Technical Report DIKU-TR-98/17
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 KBH Ø
DENMARK

October 1998

Top-trees and dynamic graph algorithms

Jacob Holm
Kristian de Lichtenberg

Top-trees and dynamic graph algorithms

Jacob Holm
samson@diku.dk

Kristian de Lichtenberg
morat@diku.dk

Master's Thesis

Department of Computer Science,
University of Copenhagen,
August 14, 1998.

Contents

Chapter 1

Introduction

In a dynamic graph problem one would like maintain a data structure to answer a certain query, such that when a simple change is introduced, e.g. the insertion or deletion of an edge, the data structure can be updated faster than recomputing from scratch. As an example, think of a graph of a communications network, where nodes are relay stations, and edges are communication links (phone lines). In such a system, it is relevant to know

1. If a link accidentally breaks down or is rebuild, are all stations still connected?
2. If a link breaks down, is it cheaper to build a new link (and which) than to repair the old?
3. Is the system secure such that all stations will still be connected, no matter which link breaks down? Which links should we build to secure it?
4. If a station is suddenly out of order, are the other stations still connected about it? Which links should we build to secure this?

These are important questions with applications to assembly planning, chip-design, graphics, communication networks, and more [6]. Thus as cpu's grow larger, and the Internet expands, it becomes increasingly more important to find efficient dynamic graph algorithms.

This thesis address questions 1 through 4, more formally known as connectivity, minimum spanning forest, 2-edge connectivity and biconnectivity in fully dynamic graphs. For connectivity we provide an $O(\log^2 n)$ time algorithm, for minimum spanning forest, 2-edge connectivity and biconnectivity the time complexity is $O(\log^4 n)$. Our solutions to these problems have been presented in [28]. Chapter 5 contains further introduction to these problems and an overview of previous solutions.

In many dynamic graph algorithms, a data structure for dynamic trees is used. A well known data structure for dynamic trees is the topology trees of Frederickson [12]. The topology trees are only defined for ternary trees, and it has been an open problem whether topology trees can be generalized to general trees with unbounded degree. Such a variant, called *top-trees,* is provided in this thesis. The top-tree data structure is subsequently used to provide $O(\log n)$ time solutions to the Diameter, 1-Center, and 1-Median problems, and is basis of the data structures used in the dynamic graph algorithms described above. Further introduction to these problems is found in chapter 2. The top-tree data structure and the diameter algorithm has previously been presented in [1].

1.1 Acknowledgements

We would like to thank Mikkel Thorup who has been our supervisor on this masters thesis. We have had many valuable discussions, and Mikkel has been a great help in getting the algorithms right and presenting them properly.

Stephen Alstrup, our co-supervisor, started this project up by showing us the diameter problem. In addition, we must thank him for many valuable discussions. Stephen has worked hard to push students to research level. This thesis is one of the results of this work.

Finally we must thank the following people: Niels Christensen and Hans Henrik Stærfeldt, for always being ready to listen to another algorithm and helping us present them in a understandable manner. In addition to Niels and Hans Henrik, Martin Appel, and Peer Sommerlund also proof read the final thesis and gave many helpful suggestions.

1.2 Organisation of this thesis

This thesis has two main parts. Part I, chapters 2 through 4, is about dynamic tree algorithms. Part II, chapters 5 through chapter 9, is about dynamic graph algorithms.

In part I we present our dynamic tree data structure called *top-trees* and apply them to give simple solutions to the 1-Median, 1-Center, and Diameter problems in fully dynamic trees. In the process we provide a general framework for finding edges with certain *global* properties. Chapter 2 contains an introduction to other data structures for dynamic trees and to previous work on the 1-Median, 1-Center, and Diameter problems.

In part II we consider the connectivity, minimum spanning forest, 2-edge connectivity, and biconnectivity problems in fully dynamic graphs. For the minimum spanning forest, 2-edge connectivity, and biconnectivity, the algorithms are implemented through the top-tree black box interface, and the reader should consult sections 3.1 and 3.2 before reading the implementation details. Chapter 5 contains an introduction to the problems considered and an overview of previous work.

In chapter 10 we summarize the contents of this thesis.

1.3 Readers prerequisites

In general the terms used are defined in this thesis, but the reader is assumed to have knowledge of algorithmics and data structures corresponding to the book “Introduction to algorithms”, Cormen et. al. [4].

Part I

Top-trees

Chapter 2

Introduction to part I

In this part we consider fully dynamic tree problems. In a dynamic tree problem, we have a forest of trees over a fixed set of nodes V of size $|V| = n$. The forest may be *updated* with insertions and deletions of edges. In chapter 3 we describe top-trees, a data structures for dynamic trees. In chapter 4 we consider the fully dynamic tree problems of Maxweight, Diameter, 1-Center, and 1-Median, implemented through top-trees. In connection with the 1-Center and 1-Median problem, we provide a general tool for maintaining edges with global properties. The Maxweight problem was first considered by Sleator and Tarjan [35]. The top-tree data structure and the algorithm for the Diameter problem was first described in [1] while the results for 1-Center and 1-Median are to be included in a forthcoming article on experimental implementation of top-trees.

2.1 Top-trees

Many dynamic graph data structures are based on a data structure for dynamic trees. Therefore many different data structures have been developed for dynamic trees. Examples are the dynamic trees (here *ST-trees*) of Sleator and Tarjan [35], the *ET-trees* developed in [9] based on Euler tours and finally the *topology trees* of Frederickson [12, 13, 14, 15].

The ET-trees are simple and efficient but limited, in that they cannot maintain information about paths. Both the ST-trees and the topology trees partition the tree into node-disjoint parts. The ST-trees partition the tree according to paths, whereas topology trees use the topology of the tree in a more clever way. The structure of the topology tree implies more complicated algorithms but simpler data structures for representing paths [6]. The problem is that the topology trees work only for trees of maximum degree 3, thus it is necessary to transform the tree to a tree of maximum degree 3 and the solve the problem there. This has given rise to a number of complicated transformations, and previous articles usually start by presenting the entire topology tree data structure from scratch, see e.g. [6, 14, 29].

Our contribution is a variant of the topology trees called *top-trees*. The top-trees are based on Fredericksons idea of partitioning the tree according to topology, but improves over the topology trees in two ways. First, they work directly for trees of unbounded degree. Second, they come with a complete black box interface, making them applicable without knowledge of the underlying data structure.

In section 3.1 we introduce the terminology of the top-trees. In section 3.2 we provide a black box interface for their usage. Finally in section 3.3 we describe how to implement the top-tree data structure.

To round off this brief introduction to data structures for dynamic trees, in chapter 7 we apply the top-trees to the minimum spanning forest problem, thus all of the above the mentioned data structures for dynamic trees, ST-trees, topology trees, ET-trees and top-trees, have been applied to this problem [10, 24, 35].

2.2 Applications

In chapter 4 we give examples of top-tree algorithms.

In the *fully dynamic Maxweight problem*, we have a tree with weighted edges. In this tree, we may insert or delete edges and perform path updates (adding a constant to all edges on a specified path) and Maxweight queries (which returns the maximum weight of an edge on a specified path). That this can be done in $O(\log n)$ time per operation is a classic result by Sleator and Tarjan, first published in 1983 [35]. This result has applications in many dynamic graph algorithms [6]. In section 4.1 an implementation using top-trees is included as an illustrative example of a top-tree algorithm.

In the *fully dynamic tree Diameter problem*, the weighted edges can be inserted and deleted, and our task is to maintain the diameters of the trees in the forest. The diameter of a tree is the length of the longest simple path in the tree. The length of a path is the sum of the weights of its edges. The diameter of a tree is an interesting problem, as many networks use spanning trees to route their packets. The diameter of a spanning tree is then the longest time it takes a packet to arrive at its destination, thus it is important to have spanning trees of small diameter.

For static trees, several linear time algorithms exists. In 1992 Rauch[1] [32] pointed out that too little work has been done to dynamically maintain the diameter. In section 4.2 we provide an algorithm taking $O(\log n)$ time per operation.

In the *fully dynamic 1-Center problem in dynamic trees,* we again have a fully dynamic forest of trees with weighted edges. For each tree in the forest we must be able to find the node minimizing the maximum distance to any other node. The distance between two connected nodes v and w is the sum of the edges on the path between them and is denoted *dist*(v, w). In section 4.4 we provide an algorithm using $O(\log n)$ time per operation.

In the *fully dynamic 1-Median problem in dynamic trees*, the nodes of the tree are weighted. In addition to insertion and deletion of edges, a constant may be added to the weight of a node, and for each tree we must be able to find a node v minimizing $\sum_{w \in V(T)} \textit{weight}(w) \cdot \textit{dist}(v, w)$. In section 4.5 we provide an algorithm using $O(\log n)$ time per operation.

For the 1-Center problem, the previous best was the $O(\log^2 n)$ time solution provided by Cheng and Ng in [3]. For the 1-Median problem, the previous best solution was provided by Auletta, Parente and Persiano in [2], using $O(\log^2 n)$ time per operation. 1-Center and 1-Median were originally static problems with $O(n)$ time solutions, see e.g. [2, 13, 17, 19]. A long list of references to the 1-Median and 1-Center problem and similar problems can be found in [34].

The Maxweight problem has the nice *local property,* that if T is a tree, T' a subtree, then if $e \in E(T')$ is a solution in T, then e is a solution to T'. As an intermediate step to the 1-Center and 1-Median problems we provide a general tool for maintaining edges with global (i.e. non-local) properties. We require the user to define some custom information for the top-tree clusters, provide the update procedures, and finally, given a root cluster with two children, the user must decide which child cluster contain the desired edge. After $O(\log n)$ decisions, Merge and Split operations, the desired edge is found. To our knowledge, no such tool has been presented before.

[1]Now Monica Rauch Henzinger

Chapter 3

Top-trees

In this chapter we will introduce a variant of Fredericksons topology trees [10]. The original topology trees are defined for ternary trees which can then be used to encode trees of unbounded degrees. This is often quite technical, so instead we have developed a variant (first presented in [1]), called *top-trees*, which works directly for trees of unbounded degree, and which gives rise to much simpler algorithms. For most purposes, top-trees are also easier to use than the dynamic trees of Sleator and Tarjan [35].

The top-tree is a data structure for unrooted dynamic trees that allows simple divide and conquer algorithms. The basic idea is to maintain a balanced binary tree $\mathcal{T}$ of logarithmic height representing a recursive subdivision of the tree T into *clusters*, which are subtrees of T that are connected to the rest of T through at most two *boundary nodes*. Each leaf of $\mathcal{T}$ represents a unique edge of T, each internal node of $\mathcal{T}$ represents the cluster that is the union of the clusters represented by its children, and the root of $\mathcal{T}$ represents all of T. We will abuse the notation and use the word cluster to mean both a subtree of T with at most two boundary nodes and the node in $\mathcal{T}$ which represents it, if any.

3.1 Terminology

A node in a connected subtree is a *boundary node*, if it is connected to a node outside the subtree by a single edge. A *cluster* is then a subtree with at most two boundary nodes.

The set of boundary nodes of a given cluster $\mathcal{C}$ is denoted $\partial\mathcal{C}$, and a node in $\mathcal{C} \setminus \partial\mathcal{C}$ is called an *internal node* of $\mathcal{C}$. The simple path between the boundary nodes of $\mathcal{C}$ is called the *cluster path* of $\mathcal{C}$ and is denoted $\pi(\mathcal{C})$. If $\pi(\mathcal{C})$ contains at least one edge, $\mathcal{C}$ is called a *path-cluster,* otherwise $\mathcal{C}$ has only one boundary node and $\mathcal{C}$ is called a *leaf-cluster*.

If $\mathcal{A}$ and $\mathcal{B}$ are clusters with *exactly one* node in common, and $\mathcal{A} \cup \mathcal{B}$ is a cluster, $\mathcal{A}$ and $\mathcal{B}$ are said to be *mergeable*.

As clusters are represented by nodes in a rooted tree, we can apply the (proper) ancestor/descendant/child/parent relationship among these nodes to the clusters they represent. The cluster $\mathcal{C}$ is said to be a *(proper) path-ancestor/path-descendant/path-child* of a cluster $\mathcal{A}$ if $\mathcal{C}$ is a (proper) ancestor/descendant/child of $\mathcal{A}$ and the cluster paths of $\mathcal{C}$ and $\mathcal{A}$ have at least one edge in common.

A cluster containing a single edge is called an *edge-cluster*. A leaf in the original tree corresponds to an edge-cluster with only one boundary node. Such a cluster is called a *leaf-edge-cluster*. Edge-clusters with two boundary nodes are called *path-edge-clusters*.

In figure 3.1 is shown an example tree of edge-clusters with its top-tree. The node a is internal to A, while c is the single boundary node of the leaf-clusters A, B and $(A \cup B)$, but is internal to the leaf-cluster $((A \cup B) \cup (C \cup D))$ whose single boundary node is e. C ,D and F are path-clusters, while A, B, E, G, H and I are leaf-clusters. F is the sole path-child of $(E \cup F)$ while C and D are both path-children of $(C \cup D)$. Finally note that the cluster $((A \cup B) \cup (C \cup D))$ is a leaf-cluster and $(C \cup D)$ is an example of a path-cluster, that is not a path-child of its parent.
As a slight generalization from the above description we may have up to two *external boundary nodes* for each top-tree $\mathcal{T}$. These nodes are always considered boundary nodes, even if they do not have an edge out of the cluster. In particular, they are the only boundary nodes of the root cluster of $\mathcal{T}$. This enables us to create a path-cluster with any given path as cluster-path. For a simple example of why this is useful, see Corollary 3.3 below and much of the rest of the thesis.

3.2 Black box

We are now ready to describe the black box interface.
The top-tree supports the following update operations:

Link(v, w)**:** Where v and w are nodes in different top-trees $\mathcal{T}_v$ and $\mathcal{T}_w$. Creates a single new top-tree $\mathcal{T}$ representing $\mathcal{T}_v \cup \mathcal{T}_w \cup \{(v, w)\}$.

Cut(v, w)**:** Removes the edge (v, w) from the top-tree $\mathcal{T}$ containing it, thus separating the endpoints and creating two top-trees $\mathcal{T}_v$ and $\mathcal{T}_w$.

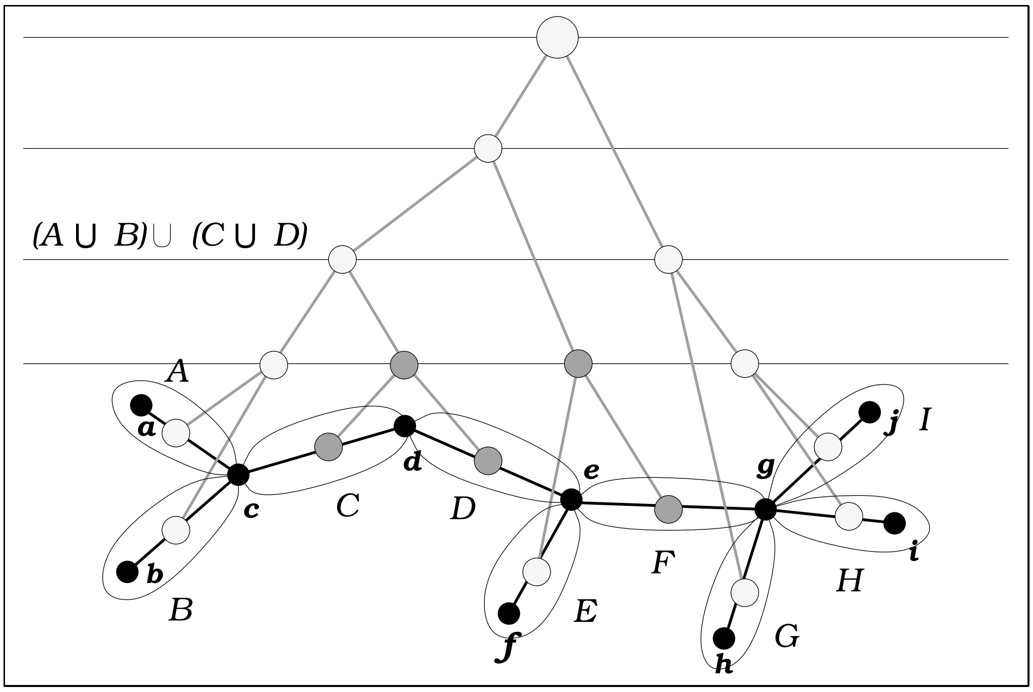

Figure 3.1: *A tree divided into edge clusters and the complete top-tree for it. Filled nodes in the top-tree are path-clusters, while small circle nodes are leaf-clusters. The big circle node is the root. Capital letters denote clusters, non-capital letters are nodes.*

Expose(v, w)**:** Makes v and w external boundary nodes of the tree $\mathcal{T}$ containing them and returns the new root cluster.

With each cluster $\mathcal{C}$, the user may associate some information *Info*$(\mathcal{C})$ and provide some procedures to maintain it. To be more specific, every update to the top-tree is implemented as a sequence of the following four *internal operations*:

Merge$(\mathcal{A}, \mathcal{B})$**:** Where $\mathcal{A}$ and $\mathcal{B}$ are mergeable clusters. Create $\mathcal{C} = \mathcal{A} \cup \mathcal{B}$ as the parent of $\mathcal{A}$ and $\mathcal{B}$, set $\partial\mathcal{C} = \partial(\mathcal{A} \cup \mathcal{B})$. Set *Info*$(\mathcal{C}) :=$ Merge$(\mathcal{C} : \mathcal{A}, \mathcal{B})$, where Merge$(\mathcal{C} : \mathcal{A}, \mathcal{B})$ is a user defined procedure that computes *Info*$(\mathcal{C})$ from $\mathcal{A}$ and $\mathcal{B}$ and their associated information.

Split$(\mathcal{C})$**:** Where $\mathcal{C} = \mathcal{A} \cup \mathcal{B}$. Call Split$(\mathcal{C} : \mathcal{A}, \mathcal{B})$, delete the node $\mathcal{C}$ from $\mathcal{T}$; where Split$(\mathcal{C} : \mathcal{A}, \mathcal{B})$ is a user defined procedure that updates *Info*$(\mathcal{A})$ and *Info*$(\mathcal{B})$ before $\mathcal{C}$ is deleted, allowing *Info*$(\mathcal{C})$ to be propagated down to its children.

Create$((v,w))$**:** Create an edge cluster $\mathcal{C}$ for the edge (v,w). Set $\partial\mathcal{C} := \partial(v,w)$. Set *Info*$(\mathcal{C})$:= Create$(\mathcal{C} : (v,w))$, where Create$(\mathcal{C} : (v,w))$ is a user defined procedure that computes *Info*$(\mathcal{C})$.

Eradicate$(\mathcal{C})$**:** Where $\mathcal{C}$ is an edge-cluster for an edge (v,w). Call Eradicate$(\mathcal{C} : (v,w))$, delete $\mathcal{C}$, where Eradicate$(\mathcal{C} : (v,w))$, is a user defined procedure that stores *Info*$(\mathcal{C} = (v,w))$. When updating the tree, an edge cluster may change from a leaf-cluster to a path-cluster or vice versa. Since the information stored in leaf-clusters and path-clusters may be different, to implement the change, we delete the edge-cluster using Eradicate$(\mathcal{C} = (v,w))$ and then reinsert it changed with Create$(\mathcal{C} = (v,w))$ using the stored information.

Each of the above four internal operations uses constant time plus the time used by the corresponding user-defined procedure. We can therefore state the time complexities involved in maintaining the top-tree as the number of internal operations used:

THEOREM 3.1 (MAIN THEOREM). *We can maintain a forest of top-trees of height $O(\log n)$ supporting the operations* Link, Cut *and* Expose, *using a sequence of at most $O(\log n)$* Merge *and* Split *plus a constant number of* Create *and* Eradicate *per operation. In addition this sequence can be computed in $O(\log n)$ time. Furthermore the top-tree data structure can be initialized using $O(n)$* Create *and* Merge *operations.*

The proof of the Theorem is rather involved and is the subject of section 3.3 which also includes a more detailed description of the top-tree data structure.

Note that since the height of any top-tree is $O(\log n)$, we have that an edge is contained in at most $O(\log n)$ clusters. A node is internal to at most $O(\log n)$ clusters, and we assume a pointer from each node v to the cluster with least height it is internal to, or if v is an external boundary node, the pointer points to the root-cluster. As a result, we can test whether a pair of nodes has an edge between them in constant time.

COROLLARY 3.2. *Given a pair of nodes v, w, we can test whether the edge (v,w) is in the top-tree data structure in constant time per operation.*

PROOF. Let $\mathcal{T}$ be the toptree, and let $\mathcal{C}_v$ denote the unique smallest cluster in $\mathcal{T}$ in which v is internal. If $\mathcal{C}_v$ is a leaf-edge cluster, (v,w) is an edge if and only if w is the boundary node of $\mathcal{C}_v$. Otherwise $\mathcal{C}_v$ is not an edge cluster. Assume its height is less than the height of the corresponding $\mathcal{C}_w$. This means that w must be a boundary node of $\mathcal{C}_v$ if (v,w) is an edge and that a child cluster $\mathcal{A}$ of $\mathcal{C}_v$ has boundary nodes $\partial\mathcal{A} = \{v, w\}$. We then just need to check if $\pi(\mathcal{A})$ consists of a single edge. This can easily be stored in the cluster.

If $\mathcal{C}_v$ and $\mathcal{C}_w$ have the same height, they cannot be the same cluster, since at most one internal node of a cluster $\mathcal{C}$ can be a boundary node of a child of $\mathcal{C}$. □

To illustrate the power of our machinery, we now give a short proof of a result from [35]:

COROLLARY 3.3. *We can maintain a fully dynamic forest F and support queries about the maximum weight edge between any two nodes in $O(\log n)$ time per operation.*

PROOF. For each path-cluster $\mathcal{C}$ we maintain the maximum weight $W_\mathcal{C}$ on the cluster path. Create$(\mathcal{C} : (v, w))$ sets $W_\mathcal{C} :=$ *weight*(v, w) if $\mathcal{C}$ is a path-edge cluster. Merge$(\mathcal{C} : \mathcal{A}, \mathcal{B})$ sets $W_\mathcal{C} := \max\{W_\mathcal{D} | \mathcal{D} \in \{\mathcal{A}, \mathcal{B}\}$ is a path-child of $\mathcal{C}\}$, while Split$(\mathcal{C} : \mathcal{A}, \mathcal{B})$ and Eradicate$(\mathcal{C} : (v, w))$ does nothing. All internal operations take constant time. To answer the query *Maxweight*$(v \cdots w)$ we just call $\mathcal{C} :=$ Expose(v, w) and return $W_\mathcal{C}$. The Corollary is established by Theorem 3.1. □

3.3 Implementation

In this section we describe how to build the top-tree data structure and how to maintain it under Link, Cut and Expose operations.

First, in section 3.3.1 we give a high level description of the data structure assuming we have a strategy for creating partitions of a tree into edge-disjoint clusters. It turns out that the main difficulty is to select the right such strategy, and this problem is addressed in section 3.3.2. In section 3.3.3 we show that our strategy yields top-trees with $O(\log n)$ height, and finally in section 3.3.4 we show how to maintain our data structure.

3.3.1 The top-tree data structure

Recall from the high level description of the top-trees, that a cluster in a tree T is a subtree of T, that is connected to the rest of T through at most two *boundary nodes*. By a *cluster-partition* of T, we mean a partition of T into edge-disjoint clusters, such that no cluster contains more than two edges.

Any cluster-partition of T can be represented by a *cluster-partition tree* (*CPT*) T', by replacing each cluster in T with an edge.

Each edge in T' is said to be *composite* if its cluster contains two edges from T, and *simple* if it contains only one. Similarly, an edge is a *leaf-edge* if it represents a leaf-cluster, and a *path-edge* if it represents a path-cluster.

Let P be a strategy for creating cluster-partitions for trees (to be specified later). By a *partition strategy* we mean a set of rules the clusters in the partition must obey. Let $CPT_P(T)$ denote the function that constructs a tree representing a cluster-partition of T in accordance with the rules of P.

DEFINITION 3.4. *Let T be a tree, and P a cluster-partitioning strategy. A* multilevel (cluster-)partition *of T with respect to P is then a set of trees* (levels) $F_0, \ldots, F_s$ *where $F_0 = T$ and $F_i = CPT_P(F_{i-1})$, $i \in \{1, \ldots, s\}$.*

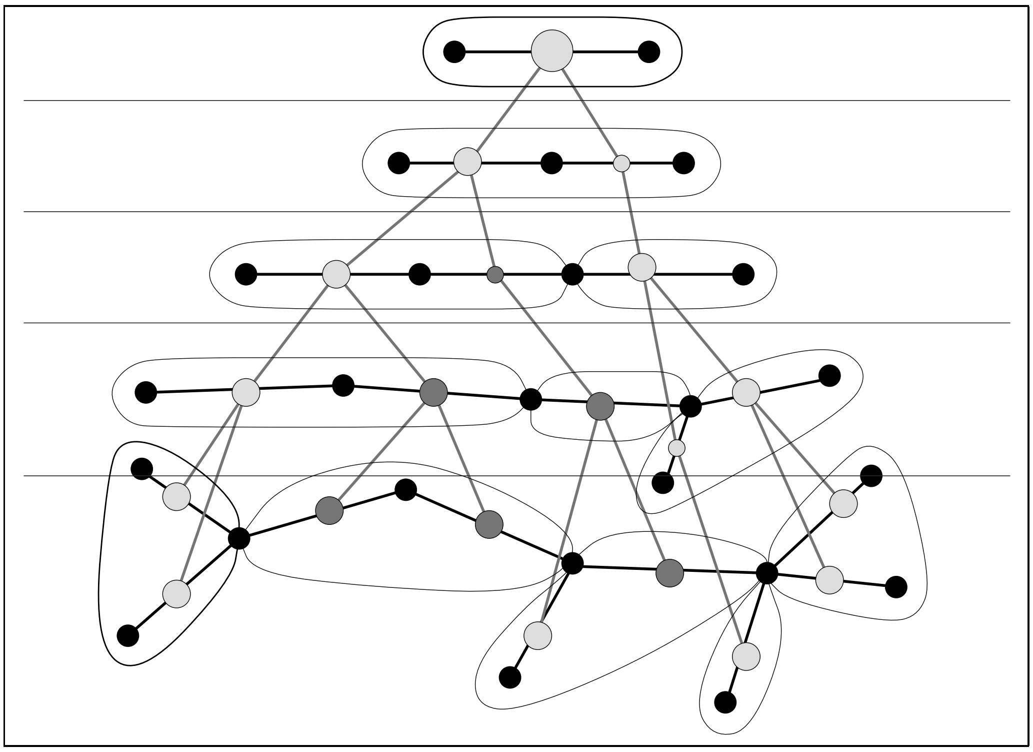

Figure 3.2: *An example of a multilevel partition.*

Let $F_0, \ldots, F_s$ be a multilevel partition. An edge e in $F_i, i > 0$ is said to be the parent of the edges in its cluster in F_{i-1} and they are said to be e's children. The definition of ancestor and descendant then follows. Each node in F_i, is also a node in F_{i-1}, an for this reason, they are considered identical. Each edge in F_0 represents a unique edge of T and each edge in F_i, $i \in \{1, \ldots, s\}$ represents the cluster that is the union of the clusters represented by its children.

A top-tree $\mathcal{T}$ for a tree T is then implemented by maintaining a multilevel partition for T: Each node v in the toptree corresponds to an edge e_v in the multilevel partition, and its children/parent in the top-tree is the nodes corresponding to the children/parent of e_v. Note that a multi-level partition may have edges which have only a single child. This corresponds to nodes in the top-tree that have only one child, which were not allowed in the high-level description. This modification is introduced to simplify the description of the data structure. Creating a composite edge is what corresponds to a Merge, deleting a composite edge corresponds to a Split. The edges in F_0 corresponds to edge-clusters and are created by Create and deleted by Eradicate. The remaining simple edges in $F_i, i > 0$ is made by copying their child on the previous level.

With the reduction from the multilevel partition to the top-tree data structure described, we will now only consider how to create and maintain a multilevel partition.

3.3.2 Selecting a partition strategy

To complete the description of the multilevel partition data structure, we now consider the problem of selecting a partition strategy. We will select a strategy that fulfills the following requirements.

R1. For any tree T, $|E(CPT_P(T))| < k|E(T)|$, for some contraction constant $k < 1$.

R2. If an insertion or deletion is introduced into F_0 in a multilevel partition, the remaining levels $F_1, \ldots, F_s$ can be updated to a multilevel partition with respect to P for F_0 using at most a constant number of insertions and deletions in each F_i.

Requirement R1 ensures that F_i in the multilevel partition of T has less than $k|E(F_{i-1})| \leq k^i|E(F_0)|$ edges, and thus the multilevel partition based on P contains at most $s = \lfloor log_{\frac{1}{k}} n \rfloor$ trees. Since the height of the top-tree is the same as the number of levels in the multilevel partition, the height of the top-tree is $\lfloor log_{\frac{1}{k}} n \rfloor$.

When the logarithmic number of trees in the multilevel partition has been established, Requirement R2 ensures that the multilevel partition can be updated, using at most $O(\log n)$ insertions and deletions to the trees.

We now describe a partition strategy that fulfills requirement R1 and R2. We begin by defining which clusters we will allow.

For each node v we introduce a cyclic order on the incident edges (see Figure 3.3 for an example).

For an edge (v, w), we call the next edge in the cycle around v in its *successor* around v, and similarly the preceding edge in the cycle around v is said to be its *predecessor* around v. If v only has a single incident edge, this edge is its own successor/predecessor.

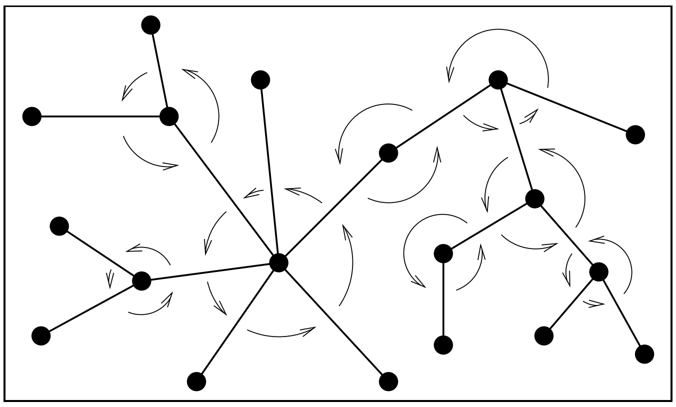

Figure 3.3: *An example tree. For each node for each edge an arrow points to its successor (leaf nodes excluded).*

A pair of edges (v, a) and (v, b) are said to be *neighbours around* v if (v, b) is the successor of (v, a) around v. Thus an edge can have at most four neighbors.

DEFINITION 3.5. *A cluster $\mathcal{C}$ is* good *if either:*

- *$\mathcal{C}$ is a simple cluster.*
- *$\mathcal{C}$ consists of a leaf-edge and its successor.*
- *$\mathcal{C}$ consists of two path-edges which are neighbors around a common endpoint.*

Figure 3.4 is an example of a tree with an arrow for each of the possible good composite clusters. Note that this definition excludes composite clusters that consist of a path-edge and a leaf-edge, where the path-edge is not the successor of the leaf-edge.

A pair of edges whose union is a good cluster are said to be *mergeable*. If an edge forms a mergeable pair with a neighboring edge, that edge is called a *mergeable neighbor*. The most important property of definition 3.5 is that no edge can have more than two mergeable neighbors.

Based on the definition of a good cluster, we now define our partition strategy.

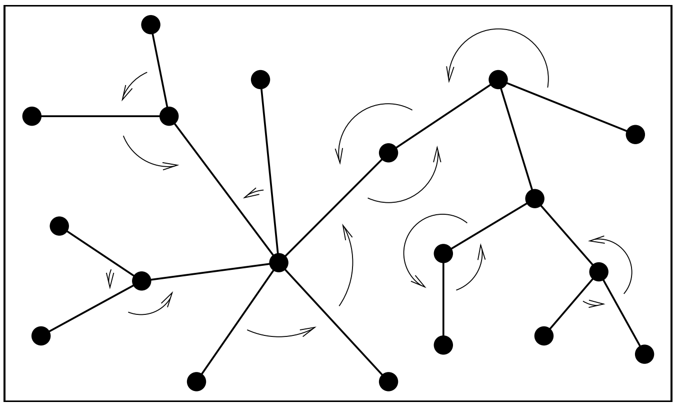

Figure 3.4: *An example tree as in Figure 3.3. For each good composite cluster, an arrow points from the predecessor to the successor.*

DEFINITION 3.6. *A* Maximal Restricted Partition *(*MRP*) of T, is a partition of the set of edges of T into good clusters, such that*

- *Each cluster is a* good *cluster.*
- *No two clusters can be combined to a good cluster.*

This partition strategy does not necessarily result in the minimum number of clusters, and for each tree several such partitions may exist. It does however have the nice property that if the parent of an edge is deleted, we only have to check if it has a mergeable neighbor that is not represented by a composite edge. For this reason. the MRP is said to be a *locally greedy* strategy.

For a tree $T' = \mathit{CPT}_{\text{MRP}}(T)$, in addition to representing an MRP partition of T, the order of the parents around each node is synthesized from its children such that if (a, b) is the predecessor of (b, c) are around b and they have different parents, then the parent of (a, b) is the predecessor of the parent of (b, c). See Figures 3.5 and 3.6.

LEMMA 3.7. *Let e and f be mergeable edges in T, with different parents e' and f' in $T' = \mathit{CPT}_{\text{MRP}}(T)$, then e' and f' are mergeable.*

PROOF. Assume w.l.o.g. that e is the predecessor of f around their common node v. If e is a leaf-edge then e' is a leaf-edge that is a predecessor of f', thus e' and f' are mergeable. If e is a path-edge, then since e and f are mergeable they must be the only edges incident to v in T. In this case e' and f' are the only edges incident to v in T', and they must be mergeable. □

As a consequence of the lemma, if a pair of edges e' and f' in T' are not mergeable, then e' cannot be parent of an edge e, f' parent of an edge f' such that e and f are mergeable.

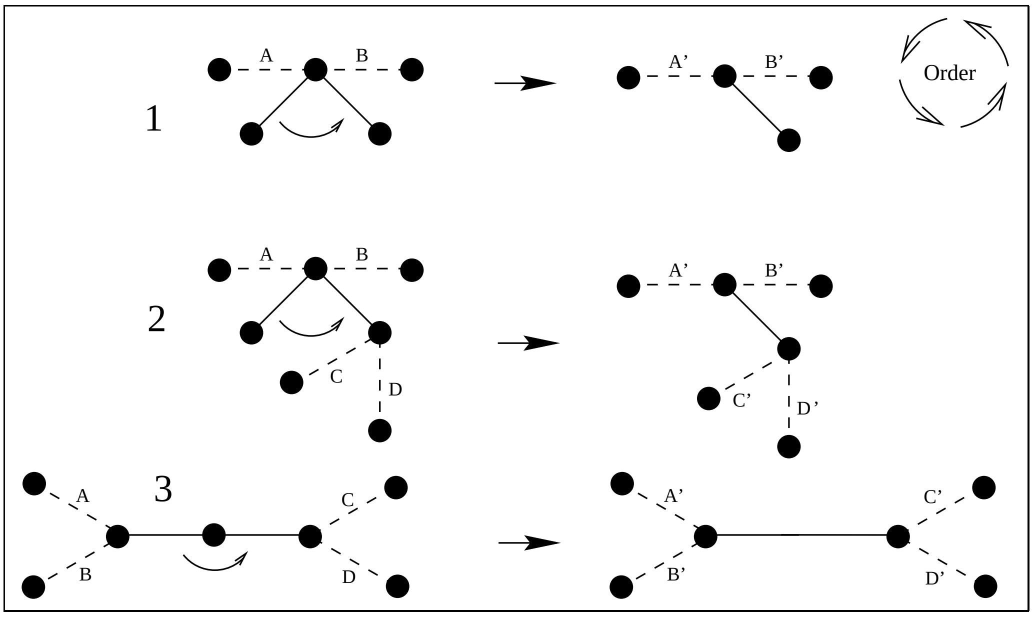

Figure 3.5: *The tree ways a composite cluster can be created, and its parents relation to the parents of their neighbors. Left, dashed edges denotes neighbors, right parents of neighbors. The circle of arrows indicate the successor direction around all nodes.*

For a tree T and a tree $T' = CPT_{\text{MRP}}(T)$, if an edge $e' \in T'$ is simple, the edge it represents in T is said to be *single* and similarly if an edge $e' \in T'$ is composite, the two edges it represents in T are said to be *matched*. An edge with no parent in T' is said to be an *orphan*. To compute a CPT_{MRP} of a tree T, we take each edge $e \in T$ and if it has a mergeable orphan neighbor, they are matched. Since each edge has at most 2 mergeable neighbors, this can be done in linear time.

COROLLARY 3.8. *A maximal restricted partition of a tree T can be computed in linear time.*

We have now presented our partition strategy. In the next section we prove that any MRP of a tree T fulfills requirement R1 and in section 3.3.4 we prove that it fulfills requirement R2.

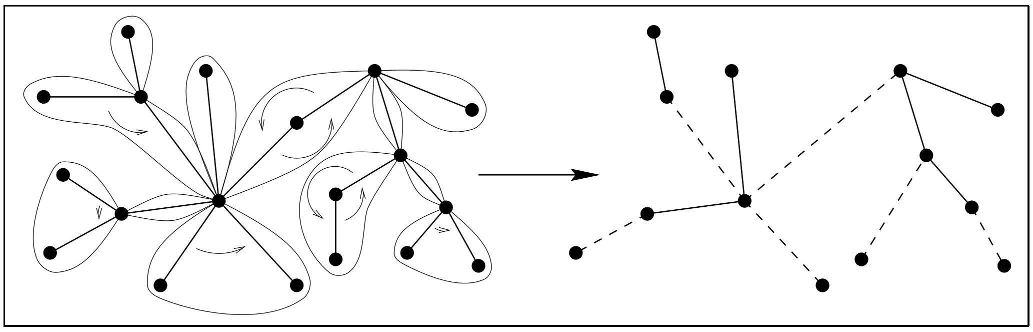

Figure 3.6: *To the left is an example tree with a maximal restricted partition. To the right is a representation of the partition. Normal lines denote simple edges, dashed lines denote composite edges.*

3.3.3 Bounding the number of levels

LEMMA 3.9. *Let T' represent a* MRP *for a tree T with $|T| > 1$, then $|T'| < \frac{5}{6}|T|$.*

PROOF. Let C be the set of composite edges from T'. We will prove that $|C| > \frac{|T'|}{5}$. Since $|T'| + |C| = |T|$ it then follows that $|T'| + \frac{|T'|}{5} < |T|$ and thus $|T'| < \frac{5}{6}|T|$ as desired.

For the sake of the proof, we look at the unique partition Q of T' into clusters $Q = \{\mathcal{C}_1, \ldots, \mathcal{C}_{|Q|}\}$, such that a pair of edges e and f are in the same cluster $\mathcal{C}_i$ if and only if there is a sequence of edges $e = e_0, \ldots, e_l = f$ such that $\{e_{j-1}, e_j\}$ is a good cluster for $j > 0$ (see fig 3.7).
Note that there is no upper limit to the number of edges in a cluster in Q.

All leaf-clusters in Q must contain at least one leaf of T', and since $|T'| > 1$, any leaf-edge e has a mergeable neighbor, thus all leaf-clusters in Q contain at least two edges.

Since T' represents an MRP, every good cluster in T' must contain at least one composite edge and since an edge can be part of at most two good clusters, we must have that $|C \cap \mathcal{C}_i| \geq \lceil \frac{|\mathcal{C}_i|-1}{2} \rceil$. Thus if $\mathcal{C}_i$ has at least 2 edges then $|C \cap \mathcal{C}_i| \geq \frac{|\mathcal{C}_i|}{3}$.

Let L be the set of edges in leaf-clusters in Q, let P be the set of edges in path-clusters in Q with at least 2 edges, and let S be the set of edges in simple clusters in Q. Then L, P and S form a partition of the edges of T', and since all clusters in Q containing edges from L and P are composite, we must have $|C \cap L| + |C \cap P| \geq$

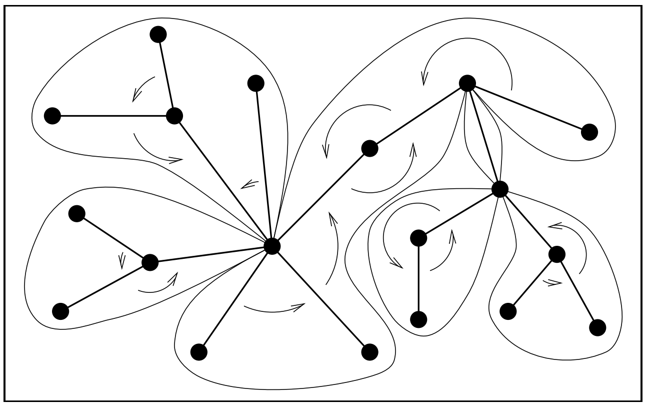
Figure 3.7: *An example of a tree T' and the partition Q in the proof of Lemma 3.9. Here $|L| = 4$, $|P| = 1$ and $|S| = 1$.*

$\frac{|L|+|P|}{3}$ and thus

$$3|C \cap L| + 3|C \cap P| \geq |L| + |P| \tag{3.1}$$

Let T_Q represent Q and let T_R be the subtree of T_Q representing all clusters from Q that has a boundary node that is shared by at least 2 path-clusters, then T_R has at most $|C \cap L|$ leaves. For each node of degree 2 in T_R there must be a path-cluster in Q with more than one edge and this cluster can be shared with at most one other node of degree 2 in T_R. Thus the number of nodes of degree 2 in T_R is at most $2|C \cap P|$. Since a tree with l leaves and p nodes of degree 2 has at most $2l + p - 3$ edges and all edges in S correspond to edges in T_R, we must have that

$$2|C \cap L| + 2|C \cap P| - 3 \geq |T_R| \geq |S| \tag{3.2}$$

Combining (3.1) and (3.2) we get

$$3|C \cap L| + 3|C \cap P| + (2|C \cap L| + 2|C \cap P| - 3) \geq |L| + |P| + |S|$$

and since $|C| \geq |C \cap L| + |C \cap P|$ and $|L| + |P| + |S| = |T'|$ we have $5|C| - 3 \geq |T'|$ and thus $|C| \geq \frac{|T'|+3}{5} > \frac{|T'|}{5}$ as desired. □

We how now proven that the MRP fulfills Requirement R1.

COROLLARY 3.10. *Let T be a tree, any multilevel partition by* MRP *of T has at most $\lfloor \log_{\frac{6}{5}} n \rfloor \approx 3.8 \log_2 n$ levels.*

Note that this estimate of the number of levels is based on the assumption that each level is a worst case tree. In reality this cannot happen, and we believe that a more detailed proof can be constructed, based on a multilevel analysis of the behaviour of this partition strategy, that will show the worst case height to be $3\log_2 n$. However, such a multilevel analysis is beyond the scope of this thesis.

Combining Lemma 3.9 with Corollary 3.8:

LEMMA 3.11. *Any multilevel partition of a tree based on an* MRP *can be created in linear time and uses linear space.*

PROOF. By Corollary 3.8 we use at most $t(|E(F_i)|)$ time to construct F_{i+1}, $i \in 0, \ldots, s$ for some constant $t > 0$. By Lemma 3.9 the combined time used can be bounded by

$$\sum_{i=0}^{s} t|E(F_i)| < t\sum_{i=0}^{s}(\tfrac{5}{6})^i|E(F_0)| < t|E(F_0)|\sum_{i=0}^{\infty}(\tfrac{5}{6})^i = 6t|E(F_0)|$$

By the same equation, the multilevel partition contains at most $6|E(F_0)|$ edges. □

3.3.4 Updating the multilevel partition

When an edge is deleted or inserted in a dynamic tree T, this introduces a change in F_0 in the multilevel partition of T. Our task is then for each $i = 1, \ldots, s$ to update F_i to a *CPT*$_{\text{MRP}}$ of F_{i-1}.

Each step of making F_i a *CPT*$_{\text{MRP}}$ of F_{i-1} involves replacing a set of edges D_i of F_i with another set of edges I_i, each depending on the changes introduced in F_{i-1} and therefore on D_{i-1} and I_{i-1}. For all i, we show that the size of D_i and I_i is bounded by a constant, implying that the multilevel partition can be updated using $O(\log n)$ insertions and deletions.

We start by handling the case where an edge is inserted. When this case has been handled, proving the deletions case is a corollary. Finally we implement Expose.

Link

When inserting an edge (v, w) (we assume v, w were not connected), we insert (v, w) in an arbitrary position in the cyclic order of the edges around v and w in F_0. If v was a leaf node, the corresponding leaf edge (v, v') is now a path-edge in

F_0 and so we delete it with Eradicate, put it in D_0, reinsert (v, v') as a path-edge and put it in I_0. Symmetrically for w and (w, w'). Thus the set of edges I_0 inserted in F_0 is $\{(v, w)\}$ plus possibly (v, v') and (w, w'). The set of edges deleted, D_0 is $I_0 \setminus \{v, w\}$.

Computing D_i and I_i of F_i, $i = 1, \ldots, s$ consists of 3 steps.

1. Delete all parents of edges D_{i-1} in F_i.
2. Delete edges in F_i which, as a result of replacing D_{i-1} with I_{i-1}, no longer represent good clusters.
3. Compute I_i from the children of the edges deleted in F_i in step 1 and 2.

To create I_i, we need to take all the orphans created in step 1 and 2 and match them with a single neighbor. If no match is possible without breaking a previously created match, the edge is single.

In step 1, we delete all edges in F_i whose children have been deleted/replaced (possibly by an edge with same endpoints). In step 2, we delete all edges in F_i that represent pairs of matched edges in F_{i-1} which are no longer mergeable neighbors. This can only happen if an edge of I_{i-1} has been inserted as a neighbor between them, e.g. in F_0. It follows that all edges in I_i created in step 3 are parents of edges in I_{i-1} or parents of orphaned neighbors to edges in I_{i-1}.

To bound the size of I_i, we divide it into two disjoint sets and then bound them independently: Let G_i denote the set of *good* edges in I_i that has a child in I_{i-1} and let $B_i = I_i \setminus G_i$ be the set of *bad* edges in I_i which has no child in I_{i-1}. We will bound $|I_i| = |G_i| + |B_i|$ by bounding $|B_i|$ and $|G_i|$. For all i, we will bound $|B_i|$ by a constant k_B. For $|G_i|$ we will show that for all i, $|G_i| \leq k|I_{i-1}| + k_G$, where $k < 1$ and k_G are constants. This bounds $|I_i|$ by $k|I_{i-1}| + k_G + k_B$ implying $|I_i| \leq \frac{k_G + k_B}{1-k}$.

As each edge in B_i is a parent of a neighbor to an edge in I_{i-1}, we can bound $|B_i|$ by the number edges in $F_{i-1} \setminus I_{i-1}$ which has a neighbor in I_{i-1}. For a set of edges $S_i \subseteq E(F_i)$, we define the *boundary edges* ($\partial_e S_i$) of S_i to be the set of edges in S_i which has a neighbor edge in $F_i \setminus S_i$. Clearly $|B_i| \leq |\partial_e(F_{i-1} \setminus I_{i-1})|$, and since each edge has at most 4 neighbors, $|\partial_e(F_{i-1} \setminus I_{i-1})| \leq 4|\partial_e I_{i-1}|$.

To bound the size of B_i further, we will construct I_i by greedily trying to match each edge in $\partial_e I_{i-1}$ with an orphan edge in $\partial_e(F_{i-1} \setminus I_{i-1})$. This maximizes the number of children of edges in G_i, and since all edges in G_i and B_i are parents of orphans in F_{i-1}, this minimizes the number of orphaned children of edges in B_i.

An edge in F_{i-1} is said to be *free*, if it is either orphaned or single. Step 3 in the algorithm above is then implemented as:

3a. For each edge e of $\partial_e I_{i-1}$, if e has a mergeable orphaned neighbor f in $\partial_e(F_{i-1} \setminus I_{i-1})$, then e and f are matched.

3b. For each remaining orphan e in I_{i-1}, if it has a free mergeable neighbor f, then e and f are matched. Otherwise e is simple.

3c. For each remaining orphan e in $\partial_e F_{i-1} \setminus I_{i-1}$, if e has a free mergeable neighbor f, then e and f is matched. Otherwise e is simple.

Note that step 3a and 3b computes G_i, while step 3c computes B_i and therefore this algorithm correctly computes $I_i = G_i \cup B_i$.

We wish to bound the number of boundary edges of I_i. We can divide $\partial_e I_i$ into two sets $(\partial_e I_i) \cap G_i$ and $(\partial_e I_i) \cap B_i$. As all edges in $(\partial_e I_i) \cap G_i$ are parents of boundary edges in I_{i-1}, if $|(\partial_e I_i) \cap B_i| = 0$, $|\partial_e I_i| = |(\partial_e I_i) \cap G_i| \leq |\partial_e I_{i-1}|$. Thus $|\partial_e I_{i-1}|$ bounds $|\partial_e I_i|$. If however, $|(\partial_e I_i) \cap B_i| > 0$, we wish to show that for all i $|(\partial_e I_i) \cap G_i| + |(\partial_e I_i) \cap B_i|$ is bounded by a constant regardless of the set boundary edges of I_{i-1}. This involves a careful analysis of the parents of the boundary edges of $G_i \cup B_i$, $i \in \{1, \ldots, s\}$.

The update forest

We now consider the structure of the boundary edges of $I_i, i \in \{0, \ldots, s\}$. We divide the nodes connected to the inserted edge (v, w) into two sets: nodes in T_v are those connected to v in $F_0 \setminus \{(v, w)\}$ and nodes in T_w are those connected to w in $F_0 \setminus \{(w, v)\}$.

For a node z in F_i, where z in F_0 is also in $T_v \setminus \{v\}$, the *expansion edge* is the edge incident to z which is ancestor to the first edge on the path from z to v in F_0. Similarly for z in T_w and w. For v and w, the expansion edge is defined to be (v, w).

DEFINITION 3.12. *An* expansion node *of I_i corresponding to v is a node t such that:*

- *The only edges in I_i incident to t are the expansion edge and at most two boundary edges of I_i, which are (predecessor/successor) neighbors to the expansion edge around t.*
- *If the expansion edge has a preceding boundary path-edge in I_i then it also has a succeeding boundary path-edge in I_i and no other edges are incident to t.*

DEFINITION 3.13. *An* update forest *of a forest F_i in a multilevel partition, with respect to (v, w), is a forest of subtrees of F_i with at most one expansion node t_v in*

T_v and at most one expansion node t_w in T_w and all boundary edges in the update forest of (v, w) are incident to t_v or t_w.

It follows that an update forest contains at most two trees and four boundary nodes and edges.

We now proceed to show that for $i \in \{1, \ldots, s\}$, if I_{i-1} is an update forest of F_{i-1} with respect to (v, w), then I_i is an update forest of F_i with respect to (v, w). Initially I_0 is just a path of maximum length 3, which is an update forest with its endpoints as expansion nodes.

The proof runs in stages. First we prove that if U_{i-1} is an update forest of F_{i-1} with respect to (v, w), with a set of parents U_i created by applying step 3a and 3b, then U_i is an update forest of F_i with respect to (v, w). Let C_B be the set of orphaned children remaining before step 3c. Clearly all edges in B_i are parents of C_B. We show that $|C_B| \leq 4$ and $I_{i-1} \cup C_B$ is an update forest of F_{i-1} with respect to (v, w). We can then apply step 3a to find C_B, and then apply steps 3a and 3b on $I_{i-1} \cup C_B$. This implies that we can create $I_i = G_i \cup B_i$ from $I_{i-1} \cup C_B$ such that I_i is an update forest and B_i contains at most 4 edges.

In fact, when step 3a has been applied on I_{i-1}, we have applied step 3a on all edges in $(\partial_e(I_{i-1} \cup C_B)) \cap I_{i-1}$ and what remains is to apply step 3a to the edges of C_B. This is what step 3c does. So applying steps 3a-3c on I_{i-1} creates I_i as an update forest as described.

We now proceed to show that if U_{i-1} is an update forest (of orphans) of F_{i-1}, and U_i is the parents of the edges in U_{i-1} created by applying step 3a and 3b, then U_i is an update forest of F_i with respect to (v, w). The proof is rather involved, and is based on a number of cases. It is therefore necessary to introduce further terminology about the edges incident to an expansion node.

For each expansion node t of an update forest U_{i-1}, if the predecessor of the expansion edge of t is in U_{i-1}, it is a boundary edge and we call it the *left boundary edge*. Similarly, if the successor of the expansion edge is in U_{i-1}, it is called the *right boundary edge*, and by the definition of an expansion node, no more than the left and right boundary edges and the expansion edge may be incident to t in U_{i-1}. Further, if the left boundary edge exists at t and is a path-edge, then the right boundary edge must exist an be a path-edge. Figure 3.8 contains an example of how edges can be arranged around expansion nodes. Note that the expansion edge may still be a boundary edge, if either the right boundary edge or the left boundary edge does not exist. However it is only when the predecessor of the expansion edge around the expansion node is in U_{i-1}, that we have a boundary edge.

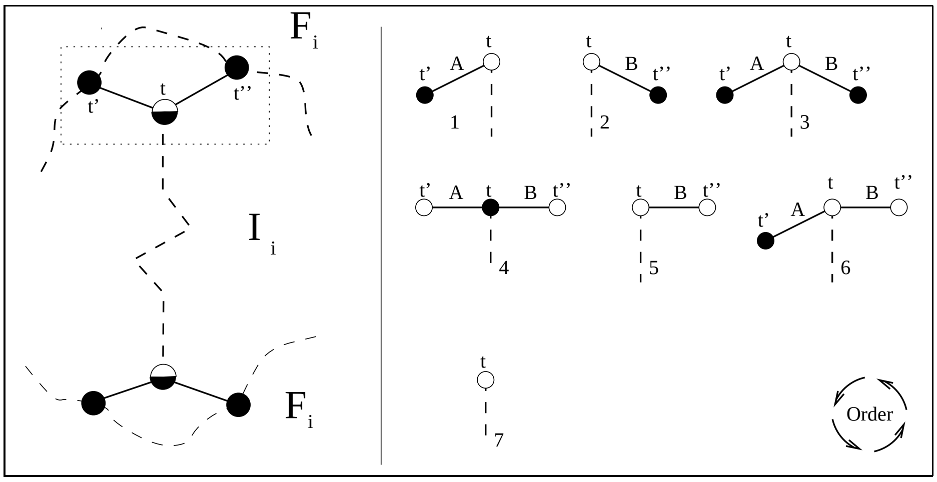

Figure 3.8: *To the left is the general case of an update forest with an expansion node t and the other expansion node at the bottom. To the right, the 7 cases of how the expansion edge (dashed) and the left (A) and right (B) boundary edges may appear around a expansion node. To the right, black nodes are internal nodes of the update forest, whereas white nodes are boundary nodes. Thus A in case 1 is a left boundary edge that is a leaf. The only allowed case where the left boundary edge is a path-edge, is 4.*

As can be seen in Figure 3.8, ex. 4, if the left boundary edge (A) is a path-edge, then the right boundary edge (B) is its predecessor around the expansion node T. This means that each left/right boundary edge has at most one external mergeable neighbor. If the left boundary edge is a leaf, it is its predecessor around t, (and by the definition of a good cluster, it must be leaf). If the left boundary edge is a path-edge, it is its predecessor around t'', (and then it must be a leaf, unless no other edges are incident to t'). If the right boundary edge is a leaf edge, its external neighbor is its successor around t, and it is always a mergeable neighbor (if it is free). Finally, if the right boundary edge is a path-edge, only its predecessor around t'' can be a mergeable neighbor.

To prove that U_i is an update forest, we need to prove that its boundary edges are arranged around at most two expansion nodes. We first state a simple lemma.

LEMMA 3.14. *If an edge is a boundary edge in U_i, then it is the parent of a boundary edge in U_{i-1}.*

PROOF. As the order of the edges i U_i are synthesized from the children, a boundary edge in U_i must have a child in U_{i-1} which had an external neighbor. $\square$

Thus we need to examine the boundary nodes at each expansion node of U and their parents to determine which parents are boundary edges of U_i. By symmetry we can examine one expansion node $t \in \{t_v, t_w\}$ of U_{i-1} at a time.

LEMMA 3.15. *The boundary edges of U_i, which are parents of boundary edges arranged around the same expansion node t in U_{i-1}, are arranged around the same expansion node of U_i.*

PROOF. Let t be an expansion node of U_{i-1}. We examine how the parents of the expansion edge, the right and the left boundary edge are arranged in U_i. If none of the edges can match a free mergeable boundary edge of $F_{i-1} \setminus U_{i-1}$ or they do not exist, t is an expansion node of U_i (possibly) with parents of the left and the right boundary node as left or right boundary nodes, and an expansion edge which is the parent of the expansion edge in U_{i-1}.

Figure 3.9 and Figure 3.10 contains an overview of which external neighbors the left/right boundary leaf/path-edge can match, and the result in U_i.

Boundary edges in U_i must be parents of boundary edges of U_{i-1} and incident to t in U_i. Since there is at most two boundary edges incident to t, there is at most two boundary edges incident to t in U_i.

If none of the boundary edges incident to t is a boundary edge of U_i, it means that all boundary edges incident to t become leaf edges in U_i.

If (t', t) is the only boundary edge, then either t' or t is a boundary node with a single (now expansion edge of the boundary node) edge (t', t) incident. And therefore an expansion node, see fig 3.8, ex. 7.

Assume that there are two boundary edges. Then we know that they are incident to t, and t is an expansion node of U_i if the boundary edges are arranged as in Figure 3.8 ex. 1-7. In particular we only need to show that, if t has a left boundary path-edge, it also has a right boundary path-edge in U_i.

If (t', t) is a left boundary path-edge of t in U_i, by the definition of the cyclic order and since t was an expansion node in U_{i-1}, it must be the parent of a left boundary path-edge of t in U_{i-1}. Since t was an expansion node of U_{i-1}, this implies that t had a right boundary path-edge in U_{i-1}. In particular, only the left, the right and the expansion edge are incident to t in U_{i-1}, see Figure 3.11 (left). If the right boundary edge's parent in U_i is a leaf-edge, Figure 3.11 (middle), this leaf-edge is not a boundary edge, since its successor around t in U_i is the left boundary path-edge (t', t) (in U_i) and its predecessor around t must be the

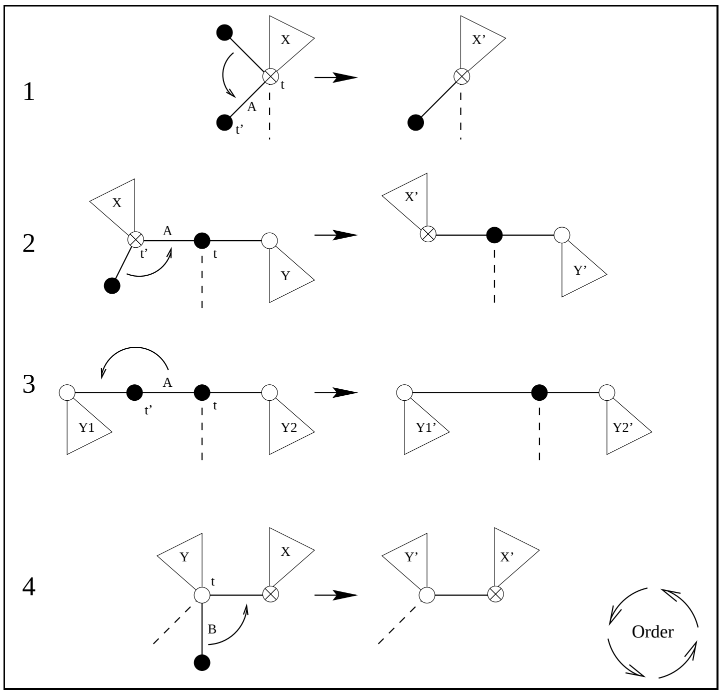

Figure 3.9: *Four general cases of how boundary edges are matched at a expansion node t for U. (1) left leaf-edge. (2,3) left path-edge. (4) right leaf-edge. White nodes are boundary nodes, black nodes are internal, and $\otimes$ nodes are boundary nodes if the incident tree contains edges outside U. X is a possibly empty subtree outside U and Y is a non-empty subtree.*

parent of the expansion edge. In this case, Figure 3.11 (right), (t', t), is the single boundary edge adjacent to t in U_i (and therefore not a left boundary edge) and t' is the new expansion node of U_i with expansion edge (t', t).

Note that if t has been replaced by an expansion node t' in U_i, then (t', t) is an edge in U_i which is not a parent of the expansion edge of t in U_{i-1}, (since t' is a boundary node of U_i). □

COROLLARY 3.16. *If U_{i-1} is an update forest (of orphans) of F_{i-1}, and U_i is the parents of the edges in U_{i-1} created by applying step 3a and 3b, then U_i is an update forest of F_i with respect to (v, w).*

PROOF. By Lemma 3.14 all boundary edges of U_i are parents of a boundary edge U_{i-1}. All boundary edges of U_{i-1} are either arranged around an expansion node

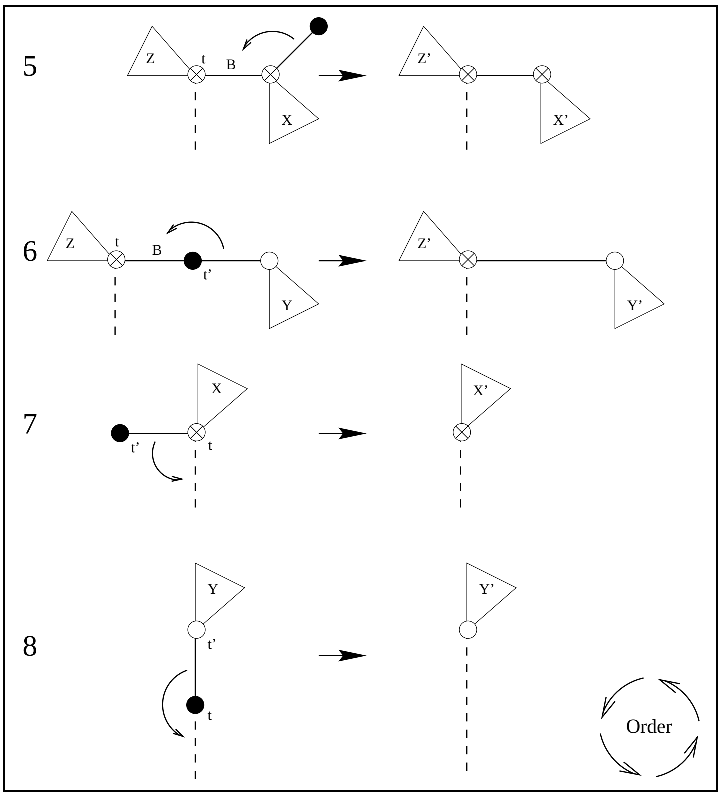

Figure 3.10: *The remaining four general cases of how edges are matched at a expansion node t of U. (5,6) right path-edge. (7,8) expansion edge. White nodes are boundary nodes, black nodes are internal, and $\otimes$ nodes are boundary nodes if the incident tree contains edges outside U. X is a possibly empty subtree outside U, Y is a non-empty subtree outside U, and Z is a subtree with edges both in and out of U.*

t_v or t_w. By Lemma 3.15, for each expansion node t_v, there exists at most one expansion node in U_i, and boundary edges in U_i which are parents of boundary edges incident to t_v in U_{i-1} are incident to the new expansion node. $\square$

We have now shown that applying step 3a and 3b to produce parents of an update forest of F_{i-1} yields a corresponding update forest of F_i. To complete the proof that I_i is an update forest, we need to bound the set of bad orphans C_B and show that $C_B \cup I_{i-1}$ is an update forest.

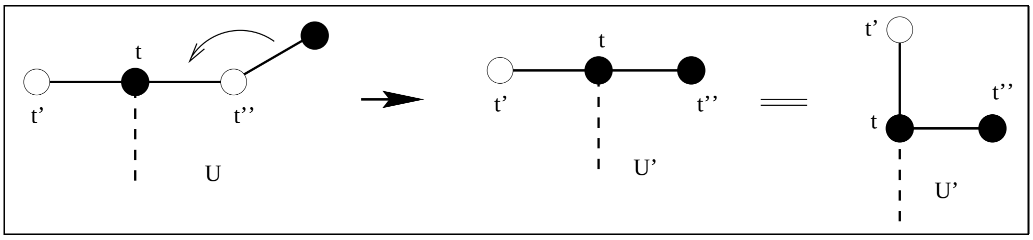

Figure 3.11: *An example of what happens when the left boundary edge is a path-edge, and the right boundary edges parent is a leaf. Left, how it looks in U_{i-1}. In the middle, how it looks in U_i when the right boundary edge's parent is a leaf, (we assume the left boundary edge is a single edge). Right, how the middle tree is oriented in relation to the new expansion node t'.*

LEMMA 3.17. *The remaining set of orphans C_B after step 3a and 3b is bounded and $I_{i-1} \cup C_B$ is an update forest.*

PROOF. After step 3b, no orphans exists in I_{i-1}. In step 3a we match all boundary edges, that can match any free edge outside I_{i-1}. This leaves C_B, the neighbors of boundary edges which cannot match any boundary edges. Recall from the description of left and right boundary edge, that each boundary edge that is not the expansion edge, has at most one mergeable neighbor, thus if an edge is in C_B, this means that it could not match its boundary edge neighbor.

We show that any such edge is a neighbor of the expansion edge, by proving that an unmergeable neighbor of any left or right boundary edge either cannot exist or is a neighbor to the expansion edge. We then consider the possible neighbors of the expansion edge, and show that they can be included as boundary edges in $I_{i-1} \cup C_B$.

An edge e can be in C_B for two reasons. First, if e were previously matched with a neighbor edge that has been deleted and replaced by an edge in I_{i-1}, but with which it can no longer merge. Second, if as a result of inserting an edge as a predecessor of e, e can no longer merge with a neighbor not in I_{i-1}.

If the left or right boundary edge is a path-edge, there are two boundary nodes t', t associated with the expansion node, (see Figure 3.8). Since the expansion edge (and therefore (v, w)) is not on the path between the boundary nodes, this implies that t' and t was connected in F_{i-1} before. This in turn means that a left/right boundary path-edge cannot have replaced a leaf edge, but since they do represent a path in F_{i-1} and previously another path-edge represented the same path (from its neighbors point of view), a left/right boundary path-edge cannot have an external neighbor in C_B since this would contradict 3.7. See Figure 3.12 ex. 1 and 2.

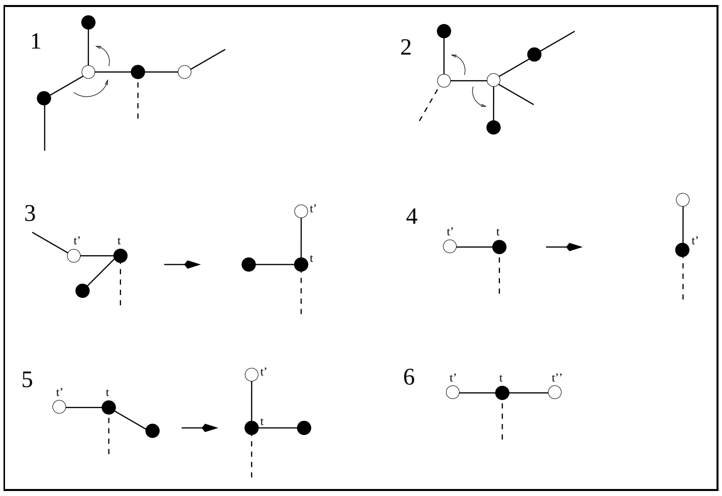

Figure 3.12: *Ex.1 and 2, a sketch of how unmergeable neighbors of the left/right boundary path-edge are arranged. Ex.3, if the left boundary edge is a leaf edge, the only way it could have been matched with a predecessor path-edge is if only the expansion edge is incident to* t. t' *is then the new expansion node. Ex.4. If at most one other edge is incident to* t *and the left/right boundary edges does not exist. Ex. 5. If the expansion edge has a predecessor path-edge and a successor leaf, the path-edge can only be an orphan if it was previously matched with the successor leaf edge. Ex. 6. If the predecessor of the expansion edge is a path-edge and the successor of the expansion edge is a path edge, the left could only be an orphan if it was previously matched with the right.*

A right boundary leaf edge can match any free successor, so it cannot have an orphan neighbor in C_B.

By Lemma 3.7, a left boundary leaf edge can only have a neighbor in C_B if it was an orphan predecessor path-edge, Figure 3.12, ex.3. Since the path-edge could not have matched a successor leaf edge, they can only have been split as a result of an insertion in F_{i-1}. This means that before deleting D_{i-1} and inserting I_{i-1}, the path-edge must have been a successor of an edge represented by the leaf. But this means that the left leaf, the expansion edge and the orphan predecessor path-edge are the only edges incident to the expansion node, particularly, the predecessor path-edge is the successor of the expansion edge and so we make the predecessor path-edge's other endpoint the new expansion node, and the edge itself the new expansion edge.

Finally, if the expansion edge has an orphan neighbor, Figure 3.12, ex.4., we simply make it a new boundary edge unless its a predecessor path-edge and not also the successor of the expansion edge, Figure 3.12, ex.5. If such a predecessor path-edge is an orphan, it must have become an orphan because of the change introduced by inserting the expansion edge or changing a leaf to a path-edge (the expansion edge), and therefore can only have become an orphan as a result of having previously been matched with a path-edge that is also a neighbor of the expansion edge (since it can only affect neighbors), Figure 3.12, ex.6. However, then we make them both boundary edges, and we have not violated the rule that a left boundary edge can only be a boundary path-edge, if the right boundary edge is a path-edge. □

From Lemma 3.17 and Corollary 3.16 and the above discussion we get:

COROLLARY 3.18. *If I_{i-1} is an update forest of F_{i-1} with respect to (v, w), then I_i is an update forest of F_i with respect to (v, w).*

For an update forest, we define the *internal forest* as the set of edges that are not left or right boundary edges.

LEMMA 3.19. *For each expansion node t of I_{i-1}, if t is not an expansion node in I_i, then the internal forest of I_i contains at most 2 edges (parents of boundary edges adjacent to t in I_{i-1}) which are not a parent of an edge in the internal forest of I_{i-1}.*

PROOF. If t is an expansion node in I_{i-1}, then it has at most two adjacent boundary edges. If t is not an expansion node in I_i, then at least one parent of one of its boundary edges is not a boundary edge in I_i (and thus internal) and the endpoint of the other is the replacing expansion node of I_i. Thus the parents of the two boundary edges can become edges of the internal forest of I_i. □

LEMMA 3.20. *If the internal forest in I_{i-1} has s edges, then the internal forest in I_i has at most $\frac{5}{6}s + 6$ edges.*

PROOF. Let S denote the internal forest in I_{i-1}, and let $s = |S|$. By Lemma 3.9 any MRP for S has at most $\frac{5}{6}s$ clusters. In particular this holds for any MRP created by taking any MRP for F_{i-1}, removing all clusters containing edges not in S, and merging each orphaned edge in S with its mergeable neighbor (if any).

This MRP clearly has at most two clusters less than the number of edges in I_i containing edges from S. Thus the number of edges in the internal forest of I_i representing edges in S is at most $\frac{5}{6}s + 2$, and by Lemma 3.19 at most four edges in the internal forest of I_i does not contain edges from S, thus the total number of edges in the internal forest of I_i is at most $\frac{5}{6}s + 6$. □

COROLLARY 3.21. *Each update forest I_i contains at most a constant number of edges.*

PROOF. By Lemma 3.20, we have that the size of the internal forest is bounded. Since an update forest has at most 4 boundary edges, this bounds the size of I_i. □

We have now bounded the size of I_i to a constant. We now proceed to bound the size of D_i, the forest of deleted edges. We first bound the number of edges which was not a parent of an edge in D_{i-1} and we then use this to bound the size of D_i.

LEMMA 3.22. *At most 8 edges of D_i does not represent edges from D_{i-1}.*

PROOF. Each edge in D_i that is not a parent of a an edge in D_{i-1}, is either in C_B or is a neighbor to $I_{i-1} \cup C_B$. C_B contains at most 4 edges, and $I_{i-1} \cup C_B$ is an update forest, so it has at most 4 mergeable neighbors, which could get their parent deleted when merging with a boundary edge of $I_{i-1} \cup C_B$. □

As an important observation, notice that $\partial D_i = \partial I_i$. This is clear, since I_i reconnects the disconnected subtrees of F_i.

LEMMA 3.23. $|D_i| \leq \frac{5}{6}|D_{i-1}| + 12$

PROOF. By Lemma 3.9 any MRP for D_{i-1} has at most $\frac{5}{6}|D_{i-1}|$ clusters. In particular this holds for any MRP created by taking any MRP for F_{i-1}, removing all clusters containing edges not in D_{i-1}, and merging each orphan edge in D_{i-1} with its mergeable neighbor (if any). This MRP clearly has at most four clusters less than the number of edges in D_i containing edges from D_{i-1}. Thus the number of edges in D_i representing edges in D_{i-1} is at most $\frac{5}{6}|D_{i-1}|+4$, and by Lemma 3.22 at most 8 edges in D_i does not represent edges from D_{i-1}, thus the total number of edges in D_i is at most $\frac{5}{6}|D_{i-1}| + 12$. □

COROLLARY 3.24. *Each deletion tree D_i contains at most a constant number of edges.*

We have now bounded the size of I_i and D_i. In addition we have provided an algorithm to compute them. The simple algorithm of steps 1,2,3a-3c can then be implemented by using Corollary 3.8 using constant time per level.

COROLLARY 3.25. *After a* Link *operation, the multilevel partition can be updated by deleting and inserting a constant number of edges per level.*

Cut

Handling the cut operation is analogous to the link operation. The only difference is that step 2 is replaced by a corresponding step 2'.

2'. Delete parents of mergeable pairs of single edges, which have become mergeable as a result of replacing D_{i-1} with I_{i-1}.

The edges whose parents are deleted in step 2' are identical to the edges who had their parent deleted in step 2. Similarly these becomes the boundary edges of I_i and the rest of the proof is analogous, except that here, at each expansion node boundary edges may match each other.

COROLLARY 3.26. *After a cut operation, the multilevel partition can be updated by deleting and inserting a constant number of edges per level.*

Expose/Deexpose

Expose(v, w) is implemented by inserting a *special* leaf to each of the exposed nodes such that this special leaf can never be matched by its neighbors. Because of the definition of the MRP and the special edge, at some time, we may end up with a set of edges where no one can match another. See Figure 3.13.

As a special case, if this occurs, we allow the two leaf clusters to ignore the cyclic order and match (v, w) with a leaf neighbor. The Expose is complete when only the path-edge (v, w) and the two special leafs are left, and we return the path-edge (v, w).

Since this corresponds to linking each external boundary node of T to a tree T' of size $2|T|$ such that the special edge is always matched in T', the resulting Exposed top-tree has height $O(\log 5n) = O(\log n)$.

De-Expose is the operation of turning a boundary tree with external boundary nodes into a top-tree with no external boundary nodes. To De-Expose(v, w) we cut the special leafs.

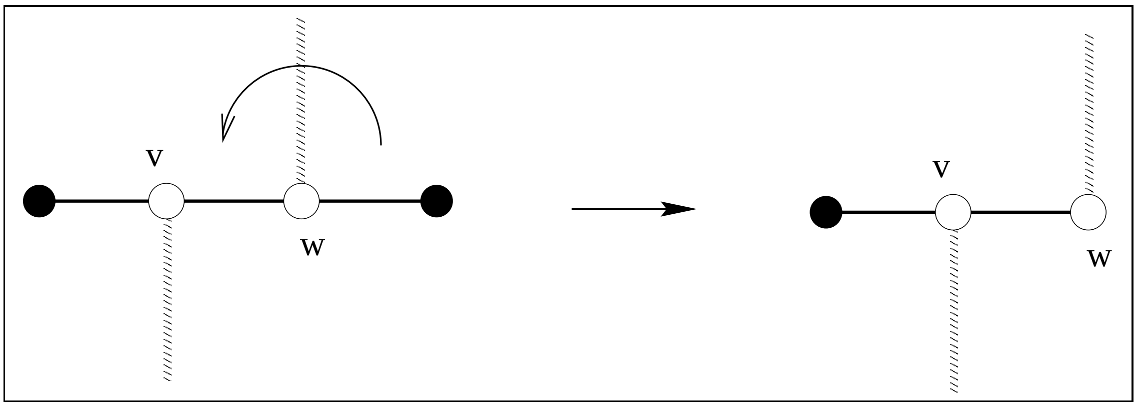

Figure 3.13: *When the* Expose*(v,w) reach the root, the leaf clusters, left and right, cannot merge with* (v, w) *because of the special leaves. If this happen, we merge past the special leaves (right side).*

We have not defined link, cut or Expose in top-trees with external boundary nodes, therefore, if a top-tree has external boundary nodes, we De-Expose it before updating.

COROLLARY 3.27. Expose *and* De-Expose *changes at most a constant number of edges per level.*

PROOF (OF THEOREM 3.1). We have shown (corollaries 3.25, 3.26, and 3.27) how to maintain a multilevel partition under Link, Cut, Expose, and De-Expose by deleting and creating at most a constant number of edges in each level. Whenever we delete an edge in the multilevel partition we know that all its ancestors will also be deleted, thus we can start by deleting them from the top. Recall from section 3.3.1 that whenever we create a composite edge we call Merge and when we delete a composite edge we call Split. Thus we use at most $O(\log n)$ Merge and Split operations to maintain the top-tree. In addition, we may have to use a constant number of Create and Eradicate in F_0.

Finally by Lemma 3.11) that we can build a top-tree in linear time plus a linear number of Create and Merge operations. □

Chapter 4

Applications

In this chapter we show how to use top-trees. We give $O(\log n)$ time fully dynamic tree-algorithms which supports queries about the diameter and Maxweight. In addition we provide a general tool to help searching for edges with global properties, and apply it to provide $O(\log n)$ time fully dynamic tree algorithms for the 1-center and 1-median problem.

Let $T = (V, E)$ be a tree with n nodes. For a subtree T' of T let $V(T')$ and $E(T')$ denote the nodes and edges of T' respectively. With each edge is associated a non-negative *weight*. For two nodes $v, w \in V$, let $v \cdots w$ denote the simple path from v to w in T. In this chapter, $\mathcal{A}, \mathcal{B}$ denotes mergeable clusters and $\mathcal{C} = \mathcal{A} \cup \mathcal{B}$ is their parent.

4.1 Maxweight

Let v, w be nodes in T. *Maxweight*(v, w) is then the maximum weight of an edge on $v \cdots w$. *Addpath*(Δ, v, w) adds Δ to all edges on $v \cdots w$. It is assumed that no weight becomes negative as a result of this. The ST-trees of Sleator and Tarjan [35] supported both of the above operations in $O(\log n)$ time. We now expand on Corollary 3.3 and show how to support both of these operations in $O(\log n)$ time using top-trees. This serves as a simple example of how to use Split; in chapters 8 and 9, the usage of Split is much more involved. This also serves as an example of how an ST-tree algorithm might be implemented in top-trees.

To implement *Addpath* we store Δ in a lazy fashion, that is for each cluster $\mathcal{C}$ we store $\Delta_{\mathcal{C}}$ a weight that should be added to all edges on $\pi(\mathcal{C})$. In addition we maintain $W_{\mathcal{C}}$ which is the maximum weight of an edge on $\pi(\mathcal{C})$ when only $\Delta_{\mathcal{C}'}$ for path-descendants of $\mathcal{C}$ are considered. Thus for any cluster $\mathcal{C}$ the maximum weight of an edge on $\pi(\mathcal{C})$ is $W_{\mathcal{C}}$ plus the sum of $\Delta_{\mathcal{C}'}$ over all proper path-ancestors $\mathcal{C}'$ of $\mathcal{C}$. In particular, if v, w is a pair of nodes, and $\mathcal{C} = \text{Expose}(v, w)$, then $\mathcal{C}$ has no proper path-ancestors and $W_{\mathcal{C}}$ is the correct maximum weight of an edge on $v \cdots w$. Maxweight and Addpath are then:

Maxweight(v, w)**:** Set $\mathcal{C} := \text{Expose}(v, w)$, return $W_{\mathcal{C}}$.

Addpath(Δ, v, w)**:** Set $\mathcal{C} := \text{Expose}(v, w)$, set $W_{\mathcal{C}} := W_{\mathcal{C}} + \Delta$ and $\Delta_{\mathcal{C}} := \Delta_{\mathcal{C}} + \Delta$.

To maintain this information we must ensure that $\Delta_{\mathcal{C}}$ is propagated down when $\mathcal{C}$ is deleted. The rest of the internal operations are straightforward.

Create$(\mathcal{C} : (v, w))$**:** Set $\Delta_{\mathcal{C}} := 0$. If $\mathcal{C}$ is a path-cluster, set $W_{\mathcal{C}} :=$ *weight*(v, w).

Merge$(\mathcal{C} : \mathcal{A}, \mathcal{B})$**:** $W_{\mathcal{C}} := \max\{W_{\mathcal{D}} | \mathcal{D} \in \{\mathcal{A}, \mathcal{B}\}$ is a path-child of $\mathcal{C}\}$. $\Delta_{\mathcal{C}} := 0$.

Split$(\mathcal{C} : \mathcal{A}, \mathcal{B})$**:** For each path-child $\mathcal{D}$ of $\mathcal{C}$ set $W_{\mathcal{D}} := W_{\mathcal{D}} + \Delta_{\mathcal{C}}$, $\Delta_{\mathcal{D}} := \Delta_{\mathcal{D}} + \Delta_{\mathcal{C}}$.

Eradicate($\mathcal{C} : (v, w)$**):** If $\mathcal{C}$ is a path-cluster, *weight*$(v, w) :=$ *weight*$(v, w) + \Delta_{\mathcal{C}}$;

LEMMA 4.1. *We can maintain a fully dynamic forest F and support queries about the maximum weight between any two nodes interspersed with path-updates in $O(\log n)$ time per operation.*

PROOF. The above procedures maintains $W_{\mathcal{C}}$ and $\Delta_{\mathcal{C}}$ in constant time. The Lemma is established by Theorem 3.1. $\square$

4.2 Diameters

We define the distance between two nodes v and w, denoted *dist*(v, w), to be the sum of the weights of all edges on the simple path from v to w. If v and w are not connected the distance is undefined. The *diameter* of a tree T is the maximum distance between any pair of nodes in T. In this section we show how to maintain the diameter of trees in a dynamic forest of trees. Let *diam*$(\mathcal{C})$ denote the diameter of a cluster $\mathcal{C}$. For a node v in $\mathcal{C}$, let *radius*$_{\mathcal{C}}(v)$ denote the maximum distance from v to any node in $\mathcal{C}$.

LEMMA 4.2. *diam*$(\mathcal{C})$ *is the maximum of diam*$(\mathcal{A})$, *diam*$(\mathcal{B})$ *and radius*$_{\mathcal{A}}(c)$ + *radius*$_{\mathcal{B}}(c)$, *where c is the common boundary node of $\mathcal{A}$ and $\mathcal{B}$.*

PROOF. Let $d_1, d_2 \in \mathcal{C}$ be nodes of $\mathcal{C}$ with maximum distance: $\mathit{dist}(d_1, d_2) = \mathit{diam}(\mathcal{C})$. If d_1, d_2 are both contained in the same child of $\mathcal{C}$, $\mathit{diam}(\mathcal{C})$ is the diameter of that child. Otherwise assume w.l.o.g. $d_1 \in \mathcal{A}$ and $d_2 \in \mathcal{B}$, then $c \in d_1 \cdots d_2$ and $\mathit{dist}(d_1, d_2) = \mathit{dist}(d_1, c) + \mathit{dist}(c, d_2) \leq \mathit{radius}_{\mathcal{A}}(c) + \mathit{radius}_{\mathcal{B}}(c)$. □

From the lemma it follows that we can maintain the diameter of a cluster, if we can maintain the radius of the boundary nodes. Computing *radius* for boundary nodes reduces (with symmetry) to

$$(4.1) \qquad \mathit{radius}_{\mathcal{C}}(a) \quad = \quad \max\{\mathit{radius}_{\mathcal{A}}(a), \mathit{dist}(a, c) + \mathit{radius}_{\mathcal{B}}(c)\}.$$

Thus to maintain the radius of the boundary nodes we need $\mathit{dist}(\partial\mathcal{C})$, the length of $\pi(\mathcal{C})$. For each cluster $\mathcal{C}$ we maintain:

- $\mathit{diam}(\mathcal{C})$.
- $\mathit{dist}(\partial\mathcal{C})$.
- $\mathit{radius}_{\mathcal{C}}(c')$, $c' \in \partial\mathcal{C}$.

From the above discussion the following internal operations correctly maintains the information. Split and Eradicate are not used in this algorithm and so do nothing:

Create$(\mathcal{C} : (v, w))$**:** Set $\mathit{diam}(\mathcal{C}) := \mathit{weight}(v, w)$. If $\mathcal{C}$ is a path-cluster, set $\mathit{dist}(\partial\mathcal{C}) := \mathit{weight}(v, w)$, otherwise set $\mathit{dist}(\partial\mathcal{C}) := 0$. For each $c' \in \partial\mathcal{C}$ set $\mathit{radius}_{\mathcal{C}}(c') := \mathit{weight}(v, w)$.

Merge($\mathcal{C}$**:**$\mathcal{A}$**,**$\mathcal{B}$**):** Set $\mathit{diam}(\mathcal{C}) := \max\{\mathit{diam}(\mathcal{A}), \mathit{diam}(\mathcal{B}), \mathit{radius}_{\mathcal{A}}(c) + \mathit{radius}_{\mathcal{B}}(c)\}$. Set $\mathit{dist}(\partial\mathcal{C}) := \sum \mathit{dist}(\partial\mathcal{C}')$, $\mathcal{C}' \in \{\mathcal{A}, \mathcal{B}\}$ is a path-child of $\mathcal{C}$. Compute $\mathit{radius}_{\mathcal{C}}(c'), c' \in \partial\mathcal{C}$ using equation (4.1).

THEOREM 4.3. *We can maintain a fully dynamic forest F and support queries about the diameter of trees and the distance between nodes in $O(\log n)$ time per operation.*

PROOF. By Lemma 4.2 the diameter is maintained in the root cluster of the tree. Both Merge and Create takes constant time. To answer the query $\mathit{dist}(v, w)$, set $\mathcal{C} := \text{Expose}(v, w)$, then $\mathit{dist}(v, w) = \mathit{dist}(\partial\mathcal{C})$. The time complexity is established by Theorem 3.1. □

4.3 Global search

In this section we present a black-box for searching **edges** with global properties. The black box is applied in sections 4.4 and 4.5 to maintain nodes with global properties. The black box works only for top-trees with no external boundary nodes.

Recall from section 4.1 how the maximum weight of an edge on a path was found. The Maxweight problem has the nice property, that if T is a tree, T' a subtree, then if $e \in E(T')$ is a solution in T, then e is a solution to T'. Global search was developed for problems that do not have this property, where the solution depends on the entire tree.

The search requires that given a root cluster $\mathcal{C}$ of a tree with children $\mathcal{A}$ and $\mathcal{B}$, the user must decide whether the desired edge is in $\mathcal{A}$ or in $\mathcal{B}$. The point in $\mathcal{C}$ being a root cluster is that it represents the whole tree, which is needed if we are considering global properties. In the style of a binary search, the black box is going to present us with at most $O(\log n)$ such decisions, and at the end tell us which edge we were looking for. More formally, beyond the Split and Merge procedures, the user needs to present a procedure Decide$(\mathcal{C} : \mathcal{A} \cup \mathcal{B})$ that given the root cluster $\mathcal{C}$ with children $\mathcal{A}$ and $\mathcal{B}$ tells whether the desired edge is in $\mathcal{A}$ or in $\mathcal{B}$. Possibly, Decide can also interrupt the search because the edge has been found by other means.

4.3.1 Implementation

Throughout the search we have a *selected* cluster, subject to the following invariant.

(i) The selected cluster contains the desired edge.

Starting from the root-cluster, in each step, a child of the selected cluster becomes the next selected cluster, thus the search ends after $O(\log n)$ steps with an edge-cluster. By (i) the edge-cluster must contain the desired edge.

Initially we use Decide to select one of the child-clusters of the root. This then becomes the first selected cluster and (i) is initially true. In each step, the selected cluster $\mathcal{D}$ and its ancestors are Split. Let $\mathcal{D}_1, \mathcal{D}_2$ denote the children of the selected cluster. The top-tree is then reassembled such that $\mathcal{D}_1$ and $\mathcal{D}_2$ are contained in distinct children of the root. Note that this is only possible when the top-tree has no external boundary nodes. We then use Decide to choose between these children and thus between $\mathcal{D}_1$ and $\mathcal{D}_2$. Assume w.l.o.g. Decide chose the child $\mathcal{A}$, containing $\mathcal{D}_1$. We then know the desired edge is in $\mathcal{D}_1$ since $\mathcal{D} \cap \mathcal{A} = \mathcal{D}_1$. $\mathcal{D}_1$ then becomes the next selected cluster, and (i) is maintained.

THEOREM 4.4 (GLOBAL SEARCH). *Given a top-tree with no external boundary nodes, after $O(\log n)$ calls to* Decide, Merge *and* Split, *there is a unique edge (v, w) contained in all clusters chosen by* Decide, *and then (v, w) is returned. Furthermore the top-tree is restored in its shape from before the search.*

PROOF. By the above discussion the search correctly finds the desired edge in $O(\log n)$ steps. What remains is to bound the number of Splits and Merges used in a step: Throughout the search, the selected cluster is either a child of the root, or a grand-child of the root of the current top-tree, thus we use at most 3 Splits per step, resulting in at most 4 clusters. It is always possible to remerge 4 clusters into 2, such that any given two clusters are not merged, but it may cause the height of the top-tree to grow by a constant. Thus we use at most 3 Merges in each step to reassemble the top-tree and through-out the search, its height is $O(\log n)$. To ensure that the height doesn't grow too large, when the search is complete we reassemble the top-tree as it was, using another 3 Splits and Merges per step. □

4.4 1-center by global search

In this section, we will show how to maintain 1-center nodes efficiently. The *1-center* of a tree is a minimum radius node, where the radius of a node v is defined as the maximum distance to any node in T. The median of a tree is a node minimizing minimizing $H_T(w) = \sum_{v \in V} \textit{weight}(v) \cdot \textit{dist}(v, w)$, where in addition to the non-negative weight associated with each edge, we have a non-negative weight associated with each node.

From Jordan, 1869, we have

THEOREM 4.5 ([30]). *Every tree has a center consisting of either one vertex or or two adjacent vertices; every tree has a median consisting of either one vertex or or two adjacent vertices.*

Both for the 1-center and the 1-median we will use Theorem 4.5 with the global search to find an edge containing the desired (pair of) nodes, and having found this edge, we just need one additional test to check if both endpoints are a center/median.

Suppose we had found an edge (v, w) containing the 1-center(s). Then to find out which of v and w is a 1-center, we maintain the radius of the boundary nodes as in section 4.2, and we set $\mathcal{C} := \mathrm{Expose}(v, w)$, and then return the one of v and w minimizing $\mathit{radius}_{\mathcal{C}}(\cdot)$ or both if $\mathit{radius}_{\mathcal{C}}(v) = \mathit{radius}_{\mathcal{C}}(w)$. Thus it suffices to find the edge (v, w).

Our Decide is based on the following lemma:

LEMMA 4.6. *Let T be a tree, let $\mathfrak{C}$ be a 1-center of T and let $\mathcal{A}$ and $\mathcal{B}$ be child clusters of the root cluster and $c \in \partial A \cap \partial B$. Then $\mathit{radius}_{\mathcal{A}}(c) \geq \mathit{radius}_{\mathcal{B}}(c) \Rightarrow \mathfrak{C} \in \mathcal{A}$*

PROOF. If $\mathit{radius}_{\mathcal{A}}(c) \geq \mathit{radius}_{\mathcal{B}}(c)$ then $\mathit{radius}_T(c) = \mathit{radius}_{\mathcal{A}}(c)$. For $b' \in V(\mathcal{B})$:

$$\begin{aligned} \mathit{radius}_T(b') &= \max\{\mathit{radius}_{\mathcal{B}}(b'), \mathit{dist}(b', c) + \mathit{radius}_{\mathcal{A}}(c)\} \\ &\geq \mathit{radius}_{\mathcal{A}}(c) = \mathit{radius}_T(c). \end{aligned}$$

We conclude $\mathfrak{C} \notin \mathcal{B} \setminus \{c\}$. □

Thus Decide selects from $\mathcal{A}$ and $\mathcal{B}$ the cluster in which their common boundary nodes has the greatest radius. By Lemma 4.6, no $\mathfrak{C}$ is in the other cluster.

THEOREM 4.7. *We can maintain a fully dynamic forest F and support queries about the 1-centers of trees in $O(\log n)$ time per operation.*

PROOF. The radius of the boundary nodes is maintained as in section 4.2. Decide takes constant time and the Theorem is established by Theorem 4.4. □

4.5 1-median by global search

The *1-median* problem in a tree $T(V, E)$ is finding a (pair of) nodes $\mathfrak{M}$ minimizing $H_T(w) = \sum_{v \in V} \mathit{weight}(v) \cdot \mathit{dist}(v, w)$, where in addition to the non-negative weight associated with each edge, we have a non-negative weight associated with each node. As before, we use Theorem 4.4 to find an edge containing the $\mathfrak{M}$(s), and then test if both endpoints apply.

Suppose the edge (v, w) contains $\mathfrak{M}$. To test which endpoints is a 1-median, we maintain $H_{\mathcal{C}}(\cdot)$ for each boundary node of a cluster and we set $\mathcal{C} = \mathrm{Expose}(v, w)$ and return the node minimizing $H_{\mathcal{C}}(\cdot)$ or both. Thus it suffices to find the edge (v, w).

In addition to linking and cutting edges we wish to be able to change the weight of a node. Changing the weight of a node v affects $H_{\mathcal{C}}(\cdot)$ in all the up to $\Theta(n)$ clusters in which v is a boundary node. Instead of doing this explicitly, for each boundary node $c \in \partial C$ we maintain $h_{\mathcal{C}}(c) = \sum_{v \in V(\mathcal{C}) \setminus \partial C} \mathit{dist}(c, v) \cdot \mathit{weight}(v)$, since changing the weight of a node v only affects $h_{\mathcal{C}}(\cdot)$ in the $O(\log n)$ clusters in which v is internal (not a boundary node). Furthermore if $\mathcal{C}$ has boundary nodes a, b then $H_{\mathcal{C}}(a) = h_{\mathcal{C}}(a) + \mathit{dist}(a, b) \cdot \mathit{weight}(b)$, thus $h_{\mathcal{C}}(\cdot)$ is sufficient to compute $H_{\mathcal{C}}(\cdot)$ and thus to find the 1-median nodes once (v, w) has been obtained.

For any tree T, let $W(T)$ denote the sum of node weights of T. To find an edge containing the 1-median we use the global-search provided in Theorem 4.4. Our Decide is based on this lemma below which follows from Goldman [18].

LEMMA 4.8 ([18]). *Let T be a tree, let $\mathfrak{M}$ be a 1-median of T and let $\mathcal{A}$ and $\mathcal{B}$ be child clusters of the root-cluster. Then $W(\mathcal{A}) \geq W(\mathcal{B}) \Rightarrow \mathfrak{M} \in \mathcal{A}$.*

Again we cannot maintain $W(\mathcal{C})$ but instead we maintain $w(\mathcal{C}) = \sum_{v \in V(\mathcal{C}) \setminus \partial C} \mathit{weight}(v)$ still allowing a $O(\log n)$ update to the node weight. Correspondingly $W(\mathcal{C}) = w(\mathcal{C}) + \sum_{v \in \partial C} \mathit{weight}(v)$.

To decide whether $\mathfrak{M}$ is in $\mathcal{A}$ or $\mathcal{B}$, compute $W(\mathcal{A})$ and $W(\mathcal{B})$, and use Lemma 4.8.

To maintain $h_{\mathcal{C}}(\cdot)$ we maintain the distance between boundary nodes as in section 4.2. Create is straight-forward, and Split and Eradicate are not used.

Merge($\mathcal{C}$:$\mathcal{A}$,$\mathcal{B}$): Set $w_{\mathcal{C}} := w_{\mathcal{A}} + w_{\mathcal{B}} + w(c'), c' \in (\partial\mathcal{A} \cup \partial\mathcal{B}) \setminus \partial\mathcal{C}$. If $\mathcal{A}$ and $\mathcal{B}$ are both leaf-clusters with boundary node c, $h_{\mathcal{C}}(c) := h_{\mathcal{A}}(c) + h_{\mathcal{B}}(c)$. Otherwise assume without loss of generality $\mathcal{A}$ is a path-cluster, $\partial\mathcal{A} = \{a, c\}$ and set $h_{\mathcal{C}}(a) := h_{\mathcal{A}}(a) + h_{\mathcal{B}}(c) + \mathit{dist}(a, c) \cdot w_{\mathcal{B}}(+\mathit{dist}(a, c) \cdot \mathit{weight}(c)$, if $c \notin \partial\mathcal{C})$. If $\mathcal{B}$ is a path-cluster, let $\partial\mathcal{B} = \{c, b\}$ and compute $h_{\mathcal{C}}(b)$ symmetrically, otherwise $b = c$ and $h_{\mathcal{C}}(c) := h_{\mathcal{A}}(c) + h_{\mathcal{B}}(c)$.

To implement Changeweight, we need to update the information for all the clusters where v is internal:

Changeweight(v, weight)**:** Set $\mathcal{C} :=$ Expose(v, w), $\mathit{weight}(v) := \mathit{weight}$, where w is an arbitrary node connected to v.

When v is an external boundary node, it is not internal to any cluster.

THEOREM 4.9. *We can maintain a fully dynamic forest F with* Changeweight *updates and support queries about the 1-median's of trees in $O(\log n)$ time per operation.*

PROOF. By Lemma 4.8 the search finds $\mathfrak{M}$. Merge, Create, Split and Decide all take constant time. The Theorem is established by Theorem 3.1. $\square$

Part II

Dynamic graph algorithms

Chapter 5

Introduction to part II

In this part we consider the fully dynamic graph problems of connectivity, minimum spanning forest, 2-edge connectivity and biconnectivity. These results were first published in [28]. In a *fully dynamic graph problem*, we are considering an undirected graph G over a fixed vertex set V, $|V| = n$. The graph G may be *updated* by insertions and deletions of edges. Unless otherwise stated, we assume that we start with an empty edge set.

For the *fully dynamic connectivity problem*, the updates may be interspersed with *connectivity queries*, asking whether two given vertices are connected in G. Both updates and queries are presented *on-line*, meaning that we have to respond to an update or query without knowing anything about the future.

The connectivity problem reduces to the problem of maintaining a spanning forest (a spanning tree for each component) in that if we can maintain *any* spanning forest F for G at cost $O(t(n) \log n)$ per update, then, using dynamic trees [35], we can answer connectivity queries in time $O(\log n / \log t(n))$. In chapter 6, we present a very simple deterministic algorithm for maintaining a spanning forest in a graph in amortized time $O(\log^2 n)$ per update. Connectivity queries are then answered in time $O(\log n / \log \log n)$.

In the *fully dynamic minimum spanning forest problem*, we have weights on the edges, and we wish to maintain a minimum weight spanning forest F of G. Thus, in connection with any update to G, we need to respond with the corresponding updates for F, if any. In chapter 7 we will present a deterministic algorithm for maintaining a minimum spanning forest in a graph in $O(\log^4 n)$ amortized time per operation.

A graph is *2-edge connected* if and only if it is connected and no single edge deletion disconnects it. The 2-edge connected components are the maximal 2-

edge connected subgraphs, and two vertices v and w are 2-edge connected if and only if they are in the same 2-edge connected component, or equivalently, if and only if v and w are connected by two edge-disjoint paths. A graph is *biconnected* if and only if it is connected and no single vertex deletion disconnects it. The biconnected components are the maximal biconnected subgraphs, and two vertices v and w are biconnected if and only if they are in the same biconnected component, or equivalently, if and only if either (v, w) is an edge or v and w are connected by two internally disjoint paths. In the *fully dynamic 2-edge and biconnectivity problems*, the updates may be interspersed with queries asking whether two given vertices are 2-edge or biconnected. In chapters 8 and 9 we present $O(\log^4 n)$ algorithms for these two problems.

5.1 Previous work

For deterministic algorithms, all the previous best solutions to the fully dynamic connectivity problem were also solutions to the minimum spanning tree problem. In 1985 [10], Frederickson introduced a data structure known as *topology trees* for the fully dynamic minimum spanning tree problem with a worst case cost of $O(\sqrt{m})$ per update, permitting connectivity queries in time $O(\log n/\log(\sqrt{m}/\log n)) = O(1)$. In 1992, Epstein et. al. [7] improved the update time to $O(\sqrt{n})$ using the *sparsification technique*. Finally in 1997 Henzinger and King [25] gave an algorithm with $O(\sqrt[3]{n}\log n)$ update time and constant time per connectivity query.

Randomization has been used to improve the bounds for the connectivity problem. In 1995 [22], Henzinger and King showed that a spanning forest could be maintained in $O(\log^3 n)$ expected amortized time per update. Then connectivity queries are supported in $O(\log n/\log\log n)$ time. The update time was further improved to $O(\log^2 n)$ in 1996 [27] by Henzinger and Thorup. No randomized technique was known for improving the deterministic $O(\sqrt[3]{n}\log n)$ update cost for the minimum spanning tree problem.

In 1991 [11], Frederickson succeeded in generalizing his $O(\sqrt{m})$ bound from 1983 [10] for fully dynamic connectivity to fully dynamic 2-edge connectivity. As for connectivity, the sparsification technique of Eppstein et.al. [7] improved this bound to $O(\sqrt{n})$. Further, Henzinger and King generalized their randomization technique for connectivity to give an $O(\log^5 n)$ expected amortized bound [22, 23]. It should be noted that the above mentioned improvement for connectivity of Henzinger and Thorup [27], does not affect the $O(\log^5 n)$ bound for 2-edge connectivity.

For biconnectivity, the previous results are a lot worse. The first non-trivial result was a deterministic bound of $O(m^{2/3})$ from 1992 by Rauch [20]. In 1994 [33], Rauch improved this bound to $O(\min\{\sqrt{m}\log n, n\})$. In 1995, (Rauch) Henzinger and Poutré further improved the deterministic bound to $O(\sqrt{n\log n}\log\lceil m/n\rceil)$ [26]. In 1995 [21], Henzinger and King generalized their randomized algorithm from [22] to the biconnectivity problem to achieve an $O(\Delta\log^4 n)$ expected amortized cost per operation, where Δ is the maximal degree (In [21], the bound is incorrectly quoted as $O(\log^4 n)$ [Henzinger, personal communication, 1997]).

Finally, we note that for all of the above problems, we have a lower bound of $\Omega(\log n/\log\log n)$ which was proved independently by Fredman and Henzinger [16] and Miltersen, Subramanian, Vitter, and Tamassia [31]. Figure 5.1 contains an overview over the previous results together with our own.

For the incremental (no deletions) and decremental (no insertions) problems, the bounds are as follows. Incremental connectivity is the union-find problem, for which Tarjan has provided an $O(\alpha(m,n))$ bound [36]. Westbrook and Tarjan have obtained the same time bound for incremental 2-edge and biconnectivity [38]. Further Sleator and Tarjan have provided an $O(\log n)$ bound [35] for incremental minimum spanning forest.

Decrementally, for connectivity and 2-edge connectivity, Thorup has provided an $O(\log n)$ bound if we start with $\Omega(n\log^6 n)$ edges, and an $O(1)$ bound if we start with $\Omega(n^2)$ edges [37]. For decremental minimum spanning tree and biconnectivity, no better bounds were known than those for the fully dynamic case.

5.2 Our results

First we present a very simple deterministic fully dynamic connectivity algorithm with an update cost of $O(\log^2 n)$, thus matching the previous best randomized bound and improving substantially over the previous best deterministic bound of $O(\sqrt[3]{n}\log n)$.

Our technique relies on some of the same intuition as was used in Henzinger and King [22] in their randomized algorithm. Our deterministic algorithm is, however, much simpler, and in contrast to their algorithm, it generalizes to the minimum spanning tree problem. More precisely, a specialization of our connectivity algorithm gives a simple decremental minimum spanning tree with an amortized cost of $O(\log^2 n)$ per operation for any sequence of $\Omega(m)$ operations. Then we use

	Connectivity	MSF	2-edge Connectivity	Bi-Connectivity
Fre'85	$O(\sqrt{m})$	$O(\sqrt{m})$	-	-
Fre'91	-	-	$O(\sqrt{m})$	-
EGIN'92	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	-
Rau'92	-	-	-	$O(m^{2/3})$
Rau'94	-	-	-	$\tilde{O}(\sqrt{m})$
HK'95	$\boxed{O(\log^3 n)}$	-	$\boxed{O(\log^5 n)}$	$\boxed{\tilde{O}(\Delta)}$
HP'95	-	-	-	$\tilde{O}(\sqrt{n})$
HT'96	$\boxed{O(\log^2 n)}$	-	-	-
HK'97	$\tilde{O}(\sqrt[3]{n})$	$\tilde{O}(\sqrt[3]{n})$	-	-
Now	$O(\log^2 n)$	$O(\log^4 n)$	$O(\log^4 n)$	$O(\log^4 n)$

$\tilde{O}(f(n)) = O(f(n) \operatorname{polylog}(n))$.

$\boxed{\text{Boxed}}$ means randomized complexity.

Δ is the maximal degree of a vertex.

Figure 5.1: Overview over previous and current results

a technique from [25] to convert our deletions-only structure to a fully dynamic data structure for the minimum spanning tree problem using $O(\log^4 n)$ amortized time per update. This is the first polylogarithmic bound for the problem, even when we include randomized algorithms.

Finally, our connectivity techniques are generalized to 2-edge and biconnectivity, leading to $O(\log^4 n)$ algorithms for both of these problems. The generalization uses some of the ideas from [11, 21, 23] of organizing information around a spanning forest. However, finding a generalization that worked was rather delicate, particularly for biconnectivity, where we needed to make a careful recycling of information, leading to the first polylogarithmic algorithm for this problem.

The reader is referred to [5, 7, 8, 10, 11, 22] for discussions of other problems that get improved bounds by our new fully dynamic graph algorithms.

Chapter 6

Connectivity

In this chapter, we present a simple $O(\log^2 n)$ time deterministic fully dynamic algorithm for graph connectivity. First we give a high level description, ignoring all problems concerning data structures. Second, we implement the algorithm with concrete data structures and analyze the running times.

6.1 High level description

Our dynamic algorithm maintains a spanning forest F of a graph G. The edges in F will be referred to as *tree-edges*, while the edges in $G \setminus F$ is referred to as *non-tree edges*. Internally, the algorithm associates with each edge e a level $\ell(e) \leq L = \lfloor \log_2 n \rfloor$. For each i, F_i denotes the sub-forest of F induced by edges of level at least i. Thus, $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$. The following invariants are maintained.

(i) The maximal number of connected nodes in a tree in F_i is $\lfloor n/2^i \rfloor$. Thus, the maximal relevant level is L.

(ii) F is a maximum (w.r.t. ℓ) spanning forest of G, that is, if (v, w) is a non-tree edge, v and w are connected in $F_{\ell(v,w)}$.

Initially, all edges have level 0, and hence both invariants are satisfied. We are going to present an amortization argument based on increasing the levels of edges. The levels of edges are never decreased, so we can have at most L increases per edge. Intuitively speaking, when the level of a non-tree edge is increased, it is because we have discovered that its end points are close enough in F to fit in a smaller tree on a higher level. Concerning tree edges, note that increasing their level cannot violate (ii), but it may violate (i).

We are now ready for a high-level description of insert and delete.

Insert(e)**:** The new edge is given level 0. If the end-points were not connected in $F = F_0$, e is added to F_0.

Delete(e)**:** If e is not a tree-edge, it is simply deleted. If e is a tree-edge, it is deleted and a *replacement edge*, reconnecting F at the highest possible level, is searched for. Since F was a maximum spanning forest, we know that the replacement edge has to be of level at most $\ell(e)$. We now call Replace$(e, \ell(e))$. Note that when a tree-edge e is deleted, F may no longer be spanning, in which case (ii) is violated until we have found a replacement edge. In the time in between, if (v, w) is not a replacement edge, we still have that v and w are connected in $F_{\ell(v,w)}$.

Replace$((v, w), i)$**:** Assuming that there is no replacement edge on level $> i$, finds a replacement edge of the highest level $\leq i$, if any.

Let T_v and T_w be the trees in F_i containing v and w, respectively. Assume, without loss of generality, that $|T_v| \leq |T_w|$. Before deleting (v, w), $T = T_v \cup \{(v, w)\} \cup T_w$ was a tree on level i with at least twice as many nodes as T_v. By (i), T had at most $\lfloor n/2^i \rfloor$ nodes, so now T_v has at most $\lfloor n/2^{i+1} \rfloor$ nodes. Hence, preserving our invariants, we can take all edges of T_v of level i and increase their level to $i + 1$, so as to make T_v a tree in F_{i+1}.

Now level i edges incident to T_v are visited one by one until either a replacement edge is found, or all edges have been considered. Let f be an edge visited during the search.

If f does not connect T_v and T_w, both endpoints of f are in the same tree, either T_v or T_w, and we increase its level to $i + 1$. This increase pays for our considering f.

If f does connect T_v and T_w, it is inserted as a replacement edge and the search stops.

If there are no level i edges left, we call Replace$((v, w), i - 1)$; except if $i = 0$, in which case we conclude that there is no replacement edge for (v, w).

Figure 6.1 contains a graph and snapshot of how it might look in the connectivity data structure.

6.2 Implementation

For each i, we wish to maintain the forest F_i together with all non-tree edges on level i. For any vertex v, we wish to be able to:

- Identify the tree T_v in F_i containing v.
- Compute the size of T_v.

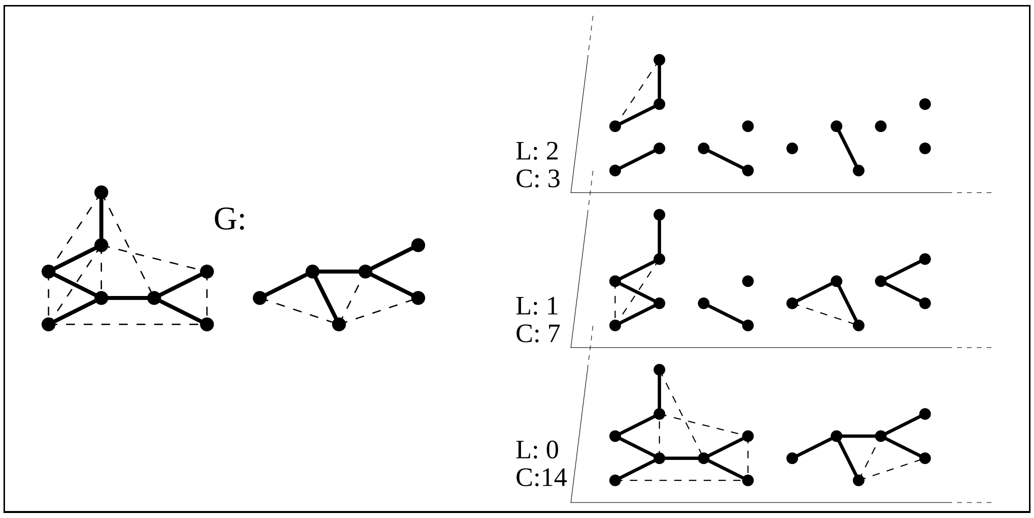

Figure 6.1: *An example graph (left), and how it might look in the connectivity data structure (right). Full lines are tree edges, dashed lines are non-tree edges. In the data structure, L denotes the level, $c = \lfloor 14/2^L \rfloor$ is the maximum number of nodes allowed in each connected component of level L.*

- Find an edge of T_v on level i, if one exists.
- Find a level i non-tree edge incident to T_v, if any.

The trees in F_i may be cut (when an edge is deleted) and linked (when a replacement edge is found, an edge is inserted or the level of a tree edge is increased). Moreover, non-tree edges may be introduced and any edge may disappear on level i (when the level of an edge is increased or when non-tree edges are inserted or deleted).

All the above operations and queries may be supported using the ET-trees from [9, 22], to which the reader is referred for additional details. An ET-tree is a standard balanced binary tree over the Euler tour of a tree. Each node in the ET-tree represents the segment of the Euler tour below it. The point in considering Euler tours is that if trees in a forest are linked or cut, the new Euler tours can be constructed by at most 2 splits and 2 concatenations of the original Euler tours. Rebalancing the ET-trees affects only $O(\log n)$ nodes. Top-trees also supports these operations in $O(\log n)$ time, but ET-trees are much more simple to use.

Here we have an ET-tree over each tree in F_i. Each node of the ET-tree contains a number telling the size of the Euler tour segment below it, a bit telling if any tree edges in the segment have level i, and a bit telling whether there is any level i non-tree edges incident to a vertex in the segment.

Given a vertex v we can find the tree T_v containing v by moving $O(\log n)$ steps up till we find a root of an ET-tree. This root represents the Euler tour of T_v. The size s of the Euler tour of a tree is twice the number of edges, so the number of vertices is $s/2 + 1$. To find a tree edge of level i or an incident non-tree edge, if any, we move $O(\log n)$ steps down the ET-tree, using the bits telling us under which nodes such edges are to be found. If a tree edge (v, w) is moved from level i, we only need to update the bits on the paths from (v, w) and (w, v) to the root, using $O(\log n)$ time. If a non-tree edge (v, w) is introduced/disappear, we only need to update the bits on the paths from v and w to their respective roots. This takes $O(\log n)$ time. When the trees are cut or linked, only $O(\log n)$ nodes are affected, and the information in each node is updated in constant time.

It is now straightforward to analyze the amortized cost of the different operations. When an edge e is inserted on level 0, the direct cost is $O(\log n)$. However, its level may increase $O(\log n)$ times, so the amortized cost is $O(\log^2 n)$.

Deleting a non-tree edge e takes time $O(\log n)$. When a tree edge e is deleted, we have to cut all forests F_j, $j \leq \ell(e)$, giving an immediate cost of $O(\log^2 n)$. We then have $O(\log n)$ recursive calls to Replace, each of cost $O(\log n)$ plus the cost amortized over increases of edge levels. Finally, if a replacement edge is found, we have to link $O(\log n)$ forests, in $O(\log^2 n)$ total time.

Thus, the cost of inserting and deleting edges from G is $O(\log^2 n)$. The balanced binary tree over $F_0 = F$ immediately allows us to answer connectivity queries between arbitrary nodes in time $O(\log n)$. In order to reduce this time to $O(\log n / \log\log n)$, as in [22], we introduce an extra balanced $\Theta(\log n)$-ary B-tree over the Euler tour of each tree in F. The B-tree has depth $O(\log n / \log\log n)$, which is hence the time it takes for a connectivity query. Each delete or insert gives rise to at most one cut and one link in F, and for $\Theta(\log n)$-ary B-trees, such operations can be supported in $O(\log^2 n / \log\log n)$ time. Thus, we conclude:

THEOREM 6.1. *Given a graph G with m edges and n vertices, there exists a deterministic fully dynamic algorithm that answers connectivity queries in $O(\log n / \log\log n)$ time worst case, and uses $O(\log^2 n)$ amortized time per* Link *or* Cut.

Curiously, the $O(\log^2 n)$ amortized time-bound is completely independent of the number of edges in G.

Chapter 7

Minimum spanning forest

We will now expand on the ideas from the previous chapter to the problem of maintaining a minimum spanning forest (MSF). First we present an $O(\log^2 n)$ deletions-only algorithm, and then we apply a general construction from [25] transforming a deletions-only MSF algorithm into a fully dynamic MSF algorithm.

7.1 Decremental minimum spanning forest

It turns out that if we only want to support deletions, we can obtain an MSF-algorithm from our connectivity algorithm by some very simple changes. The first is, of course, the initial spanning forest F has to be a minimal spanning forest. The second is that when in replace (cf. page 49). we consider the level i non-tree edges incident to T_v, instead of doing it in an arbitrary order, we should do it in order of increasing weights. That is, we repeatedly take the lightest incident level i edge e: if e is a replacement edge, we are done; otherwise, we move e to level $i+1$, and repeat with the new lightest incident level i edge, if any. For the above changes to work, it is crucial, that *all weights are distinct*. To ensure this, we associate a unique number with each edge. If two edges have the same weight, it is the one with the smaller number that is the smaller.

To see that the above simple changes suffice to maintain that F is a minimum spanning forest, we will prove that in addition to (i) and (ii), the following invariant is maintained:

(iii): If e is the heaviest edge on a cycle C, then e has the lowest level on C.

The original replace function found a replacement edge on the highest possible level, but now, among the replacement edges on the highest possible level, we choose the one of minimum weight. Using (iii), we will show that this edge has minimum weight among all replacement edges.

LEMMA 7.1. *For any tree edge e, among all replacement edges, the lightest edge is on the maximum level.*

PROOF. Let e_1 and e_2 be replacement edges for e. Let C_i be the cycle induced by e_i; then $e \in C_i$. Suppose e_1 is lighter than e_2. We want to show that $\ell(e_1) \geq \ell(e_2)$.

Consider the cycle $C = (C_1 \cup C_2) \setminus (C_1 \cap C_2)$. Since F is a minimum spanning forest, we know that e_i is the heaviest edge on C_i. Hence e_2 is the heaviest edge on C. By (iii) this implies that e_2 has the lowest level on C. In particular, $\ell(e_1) \geq \ell(e_2)$. □

Since our algorithm is just a specialized version of the decremental connectivity algorithm, we already know that (i) and (ii) are maintained.

LEMMA 7.2. *(iii) is maintained.*

PROOF. Initially (iii) is satisfied since all edges are on level 0. We will now show that (iii) is maintained under all the different changes we make to our structure during the deletion of an edge. If an edge e is just deleted, any cycle in $G \setminus \{e\}$ also existed in G, so (iii) is trivially preserved. Also note that replacing a deleted tree-edge cannot in itself violate (iii) since it does not change the levels or weights of any edges.

Our real problem is to show that (iii) is preserved during Replace when the level of an edge e is increased. This cannot violate (iii) if e is not the heaviest edge on some cycle, so assume that e is the heaviest edge on a cycle C. To prove that (iii) is not violated, we want to show that before the increase, all other edges in C have level $\geq i+1$.

No tree edge is heaviest on any cycle, so e is a non-tree edge. When $\ell(e)$ is to be increased from i to $i+1$, we know it is the lightest level i edge incident to T_v (cf. the description of replace on page 49). Moreover, by (iii), all other edges on C have level at least i. Thus, all other edges from C incident to T_v have level at least $i+1$.

To complete the proof, we show that all edges in C are incident to T_v. Suppose, for a contradiction, that C contained an edge f leaving T_v. Since e is to be increased, $e \neq f$. Also, the call to Replace requires that there is no replacement edge of level $> i$, so $\ell(f) \leq i$. This contradicts that all edges $\neq e$ from C incident to T_v have level $\geq i+1$. □

It has now been established that the above change in replace suffice to maintain a minimum spanning forest. A last point is that we need to modify our ET-trees to give us the lightest non-tree edge incident to a tree. So far, for each node in the ET-trees, we had a bit telling us whether the Euler tour segment below it had an incident non-tree edge. Now, with the node, we store the minimum weight of a non-tree edge incident to the Euler tour segment below it. Clearly, we can still support the different operations in $O(\log n)$ time. We conclude

THEOREM 7.3. *There exists a deletions-only MSF algorithm that can be initialized on a graph with n nodes and m edges and support any sequence of $\Omega(m)$ deletions in total time $O(m \log^2 n)$.*

7.2 Fully dynamic minimum spanning forest

To obtain a fully dynamic minimum spanning forest algorithm we apply a general reduction, which is a slight generalization of the one provided by Henzinger and King [25, pp. 600-603]. The reduction, proven later, is described as follows.

THEOREM 7.4. *Suppose we have a deletions-only MSF algorithm that for any k, l, can be initialized on a graph with k nodes and l edges and support any sequence of $\Omega(l)$ deletions in total time $O(l \cdot t(k,l))$ where t is non-decreasing. Then there exists a fully-dynamic MSF algorithm for a graph on n nodes starting with no edges, that for m edges, supports an update in amortized time*

$$O\left(\log^3 n + \sum_{i=1}^{\log_2 m} \sum_{j=1}^{i} t\left(\min\{n, 2^j\}, 2^j\right)\right).$$

Applying the Theorem to our deletions-only algorithm:

THEOREM 7.5. *There is a fully-dynamic MSF algorithm that for a graph with n nodes and starting with no edges maintains a minimum spanning forest in $O(\log^4 n)$ amortized time per edge insertion or deletion.*

PROOF. From Theorem 7.3, we get $t(k,l) = O(\log^2 k)$, and hence, Theorem 7.4, we get a fully dynamic algorithm with update cost

$$O\left(\log^3 n + \sum_{i=1}^{\log_2 m} \sum_{j=1}^{i} \log^2\left(\min\{n, 2^j\}\right)\right) = O\left(\log^4 n\right). \qquad \square$$

Note for comparison, that in [25], Henzinger and King had $t(k,l) = O(\sqrt[3]{l}\log k)$, giving them an update cost of $O(\sqrt[3]{m}\log n)$. Then sparsification [7] reduces the cost to $O(\sqrt[3]{n}\log n)$.

7.2.1 High level description

We will support insertions via a logarithmic number of decremental MSF-structures. When an edge is deleted, it will be deleted from all the decremental structures, and a replacement edge will be sought among the replacement edges returned by these. When an edge is inserted, we will union it with some of the decremental structures into a new decremental structure.

More precisely, besides maintaining a minimum spanning forest F of G, we will maintain a set $A = \{A_0, \ldots, A_s\}$, $s = \log_2 m$, of subgraphs of G, and for each A_i, we will maintain a minimum spanning forest F_i. All edges of G will be in at least one A_i, so $F \subseteq \bigcup_i F_i$. Further, we have the following invariant:

(iv) For each edge $f \in G \setminus F$, there is at exactly one i such that $f \in A_i \setminus F_i$, and if $f \in F_j$, $j > i$.

LEMMA 7.6. *If f is the lightest replacement edge for a tree edge e in G, then f is the lightest replacement edge for e in at least one A_i.*

PROOF. Since f is not in F yet, there is an i such that $f \in A_i \setminus F_i$. When e has been deleted from G, there is no path between the end-points of f with edges lighter than f. Hence, there cannot such a path in $A_i \subseteq G$, so f has to be in F_i after e has been deleted. □

Thus, when a tree edge is deleted, we delete it from all A_i. If e was a tree edge of A_i and not a bridge, we get a replacement edge f_i. If e was a tree edge of G and not a bridge, we just check if one of the at most $s = \log_2 m$ f_i is a replacement edge for e in G. Our problem now is to reinsert all the replacement edges e_i into A so as to restore (iv). The replacement edges are inserted one by one following the scheme below.

Inserting edges When an edge e is to be inserted in A, we first find the smallest i such that $\sum_{j \le i} |E(A_j \setminus F_j)| < 2^i$. Then we set

$$A_i := F \cup \{e\} \bigcup_{j \le i} (A_j \setminus F_j), \tag{7.1}$$

and afterwards, we set $A_j := \emptyset$ for all $j < i$. Clearly, this restores (iv), and the number of non-tree edges in the resulting graph A_i is at most $1+\sum_{j<i} 2^j \leq 2^i$ and at least $2^{i-1}+1$, thus $s = \log_2 m$, as desired. Since $\sum_{j<i} |E(A_j \setminus F_j)| \geq 2^{i-1}$ we let each of the non-tree edges of $A_j, j < i$ pay for the reinitialization of a non-tree edge of A_i. We can now bound the number of times a non-tree edge must pay for its initialization in (7.1).

LEMMA 7.7. *Before it is deleted, a non-tree edge must pay for its initialization at most once for each $i = 0, \ldots, s$ and $j = 0, \ldots, i$.*

PROOF. When a non-tree edge is deleted, we remove at most one copy of it from each A_i. By (iv) the copy in A_i can have been used at most once as a non-tree edge to initialize each A_j, $j \leq i$. □

At present the number of initializations of tree edges is not bounded. To resolve this, we will only maintain F implicitly. Instead of adding F to A_i, we will add a forest F' of "super edges", each representing a path in F. The super edges are chosen to be the minimal maximally connected set of super edges containing all endpoints of non-tree edges, and the super edge $e_P = (v, w)$ in F' represents the path $P = v \cdots w$, and has the maximum weight on P. When an edge $e \in F$ is deleted, we delete all super edges e_P with $e \in P$. Note that there can be at most one such edge for each A_i. When creating A_i with super edges, we get at most 3 super edges for every non-tree edge. Hence the total number of edge initializations is at most 3 times the number of non-tree edge initializations. Thus, from Lemma 7.7, we get:

LEMMA 7.8. *When an edge is deleted it has for each $i = 0, \ldots, s$ and $j = 0, \ldots, i$ payed for the initialization of at most 3 super-edges and 2 non-tree edges into A_j.*

7.2.2 Implementation

In order to complete the reduction, we need to show how to identify the super edges when A_i is created, and how to identify the super edges e_P with $e \in P$, when an edge is deleted. To check whether any of the replacement edges f_i is the real replacement edge we use the top-tree based algorithm of Corollary 3.3. This takes $O(\log n)$ time per f_i.

For each A_i we maintain a copy of F, F^i. To identify the super-edges when A_i is created, we take the endpoints of the non-tree edges one at a time, incrementally

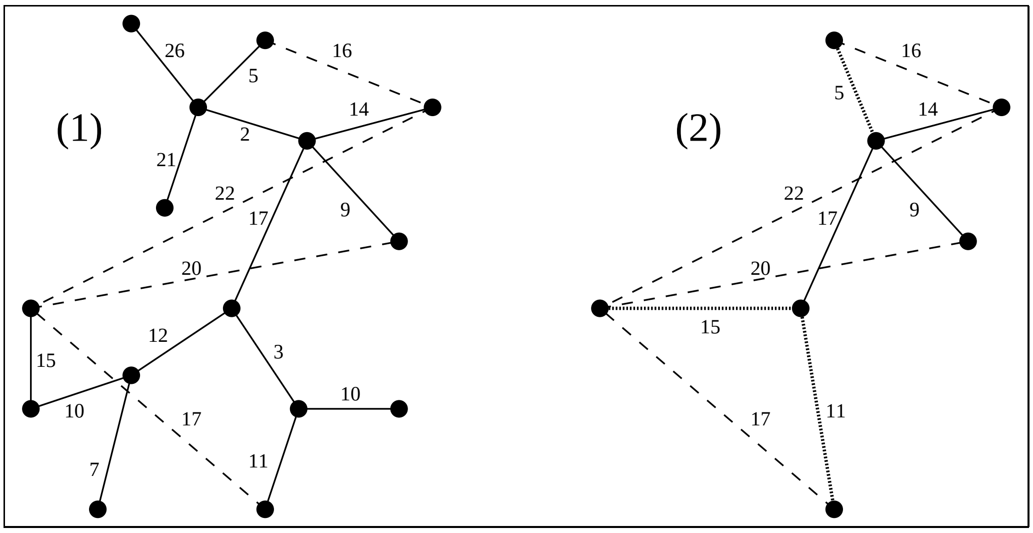

Figure 7.1: *In (1) is a tree with non-tree edges (dashed lines). In (2) the same tree with non-tree edges, after super-paths consisting of more than one edge (dotted) has been contracted.*

adding nodes to the tree of super-edges. The paths between these nodes in F^i is then the paths to be contracted to super-edges. In each step of the algorithm these *super*-paths corresponds to the paths to be contracted into super-edges if no more endpoints are added. To be more precise, we take each non-tree edge endpoint v in turn and mark it: if it is not connected to another marked node, we do nothing. Otherwise we find the nearest node which is either marked, or on the path between two marked nodes and mark it. Let v be an endpoint, z the found node on the super-path $a \cdots b$. Then the super-path $a \cdots b$ is split into $a \cdots z$ and $z \cdots b$ and in addition the new super-path $z \cdots v$ is created. If $z = a$ or $z = b$ no paths needs to be split. When there are no more non-tree edge endpoints to take, we have found all the super-edges. To implement this we need to be able to find the nearest node on a super-path, and which super-path it was on. Note that if the nearest node is on more than one super-path, then it must be marked.

We use top-trees for the implementation. For each tree we store which node is the first marked and call it r for root.

When a new endpoint v is introduced, we first check if it is connected to another marked node. If it is not, we set $r := v$ making v the root of the tree. If v is connected to another marked node we examine the path from v to r. The nearest node on a super-path, z must be on the path to the root. Thus for each cluster $\mathcal{C}$ we maintain for each boundary node $c' \in \partial\mathcal{C}$, $\text{NSPnode}_{\mathcal{C}}(c')$: the nearest node on the cluster-path which is on a super-path containing an edge from $\pi(\mathcal{C})$, and $\text{NSP}_{\mathcal{C}}(c')$: which super-path was it. To find z we just set $\mathcal{C} = \text{Expose}(v, r)$ and $\text{NSPnode}_{\mathcal{C}}(v)$ is the desired node and $\text{NSP}_{\mathcal{C}}(v)$ is the desired super-path. Furthermore, an edge (v, w) is contained in a super-path if $\mathcal{C} = \text{Expose}(v, w)$ contains a super-path $\text{NSP}_{\mathcal{C}}(v) = \text{NSP}_{\mathcal{C}}(w)$. Thus this information is sufficient to determine which super-edge contains (v, w) in its path.

To maintain super-paths we mark the edges of the paths in a lazy fashion, storing what information should be propagated down if the cluster was deleted: Since the lazy information can only be affected by path-ancestors, it is always correct for the root cluster and the leaf-clusters. For each cluster $\mathcal{C}$ the lazy information is:

- $c^+_{\mathcal{C}}$ and $c^-_{\mathcal{C}}$ meaning all edges $e \in \pi(\mathcal{C})$ are on the same super-path if $c^+_{\mathcal{C}} = 1$ and no edge on $\pi(\mathcal{C})$ is if $c^-_{\mathcal{C}} = 1$. If $c^+_{\mathcal{C}} = 1$, $p_{\mathcal{C}}$ is the path. If $c^+_{\mathcal{C}} = c^-_{\mathcal{C}} = -1$, the lazy information has no effect.

To mark and unmark the edges on a cluster-path:

Mark($\mathcal{C}$,(a,b)): Set $c^+_{\mathcal{C}} := 1, c^-_{\mathcal{C}} := -1, p_{\mathcal{C}} := (a, b)$ and for $c' \in \partial\mathcal{C}$ set $\text{NSPnode}_{\mathcal{C}}(c') := c', \text{NSP}_{\mathcal{C}}(c') := (a, b)$.

Unmark($\mathcal{C}$): Set $c^+_{\mathcal{C}} := -1, c^-_{\mathcal{C}} := 1, p_{\mathcal{C}} :=$ **nil** and for $c' \in \partial\mathcal{C}$ set $\text{NSPnode}_{\mathcal{C}}(c') :=$ **nil**, $\text{NSP}_{\mathcal{C}}(c') :=$ **nil**.

To mark or unmark a path $v \cdots w$, set $\mathcal{C} = \text{Expose}(v, w)$ and apply the desired procedure on $\mathcal{C}$.

Create and Eradicate are simple and therefore omitted.

Merge($\mathcal{C} : \mathcal{A}, \mathcal{B}$)**:** If $\mathcal{C}$ has no path-children, set $c^+_{\mathcal{C}} := -1, c^-_{\mathcal{C}} := -1, p_{\mathcal{C}} :=$ **nil**.
If $\mathcal{C}$ has only one path-child, $\mathcal{A}$, copy all information from $\mathcal{A}$ to $\mathcal{C}$.
If $\mathcal{C}$ has two path-children, let $\partial\mathcal{A} = \{a, c\}$ and $\partial\mathcal{C} = \{c, b\}$. Set $c^+_{\mathcal{C}} := -1, c^-_{\mathcal{C}} := -1, p_{\mathcal{C}} :=$ **nil**. If $\text{NSPnode}_{\mathcal{A}}(a) \neq$ **nil**, set $\text{NSPnode}_{\mathcal{C}}(a) := \text{NSPnode}_{\mathcal{A}}(a)$, $\text{NSP}_{\mathcal{C}}(a) := \text{NSP}_{\mathcal{A}}(a)$, otherwise $\text{NSPnode}_{\mathcal{A}}(a) =$ **nil** and we set $\text{NSPnode}_{\mathcal{C}}(a) := \text{NSPnode}_{\mathcal{B}}(c)$, $\text{NSP}_{\mathcal{C}}(a) := \text{NSP}_{\mathcal{B}}(c)$. $\text{NSPnode}_{\mathcal{C}}(b)$ and $\text{NSP}_{\mathcal{C}}(b)$ is computed symmetrically.

Split($\mathcal{C} : \mathcal{A}, \mathcal{B}$)**:** For each path-child $\mathcal{A}'$ of $\mathcal{C}$: if $c^+_{\mathcal{C}} = 1$, call mark($\mathcal{A}'$,$p_{\mathcal{C}}$), if $c^-_{\mathcal{C}} = 1$, call unmark($\mathcal{A}'$).

LEMMA 7.9. *The set of super-edges can be found from l non-tree edges in $O(\min(n, l) \log n)$ time. Furthermore to identify, insert or delete a super-edge takes $O(\log n)$ time.*

PROOF. Merge and Split takes constant time. Thus, by Theorem 3.1 Expose takes $O(\log n)$ time. To insert or delete a super-edge (v, w) set $\mathcal{C} = \text{Expose}(v, w)$ and call Mark or Unmark on $\mathcal{C}$. □

PROOF (OF THEOREM 7.4). By Lemma 7.9 we use $O(\log n)$ time per non-tree edge and super-edge per initialization. By the assumption of the Theorem and Lemma 7.8, when all edges are deleted we can have used at most

$$O\left(\sum_{i=0}^{\log_2 m} \sum_{j=1}^{j<i} \left(\log n + t(\min\{2^j, n\}, 2^j)\right)\right)$$
$$= O\left(\log^3 n + \sum_{i=1}^{\log_2 m} \sum_{j=1}^{i} t\left(\min\{n, 2^j\}, 2^j\right)\right).$$

amortized time per edge. This concludes the proof. □

Chapter 8

2-edge connectivity

In this chapter we present an $O(\log^4 n)$ deterministic algorithm for the 2-edge connectivity problem for a fully dynamic graph G. An important secondary goal is to present ideas and techniques that will be reused in the next chapter for dealing with the more complex case of biconnectivity.

As in the previous chapters we will maintain a spanning forest F of G. If v and w are connected in F, $v \cdots w$ denotes the simple path from v to w in F. If they are further connected to u, *meet*(u, v, w) denotes the intersection vertex of the three paths $u \cdots v$, $u \cdots w$, and $v \cdots w$.

A tree edge e is said to be *covered* by a non-tree edge (v, w) if $e \in v \cdots w$, that is if e is in the cycle induced by (v, w). A *bridge* is an edge e whose removal disconnects a component, or, equivalently, an edge whose end-points are not 2-edge connected. Hence e is a bridge if and only if it is a tree edge not covered by any non-tree edge. Since 2-edge connectivity is a transitive relation on vertices, it follows that two vertices x and y are 2-edge connected if and only if they are connected in F and all edges in $x \cdots y$ are covered [11].

Recall from connectivity that our spanning forest F was a certificate of connectivity in G in that vertices were connected in G if and only if they were so in F. If an edge from F was deleted, we needed to look for a replacement edge reconnecting F, if possible. An amortization argument paid for all non-replacement edges considered.

Now, for 2-edge connectivity we have a certificate consisting of F together with a set C consisting of a covering edge for each non-bridge edge in F. Thus two vertices are 2-edge connected in G if and only if they are so in $F \cup C$. However, if an edge $f \in C$ is deleted, we may need to add several “replacement edges” to C in order regain a certificate. Nevertheless, by carefully choosing the order in which potential replacement edges are considered, we will be able to amortize the cost of considering all but two of them.

8.1 High level description

The algorithm associates with each non-tree edge e a level $\ell(e) \leq L = \lfloor \log_2 n \rfloor$. However, in contrast to connectivity, the tree edges do not have associated levels. For each i, let G_i denote the subgraph of G induced by edges of level at least i together with the edges of F. Thus, $G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_L \supseteq F$. The following invariant is maintained:

(i') The maximal number of nodes in a 2-edge connected component of G_i is $\lfloor n/2^i \rfloor$. Thus, the maximal relevant level is L.

Initially, all non-tree edges have level 0, and hence the invariant is satisfied. As for connectivity, we will amortize work over level increases. We say that it is *legal* to increase the level of a non-tree edge e to j if this does not violate (i'), that is, if the 2-edge connected component of e in $G_j \bigcup \{e\}$ has at most $\lfloor n/2^j \rfloor$ vertices.

For every tree edge $e \in F$, we implicitly maintain the *cover level* $c(e)$ which is the maximum level of a covering edge. If e is a bridge, $c(e) = -1$. The definition of a cover level is extended to paths by defining $c(P) = \min_{e \in P} c(e)$. During the implementation of an edge deletion or insertion, the c-values may temporarily have too small values. We say that v and w are *c-2-edge connected on level i* if they are connected and $c(v \cdots w) \geq i$. Assuming that all c-values are updated, we have our basic 2-edge connectivity query:

2-edge-connected(v, w)**:** Decides if v and w are c-2-edge connected on level 0.

Further note that with updated c-values, $e \in F$ is a bridge in G_i if and only if $c(e) < i$. For basic updates of c-values, we need

InitTreeEdge(v, w)**:** $c(v, w) := -1$.

Cover(v, w, i)**:** Where v and w are connected. For all $e \in v \cdots w$, if $c(e) < i$, set $c(e) := i$.

Uncover(v, w, i)**:** Where v and w are connected. For all $e \in v \cdots w$, if $c(e) \leq i$, set $c(e) := -1$.

We can now compute c-values correctly by first calling InitTreeEdge(v, w) for all tree edges (v, w), and then calling Cover$(q, r, \ell(q, r))$ for all non-tree edges (q, r). Inserting an edge is straightforward:

Insert(v, w)**:** If the end-points of (v, w) were not connected in F, (v, w) is added to F and InitTreeEdge(v, w) is called. Otherwise set $\ell(v, w) := 0$ and call Cover$(v, w, 0)$. Clearly (i') is not violated in either case.

In connection with deletion, the basic problem is to deal with the deletion of a non-tree edge. If a non-bridge tree edge (v, w) is to be deleted, we first swap it with a non-tree edge as described in Swap below. The sub-routine **FreeTreeEdge** is dummy for now, but is included so that Swap can be reused directly for the biconnectivity problem.

Swap(v, w)**:** Where (v, w) is a tree-edge which is not a bridge. Let (x, y) be a non-tree edge covering (v, w) with $\ell(x, y) = c(v, w) = i$, and set $\ell(v, w) := i$. Call FreeTreeEdge(v, w). Replace (v, w) by (x, y) in F. Call InitTreeEdge(x, y) and Cover(v, w, i).

To see that the above updates the cover information, note that it is only the edges being swapped whose covering is affected. We are now ready to describe delete.

Delete(v, w)**:** If (v, w) is a bridge, we simply delete it. If (v, w) is a tree edge, but not a bridge, we call Swap(v, w). Thus, if (v, w) is not a bridge, we are left with the problem of deleting a non-tree edge (v, w) on level $i = \ell(v, w)$. Now call Uncover(v, w, i) and delete the edge (v, w). This may leave some c-values on $v \cdots w$ to low and thus for $i = \ell(v, w), \ldots, 0$, we call Recover$(v, w, i)$.

Recover(v, w, i)**:** We divide into two symmetric phases. Set $u := v$ and let u step through the vertices of $v \cdots w$ towards w. For each value of u, consider, one at the time, the non-tree edges (x, y) with $meet(x, v, w) = u$ and $\forall e \in u \cdots x, c(e) \geq i$. If legal, increase the level of (x, y) to $i + 1$ and call Cover$(x, y, i + 1)$. Otherwise, we call Cover(x, y, i) and stop the phase.
If the first phase was stopped, we have a second symmetric phase, starting with $u = w$, and stepping through the vertices in $w \cdots v$ towards v.

The problem in seeing that the above algorithm is correct, is to check that the calls to Recover computes the correct c-values on $v \cdots w$. We say that $v \cdots w$ is *fine* on level i if all c-values in F are correct, except that c-values $< i$ on $v \cdots w$ may be too low. Clearly, $v \cdots w$ is fine on level $\ell(e) + 1$ when we make the first call Recover$(v, w, \ell(v, w))$. Thus, correctness follows if we can prove

LEMMA 8.1. *Assuming that $v \cdots w$ is fine on level $i + 1$. Then after a call Recover(v, w, i), $v \cdots w$ is fine on level i.*

PROOF. First note that we do not violate $v \cdots w$ being fine on level $i + 1$ if we take a level i edge (x, y) and either call Cover(x, y, i) directly, or first increase the level to $i + 1$, and then call Cover$(x, y, i + 1)$.

Given that $v \cdots w$ remains fine on level $i+1$, to prove that it gets fine on level i, we need to show that for any remaining level i non-tree edge (x, y), all edges e in $x \cdots y$ have $c(e) \geq i$. In particular, it follows that $v \cdots w$ does become fine on level i if phase 1 runs through without being stopped.

Now, suppose phase 1 is stopped. Let u_1 be the last value of u considered, and (x_1, y_1) be the last edge considered, thus increasing the level of (x_1, y_1) is illegal. Then phase 2 will also stop, for otherwise, it would end up illegally increasing the level of (x_1, y_1). Let u_2 be the last value of u considered, and let (x_2, y_2) be the last edge considered in phase 2.

Since the phases were not interrupted for non-tree edges (x, y) covering edges u before u_1 or after u_2, we know that if (x, y) remains on level i, it is because $x \cdots y \cap v \cdots w \subseteq u_1 \cdots u_2$. Hence, we prove fineness of level i, if we can show that all c-values in $u_1 \cdots u_2$ are $\geq i$.

From the illegality of increasing the level of (x_k, y_k), $k = 1, 2$, it follows that the 2-edge connected component C_k of x_k in $G_{i+1} \cup \{(x_k, y_k)\}$ has $> \lfloor n/2^{i+1} \rfloor$ nodes. However, we know that before the deletion of (v, w), C_1 and C_2 where both part of a 2-edge connected component D of G_i, and this component had at most $\lfloor n/2^i \rfloor$ nodes. Hence $C_1 \cap C_2 \neq \emptyset$. Thus, they are contained in the same 2-edge connected component C of $G_{i+1} \cup \{(x_1, y_1), (x_2, y_2)\}$. Since covering is done for all level $i+1$ edges, it follows that our calls Cover(x_1, y_1, i) and Cover(x_2, y_2, i) imply that all tree-edges in C has got c-values $\geq i$. Moreover $u_k \in C_k$, so $u_1 \cdots u_2 \subseteq C$, and hence all edges in $u_1 \cdots u_2$ have c-values $\geq i$. $\square$

After the last call Recover$(v, w, 0)$, we now know that $v \cdots w$ is fine on level 0, that is, all c-values in F are correct, except that c-values < 0 on $v \cdots w$ may be too low. However, since -1 is the smallest value, we conclude that all c-values are correct, and hence our fully dynamic 2-edge-connectivity algorithm is correct.

8.2 Implementation

The algorithm maintains the spanning forest in a top-tree data structure. For each cluster $\mathcal{C}$ we maintain $c_\mathcal{C} = c(\pi(\mathcal{C}))$. Thus, 2-edge connectivity queries are implemented by:

2-edge-connected(v, w)**:** Set $\mathcal{C}$:=Expose(v, w). Return $(c_\mathcal{C} \geq 0)$.

In connection with Swap, for a given tree edge (v, w), we need a covering edge e with $\ell(e) = c(v, w)$. This is done, by maintaining for each cluster $\mathcal{C}$ a non-tree edge $e_\mathcal{C}$ covering an edge on $\pi(\mathcal{C})$ with $\ell(e_\mathcal{C}) = c_\mathcal{C}$. Then the desired edge e is found by setting $\mathcal{C} := \text{Expose}(v, w)$ and returning $e_{\mathcal{C}}$. Calls to cover and uncover also reduces to operations on clusters:

Cover(v, w, i)**:** Set $\mathcal{C} := \text{Expose}(v, w)$. Call Cover$(\mathcal{C}, i, (v, w))$.

Uncover(v, w, i)**:** Set $\mathcal{C} := \text{Expose}(v, w)$. Call Uncover$(\mathcal{C}, i)$.

The point is, of course, that we cannot afford to propagate the cover/uncover information the whole way down to the edges. When these operations are called on a path-cluster $\mathcal{C}$, we will implement them directly in $\mathcal{C}$, and then store *lazy information* in $\mathcal{C}$ about what should be propagated down in case we want to look at the descendants of $\mathcal{C}$. The precise lazy information stored is

- $c^+_{\mathcal{C}}$, $c^-_{\mathcal{C}}$ and $e^+_{\mathcal{C}}$, where $c^+_{\mathcal{C}} \leq c^-_{\mathcal{C}}$ and $\ell(e^+_{\mathcal{C}}) = c^+_{\mathcal{C}}$. This represents that for all edges $e \in \pi(\mathcal{C})$, if $c(e) \leq c^-_{\mathcal{C}}$, we should set $c(e) := c^+_{\mathcal{C}}$ and $e(e) := e^+_{\mathcal{C}}$.

The lazy information has no effect if $c^+_{\mathcal{C}} = c^-_{\mathcal{C}} = -1$. Trivially, the cover information in a root cluster is always correct in the sense that there cannot be any relevant lazy information above it. Moreover, note that the lazy cover information only effects $\pi(\mathcal{C})$, hence only path descendants of $\mathcal{C}$. Thus, the cover information is always correct for all leaf clusters.

In order to guide Recover, we need two things: first we need to find the level i non-tree edges (q, r), second we need to find out if increasing the level of (q, r) to $i+1$ will create a too large level $i+1$ component. Thus, we introduce counters **size** and **incident** that are further defined so as to facilitate efficient local computation of all of Cover, Uncover, Split, and Merge.

- For any node v and any level i, let size$_{v,i} := 1$ and let incident$_{v,i}$ be the number of level i non-tree edges with an endpoint in v.
- Let i and j be levels, and let v be a boundary node of a path-cluster $\mathcal{C}$. Let $X_{v,\mathcal{C},i,j}$ be the set of internal nodes from the cluster $\mathcal{C}$ that are reachable from v by a path P where $c(P \cap \pi(\mathcal{C})) \geq i$ and $c(P \setminus \pi(\mathcal{C})) \geq j$. Then size$_{v,\mathcal{C},i,j} = (\sum_{w \in X_{v,\mathcal{C},i,j}} \text{size}_{w,i})$ is the number of nodes in $X_{v,\mathcal{C},i,j}$ and incident$_{v,\mathcal{C},i,j} = (\sum_{w \in X_{v,\mathcal{C},i,j}} \text{incident}_{w,i})$ is the number of (directed) level j non-tree edges (q, r) with $q \in X_{v,\mathcal{C},i,j}$. By directed we mean that (q, r) is counted twice if r is also in $X_{v,\mathcal{C},i,j}$.
- Similarly for any level i and any leaf cluster $\mathcal{C}$ with $\partial\mathcal{C} = \{v\}$ let $X_{v,\mathcal{C},i}$ be the set of internal nodes q from $\mathcal{C}$ such that $c(v \cdots q) \geq i$. Then size$_{v,\mathcal{C},i} = (\sum_{w \in X_{v,\mathcal{C},i}} \text{size}_{w,i})$ is the number of nodes in $X_{v,\mathcal{C},i}$ and incident$_{v,\mathcal{C},i} = (\sum_{w \in X_{v,\mathcal{C},i}} \text{incident}_{w,i})$ is the number of (directed) level i non-tree edges (q, r) with $q \in X_{v,\mathcal{C},i}$.

We are now ready to implement all the different procedures: Merge has been split in two parts, one creating a leaf cluster, and one creating a path-cluster $\mathcal{C}$.

Cover$(\mathcal{C}, i, e)$**:** If $c_\mathcal{C} < i$, set $c_\mathcal{C} := i$ and $e_\mathcal{C} := e$. If $i < c^+_\mathcal{C}$, do nothing. If $c^-_\mathcal{C} \geq i \geq c^+_\mathcal{C}$, set $c^+_\mathcal{C} := i$ and $e^+_\mathcal{C} := e$. If $i > c^-_\mathcal{C}$, set $c^-_\mathcal{C} := i$ and $c^+_\mathcal{C} := i$ and $e^+_\mathcal{C} := e$. For $X \in \{\text{size,incident}\}$ and for all $-1 \leq j \leq i$ and $-1 \leq k \leq L$ and for $v \in \partial C$ set $X_{v,\mathcal{C},j,k} := X_{v,\mathcal{C},-1,k}$.

Uncover$(\mathcal{C}, i)$**:** If $c_\mathcal{C} \leq i$, set $c_\mathcal{C} := -1$ and $e_\mathcal{C} :=$ **nil**. If $i < c^+_\mathcal{C}$, do nothing. If $i \geq c^+_\mathcal{C}$, set $c^+_\mathcal{C} := -1$ and $c^-_\mathcal{C} := \max\{c^-_\mathcal{C}, i\}$ and $e^+_\mathcal{C} :=$ **nil**. For $X \in \{\text{size,incident}\}$ and for all $-1 \leq j \leq i$ and $-1 \leq k \leq L$ and for $v \in \partial C$ set $X_{v,\mathcal{C},j,k} := X_{v,\mathcal{C},i+1,k}$.

Clean$(\mathcal{C})$**:** For each path-child $\mathcal{A}$ of $\mathcal{C}$, call Uncover$(\mathcal{A}, c^-_\mathcal{C})$ and Cover$(\mathcal{A}, c^+_\mathcal{C}, e^+_\mathcal{C})$. Set $c^+_\mathcal{C} := -1$ and $c^-_\mathcal{C} := -1$ and $e^+_\mathcal{C} :=$ **nil**.

Split$(\mathcal{C})$**:** Call Clean$(\mathcal{C})$. Delete $\mathcal{C}$.

Merge$(\mathcal{C} : \mathcal{A}, \mathcal{B})$**:** Where $a \in \partial\mathcal{A}$ and $\partial\mathcal{C} = \{a\}$. Let c be the node in $\partial\mathcal{A} \cap \partial\mathcal{B}$. For $X \in \{\text{size,incident}\}$ and for $j := 0, \ldots, L$: If $\mathcal{A}$ is a leaf cluster, set $X_{a,\mathcal{C},j} := X_{a,\mathcal{A},j} + X_{a,\mathcal{B},j}$. Otherwise set $X_{a,\mathcal{C},j} := X_{a,\mathcal{A},j,j}(+X_{c,j} + X_{c,\mathcal{B},j} \text{ if } c_\mathcal{A} \geq i)$.

Merge$(\mathcal{C} : \mathcal{A}, \mathcal{B})$**:** Where $a \in \partial A$,$b \in \partial B$, and $\partial\mathcal{C} = \{a, b\}$. Let c be the node in $\partial\mathcal{A} \cap \partial\mathcal{B}$. Let $\mathcal{D}$ be the path-child of $\mathcal{C}$ minimizing $c_\mathcal{D}$, then set $c_\mathcal{C} := c_\mathcal{D}$ and $e_\mathcal{C} := e_\mathcal{D}$. Set $c^+_\mathcal{C} := -1$ and $c^-_\mathcal{C} := -1$ and $e^+_\mathcal{C} :=$ **nil**. For $X \in \{\text{size,incident}\}$ and for $i, j := -1, \ldots, L$ compute $X_{a,\mathcal{C},i,j}$ as follows ($X_{b,\mathcal{C},i,j}$ is symmetrical): If $\mathcal{A}$ is a leaf cluster, set $X_{a,\mathcal{C},i,j} := X_{a,\mathcal{A},j} + X_{a,\mathcal{B},i,j}$. Otherwise if $\mathcal{B}$ is a leaf cluster, set $X_{a,\mathcal{C},i,j} := X_{a,\mathcal{A},i,j}(+X_{c,\mathcal{B},j} \text{ if } c_\mathcal{A} \geq i)$. Finally if both $\mathcal{A}$ and $\mathcal{B}$ are path-clusters, set $X_{a,\mathcal{C},i,j} := X_{a,\mathcal{A},i,j}(+X_{c,j} + X_{c,\mathcal{B},i,j} \text{ if } c_\mathcal{A} \geq i)$.

Recover(v, w, i)**:**

```
• For $u := v, w$
    – Set $\mathcal{C}$ :=Expose$(v, w)$.
    – While incident$_{u,\mathcal{C},-1,i}$+incident$_{u,i} > 0$ and not stopped,
        ∗ Set $(q, r)$ :=Find$(u, \mathcal{C}, i)$.
        ∗ $\mathcal{D}$ :=Expose$(q, r)$.
        ∗ If size$_{q,\mathcal{D},-1,i} + 2 > n/2^i$,
            · Cover$(\mathcal{D}, i, (q, r))$.
            · Stop the while loop.
        ∗ Else
            · Set $\ell(q, r) := i + 1$, decrement incident$_{q,i}$ and incident$_{r,i}$ and
              increment incident$_{q,i+1}$ and incident$_{r,i+1}$.
            · Cover$(\mathcal{D}, i + 1, (q, r))$.
        ∗ $\mathcal{C}$ :=Expose$(v, w)$.
```

Find$(a, \mathcal{C}, i)$**:** If incident$_{a,i} > 0$ then return a non-tree edge incident to a on level i. Otherwise call Clean$(\mathcal{C})$ and let $\mathcal{A}$ and $\mathcal{B}$ be the children of $\mathcal{C}$. Let $\mathcal{A}$ be the cluster containing a as a boundary node. If a is a boundary node in both $\mathcal{A}$, $\mathcal{B}$, at least one of them must be a leaf cluster, let $\mathcal{A}$ be a leaf-cluster. If $\mathcal{A}$ is a leaf cluster and incident$_{a,\mathcal{A},i} > 0$ or $\mathcal{A}$ is a path cluster and incident$_{a,\mathcal{A},-1,i} > 0$, then return find$(a, \mathcal{A}, i)$. Else, let b be the boundary node nearest to a in $\mathcal{B}$, return find$(b, \mathcal{B}, i)$.

THEOREM 8.2. *There exists a deterministic fully dynamic algorithm for maintaining 2-edge connectivity in a graph, using $O(\log^4 n)$ amortized time per operation.*

PROOF. Cover$(\mathcal{C}, i, e)$ and Uncover$(\mathcal{C}, i)$ both take $O(\log^2 n)$ time. This means that Clean$(\mathcal{C})$ and thus Split$(\mathcal{C})$ takes $O(\log^2 n)$ time. Since Merge$(\mathcal{A}, \mathcal{B}, S)$ also takes $O(\log^2 n)$ time we have by Theorem 3.1 that Link(v, w), Cut(e) and Expose(v, w) takes $O(\log^3 n)$ time. This again means that 2-edge-connected(v, w), Cover$(v \cdots w, i, e)$ and Uncover$(v \cdots w, i)$ take $O(\log^3 n)$ time. Find$(a, \mathcal{C}, i)$ calls Clean$(\mathcal{C})$ $O(\log n)$ times and thus takes $O(\log^3 n)$ time. Finally Recover(v, w, i) takes $O(\xi \log^3 n)$ time where ξ is the number of non-tree edges whose level is increased. Since the level of a particular edge is increased at most $O(\log n)$ times we spend at most $O(\log^4 n)$ time on a given edge between its insertion and deletion. $\square$

Chapter 9

Biconnectivity

In this chapter we present an $O(\log^4 n)$ deterministic algorithm for the biconnectivity problem for a fully dynamic graph G. We will follow the same pattern as was used for 2-edge connectivity. Historically, such a generalization is difficult. For example, it took several years to get sparsification to work for biconnectivity [7, 26]. Furthermore, the generalization in [21] of the $O(\log^5 n)$ randomized 2-edge connectivity algorithm from [23] has an expected bound of $O(\Delta \log^4 n)$, where Δ is the maximal degree [Henzinger, pc 1997]. Our main new idea for preserving the $O(\log^4 n)$ bound for biconnectivity, is an efficient recycling of the information as described in Lemma 9.2 below.

As for 2-edge connectivity we have a certificate for when vertices are biconnected. A *triple* is a length two path xyz in the graph G, and a *tree triple* xyz in F is said to be *covered* by a non-tree edge (v, w) if $xyz \subseteq v \cdots w$, that is if xyz is a segment of the cycle induced by (v, w). Covered triples are also *transitively covered*, and if xyz and $x'yz$ are transitively covered, then so is xyx'. An *articulation point* is a vertex whose removal disconnect a component of G.

LEMMA 9.1 ([20]). *v is an articulation point if and only if there is an uncovered tree triple uvw. Moreover, v and w are biconnected if and only if for all $xyz \subseteq v \cdots w$, xyz is transitively covered.*

9.1 High level description

As with 2-edge connectivity, with each non-tree edge e, we associate a level $\ell(e) \in \{0, \ldots, L\}$, $L = \lfloor \log_2 n \rfloor$, and for each i, we let G_i denote the subgraph of G induced by edges of level at least i together with the edges of F. Thus

$G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_L \supseteq F$. Here, for biconnectivity, we will maintain the invariant:

(i") The maximal number of nodes in a biconnected component of G_i is $\lfloor n/2^i \rfloor$.

As for 2-edge connectivity, the invariant is satisfied initially, by letting all non-tree edges have level 0. We say that it is *legal* to increase the level of a non-tree edge e to j if this does not violate (i"), that is, if the biconnected component of e in $G_j \bigcup \{e\}$ has at most $\lfloor n/2^j \rfloor$ vertices.

For each vertex v and each level i, we implicitly maintain the disjoint sets of neighbors biconnected on level i. If u is a neighbor of v, the set of neighbors of v biconnected to u on level i is maintained as $c_{v,i}(u)$. As for 2-edge connectivity, the c-values may temporarily not be fully updated. If P is a path in G, $c(P)$ denotes the maximal i such that for all triples $xyz \subseteq P$, $z \in c_{y,i}(x)$. If there is no such i, $c(P) = -1$. Thus $c(P) \geq i$ witnesses that the end points of P are biconnected on level i. Typically P will be a tree path, but in connection with Recover, we will consider paths where the last edge (q, r) is a non-tree edge. We say that v and w are *c-biconnected on level* i if they are connected and $c(v \cdots w) \geq i$. If all c-values are updated, we therefore have

biconnected(v, w)**:** Decides if v and w are c-biconnected on level 0.

To update c-values from scratch, we need

InitTreeEdge(v, w)**:** For $i := 0, \cdots, L$, set $c_{x,i}(y) := \{y\}$ and $c_{y,i}(x) := \{x\}$.

FreeTreeEdge(v, w)**:** Remove y from $c_{x,\cdot}(\cdot)$ and remove x from $c_{y,\cdot}(\cdot)$.

Cover(xyz, i)**:** Where xyz is a tree triple, unions $c_{y,j}(x)$ and $c_{y,j}(z)$ for $j := 0, \ldots, i$.

Cover(v, w, i)**:** Calls Cover(xyz, i) for all $xyz \subseteq v \cdots w$.

Now, as for 2-edge connectivity, we can compute all c-values by first calling InitTreeEdge(v, w) for all tree edges (v, w), and then calling Cover$(q, r, \ell(q, r))$ for all non-tree edges (q, r). The above routines immediately complete the descriptions of Insert and Swap. In order to describe Delete, we need to define both Uncover and Recover. To do this efficiently, we have to recycle cover information using the following:

LEMMA 9.2. *Let (v, w) be a level i non-tree edge covering a tree triple $xyz \subseteq v \cdots w$. Suppose s is a neighbor to y biconnected on level $j \leq i$ to x, and hence to y, and z. Then, if (v, w) is deleted, afterwards, s is biconnected on level j to x or z, and it may be biconnected to both.*

PROOF. If s was biconnected to x via (v, w), syz must be covered on level j. Otherwise xys is still covered on level $i \geq j$. □

The Lemma suggests, that when (v, w) is deleted, we should store the neighbors s mentioned. This is done in $c_{y,j}(x|z)$ by Uncover. More precisely, $c_{y,j}(x|z)$ will be the set of neighbors to y that we know are biconnected to x or z, but that are not yet c-biconnected to either. This will be used in one of two ways. Either x and z get c-biconnected on level j, in which case we just restore $c_{y,j}(x)$ and $c_{y,j}(z)$ by setting $c_{y,j}(x) := c_{y,j}(z) := c_{y,j}(x|z) \cup c_{y,j}(x) \cup c_{y,j}(z)$ and $c_{y,j}(x|z) := \emptyset$. Alternatively, suppose we know we have finished updating $c_{y,j}(x)$ and that $z \notin c_{y,j}(x)$. Then we can set $c_{y,j}(z) := c_{y,j}(x|z) \cup c_{y,j}(z)$ and $c_{y,j}(x|z) := \emptyset$.

Uncover(xyz, i)**:** where xyz is a tree triple c-biconnected on level i.
If xyz is c-biconnected on level $i + 1$, do nothing; otherwise, for $j := i, \ldots, 0$, set $c_{y,j}(x|z) := c_{y,j}(x) \setminus (c_{y,j+1}(x) \cup c_{y,j+1}(z))$, $c_{y,j}(x) := c_{y,j+1}(x)$, and $c_{y,j}(z) := c_{y,j+1}(z)$.

Strictly speaking, above, we should also set $c_{y,j}(s) := c_{y,j+1}(s)$ for all $s \in c_{y,j}(x|z)$, but our algorithm will never query any subset of $c_{y,j}(x|z)$.

Uncover(v, w, i)**:** Calls Uncover(xyz, i) for all $xyz \subseteq v \cdots w$.

Cover(xyz, i)**:** Where xyz is a tree triple. For $j = 0, \ldots, i$, if $c_{y,j}(x|z) \neq \emptyset$, union $c_{y,j}(x)$, $c_{y,j}(z)$, and $c_{y,j}(x|z)$, and set $c_{y,j}(x|z) := \emptyset$. Otherwise, union $c_{y,j}(x)$ and $c_{y,j}(z)$, subtracting them from any $c_{y,j}(\cdot|\cdot)$ they might appear in.

To complete the description of Delete, we need to define Recover.

Recover(v, w, i)**:** We divide into two symmetric phases. Phase 1 goes as follows:

```
Set $u := v$ and let $u'$ be the successor of $u$ in $v \cdots w$.
(*) While there is a level $i$ non-tree edge $(q, r)$ such that
      $u = meet(q, v, w)$ and $c(u'\, u \cdots q\, r) \geq i$, if legal, increase the level
      of $(q, r)$ to $i + 1$ and call Cover$(q, r, i + 1)$; otherwise, just call
      Cover$(q, r, i)$ and stop Phase 1.
While $\exists u'uu'' \subseteq v \cdots w$ with $c_{y,j}(x|z) \neq \emptyset$,
      Let $u'uu''$ be such a triple nearest to $v$.
      Run (*) again with the new values of $u$ and $u'$.
      Union $c_{u,j}(u'')$ and $c_{u,j}(u'|u'')$, and set $c_{u,j}(u'|u'') := \emptyset$.
      Run (*) again with $u''$ in place of $u'$.
```

If Phase 1 was stopped in (*), we have a symmetric Phase 2, which is the same except that we start with $u = w$ and in the loop choose the triple $u'uu'' \subseteq w \cdots v$ nearest to w.

The proof of correctness is essentially the same as for 2-edge connectivity. As a small point, note that different biconnected components may overlap in one vertex. Nevertheless, we cannot have two different biconnected components with $> \lfloor n/2^{i+1} \rfloor$ nodes whose combined size is $\leq \lfloor n/2^i \rfloor$.

Note that at the end of Recover(v, w, j), all sets $c_{y,j}(x|z)$, $xyz \subseteq v \cdots w$, will be empty. Hence, for each y, there can be at most one pair x and z with $c_{y,j}(x|z) \neq \emptyset$. Then we refer to x and z as the *uncovered neighbors* of y.

9.2 Implementation

The main difference between implementing biconnectivity and 2-edge connectivity, is that we need to maintain the biconnectivity of the neighbors of all vertices efficiently. For each vertex y, we will maintain $c_{y,\cdot}(\cdot)$ as a list with weights on the links between succeeding elements such that $c(xyz)$ is the minimum weight of a link between x and z in $c_{y,\cdot}(\cdot)$. Then $c_{y,i}(x)$ is a segment of $c_{y,\cdot}(\cdot)$ and using standard techniques for manipulating lists, we can easily find $c(xyz)$ or identify $c_{y,i}(x)$ in time $O(\log n)$.

Now, if $c_{y,j-1}(x) = c_{y,j-1}(z)$, we can union $c_{y,j}(x)$ and $c_{y,j}(z)$ without affecting $c_{y,j-1}(x)$, simply by *moving* $c_{y,j}(z)$ *to* $c_{y,j}(x)$ *on level* j as follows. First we *extract* $c_{y,j}(z)$, replacing it by the minimal link to its neighbors. Since both of these links are at most $j-1$, this does not affect the minimum weight between elements outside $c_{y,j}(z)$. Second we *insert* $c_{y,j}(z)$ after $c_{y,j}(x)$ with link j in between. The link after $c_{y,j}(z)$ becomes the link we had after $c_{y,j}(x)$. Note that if $x \in c_{u,\cdot}(u'|u'')$ and we move $c_{u,j}(x)$ to $c_{u,j}(u')$, then, implicitly, we delete $c_{u,j}(x)$ from $c_{u,j}(u'|u'')$, as required.

InitTreeEdge(v, w)**:** Link w to $c_{v,\cdot}(\cdot)$ on level -1 and v to $c_{w,\cdot}(\cdot)$ on level -1.

FreeTreeEdge(v, w)**:** Extract w from $c_{v,\cdot}(\cdot)$ and v from $c_{w,\cdot}(\cdot)$.

Cover(xyz, i)**:** Where xyz is a tree triple. For $j = 0, \ldots, i$, if x and z are uncovered neighbors of y and $c_{y,j}(x|z) \neq \emptyset$, move $c_{y,j}(x|z)$ and $c_{y,j}(z)$ to $c_{y,j}(x)$. Else, if x is an uncovered neighbor of y, move $c_{y,j}(z)$ to $c_{y,j}(x)$. Else move $c_{y,j}(x)$ to $c_{y,j}(z)$.

Uncover(xyz, i)**:** where $c(xyz) \geq i$, if $c(xyz) > i$, do nothing; otherwise, for $j := i, \ldots, 0$, first extract $c_{y,j}(x)$ and set $c_{y,j}(x|z) := c_{y,j}(x)$. Then move $c_{y,j+1}(x)$ and $c_{y,j+1}(z)$ back to the neighbor list $c_{y,\cdot}(\cdot)$ on level -1.

Biconnectivity by top-trees As for 2-edge connectivity, the algorithm maintains the spanning forest in a top-tree data structure. For each cluster $\mathcal{C}$ we maintain $c_{\mathcal{C}} = c(\pi(\mathcal{C}))$.

Biconnected(v, w)**:** Set $\mathcal{C}$:=Expose(v, w). Return $(c_{\mathcal{C}} \geq 0)$.

Also, $e_{\mathcal{C}}$, $c^{+}_{\mathcal{C}}$, $c^{-}_{\mathcal{C}}$, and $e^{+}_{\mathcal{C}}$ are defined analogously to in 2-edge connectivity. The cover edges $e_{\mathcal{C}}$ and $e^{+}_{\mathcal{C}}$ are exactly the same, while $c^{+}_{\mathcal{C}}$ and $c^{-}_{\mathcal{C}}$, like $c_{\mathcal{C}}$, now refer to covering of triples instead of edges.

A main new idea is that we overrule the top-trees by using the neighbor lists $c_{y,\cdot}(\cdot)$ to propagate information from minimal leaf-clusters to path-clusters. Recall that in 2-edge, the information in leaf-clusters is never missing any lazy information. Let v be the boundary node of a path-cluster $\mathcal{C}$, and let w be any neighbor to v in $\mathcal{C} \setminus \pi(\mathcal{C})$. Then we call w a *non-path neighbor of* v. It is easy to see that there is then a leaf-cluster $\mathcal{A} \subseteq \mathcal{C}$ with $\{v\} = \partial\mathcal{A}$ and $w \in \mathcal{A}$. We call the minimal such cluster $\mathcal{A}$ the *minimal leaf-cluster of* (v, w) , and denote it *MLC*(v, w). Note that the ordering of v and w matters. It is easy to see that there cannot be another (v, w') with *MLC*(v, w') = *MLC*(v, w). We are going to use the neighbor lists to propagate counters directly from minimal leaf-clusters to the minimal path-clusters containing them, skipping all leaf-clusters in between.

We are now ready for the rather delicate definitions of the counters **size** and **incident** for path-clusters and minimal leaf-clusters.

- Let j and k be levels, and let $\mathcal{C}$ be a path-cluster with $\partial\mathcal{C} = \{v, w\}$. Let size$_{v,\mathcal{C},j,k}$ denote the number of internal nodes q of $\mathcal{C}$ such that either $q \in \pi(\mathcal{C})$ and $c(v \cdots q) \geq i$ or there exist a triple $u'uu'' \subseteq \pi(\mathcal{C})$ with $u = meet(v, w, q)$ and $(u, x) \in u \cdots q$ such that $c(v \cdots u) \geq j$, $c(u \cdots q) \geq k$ and either $c(u'ux) \geq k$ or $c(u'uu'') \geq j$ and $x \in c_{u,k}(u'') \cup c_{u,k}(u'|u'')$. Let incident$_{v,\mathcal{C},j,k}$ be the number of (directed) non-tree edges (q, r) with the path $v \cdots qr$ satisfying the conditions from above for the path $v \cdots q$.
- Similarly let k be a level and let $\mathcal{C}$ be a minimal leaf-cluster of an edge (v, w). Let size$_{v,\mathcal{C},k}$ be the number of internal nodes q of $\mathcal{C}$ such that $c(vw \cdots q) \geq k$, and let incident$_{v,\mathcal{C},k}$ be the number of (directed) non-tree edges (q, r) where q is an internal node of $\mathcal{C}$ and $c(vw \cdots qr) \geq k$.

To get from minimal leaf-clusters to path-clusters, and vice versa, we need the following functions:

Size(v, W, i)**:** where W is a set of neighbors of v, returns $\sum_{w \in W}$(Size$_{v,MLC(v,w),i}$ if w non-path neighbor of v, 0 otherwise).

Incident(v, W, i)**:** where W is a set of neighbors of v, returns $\sum_{w \in W}$(1 if w non-tree neighbor of v, Incident$_{v,MLC(v,w),i}$ if w non-path neighbor of v, and 0 otherwise).

Neighbor$X(u, u', i)$**:** $X \in \{$**Size, Incident**$\}$, $X(u, c_{u,i}(u'), i)$

Neighbor$X(u, u'|u'', i)$**:** $X \in \{$**Size, Incident**$\}$,
$X(u, c_{u,i}(u') \cup c_{u,i}(u'') \cup c_{u,i}(u'|u''), i)$.

NeighborFind(u, u', i)**:** Finds $z \in c_{u,i}(u')$ such that z is either a non-tree neighbor of u or a non-path neighbor with incident$_{u,MLC(u,z),i} > 0$.

Whenever the counters of a minimal leaf-cluster *MLC*(v, w) are updated, we update corresponding counters of w in the neighbor list $c_{v,\cdot}(\cdot)$ of v. Standard list data structures allow us, in logarithmic time, to update any one of the $2L$ counters of a neighbor, or to answer a query Neighbor$X$. The remaining operations are implemented analogously to in 2-edge connectivity.

Cover$(\mathcal{C}, i, e)$**:** First we do as in 2-edge connectivity. If $\mathcal{C}$ has path-children $\mathcal{A}$ and $\mathcal{B}$ and $\{u\} = \partial\mathcal{A} \cap \partial\mathcal{B} \not\subseteq \partial\mathcal{C}$ and $u'uu''$ is the triple with $u' \in \mathcal{A}$ and $u'' \in \mathcal{B}$, then we call Cover$(u'uu'', i)$.

Uncover$(\mathcal{C}, i)$**:** First we do as in 2-edge connectivity. If $\mathcal{C}$ has path-children $\mathcal{A}$ and $\mathcal{B}$ and $\{u\} = \partial\mathcal{A} \cap \partial\mathcal{B} \not\subseteq \partial\mathcal{C}$ and $u'uu''$ is the triple with $u' \in \mathcal{A}$ and $u'' \in \mathcal{B}$, then we call Uncover$(u'uu'', i)$.

Merge$(\mathcal{C} : \mathcal{A}, \mathcal{B})$**:** Where $a \in \partial\mathcal{A}$ and $\partial\mathcal{C} = \{a\}$. If $\mathcal{A}$ is a leaf-cluster, we are done. Otherwise, let $u'uu''$ be the unique triple such that $u' \in \pi(\mathcal{A})$, $\{u\} = \partial\mathcal{A} \cap \partial\mathcal{B}$ and $\mathcal{B}$ is the minimal leaf-cluster of (u, u''). Then for $X \in \{\text{size,incident}\}$ and $k := -1, \ldots, L$, $X_{a,\mathcal{C},k} := X_{a,\mathcal{A},k,k}$ if $c_\mathcal{A} < k$,
$X_{a,\mathcal{C},k} := X_{a,\mathcal{A},k,k}+$Neighbor$X(u, u', k)$ if $c_\mathcal{A} \geq k \wedge c(u'uu'') < k$, and finally
$X_{a,\mathcal{C},k} := X_{a,\mathcal{A},k,k}+$Neighbor$X(u, u'|u'', k) + X_{u,\mathcal{B},k}$ if $c_\mathcal{A} \geq k \wedge c(u'uu'') \geq k$.
Let a' be the successor of a in $\pi(\mathcal{A})$. Then $\mathcal{C} = MLC(a, a')$, so we have to update the $2L$ counters associated with a' in a's neighbor list $c_{a,\cdot}(\cdot)$.

Merge$(\mathcal{C} : \mathcal{A}, \mathcal{B})$**:** Where $a \in \partial\mathcal{A}$, $b \in \partial\mathcal{B}$, and $\partial\mathcal{C} = \{a, b\}$. $c_\mathcal{C}$, $e_\mathcal{C}$, $c^+_\mathcal{C}$, $c^-_\mathcal{C}$ and $e^+_\mathcal{C}$ are maintained as in 2-edge connectivity. For $X \in \{\text{size,incident}\}$ and $j, k := -1, \ldots, L$ compute $X_{a,\mathcal{C},j,k}$ as follows ($X_{b,\mathcal{C},j,k}$ is symmetrical): If $\mathcal{A}$ is a leaf-cluster, set $X_{a,\mathcal{C},j,k} := X_{a,\mathcal{B},j,k}$. Otherwise if $\mathcal{B}$ is a leaf-cluster, set $X_{a,\mathcal{C},j,k} := X_{a,\mathcal{A},j,k}$. Finally if both $\mathcal{A}$ and $\mathcal{B}$ are path-clusters, let $u'uu''$ be the triple such that $u' \in \pi(\mathcal{A})$, $\{u\} = \partial\mathcal{A} \cup \partial\mathcal{B}$, and $u'' \in \pi(\mathcal{B})$. Then $X_{a,\mathcal{C},j,k} := X_{a,\mathcal{A},j,k}$ if $c_\mathcal{A} < j$,
$X_{a,\mathcal{C},j,k} := X_{a,\mathcal{A},j,k}+$Neighbor$X(u, u', k)$ if $c_\mathcal{A} \geq j \wedge c(u'uu'') < j$, and finally
$X_{a,\mathcal{C},j,k} := X_{a,\mathcal{A},j,k}+$Neighbor$X(u, u'|u'', k) + X_{u,\mathcal{B},j,k}$ if $c_\mathcal{A} \geq j \wedge c(u'uu'') \geq j$.

Recover(v, w, i)**:** We divide into two symmetric phases. Phase 1 goes as follows:

```
Set $\mathcal{C}$ :=Expose$(v, w)$.
Set $u := v$ and let $u'$ be the successor of $u$ on $u \cdots w$.
(*) While NeighborIncident$(u, u', i) > 0$,
        • Set $(q, r)$ :=VertexFind$(u, \mathcal{C}, i, u')$.
        • $\mathcal{D}$ :=Expose$(q, r)$.
        • Let $(q, q')$ and $(r', r)$ be edges on $q \cdots r$
        • If size$_{q,\mathcal{D},-1,i}$ + 2+ NeighborSize$(q, q', i)$+
          NeighborSize$(r, r', i) > n/2^i$,
            – Cover$(\mathcal{D}, i, (q, r))$.
            – Stop the phase.
        • Else
```

```
            – Set $\ell(q,r) := i+1$, updating the corresponding incidence
              counters $c_{q,\cdot}(\cdot)$ and $c_{r,\cdot}(\cdot)$.
            – Move $c_{q,i+1}(r)$ to $c_{q,i+1}(q')$ and $c_{r,i+1}(q)$ to $c_{r,i+1}(r')$ on
              level $i+1$.
            – Cover($\mathcal{D}, i+1, (q,r)$).
          • $\mathcal{C}$ :=Expose($v, w$).
    $u$ :=FindBranch($v, \mathcal{C}, i$).
    While $u \neq$ nil,
        Let $u'$ be the predecessor, and let $u''$ be the successor of $u$ in
        $v \cdots w$.
        Run (*) again with the new values of $u$ and $u'$.
        Move $c_{y,j}(x|z)$ to $c_{y,j}(z)$ and set $c_{y,j}(x|z) := \emptyset$.
        Run (*) again with $u''$ in place of $u'$.
        $u$ :=FindBranch($v, \mathcal{C}, i$).
```

If Phase 1 was stopped in (*), we have a symmetric Phase 2 with the roles of v and w interchanged.

FindBranch$(a, \mathcal{C}, i)$**:** If incident$_{a,\mathcal{C},-1,i} = 0$ return **nil**else call Clean($\mathcal{C}$). If $\mathcal{C}$ has only one path-child a then return FindBranch($a, \mathcal{A}, i$). Otherwise let $\mathcal{A}$ and $\mathcal{B}$ be the children of $\mathcal{C}$. Let $\mathcal{A}$ be the cluster containing a as a boundary node, if both contains a, let $\mathcal{A}$ be a leaf-cluster. Let $u'uu''$ be the triple such that $u' \in \pi(\mathcal{A})$ and $u'' \in \pi(\mathcal{B})$ and $u \in \partial\mathcal{A} \cup \partial\mathcal{B}$. If incident$_{a,\mathcal{A},-1,i} > 0$ then return FindBranch($a, \mathcal{A}, i$). Otherwise if $c_{u,i}(u'|u'') \neq \emptyset$ then return u else return FindBranch($u, \mathcal{B}, i$).

VertexFind$(u, \mathcal{C}, i, u')$**:** Call Clean($\mathcal{C}$). Let z :=NeighborFind(u, u', i). If z is a non-tree neighbor, return (u, z). Otherwise z is a non-path neighbor. If *MLC*(u, z) is an edge-cluster we set z' :=NeighborFind(z, u, i) and return (z, z'). If *MLC*(u, z) is not an edge-cluster it has two children $\mathcal{A}$ and $\mathcal{B}$ with $u \in \mathcal{A}$, $\mathcal{A} \cap \mathcal{B} = \{b\}$. If incident$_{u,\mathcal{A},i,i} > 0$, return PathFind($u, \mathcal{A}, i$). Otherwise, return VertexFind($b, \mathcal{B}, i, b'$) where b' is the predecessor of b in $u \cdots b$.

PathFind$(a, \mathcal{C}, i)$**:** Call Let $\mathcal{A}$ and $\mathcal{B}$ be the children of $\mathcal{C}$. Let $\mathcal{A}$ be the cluster containing a as a boundary node. If incident$_{a,\mathcal{A},i,i} > 0$ then return PathFind($a, \mathcal{A}, i$). Else let b be the boundary node nearest to a in $\mathcal{B}$. If incident$_{b,\mathcal{B},i,i} > 0$ return PathFind($b, \mathcal{B}, i$). Else return VertexFind($b, \mathcal{C}, i, b'$), where b' is the predecessor of b on $a \cdots b$.

THEOREM 9.3. *There exists a deterministic fully dynamic algorithm for maintaining biconnectivity in a graph, using $O(\log^4 n)$ amortized time per operation.*

PROOF. Cover(xyz, i) and Uncover(xyz, i) and thus Cover$(\mathcal{C}, i, e)$ and Uncover$(\mathcal{C}, i)$ all take $O(\log^2 n)$ time. This means that Clean$(\mathcal{C})$ and thus Split$(\mathcal{C})$ takes $O(\log^2 n)$ time. Since NeighborSize and NeighborIncident each take $O(\log n)$ time, Merge$(\mathcal{A}, \mathcal{B})$ also takes $O(\log^2 n)$ time and we have by Theorem 3.1 that Link(v, w), Cut(e) and Expose(v, w) takes $O(\log^3 n)$ time. This again means that biconnected(v, w) take $O(\log^3 n)$ time. NeighborFind takes $O(\log n)$ time, and FindBranch$(a, \mathcal{C}, i)$ and VertexFind$(u, \mathcal{C}, i, u')$ calls Clean$(\mathcal{C})$ and NeighborFind $O(\log n)$ times and thus takes $O(\log^3 n)$ time. Finally Recover(v, w, i) takes $O(\xi \log^3 n)$ time where ξ is the number of non-tree edges whose level is increased. Since the level of a particular edge is increased at most $O(\log n)$ times we spend at most $O(\log^4 n)$ time on a given edge between its insertion and deletion. $\square$

Chapter 10

Summary and conclusions

We have succeeded in generalizing the topology trees to unbounded degree. The result is *top-trees*, a data structure for dynamic trees, which is easy to use in theory. As a direct result, we have developed effecient dynamic tree algorithms for the diameter, 1-center and 1-median.

We have provided the first deterministic polylogarithmic time algorithms for connectivity, minimum spanning forest, 2-edge connectivity and biconnectivity. For the 2-edge connectivity and the biconnectivity the top-trees was instrumental in their development. For the connectivity problem, our $O(\log^2 n)$ bound matched the previous best randomized time complexity. For the minimum spanning forest, 2-edge connectivity, and biconnectivity, our $O(\log^4 n)$ complexity improved on the previous best randomized complexities.

Bibliography

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 270–280, 1997.

[2] V. Auletta, D. Parente, and G. Persiano. Dynamic and static algorithms for optimal placement of resources in a tree. *Theoretical Computer Science*, 165:441–461, 1996. See also ICALP'94.

[3] S. Cheng and M. Ng. Isomorphism testing and display of symmetries in dynamic trees. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, 1996.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1990.

[5] D. Eppstein. Dynamic euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.*, 13:237–250, 1995.

[6] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. ALCOM-IT Technical Report TR-056-96, Rome, 1996.

[7] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification — a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, September 1997. See also FOCS'92.

[8] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, January 1981.

[9] R. A. Finkel and J. L. Bentley. Quad trees, A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[10] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing*, 14(4):781–798, 1985. See also STOC'83.

[11] G. N. Frederickson. Ambivalent data structures for dynamic 2-Edge-Connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 26(2):484–538, April 1997. See also FOCS'91.

[12] G.N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing*, 14(4):781–798, 1985.

[13] G.N. Frederickson. Parametric search and locating supply centers in trees. In *WADS'91*, volume 519, pages 299–319, 1991. see also SODA'91.

[14] G.N. Frederickson. Ambivalent data structures for dynamic 2–edge–connectivity and k smallest spanning trees. In *SIAM Journal on computing*, volume 26, pages 484–538, 1997. see also FOCS'91.

[15] G.N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997. See also SODA'93.

[16] M. Fredman and M. R. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. Technical Report TR95-1523, Cornell University, Computer Science, 1995. To appear in *Algorithmica*. See also STOC'94 [33].

[17] B. Gavish and S. Sridhar. Computing the 2–median on tree networks in $O(n \log n)$ time. *Networks*, 26, 1995. see also Networks Vol. 27, 1996.

[18] A.J. Goldman. Optimal center location in simple networks. *Transportation Sci.*, 5:212–221, 1971.

[19] S.L. Hakimi and O. Kariv. An algorithmic approach to network location problems. ii: the p-medians. *SIAM J. APPL. MATH.*, 37(3):539–560, 1979.

[20] M. R. Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, June 1995. See also FOCS'92.

[21] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. 36th IEEE Symp. Foundations of Computer Science*, pages 664–672, 1995.

[22] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th Symp. on Theory of Computing*, pages 519–527, 1995.

[23] M. R. Henzinger and V. King. Fully dynamic 2-edge connectivity algorithm in polygarithmic time per operation. Technical Report SRC 1997-004a, Digital, 1997. A preliminary version appeared as [22].

[24] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. Technical report, Digital, 1997. A preliminary version appeared as [25].

[25] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 594–604, 1997.

[26] M. R. Henzinger and H. La Poutré. Certificates and fast algorithms for biconnectivity in fully-dynamic graphs. In *Proc. 3rd European Symp. Algorithms, LNCS 979,*, pages 171–184, 1995.

[27] M. R. Henzinger and M. Thorup. Sampling to provide or to bound: With applictions to fully dynamic graph algorithms. *Random Structures and Algorithms*, 11:369–379, 1997. See also ICALP'96.

[28] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. 30th Symp. on Theory of Computing*, 1998.

[29] G. F. Italiano and R. Ramaswami. Mantaining spanning trees of small diameter. Unpublished revised version of the ICALP paper, 1996.

[30] C. Jordan. Sur les assemblages des lignes. *J. f. die reine und angewandte Math.*, 70:185–190, 1869.

[31] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203–236, 1994.

[32] M. Rauch. *Fully dynamic graph algorithms and their data structures*. PhD thesis, Department of computer science, Princeton University, December 1992.

[33] M. Rauch. Improved data structures for fully dynamic biconnectivity. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, may 1994.

[34] A. Rosenthal and J.A. Pino. A generalized algorithm for centrality problems on trees. *Journal of the ACM*, 36:349–361, 1989.

[35] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. See also STOC'81.

[36] R. E. Tarjan. Efficiency of a good but not linear set union algorithms. *J. Assoc. Comput. Mach.*, 22:215–225, 1975.

[37] M. Thorup. Decremental dynamic connectivity. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 305–313, 1997.

[38] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

Index