected by a split can become connected again during a later split. Note, however, that every insertion can connect at most two connected components of $\tilde{G}$. Thus it can decrease the number of connected components by at most 1. There are $l - l'$ insertions and thus at most $4l' + (l - l') \leq 4l$ rebuildings caused by the test for rebuildings. Hence the total number of rebuildings during l insertions and deletions is at most $8l$.

∎

It follows from the lemma that the amortized cost of updating the internal data structure after an insertion or deletion is $O(k)$. Rebuilding the reduced graph and the block tree of one cluster takes time $O(k)$. Since there are $O(m/k)$ clusters, rebuilding all internal data structures takes time $O(m)$.

Lemma 2.3.8 *The amortized cost for updating all internal data structures after an insertion or a deletion is $O(k + m/k)$ in a planar graph and $O(k + (m/k)^2)$ in a general graph. Rebuilding all internal data structures takes time $O(m)$.*

2.4 The Shared Data Structure

For each shared vertex s of G we keep a data structure called the *shared data structure* of s. We need this data structure to contract inter-cluster dashed edges of every shared vertex s that lies on $\pi_{G'}(u, v)$ during a $query(u, v)$ operation. This allows us to test in constant time whether s is an articulation point separating u and v.

The three parts of the connectivity data structure, the local, the global, and the look-up data structure, are described in the following sections. In a plane graph the local data structure of a shared vertex s is used to test whether s is an articulation point separating any two neighbors of s. No global or look-up data structure is needed. In a general graph the local data structure is used for testing whether s separates the vertices of any tree-neighboring clusters of s in a subgraph of G. With the global data structure we test whether s separates the vertices of any tree-neighboring clusters of s in G. Finally, the look-up data structure allows us to test whether any two neighbors of s are separated by s.

Additionally we maintain the multigraph, denoted H_c, that is created by contracting all dashed edges of H. Self–loops are deleted in H_c, but not multiple edges between clusters. We rebuild this graph after every insertion and deletion and compute its blocks. This adds time $O(m/k)$ to every update operation in a plane graph and $O((m/k)^2)$ in a general graph.

2.4.1 The Local Data Structure

Let $C_1, \ldots, C_f$ be the clusters that share s. After each reconstruction of all data structures the *local data structure of* s consists of all vertices of $C_1, \ldots, C_f$ with s removed (called *inside* vertices), and all edges between these vertices. It also contains a vertex for every vertex y of G' that is adjacent to a vertex x in C_j (except s) for some j. This vertex is called an *outside* vertex and is connected to x by an edge. Every neighboring cluster C of a cluster C_j is represented by the outside vertices that it contains. These vertices are connected in the following way. If there are only two outside vertices, we connect them by one edge. Otherwise, we fix some arbitrary order of the edges at each vertex of C in a general graph and we take an order given by the embedding in a plane graph. Now an Eulerian tour of the (bidirected) spanning tree of C determines an order on the edges from C to vertices of $C_1, \ldots, C_f$. We connect the outside vertices representing these edges by a cycle in this order (see Figure 8). In what follows an *Eulerian tour of* C refers to the same Eulerian tour of the spanning tree of C.

In a plane graph, if $A_1, \ldots, A_g$ are all the neighboring clusters of s that belong to

a certain block of H_c, then we connect them (resp. their cycles) by $g-1$ *artificial* edges in a way that preserves the planarity of the local data structure. For general graphs we cannot afford this.

In a general graph the local data structure of s can also contain as inside vertices the vertices of clusters not sharing s (because of splitting of clusters), but we maintain the following invariant: *Each vertex of G is contained as an inside vertex in at most two local data structures.*

In a plane graph the local data structure of s can contain as inside vertices only vertices of clusters sharing s. If in a plane graph the edges (x,s) and (s,y) incident to a shared vertex s lie on $\pi_{G'}(u,v)$ during a $query(u,v)$ operation, we use the local data structure of s to test whether s is an articulation point separating u and v as shown in the following lemma.

Lemma 2.4.1 *Let G be a plane graph, let x and y be two vertices of the local data structure of a shared vertex s. Then x and y are connected in the local data structure of s if and only if s is not an articulation point separating x from y in G.*

Proof: If two vertices are connected in the local data structure of s, then the vertices in G that they represent are connected in G. Thus, if x and y are connected in the local data structure of s, there exists a path between x and y in G that does not use s, and hence s cannot be an articulation point separating x and y.

Conversely, if s is not an articulation point separating x and y in G, there exists a path P in G without s that connects x and y. Let $e_1,\ldots,e_l$ with

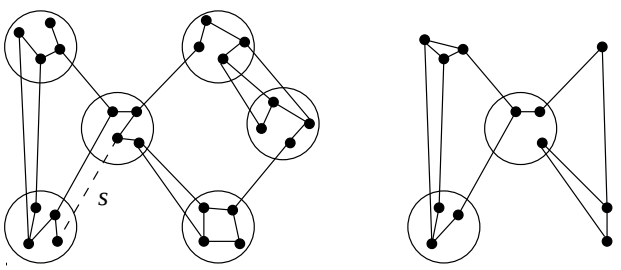

Figure 8: A graph G with a shared vertex s and the local data structure of s.

$e_i = (x_i, y_i)$ be the edges of G at which this path leaves or enters a cluster sharing s. If x is an inside vertex of the local data structure of s, let x_i denote the inside vertices of the local data structure; otherwise, let y_i denote the inside vertices. Then x_{2i} and x_{2i+1} for $1 \leq i < l/2$ and y_{2i-1} and y_{2i} for $1 \leq i \leq l/2$ are connected by a path in G that does not use s, and thus the vertices are also connected in the local data structure of s. This implies that x and y are connected in the local data structure of s. ∎

In a general graph, however, only a weaker property holds, since no artificial edges are added to the graph.

Lemma 2.4.2 *Let G be a general graph. If two neighboring clusters C and C' of s are connected in the local data structure of s, then s is not an articulation point separating the vertices of C from the vertices of C'.*

Proof: If two vertices are connected in the local data structure of s, then they are connected in G. Thus if (one of the vertices of) C is connected to (one of the vertices of) C' in the local data structure of s, there exists a path between C and C' in G that does not use s and thus s is not an articulation point separating the vertices of C from the vertices of C'. ∎

The converse of Lemma 2.4.2 is not true, however.

If G is plane, the data structure of Eppstein et al. [6] for fully-dynamic connectivity testing in plane graphs is applied to every graph of a local data structure. It allows us to maintain a plane graph under a sequence of insertions and deletions of edges while at the same time answering queries of the form: "Are u and v connected?" in time $O(\log n)$. In the case of a general graph, the data structure of Frederickson [8] for fully-dynamic connectivity testing in general graphs is applied to every graph in a local data structure. Frederickson maintains a graph under a sequence of insertions and deletions of edges and connectivity queries in time $O(\sqrt{m'})$ per update, where m' is the number of edges in the graph and in time $O(1)$ per query.

Since each vertex in $C_1, \ldots, C_f$ has degree at most 3, there are $O(fk)$ vertices and edges in the local data structure of s. In a planar and in a general graph these

data structures can be built in time $O(fk)$ [6, 8]. Each vertex is an inside vertex in a constant number of them, so the total size of the $O(m/k)$ local structures and the time to rebuild them is $O(m)$.

We now discuss how the local data structures are updated during insertions and deletions of an edge. A tree edge e is *encountered* by an Eulerian tour of C, whenever it is traversed by the tour. The *side* of a vertex in C is defined to be the set of edges embedded between the tree edges of C in the same order. Thus each vertex of tree degree i has i sides. If an Eulerian tour of C is seen as the traversal of the bidirected spanning tree of C, each side of an vertex appears exactly once in this tour. A non-tree edge with at least one endpoint in C is *encountered* by an Eulerian tour of C, whenever the tour visits the side of an endpoint of the edge to which the edge belongs.

Let s be a shared vertex, let C be a neighboring cluster of s, and let e be an edge with at least one endpoint in C. Then a vertex $z \in C$ is called a *predecessor of e relative to s and C* if z is an outside vertex in the local data structure of s, and z is the last such vertex visited by the Eulerian tour of the spanning tree of C before the edge e is encountered. By symmetry let a *successor of e relative to s and C* be the first such vertex visited after e is encountered. There are at most two different predecessors and at most two different successors relative to s and C for every edge e of C, since e is encountered at most twice in an Eulerian tour.

We first describe update operations in a plane graph and afterwards in a general graph. In a plane graph each vertex is an internal vertex in at most one local data structure. The local data structures contain artificial edges between neighboring clusters of the shared vertices. Before an edge is added to or deleted from a local data structure, we remove all artificial edges from all local data structures. Then we make the necessary changes in all local data structures. Afterwards we compute the new artificial data structures and add them to the local data structures. Since the number of artificial edges in the local data structure of a shared vertex s is proportional to the number of neighboring clusters of s, the total number of artificial edges is proportional to the number of edge bundles in H and hence $O(m/k)$. Thus they can be added or removed in time $O((m/k)\log n)$.

The new artificial edges in the local data structure of s can be computed in time $O(m/k)$ in the following way: Assume all clusters sharing s are collapsed into one cluster C. Then all edge bundles from C to a neighboring cluster of s are embedded around C in a specific order. To determine the additional edges of the local data structure of s we traverse the edge bundles in this order. For each block of H_c we create the list of edge bundles that are in this block and incident to C in the order of the traversal. Note that all the edges incident to a neighboring cluster of s belong to the same block of H_c. Consider the plane multigraph which is created by the edge bundles in each list, C and the neighboring clusters of s that are endpoints of the edges in the ordered list (see Figure 9).

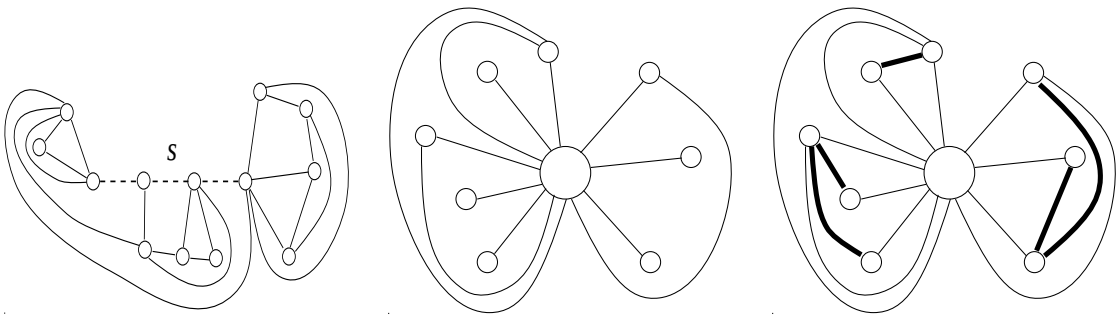

Figure 9: The high-level graph H, a plane multigraph containing all neighboring clusters of s and the multigraph with the artificial edges in bold.

Each face in the multigraph corresponds to a face in the local data structure of s. Thus it is enough to determine for each neighboring cluster of s how its artificial edge(s) have to be embedded in this multigraph. To do this we traverse each face of the subgraph and connect all neighboring clusters on the face with artificial edges to the same cluster B on the face. If there exist clusters that already have artificial edges, we choose one of them and call it B; otherwise, we choose any neighboring cluster of s on the same face as B. In this way all g neighboring clusters of s in the same block of H_c are connected with each other by $g-1$ edges in a way that preserves the planar embedding of the local data structure of s. The time for computing all edges of the local data structure of s is proportional to the high-level degree of all

clusters sharing s. Thus the total time for computing all artificial edges is $O(m/k)$.

If an edge $e = (u, v)$ is inserted or deleted, we have to consider three cases: Either $C(u) = C(v)$, or $C(u) \neq C(v)$ and the clusters $C(u)$ and $C(v)$ share a common vertex s, or $C(u) \neq C(v)$ and $C(u)$ shares a vertex s that is not shared by $C(v)$ (and vice versa).

Case 1: $C(u) = C(v)$

In the first case e is added to or removed from the local data structure of the shared vertex s of $C(u)$. Thus the time for an insertion is $O(\log n)$.

The only special case occurs if e is a tree edge and it is deleted. This splits the spanning tree of $C(u)$ into two parts. If there exists an internal edge e' of $C(u)$ that connects the two parts of the spanning tree, then e' becomes a new tree edge. This does not require any further operations in the local data structure of s, but in the local data structures of every shared vertex s' of which $C(u)$ is a neighboring cluster. The cycle of outside vertices representing $C(u)$ in the local data structure of s' has to be updated.

We find the predecessors z_1 and z_3 and the successors z_2 and z_4 of e relative to s' and $C(u)$ and the predecessors z_5 and z_7 and the successors z_6 and z_8 of e' relative to s' and $C(u)$. The numbering of the edges is such that z_{2i-1} and z_{2i} are identical or connected by an edge in the local data structure of s' for $1 \leq i \leq 4$. See Figure 10. Note that the tree edges in Figure 10 do not belong to the local data

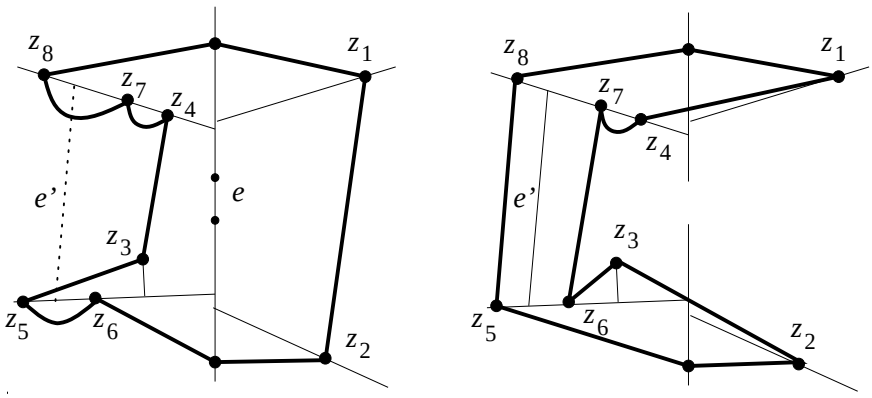

Figure 10: The tree edge e is replaced by the non-tree edge e' in C. Thin edges are tree edges of C, thick edges connect outside vertices representing C.

structure, while the edges between outside vertices do. It is possible that some of these vertices do not exist or are identical. Finding these vertices for all shared vertices s' takes total time $O(k)$. Then we remove these edges $(z_1, z_2), (z_3, z_4), (z_5, z_6)$, and (z_7, z_8) (if they exist) and insert the new edges $(z_1, z_4), (z_2, z_3), (z_5, z_8)$, and (z_6, z_7) into the local data structure of s' (if they are not self loops). This requires a constant number of operations in possibly every local data structure. Thus the total time is $O(k + (m/k) \log n)$ in a plane graph.

If the two parts of the spanning tree of $C(u)$ are not connected by internal edges of $C(u)$, then the deletion of e splits $C(u)$ into two clusters. Only one of these new clusters shares s. Thus we remove the edges (and vertices) of the cluster that does not share s from the local data structure of s. To connect the spanning tree of H, a non-tree edge connecting two clusters becomes a new tree edge. This can increase the external degree of at most two clusters to 3, even though they consist of more than one vertex. Let C' be such a cluster. To maintain a restricted partition of order k, C' has to be split into at most five new clusters, one with external degree 3 and up to four with external degree 2. Each new cluster shares at most one vertex. Thus C' is split into at most five new clusters, one with external degree 3 and at most four of tree degree 2 and each new cluster shares at most one vertex. Since this can happen to at most two old clusters, up to eight splits of clusters are necessary. For each old cluster one new shared vertex can be created for which a new local data structure has to be built. Additionally the vertices and edges of all the new clusters that do not share s have to be removed from the local data structure of s. Since there are at most $O(k)$ edges, this requires $O(k)$ operations and thus $O(k \log n)$ time.

Now the cycles of outside edges in the local data structures have to be updated. If a cluster C is split into C_1 and C_2 and e' is the tree edge that was connecting the tree of C_1 with the tree of C_2, we have to split the cycle of outside vertices representing C into two cycles in possibly all local data structures. Using the Eulerian tour traversal of C, we determine for each shared vertex s' the predecessors z_1 and z_3 and the successors z_2 and z_4 of e' relative to s' and C such that (z_1, z_2) and (z_3, z_4) are edges in the local data structure of s'. The predecessors and successors of e' relative to C and all shared vertices can be found by two traversals of the ordered list of edges

incident to C, which takes time $O(k)$. Note that it is possible that $z_1 = z_4$ or $z_2 = z_3$ or that either z_2 and z_3 or z_1 and z_4 do not exist. If all four vertices exist, we remove the edges (z_1, z_2) and (z_3, z_4) from the local data structure and insert the new edges (z_1, z_4) (if $z_1 \neq z_4$) and (z_2, z_3) (if $z_2 \neq z_3$) into the local data structure (see Figure 11). This requires a constant number of operations in possibly all local

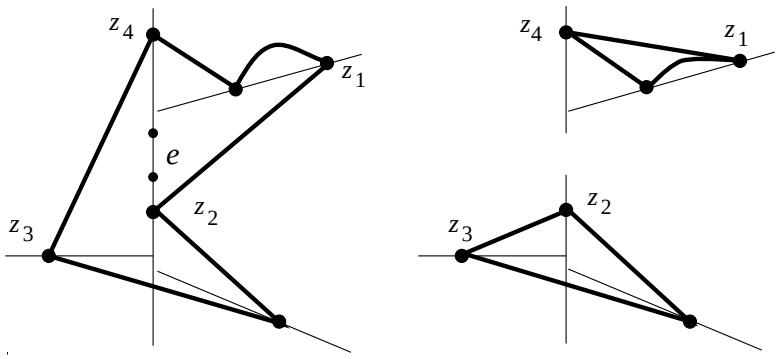

Figure 11: The tree edge e of C is deleted. Thin edges are tree edges of C, thick edges connect outside vertices representing C.

data structures. Since there are at most $O(m/k)$ shared data structures, the cost of updating the cycles of outside vertices in all local data structures after the split of one cluster is $O((m/k)\log n + k)$. Since during the deletion of e at most nine splits of clusters have occurred, all cycles can be updated in time $O((m/k)\log n + k)$. Thus if $C(u) = C(v)$ the total time for updating all local data structures after a deletion is $O((m/k + k)\log n)$.

Case 2: $C(u) \neq C(v)$ and both $C(u)$ and $C(v)$ share the same vertex s.

The edge e is added to or removed from the local data structure of s. Since the clusters of both its endpoints share the same vertex, e connects two inside vertices of the local data structure of s and thus e is not a tree edge. Hence its insertion or deletion does not influence any other local data structures and the total running time in this case is $O(\log n)$.

Case 3: $C(u) \neq C(v)$ and one of the clusters, say $C(v)$, shares a vertex s that is not shared by the other cluster.

If we inserted e, we add e to $C(u)$ and find a predecessor z_1 and a successor z_2 of e relative to s and $C(u)$ such that z_1 and z_2 are connected by an edge in the local data

structure of s. See Figure 12. Note that there exists at most one predecessor and at

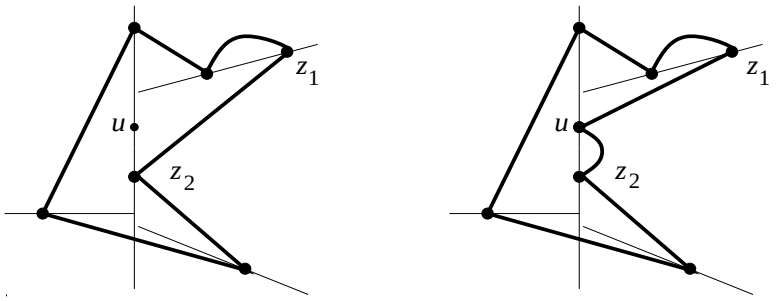

Figure 12: The edge (u, v) is inserted. Thin edges are tree edges of $C(u)$, thick edges connect outside vertices representing $C(u)$.

most one successor of e in this case. Finding these edges by an Eulerian tour traversal of $C(u)$ takes time $O(k)$. If $z_1 = u$, we simply add the new edge to z_1. Otherwise, we create a new outside vertex for u, remove (z_1, z_2) and add e, (z_1, u), and (u, z_2) to the local data structure of s. Thus the time for an insertion in this case is $O(k + \log n)$.

If e is deleted, we check whether there exists another edge from u to a cluster sharing s. If so, we only remove e from the local data structure of s. Otherwise, we also remove u and update the cycle representing $C(u)$. If the deleted edge e was a tree edge, the spanning tree of H is disconnected and a non-tree edge has to become a new tree edge. This might require the splitting of clusters and updating the cycles of outside vertices as described above. Thus the total running time of a deletion in this case is $O((m/k + k) \log n)$.

In a general graph there are no artificial edges in the local data structures and we do not have to maintain an embedding, but the definition of a local data structure has been relaxed as follows. After a reconstruction of all data structures each vertex is an inside vertex in only one local data structure. After some splits of clusters a vertex u of G' can become an inside vertex in up to two local data structures, namely, in the local data structure in which u was an inside vertex after the last reconstruction of the complete data structure and in the local data structure of the vertex that $C(u)$ currently shares. The reason is that if a cluster with a shared vertex s is split into two clusters, only one of them shares s. Thus we have to remove every vertex u of the other cluster from the local data structure of s, but we can only afford this if

the local data structure has size $O(k)$. Otherwise, we do not change the local data structure of s. Thus if the second cluster shares a newly created shared vertex, then u is an inside vertex of two local data structures and it is not an outside vertex in the local data structure of s. The size of the local data structure of every shared vertex that is created through splits of clusters (i.e., not created during the reconstruction of all data structures) is at most $9k + 2$, as we show below. Thus, except for the first local data structure that contains a vertex u as an inside vertex, we can always afford to remove u from the old local data structure. It follows that every vertex u is contained in at most two local data structures. This process guarantees also that two vertices that are contained in clusters sharing the same vertex were inside vertices in the same local data structure after the last reconstruction of the complete data structure. Thus these vertices are inside vertices in the same (up to two) local data structures.

We still have to show that each local data structure of a shared vertex created during a split has size at most $9k + 2$. There are at most k inside vertices (all the vertices of the cluster that was split); each of them is incident to at most three edges. Thus there are $3k$ edges incident to inside vertices and at most $2k+1$ outside vertices. Each of the outside vertices is adjacent to at most two outside vertices. Thus at its creation the local data structure contains at most $7k + 2$ edges and at most $3k + 1$ vertices. During the next at most k insertions (until the next reconstruction of the whole data structure) each insertion causes the creation of at most two edges and of at most one vertex. Thus the local data structure of a shared vertex that is created by a split contains at most $9k + 2$ edges and $4k + 1$ vertices.

As before we consider three cases when an edge $e = (u, v)$ is inserted or deleted. Either $C(u) \neq C(v)$ and $C(u)$ and $C(v)$ share the same vertex, or $C(u) \neq C(v)$ and one of the clusters, say $C(v)$, shares a vertex that is not shared by $C(u)$, or $C(u) = C(v)$.

Case 1: $C(u) \neq C(v)$ and $C(u)$ and $C(v)$ share the same vertex.
As discussed above u and v are inside vertices in the same local data structures and there are at most two such data structures. Thus e is added to or removed from these local data structures. Since e connects the inside vertices of two different clusters,

e is not a tree edge. Therefore its insertion or deletion does not influence any other local data structures. The running time in this case is $O(\sqrt{m})$.

Case 2: $C(u) \neq C(v)$ and one of the clusters, say $C(v)$, shares a vertex s that is not shared by $C(u)$.
As in the plane case we add or remove an outside vertex from the cycle representing $C(u)$ in the local data structure of s (if necessary) and add or remove e. If after the last complete reconstruction u and v were inside vertices in the same local data structure, we add or remove e from this data structure. If they were in different local data structures, we update the one that contains v as an inside vertex by adding or removing an outside vertex to the cycle representing $C(u)$ (if necessary) and by adding or removing e. Then we update the one that contains u and the local data structure of the shared vertex of $C(u)$ in the same way. Thus it takes time $O(\sqrt{m})$ to update the local data structures after an insertion.

If the deleted edge e is a tree edge, the spanning tree of H has to be connected again by making a non-tree edge a tree edge. Thus a cluster C with more than one vertex can become incident to three tree edges. As in the plane case this cluster has to be split into at most four clusters with external degree 2 and one cluster of external degree 3. If C shared a vertex s' and the local data structure of s' has no more than $9k + 2$ edges and vertices, it is completely rebuilt. Otherwise, it is not changed, but vertices whose clusters do not share s' are still inside vertices of it. We have to store this information at each such vertex. If a new shared vertex was created, we build a local data structure for it. Rebuilding a local data structure of size $O(k)$ takes time $O(k)$.

Finally we have to update all cycles of outside vertices representing C in the other local data structures. Notice that (except in the two already updated local data structures) if one vertex of C is an inside vertex of a local data structure, then all vertices of C are inside vertices and C is not represented by a cycle of outside vertices in this local data structure. These local data structures do not have to be changed. If C is represented in a local data structure by a cycle of outside vertices, this cycle has to be split into the appropriate subcycles. As described in the plane case this requires a constant number of operations in possibly all other local data structures.

Each such operation in a data structure of size $O(m')$ takes time $O(\sqrt{m'})$. There are m/k such data structures and their total size is $O(m)$. Thus the total time of all these update operations is $O(m/\sqrt{k})$ and the total running time of an update in this case is $O(m/\sqrt{k}+k)$.

Case 3: $C(u) = C(v)$. As mentioned earlier, u and v are inside vertices in the same local data structures in this case. Thus e is added to or removed from the at most two such local data structures. The time for updating the local data structures after an insertion is $O(\sqrt{m})$.

If the deleted edge e is a tree edge, the spanning tree of $C(u)$ is split into two parts and two subcases have to be considered. If $C(u)$ is still connected, i.e., there exists an internal non-tree edge e' that connects the two disconnected pieces of the spanning tree of $C(u)$, then e' becomes a tree edge. All local data structures in which $C(u)$ is represented by a cycle of outside edges have to be updated as in the plane case. This requires a constant number of operations in possibly all local data structures and thus takes time $O(m/\sqrt{k})$. The second subcase is that $C(u)$ is no longer connected. In this case $C(u)$ is split into two clusters, one of which contains the shared vertex s of $C(u)$. Thus if the size of the local data structure of s is at most $9k+2$, it is completely rebuilt. Otherwise, the vertices of the new cluster not sharing s stay in the local data structure. Then a non-tree edge is found that connects the two disconnected subtrees of H. It becomes a tree edge, which might require additional splits of clusters. As described above all local data structures can be updated in time $O(m/\sqrt{k}+k)$.

This establishes the following lemma.

Lemma 2.4.3 *The total time spent for updating all local data structures in a plane graph is $O(k+(m/k)\log n)$ after an insertion and $O((k+m/k)\log n)$ after a deletion. In a general graph it is $O(\sqrt{m})$ after an insertion and $O(k+m/\sqrt{k})$ after a deletion. Rebuilding all local data structures takes time $O(m)$.*

In a plane graph we maintain an embedding and we add artificial edges to the local data structure. If we relax the data structure in the plane case in the same way as in the general case, the time for a deletion can be speeded up to $O(k+(m/k)\log n)$. With an appropriate choice for k this leads to a time of $O(\sqrt{n\log n})$ per update

operation in the plane case. For simplicity of the description we did not present this approach here.

2.4.2 The Global Data Structure

The local data structures for general graphs only maintain partial connectivity information. They do not consider the fact that two clusters C and C' that are both tree-neighboring clusters of s can also be connected in G through a path that is not contained in the local data structure of s. In the latter case C and C' are biconnected in H_c. Thus we build a *global data structure of* s, which is a graph whose vertices are the tree-neighboring clusters of s. Two clusters C and C' are connected by an edge, if either the vertices of C and the vertices of C' are connected in the local data structure of s or C and C' are biconnected in H_c. We say that a vertex s of G is an articulation point *separating the vertices of* C *from the vertices of* C' if every path from a vertex of C to a vertex of C' contains s.

Lemma 2.4.4 *Let C and C' be tree-neighboring clusters of s. Then C and C' are connected in the global data structure of s if and only if s is not an articulation point separating the vertices of C from the vertices of C' in G.*

Proof: If C and C' are connected in the global data structure of s, then there is a path in the global data structure from C to C'. Each edge (A, B) on this path represents a path from cluster A to cluster B either in H_c and not using a cluster sharing s or a path in the local data structure of s. In the first case there is a path in G that uses only vertices of clusters not sharing s. In the second case Lemma 2.4.2 shows the existence of a path without s from A to B in G. Thus in both cases there exists a path from the vertices of C to the vertices of C' in G that does not use s and therefore s is not an articulation point separating the vertices of C from the vertices of C'.

If s is not an articulation point separating C from C', there exists a path P in G from C to C' not containing s. Consider all neighboring clusters of s. Let $A_1, A_2, \ldots, A_l$ be a sequence of these clusters in the order in which they

appear on P (consequent copies of the same cluster are deleted). Then A_i and A_{i+1} are connected by a subpath of P in G that does not contain s. If this subpath does not contain a cluster sharing s, then A_i and A_{i+1} are biconnected in H_c. Otherwise, A_i and A_{i+1} are connected in the local data structure of s. Thus A_i and A_{i+1} are connected by an edge in the global data structure and $(A_1, A_2), (A_2, A_3), \ldots, (A_{l-1}, A_l)$ is a path in the global data structure of s connecting C and C'. ∎

The global data structures are completely rebuilt after each insertion or deletion of an edge. Let t be the number of vertices in the global data structure of s. To build this data structure we test every pair of vertices in the global data structure to see whether they are either connected in the local data structure of s or biconnected in H_c. Since every such test can be executed in constant time, it takes time $O(t^2)$ to construct the global data structure of s. Then we compute the connected components of the global data structure, which can also be accomplished in time $O(t^2)$. Since each cluster has external degree 3, it is contained in at most three global data structures. Thus the total number of vertices in all global data structures is $O(m/k)$ and all of them can be rebuilt in time $O((m/k)^2)$. Since we computed the connected components of the global data structure of s, we can test whether two tree-neighboring clusters C and C' are connected in constant time.

Lemma 2.4.5 *The time for rebuilding all global data structures is $O((m/k)^2)$. All global data structures can be updated in time $O((m/k)^2)$ after an insertion or deletion and a connectivity query in a global data structure can be answered in constant time.*

2.4.3 The Look-up Data Structure

Let C and C' be two tree-neighboring clusters of a shared vertex s. Using the global data structures we can test in constant time whether s is an articulation point separating the vertices of C from the vertices of C', but we still cannot test whether s also separates two vertices that are contained in clusters sharing s. Thus we build a third data structure, called the *look-up data structure of s*. Let $C_1, \ldots, C_f$ be the clusters

that share s. After the reconstruction of all data structures the look-up data structure of s is a graph whose vertices are all the vertices of $C_1, \ldots, C_f$ with s removed. Its edges are the edges between vertices of $C_1, \ldots, C_f$ in G'. As in a local data structure after splits of clusters a look-up data structure can contain some additional vertices, but every vertex u is contained in at most two look-up data structures, the one to which it belonged after the last reconstruction of all data structures and the look-up data structure of the shared vertex that is contained in $C(u)$. The inside vertices of the local data structure of a shared vertex s and the vertices in the look-up data structure of s are always the same.

Whenever the whole set of data structures is rebuilt, we compute the connected components of the look-up data structures. Then we form *groups* of vertices in the following way. Each connected component that consists of at most k vertices is a group. If one of the connected components contains more than k vertices, we extend T to a spanning tree T_s of the connected component and determine a restricted partition of order k with respect to T_s (as defined in Section 2.2). Each cluster of the partition forms a group. The vertices of every group are connected and every group contains at most k vertices. Additionally we keep a *graph of groups* that consists of a vertex for every group and an edge between two groups K_1 and K_2 labeled l if there are l edges between K_1 and K_2. Note that every connected component of the graph of groups consists of $O(m/k)$ groups.

We store this information in the following way. At each vertex we maintain a *group pointer* to its group, at each group K we store a *neighbor pointer* to a neighboring cluster of s that is connected to the vertices of K in the local data structure of s if such a cluster exists. Otherwise, the neighbor pointer is set to nil. We also store at each non-tree neighboring cluster C of s a *tree neighbor pointer* to a tree-neighboring cluster of s that is biconnected to C in H_c. Note that each non-tree-neighboring cluster is biconnected to a tree-neighboring cluster in H_c, since all clusters are connected by a tree. Using this information we can find in constant time a vertex v in the look-up data structure of s a tree neighboring cluster of s that is connected to v in G on a path without s. We call this cluster $L(v)$. If the neighbor pointer of the group of v is nil, then $L(v)$ does not exist. We notice this also in constant time.

Lemma 2.4.6 *Let u and v be two vertices that lie in clusters that share s. The vertex s is an articulation point separating u and v in G if and only if u and v are not connected in the local data structure of s and $L(u)$ and $L(v)$ (if they exist) are not connected in the global data structure of s.*

Proof: Let us first consider the case where $L(u)$ and $L(v)$ exist.

First we want to show if s is an articulation point separating u and v in G, then u and v are not connected in the local data structure of s and $L(u)$ and $L(v)$ are not connected in the global data structure of s.

There is a path from u to $L(u)$ in G that does not pass through s and there is a path from v to $L(v)$ in G that does not use s. Thus if s is an articulation point separating u and v in G, then s is an articulation point separating the vertices of $L(u)$ from the vertices of $L(v)$ in G. According to Lemma 2.4.4, if s is an articulation point separating the vertices of $L(u)$ and the vertices of $L(v)$ in G, then $L(u)$ and $L(v)$ are not connected in the global data structure of s. Since a path from u to $L(u)$ and from v to $L(v)$ is contained in the local data structure of s, it follows that if $L(u)$ and $L(v)$ are not connected in the global data structure of s, then u and v are also not connected in the local data structure of s. Thus we have shown that if s is an articulation point separating u and v in G, then u and v are not connected in the local data structure of s and $L(u)$ and $L(v)$ are not connected in the global data structure of s.

Now we prove the other direction. According to Lemma 2.4.4, if $L(u)$ and $L(v)$ are not connected in the global data structure of s, then s is an articulation point separating the vertices of $L(u)$ from the vertices of $L(v)$ in G. Since there are paths in G from u to $L(u)$ and from v to $L(v)$ that do not use s, it follows that if s is an articulation point separating the vertices of $L(u)$ from the vertices of $L(v)$ in G, then s is an articulation point separating u from v in G. Thus we have shown that if $L(u)$ and $L(v)$ are not connected in the global data structure of s, then s is an articulation point separating u from v in G.

Let us now consider the case when at least one of $L(u)$ and $L(v)$ does not

exist. Without loss of generality we assume that $L(u)$ does not exist, i.e., that u is not connected to a neighboring cluster of s on a path in G that does not contain s. Thus s is not an articulation point separating u and v in G if and only if there is a path from u to v in G that only uses vertices inside clusters sharing s, but not s. But this is true if and only if u and v are connected by a path in the local data structure of s. ∎

Lemma 2.4.7 *Let u be a vertex lying in a cluster that shares s and let C be a tree-neighboring cluster of s. Then s is an articulation point separating u from the vertices of C in G if and only if u and C are not connected in the local data structure of s and $L(u)$ and C are not connected in the global data structure of s.*

Proof: Similar to the proof of Lemma 2.4.6 by symmetry. ∎

To test whether s is an articulation point separating two vertices u and v, we determine $L(u)$ and $L(v)$ in constant time using the look-up data structure and then we test whether $L(u)$ and $L(v)$ are connected in the global data structure of s and whether u and v are connected in the local data structure of s. All this takes constant time.

Whenever the edge e is inserted or deleted, the at most two look-up data structures of the endpoints of e have to be updated. Additionally, if a cluster sharing a vertex s is split during an insertion or deletion, the look-up data structure of s has to be updated. We discuss first how to maintain a look-up data structure if one of its clusters is split. Afterwards we discuss how a look-up data structure is updated that contains the endpoint of an inserted or deleted edge.

If a cluster C is split into C_1 and C_2 and a new shared vertex is created, we build a new look-up data structure for this vertex. Since it consists of at most $9k+2$ edges and at most $4k+1$ vertices, this can be accomplished in time $O(k)$. If one of the new clusters, say C_1, does not contain the shared vertex s of C, we remove the vertices of C_2 from the look-up data structure of s if and only if the size of the local data structure of s is at most $9k+2$ (i.e., if and only if we rebuild the local data structure of s). If its size is larger then $9k+2$, the look-up data structure of s is not changed.

The same argument as in Section 2.4.1 shows that no vertex is contained in more than two look-up data structures. Since each cluster is contained in at most two look-up data structures and since during an insertion or deletion of an edge at most nine splits of clusters occur, the cost of updating all look-up data structures that contain a split cluster is $O(k)$.

Next we discuss how to update a look-up data structure that contains an endpoint of an inserted or deleted edge. Since each vertex is contained in at most two look-up data structures, there are at most four such data structures. As shown in [8] it takes time $O(k)$ to maintain a topological partition of order k, the graph of groups, and the group pointers at the vertices after an insertion or deletion. To update the neighbor pointers at the groups we determine for each endpoint of e the group K to which it belongs. Then we test whether K is connected in the local data structure of s to any neighboring clusters of s and update its group pointer appropriately, which takes time $O(m/k)$. We store this information also at the group pointers of the $O(m/k)$ groups that are in the connected component of K in the graph of groups. Since none of the other groups are connected to K, they cannot be influenced by the changes in K. Additionally the tree neighbor pointers at each non-tree-neighboring cluster of s have to be updated. Removing all clusters sharing s splits the tree of H into subtrees, each containing one tree-neighboring cluster of s. Note that every non-tree-neighboring cluster of s is biconnected in H_c with the tree-neighboring cluster in its subtree. Thus we set the tree neighbor cluster pointers of all clusters in a subtree to the tree-neighboring cluster of this subtree. This takes time $O(m/k)$ per shared vertex. It follows that the time for updating a look-up data structure that contains the endpoint of an inserted or deleted edge is $O(k + (m/k)^2)$.

When we reconstruct all look-up data structures, we first compute the connected components and the topological partitions in each data structure. Then we set the group pointer at each vertex. This takes time $O(m)$ since each vertex is contained in at most one look-up data structure. To set the neighbor pointer in the look-up data structure of a shared vertex s we check all the edges leaving the vertices of each group K to find a neighboring cluster C of s that is adjacent to a vertex in K. If there is such an edge, we know that C is connected with the vertices of K in the

local data structure of s. Thus we set the neighbor pointer of K and of all groups in the connected component of K to C. If, however, neither K nor a group in the connected component of K has an edge to a neighboring cluster of s, none of the vertices in the connected component is connected to a neighboring cluster of s in the local data structure of s. Thus we set the neighbor pointer of all the groups in the connected component to nil. Computing all these pointers takes time proportional to the number of vertices in the look-up data structures that is $O(m)$. As described above we can find the tree neighbor pointers of the non-tree neighboring clusters in all data structures in time $O((m/k)^2)$. Thus the total running time for rebuilding the look-up data structures is $O(m + (m/k)^2)$, which is $O(m)$ for $k \geq m^{1/2}$.

Lemma 2.4.8 *A connectivity query in a look-up data structure can be answered in time $O(1)$ and it takes time $O(k + (m/k)^2)$ to update all look-up data structures after an insertion or deletion. The reconstruction time for all look-up data structures is $O(m)$.*

Lemma 2.4.9 *A connectivity query in a shared data structure can be answered in time $O(\log n)$ in a plane graph and in constant time in a general graph. In a plane graph the time for an insertion is $O(k + (m/k)\log n)$ and for a deletion $O((k + m/k)\log n)$. In a general graph it is $O(k + m/\sqrt{k} + (m/k)^2)$ after both operations. Rebuilding all shared data structures takes time $O(m)$.*

2.5 Updates

If an edge (u, v) is inserted, we add it to H, to H_c and to the appropriate internal and shared data structures. Additionally we rebuild all data structures every $\min\{k, m/k\}$ insertions. Together with the time for updating all internal data structures and all shared data structures (see Section 2.3 and 2.4) the amortized time of an insertion is $O(k + (m/k)\log n)$ in a plane graph and $O(k + (m/k)^2 + m/\sqrt{k})$ in a general graph.

If an edge $e = (u, v)$ is deleted, we test whether u and v lie in the same cluster. If they do and if e is a tree edge, the spanning tree of $C(u)$ is disconnected by the deletion. Thus we look for a non-tree edge that connects the two pieces and add it

to the tree. If we cannot find such an edge in $C(u)$, $C(u)$ is split into two clusters, each corresponding to a connected component. Since $C(u)$ consists of $O(k)$ edges and vertices, this takes time $O(k)$. If e is a tree edge between two clusters, we test all edges in H to find a non-tree edge that connects the two disconnected subtrees of H and add it to the tree.

This can increase the external degree of a cluster C to 3, even though the cluster contains more than one vertex. To maintain a restricted partition of order k, C has to be split into at most four new clusters, one with external degree 3 and up to three with external degree 2. Since the new tree edge is not dashed (it was a non-tree edge before) and since C shared at most one vertex, at most one of the new clusters with external degree 2 shares two vertices. If this is the case, it has to be split again into two clusters. Thus C is split into at most five new clusters (one with external degree 3 and up to four of tree degree 2). Each new cluster shares at most one vertex.

All this takes time $O(k + m/k)$ in a plane graph and $O(k + (m/k)^2)$ in a general graph. Together with the time for updating the internal data structure and the shared data structures (see Section 2.3 and 2.4) and the cost of rebuilding the whole data structure every m/k deletions, the amortized running time of a delete operation is $O((k + m/k)\log n)$ in the plane case and $O(k + m/\sqrt{k} + (m/k)^2)$ in the general case.

To speed up queries we compute after each update operation a bit called *coverage bit* at each cluster C with two tree neighbors C_1 and C_2. It is set to one if C_1 and C_2 are biconnected in $G_b(C)$. Otherwise, it is zero. Since biconnectivity in $G_b(C)$ can be tested in constant time according to Lemma 2.3.5, the coverage bit of C can be computed in constant time.

Lemma 2.5.1 *Let C_1 and C_2 be the tree neighbors of C. Let $b_i = b(C_i, C)$ for $i = 1, 2$ if (C_i, C) is solid and let it be the shared vertex to which (C, C_i) belongs otherwise. The coverage bit of C is one if and only if b_1 and b_2 are biconnected in G.*

Proof: The coverage bit of C is set to one if and only if C_1 and C_2 are biconnected in $G_b(C)$. If (C_1, C) and (C_2, C) are solid, C does not share a vertex and the correctness of the lemma follows immediately from Lemma 2.3.2. Otherwise the shared vertex of C is one of the endpoints of $\pi_G(b_1, b_2)$. Thus

is cannot be an articulation point separating b_1 and b_2. Hence Lemma 2.3.2 shows that b_1 and b_2 are biconnected in G if and only if they are biconnected in $G_b(C)$. ∎

Lemma 2.5.2 *Let $C_0, C_1, \ldots, C_{g+1}$ $(g \geq 1)$ be the clusters on $\pi_H(C_0, C_{g+1})$. If the edges (C_0, C_1) and (C_g, C_{g+1}) are dashed and every tree edge between C_i and C_{i+1} is solid $(1 \leq i \leq g)$, then $b_1 = b(C_1, C_0)$ and $b_2 = b(C_g, C_{g+1})$ are biconnected in G if and only if the coverage bits of all clusters C_i are one.*

Proof: Let (w, x) and (x, y) be edges on $\pi(b_1, b_2)$, and let C_j for a j with $1 \leq j \leq g$ be the cluster containing x. If $2 \leq j \leq g-1$, then x is an articulation point separating b_1 and b_2 in G if and only if every path in G from $b(C_{j-1}, C_j)$ to $b(C_{j+1}, C_j)$ goes through x and thus if and only if $b(C_{j-1}, C_j)$ and $b(C_{j+1}, C_j)$ are not biconnected in G. But according to Lemma 2.5.1 this is exactly the condition for the coverage bit of C_j to be zero. If $j = 1$, then x is an articulation point separating b_1 and b_2 in G if and only if every path in G from b_1 to $b(C_2, C_1)$ goes through x and thus (with Lemma 2.5.1) if and only if the coverage bit of C_1 is 0. The case $j = g$ is handled symmetrically. ∎

Now we maintain a flag after each update operation in a general graph for each pair of clusters C and C'. The *flag* is set to one if and only if the following two conditions are fulfilled:

1. the coverage bits of all clusters on $\pi_H(C, C')$ (excluding C and C') are one, and
2. for each shared vertex s on $\pi_H(C, C')$ the $query(C_1, C_2)$ operation in the shared data structure of s returns yes, where C_1 and $C_2 \neq C_1$ are the tree-neighboring clusters of s on $\pi_H(C, C')$. (If only one tree-neighboring cluster of a shared vertex is on $\pi_H(C, C')$, we do not have to ask a query in the shared data structure.)

Otherwise, the flag is set to zero.

Lemma 2.5.3 *Let $b_1 = b(C, C')$ and $b_2 = b(C', C)$. The flag for C and C' is 1 if and only if b_1 and b_2 are biconnected in G.*

Proof: Let $C_1, \ldots, C_{2g}$ be the clusters on the tree path in G from u to v in the order of their occurrence such that there is a dashed edge between C_{2i-1} and C_{2i} for $1 \leq i \leq g$. Let C_0 be C, let C_{2g+1} be C', let C_{-1} be C_1, and let C_{2g+2} be C_{2g}.

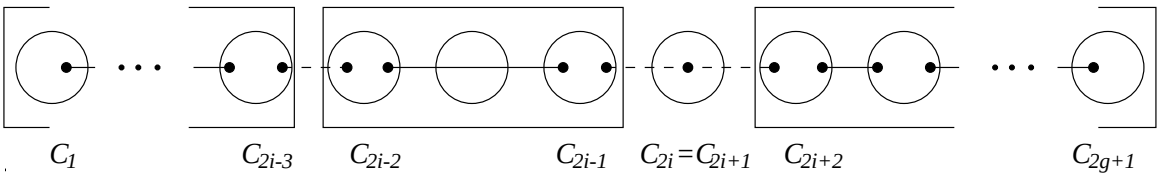

Figure 13: The clusters on $\pi(u, v)$. If all coverage bits are one, the boxes indicate the blocks on $\pi(u, v)$.

Lemma 2.5.2 showed that for $0 \leq i \leq g$ the coverage bit of all clusters on the tree path between C_{2i} and $C_{2i+1} \neq C_{2i}$ are 1 if and only if $b(C_{2i}, C_{2i-1})$ and $b(C_{2i+1}, C_{2i+2})$ are biconnected (or identical) in G. See Figure 13. Since $b(C_{2i-1}, C_{2i})$ and $b(C_{2i}, C_{2i-1})$ (for $0 < i \leq g$) belong to the same shared vertex in G, the only possible articulation points on $\pi_G(b_1, b_2)$ are the shared vertices on $\pi_G(b_1, b_2)$. According to Lemma 2.4.1 and Lemma 2.4.4, none of these shared vertices is an articulation point separating b_1 and b_2 if and only if querying the connectivity of the two tree-neighboring clusters of each shared vertex in its shared data structure returns yes. ∎

If one of the queries in the shared data structure of a shared vertex s returns no, we store s as an articulation point separating C_1 and C_2 along with the flag. If the coverage bit of a cluster C on $\pi_H(C_1, C_2)$ is zero, we store the articulation point separating the tree neighbors of C (which is stored at one of them in $Comp(C)$) as an articulation point. If C_1 and C_2 share a vertex, we record this vertex. If they have no common shared vertex, but the edge incident to C_1 on $\pi_H(C_1, C_2)$ is dashed and belongs to the shared vertex s, we store s and the tree-neighboring cluster of s on $\pi_G(s, b(C_2, C_1))$ at C_1. All information stored at C_1 can be computed during one traversal of the spanning tree of H, which takes time $O(m/k)$. Thus all this information can be computed in time $O((m/k)^2)$.

This method is too expensive in the plane case. Thus we keep two dynamic trees, one for the spanning trees of G and one for the spanning tree of H. A dynamic tree [22] represents a rooted tree by solid and dashed edges. At most two edges incident to a vertex are solid. To avoid confusion with the dashed edges in our data structure we call the dashed edges in dynamic trees *dotted* edges. We use a slight variation of dynamic trees whereby costs are associated with vertices rather than with edges. In a dynamic tree the operations $link(x, y)$, $cut(x)$, $root(x)$ and $mincost(x)$ are allowed. A $link(x, y)$ operation links the tree of x with the tree of y by adding the edge (x, y). A $cut(x)$ operation removes the edge from x to its parent in the tree. A $root(x)$ operation roots the tree at x and a $mincost(x)$ query computes the minimum cost stored at a vertex on the path from x to the root of the tree of x. Each operations takes time $O(\log n)$, where n is the number of nodes in the tree.

Our first dynamic tree contains the spanning tree of H. The cost at every cluster with external degree 2 is its coverage bit. If a cluster C has three tree neighbors C_1, C_2, and C_3 and the tree edges (C_1, C) and (C, C_2) are the current solid edges incident to C, we store the coverage bit for C_1 and C_2 at C. Note that whenever a dotted edge (C_1, C) is converted into a solid edge, we know the other solid edge (C, C_2) that is leaving C and we can change in time $O(1)$ the flag at C to be the coverage bit of C_1 and C_2. Thus the coverage bits can be updated during the operations in the dynamic tree without increasing their running time. After each update operation we update the coverage bits in the dynamic tree in time $O(\log n)$ per bit. If a tree edge of H is deleted and a non-tree edge becomes a tree edge, we execute the appropriate *cut* and *link* operations in the dynamic tree. There are $O(1)$ of these operations, at a cost of $O(\log n)$ each. Thus, the total time for updating the tree after an update operation is $O((m/k) \log n)$.

A similar idea is used to handle dashed edges between clusters. We store the spanning tree of G in the second dynamic tree. We store cost one at each shared vertex s if (a, s) and (s, b) are the current solid edges at s in the dynamic tree and the query(a, b) operation in the shared data structure of s returns yes. Otherwise, we store cost zero at s. As above, these costs can be updated during the operations of the dynamic tree. Computing the bit of one vertex takes time $O(\log n)$. Since $O(\log n)$

costs are computed during an operation in the dynamic tree, the running time of each operation in this dynamic tree is $O(\log^2 n)$. After an insertion or deletion we execute a constant number of *cut* and *link* operations and update all the costs, which takes time $O((m/k)\log n)$.

Lemma 2.5.4 *Let $b_1 = b(C_1, C_2)$ and $b_2 = b(C_2, C_1)$. The vertices b_1 and b_2 are biconnected in G if and only if all costs on $\pi(b_1, b_2)$ (excluding b_1 and b_2) in the second dynamic tree are one and all costs on $\pi(C_1, C_2)$ (excluding C_1 and C_2) in the first dynamic tree are one.*

Proof: The costs on $\pi(b_1, b_2)$ (excluding b_1 and b_2) in the second dynamic tree and the costs on $\pi(C_1, C_2)$ (excluding C_1 and C_2) in the first dynamic tree are exactly the bits that are used in a general graph to compute the flag for C_1 and C_2. Thus Lemma 2.5.3 shows the correctness of this lemma. ∎

Lemma 2.5.5 *The amortized time of an insertion is $O(k + (m/k)\log n)$ and the amortized time of a deletion is $O((k+m/k)\log n)$ in a plane graph and $O(k+m/\sqrt{k}+(m/k)^2)$ in a general graph.*

2.6 Queries

In this section we want to discuss how to answer a $query(u, v)$ operation. As shown in Lemma 2.3.5 the biconnectivity of two vertices u and v in the same cluster can be tested with one query in the internal data structure of $C(u)$ and in the shared data structure of the share vertex of $C(u)$. This takes constant time in a general graph and $O(\log n)$ time in a plane graph.

If u and v are not in the same clusters, we distinguish again between general graphs and plane graphs. To answer a query(u, v) operation in a general graph we check the flag stored for the tuple $(C(u), C(v))$. Let $b(u) = b(C(u), C(v))$ and $b(v) = b(C(v), C(u))$. If $C(u)$ and $C(v)$ share a vertex s, we test in the shared data structure of s whether u and v are connected. If they are not connected, s is an articulation point separating u and v. If they are connected, we know that s is not

an articulation point on $\pi_G(u,v)$ and we have to test the biconnectivity of $b(u)$ and u and of $b(v)$ and v. Vertex u and vertex v are biconnected if and only if both tests return yes. These tests can be executed in constant time, since the two vertices used in the test are in the same cluster.

If $C(u)$ and $C(v)$ do not share a vertex, we check the flag of the tuple $(C(u), C(v))$. If it is zero, u and v are not biconnected. If it is one, there is no articulation point on $\pi(b(u), b(v))$ according to Lemma 2.5.3.

Let C be the tree neighbor of $C(u)$ on $\pi(C(u), C(v))$. We have to test next whether the edge $(C(u), C)$ is dashed. If it belongs to the shared vertex s, we test in the shared data structure of s whether u and the cluster stored at $C(u)$ as tree-neighboring cluster for this dashed edge are connected. Additionally we check the biconnectivity of $b(u)$ and u in $G_b(C(u))$. If the edge $(C(u), C)$ on $\pi(C(u), C(v))$ is solid, we test the biconnectivity of u and C in $G_b(C(u))$. The edge incident to $C(v)$ is treated symmetrically. Vertex u and vertex v are biconnected if and only if all connectivity and biconnectivity tests return yes. It takes constant time to execute all these tests. Thus the running time of a query is $O(1)$.

In a plane graph we ask a *mincost* query in both dynamic trees. Let u_1 (resp. v_1) be the tree neighbor of u (resp. v) on $\pi(u,v)$. First we root the dynamic tree of the spanning tree of G at u_1 and ask a $mincost(v_1)$ query. If the answer is one, no shared vertex is an articulation point separating u and v. Otherwise, we return one of the shared vertices on $\pi(u_1, v_1)$ whose cost is zero as an articulation point separating u and v. These operations take time $O(\log^2 n)$ in the dynamic tree.

Let C_1 (resp. C_2) be the tree neighbor of $C(u)$ (resp. $C(v)$) on $\pi(C(u), C(v))$. We root the dynamic tree of the spanning tree of the high-level graph at C_1. If the $mincost(C_2)$ query returns 1, there is no articulation point separating u and v on $\pi(b(u), b(v))$ and we check the biconnectivity of u and C_1 in $G_b(C(u))$ and of v and C_2 in $G_b(C(v))$. If the $mincost(C_2)$ query returns zero, the algorithm outputs the articulation point stored at the tree neighbors of one of the clusters on $\pi(C_1, C_2)$ with cost zero. Since operations in this dynamic tree take time $O(\log n)$, the total running time of a query in a plane graph is $O(\log^2 n)$.

Theorem 2.6.1 *In a plane graph the worst-case query time is $O(\log^2 n)$, and the*

amortized update time is $O(\sqrt{n}\log n)$. *In a general graph the query time is* $O(1)$, *and the amortized update time is* $O(m^{2/3})$.

Proof: Let m_0 be the number of edges in G during the last reconstruction. Then the current number of edges is $O(m_0)$. In the plane case the running time of a query is $O(\log^2 n)$, the amortized running time of an insertion is $O(k + (m/k)\log n)$, and the amortized running time of a deletion is $O((k + m/k)\log n)$. Setting $k = \lfloor\sqrt{m_0}\rfloor$ gives the desired result. In a general graph the worst-case running time of a query is $O(1)$ and the amortized running time of an insertion or a deletion is $O(k + (m/k)^2 + m/\sqrt{k})$. Setting $k = \lfloor m_0^{2/3}\rfloor$ gives a running time of $O(m^{2/3})$ per update. ∎

Chapter 3

Two-Edge Connectivity in Plane Graphs

3.1 Introduction

An undirected graph $G = (V, E)$ is *two-edge connected* if there are at least two edge-disjoint paths from each vertex to every other vertex. Two vertices are *two-edge connected* if there are at least two edge-disjoint paths between them. A graph G is not two-edge connected if and only if there is a edge whose removal disconnects G. Such an edge is called a *bridge* of G. Each maximal subgraph of G that is two-edge connected is called a *two-edge connected component*.

In this chapter we present a data structure that maintains two-edge connected components in plane graphs under an arbitrary sequence of the following operations:

$Insert(u, v)$: Insert an edge between the two vertices u and v.

$Delete(u, v)$: Delete the edge (u, v).

$Query(u, v)$: Return *true* if vertices u and v are in the same two-edge connected component and *false* otherwise.

Our data structure supports online updates (insertion and deletion of an edge) in $O(\log^2 n)$ time, and answers a query in $O(\log n)$ time, where n is the number of

vertices in the graph. The data structure can be built in linear time and requires linear storage. The machine model is a RAM with wordsize $\Omega(\log n)$.

Our algorithm introduces two new concepts, *edge bundles* and *coverage graph recipes*. The vertices of the graph are partitioned into clusters which defines an equivalence relation on the edges. An edge bundle is an equivalence class of this relation. Coverage graph recipes are a method for compressing and uncompressing portions of a planar graph, for an efficient traversal of a data structure called a topology tree. These concepts are central to our improved data structure.

This chapter is organized in eight sections. In Section 3.2, we introduce the basic concepts; in Section 3.3, we describe the topology tree and prove a structural lemma about partially expanded topology trees. In Sections 3.4 and 3.5, we introduce edge bundles and coverage graphs. Section 3.6 describes the concept of recipes for building coverage graphs, and Sections 3.7 and 3.8 describe our query and update procedures.

3.2 Preliminaries

Let $G = (V, E)$ be a connected undirected planar graph, embedded in the plane. The initial embedding of G remains fixed throughout the course of the algorithm, and all the updates must respect the embedding. Let n denote the number of vertices of G and m the number of edges; the number of edges changes with updates, but planarity implies that $m \leq 3n - 6$.

As in Section 2.2 we perform a standard transformation on G to convert it into a graph with maximum vertex degree 3. Suppose $v \in V$ is a vertex of degree $d > 3$, adjacent to vertices $u_1, u_2, \ldots, u_d$ in this cyclic order. In the transformed graph, the vertex v is replaced by a cycle $(v_1, v_2, \ldots, v_d)$, and the edge (v, u_j), for $1 \leq j \leq d$, is replaced by the edge (v_j, u_j). In order to maintain the vertex identity, we label exactly one vertex on the cycle to be v, and leave the others unlabeled. At v we also store an ordered list of all the other vertices on the cycle. Observe that the transformed graph has $O(m)$ vertices, and that it inherits the two-edge connectivity properties of the original graph: two vertices are two-edge connected in G if and only if they are two-edge connected in the transformed graph [14]. Thus, from now on, we assume

that G has maximum vertex degree 3.

Our data structure is based on a spanning tree of G. The updates to the graph may force us to change this spanning tree, but between two successive updates we always maintain a spanning tree of G. We use this tree to build a hierarchical decomposition of G, define unique paths between two vertices, and other tasks. We use the notation T for the current spanning tree of G, and the notation $\pi(u, v)$ for the unique path in T between two vertices u and v.

Our first lemma establishes the important connection between two-edge connectivity and *edge-covering*. We say that a tree edge (x, y) is *covered by a non-tree edge* (v, u) if (x, y) lies on $\pi(u, v)$. The edge (x, y) is *covered* if any non-tree edge covers it.

Lemma 3.2.1 *Two vertices u and v in G are two-edge connected if and only if every edge of $\pi(u, v)$ is covered.*

Proof: If an edge of $\pi(u, v)$ is not covered, then its removal disconnects the graph and thus u and v are not two-edge connected.

If u and v are not two-edge connected, then there exists an edge e that is used by all paths from u to v (and thus also by the tree path). If e were covered, then there would exist a path from u to v that does not use e. Thus e is an edge on $\pi(u, v)$ that is not covered. ∎

This shows that a bridge is not covered.

3.3 Topology trees

We build a hierarchical representation of G based on the spanning tree T. The representation is a tree, called the *topology tree*, that has depth $O(\log n)$ [8]. Each level of the topology tree partitions the vertices of G into connected subsets called *clusters*. The partitioning on each level is similar to the rake/compress techniques used in parallel algorithms. Two clusters are said to be *adjacent* if they are joined by an edge in the spanning tree T. The *external degree* of a cluster is the number of tree edges with exactly one endpoint inside the cluster. A *boundary vertex* of a cluster

is the endpoint of such a tree edge that lies in the cluster. Our clusters will have maximum external degree 3. We now describe the rules for building the topology tree.

Every leaf of the topology tree corresponds to a cluster at level 0, which is a singleton vertex. An internal node of the topology tree corresponds to either a cluster of lower level, or the union of two adjacent lower level clusters subject to an external degree constraint. More precisely, a cluster at level i, for $i > 0$, is formed by either

1. the union of two *adjacent* clusters of level $i-1$ such that the external degree of one cluster is 1 or the external degree of both is 2, or

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

A cluster of level $i-1$ belongs to exactly one cluster of level i, thus ensuring that the vertices of G are partitioned at each level. Notice that our cluster-forming scheme implements a locally greedy heuristic and it does not always perform the maximum number of cluster unions. However, this locally greedy method of forming clusters is powerful enough for our purposes.

The topology tree has a node corresponding to each cluster that we form. The ancestor-descendant relationships in the tree encode the cluster containment information: the node corresponding to a cluster C at level i is the parent of the (one or two) clusters of level $i-1$ whose union produced C. The greedy method of forming clusters reduces the number of clusters at each level by a constant fraction, thus ensuring that the height of the topology tree is $O(\log n)$. The following lemma is due to Frederickson [9].

Lemma 3.3.1 ([9]) *The number of clusters at level $i+1$ is at most $5/6$ times the number of clusters at level i.*

Corollary 3.3.2 *The topology tree for the graph G has height $\Theta(\log n)$.*

When G is updated, the spanning tree T may change, and hence the topology tree may need restructuring. All restructuring operations follow a common routine: we break up all clusters containing some constant number of vertices, and then recombine

the unbroken clusters, perhaps in different combinations. In terms of the topology tree, this corresponds to removing all the nodes along a constant number of root-to-leaf paths, transplanting certain subtrees, and rebalancing the tree. To facilitate our discussion, we introduce the following terminology. To *expand the topology tree* at a vertex u means that we break up all the clusters that contain u; observe that the clusters containing u correspond precisely to the nodes lying along the path between u and the root of the topology tree. We denote by C_i a cluster at level i and by $C_i(u)$ the cluster at level i that contains the vertex u. A level $i-1$ cluster C_{i-1} contained in C_i is called a *sub-cluster* of C_i.

Let us consider the expansion of the topology tree at a vertex u. We start from the root, whose level is defined to be k, and walk down the tree, expanding the cluster $C_i(u)$ at each level. If $C_i(u)$ has only one child $C_{i-1}(u)$, we replace $C_i(u)$ by its child. Otherwise, $C_i(u)$ has two children, and we replace $C_i(u)$ by its two children and draw appropriate tree edges incident to the children from other clusters. In either case, we then recursively expand $C_{i-1}(u)$. The expansion stops when we reach the leaf-cluster representing the vertex u.

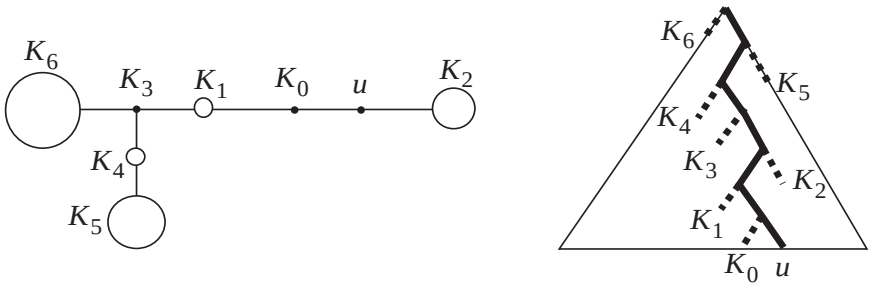

Figure 14: Expanding the topology tree at vertex u.

It is easy to see that, in the end, there is at most one cluster for each level i, $0 < i \leq k$: we create at most two clusters per level and recur on one of them. We use the notation K_i for the cluster that exists at level i after this expansion; at the leaf-level, K_0 denotes the cluster containing the vertex other than u. See Figure 14.

The expansion procedure results in a tree of clusters linked by edges of T, called the *expansion tree*. The following lemma shows that the tree is well structured.

Lemma 3.3.3 [Structural Lemma] *If we expand the topology tree of depth $D = O(\log n)$ at one vertex, the sum of the level differences between neighboring clusters in the expansion tree is at most $6D$.*

Proof: Removing u from the expansion tree splits the expansion tree into up to three subtrees. Thus it suffices to show that the sum of level differences between neighboring clusters in each subtree is at most $2D$.

Let T_u be one of these subtrees with u added. From the way we expanded the topology tree it follows that T_u has the two numbered properties below.

1. The levels of the clusters on each path to u are decreasing.
2. Let K_i have degree 3 in the expansion tree. At most one of the edges incident to K_i leads to a higher-level cluster with degree 3. If such a cluster exists, let its level be l. All the clusters on the other paths incident to K_i have level less than l.

Let K_i have degree 2 in the expansion tree. It follows from the first property that the levels of all the clusters on the one side of K_i are smaller than i and that the levels of all the clusters on the other are larger than i. Property 2 shows that for clusters with tree degree 3 an even stronger property holds.

Let K_p be the cluster at the highest level in T_u. According to Corollary 3.3.2, $p \leq D$. From property 1 it follows that the sum of the level differences between neighboring clusters on the path from K_p to u is $p \leq D$. Let K_q be the cluster at the highest level in T_u that is not on the path from K_p to u. Property 2 guarantees that the sum of the level differences on all other paths is at most $q \leq D$. Thus the sum of all the level differences in T_u is at most $2D$. ∎

Now suppose we wish to expand at two vertices u and v. Suppose that after expanding at u, K_j is the unexpanded cluster that contains v. We apply the same expansion procedure to $K_j = C_j(v)$ as we applied to $C_i(u)$. The same procedure lets

us expand the topology tree at any constant number of vertices in $O(\log n)$ time. The resulting partially-expanded tree has a constant number of clusters of each level. The proof of Lemma 3.3.3 extends to this case as well, establishing the following more general lemma.

Lemma 3.3.4 *If we expand the topology tree at a constant number of vertices, the sum of the level differences between neighboring clusters in the expansion tree is* $O(\log n)$.

To execute a query (u, v) or an update of an edge (u, v), we first expand the topology tree at u and then at v. In the case of a query, we answer the query and then merge the topology tree together in the same way as we expanded it. If we perform an update, edges and vertices are inserted and deleted (vertices are inserted or deleted to make sure that each vertex still has degree at most 3), and we may have to expand further at additional vertices. This changes the spanning tree and leads to a different topology tree, since only clusters that are neighbors in the spanning tree can be combined into larger clusters.

We merge back the clusters using a locally greedy heuristic: whenever possible, we merge two adjacent clusters of the lowest level, until the final topology tree is obtained. The merges respect the two clustering rules given earlier (see the full version of [9] for details). This process may force further expansion of some clusters. The number of expansions in each cluster is proportional to the sum of the difference between the level of the cluster and those of its neighbors. By Lemma 3.3.4, the total number of additional expansions clusters is $O(\log n)$.

3.4 Edge Bundles

In the previous section, we described a hierarchical method for storing the spanning tree T. We now describe a data structure, called an *edge bundle*, for storing non-tree edges. Edge bundles are a compromise between two conflicting requirements: (1) we want to be able to insert and delete edges quickly, and (2) we want to maintain

enough information about the non-tree edges to decide quickly whether a tree edge is covered or not.

Each edge (x, y) is stored twice, once as a directed edge from x to y, and once as a directed edge from y to x. Each edge is assigned a *target* as described below. The edges with one endpoint in a cluster C form a circular sequence in the embedding; an edge bundle is a maximal subsequence of non-tree edges with the same target. The target of an edge bundle is the target of its edges.

We use two different types of targets of edge bundles in our data structure. The default target of an edge bundle is the lowest common ancestor in the topology tree (LCA) of the endpoints of the constituent edges. That is, the target cluster of the bundle is the lowest node v in the topology tree such that the edges in the bundle have both endpoints inside the cluster represented by v. The advantage of this targeting scheme is that the target of an edge bundle is independent of how the topology tree has been expanded. A possible disadvantage is that the other ends of the edges in a single edge bundle may be incident to multiple clusters. In Figure 15(a), three clusters A, B, and C and three edge bundles are shown, one incident to each solid cluster. The target of all three bundles is the largest enclosing cluster $LCA(A, B, C)$, shown dashed.

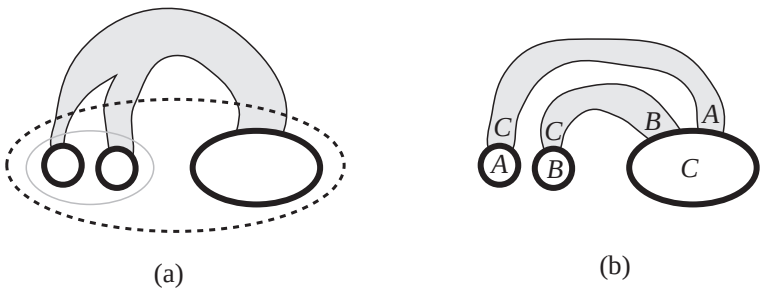

Figure 15: Different targeting schemes for edge bundles.

During queries and restructuring of the topology tree, we use *precise targeting* for edge bundles. In this case, the targets are defined relative to a particular expansion of the topology tree. Each edge bundle consists of a set of edges connecting exactly two clusters, and the target stored at each bundle is just the other cluster to which the bundled edges are incident. Figure 15(b) shows the edge bundles for the clusters

and edges of Figure 15(a), using precise targeting. The target of each of the four edge bundles is shown by a label near the bundle.

The data structure that represents an edge bundle consists of (1) a count of the number of edges in the bundle, and (2) the target cluster of the bundle.

3.5 Supernodes and Coverage Graphs

At each cluster that has external degree at most 2, we maintain one additional data structure, called a *coverage graph*. Suppose C is a cluster with external degree 2 and y, z are its boundary vertices. Furthermore, let $p = \pi(y, z)$ be the path in T between y and z, and let T' be the subtree of T that connects the vertices in C. We build a data structure that records which parts of p are covered by edges incident to C.

The following scenario provides a motivation for this data structure. Let (a, b) be a non-tree edge, with $a \in C$ and $b \notin C$. Suppose that b lies in the part of T that is connected to C by z. Then (a, b) covers the whole path from a to z in C. However, if an update changed T outside C, then b could suddenly be connected to C through y, in which case (a, b) would cover the path from a to y. See Figure 16.

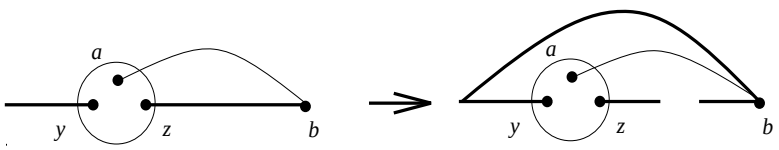

Figure 16: Changes in the path covered by an edge.

If we had stored the information that (a, b) covers the path from a to z, we would have to change it, which would make update operations too expensive. Now suppose that there is a second edge (a', b'), with $a' \in C$ and $b' \notin C$, and further suppose that b and b' are in the same cluster when the topology tree is expanded at a. Then the tree path between a and a' is covered, no matter how b and b' are connected to C. Thus, for the purpose of computing coverage, we can collapse the path $p \cap \pi(a, a')$ to a single node, which we call a *supernode*. Instead of remembering the path p, we store only the ordered list of supernodes at C.

More precisely, we define the *tree path* $p(C)$ *of a cluster* C. If C has external degree 1, $p(C)$ consists of its boundary vertex; if C has external degree 2, $p(C)$ consists of the path in T between the boundary vertices of C. As described below, we store $p(C)$ at C in a condensed form. Assume (a, b) and (a', b') are the extreme edges of an edge bundle, such that a and a' are contained in C. We say that the *projection of* a *on* p is the closest vertex to a in T' that lies on p. We call the tree path from the projection of a to the projection of a' the *projection of the edge bundle on* p. A subpath of $p(C)$ is *internally covered* if it is covered by edges that have both endpoints in C.

A *supernode* of a cluster C is a maximal subpath $\pi(x, y)$ of $p(C)$ such that $\pi(x, y)$ contains the projection of some edge bundle, and every edge of $\pi(x, y)$ either is covered by an edge internal to C or lies on the projection of some edge bundle on $p(C)$.

Since an edge bundle may be a single edge, a supernode may be just a vertex. The path from x to y is called the *path of the supernode*. The paths of two supernodes are disjoint; otherwise they would create a single supernode.

Instead of $p(C)$ we store at each C the list of supernodes in the order they appear along $p(C)$. Two consecutive supernodes are connected by a *superedge*. A superedge represents a subpath of $p(C)$ that is not completely internally covered. We call this representation of $p(C)$ by supernodes and superedges the *coverage graph of* C. If no supernode lies on $p(C)$, we introduce a single supernode to represent $p(C)$. This guarantees that if $p(C)$ has more than one supernode, then every supernode is incident to at least one edge bundle.

The data structure we use for a coverage graph of C is the following. If C has external degree 2, we have two doubly linked lists of edge bundles, one for each side of the tree path. Each list contains the edge bundles in the counterclockwise order of their embedding. The first edge bundle of each supernode in a counterclockwise traversal of the edge bundles is *marked*. The marker also keeps a skip count, that is, the number of consecutive supernodes following this supernode that have no edge bundles incident on this side. At the head of the list we keep an additional marker that indicates the number of consecutive supernodes with no edge bundles incident on this side before the first edge bundle. See Figure 17. If C has external degree 1, the coverage graph is much simpler. It consists of a single doubly linked list of the

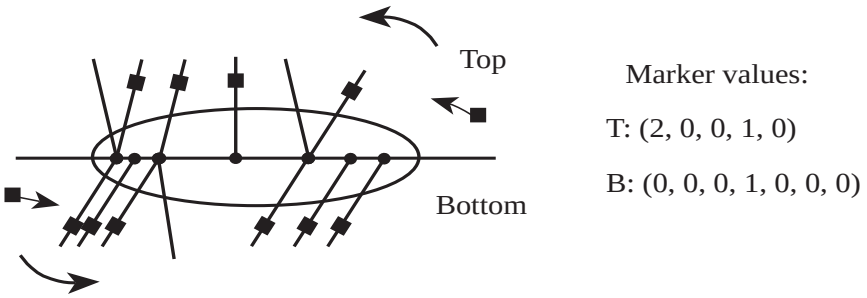

Figure 17: Storing the coverage graph.

edge bundles incident to C, in the counterclockwise order of their embedding. There is only one supernode in the coverage graph, so only one marker is needed in the list.

3.6 Recipes

We would like to store the coverage graph at each cluster, but that turns out to be too expensive, both in storage and time. Therefore we keep a *recipe* at each cluster C that explains how the coverage graphs of C's children can be computed from C's coverage graph. Concepts similar to recipes have been used before (see for example [**?**]. The coverage graph of the root of the topology tree is empty. We compute coverage graphs during a top-down traversal of the topology tree.

To compute the recipe for C, we build C's coverage graph by modifying the coverage graphs of its children, and then store at C a procedure that tells us how to reverse the construction. This procedure is the recipe. 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. Section 3.8 gives more details of the merging process.

Recipes use a specialized kind of pointer called a *location descriptor* to remember where the coverage graph of C has to be modified to create the coverage graphs of C's children. A location descriptor consists of a pointer to an edge bundle and an offset into the edge bundle (in terms of number of edges). It takes constant time to

follow a location descriptor.

We now describe the structure of the recipes, which varies depending on the number of children of C and their external degrees. We will describe the process of going from a child cluster to its parent. The recipe describes exactly the reversal of this process.

Case 1: C has only one child.

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.

Since the external degree of C is 2, its coverage graph consists of two edge bundle lists. Let Y and Z be the children of C and let Y have external degree 3. Then the coverage graph of Y is empty and the coverage graph of Z consists of one list of edge bundles (see Figure 18). Thus one of the edge bundle lists of C is empty, and the other one consists of the edge bundle list of Z. The coverage graph of C contains only one supernode, namely the vertex of Y. We set the markers appropriately. Since we can derive from the external degree of C and its children which actions to take when the topology tree is expanded, the recipe is empty.

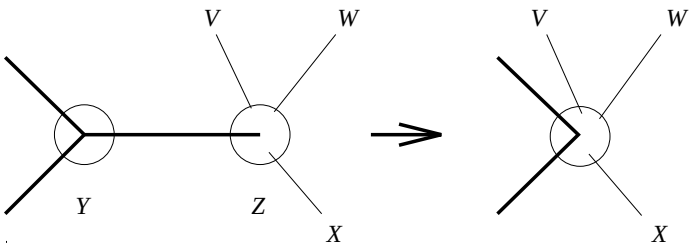

Figure 18: Merging a degree-3 cluster with a degree-1 cluster (Case 2).

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 graph of each child contains at most one edge bundle that

connects the two children. The recipe stores this edge bundle, i.e. the number of its edges. See Figure 19.

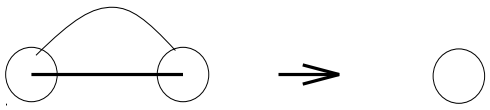

Figure 19: Merging a degree-1 cluster with a degree-1 cluster (Case 3).

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. On each side of the tree edge between Y and Z there may be an edge bundle that connects Y and Z. To create the coverage graph for C, these edge bundles have to be removed. Then all the supernodes between the bundle endpoints are compressed into one supernode. See Figure 20. To do this we concatenate each edge bundle list of Y with one end of the edge bundle list of Z so that the adjacent edge bundles in the embedding are also adjacent in the new edge bundle list. If two newly neighboring edge bundles have the same target, we merge them into one edge bundle. Then the markers for the supernodes are updated appropriately.

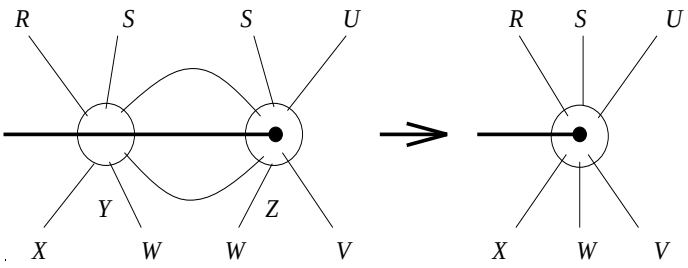

Figure 20: Merging a degree-2 cluster with a degree-1 cluster (Case 4).

The recipe contains two location descriptors to indicate where we concatenated the edge bundle lists (and possibly merged adjacent bundles). Additionally we store location descriptors pointing to the merged edge bundles. Finally we store a location descriptor for each compressed supernode in Y.

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

Let Y and Z be the children of C. On each side of the tree edge between Y and Z, there may be an edge bundle that connects Y and Z. To create the coverage graph for C, these edge bundles have to be removed. There may also be an edge bundle that starts in Y on one side of the spanning tree path, loops around the whole tree and ends on the other side of the spanning tree in Z. Section 3.8 describes how we find such an edge. We remove all the edge bundles connecting Y and Z and compress all the supernodes between the bundle endpoints into one supernode. To do this we concatenate the bundle lists of each side of the spanning tree path with each other and change the markers for the supernodes. We also merge newly adjacent edge bundles into a single edge bundle if they have the same target (see Figure 21).

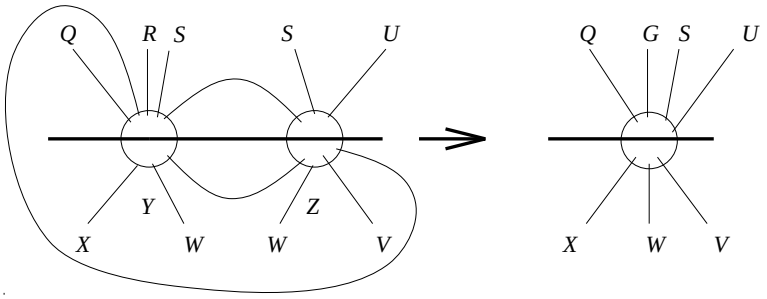

Figure 21: Merging a degree-2 cluster with a degree-2 cluster (Case 5).

The recipe contains two location descriptors to indicate where we concatenated the edge bundle lists (and possibly merged adjacent bundles). We also have to store location descriptors pointing to the merged edge bundles. If there is an edge bundle that loops around the tree we need two more location descriptors to mark its endpoints. Finally we have to store a location descriptor for each compressed supernode in Y and in Z.

Edge bundles may be created during recipe evaluation in cases 3, 4, and 5. For each new edge bundle, the recipe stores a bundle record, preloaded with the count of bundle edges, and a location descriptor pointing to the place in the old edge bundle list where

the new bundle is to be inserted. The target field of the bundle is easy to set: the LCA of the bundle edges is exactly the node at which the recipe is being evaluated. Each recipe evaluation creates $O(1)$ new bundles; this leads to the following lemma.

Lemma 3.6.1 *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 $O(\log n)$ and the recipe evaluation at each level creates $O(1)$ edge bundles, $O(\log n)$ edge bundles are created.

A supernode that is not incident to an edge bundle is the only supernode of a cluster. Since there are $O(\log n)$ clusters, there are $O(\log n)$ such supernodes. Each edge bundle creates at most two supernodes. Thus there are $O(\log n)$ supernodes incident to bundles and $O(\log n)$ supernodes altogether. Since the supernodes and superedges form a tree, there are $O(\log n)$ superedges.

The expansion of a cluster takes time proportional to the number of edge bundles and supernodes created. Thus the expansion takes time $O(\log n)$. ∎

After we expand the topology tree at a constant number of vertices, we are left with an $O(\log n)$-size collection of clusters and edge bundles. The bundles use LCA targeting. To answer queries or perform updates, we need to transform the edge bundles to use precise targeting.

During the retargeting procedure, each edge bundle may be split into several smaller bundles, though all remain incident to the same supernode. The reason for this follows: When edge bundles are created, they identify a set of parallel edges connecting two clusters Y and Z. During the rest of the expansion process, Y and/or Z may be further subdivided into clusters, and hence the original set of parallel edges may connect up to $O(\log n)$ clusters. However, planarity ensures that the edges do not cross, and hence we can match the edge bundles originally incident to Y with those originally incident to Z in the following way.

To retarget the bundles we traverse the (bidirected) expansion tree of supernodes and superedges by an Eulerian tour and number the edge bundles and spanning tree

edges we encounter consecutively starting with 1. See Figure 22. During the traversal we build at each cluster C an ordered list of edge bundles whose target is C. We can uniquely assign to each cluster a tree edge, namely the tree edge which is created when the cluster is expanded. Every tree edge receives two numbers during the Eulerian tour. Let n_1 and n_2 with $n_1 < n_2$ be the numbers of the tree edge of C. We want to merge the edge bundles with numbers between n_1 and n_2 with the edge bundles with numbers larger than n_2 or smaller than n_1.

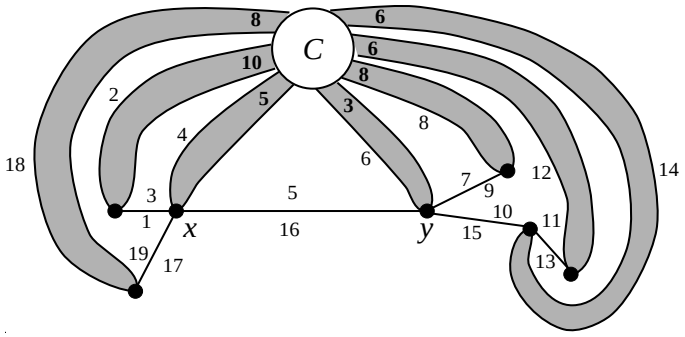

Figure 22: Before retargeting: All bundles in the cluster graph with target C. The number inside a bundle indicates its number of edges.

We start with the edge bundle with the smallest number larger than n_1 as a and with the edge bundle with the largest number smaller than n_1 as b and we perform the merging in the following way. (If $n_1 = 1$, then take the edge bundle with the largest number as b.) Let $n(a)$ be the number of edges in the edge bundle a. If $n(a) = n(b)$, the cluster of b becomes the target of a and vice versa. Then we repeat the process with the successor of a in the list as a and the predecessor of b as b. If $n(a) > n(b)$, the cluster of a becomes the target of b. We split a into two edge bundles, one with $n(b)$ edges going from the cluster of a to the cluster of b and one with $n(a) - n(b)$ edges (see Figure 23). Then we recurse with the predecessor of b as the new b. The case $n(a) < n(b)$ is symmetric. Since at each step the target of at least one edge bundle is found and there are $O(\log n)$ edge bundles, the total running time of the retargeting is $O(\log n)$. The number of edge bundles is at most doubled.

Lemma 3.6.2 *Suppose that the topology tree has been partially expanded to get a collection of $O(\log n)$ clusters joined by tree edges and LCA-targeted edge bundles.*

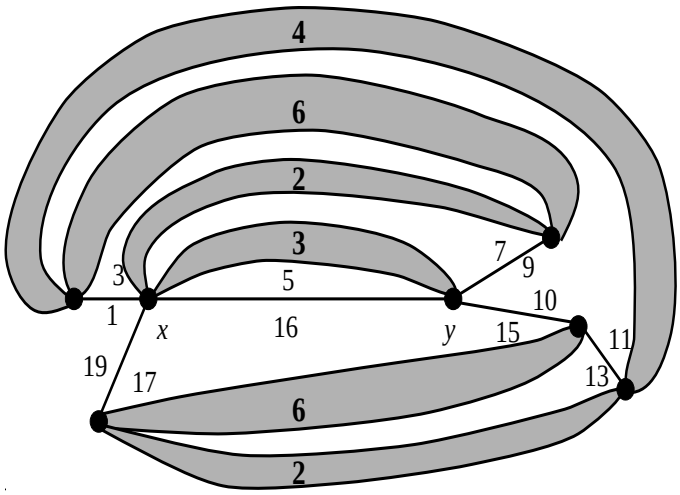

Figure 23: The cluster graph after the retargeting.

Then we can compute a new set of precisely targeted edge bundles, incident to the same supernodes, in $O(\log n)$ *time.*

For convenience, we refer to the graph formed by supernodes, superedges, and precisely targeted edge bundles as the *cluster graph*. The previous two lemmas imply the following lemma.

Lemma 3.6.3 *The cost of expanding the topology tree to obtain a cluster graph is* $O(\log n)$.

3.7 Queries

To answer the query whether two vertices u and v are two-edge connected, we want to test whether there is a bridge on $\pi(u, v)$; that is, whether any edge on $\pi(u, v)$ is uncovered. We expand the topology tree at u and v, which results in a cluster graph. By Lemmas 3.6.1 and 3.6.2, this graph has $O(\log n)$ supernodes, superedges and edge bundles. The vertices u and v are two-edge connected in the original graph if and only if $s(u)$ and $s(v)$ are two-edge connected in the reduced graph, where $s(u)$ and $s(v)$ are the supernodes containing u and v, respectively. We can check this in $O(\log n)$

time using a classical static algorithm. Then we merge the topology tree together in the same way we expanded it, leaving it as it was before the query.

Lemma 3.7.1 *Using the data structure described in Sections 3.3– 3.6, we can determine whether two query vertices u and v are two-edge connected in $O(\log n)$ time.*

3.8 Updates

The general structure of an update is as follows. First we expand the topology tree at a constant number of vertices. Next we retarget the edge bundles to give each edge bundle a supernode as a target (see Lemma 3.6.2). Now a cluster graph has been created. On this graph we perform our changes, which may include insertion and deletion of edges and vertices, and making non-tree edges into spanning tree edges and vice versa. This part varies depending on what kind of update we are performing. Finally we merge the topology tree back together. In the previous sections we explained how to create the cluster graph. In this section we explain the changes to the graph during updates and the merging of the topology tree. We support two kinds of updates, edge insertions and deletions. (It is straightforward to extend this to allow insertions/deletions of isolated vertices.)

3.8.1 Inserting an Edge

To insert an edge (u, v), we expand the topology tree at u and v, after first checking the degrees of u and v in the original graph. Let x stand for either u or v. If the degree of x before the insertion is 1 or 2, we expand the topology tree at x, create the cluster graph, and give x a new non-tree edge.

If the degree of x before the insertion is 3, we must create a four-vertex cycle for x and connect it appropriately. We expand the topology tree at x, then replace x by a four-vertex cycle. We connect the three original edges plus the newly inserted edge to the cycle in the proper order. We make three of the four cycle edges be tree edges; the last cycle edge and the newly inserted edge are non-tree edges.

If the degree of x becomes larger than 4, we already have a cycle for x. We are told between which edges incident to x in the original graph the new edge has to be inserted. Each of these edges is incident to exactly one vertex in the cycle. Therefore, we know between which two vertices x_1 and x_2 a new vertex has to be inserted. We expand the topology tree at these two vertices and create the cluster graph. We insert a new vertex x_3 between x_1 and x_2. The edge (x_2, x_3) is a tree edge; (x_1, x_3) is a tree edge if and only if (x_1, x_2) was a tree edge. The newly inserted edge incident to x_3 is a non-tree edge.

After making the appropriate modifications at u and v in the cluster graph, we merge the topology tree back together. (The details of this merge appear below.)

3.8.2 Deleting a Non-Tree Edge

Deleting a non-tree edge (u, v) is essentially the inverse of an insertion. Let x stand for either u or v. If the degree of x before the deletion is 1, 2, or 3, we expand the topology tree at x, create the cluster graph, and remove the non-tree edge from x.

If the degree of x before the insertion is 4, we must remove the four-vertex cycle for x. We expand the topology tree at all four vertices of the cycle and replace them by one vertex, which we connect appropriately to its neighbors.

If the degree of x is larger than 4 before the insertion, we already have a cycle for x. We are told (or can compute in constant time) between which edges incident to x in the original graph the edge to be deleted lies. Each of these edges is incident to exactly one vertex in the cycle. Therefore, we know the vertex x_3 that has to be deleted and its neighbors x_1 and x_2 in the cycle. We expand the topology tree at these three vertices, create the cluster graph, and delete the vertex x_3 and all three edges incident to it. If x_3 was connected to both x_1 and x_2 by tree edges, we connect x_1 and x_2 by a tree edge. Otherwise we connect x_1 and x_2 by a non-tree edge.

After making the appropriate modifications at u and v in the cluster graph, we merge the topology tree back together. (The details of this merge appear below.)

3.8.3 Deleting a Tree Edge

When a tree edge (u, v) is deleted, the spanning tree is broken up into two parts. We expand the topology tree at the endpoints of the edge and create the cluster graph. Then we run along one of the two faces adjacent to (u, v) and find a group of parallel edges (represented by a pair of edge bundles) that connects the two parts of the spanning tree. This can be done in $O(\log n)$ time by examining all the edge bundles in the cluster graph. (We are assuming here that the edge (u, v) was not a bridge edge of the graph G.) By repeatedly expanding the clusters incident to the pair of edge bundles, we can in $O(\log n)$ time identify a non-tree edge (u', v') that connects the two spanning tree components. We expand the topology tree at u' and v' and make (u', v') into a tree edge and (u, v) into a non-tree edge. Then we continue as for the deletion of a non-tree edge.

3.8.4 Merging the Topology Tree

We now describe how the topology tree is merged together after an edge insertion or deletion. The procedure has three steps. First, we compute the new topology tree for the updated cluster graph. Second, we compute the new edge bundles and their LCA targets based on the new topology tree. Third, we update the recipes in all the clusters affected by the changed edge bundles. There are $O(\log^2 n)$ such clusters, and we spend amortized constant time apiece, for a total update time of $O(\log^2 n)$; all the other operations take $O(\log n)$ time.

We first determine the structure of the new topology tree for the modified cluster graph, as described in Section 3.3. This step is just preparatory for the full reconstruction of edge bundles, supernodes, and recipes, and does not involve any of those elements. We expand all the clusters affected by the restructuring algorithm (still only $O(\log n)$ clusters) to get a larger cluster graph.

The clusters of the cluster graph correspond to a *fringe* of nodes in the new topology tree: every node above them in the tree has been expanded, and no node below them has been expanded. To create recipes for the nodes above the fringe, we must compute the new edge bundles and their LCA targets based on the new

topology tree. To do this we first compute precise targets for the edge bundles in the cluster graph. Now each edge bundle links two clusters. For each pair of linked clusters, we compute their lowest common ancestor in the new topology tree and label each edge bundle with the LCA as its target. This takes $O(\log n)$ time altogether. Finally we merge any edge bundles that have the same target and are adjacent along the boundary of some cluster. This may involve merging supernodes as well.

There are two sets of topology tree nodes (clusters) for which recipes need to be computed. First, it is clear that the nodes above the fringe in the topology tree must have recipes created for them: these nodes may never have had recipes at all. Second, and less obviously, we must recompute recipes for certain nodes (clusters) below the fringe. The affected clusters are those that contain any of $O(\log n)$ vertices, namely the extreme vertices of edge bundles in the original or revised cluster graph. The reason is as follows: Even if a cluster C was not expanded, its ancestors in the new topology tree might be different from the ones in the old topology tree. Assume the recipe of C has a location descriptor pointing to an edge bundle e. If e was split during the retargeting into e' and e'', then the location descriptor in the recipe of C is outdated and thus has to be updated. Additionally the supernode to which e was incident might be split into two supernodes. Note, however, that this can happen only in clusters that contain the extreme vertices of edges bundles in the original or revised cluster graph.

To prepare for recipe (re)creation, we expand the topology tree yet again at all $O(\log n)$ of the vertices identified above, then set the targets for edge bundles according to LCAs in the new topology tree. The $O(\log^2 n)$ expanded nodes lie on $O(\log n)$ disjoint monotone paths in the topology tree; call these *expansion paths*. The expansion paths can be partially ordered by "aboveness": one path is above another if any of its nodes is an ancestor of a node on the other path. We process the expansion paths in ascending order of aboveness. The clusters that are children of a path are incident to a total of $O(\log n)$ edge bundles: this follows because the cluster at the top of the path has $O(\log n)$ incident bundles, and we know from Section 3.6 that the expansion of each cluster along the path adds $O(1)$ LCA-targeted edge bundles incident to the child clusters of the path.

We compute the coverage graphs and the recipes along each expansion path by bottom-up merging (cf. Section 3.6): for each cluster we combine the coverage graphs of its sub-clusters, then record how to reverse the operation in a recipe. The aboveness ordering of the expansion paths ensures that when we want to merge a node on the path with one off the path, we will have already computed coverage graphs for both nodes. We process each path in $O(\log n)$ time.

We now present a high level algorithm for merging two clusters; the subroutine details appear below. Each of the two clusters to be merged has a coverage graph as described in Section 3.5. We also assume that each edge bundle has been labeled with the depth of its target in the topology tree (labeling takes constant time per bundle).

We first identify the edge bundles incident to both clusters that will be internal to the merged cluster. These are exactly the bundles whose targets have the same depth as the merged cluster; we find them by selecting bundles by depth. Second, we identify the supernodes of the coverage graphs to which these bundles are incident. These supernodes and all the supernodes on the tree path connecting them will be coalesced. We identify the supernodes to coalesce by walking outward from the tree edge joining the two clusters. It follows that coalescing takes time proportional to the number of supernodes being coalesced, and hence amortizes to $O(\log^2 n)$ overall. (This description is most applicable to the case of merging two degree-two nodes into one; there are minor differences when any of the nodes has degree one or three, but the basic ideas are the same.)

After we identify the bundles that will be internal to the merged node, we follow three steps: (1) delete these bundles; (2) coalesce supernodes; and (3) merge adjacent bundles that have the same target. The recipe is just the reverse of the merging operation.

There are two subtasks to be explained more fully: (1) identifying bundles of a particular depth, and (2) locating these bundles' supernodes in the coverage graph.

Each of the $O(\log n)$ expansion paths in the topology tree has $O(\log n)$ nodes that are children of the path. When we process the path, these nodes already have coverage graphs; we call these nodes *static nodes*. The nodes on the path are called

dynamic nodes. We compute recipes and coverage graphs for the dynamic nodes by walking up the path from bottom to top. The following operations are applied on each expansion path separately.

Operation 1: *Identify the bundles of a given depth at each cluster along an expansion path*. We maintain an array of length $O(\log n)$, indexed by depth, of the edge bundles incident to the current dynamic cluster. Each array entry points to a circular list of bundles with the same depth. We initialize the array at the node at the bottom of the expansion path, then maintain it as we walk to the top of the path. Thus, the initialization overhead is $O(\log n)$, but the lookup time is constant, as desired. When we merge in the bundles from a static node, we can afford to look at all of them (because there are only $O(\log n)$ edge bundles incident to the static nodes altogether) and insert them into the proper array slots.

Operation 2: *Given an edge bundle, map it to the supernode in the coverage graph to which it is incident*. To do this we use a bit vector of length $O(\log n)$, each bit corresponding to one of the edge bundles incident to the clusters on the expansion path. We use the predecessor operation ("find the first 1-bit left of the query position") to identify the supernode to which an edge bundle is incident. This operation takes constant time on a RAM with word size $\Omega(\log n)$.

To map edge bundles to bit-vector indices, we perform an Eulerian tour of the static clusters belonging to the expansion path, numbering the edge bundles in order along the way. The bundles incident to any cluster are ordered consistently with this numbering.

The 1-bits in the bit vector represent the edge bundles that are directly linked to their supernode (these are the first and last bundles in the supernodes' edge bundle lists). To find the supernode to which an edge bundle belongs, we do a predecessor query on the bit vector. For each expansion path, setting up the bit vector takes $O(\log n)$ time. Maintaining the bit vector as the supernodes coalesce and as the clusters represented by static nodes are merged into the cluster represented by the dynamic node also takes $O(\log n)$ time per expansion path. We use the bit vector to

perform $O(\log n)$ constant-time queries.

The preceding discussion establishes the following update lemma.

Lemma 3.8.1 *The two-edge connectivity data structure can be updated in response to an edge insertion or deletion in $O(\log^2 n)$ time.*

All the operations needed to merge two clusters can easily be performed in time proportional to the number of edge bundles incident to the clusters. This means that our data structure can be built from scratch by bottom-up merging in $O(n)$ time. This observation finishes the proof of our main theorem.

Theorem 3.8.2 *An embedded planar graph can be preprocessed into a linear-space data structure that supports insertion or deletion of an edge in $O(\log^2 n)$ time and answers a two-edge connectivity query between two vertices in $O(\log n)$ time. If the graph has n vertices, then the preprocessing cost is $O(n)$.*

Chapter 4

Planarity Testing in Graphs

4.1 Introduction

The study of dynamic connectivity algorithms for plane graphs has been very active. As described in Chapter 1 fully dynamic algorithms in plane graphs are faster than in general graphs. Additionally, they have many applications, for example in VLSI design, graphics, and vision. These algorithms expect to be told during every edge insertion in which face this edge is inserted. If the insertion of the edge in the given face does not preserve planarity, the algorithms are not guaranteed to be correct. However, none of these algorithms can test whether this condition is fulfilled. This is the motivation to study a fully dynamic algorithm that allows queries of the form "Can an edge be inserted between vertex u and v without destroying the planarity of the embedding". Such an algorithm is called an algorithm for *fully-dynamic planarity testing in plane graphs*. Given a plane graph $G = (V, E)$ the algorithm supports the following operations:

$Insert(u, v)$: Insert an edge between the two vertices u and v.

$Delete(u, v)$: Delete the edge (u, v).

$Query(u, v)$: Return *false* if the insertion of an edge between the vertices u and v does not maintain the planar embedding of the graph and return a face into which the edge can be inserted otherwise.

Our data structure supports online updates and queries in $O(\log^2 n)$ time, where n is the number of vertices in the graph. It can be built in linear time and requires linear storage.

The new techniques introduced in this chapter are *coverage bits*, *coverage lists*, and *face lists*. A coverage bit of a face stores information whether a certain vertex is incident to this face. A coverage list of a cluster is a list of coverage bits of faces incident to the cluster. A face list of a cluster stores faces incident to the cluster whose coverage bit is one. In Sections 3.2, 3.3, and 3.4 we explained how to convert G into a degree three graph and we defined clusters, topology trees, and edge bundles. We will assume in this chapter that the reader is familiar with these definitions.

In Section 4.2 we describe the basic data structure. Section 4.3 introduces coverage bits, coverage lists, and face lists. Section 4.4 contains a description of recipes. Sections 4.5 and 4.6 discuss queries and updates, respectively.

4.2 Preliminaries

We are given a plane graph G with n vertices and $m = O(n)$ edges that we want to transform into a degree-3 graph. In Section 3.2 each vertex of degree $d > 3$ is replaced by a cycle of d vertices and edges. In this chapter each such vertex is replaced by a chain of $d-1$ dashed edges in the following way. Every vertex of degree $d > 3$ is replaced by d vertices $v_1, v_2, \ldots, v_d$. Vertex v_i is connected by an edge to the ith neighbor u_i of v. Additionally, *dashed* edges (v_i, v_{i+1}) for $1 \leq i < d$ are added to the graph so that the order of the edges at vertex v_i is $(v_i, v_{i-1}), (v_i, u_i), (v_i, v_{i+1})$. Thus the transformation replaces each vertex by a chain of dashed edges and creates a plane graph G' that fulfills the following lemma.

Lemma 4.2.1 *An edge between vertex u and vertex v can be added to G without destroying the embedding if and only if an edge can be added between a vertex of the chain of u and a vertex of the chain of v without destroying the embedding of G'.*

Now we find a spanning tree T of G' that contains all dashed edges and build a topology tree for T. This tree has depth $O(\log n)$ (see Corollary 3.3.2). Each node in the topology tree represents a cluster.

As in Chapter 3 a data structure called an *edge bundle* is used for storing non-tree edges. Let C be a cluster and let e be a non-tree edge incident to C. As described below each edge is assigned a *target*. All edges incident to C are embedded in a circular sequence around C: an *edge bundle* is a maximal subsequence of edges that share the same target.

We actually use two different types of targets. The grouping of the edges into bundles varies with the targets. Usually the target of an edge is the least common ancestor (LCA) of its endpoints in the topology tree and an edge bundle is a maximal subsequence of edges at C with the same LCA. After the topology tree has been expanded and $O(\log n)$ clusters are created, the target of an edge is changed to be the one cluster (of the $O(\log n)$ created clusters) that contains the other endpoint of the edge. The edge bundles are thus broken up appropriately. The target of a bundle is the target of its edges. The first representation is called *LCA-targeting*, the second *precise targeting*. The process of changing the target is called *retargeting*. During precise targeting each edge bundle is stored as two directed edge bundles, one in each direction. As in the previous chapter we expand the topology tree at a constant number of vertices to execute a query or an update operation. Then we retarget the edge bundles to create the *cluster graph*. The vertices of the cluster graph are the clusters in the topology tree that are children of expanded clusters, but that have not been expanded. The cluster graph has size $O(\log n)$ and is used to answer the query or execute update operations. Afterwards the clusters in the cluster graph are merged together according to the rules for topology trees, thereby building a new topology tree.

The data structure for an edge bundle is a record that contains (1) the number of edges in the bundle, (2) the two extreme (i.e., the first and the last) edges of the bundle, (3) the target cluster of the bundle, (4) the reverse edge bundle if precise targeting is used, and (5) the one or two faces that point to it (see next section).

During the expansion of the topology tree we build a list of edge bundles at each expanded cluster C. Whenever an edge bundle is split into two, we are given the number of edges incident to C and to the left of a specific edge (directed away from C) and we have to split the edge bundle list at this edge into two lists. Whenever

two edge bundles are joined together, two edge bundle lists have to be merged.

We store every edge bundle list in a balanced binary tree. At each edge bundle we keep its edge count and at each internal node of the balanced tree we record the sum of the edge counts of the leaves in its subtree. If the topology tree is expanded at a constant number of vertices, no more than $O(\log n)$ edge bundles are stored in an edge bundle list, as we show later. Thus each of the above operations can be supported in time $O(\log \log n)$.

4.3 Coverage Lists and Face Lists

As in Chapter 3 we need some additional information at the nodes of the tree. However, we do not use supernodes or coverage graphs. Instead we keep with each edge bundle list two additional lists, called the *coverage list* and the *face list*.

Let C be a cluster. An edge $e = (u, v)$ of T whose endpoints are both in C is said to be *internal* to C. If exactly one endpoint of e, say u, is in C, e is said to be *incident* to C. If e is additionally a tree edge, then u is called a *boundary vertex* of C. We define the *tree path* $p(C)$ *of a cluster* C as follows. If the cluster contains only one vertex, the path of the cluster consists of this one vertex. A tree path of a cluster with more than one vertex consists of the tree path of its child (in the topology tree), if it has only one child, and it consists of the concatenation of the tree paths of the two children otherwise (see Figures 25– 29). Thus the tree path of a cluster C with external degree 3 consists of the only vertex of C. The tree path of a cluster C with external degree 2 consists of the path in T between the boundary vertices of C. In a cluster C with external degree 1, $p(C)$ consists of the path in T from a leaf of T to the boundary vertex of C. It seems that there might be several ways of defining a tree path. We choose this particular representation in order to avoid technical problems that will become apparent in Section 4.4.

Let (a, b) be an edge incident to C and let a be contained in C. We define the *projection of* a *on* $p(C)$ to be the vertex closest to a in T lying on $p(C)$.

For each cluster C and for each face incident to C we define a bit called *coverage bit*. Let (x, y) and (x', y') be the edges incident to F such that x and x' are contained

in C. The bit of F is set to one if and only if at least one of the vertices on the tree path between the projection of x on $p(C)$ and the projection of x' on $p(C)$ is incident to F. Otherwise it is set to zero. A face that is incident to two edges between cluster C and cluster C' is called a *good* face between C and C' if and only if the coverage bit at C and the coverage bit at C' are one. The motivation is that an edge between a vertex on $p(C)$ and a vertex on $p(C')$ can be inserted into G if and only if there exists a good face between C and C'.

We keep a list of the coverage bits of every face that lies between two neighboring edge bundles incident to C or between the first or last edge bundle and a tree edge incident to C. If C has external degree 1, we build one coverage list for C. If it has external degree 2, we create two coverage lists, one for each side of $p(C)$. See Figure 24. If C has external degree 3, no coverage list is stored. The same operations

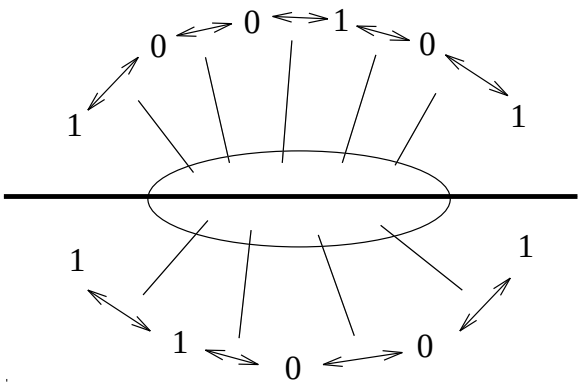

Figure 24: A cluster with is edge bundles and coverage lists.

as on edge bundle lists are executed on coverage lists. Thus each coverage list is stored in a balanced binary tree with the faces in counterclockwise order of their embedding at C. At each face F we keep the edge count of the edge bundle that is incident to C and following F in the counterclockwise order of the embedding at C and a pointer to the edge bundle. The last face in the coverage list points to the edge bundle preceding it in the counterclockwise order. At each edge bundle we keep a pointer to the one or two faces that point to it. As in the balanced tree for the edge bundle list, each internal node of the balanced tree records the sum of the edge counts of the leaves in its subtree. Since there are $O(\log n)$ coverage bits in a coverage list

(when the topology tree is expanded at a constant number of vertices), each split or merge can be performed in time $O(\log \log n)$.

Additionally we store at each cluster C a list of all of the good faces between C and other clusters, called the *face list*. For each face in the face list of C there exists a cluster C' such that the face is incident to two neighboring edges of the same bundle between C and C' and is a good face between C and C'. The face is said to *belong* to this edge bundle. Throughout any sequence of operations, the face list will be subject to splits (whenever an edge bundle is split into two), merges (whenever two edge bundles are joined together), and tests for emptiness (whenever during a query, we check if the edge to be inserted can be embedded inside a bundle). There can be $O(n)$ faces in a face list. To support these primitives in $O(\log n)$ time, we store each face list in a balanced binary tree with the faces in the counterclockwise order of their embedding around C. At each face F we keep the number of edges incident to C and lying between F and the next good face of C to the left of F or the next tree edge, whichever comes first. At each internal node we store the sum of the edges at the leaves in its subtree. Given an edge e incident to C (identified by the number of edges incident to C and to the left of e), we can split the face list in time $O(\log n)$.

4.4 Recipes

As in Section 3.6 we cannot afford to store the edge bundle, coverage, and face lists at each cluster. So, again we use the concept of recipes. At each cluster C we keep a *recipe* that explains how the lists of C's children can be computed from C's lists. Using this recipe, we are able to reconstruct all the information about the edge bundle, coverage, and face lists of each cluster from its parent cluster. The lists of the root of the topology tree are empty. The lists of edge bundles, the coverage lists, and the face lists are computed during a top–down traversal of the topology tree.

The recipe is a procedure that describes how to build the lists of the children of C if the lists at C are known. The recipe is created by recording all changes made to the lists of the children of C when they are merged to form the lists for C during the merging of the topology tree. We use the recipe when we expand the topology tree.

Recipes store pointers to elements of the lists to record where the lists have to be changed. These pointers are called *location descriptors*. We say that an edge e incident to a cluster C *lies to the left* of an edge or face incident to C if e follows the other edge or face in the counterclockwise embedding of the edges at C and there is no tree edge between them in this order. A location descriptor is a number indicating the number of edges incident to C, lying to the left of the edge or face indicated by the location descriptor. (An edge bundle consisting of p edges is counted as p edges.) We thus define a location descriptor independently of edge bundles, because the number of edge bundles to the left of an edge or a face incident to C can vary during the algorithm, but not the number of edges to their left. During the expansion of the topology tree and during the merging back of the topology tree each edge bundle list and each coverage list contains $O(\log n)$ elements. Thus it takes time $O(\log \log n)$ to find the edge or face to which a location descriptor points. Computing the location descriptor for an edge or face takes time $O(\log \log n)$ as well. Since there can be $O(n)$ faces in a face list, these operations take time $O(\log n)$ in a face list.

We now discuss what information is stored in the recipes. Since this depends on the number of children of C and their external degrees, we consider the following five cases.

Case 1: C has only one child.

> The lists of C are identical to the lists of its child. The recipe is therefore empty.

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

> Since C is incident to two tree edges, there are two edge bundle lists, two coverage lists, and two face lists stored at C. Let Y and Z be the children of C, with Y having external degree 3 (see Figure 25). From the construction of the topology tree it follows that Y consists of only one degree-3 vertex that is incident to three tree edges. This implies that the lists of Y are empty. Hence the edge bundle list and the face list on one side of the tree path of C are empty and the coverage list consists of one bit set to one (since the node

in Y is incident to the face between the two tree edges of C). Now let us discuss the lists on the other side. Cluster Z keeps one edge bundle list, one coverage list, and one face list. The edge bundle list on the other side is the list of Z. The tree path of C consists only of one vertex, namely the vertex of Y. Only the faces incident to the tree edge between Y and Z are incident to this vertex. Thus the coverage list of C consists of a list of zeros (as many as there are faces incident to Z that are not incident to Y) with a one at the beginning and at the end. The face list is empty. The coverage list and the face list of Z are stored in the recipe.

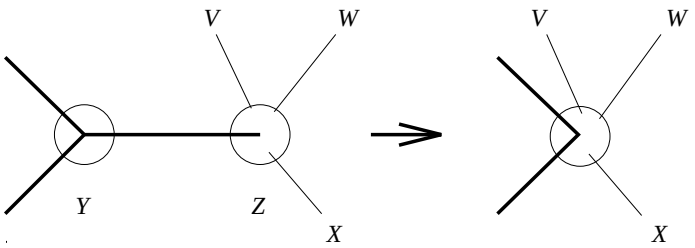

Figure 25: Merging a degree-3 cluster with a degree-1 cluster (Case 2).

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

This is only possible if C is the root of the topology tree. Thus its lists are empty. The children are connected by at most one edge bundle that is stored in the recipe together with the two coverage bits, one for each face (see Figure 26). Storing an edge bundle means that we record its number of edges and its extreme edges. Additionally we keep the face list of Y (that is identical to the face list of Z), the coverage list of Y, and the coverage list of Z.

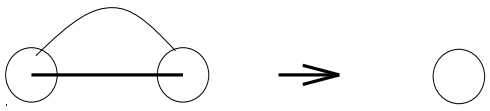

Figure 26: Merging a degree-1 cluster with a degree-1 cluster (Case 3).

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

Let Y and Z be the children of C, let Y be incident to two tree edges, and let Z be incident to one tree edge. The clusters Y and Z can be connected by an edge bundle on each side of the tree edge between Y and Z (see Figure 27). We have to remove these edge bundles from the edge bundle lists and the corresponding coverage bits from the coverage lists. Section 4.6 describes

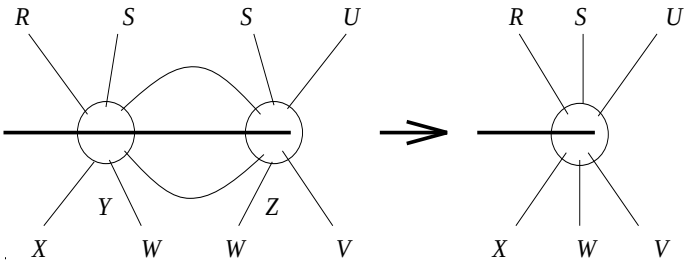

Figure 27: Merging a degree-2 cluster with a degree-1 cluster (Case 4).

how to find these edge bundles. Then we delete all faces of the face lists that belong to these bundles. These faces are contiguous in the face list.

Afterwards we concatenate each edge bundle list of Y with one end of the edge bundle list of Z so that edge bundles that are adjacent in the embedding are adjacent in the new edge bundle list and we compute the coverage bits of the faces between them. Each such coverage bit is computed by executing an OR-operation on the last coverage bit of the coverage list of one child and on the first coverage bit of the coverage list of the other child. The coverage bits are added to the beginning and to the end of the coverage list of Z. Then the coverage lists of Y and of Z and the face lists are concatenated in the same way as the edge bundle lists. If two newly neighboring edge bundles have the same target, we join them to one edge bundle in the edge bundle list. Then we remove the coverage bit of the face between them from the coverage list and add the face to the face list if it is good. We describe below how to test for goodness.

If Y has a self–loop, i.e. an edge bundle that is contained in both edge bundle lists, we remove it and its corresponding bits. See Figure 28. Note that such an edge is easy to find. If it exists, Y is the only cluster adjacent to Z. The

self–loop is either adjacent to the edges between Y and Z or (if no such edges exist) it is the last edge of one edge bundle list of Y and the first edge of the other edge bundle list.

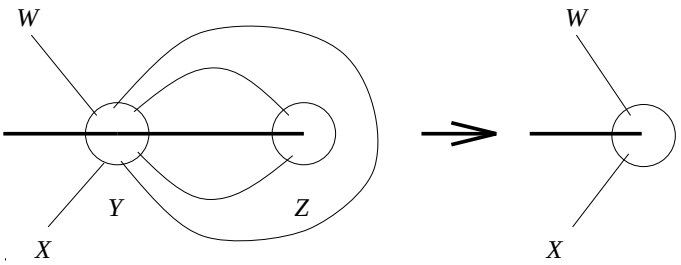

Figure 28: Merging with self–loops (Case 4).

We describe the recipe for one side of the tree path of Y. The recipe for the other side is found in the same way. The recipe contains the number of edges incident to the left cluster (on this side if it is a degree two cluster). This number is used as a location descriptor to split the edge bundle, coverage, and face list on this side. If there is an edge bundle between Y and Z we keep its edge count, its extreme edges, the part of the face list that contains faces belonging to the bundle, and the coverage bits (at Y and at Z) of the face defined by the bundle and the tree edge between Y and Z.

After removing the edge bundle between Y and Z, if there is no self–loop, we additionally keep the rightmost edge of the left cluster, the leftmost edge of the right cluster, the coverage bit at Y, and the coverage bit at Z of the face between the two edges. It is necessary to store the edges, since splitting the cluster might cause splitting an edge bundle, and the stored edges are the extreme edges of the two new bundles where the original bundle was split.

If there is a self–loop, we store its edge count, its extreme edges, the part of the face list that contains faces belonging to it, and the coverage bits at both sides of Y of the face defined by the self–loop and Y.

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

Let the children of C be called Y and Z. The clusters Y and Z can be connected on each side of the tree path by an edge bundle. These edge bundles have to be removed to build the lists for C. It is also possible that an edge bundle starts in Y on one side of the spanning tree path, loops around the whole tree and ends on the other side of the spanning tree in Z (see Figure 29). Such an edge is called a *loopy* edge. We describe in Section 4.6

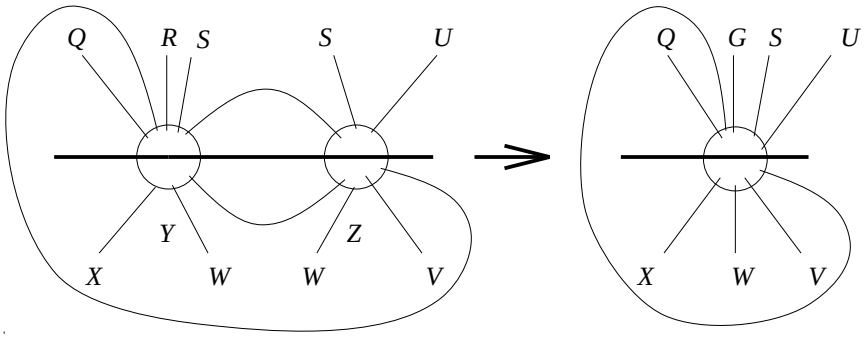

Figure 29: Merging a degree-2 cluster with a degree-2 cluster (Case 5).

how we find such an edge. We remove all the edge bundles connecting Y and Z, except for the loopy edge bundle. Then we delete the appropriate coverage bits from the coverage lists, and the appropriate parts of the face list from the face list. The loopy edge splits the plane into two parts, each containing at least one cluster. It is the witness that no edge between the inside and the outside clusters can be inserted. Thus we have to keep it as long as there are inside or outside clusters. Then it is removed (as a self–loop in Case 4).

Next we concatenate the bundle lists on each side of the spanning tree path with each other and determine the coverage bits for the faces between them. Afterwards we add the new coverage bits to the coverage lists and concatenate them and also the face lists appropriately. If two adjacent edge bundles have the same target, they are joined into one edge bundle. Additionally we remove the according coverage bits from the coverage list and record their faces in the face lists if they are good. The test whether a face is good is described below.

We explain next what we store in the recipe for one side of the tree path of

C; the recipe for the other side can be found symmetrically. Without loss of generality let Y be the left cluster on this side. The recipe keeps the number of edges incident to Y. This number is used as a location descriptor to split the edge bundle, the coverage, and the face list when C is expanded. If there is an edge between Y and Z, the recipe also contains this edge bundle (its edge count and extreme edges), the part of the face list that contains faces that belong to the bundle, and the coverage bits (at Y and at Z) that belong to the face between the bundle and the tree edge between Y and Z. After removing this bundle from the edge bundle lists of Y and Z, we store the rightmost edge of Y and the leftmost edge of Z in the recipe. We also keep the coverage bits that are stored at Y and at Z for the face between these two edges. If expanding C splits an edge bundle (on this side of $p(C)$), these edges are two of the extreme edges of the new bundles.

During each merge of cluster Y and Z (with Y a cluster of external degree 2) we might have to test whether up to two faces are good faces, one on each side of the tree path of Y. Each face F is incident to Y and Z. Without loss of generality let Y be on the left of Z. To test the goodness of F we check the coverage bits of all the clusters in the cluster graph that is incident to F and not contained in the two clusters. Starting at the bundle incident to Y and adjacent to F we process all appropriate clusters in the following way. We go to the reverse bundle of the current bundle, make its successor in the counterclockwise order of bundles at its cluster the current bundle, and check the coverage bit of the face between them to whom the current bundle keeps a pointer. We repeat this step until we reach Z. If any of the coverage bits was one and the coverage bit at Y or Z is one, F is a good face. Otherwise, F is not a good face. Since there are $O(\log n)$ clusters in the cluster graph, this takes time $O(\log n)$.

The creation of the list for C from the lists for children of C requires either the join of two edge bundle, coverage, and face lists (Case 4 and 5) or the creation of a new coverage list with $O(\log n)$ entries (Case 2). Each concatenation of two edge bundle lists or two coverage lists requires the merging of two balanced binary trees of size $O(\log n)$ and thus takes time $O(\log \log n)$. Each insertion into a face list and

each join of two face lists takes time $O(\log n)$, since $O(n)$ faces can be stored in a face list. Thus two clusters can be merged in time $O(\log n)$ to form their parent.

As in Section 3.6 a constant number of edge bundles is created during each recipe evaluation and the target of each edge bundle is set to be the node at which the bundle was created. The reason is that this node is the LCA of the endpoints of the bundle.

Lemma 4.4.1 *The expansion of the topology tree at a constant number of vertices takes $O(\log^2 n)$ time and creates $O(\log n)$ edge bundles and coverage bits.*

Proof: The topology tree has depth $O(\log n)$ and one recipe is evaluated at each level. As mentioned above, a constant number of edge bundles is created during each recipe evaluation. Thus $O(\log n)$ edge bundles are created and there are $O(\log n)$ faces in the graph created by the expansion. Each face contributes one coverage bit. Thus there are $O(\log n)$ bits altogether. During the expansion, $O(\log n)$ location descriptors are evaluated and $O(\log n)$ lists are joined. An evaluation or a join in an edge bundle list or a coverage list takes time $O(\log \log n)$, an evaluation or a join in the face list takes time $O(\log n)$. Additionally at most $O(\log n)$ new coverage lists of $O(\log n)$ bits each are created. Thus the total time for expanding the topology tree is $O(\log^2 n)$. ∎

After the expansion of the topology tree at a constant number of vertices, all edge bundles are retargeted and the cluster graph is built. We showed in Section 3.6 that retargeting takes time $O(\log n)$ plus the time for splitting $O(\log n)$ edge bundles. Each split of an edge bundle causes an insertion in an edge bundle list and in a coverage list and a possible deletion in a face list. The first two operations take time $O(\log \log n)$, while a deletion in a face list takes time $O(\log n)$.

Additionally, for each cluster C at which an edge bundle was split and a new face F was created we have to compute the coverage bit for F at C. Let (x, y) and (x', y') be the two edges of the split bundle that are incident to F. These edges are known since we store with each edge bundle its extreme edges. Further, let x and x' be in C. To find the coverage bit for F we run up the paths in the topology tree

from x and from x' until we find the first node (cluster) that contains both x and x'. The edge bundle containing (x, y) and the edge bundle containing (x', y') cannot be merged together before this node and they are merged together at C. Thus they are combined either at the least common ancestor of x and x' or at one of its ancestors in the topology tree (up to C). At the node C' at which they are merged, the coverage bits of F for both children of C' are stored. An OR-operation on the two bits gives us the coverage bit for F at C. The node C' can be found by testing whether (x, y) and (x', y') are stored in the recipe starting from the least common ancestor of x and x' and going up the topology tree until such a node is found. This takes time $O(\log n)$ per split edge bundle. Since there are $O(\log n)$ edge bundles split altogether, the following lemma is established.

Lemma 4.4.2 *If the topology tree is expanded at a constant number of vertices, we can compute a new set of precisely targeted edge bundles in $O(\log^2 n)$ time.*

The previous two lemmas show the following lemma.

Lemma 4.4.3 *The cost of expanding the topology tree at a constant number of vertices to obtain a cluster graph is $O(\log^2 n)$.*

4.5 Queries

To decide whether a new edge between u and v destroys the planar embedding of the graph, we expand the topology tree at the first and last node of the dashed chain of u and at the first and last node of the dashed chain of v, which results in a cluster graph. By Lemma 4.4.1, this graph has $O(\log n)$ clusters, edge bundles, and faces. Then we split the face list(s) at each cluster according to edge bundles. Thus each edge bundle receives a list of good faces that are incident to two edges of the bundle.

In this section a cluster that contains one of the vertices on the dashed chain of x is called an *x-cluster*. Since we expand the extreme nodes of the dashed chain of u, all vertices on the tree path of any u-cluster are vertices of the dashed chain of u. Thus all faces in the face list of an edge bundle incident to a u-cluster are incident

to u. This implies that if there is an edge bundle between a u-cluster and a v-cluster and the face list of this bundle is not empty, then there exists a face that is incident to u and to v. Hence an edge between u and v can be inserted without destroying the planar embedding. Conversely, if the face list of such an edge bundle is empty, then (u, v) cannot be embedded into any face belonging to the bundle. If there exists a face F in the cluster graph that is incident to a u-cluster and incident to a v-cluster, then the coverage bit of F at the u-cluster and at the v-cluster is one if and only if (u, v) can be inserted in F.

Thus an edge between u and v does not destroy the planar embedding in G if and only if there exists a face in the cluster graph whose coverage bits at the u-cluster and at the v-cluster are one or there exists an edge bundle between a u-cluster and a v-cluster whose face list is nonempty. Since there are $O(\log n)$ faces and edge bundles in the cluster graph and each can be tested in constant time, we can decide whether an edge between u and v can be inserted; and, if so, find such a face, in time $O(\log n)$. Then we merge the topology tree together in the same way we expanded it.

Theorem 4.5.1 *We can determine whether an edge between vertices u and v is consistent with the planar embedding in $O(\log^2 n)$ time.*

4.6 Updates

An update consists of three steps. First we expand the topology tree at a constant number of vertices to create the cluster graph as discussed in Section 3.3. Next we change the targets at the edge bundles to become precise targets. In the resulting graph we insert or delete the appropriate edges or vertices. This step depends on what operation we are executing. The third step is to merge the topology tree back together. We explained already how to perform step one in time $O(\log^2 n)$ in Section 4.4. Depending on the type of update we either add edges (and possibly vertices) in step two or remove them from the cluster graph. The three cases that occur are identical to Sections 3.8.1– 3.8.3 except that a vertex is replaced by a chain rather than a cycle. It is straightforward how to adapt the description to this and how to update

the coverage and the face lists in all three cases. The merging back together of the topology tree is, however, somewhat different and described in the following section.

4.6.1 Merging the Topology Tree

Merging the topology tree has three steps. First, we compute the new topology tree for the updated cluster graph, which takes $O(\log n)$ time. Second, we compute the LCA targets of the edge bundles in the cluster graph based on the new topology tree and we update the data structure appropriately. This step takes time $O(\log^2 n)$. Third we compute a recipe for all the clusters without one. There are $O(\log n)$ such clusters, and we spend amortized time $O(\log n)$ apiece. Thus the update time is $O(\log^2 n)$.

In Section 3.3 we described how the structure of the new topology tree for the updated cluster graph is computed. We need the new topology tree to be able to find the LCA targets in step two. Step one does not create new recipes or edge bundles, but additional clusters are expanded (see Section 3.3). However, the total number of clusters affected is still only $O(\log n)$.

In step two we first compute the precise targets for the newly created edge bundles, which takes time $O(\log^2 n)$. Then we compute the LCA of all the edge bundles in the cluster graph based on the new topology tree. This takes time $O(\log n \log\log\log n)$ on a pointer machine [15]. Adjacent edge bundles with the same target are merged together. This requires removing $O(\log n)$ old edge bundles from the edge bundle list and inserting $O(\log n)$ new bundles. Additionally $O(\log n)$ coverage bits are deleted from the coverage lists and $O(\log n)$ faces are inserted into the face lists. To determine whether a face should be inserted into the face list we check whether it is a good face in time $O(\log n)$ as described above. This takes time $O(\log n)$ per face. Since each operation in an edge bundle or coverage list takes time $O(\log\log n)$ and each operation in a face list takes time $O(\log n)$, the time for updating the data structure after the LCA targeting is $O(\log^2 n)$

In step three we use the concept of a fringe. The node representing each cluster in the cluster graph belongs to the *fringe* in the new topology tree. The important property of the fringe is that no node below it has been expanded during the current

operation and, as we show, the recipes of nodes below the fringe do not have to be modified. Every node above the fringe either does not have a recipe or has an outdated recipe. This implies that for each of the nodes above the fringe a recipe has to be created.

The recipes in clusters below the fringe are not affected by the operation for the following reasons. A location descriptor stored in the recipe of a cluster C that points into an edge bundle list, a coverage list, or a face list is defined relative to the number of edges that are incident to C and to the left of the edge or face indicated by the location descriptor. Since C was not expanded, the edges incident to C did not change. Thus all the location descriptors are still valid. The other information in a recipe (internal edge bundles, extreme edges and coverage bits) are stored independently of how the edges leaving C are grouped into edge bundles. Thus they are not influenced by any changes in the topology tree above C.

The recipes are computed by bottom–up merging: for each cluster we combine the coverage graphs of its sub–clusters, then record how to reverse the operation in a recipe. We now consider the process of merging two clusters. When we start, each cluster of the partially expanded topology tree has one or two edge bundle, coverage, and face lists. The edge bundles are LCA-targeted; at constant cost per edge bundle we label them with the depths of their targets in the topology tree. We compute the edge bundle, coverage and face lists for the nodes by walking up from the fringe to the root of the topology tree merging neighboring clusters.

To merge two clusters we maintain a data structure at each node on or above the fringe to identify the bundles of a given depth at each cluster along a fringe–to–root path. We store an array of length $O(\log n)$, indexed by depth, of the edge bundles incident to the node (cluster). Each array entry contains a pointer to a list of bundles with the same depth. We also record the total number of edge bundles in the array and maintain a list of the indices of non-empty array fields. We initialize the array at each cluster on the fringe in time $O(\log n)$; then we maintain it as we walk toward the root.

When two clusters C and C' at depth i are merged, we check the edge bundles at depth $i+1$ in the array at either cluster. Since the edge bundles are LCA-targeted,

the target of all these edge bundles is the parent of C and C'. Note that these are exactly the edge bundles that connect C and C' and there are at most three of them incident to each cluster. By climbing to the root of the balanced tree in the edge bundle list, we can determine for each edge bundle in time $O(\log\log n)$ to which side of $p(C)$ or $p(C')$ it belongs. With this information we can find a loopy edge (see Case 5, Section 4.4) between C and C' in constant time if one exists. Now the edge bundle, coverage, and face lists of C and C' are merged according to one of the five cases described in Section 4.4. This takes time $O(\log n)$. Afterwards the edges of the cluster with the fewer edges are copied into the array of the other cluster and the index list is updated appropriately. Since there are $O(\log n)$ edge bundles altogether, each edge bundle is copied at most $O(\log\log n)$ times. Thus the total cost for copying during all merges of two clusters is $O(\log n \log\log n)$.

Since the number of edge bundles merged is $O(\log n)$, the cost of merging the topology tree back together is $O(\log^2 n)$.

Theorem 4.6.1 *There exists a data structure for fully-dynamic planarity testing in embedded planar graphs that requires time $O(\log^2 n)$ for any query and for any edge insertion or deletions.*

Chapter 5

Lower Bounds for Fully Dynamic Problems

5.1 Introduction

In this chapter we establish a lower bound on the time per operation for fully-dynamic planarity testing of embedded graphs with n vertices, in the cell probe model with wordsize $O(\log n)$. We show that each operation takes amortized time $\Omega(\log / \log \log n)$. We also describe a similar construction providing lower bounds of $\Omega(\log / \log \log n)$ on the amortized time per operation for fully dynamic k-edge connectivity and fully dynamic k-vertex connectivity in plane graphs for constant k.

In the cell probe model of computation [33], the time complexity of a sequential computation is defined to be the number of accessed memory words. Any other operations are free. This model is powerful enough to express all algorithms for random access machines.

We first review some related previous work. Westbrook and Tarjan [28] gave a lower bound for maintaining the connected components of a graph under a sequence of edge insertions and backtracking edge deletions. They proved that in the separable pointer machine model for any n and any m there exists a sequence of m operations whose cost is $\Omega(m \log n / \log \log n)$.

Fredman and Saks [11] gave a lower bound for the modular prefix sum problem

in the cell probe model of computation. The modular prefix sum problem is defined as follows: Given an array $A[1], \ldots, A[n]$ of integers mod k with initial value zero execute the following operations in arbitrary order:

$Add(l)$: Increase $A[l]$ by 1.

$Sum(l)$: Return the sum of $A[1], \ldots, A[l]$ mod k.

The lower bound shows that each operation takes amortized time $\Omega(\log n/(\log\log n + \log b))$, where b is the number of bits in a memory word. To be precise, given an algorithm in the cell probe model they construct a sequence of m operations such that the algorithm takes time $\Omega(m \log n/(\log\log n + \log b))$. The lower bound holds for arbitrary values of k. We make in the following section use of the lower bound for the case $k = 2$. We call this the *parity prefix sum problem.*

5.2 The Lower Bounds

Fredman [10] observed that the lower bound for the parity prefix sum problem implies a lower bound of $\Omega(\log n/(\log\log n + \log b))$ for maintaining connected components under a sequence of arbitrary insertions and deletions. This reduction uses a plane graph. Thus it gives a lower bound of $\Omega(\log n/(\log\log n + \log b))$ on the amortized time per operation for the problem of maintaining the connected components of a plane graph under arbitrary insertions and deletions of edges. The best known upper bound is $O(\log n)$ [6].

A similar reduction gives a lower bound for fully-dynamic planarity testing in plane graphs. In the following paragraphs we describe this reduction which yields a lower bound of $\Omega(\log n/(\log\log n + \log b))$ on the amortized time per operation for fully-dynamic planarity testing in the cell probe model of computation. Afterwards we give lower bounds for fully-dynamic k-edge and k-vertex connectivity.

We are given an instance of the parity prefix sum problem and we want to solve it using an algorithm for fully-dynamic planarity testing. Let S_i be $\sum_{j=1}^{i} A[j]$. To represent A we construct the following graph with $n + 3$ vertices. The vertices are labeled with consecutive numbers starting with 0. Vertex i represents S_i for $1 \leq i \leq n$.

Vertices 0, $n+1$, and $n+2$ form a triangle whose face is not incident to any of the other vertices. All vertices i with odd S_i are connected by a chain in the following way. If S_l is odd and i is the largest index smaller than l such that S_i is odd, there is an edge between vertex i and vertex l. All vertices with even S_i are connected in the same way. Additionally the first vertex f_{even} of the even chain and the first vertex f_{odd} of the odd chain are connected by an edge to vertex 0. The clockwise order of the edges at vertex 0 is as follows: $(0, n+2)$, $(0, n+1)$, $(0, f_{even})$, $(0, f_{odd})$. The last vertex l_{even} of the even chain and the last vertex l_{odd} of the odd chain are connected to vertex $n+1$ by an edge. The clockwise order at vertex $n+1$ is $(n+1, l_{odd})$, $(n+1, l_{even})$, $(n+1, 0)$, and $(n+1, n+2)$. Additionally there is an edge between vertex l_{odd} and vertex $n+2$ (see Figure 30).

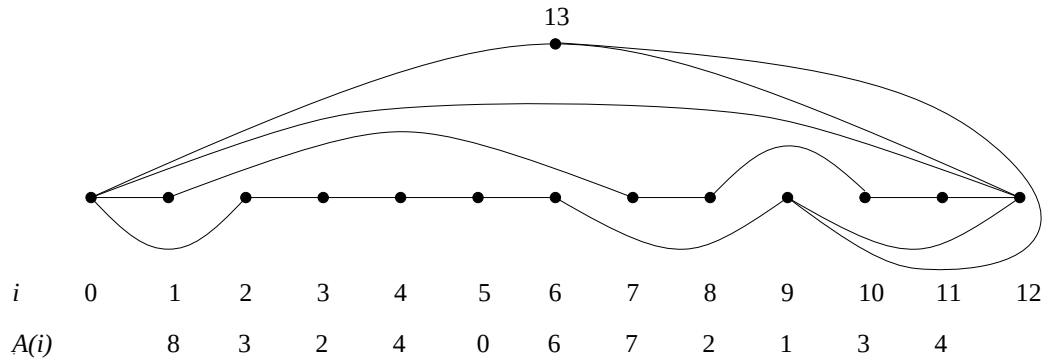

Figure 30: The plane graph for the array 8 3 2 4 0 6 7 2 1 3 4.

The odd chain, along with the edges $(f_{odd}, 0)$, $(0, n+1)$, and $(n+1, l_{odd})$ forms a cycle of edges. All vertices on the even chain are inside this cycle, but vertex $n+2$ is outside. Thus adding an edge between a vertex on the even chain and vertex $n+2$ destroys the planarity of the embedding while adding an edge between an vertex on the odd chain and vertex $n+2$ preserves planarity. Hence, an edge between vertex l and vertex $n+2$ can be added to the graph if and only if S_l is odd. This implies that a prefix sum mod 2 query $sum(l)$ can be answered by testing whether an edge between vertex $n+2$ and vertex l preserves the planarity of the graph.

Each vertex, except vertex 0 and vertex $n+1$, has degree 2. Thus there is only one order of the edges incident to these vertices. If we guarantee that the graph is planar, then this order will determine a planar embedding of the two chains. During

insertions or deletions of edges we adapt the order at vertex $n+1$ to guarantee that the rest of the graph also maintains the planar embedding.

To speed up updates we keep pointers to f_{even}, f_{odd}, l_{even}, and l_{odd}. Additionally we build Van Emde Boas' data structure [29] (called *Van-Emde-Boas data structure*) for finding predecessors in integer sets. Given the universe $\{0, 1, \ldots, n\}$ and a subset S of the universe this data structure allows us to execute the following operation in arbitrary order:

$Insert(x)$: Insert x into S, if $x \notin S$.

$Delete(x)$: Delete x from S, if $x \in S$.

$Pred(x)$: Find the largest $y \in S$ such that $y \leq x$, or indicate that no such element exists.

$Succ(x)$: Find the smallest $y \in S$ such that $y \geq x$, or indicate that no such element exists.

Each of these operations can be executed in time $O(\log\log n)$ in the cell probe model [29]. In the following we denote by $pred(x)$ (resp. $succ(x)$) the vertex returned by the call $pred(x)$ (resp. $succ(x)$) in the Van-Emde-Boas data structure. Note that $x \in S$ implies that $succ(x) = pred(x) = x$.

Let $i_1 \leq i_2 \leq \ldots \leq i_l$ be indices between 1 and n such that

1. S_{i_j-1} is even and S_{i_j} is odd or
2. S_{i_j-1} is odd and S_{i_j} is even.

We store the indices $i_1, \ldots, i_j$ and the value 1 and $n+1$ in a Van-Emde-Boas data structure. The value 1 is inserted to guarantee that $pred(l) - 1 \geq 0$ for all l. The value $n+1$ is inserted to guarantee that $succ(l+1) \leq n+1$.

We are left with showing how the graph is updated during an $add(l)$ operation. Note that this operation causes S_l to change from odd to even or vice versa. Thus the chain of odd vertices has to be cut into two pieces and so does the chain of even vertices. With one *pred* query and one *succ* query in the Van-Emde-Boas data structure we determine the two edges which have to be deleted. Then we insert two

edges to correctly connect the four subchains again and we update the embedding at vertex $n+1$.

Let us now describe the reduction in more detail. To execute an $add(l)$ operation we first have to find the two edges to be cut. Without loss of generality let S_l be odd before the $add(l)$ operation. There is exactly one edge which connects l to a vertex g with $g \leq l$. This is the edge of the odd chain which has to be deleted. The second edge that has to be deleted is the edge which connects a vertex of the even chain smaller than l with a vertex of the even chain larger than l. The vertex of the even chain with largest index smaller than l is $pred(l)-1$, the one with smallest index larger than l is $succ(l+1)$. Thus the second edge to be deleted is the edge $(pred(l)-1, succ(l+1))$ which can be found with two queries in the Van-Emde-Boas data structure. If we execute an $add(4)$ operation in the example of Figure 30, then $(3,4)$ and $(1,7)$ are the edges to be deleted.

Since now S_l is even and $S_{succ(l+1)}$ is odd, we connect vertex $pred(l)-1$ with vertex l by an edge as well as vertex g with vertex $succ(l+1)$. In the example of Figure 30 the edges $(3,7)$ and $(1,4)$ have to be added. See the result in Figure 31.

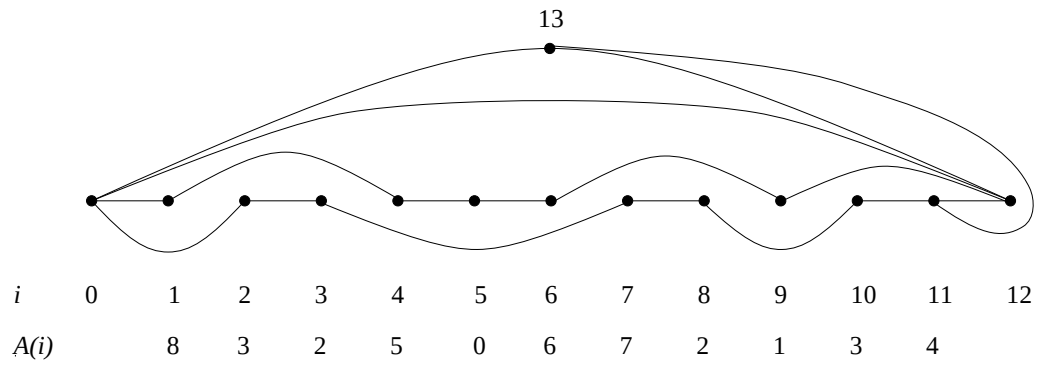

Figure 31: The plane graph of the array 8 3 2 5 0 6 7 2 1 3 4.

Vertex l_{even} is now the end of the odd chain and vertex l_{odd} is now the end of the even chain. Therefore we change the order of the edges at vertex $n+1$ by first deleting the edge $(n+1, l_{odd})$ (which was embedded before edge $(n+1, l_{even})$ in clockwise order) and by then inserting it again after edge $(n+1, l_{even})$. This preserves the planarity of the embedding. Next we swap the pointers of l_{even} and l_{odd} and update the edge

between vertex $n+2$ and l_{odd} appropriatelly. Additionally we insert l into the Van-Emde-Boas data structure if $pred(l) \neq l$ and we remove it if $pred(l) = l$ (except if $l = 1$).

Note that an $add(l)$ operation can be executed in a way that preserves the planar embedding and does not disconnect the graph by executing the insertions and deletions in the following order: Delete the edge (g, l) and $(l_{odd}, n+2)$, insert the edge $(pred(l)-1, l)$ (such that it is consistent with the embedding), delete the edge $(l_{odd}, n+1)$ and the edge $(pred(l)-1, succ(l+1))$, insert the edge $(g, succ(l+1))$, and finally insert the edge $(l_{odd}, n+1)$ and $(l_{odd}, n+2)$ (in the right order of the embedding at the vertices l_{odd}, $n+1$, and $n+2$).

This shows that for updating the data structure after an $add(l)$ operation a constant number of operations in the Van-Emde-Boas data structure and a constant number of insertion and deletions in the graph are executed.

Fredman and Saks [11] show that for this algorithm for the parity prefix sum problem a sequence of m operations can be constructed such that the algorithm takes time $\Omega(m \log n/(\log\log n + \log b))$ in the cell probe model. This sequence uses $O(m)$ edge insertions, edge deletions, and planarity queries in the plane graph and $O(m)$ operations in the Van-Emde-Boas data structure. Since each operation in the Van-Emde-Boas data structure takes time $O(\log\log n)$ in the cell probe model, it follows that the sequence of $O(m)$ edge insertions, edge deletions, and planarity queries takes time $\Omega(m \log n/(\log\log n + \log b))$. This establishes the following bound.

Theorem 5.2.1 *Any fully-dynamic planarity testing algorithm for an embedded graph requires* $\Omega(\log n/(\log\log n + \log b))$ *amortized time per operation in the cell probe model.*

The following theorems hold for fully dynamic connectivity.

Theorem 5.2.2 *In the cell probe model any algorithm for maintaining k-edge connectivity (for constant k) in a plane graph under a sequence of insertions and deletions of edges requires* $\Omega(\log n/(\log\log n + \log b))$ *amortized time per operation.*

Proof: Let S_0 be one. We construct the following graph. There are $n+2$ vertices labeled from 0 to $n+1$. Vertex i represents S_i. Additionally for each

$i < n+1$ there are k vertices labeled $(i+1/2,1),(i+1/2,2),\ldots(i+1/2,k)$. Each vertex $(i+1/2,j)$ has an edge to vertex i. If S_l is odd and i is the largest index smaller than l such that S_i is odd, there is an edge between every vertex $(i+1/2,j)$ and l. The vertices with even S_l are connected in the same way. Let f_{odd} be the last vertex with integer label of the odd chain and let f_{even} be the last vertex with integer label of the even chain. If $k > 1$, there is an edge from the vertex $(f_{odd},1)$ to vertex $n+1$ and from vertex $(f_{even},1)$ to vertex $n+1$. See Figure 32. We maintain the same indices in a Van-Emde-Boas data structure as in the previous proof.

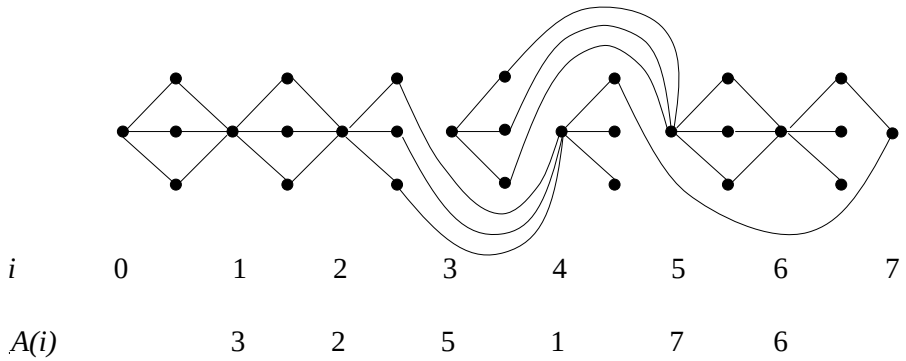

Figure 32: The plane graph constructed for the array 3 2 5 1 7 6 to show a lower bound for 3-edge connectivity.

Vertex l and vertex 0 are k-edge connected if and only if S_l is odd. Thus a $sum(l)$ query corresponds to one connectivity query in the graph. Each $add(l)$ operation corresponds to a constant number of operations in the Van-Emde-Boas data structure and k insertions and deletions of edges in the graph. If $k > 1$, the insertions and deletions of edges can be executed so that the graph is never disconnected and that the planar embedding is always preserved.

This gives an algorithm for the parity prefix sum problem. As shown in [11] there exists a sequence of m operations that requires time $\Omega(m \log n/(\log\log n + \log b))$. This sequence can be handled with $O(m)$ operations in the fully-dynamic k-edge connectivity data structure and $O(m)$ operations in the Van-Emde-Boas data structure. Since each operation in the Van-Emde-Boas data

structure takes time $O(\log\log n)$, only $O(m \log\log n)$ time is spent execute these operations. It follows that the sequence of $O(m)$ operations in the fully-dynamic k-edge connectivity data structure takes time $\Omega(m \log n/(\log\log n + \log b))$. ∎

Theorem 5.2.3 *In the cell probe model any algorithm for maintaining k-vertex connectivity (for constant k) in a plane graph under a sequence of insertions and deletions of edges requires $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation.*

Proof: Let S_0 be one. We use the following graph. There are k vertices $(i,1),(i,2),\ldots,(i,k)$ to represent S_i and one vertex labeled 0 and one vertex labeled $n+1$. There is an edge between (i,j) and $(i,j+1)$ for $1 \leq j < k$. If S_l is odd and i is the largest index smaller than l such that S_i is odd, there is an edge between (i,j) and (l,j) for all j. All vertices with even S_l are connected in the same way. Thus k chains of odd vertices and k chains of even vertices are created. Note that this gives a plane graph. Let f_{odd} be the last vertex of the odd chain and let f_{even} be the last vertex of the even chain. There is an edge from the vertex $(f_{odd},1)$ to vertex $n+1$ and from vertex $(f_{even},1)$ to vertex $n+1$. See Figure 33.

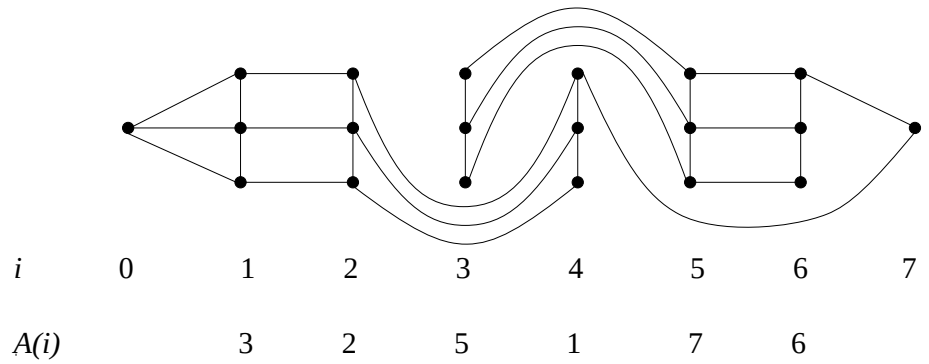

Figure 33: The plane graph constructed for the array 3 2 5 1 7 6 to show a lower bound for 3-vertex connectivity.

In the same way as in the previous proof each $add(l)$ operation causes $O(k)$ insertions and deletions of edges. To answer a $sum(l)$ query, every edge connecting (l,j) to (l',j) for every j with $l' > l$ is deleted, an edge from vertex 0

to vertex $(l', 1)$ is inserted, and an edge from every vertex (l, j) for $j > 1$ to vertex $(l, 1)$ is inserted. This is necessary to preserve a planar embedding. Now $sum(l)$ is 1 if and only if vertex $(l, 1)$ and vertex 0 are k-vertex connected. Afterwards the original graph is restored. Thus each $sum(l)$ query requires $O(k)$ edge insertions and deletions and one connectivity query in the graph. Each operation in the parity prefix sum problem can be execute in a way that does not disconnect the graph and preserves the planarity of the embedding.

This gives an algorithm for the parity prefix sum problem. As shown in [11] there exists a sequence of operations for this algorithm that takes time $\Omega(m \log n/(\log\log n + \log b))$. This sequence generates a sequence of $O(m)$ operations in the Van-Emde-Boas data structure and $O(m)$ operations in the fully-dynamic k-vertex connectivity data structure. Since each operation in the Van-Emde-Boas data structure takes time $O(\log\log n)$, it follows that the $O(m)$ operations in the fully-dynamic k-vertex connectivity data structure take time $\Omega(m \log n/(\log\log n + \log b))$. ∎

Chapter 6

Finding Predecessors in Integer Sets

6.1 Introduction

Let U be the universe $\{0, 1, \ldots, u-1\}$. Starting with the empty set, we want to store a set $S \subseteq U$, so that the following operations on S can be executed efficiently:

Insert(x): Insert x into S, if $x \notin S$.

Delete(x): Delete x from S, if $x \in S$.

Pred(x): Find the largest $y \in S$ such that $y \leq x$, or indicate that no such y exists.

Van Emde Boas [29] gave an algorithm with $O(\log \log u)$ worst-case time per operation and $O(u)$ space. Using dynamic perfect hashing [5] a probabilistic algorithm can be found with expected $O(\log \log u)$ time per operation and $O(n)$ space, where n is the current size of S. The same time and space bounds are achieved by a deterministic algorithm of Willard [31]. The data structure in his algorithm is called the *Van-Emde-Boas data structure*.

Ajtai, Fredman and Komlós [4] gave a constant-time solution in Yao's cell probe model [33] for $n = (\log u)^c$ for any constant c. Each set of size at most $(\log u)^c$ is

represented by an identifier of $O(\log u)$ bits and a list of set elements in such a way that all operations can be executed in constant time in Yao's cell probe model. We use their result and refer to this representation of a set as the *AFK-encoding* of the set. To implement their algorithm on a RAM with word size $O(\log u)$, they precompute all possible operations on S in time $O(u^\epsilon)$ for an arbitrarily small constant ϵ and store the results in a lookup table T of size $O(u^\epsilon)$. To execute an operation the result is looked up in the table in constant time.
Ajtai [2] proved a lower bound in Yao's cell probe model of $\Omega(\log\log u)$ on the query time of any algorithm that works for arbitrary size n and uses space polynomial in n. He showed that there exists a distribution of the input sets S such that each algorithm that starts with a random S and uses space polynomial in n needs expected time $\Omega(\log\log u)$ to answer a query in Yao's cell probe model. Together with Yao's minmax principle [32] this shows that any algorithm, deterministic or randomized, with space polynomial in n cannot achieve a faster query time.

Therefore, we restrict ourselves to an average case analysis of a deterministic algorithm, assuming a uniform distribution on all sets S. We achieve an amortized expected time of $O(1)$ per operation in Yao's cell probe model with $O(n)$ space for arbitrary set size n. To be more precise, the expected time to answer the hardest query is $O(1)$ and the expected time for m updates is $O(m)$. We need $O(u^\epsilon)$ preprocessing time to implement our algorithm on a RAM with word size $O(\log u)$. The worst-case running time of an operation is $O(\log\log u)$.

The basic idea behind this result is the following: Assume that U is split into n contiguous, equal-size pieces called *groups*. Choose n elements at random uniformly from U and put each into the group to which it belongs. Then it is very unlikely that more than $\log n$ elements are in the same group and thus each group can be stored using the above mentioned result by Ajtai, Fredman and Komlòs [4].

In Section 6.2 we describe a simple algorithm whose expected running time for the hardest query is constant. The expected running time of m insertions is $O(m)$ and of m deletions is $O(m\log u)$. In Section 6.3 we improve the expected running time of m deletions to $O(m)$. Both algorithms need $O(u^\epsilon)$ preprocessing time.

6.2 A Simple Solution

To answer predecessor queries in a set S, the algorithm maintains a periodically-rebuilt data structure. Let k be the size of S during the last reconstruction and let n be the current size of S. At the beginning of the algorithm $k = n = 0$. The algorithm maintains an array a of indices (used as pointers) with k entries between 0 and $k-1$. Each entry points to what we call a *bucket pointer*.

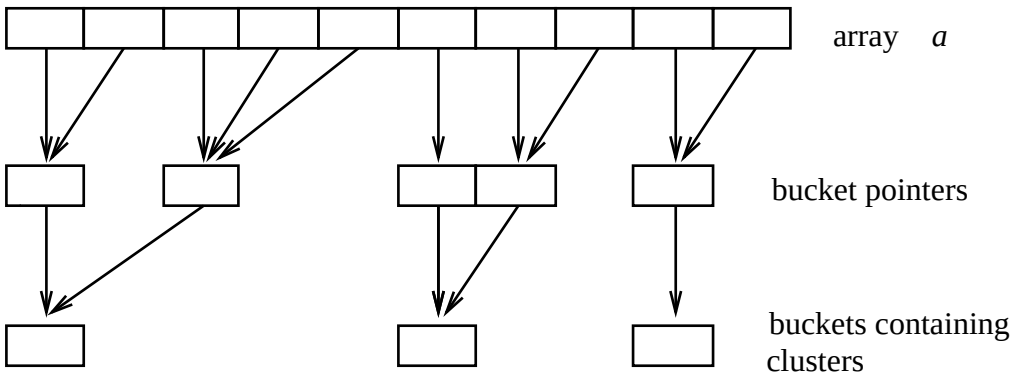

Figure 34: An array a with bucket pointers and buckets.

An array entry points to a bucket pointer, and a bucket pointer points to a data structure called *bucket*. All array entries that point to the same bucket pointer are contiguous, and all bucket pointers that point to the same bucket are contiguous. See Figure 34. Each bucket contains the AFK-encoding of a set of size $O(\log u)$. This set is called a *cluster*. Bucket pointers are needed to make delete operations faster. Without deletions the array entries could point directly to the buckets. The buckets are connected by a doubly-linked list, i.e. they have a predecessor pointer *left* and successor pointer *right*. Additionally we store at each bucket the number of bucket pointers pointing to it.

Let the *range of* $a[i]$ (or short $range(i)$) be the range of items $i\lceil\frac{u}{k}\rceil, \ldots, (i+1)\lceil\frac{u}{k}\rceil - 1$. To answer a $pred(x)$ operation, we first look up the entry $a[\lfloor x/\lceil\frac{u}{k}\rceil\rfloor]$, i.e. $a[i]$ can be accessed by the items from $range(i)$, for $0 \leq i \leq k$. The bucket pointer of $a[i]$ points to a bucket that contains all the elements of $range(i) \cap S$ (and possibly more elements). Let b_i be the bucket that is reachable from $a[i]$. In the following discussion we assume that $i = \lfloor x/\lceil\frac{u}{k}\rceil\rfloor$.

The operation $pred(x)$ determines the cluster C stored in b_i and looks up the answer for C and x in the lookup table T. If x is smaller than any element in C, it looks up the answer for x and $left(b_i)$.

An $insert(x)$ operation also determines the cluster C stored in b_i. It uses C and x to address a field in T and gets from T the AFK-encoding of the cluster C' that results from inserting x into C. It then stores C' in b_i. If, however, the number of elements in a cluster is larger than $2f \log u$ for a constant f that we will choose later, we build a Van-Emde-Boas data structure for the elements in the cluster and store a pointer to this data structure in b_i. This takes time $O(\log u)$. All further operations use the Van-Emde-Boas data structure until the size of the cluster is reduced to $f \log u$. Then we encode the cluster again (in time $O(\log u)$) and use the lookup table for further operations. Whenever we rebuild the data structure we store every cluster of size at most $3/2f \log u$ in an AFK-encoding and every larger cluster in a Van-Emde-Boas data structure. Since $O(\log u)$ update operations are necessary to force a change of representation, the cost of the change can be amortized over these operations, adding only a constant factor to their running time.

A $delete(x)$ operation is similar to an insertion. It behaves differently only when the new cluster C' is empty. In this case, if the number of bucket pointers pointing to $left(b_i)$ is larger than the number pointing to b_i, the algorithm makes all pointers to b_i point to $left(b_i)$ and deletes b_i. Otherwise it makes all the pointers of $left(b_i)$ point to b_i. It then stores the content of $left(b_i)$ in b_i and deletes $left(b_i)$.

We rebuild the data structure for the first time whenever the size of S reaches four. After this, we rebuild it whenever the number of operations since the last reconstruction is equal to the number of elements in S at the last reconstruction. This guarantees that the size of S during a reconstruction is linear in the number of operations that have occurred since the previous reconstruction. We will show that the cost of a reconstruction is proportional to the size of S and thus can be amortized over the operations between the two reconstructions.

Let k_1 be the number of elements stored in S at the previous reconstruction and let k_2 be the size of S at the current reconstruction. Since there were at most k_1

operations since the last reconstruction, $k_2 \leq 2k_1$. To execute the current reconstruction we construct a new array a with k_2 entries. Each $a[i]$ points to a bucket pointer whose bucket contains exactly the elements of $S \cap range(i)$, if at least one such element exists. Otherwise $a[i]$ points to the bucket pointer of $a[i']$, where i' is the largest index smaller than i such that $S \cap range(i') \neq \emptyset$. This takes time linear in the size of S.

If the set size is smaller than $\frac{3}{2} f \log u$, the buckets contain the AFK-encoding of their set, otherwise, each bucket contains the Van-Emde-Boas data structure for the set. To build either representation takes time linear in the size of the set. Thus the cost of the whole rebuilding is proportional to k_2, the size of S during the second reconstruction, and the cost can be amortized over the operations between the two reconstructions, increasing their running time by only a constant factor.

If k does not divide u (i.e. the range of the last bucket is smaller than $\lceil \frac{u}{k} \rceil$), we insert the element u into the last bucket for ease of the analysis. This does not change the correctness of the algorithm.

Notice that after a reconstruction each bucket is addressed by exactly one bucket pointer, while a bucket pointer can be addressed by many entries of a. Deletions are the only operations that change bucket pointers. The deletions execute a union-by-weight algorithm on the bucket pointers. Assume that between two reconstructions there are d deletions that change bucket pointers. Then $d = O(u)$, since each deletion reduces the number of buckets by at least one and there are at most u buckets after a reconstruction. After d deletions at most d of the bucket pointers point to the same bucket. Every time a bucket pointer is changed, the number of bucket pointers that point to its bucket at least doubles. Therefore each bucket pointer can be changed at most $O(\log d)$ times and the total running time of the d deletions is $O(d \log d)$. This gives an amortized cost of $O(\log d) = O(\log u)$ per deletion. In the next section we will see how this can be reduced to $O(1)$.

The algorithm is similar to hashing using the hash function $h(x) = \lfloor x / \lceil u/k \rceil \rfloor$ It is different, however, in that the first elements that would be put into an empty bucket using the above hash function have to wait until the next reconstruction to get a new bucket. Until then they are put into another bucket, which increases the number

of elements in a bucket slightly, but keeps us from having to update predecessor information between buckets. For the remainder of the section we prove bounds on the number of elements in the buckets at any point in the sequence. This is important to show that we do not use the more costly Van-Emde-Boas data structure too often.

Let F be a sequence of insertions and deletions such that each element of $U \setminus S$ is equally likely to be inserted and any element of S is equally likely to be deleted. We want to prove an average cost of $O(t)$ for such a sequence F of length t. The algorithm starts with an empty set S and executes the operations of F. Let $b_i(F,t)$ be the bucket b_i after the tth operation of the sequence F and let $X_i(F,t)$ be the size of the cluster in $b_i(F,t)$. If the operation t is followed by a reconstruction, $X_i(F,t)$ is the size of the cluster in $b_i(F,t)$ after the reconstruction. Note that b_i can be indicated by the bucket pointers of the contiguous set $a[i],\ldots,a[i']$ of entries of a, for some $i' > i$. In this case, only $X_i(F,t)$, the value for the leftmost entry, stores the size of the cluster in $b_i(F,t)$ and all the other values $X_j(F,t)$ for $i < j \leq i'$ are zero. By doing this, we make sure to count every element exactly once. The value $X_i(F,t)$ is a random variable depending on F. We want to show that, for any fixed i, the expected value of $X_i(F,t)$ is constant and that the expected value of $\max_i X_i(F,t)$ is $O(\log u)$. To prove this the following technical lemma will be useful.

Lemma 6.2.1

$$\frac{\binom{u-j\lceil\frac{u}{k}\rceil}{n}}{\binom{u}{n}} \leq e^{-j}$$

for all positive integers u, j, k, n with $j\lceil\frac{u}{k}\rceil \leq u-n+1$ and $n \geq k$.

Proof: Let $b \leq u-n+1$. Then

$$\frac{(u-b)!(u-n)!}{u!(u-n-b)!} = (1-\frac{b}{u}) \times \cdots \times (1-\frac{b}{u-n+1}) \leq (1-\frac{b}{u})^n.$$

Setting $b = j\lceil\frac{u}{k}\rceil$ gives

$$\frac{\binom{u-j\lceil\frac{u}{k}\rceil}{n}}{\binom{u}{n}} = \frac{(u-j\lceil\frac{u}{k}\rceil)!(u-n)!}{(u-j\lceil\frac{u}{k}\rceil-n)!u!} \leq (1-\frac{j\lceil\frac{u}{k}\rceil}{u})^n \leq (1-\frac{j}{k})^k \leq e^{-j}$$

as claimed in the lemma. ∎

Now we want to show that it is very unlikely to have many elements in a bucket.

Lemma 6.2.2 *There exists a constant* $0 < c_0 < 1$ *such that*

$$P(X_i(F,t) \geq c) \leq 2^{\log 3 - c_0 c} \text{ for } c \geq 13.$$

Proof: Let us consider some arbitrary but fixed sequence F of t insertions and deletions and let the last reconstruction be executed after operation t_1 for $t_1 \leq t$. Let d be the number of elements that are inserted before operation $t_1 + 1$ and deleted after operation t_1 and let $k + d$ be the size of S during the last reconstruction. From the timing of the reconstructions we know that $t - t_1 \leq k + d$. For $x \in U$ and for a $delete(x)$ operation we call the closest $insert(x)$ operation that precedes the $delete(x)$ operation, the *insertion that corresponds to the deletion.*

Now consider the sequence $\tilde{F}$, which is created from F by removing every deletion and its corresponding insertion. This leaves exactly the insertions with no corresponding deletions. Assume further that the reconstructions in $\tilde{F}$ happen between the same two operations as in F. Thus the last reconstruction of $\tilde{F}$ is after the kth operation and there are at most $t - t_1 - d \leq k$ insertions after the last reconstruction. Thus $\tilde{F}$ consists of n operations with $n \leq k + t - t_1 - d$ and $n - k \leq k$.

To prove the lemma we show that $P(X_i(\tilde{F}, n) \geq c) \leq 2^{-c/42}$ for $c \geq 13$. This is sufficient for the following reasons. Let the *scope* of a bucket be the set of elements which, if inserted, would be put into this bucket. If $b_{i-1} \neq b_i = b_{i+1} = \ldots = b_{i+j} \neq b_{i+j+1}$, then the scope of b_i is $\bigcup_{l=0}^{j} range(i + l)$. We show that the scope of each bucket for F is contained in the scopes of at most three consecutive buckets of $\tilde{F}$. This then gives a bound for $P(X_i(F,t) \geq c)$.

Let $x_{\min}$ (resp. x_{max}) be the smallest (resp. largest) element in the scope of $b_i(F,t)$. If the size of the scope of $b_i(F,t)$ is equal to the size of $range(i)$ for F, namely $\lceil \frac{u}{k+d} \rceil$, then the scope of $b_i(F,t)$ is contained in the scopes of at most two buckets of $\tilde{F}$, since the size of a scope of $\tilde{F}$ is at least $\lceil \frac{u}{k} \rceil \geq \lceil \frac{u}{k+d} \rceil$.

If the size of the scope of $b_i(F,t)$ is larger than $\lceil \frac{u}{k+d} \rceil$, note that no element larger than $x_{min} + \lceil \frac{u}{k+d} \rceil - 1$ is in $b_i(F,t_1)$ (otherwise a new bucket for this element would have been created during the reconstruction at time t_1). Assume now that the scope of $b_i(F,t)$ is contained in the scopes of more than three buckets of $\tilde{F}$. Then all the elements of (at least) two of the buckets are larger than $x_{\min}$ and smaller than $x_{\max}$ and both buckets contain at least one element that was inserted before operation $t_1 + 1$. Let y be this element in the second such bucket. It follows that $y \in b_i(F,t_1)$ and $y - x_{min} \geq \lceil \frac{u}{k} \rceil \geq \lceil \frac{u}{k+d} \rceil$. But this is a contradiction to the fact mentioned above that no element in $b_i(F,t_1)$ is larger than $x_{min} + \lceil \frac{u}{k+d} \rceil - 1$. This shows that the scope of each bucket for F is contained in the scopes of at most three buckets of $\tilde{F}$.

Thus for each $0 \leq i \leq k+d$ there exists an integer $0 \leq j \leq k$ such that

$$P(X_i(F,t) \geq c) \leq P(X_j(\tilde{F},n) \geq c/3) + P(X_{j+1}(\tilde{F},n) \geq c/3) + P(X_{j+2}(\tilde{F},n) \geq c/3) \leq 2^{\log 3 - c/126}.$$

This shows that a bound on $P(X_j(\tilde{F},n) \geq c)$ gives the desired bound on $P(X_i(F,t) \geq c)$. Hence it is enough to consider a sequence of n insertions whose last reconstruction was at time k with $g = n - k \leq k$ operations after this reconstruction. Since the rest of the proof always refers to the sequence $\tilde{F}$, we abbreviate $X_i(\tilde{F},n)$ by $X_i(n)$ in the rest of this section.

To obtain a bound on the probabilities for $X_i(n)$ we consider the situation that b_i contains l elements during the last reconstruction at time k for each possible l and that $b_{i-1} \neq b_i = b_{i+1} = \ldots = b_{i+j} \neq b_{i+j+1}$, for each possible j. Expressed in a different way this means that $X_i(k) = l$ for $0 \leq l < k$ and $X_{i+1}(k) = 0, \ldots, X_{i+j}(k) = 0$, $X_{i+j+1}(k) > 0$ for $0 \leq j \leq k - i - 1$. Notice that $X_{k-1}(k) > 0$ if k does not divide u. This is important, since otherwise the probability that bucket $k-1$ is empty after a reconstruction would be larger than the probability p that any other bucket is empty after a reconstruction (since the range of bucket $k-1$ is smaller than the range of the other buckets). By inserting element u into the data structure after the reconstruction, we set

the probability that bucket $k-1$ is empty to zero, which can be bounded above by p. Thus bucket $k-1$ can be treated in the following computations like any other bucket.

To bound $P(X_i(n) \geq c)$ we split the event $(X_i(n) \geq c)$ into a set of disjoint events $(X_i(k) = l, X_{i+1}(k) = 0, \ldots, X_{i+j+1}(k) > 0, X_i(n) \geq c)$ for $0 \leq l < k$ and $0 \leq j \leq k-i-1$.

$$\begin{aligned} P(X_i(n) \geq c) &= \sum_{j=0}^{k-i-1} \sum_{l=0}^{k-1} P\Big(X_i(k) = l, X_{i+1}(k) = 0, \ldots, X_{i+j+1}(k) > 0, \\ &\qquad\qquad X_i(n) \geq c\Big) \\ &\leq P(X_i(k) > c/4) + P(j > c/25) + A, \end{aligned}$$

where $(j > c/25)$ describes the event that $X_{i+1}(k) = \ldots = X_{i+c/25+1}(k) = 0$ and

$$A = \sum_{j=0}^{\lfloor c/25 \rfloor} \sum_{l=0}^{\lfloor c/4 \rfloor} P\Big(X_i(k) = l, X_{i+1}(k) = 0, \ldots, X_{i+j+1}(k) > 0, X_i(n) \geq c\Big).$$

Since the probability that there is an insertion into bucket i is at most $\frac{1}{k}$, it follows that

$$P(X_i(k) > c/4) \leq \binom{k}{\lfloor c/4 \rfloor + 1} \Big(\frac{1}{k}\Big)^{\lfloor c/4 \rfloor + 1} \leq e^{(\lfloor c/4 \rfloor + 1) - (\lfloor c/4 \rfloor + 1)\ln(\lfloor c/4 \rfloor + 1)}.$$

Using Lemma 6.2.1 gives

$$P(j > c/25) \leq \frac{\binom{u - (\lfloor c/25 \rfloor + 1)\lceil \frac{u}{k} \rceil}{k}}{\binom{u}{k}} \leq e^{-(\lfloor c/25 \rfloor + 1)}.$$

Using for A the definition $P(B \cap C) = P(B)\, P(C|B)$ with $C = (X_i(n) \geq c)$ and $B = (X_i(k) = l, X_{i+1}(k) = 0, \ldots, X_{i+j}(k) = 0, X_{i+j+1}(k) > 0)$ and plugging in values for the probabilities gives

$$A \leq \sum_{j=0}^{\lfloor c/25 \rfloor} \sum_{l=0}^{\lfloor c/4 \rfloor} \frac{\binom{\lceil \frac{u}{k} \rceil}{l} \binom{u - (j+1)\lceil \frac{u}{k} \rceil}{k-l}}{\binom{u}{k}} \times \frac{\binom{(j+1)\lceil \frac{u}{k} \rceil - l}{c-l} \binom{u-k-(c-l)}{g-(c-l)}}{\binom{u-k}{g}}.$$

Now we apply

$$\frac{\binom{\lceil \frac{u}{k} \rceil}{l}\binom{u-(j+1)\lceil \frac{u}{k} \rceil}{k-l}}{\binom{u}{k}} \leq \frac{\binom{u-j\lceil \frac{u}{k} \rceil}{k}}{\binom{u}{k}} \leq e^{-j}.$$

The latter inequality follows from Lemma 6.2.1. We know that $g \leq k$ and $g \leq u-k$.

If $k < u/3$, then

$$\begin{aligned}
\frac{\binom{(j+1)\lceil \frac{u}{k} \rceil - l}{c-l}\binom{u-k-(c-l)}{g-(c-l)}}{\binom{u-k}{g}} &\leq \frac{\binom{(j+1)(\frac{u}{k}+1)-l}{c-l}\binom{g}{c-l}}{\binom{u-k}{c-l}} \\
&\leq \Big(\frac{2(j+1)}{k}\Big)^{c-l}\binom{g}{c-l} \\
&\leq \Big(\frac{2e(j+1)}{c-l}\Big)^{c-l}.
\end{aligned}$$

If $k \geq u/3$, then

$$\frac{\binom{(j+1)\lceil \frac{u}{k} \rceil - l}{c-l}\binom{u-k-(c-l)}{g-(c-l)}}{\binom{u-k}{g}} \leq \binom{3(j+1)-l}{c-l} \leq \Big(\frac{3e(j+1)}{c-l}\Big)^{c-l}.$$

It follows that

$$\begin{aligned}
A &\leq \sum_{j=0}^{\lfloor c/25 \rfloor}\sum_{l=0}^{\lfloor c/4 \rfloor} e^{-j}\Big(\frac{3e(j+1)}{c-l}\Big)^{c-l} \\
&\leq \sum_{j=1}^{\lfloor c/25 \rfloor}\sum_{l=0}^{\lfloor c/4 \rfloor} 2^{-j\log e+(c-l)\log(3e(j+1))-(c-l)\log(c-l)} \\
&\leq 2^{-c/6} \text{ for } c \geq 13.
\end{aligned}$$

Altogether this gives

$$P(X_i(n) \geq c) \leq e^{(\lfloor c/4 \rfloor+1)-(\lfloor c/4 \rfloor+1)\ln(\lfloor c/4 \rfloor+1)} + e^{-(\lfloor c/25 \rfloor+1)} + 2^{-c/6} \leq 2^{-c/42}$$

for $c \geq 13$. Thus it follows that $c_0 = 1/126$. ∎

Now it is easy to prove that the expected bucket size is constant and that the expected size of $\max_i X_i(F,t)$ is $O(\log u)$. Let $\tilde{X}(F,t) = \max_i X_i(F,t)$.

Lemma 6.2.3 *$E[X_i(F,t)]$ is bounded by a constant for all i and t and $E[\tilde{X}(F,t)]$ is $O(\log u)$ for all t.*

Proof: From Lemma 6.2.2 it follows that

$$E[X_i(F,t)] = \sum_{c=0}^{s} P(X_i(F,t) \geq c) \leq 12 + \sum_{c=13}^{s} 2^{\log 3 - c/126} \leq 15.1 \ .$$

The definition of $\tilde{X}$ shows that $P(\tilde{X}(F,t) \geq c) \leq \min(1, k\ P(X_i(F,t) \geq c))$, where k is the number of array entries. Therefore,

$$E[\tilde{X}(F,t)] = \sum_{c=0}^{s} P(\tilde{X}(F,t) \geq c) = 12 + \sum_{c=12}^{s} \min(1, 2^{\log k + \log 3 - c_0\, c}).$$

Let $c_1 = \min\{c : 2^{\log k + \log 3 - c_0\, c} \leq 1\}$. We have $c_1 = O(\log k) = O(\log u)$; thus

$$E[\tilde{X}(F,t)] \leq 12 + \sum_{c=12}^{c_1} 1 + \sum_{c > c_1} \left(\frac{1}{2}\right)^{c - c_1} = O(\log u).$$

This proves the lemma. ∎

Each operation spends constant time to find the cluster that must be accessed. Then it manipulates this cluster or it searches this cluster and possibly its left neighbor. Additionally a deletion might change $O(\log u)$ bucket pointers. Thus the amortized running time of insertions and queries is proportional to the time spent in a cluster and the amortized running time of deletions is proportional to $O(\log u)$ plus the time spent in a cluster. We will prove in the next lemma that the expected time spent in the largest cluster is constant. This shows that the expected running time for even the hardest query is constant. This data structure is called the *simple data structure*.

Lemma 6.2.4 *The expected time spent in the largest cluster is constant.*

Proof: Consider the execution of operation $t+1$ for $t \geq 0$ and let $T(t+1)$ be the expected time spent in the largest cluster. Assume that c is the size of this cluster and that k is the size of S after the last reconstruction. If $c < f \log u$ then the answer can be looked up in the table T. This takes constant time. Otherwise the time spent is $O(\log \log u)$.

$$T(t+1) = O(1 + \sum_{c \geq f \log u} P(\tilde{X}(F,t) = c) \log\log u)$$

$$= O(1 + P(\tilde{X}(F,t) \geq f \log u) \log\log u)$$

$$= O(1 + 2^{\log k + \log 3 - (c_0 f)\log u} \log\log u) = O(1),$$

if we choose $f \geq 1/c_0 + 1$, since $k \leq u$. ∎

Theorem 6.2.5 *With $O(u^\epsilon)$ preprocessing time the expected running time of m insertions is $O(m)$ and the expected running time of m deletions is $O(m \log u)$. The expected running time of the hardest query is constant.*

6.3 An Improved Solution

To reduce the amortized running time of deletions to the running time of queries, we employ the following technique: Consider the ordered list of elements in S. We break this list into sublists $L_1, \ldots, L_l$. Let $|L_i|$ be the length of the ith sublist. We guarantee that $|L_i| = O(\log u)$ for all $1 \leq i \leq l$ and $l = O(n/\log u)$, and we represent each sublist by its AFK-encoding. A new item x is added to L_i so that $\min L_i < x < \min L_{i+1}$, or in L_1 if no such list exists. The sublists are stored in the simple data structure of Section 6.2 indexed by their minimum element. Whenever the size of a sublist becomes larger than $2 \log u$, we split the sublist in two. Whenever it is smaller than $\frac{1}{2} \log u$ and $l > 1$, we join it with L_{i-1} or L_{i+1} and the resulting set is split in half, if its size is larger than $2 \log u$. Since the subsets can only split and join $O(m/\log u)$ times during m insertions and deletions, the time of each update operation in the simple data structure is proportional to the time for finding the right sublist to insert or delete. This is proportional to the time for executing a query operation. Therefore the expected running time of m deletions is $O(m)$.

Theorem 6.3.1 *With $O(u^\epsilon)$ preprocessing time, the expected running time of m insertions and deletions is $O(m)$. The expected running time of the hardest query is constant.*

Chapter 7

Conclusions

We have proposed data structures for (i) maintaining biconnectivity in plane and general graphs, (ii) maintaining two-edge connectivity in plane graphs, and (iii) maintaining a planar embedding of a graph during a sequence of insertions and deletions of edges and isolated vertices. Additionally we have given lower bounds for fully-dynamic planarity testing and for fully dynamic k-edge and k-vertex connectivity in plane graphs. Finally we have presented a data structure for finding predecessors in integer sets. We consider below some interesting open problems for future work.

The best known solution for the problem of fully dynamic maintenance of connectivity or two-edge connectivity requires time $O(\sqrt{n}\log m/n)$ per operation. The only known lower bound for the problem is $\Omega(\log n/\log\log n)$ (see Chapter 5). Thus an obvious problem is to close the gap between these two bounds.

A second question is whether biconnectivity information in general graphs can be maintained in the same time as connectivity information and two-edge connectivity information under insertions and deletions of edges. In the static case and in the insertions-only case all three properties can be computed in the same running time. One possible approach for improving the running time would be to further refine the two-level scheme of the algorithm presented in Chapter 2 into a multi-level hierarchy.

The semi-dynamic version of any of the above problems with deletions-only is largely unexplored. As mentioned in the introduction, no faster solutions for the deletions-only versions of these connectivity problems are known than those for the

fully dynamic versions. There are also no known lower bounds for these problems.

As was shown in Chapter 3, there are very efficient solutions for fully dynamic problems in plane graphs. A possible extension of this work would be to investigate generalizations of plane graphs, such as graphs embedded in fixed-genus surfaces..

The best current solution for fully dynamic k-edge connectivity for constant $k > 4$ is $O(n \log n)$ time per operation [7]. However, nothing better than recomputation from scratch is known for k-vertex connectivity for $k > 2$ in a fully dynamic environment.

There is only a little known about the fully dynamic maintenance of other graph properties, like the diameter of a graph, shortest paths, or a small separator in a planar graph. All these problems provide interesting areas for future research.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, MA, 1974.

[2] M. Ajtai, "A lower bound for finding predecessors in Yao's cell probe model", *Combinatorica*, 8(3), 1988, 235–247.

[3] M. Ajtai, M. Fredman, and J. Komlós, "Hash Functions for Priority Queues", *Information and Control*, 63(3), 1984, 217–225.

[4] H. Booth, "Some Fast Algorithms on Graphs and Trees", *Ph.D. Thesis, Princeton University*, 1991.

[5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic perfect hashing: Upper and lower bounds", *Proc. 29th Annual Sympos. on Foundations of Computer Science*, 1988, 524–531.

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

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

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

[9] G. N. Frederickson. "Ambivalent data structures for dynamic 2-edge connectivity and k smallest spanning trees." *Proc. of 32nd Annual Symp. on Foundations of Computer Science*, 1991, 632–641.

[10] M. L. Fredman, private communication.

[11] 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.

[12] Z. Galil, and G. F. Italiano, "Maintaining Biconnected Components of Dynamic Planar Graphs", *Proc. 18th ICALP*, Lecture Notes in Computer Science, Vol., Springer-Verlag, Berlin, 1991, 339–350.

[13] Z. Galil, and G. F. Italiano, "A Note on Set Union with Arbitrary Deunions", *Information Processing Letters*, 37(6), 1991, 331–335.

[14] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[15] D. Harel, and R. E. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors", *SIAM J. Comput.*, 13(2), 1984, 338–355.

[16] J. Hershberger, M. Rauch, and 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] J. H. Hopcroft, and R. E. Tarjan, "Dividing a Graph into Tri-Connected Components," *SIAM J. Comput.*, 2(3), 1973, 135–158.

[18] G. F. Italiano, private communication.

[19] H. LaPoutré, private communication.

[20] G. L. Miller, and V. Ramachandran, "A New Graph Triconnectivity Algorithm and its Parallelization", *Proc. 19th Annual Symp. on Theory of Computing*, 1987, 335–344.

[21] M. Rauch, "Fully Dynamic Biconnectivity in Graphs", *Proc. 33nd Annual Symp. on Foundations of Computer Science*, 1992, 50–59.

[22] D. D. Sleator, and R. E. Tarjan, "A Data Structure for Dynamic Trees", *J. Comput. System Sci.*, 26(3), 1983, 362–381.

[23] R. Tamassia "A Dynamic Data Structure for Planar Graph Embedding" *Proc. 15th International Colloquium on Automata, Languages and Prgramming*, LNCS 317, Springer-Verlag, 1988, 576–590.

[24] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, 1983.

[25] R. E. Tarjan, "Depth First Search and Linear Graph Algorithms", *SIAM J. Comput.*, 1(2), 1972, 146–160.

[26] R. E. Tarjan, and U. Vishkin, "An Efficient Parallel Biconnectivity Algorithm", *SIAM J. Comput.*, 14(4), 1985, 862–874.

[27] R. E. Tarjan, and J. Westbrook, "Maintaining bridge-connected and biconnected components on-line", *Algorithmica*, 7(5), 1992, 433–464.

[28] R. E. Tarjan, and J. Westbrook, "Amortized Analysis of Algorithms for Set Union with Backtracking", *SIAM J. Comput.*, 18(1), 1989, 1–11.

[29] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space", *Inform. Process. Lett.*, 6(3), 1977, 80–82.

[30] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue", *Math. Sys. Theo.*, 10(2), 1977, 99–127.

[31] D. Willard, "Log-logarithmic worst-case range queries are possible in space $\Theta(n)$", *Inform. Process. Lett.*, 17(2), 1983, 81–84.

[32] A. Yao, "Probabilistic computations: Towards a unified measure of complexity", *Proc. 33nd Annual Symp. on Foundations of Computer Science*, 1977, 222–227.

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