DTU Compute
Department of Applied Mathematics
and Computer Science

Bachelor's Thesis

Data structures for dynamic graphs in theory and practice

Authors:

Magnus Kragh Siegumfeldt
s204472

Niels Inselmann Christiansen
s204453

Supervisor:

Eva Rotenberg

June, 2023

Abstract

In this project we implement a top tree library allowing user to implement dynamic graph algorithms, using a recently developed splaying approach. The project covers two simpler dynamic tree algorithms solving the maximum edge weight and diameter problem, along with a more advanced algorithm solving the 2-edge connectivity problem in dynamic graphs. All of the covered algorithms are implemented to demonstrate the functionality of our library.

In this project we implement a top tree library using a recently developed splaying approach, allowing users to implement dynamic graph algorithms on top. The project covers two simpler dynamic tree algorithms solving the maximum edge weight and diameter problem, along with a more advanced algorithm solving the 2-edge connectivity problem in dynamic graphs. All of the covered algorithms are implemented to demonstrate the functionality of our library.

The top tree library is then benchmarked and compared to previous top tree implementations. Our tests show that our top tree performs better in practice for workloads where amortized operation times are acceptable.

Acknowledgements

We would like to thank our supervisor Eva Rotenberg for the many helpful discussions and insightful comments. We would also like to thank Alice Ryhl for her introduction to top trees and her interest in our project. At last we thank Jiří Setnika for sharing his implementations publicly.

Contents

1 Introduction **1**

2 Top Trees **3**

2.1 The top tree data structure . . . 3

2.1.1 Operations on top trees . . . 4

2.1.2 Modifying top trees correctly . . . 5

2.1.3 Implementing top trees . . . 6

2.2 Our top tree library . . . 7

2.2.1 Introduction . . . 7

2.2.2 Overview . . . 9

2.2.3 Design choices and problems . . . 12

2.2.4 Summing up . . . 13

3 Top Tree Algorithms **14**

3.1 Maximum Weight . . . 14

3.2 Diameter . . . 16

3.3 Two-edge connectivity . . . 18

3.3.1 Internal operations . . . 20

3.3.2 The high-level algorithm . . . 21

3.3.3 Supporting connected . . . 23

3.3.4 Maintaining cover levels . . . 24

3.3.5 Maintaining sizes . . . 26

3.3.6 Supporting find first label . . . 30

3.3.7 Supporting cover and uncover . . . 33

3.3.8 Adding and removing edges . . . 36

3.3.9 Analysis of the algorithm . . . 37

3.4 Implementation . . . 37

4 Testing **39**

4.1 Correctness . . . 39

4.2 Benchmarking . . . 40

4.2.1 Methodology . . . 40

4.2.2 Results . . . 41

5 Conclusion **45**

References **46**

A Swap and Recover implementation **48**

1 Introduction

In this project, we seek to examine, implement and analyze a data structure for dynamic trees, namely top trees, introduced first in [2]. Since their initial introduction, top trees have found a number of use cases [1, 4, 6, 12, 7, 11], including minimum spanning trees, connectivity, maximum flow and shortest paths in dynamic graphs.

One can implement top trees in different ways. One approach is to implement top trees, as an interface on Frederickson's topology trees[3]. Here a given tree is first transformed into a tree with maximum degree < 3. Then the vertices are partitioned into sets of size between z and $3z - 2$ for some integer z and then stored in a 2-3-4 tree, keeping the height in order $O(\log n)$ thus allowing effective updates.

Another approach is built on the dynamic trees by Sleator and Tarjan [9] which partitions a given tree into vertex disjoint paths adding and removing edges by operating on these. This can be used to implement top trees, as described in [12], where it is covered how to modify an ST-tree to support the `expose` and `deexpose` operations required by a top tree.

However, a more direct approach called splay top trees, was recently described in [5]. Splay top trees directly represent the underlying tree as a binary tree of connected sets of edges. Balancing is here done by a splay mechanism similar to the splay operation of binary trees described by Sleator and Tarjan in [10]. Here each time a leaf node is touched its height is reduced by a constant factor. It is shown that this yields an amortized cost of $O(\log n)$ for insertion and deletion in a traditional binary tree. This idea can be extended to a binary tree of edge sets yielding the main result of [5].

In this project, we implement a splay top tree library in `C++`. The library lets the user define and extend only relevant components of the data structure and exposes general top tree features through a simple-to-use interface. To demonstrate the utility of this library we introduce three algorithms solving different dynamic graph problems. The two first algorithms serve as an introduction to the functionality of the top trees and said library. A more complex example is then given in section 3.3 where an algorithm solving the 2-edge connectivity problem for dynamic graphs in poly-logarithmic time is explained, analyzed and implemented.

At last, we compare the performance of the library and the implemented algorithms by benchmarking against already existing implementations. In [8] the performance of Topology tree and ST-tree based implementations are compared using algorithms for the maximum weight problem and the 2-edge connectivty problem. We extend these tests to include benchmarks for splay top trees and compare them to previous results.

2 | Top Trees

2.1 The top tree data structure

A *top tree* is data structure for maintaining information about a tree in a dynamic forest. The idea of a top tree is to maintain a binary tree of summaries of arbitrary data, such that we can answer queries about the underlying tree efficiently.

Top trees allows the user to restructure the top tree, picking up to two vertices and changing the tree such that the summary in the root corresponds to the chosen vertices. With a single vertex chosen, we usually look to answer queries about the tree containing it, e.g. the diameter or the maximum edge weight in that tree. When two vertices are chosen, we usually look to answer some query related to the tree path between them, e.g. the length of the path or the maximum edge weight between them. We give a formal definition as follows.

A top tree $\mathcal{T}$ is defined based on a tuple $(T, \partial T)$, consisting of a tree T usually referred to as the *underlying tree* and a set ∂T of at most two vertices of T denoted the *external boundary vertices* or simply the *exposed vertices.* Then any connected subtree C of T (containing at least one edge), has *boundary vertices* $\partial_{(T,\partial T)}C$ or just ∂C. The boundary vertices of C are the vertices of C either in ∂T or incident to an edge in T not in C. We say that C is a *cluster* of $(T, \partial T)$ if it has at most two boundary vertices. By this definition $(T, \partial T)$ is a cluster itself. We say that two clusters are *neighbors* if they intersect only in a single vertex. The vertex where two clusters intersect is called the central vertex.

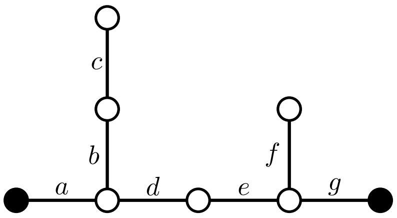
A top tree $\mathcal{T}$ is a binary tree, representing a partition of T into clusters, such that

(i) The root of $\mathcal{T}$ is T itself.

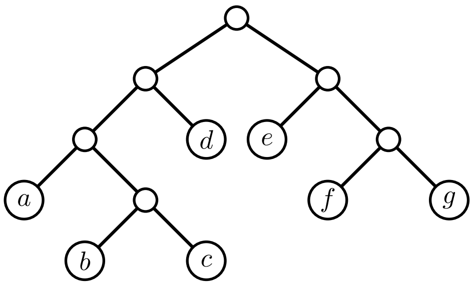
(ii) If C is a parent cluster of two neighboring clusters A and B, then $C = A \cup B$.

(iii) The leaves of $\mathcal{T}$ are clusters consisting only of edges of T.

We say the leaves of $\mathcal{T}$, are *leaf nodes* and the non-leaf nodes of $\mathcal{T}$ are *internal nodes.* For every cluster C in $\mathcal{T}$, we define $\pi(C)$ as the *cluster path* of C. If $\{u, v\} = \partial C$, $\pi(C)$ denotes the unique tree path $u \ldots v$. If $\{u\} = \partial C$, then $\pi(C) = u$. The cluster path is not defined when $|\partial C| = 0$. We say that a cluster with $|\partial C| = 2$ is a *path cluster*, and a *point cluster* if $|\partial C| = 1$ or $|\partial C| = 0$. See Figure 2.1 for an example of an underlying tree and a top tree.



(a) An underlying tree T with exposed vertices ∂T marked by ●.



(b) A top tree over $(T, \partial T)$.

Figure 2.1: Example of a top tree based on an underlying tree with two exposed vertices.

2.1.1 Operations on top trees

Let T_u denote the underlying tree containing u and $\mathcal{T}_u$ a top tree over $(T_u, \partial T_u)$. Then we support the following external operations.

expose(u) Adds u to ∂T_u and returns the root of a new corresponding top tree $\mathcal{T}_u$. Requires that u is not already exposed and that $|\partial T_u| \leq 1$.

deexpose(u) Removes u from ∂T_u and returns the root of a new top tree $\mathcal{T}_u$. Requires that $u \in \partial T_u$.

link(u, v) Connects T_u and T_v by adding the edge (u, v) and returns the root of the resulting top tree. Requires that no vertices are exposed and that u and v are not connected.

cut(e) Removes the edge $e = (u, v)$ from the tree containing it and returns the roots of the two top trees $\mathcal{T}_u$ and $\mathcal{T}_v$. Requires that no vertices are exposed.

We will throughout the paper also use expose(u, v) and deexpose(u, v), which are performed as two consecutive expose and deexpose operations respectively. The four external operations of the top tree are supported using the following internal operations.

create(e) Creates a cluster based on the single edge e.

merge(A, B) Creates a parent cluster $C = A \cup B$ based on the neighboring clusters A and B. Requires that $A \cup B$ is a cluster.

split(C) Deletes the cluster C with children A and B, thus resulting in two top trees with A and B as roots.

destroy(C) Deletes the cluster C, consisting only of a single edge.

Simply creating and deleting clusters, will solve a few simple dynamic tree problems e.g. answering if two vertices are connected, but does not provide the functionality we seek. Therefore we will let create, merge, split and destroy be extendable, that is, give us the opportunity to add computations to these operations. This allows us to store and maintain a wide variety of data in our cluster. We can implement the base version of these operations use $O(1)$ time, but any further extensions might affect this.

2.1.2 Modifying top trees correctly

When modifying the underlying forest, it is important that we update our top trees accordingly. The general procedure when the underlying forest is updated, is as follows, as stated in [1].

(i) First, we split relevant internal nodes top-down.

(ii) Then we destroy relevant leaf nodes.

(iii) Then we update the forest.

(iv) Then we create clusters of the relevant edges.

(v) Finally, we recompute the necessary clusters of the top tree bottom-up with merge.

This procedure allows us to change and update the underlying forest, using link, cut, expose and deexpose, while still maintaining our top tree and the data within the clusters. We will use the following result.

Theorem 1 ([1]). *For a dynamic forest, we can maintain top trees of height $O(\log n)$ supporting each link, cut, expose or deexpose with a sequence of $O(1)$ calls to create and destroy and $O(\log n)$ calls to merge and split. These top tree modifications are identified in $O(\log n)$ time. The space usage of a top tree is linear in the size of the underlying tree.*

The following corollary derived from Theorem 1 and the update order is useful when proving properties of clusters in a top tree.

Corollary 2. *Given an invariant I on the data of the clusters of a top tree, $O(x)$ extensions of merge and split, and $O(y)$ extensions of create and destroy where*

(i) The resulting cluster C of a create(e) satisfies I, if the data in e is correct.

(ii) The resulting cluster C of a merge(A, B) satisfies I, if both A and B satisfies I.

(iii) The resulting child clusters A and B of a split(C) both satisfies I, if C satisfies I.

(iv) The data in the edge e of the corresponding cluster C is correct, after destroy(C).

Then link, cut, expose and deexpose preserve I in the root of the top tree and have asymptotic run time of $O(x \log n + y)$.

Proof: That I is preserved by the top tree is clear given the general procedure for updating top trees above. The running time is given by Theorem 1. □

We shall now introduce a lemma, describing another useful observation, which will be used throughout the later sections. See Figure 2.2 for an illustration of this.

Lemma 3. *There are exactly five valid ways to merge two clusters A and B into their parent cluster C.*

(i) A, B are point clusters, merged into C, a point cluster with no boundary vertices.

(ii) A, B are point clusters, merged into C, a point cluster with one boundary vertex.

(iii) A, B are path clusters merged into C, a path cluster.

(iv) A, B are path and point clusters respectively, merged into C, a path cluster.

(v) A, B are path and point clusters respectively, merged into C, a point cluster.

Proof: Notice that A and B must always have at least one boundary vertex, i.e. the central vertex of C. Otherwise, they would not be neighbors and thus they can not be merged. Notice that the central vertex shared by A or B may or may not be a boundary vertex of C. This is exactly the cases (i) and (ii).

Assume now w.l.o.g. that A has one boundary vertex and B has two boundary vertices. Again the central vertex of C shared by A and B, may or may not become a boundary vertex of C. This is exactly the cases (iv) and (v).

The only remaining case to consider is when A and B both have two boundary vertices. Then the central vertex of C, can not become a boundary vertex, as this would clearly result in C containing three boundary vertices. Thus the only possibility is covered by case (iii).

Thus we have shown, that there are exactly five cases to consider when merging two clusters. □

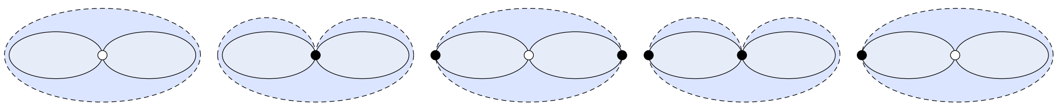


Figure 2.2: The five cases when two clusters are merged into one. The vertices ● are the boundary vertices of the parent cluster and the vertices ○ are the boundary vertices of the children clusters, which did not become boundary vertices of the merged cluster.

2.1.3 Implementing top trees

There exists a number of different ways to implement top trees [3, 9, 12]. In this project we seek to implement and test the splay top tree described in [5], which supports link, cut, expose and deexpose in $O(\log n)$ amortized time. We will not cover the splay top trees in detail nor discuss the analysis, but we will give a brief recap.

In [5] it is claimed that splay top trees are a simpler way to implement top trees, using a splaying approach similar to the one of splay trees introduced by Sleator and Tarjan [10].

Recall that splaying relies on the ability to perform rotations of nodes in the tree. Thus to help keep track of the boundary vertices and minimize the number of cases to consider when rotating, it is useful to impose an *orientation* on the edges of the underlying tree and the nodes of the top tree. We think of the children of a cluster as a left child and a right child. We say that a boundary vertex of an internal node is to the left if the boundary vertex comes from the left child. If a boundary vertex is the central vertex, we say it is in the middle. The left boundary vertex of a leaf node is the left endpoint if it is a boundary vertex. Note that a leaf node does not have a central vertex and thus no middle boundary vertex. Now define the leftmost boundary vertex of a cluster to be the left boundary vertex if it exists, and otherwise the middle, if it exists. The right and rightmost boundary vertices are analogous.

Then the following invariant should hold for all clusters: For any internal node, the leftmost boundary vertex of the right child and the rightmost boundary vertex of the left child must both exist and be equal to the central vertex of the node.

With the orientations in place, a single rotation rotate_up is introduced. We note, unlike the binary search tree rotation, that rotate_up($node$) is only allowed when $sibling(node) \cup sibling(parent(node))$ is a cluster. It is proved that these rotations are allowed often enough to give the amortized bounds we seek. Thus two types of splaying are defined – that is semi_splay and full_splay.

The semi_splay and full_splay give different guarantees, which we shall recap briefly. A semi_splay($node$) reduces the depth of $node$ to at most $\lceil \frac{4}{5} depth(node) \rceil$, i.e. by a constant factor. A full_splay($node$) reduces the depth of $node$ to at most 4. Using these splaying procedures, we can now implement link, cut, expose and deexpose in $O(\log n)$ amortized time.

Thus we state the following theorem as a result

Theorem 4. *For a dynamic forest, we can maintain splay top trees supporting each link, cut, expose or deexpose with a sequence of $O(1)$ calls to create and destroy and $O(\log n)$ amortized calls to merge and split. These top tree modifications are identified in $O(\log n)$ amortized time. The space usage of the top trees is linear in the size of the dynamic forest.*

Proof: Proof of correctness and analysis is given in [5]. □

2.2 Our top tree library

2.2.1 Introduction

With top trees defined will we now discuss our splay top tree library. Before diving into the details, will we first present what we want to achieve and how an implementation of this goal should look. While doing this, it becomes useful to talk about *user-space*, which we define as areas of code which does not include the internals of the top tree implementation. Hence user-space describes the parts of code where the user interacts with only the exposed parts of our library. We start by defining the overarching goal of the library.

Specification

The library we implement should allow a user to implement virtually all algorithms utilizing top trees, while not worrying about the implementation details of the top tree itself. A user should, ideally, only tell the library what data to store in clusters and how to create, merge, split and destroy them. However, the library should also expose all relevant information about clusters (e.g. the number of boundary vertices, as this often will be necessary to consider during create, merge, split and destroy), to the user, but still encapsulate it properly. All this should also be possible in as few lines of code (in user-space) as possible.

While we can implement the above in a number of ways, we decided early on to further specify how exactly a ’good’ solution would look and how to achieve this. This ended up with us defining a set of design principles that came to be, partially through iterative development and partially through preconceived notions of good and bad coding habits. We have grouped these into four major concepts.

Simple Interface

The user should use the top tree library as a black box. Specifically, the user should define, what we call user-defined data or user-data. User-defined data are the extra data stored within clusters, edges or vertices. It should only be required to define the information necessary for the actual algorithm and nothing more. If the user has implemented their class correctly, everything related to the top tree, should simply work and with an interface similar to the one described in section 2.1.

Reusability

The library must be able to be reused easily and concisely without having to change internal code. Ideally, this should result in a library that is able to implement every top tree algorithm easily and without limitations.

Clear structure

Our code should be easy to understand and read by an outside user familiar with splay top trees. This also means that we should not have an unnecessarily complex class structure and that code should exist in well-named files with a clearly defined use.

Minimal clutter

There should be minimal clutter in two ways. First, there should be a minimum amount of boilerplate code, both in library code and in user-space. Secondly, the final compiled data structures and code should also be minimal. A result of this should be that having a graph without vertex data should result in the memory representation also having zero extra bytes. Another consequence would be that if an extensions to split is not needed, it should not need to be mentioned by the user.

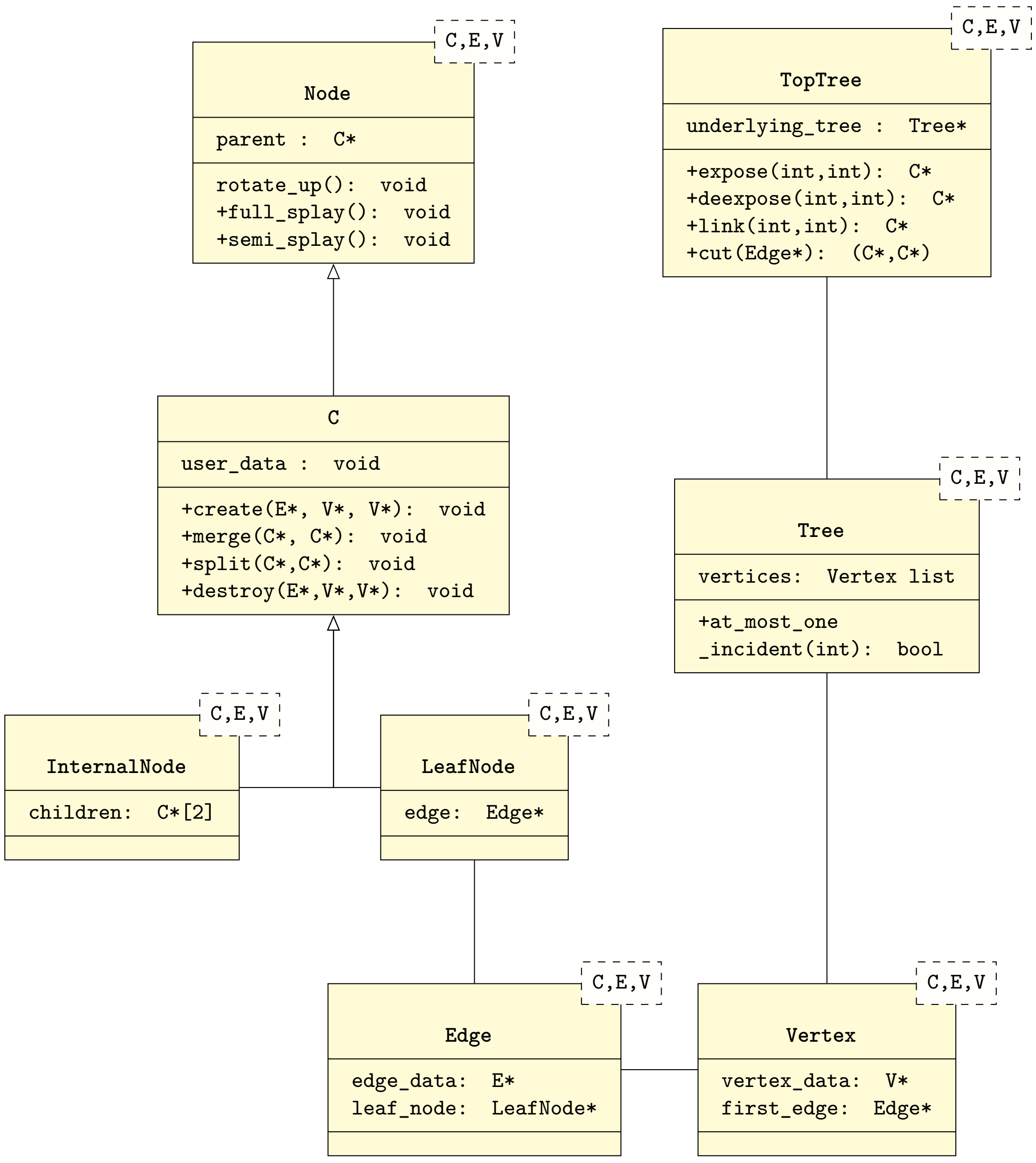


Figure 2.3: Abbreviated class diagram for the top tree library.

2.2.2 Overview

With the requirements for design and functionality in place, were we able to start the actual implementation. To provide the best overview for the reader, will we first present the final implementation of top trees, and how this implementation satisfies our requirements. With the implementation described will we then, in the coming sections, venture into some of the problems which we stumbled upon along the way. We also comment on areas where our implementation may not be ideal.

Our top tree implementation is written to be easily utilized for many different top tree algorithms, regardless of what user-defined data should be stored and how one needs to create, merge, split and destroy clusters. This is achieved with the usage of templates in C++, which provide generic functionality to our top tree. This allows us to define the top tree functionality independently of what user-defined data eventually will be used with it. We note here that the top trees for different algorithms not only vary in the cluster data, that it stores, but also in what – if any – edge and vertex data is stored. The algorithm which determines the maximum weight on a path in a tree uses cluster and edge data, while no vertex data is required. However, more complex algorithms could require a collection of data stored on each edge or data associated with the vertices. Hence our top tree implementation has to be generic, not only over the user-defined clusters, but over user-defined cluster data, edge data, and vertex data. This does infer a lot of complexity into the code as almost all functions and classes need to be generic over these data, but the payoff of being able to use top trees with all different kinds of user-data is much greater and will allow for much simpler implementation of algorithms. We will illustrate the power of this generic design in the following example.

Assume we have a defined `class ClusterData` which implements some version of create and merge. Assume also that we want to associate a single integer, say the weight, with edges, and no data with the vertices. Then we can initialize our top tree functionality over a forest of size 10, by simply typing

```
TopTree<ClusterData,int,None> top_tree = TopTree<ClusterData,int,None>(10);
```

The `ClusterData` definition looks like this

```
class ClusterData : Node<ClusterData, int, None> {
    void create(int* edge, None* left, None* right) {
        ...
    };
    void merge(ClusterData* left, ClusterData* right) {
        ...
    };
};
```

If one were to add data or functionality to clusters, edges or vertices, one has the option to do this without changing a single line of code in our splay top tree implementation. We will in a later section provide an actual example of an algorithm implementation. Returning to our requirements, we feel that this provides the simplest possible way to interact and utilize the top tree implementation without compromising reusability.

The implementation structure

With the usage described, we now move deeper into the implementation to summarize the general structure of the code. Recall that we wanted to prioritize a clear structure of the code and that the implementation should minimize clutter, that is, unnecessary information and functionality.

From the start, we wanted to utilize the object-oriented features of `C++`. It seemed obvious to utilize inheritance as top trees at their simplest are represented as a binary tree of both internal and leaf nodes. We note how both internal and leaf nodes have some behavior in common, e.g. both have parents, both have boundary vertices and so forth. However, there is also specific behavior to each of the two. Internal nodes have children, while leaf nodes have their specific edge associated. By this observation it seemed obvious to induce some inheritance in the implementation, that is, we create three classes `Node`, `InternalNode` and `LeafNode` and let the latter two inherit from `Node`. Hence we are able to define common behavior in the parent class `Node`, while their specific behavior can be defined in the respective classes. With the two types of nodes defined, we let a top tree, be a collection of these, connected by pointers. Each node, except the root, has a pointer to its parent. Internal nodes have two pointers to their children. Leaf nodes contain a pointer to their specific edge. This was the first approach taken to achieve a clear and easy-to-understand structure with minimal clutter.

However, the generic features discussed in the previous section imposed some problems with the design. We wanted to allow the user to extend the `Node` functionality, that is, specify the create, merge, split and destroy functions, along with possible custom user-data. After a few design iterations, we found a solution that achieves the desired behavior, while still keeping the structure of the implementation relatively clear.

The solution to this problem was to let the user define a custom class, which inherits from `Node`. We then let the `InternalNode` and `LeafNode` further inherit from this user-defined `class`. This allows for both `InternalNode` and `LeafNode` to inherit, both the general common `Node` behaviour, and the user-defined behaviour. This does however infer another layer of inheritance, which complicates the structure of our implementation – however, we justify this by the features provided. As is common with these kinds of tree structures we let a class `TopTree` handle all external operations together with construction and destruction of `Node` types. This class also owns the underlying tree on which the top tree is built. We made the choice here to not let the user decide which tree implementation to use and instead force them to use our linked list implementation. The details of this tree are not very interesting however we note that any tree, that supports deletion and addition of edges in $O(\log n)$ time, and can answer if a vertex has ≤ 2 incident edges can be used. The vertices and edges of this tree are also where we store the user-defined edge and vertex data. We want to stress that `TopTree` is not a top tree in the formal sense, it is instead a class that manages the many top trees on a given underlying forest and exposes the external operations to the end user.

2.2.3 Design choices and problems

In this section we will run through some major and minor design considerations and evaluate the design features further

Motivation of structure

We wanted to create a clear and readable code, for a generic top tree supporting user-defined cluster, edge and vertex data. When designing the inheritance structure of our code we had to balance two conflicting design principles. First, we quickly found that generic code in `C++` is very boiler-plate heavy. Furthermore, it also forces us to have all implementation in header files, as all generic code must be in the same compilation unit (this also increases compile time). Thus we sought to contain the generic parts, in self-contained smaller components. Unfortunately, the user-defined cluster, edge and vertex data, had to be involved in the generic parts, as it would otherwise be cumbersome to access information about the node in which the user data resides. Ideally, we would have liked for the user data to live only in a small part of the code while keeping methods like `rotate_up` free from the generic parts, as the behavior of this is not affected by the user-defined data. Throughout development, we were hoping to achieve a solution, where only the methods affected by the user-defined data, were generically implemented, while the rest could be standard for all generic types. We were not able to make such a structure work to our liking, partly due to our inexperience with `C++` and partly due to the time it was taking. In the end, we decided that it was simpler to make everything generic, as we achieved the desired behavior, even though we sacrificed some readability. A consequence of this is that the library is now a header-only library.

Reducing indirection and memory footprint

During the development, we considered how to best introduce the user-defined data. An obvious possibility is to simply store a pointer to the user-defined data (through an abstract class), which at first glance seems like a great solution. However, this adds a level of indirection and uses unnecessary memory, when no data is stored. Most algorithms we look at don't associate any data with the vertices and it would be a waste of memory to store pointers, leading to no data, for every vertex as the memory usage quickly adds up in large graphs. By letting the `Edge` and `Vertex` classes contain a field storing our generic user-defined data, we solve both of these issues as the user-defined data now lives in the same part of memory as the associated edge or vertex, thus saving the space used for a pointer. There is one caveat to this though. The `C++`-standard forces all fields to have some size (1 byte), even if the class has no information. And due to alignment, this meant that our `Vertex` class with no empty user data would use 24 bytes, instead of the 16 bytes strictly required. We solved this by utilizing the so-called *empty base class optimization.* This uses the fact that an empty field in a parent class is exempt from the previous requirement of at least one byte, meaning that if we simply had a parent class for `Vertex` holding only the data we could reduce the memory footprint by 8 bytes. Hence introducing the classes `EHolder` and `VHolder` (not shown in diagram) for `Edge` and `Vertex` respectively.

Proper encapsulation

With the user-defined class as both a parent and child of internal top tree classes, we are unable to use standard techniques for encapsulation. We want full control over which methods are made available in user space, but also want the parent-child relationship intact. If we were programming in Java, this would be a tall order, however as we are programming in `C++` we can use the concept of 'friend' classes. A friend class is simply a way to specify that a given class should act as if all methods and fields of another class are public. The way we then 'skip' the user-defined class in our inheritance hierarchy is by making internal `Node` classes private, except for methods exposed in user space, which are public. We then make all these classes friends of each other, which allows the internal top tree classes, but not the user to edit any given node using methods on internal classes. This allows us to fully control what internal top tree fields and methods a user-defined class has access to and thus minimizes the chance of a user introducing an error.

2.2.4 Summing up

Overall, we achieved a fairly well-structured implementation of splay top trees, allowing a user of our library to easily implement user-defined data and behaviour. The workflow of creating a class and then instantiating a top tree over it, in a single line of code is satisfies our goal of a simple interface and makes the library easy to use for many different algorithms.

A slight disadvantage is the error messages given by the library when a malformed user-defined class is given. These simply fail inside the library code, without giving a hint as to where the error is. This means that the user must be rather familiar with the library in order to decipher these. We did not pursue this, but a possible solution would be to use the `concepts` feature added in `C++20` which might give some more human-readable errors.

At last, it is relevant to state that our top tree is not as clear structure-wise as we hoped, but we think the gains outweigh the sacrifices we made.

3 | Top Tree Algorithms

We will in this chapter seek to cover algorithms solving dynamic graph problems, using top trees. We will as a short introduction to top trees and their applications, show how to solve the maximum weight problem. It should also serve as a gentle introduction to how one uses our library to implement this sort of algorithm.

We will also discuss the diameter problem for dynamic trees and show how to solve this using top trees.

As the primary goal of this paper will we, at last, cover the 2-edge connectivity problem for dynamic graphs. This algorithm is, as we shall see, much more advanced, and will utilize a lot of interesting observations. We will also discuss how this algorithm is implemented utilizing our top tree library.

3.1 Maximum Weight

Let T be a weighted tree, and denote the weight of an edge e, by $weight(e)$. Now let u, v be two vertices of T, and let $u \ldots v$ denote the unique tree path between them. Then define

MaxWeight(u, v) Returns the maximum edge weight on the tree path $u \ldots v$.

We can solve this problem, using top trees, by maintaining an integer, $max\_weight(C)$, for every cluster C and then extending create and merge with constant time operations. When a cluster C is created from an edge e we let

$$max\_weight(C) = \begin{cases} weight(e) & C \text{ is a path cluster} \\ -\infty & \text{otherwise} \end{cases}$$

When A and B are merged into C we let

$$max\_weight(C) = \max\{max\_weight(D) \mid D \in \{A, B\},\ D \text{ is a path cluster}\}.$$

With the above two operations to maintain top trees, one can implement MaxWeight(u, v) by letting $C = \mathsf{expose}(u, v)$ and then

$$\mathsf{MaxWeight}(u, v) = max\_weight(C).$$

This leads to the following

Theorem 5. *We can maintain a fully dynamic forest F, supporting* MaxWeight(u, v) *queries in $O(\log n)$ amortized time using splay top trees.*

Proof: We utilize Corollary 2 to prove this. We want to maintain the following invariant I: If C is a path cluster then $max\_weight(C)$ is the maximum edge weight on $\pi(C)$.

We first show that C = create(e) satisfies this. If the resulting cluster C is a path cluster we set $max\_weight(C) = weight(e)$. Clearly this satisfies I, as e is the only edge on $\pi(C)$ when C is a path cluster.

We now look at merge C = merge(A, B). If C is a path cluster, at least one of A and B are path clusters. Assume that only A is a path cluster, then $\pi(C) = \pi(A)$ and thus I is satisfied, when we let $max\_weight(C) = max\_weight(A)$. This holds similarly when B is the only path cluster.

Assume now both A and B are path clusters. Let $\{a, b\} \in \partial C$ with $a \in \partial A$ and $b \in \partial B$. Let c be the central vertex. Then $\pi(C) = a \ldots c \ldots b$. Hence the assignment $max\_weight(C) = \max(max\_weight(A), max\_weight(B))$ satisfies I as A and B must both satisfy I.

As the above extensions of create and merge are both $O(1)$ time extensions, we support link, cut, expose and deexpose in $O(\log n)$ amortized time, and thus we can answer MaxWeight(u, v) queries in $O(\log n)$ amortized time. □

We turn now to the implementation of this algorithm with our top tree library. As mentioned earlier one should only implement the cluster class storing the necessary user-data, along with the required `create` and `merge` operations.

```
class MaxWeightCluster : Node<MaxWeightCluster, int, None> {
    int max_weight;
    void create(int* edge, None* left, None* right) {
        this->max_weight = this->is_path() ? *edge : INT_MIN;
    };
    void merge(MaxWeightCluster* left, MaxWeightCluster* right) {
        this->max_weight = std::max(
            left->is_path() ? left->max_weight : INT_MIN,
            right->is_path() ? right->max_weight : INT_MIN
        );
    };
};
```

With the above class defined, one is now able to utilize the top tree implementation to solve the maximum path problem in a dynamic forest. We provide a simple example below, illustrating how one would initialize and use the structure. We note here that cut and deexpose is called similarly to link and expose, when necessary.

```
// Initialize data structure of size 4.
TopTree<MaxWeightCluster,int,None> top_tree = TopTree<MaxWeightCluster,int,None>(4);

top_tree.link(1, 2, 1); // Create edge between vertices 1 and 2 with weight 1
top_tree.link(2, 3, 2); // Create edge between vertices 2 and 3 with weight 2

MaxWeightCluster* root = top_tree.expose(1, 3);
```

```
std::cout << root->max_weight << std::endl; // Prints 2
```

As a last note in this section, we hope to have shown how our top tree library provides a simple-to-use interface, which can be used to implement many different algorithms. Recall also how this achieves the goals set in section 2.2.1.

3.2 Diameter

Another more advanced application of top trees is the diameter problem which we define as follows: Given a weighted tree T, then the diameter of T is the maximal path length between any two vertices in T.

Diameter(u) Returns the diameter of the tree containing u.

We can solve this problem using top trees, by storing integers $diameter(C)$, $length(C)$ and $max\_dist(C, a),\ a \in \partial C$ for each cluster C. Here *diameter* represents the diameter of the cluster, *length* stores the length of the cluster path and *max_dist* stores the maximum distance from each of the two boundary vertices to another vertex in the cluster. These definitions are illustrated in Figure 3.1.

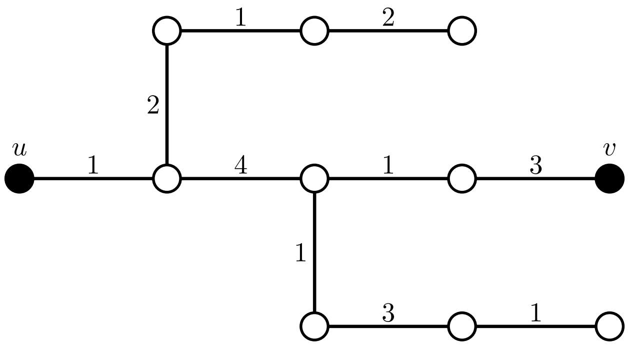


Figure 3.1: A cluster C with exposed vertices u and v marked by ●. Here $length(C) = 9$, $max\_dist(C, u) = 10$, $max\_dist(C, v) = 12$ and $diameter = 14$.

We maintain this information by the following constant time extensions of create and merge. When C is created from an edge e, we let

$$length(C) = \begin{cases} weight(e) & C \text{ is a path cluster} \\ 0 & \text{otherwise} \end{cases}$$
$$diameter(C) = weight(e),$$

and for each $a \in \partial C$, we let

$$max\_dist(C, a) = weight(e).$$

When C is created by a $\mathsf{merge}(A, B)$ with central vertex c, we maintain

$$length(C) = \sum_{\substack{D \in \{A,B\},\\ D \text{ is a path cluster}}} length(D), \tag{3.1}$$

$$diameter(C) = \max(diameter(A),\ diameter(B),\ max\_dist(A, c) + max\_dist(B, c)). \tag{3.2}$$

At last we compute $max\_dist(C, a)$ for all $a \in \partial C$. Assume w.l.o.g. that $a \in \partial A$.

$$max\_dist(C, a) = \begin{cases} \max(max\_dist(A, c),\ max\_dist(B, c)) & \text{if } a = c \\ \max(max\_dist(A, a),\ length(A) + max\_dist(B, c)) & \text{otherwise} \end{cases} \tag{3.3}$$

With the above two extensions maintaining the top tree, one can implement $\mathsf{Diameter}(u)$ by letting $C = \mathsf{expose}(u)$ and then

$$\mathsf{Diameter}(u) = diameter(C).$$

Theorem 6. *We can maintain a fully dynamic forest F, supporting $\mathsf{Diameter}(u)$ queries in $O(\log n)$ amortized time using splay top trees.*

Proof: We again use Corollary 2. For a cluster C we let I be the conjunction of the following statements

(i) $length(C)$ is the length of $\pi(C)$ if C is a path cluster,

(ii) $max\_dist(C, c)$, $c \in \partial C$ is the maximal distance from c to any vertex in C,

(iii) $diameter(C)$ is the diameter of the subtree spanned by C.

We argue first that a cluster resulting from a create satisfies I. As the cluster resulting from a create only spans one edge, we know that if this cluster is a path cluster, then the length of the path can only be the weight of e. This also means that the maximum distance from any boundary vertex and the diameter must be the length of the edge. Altogether this satisfies (i), (ii) and (iii).

We now argue the correctness of $C = \mathsf{merge}(A, B)$ and let c be the central vertex of C.

(i) If C is a path cluster we have from Lemma 3 that it may be created from either two paths or a path and a point. Assume by symmetry that A is a path with $\{a, c\} = \partial A$ and that B is a point with $\{c\} = \partial B$. Then $\pi(C) = a \dots c = \pi(A)$ and thus $length(C) = length(A)$ in this case. If instead A and B are both path clusters with $\{a, c\} = \partial A$ and $\{c, b\} = \partial B$, then $\pi(C) = \pi(A) \cup \pi(B)$. Thus by I we know that the length of $\pi(C) = length(A) + length(B)$.

(ii) Note how the boundary vertices $a \in \partial C$, may or may not be the central vertex c. Assume first that $a \neq c$ and by symmetry $a \in \partial A$. Denote by u the vertex with the maximum distance from a. u must clearly exist in either A or B. If u exists in A, this distance is exactly $max\_dist(A, a)$. If u exist in B, the distance to u, must be the length of $\pi(A) = a \dots c$ plus the length of $c \dots u$. Note how the length of $c \dots u$ must be $max\_dist(B, c)$ as u is furthest from a and therefore also furthest from c. By this the correctness of $max\_dist(C, a) = \max(max\_dist(A, a),\ max\_dist(B, c) + length(A))$ follows.

We now cover the case where $a = c$, that is, a is the central vertex. Denote by u the vertex furthest from a. Clearly, u must exist only in one of A or B. Thus $max\_dist(C, a)$ must be either $max\_dist(A, a)$ or $max\_dist(B, a)$. If $u \in A$, we have $max\_dist(A, a) \geq max\_dist(B, a)$. By this the correctness of $max\_dist(C, a) = \max(max\_dist(A, a), max\_dist(B, a))$ follows.

Together these two cases correspond exactly to (3.3).

(*iii*) Denote by P the longest path contained in C. We then have three cases to consider. P's edges are all contained in A, P's edges are all contained in B, or P's edges are contained in both A and B.

First assume that the edges of P are contained only in A. Therefore $diameter(C) = diameter(A)$. This argument can be applied analogously when the edges are completely contained in B.

Assume now that the edges of P are contained in both A and B. Therefore we have $c \in P$. The length of P must then be the sum of maximum distances from c to arbitrary vertices in A and B respectively. Thus the diameter must be $diameter(C) = max\_dist(A, c) + max\_dist(B, c)$.

Evaluating all possibilities and taking the largest one thus computes the diameter correctly.

As the root is never changed directly, I is never invalidated for the children and thus we can be sure that I always holds for its children which satisfies the requirements of Corollary 2 for split and destroy.

Note also that both of the above extensions of create and merge are $O(1)$ time extensions, and thus we support link, cut, expose and deexpose in $O(\log n)$ amortized time using splay top trees, and therefore we can also answer Diameter(u) queries in $O(\log n)$ amortized time. □

3.3 Two-edge connectivity

We have now covered some examples of applications of top trees and demonstrated how these algorithms can be implemented efficiently using our top tree library. We will now move on to the main challenge of the project, and discuss a much more complicated problem. We seek to describe and implement the dynamic 2-edge connectivity problem. Our description lies close to both [6, 4], but we seek to provide a description that translates more directly to an actual implementation using our splay top tree library. In our description and implementation, we also changed a few concepts, allowing for simpler implementation. This project does not include as strict proof as in the earlier sections, as this is out of the scope of the project. Instead, we give arguments of correctness throughout the description.

Let's first define the problem. Given a fully dynamic graph G and two arbitrary vertices $u, v \in G$ then u and v are 2-edge connected if there exist two edge-disjoint paths between them. We seek to answer the following three queries.

TwoEdgeConnected(u, v) Return *true* if u and v are 2-edge connected in G. Otherwise return $false$.

FindBridge(u) Return any bridge in the connected component of G, containing u, if one such exist. Otherwise return *null*.

FindBridge(u, v) Return any bridge between u and v in G, if one such exist. Otherwise return *null*.

Along with the three above queries, we also want to support the following two operations, to dynamically insert and delete edges.

Insert(u, v) Add an edge e between vertices u and v in G and return e.

Delete(e) Remove the edge e from G.

It should come as no surprise that we, to solve this problem, want to use the top trees to answer the above-defined queries. Recall that top trees are based on an underlying forest, i.e. a collection of trees. For this problem, we are working with a graph G, which is not necessarily a forest. Hence we need to adjust our view of the graph, in order to utilize top trees on the graph.

Let F be a spanning forest of G. We shall refer to all edges $e \in F$ as *tree edges* and all edges $e \notin F$ as *non-tree edges*. We do not store G explicitly – instead, we store the aforementioned spanning forest F of G using top trees. Non-tree edges are not stored as part of the spanning forest F but are instead associated with its vertex endpoints.

As we will need data associated with vertices, we introduce *vertex labels*. A vertex label associates arbitrary data with our vertices. In [6] vertex labels are introduced as point clusters, being handled slightly differently. For this algorithm, we will use a different approach, which translates easier to a splay top tree implementation. Note that a given vertex might be part of many leaf nodes across the top tree, requiring us to potentially, recompute the complete top tree after every vertex label update. As this is too expensive, we let every vertex be *owned* by a single leaf node C, consisting of a neighboring edge e, if one such exists. Then we let the data of the vertex label affect only the owning cluster, enabling us to update only the leaf-to-root path of the owning cluster. Using this idea, we let every vertex store its neighboring non-tree edges. We will also use *user label* to denote non-tree edges stored in a vertex label.

Additionally we also associate with every non-tree edge e of the graph a *level*, $l(e)$, where $0 \leq l(e) < l_{max}$ with $l_{max} = \lfloor \log_2 n \rfloor$. We say that a tree edge e is *covered* by a non-tree edge (u, v) if the tree path $u \dots v$ contains e. Now define the *cover level* of all tree edges e, $c(e)$ as the minimum level of a covering edge. If no non-tree edge covers e, let $c(e) = -1$. Define also the cover level of a tree path $P = u \dots v$ as $c(P) = \min_{e \in P} c(e)$. A key result of this is that a tree edge e has $c(e) = -1$ if and only if it is a bridge in G.

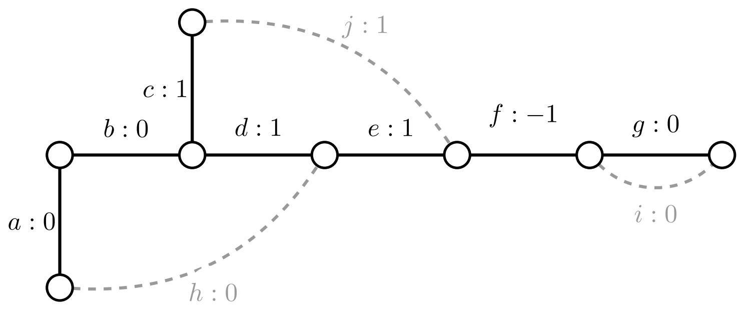


Figure 3.2: A graph with tree edges a, b, c, d, e, f, g and non-tree edges h, i, j with levels $0, 0, 1$ respectively. Here h covers a, b, d on level 0, i covers g on level 0, and j covers c, d, e on level 1. Observe also that f is not covered and therefore has cover level -1 and thus is a bridge.

We let the tree edges with cover level $\geq i$, form a subforest F_i of F, such that $F = F_{-1} \supseteq F_0 \supseteq F_1 \supseteq \cdots \supseteq F_{l_{max}-1}$. Furthermore we maintain the following invariant concerning the size of the 2-edge connected components on different levels: $|F_i| \leq \lfloor \frac{n}{2^i} \rfloor$. Thus any 2-edge connected component on level i has size at most $\lfloor \frac{n}{2^i} \rfloor$. We will use this level partition to search for edges in a structured way, leading to a nice amortization argument.

At last we need to define $meet(u, v, w)$, of three vertices u, v, w in an underlying tree, as the unique vertex where $u \dots v$, $u \dots w$ and $v \dots w$ intersect.

3.3.1 Internal operations

To answer the above-mentioned queries we will need a data structure that provides the following internal operations. Firstly, we need to be able to add and remove tree and non-tree edges.

1. **Link**(u, v) Add a tree edge (u, v) to F and set its cover level to -1.

2. **Cut**(e) Remove the tree edge (e) from F.

3. **AddLabel**(u, e) Add a non-tree edge e, to the list of associated non-tree edges of u.

4. **RemoveLabel**(u, e) Remove e from the list of associated non-tree edges of u.

We will also need an operation to determine whether two vertices are connected.

5. **Connected**(u, v) Return *true* if u and v is connected in F, otherwise, return $false$.

We will need the following operations to maintain level and cover level of edges.

6. **Cover**(u, v, i) For every edge $e \in u \dots v$, set $c(e) = i$ if $c(e) < i$.

7. **Uncover**(u, v, i) For every edge $e \in u \dots v$, set $c(e) = -1$ if $c(e) \leq i$.

8. **CoverLevel**(u) Return the minimal cover level of any edge in the tree containing u.

9. **CoverLevel**(u, v) Return the minimal cover level of any edge on the tree path $u \dots v$.
10. **MinCoveredEdge**(u) Return the edge with the minimal cover level of any edge in the tree containing u.
11. **MinCoveredEdge**(u, v) Return the edge with minimal cover level of any edge on the tree path $u \dots v$.

We will also need an operation to determine the size of the 2-edge connected components at different levels so that we can tell whether a level increase of some non-tree edge will violate the size invariant.

12. **FindSize**(u, v, i) Find the size of the 2-edge connected component at level i which results if a non-tree edge (u, v) has its level raised to i.

At last, we will need an operation, which locates relevant non-tree edges.

13. **FindFirstLabel**(u, v, i) Find a user label associated with a vertex label w where CoverLevel$(w, meet(u, v, w)) \geq i$. Return a user label, which minimizes the distance from u to $meet(u, v, w)$.

3.3.2 The high-level algorithm

Assuming we are able to support the above-mentioned internal operations, can we proceed to describe how to answer the queries TwoEdgeConnected(u, v), FindBridge(u) and FindBridge(u, v) along with how to support the external operations Insert(u, v) and Delete(e). The queries are easy to support as shown in Algorithm 1. Insert(u, v) and Delete(e) however, require a little more work to implement, as we need to handle tree and non-tree edges differently.

To insert an edge, we add it to our spanning forest F, if u and v are not connected. If however u and v are connected we add the edge as a non-tree edge, associated with both of the vertices u and v and give it level 0. At last, we cover $u \dots v$ on level 0, such that any prior bridges, which are now covered, do not have cover level -1 anymore.

We come now to the deletion of an edge. If the edge is a bridge it can simply be Cut. If the edge is a non-bridge tree edge, we can not delete it, as F would not be spanning in that case. Hence we need to swap the tree edge in F with some non-tree edge, such that F is still spanning after the edge deletion, and then delete the non-tree edge as follows.

When we have some non-tree edge $e = (u, v)$ to delete, we remove e from the non-tree edge list of both u and v and Uncover$(u, v, l(e))$. Unfortunately, cover levels on the tree path $u \dots v$ might become too low, if a section of this path was covered on a level $\leq l(e)$. Thus we need to recover the lost cover levels on the $u \dots v$ path. To support Delete we introduce

Swap(e) Swaps the tree edge e where $c(e) \geq 0$, with a non-tree edge f which covers e, with $l(f) = c(e)$.

We say that $u \dots v$ is *fine* on level i if $c(e)$ is correct for all $e \in u \dots v$, except when $c(e) < i$.

Recover(u, v, i) Makes $u \dots v$ fine on level i, assuming that $u \dots v$ is fine on level $i + 1$.

Intuitively, we support Swap(e), where $e = (u, v)$ by executing Cut(u, v), such that u and v are disconnected. We then look through the level i non-tree edges using FindFirstLabel within the smaller of the two components containing u and v respectively. If we encounter some non-tree edge (q, r), which connects u and v return this non-tree edge immediately as we have found a replacement. If the non-tree edge does not connect u and v, we are able to increment the level of (q, r). We continue until we find some replacement – note that a replacement must always exist, as (u, v) had cover level i and thus it was covered by some level i non-tree edge. Also note that all edges which do not connect u and v are guaranteed to fit on the next level, as it lies within the smaller of the two components containing u and v respectively.

We support Recover in two symmetric phases by letting w step through the $u \dots v$ path, first from u to v and then from v to u. During each phase, we increase the levels of the level i non-tree edges (q, r) with $meet(u, v, q) = w$, if FindSize$(q, r, i + 1) \leq \lfloor \frac{n}{2^{i+1}} \rfloor$. If this is not the case we cover $q \dots r$ on level i and stop the phase.

This procedure must always make $u \dots v$ fine on level i. Suppose the first phase is not stopped, then all edges of $u \dots v$ must be fine on level i as all non-tree edges (q, r) covering some part of $u \dots v$ were raised to level $i + 1$, and all $q \dots r$ edges were covered on level $i + 1$. Thus all edges of $u \dots v$ have cover level $\geq i$ if they are covered by a level $\geq i$ non-tree edge, and thus the edges of $u \dots v$ are fine on level i. As there are no more edges on level i to consider, the second phase does nothing.

If the first phase was stopped when some edge (q, r) was considered, then the second phase must also stop as the contrary would mean that (q, r) had its level increased illegally. Denote the two 2-edge connected components on level k containing u and v, U_k and V_k respectively. As we could not raise (q, r) to level $i + 1$, we know that $U_{i+1} \cap V_{i+1} = \emptyset$. We also know that $|U_i| > \lfloor \frac{n}{2^{i+1}} \rfloor$, and $|V_i| > \lfloor \frac{n}{2^{i+1}} \rfloor$. However as $u \dots v$ was covered on at least level i, we know that $|U_i \cup V_i| \leq \lfloor \frac{n}{2^i} \rfloor$. Thus it is clear that U_i and V_i overlap, and therefore all edges on $u \dots v$ have cover level $\geq i$ if covered by a level $\geq i$ non-tree edge. Therefore we conclude that all edges on $u \dots v$ must be fine on level i after a Recover(u, v, i).

Swap and Recover can be implemented using operations *1.* to *13.*. We refer to Algorithm 4 in Appendix A for pseudocode.

Now, that we can Swap and Recover, we are able to support Insert and Delete as shown in Algorithm 2.

Hence the challenge is to support operations *1.* to *13.*. We shall describe how to implement and support these in the following sections. In section 3.3.3 we describe how to answer the Connected query. In section 3.3.4 we describe how to support *8.* to *11.*. In section 3.3.5 we describe how to implement FindSize and in section 3.3.6 we describe how to implement FindFirstLabel. Section 3.3.7 describes how to support Cover and Uncover. At

Algorithm 1 Two-Edge Connectivity queries

```
function TwoEdgeConnected($u, v$):
    return Connected($u, v$) $\wedge$ CoverLevel($u, v$) $\geq 0$

function FindBridge($u$):
    if CoverLevel($u$) $= -1$ then
        return MinCoveredEdge($u$)
    else
        return $null$

function FindBridge($u, v$):
    if CoverLevel($u, v$) $= -1$ then
        return MinCoveredEdge($u, v$)
    else
        return $null$
```

Algorithm 2 Two-Edge Connectivity delete and insert

```
function Insert($u, v$):
    if $\neg$Connected($u, v$) then
        $e \leftarrow$ Link($u, v$)
    else
        $e \leftarrow$ new non-tree edge $(u, v)$
        $l(e) \leftarrow 0$
        AddLabel($u, e$)
        AddLabel($v, e$)
        Cover($u, v, 0$)

function Delete($e$):
    $(u, v) \leftarrow e$
    if $e$ is a tree edge then                          ▷ $e$ is a tree edge
        $\alpha \leftarrow$ CoverLevel($u, v$)
        if $\alpha = -1$ then                           ▷ $e$ is a bridge
            Cut($e$)
            return
        Swap($e$)
    else
        $\alpha \leftarrow l(e)$
    RemoveLabel($u, e$)                                 ▷ $e$ is a non-tree edge
    RemoveLabel($v, e$)
    Uncover($u, v, \alpha$)
    for $i \leftarrow \alpha \ldots 0$ do
        Recover($u, v, i$)
```

last, we describe how to support *1.* to *4.* in section 3.3.8.

3.3.3 Supporting connected

In order to answer Connected(u, v) we could implement extensions of create, merge, split and destroy, such that every top tree, based on the underlying forest, had unique id's, however, a more elegant solution is possible if we simply utilize the internal structure of splay top trees.

We recall that a rotation on *node* is allowed when $sibling(node) \cup sibling(parent(node))$ is a cluster. Thus a rotation of *node* is only allowed when it has depth ≥ 2. Note also that rotate_up does not change the depth of the grandparent of *node*. A consequence of this is that the root cannot change during a splay operation. An expose operation, changes the structure of the tree using splays only, and thus the root does not change during expose operations. Thus we can implement Connected(u, v) in $O(\log n)$ amortized time (when constant time extensions of merge and split are used) as follows. Let $C_u = \mathsf{expose}(u)$ and $C_v = \mathsf{expose}(v)$, then

$$\mathsf{Connected}(u, v) = \begin{cases} true & \text{if } C_u = C_v \\ false & \text{otherwise} \end{cases}$$

3.3.4 Maintaining cover levels

In this section we show how to support operations *8.* to *11.*. We can answer these operations by maintaining the following information in the clusters of our top trees.

$$\begin{aligned} cover(C) &= \min\left(\{c(e) \mid e \in \pi(C)\} \cup \{l_{max}\}\right) \\ globalcover(C) &= \min\left(\{c(e) \mid e \in C \setminus \pi(C)\} \cup \{l_{max}\}\right) \\ minpathedge(C) &= \begin{cases} \arg\min_{e \in \pi(C)} c(e) & \text{if } C \text{ is a path cluster} \\ null & \text{otherwise} \end{cases} \\ minglobaledge(C) &= \begin{cases} \arg\min_{e \in C \setminus \pi(C)} c(e) & \text{if } C \text{ contains edges } e \in C \setminus \pi(C) \\ null & \text{otherwise} \end{cases} \end{aligned}$$

An illustration of these values are provided in Figure 3.3. With these values in place, are we able to easily answer CoverLevel(v) and MinCoveredEdge(v). Let $C = \mathsf{expose}(v)$. Then

$$\begin{aligned} \mathsf{CoverLevel}(v) &= globalcover(C), \\ \mathsf{MinCoveredEdge}(v) &= minglobaledge(C). \end{aligned}$$

Similarly, one can answer CoverLevel(u, v) and MinCoveredEdge(u, v) by $C = \mathsf{expose}(u, v)$ and

$$\begin{aligned} \mathsf{CoverLevel}(u, v) &= cover(C), \\ \mathsf{MinCoveredEdge}(u, v) &= minpathedge(C). \end{aligned}$$

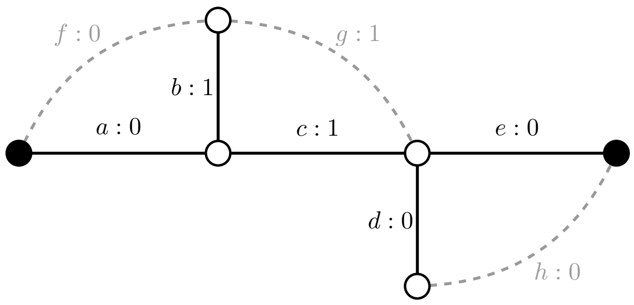


Figure 3.3: A cluster C with exposed vertices ● and tree edges a, b, c, d, e and non-tree edges f, g, h with levels $0, 1, 0$ respectively. Here a is covered on level 0 by f, and b, c are covered on level 1 by g, while d, e are covered on level 0 by h. Then $\mathit{cover}(C) = \min(c(a), c(c), c(e), l_{max}) = 0$, $\mathit{minpathedge}(C) = a$, $\mathit{globalcover}(C) = \min(c(b), c(d), l_{max}) = 0$ and $\mathit{minglobaledge}(C) = d$.

To maintain this information, we will need to specify how create, merge, split and destroy should operate, such that the information in the root, always is correct. We note that all of the above information is computed bottom-up only, and thus split and destroy won't be necessary to maintain this information. We shall however in section 3.3.7 introduce operations *6.* and *7.*, i.e. Cover(u, v, i) and Uncover(u, v, i), which introduces a need for split and destroy.

Let e be an edge, of which the cluster C is created. If C is a path cluster, let

$$\mathit{cover}(C) = c(e), \quad \mathit{globalcover}(C) = l_{max}, \quad \mathit{minpathedge}(C) = e, \quad \mathit{minglobaledge}(C) = \mathit{null}.$$

Otherwise C is not a path cluster and we let

$$\mathit{cover}(C) = l_{max}, \quad \mathit{globalcover}(C) = c(e), \quad \mathit{minpathedge}(C) = \mathit{null}, \quad \mathit{minglobaledge}(C) = e.$$

It is not hard to see that these assignments correspond to the definitions given at the start of this section.

Now suppose two clusters A, B are merged into their parent cluster C. If C is a point cluster then the cover level of C is l_{max} and *minpathedge* is *null*. If however, C is a path cluster, we have $\pi(C) = \pi(A) \cup \pi(B)$, and then the cover level of C is the minimum of $\mathit{cover}(A)$ and $\mathit{cover}(B)$. The *minpathedge* is the corresponding edge giving the minimum cover level. To compute *globalcover* we need to find the minimum of the two *globalcovers* of the children, except in case (v) of Lemma 3. Here the cluster path of the path child, say A, is not a part of the cluster path of C, and thus $\mathit{cover}(A)$ and $\mathit{minpathedge}(A)$, are candidates for *globalcover* and *minglobaledge* as well, and should therefore also be considered.

$$cover(C) = \begin{cases} \min(cover(A), cover(B)) & \text{if } C \text{ is a path cluster} \\ l_{max} & \text{otherwise} \end{cases}$$

$$minpathedge(C) = \begin{cases} minpathedge(A) & \text{if } C \text{ is a path cluster and } cover(A) \leq cover(B) \\ minpathedge(B) & \text{if } C \text{ is a path cluster and } cover(A) > cover(B) \\ null & \text{otherwise} \end{cases}$$

Define for $D \in \{A, B\}$

$$globalcover'(C, D) = \begin{cases} globalcover(D) & \text{if } \partial D \subseteq \pi(C) \text{ or } globalcover(D) \leq cover(D) \\ cover(D) & \text{otherwise} \end{cases}$$

And then let

$$globalcover(C) = \min(globalcover'(C, A), globalcover'(C, B))$$

$$minglobaledge(C) = \begin{cases} minglobaledge(A) & \text{if } globalcover'(C, A) \leq globalcover'(C, B) \\ minglobaledge(B) & \text{otherwise} \end{cases}$$

We have now shown how we can maintain the necessary information in our top tree, and thus how to implement operations *8.* to *11.*.

Analysis

By representing $cover(C)$ and $globalcover(C)$ as integers and $minpathedge(C)$ and $minglobal$-$edge(C)$ as pairs of endpoints, we leave the asymptotic memory usage unchanged. The computations of these values can clearly be done in constant time and are thus a $O(1)$ extension to create and merge.

3.3.5 Maintaining sizes

We will now show how to support operation *12.*, i.e. FindSize(u, v, i). We look to answer this operation efficiently by storing information about the sizes of clusters, at different levels. We introduce a *size* vector of size l_{max}, which is maintained in all clusters. Suppose we have a cluster C with boundary vertices u, v. Then $size(C)_i$ counts the size of the 2-edge connected component at level i if $u \ldots v$ were to be covered on level i. We shall now give a formal definition.

First, we define $pointset(C, u)$, for all clusters C and vertices on their cluster path

$u \in \pi(C)$. For $i \in \{0, \dots l_{max} - 1\}$ let

$$pointset(C,u)_i = \{v \in C \;\mid\; \pi(C) \cap u \dots v = \{u\} \wedge \mathsf{CoverLevel}(u,v) \geq i\}$$

Now let the size of the pointsets be stored in a single l_{max} sized vector, indexed from 0 to $l_{max} - 1$ as follows

$$pointsize(C,u)_i = |pointset(C,u)_i|$$

At last, we store the size of a cluster C, which is the sum of pointsizes over the cluster path. Thus $size(C)$ is an l_{max} sized vector indexed from 0 to $l_{max} - 1$.

$$size(C)_i = \sum_{u \in \pi(C)} pointsize(C,u)_i$$

These definitions are illustrated in Figure 3.4. With the *size* vector defined, are we able to easily answer the $\mathsf{FindSize}(u,v,i)$ operation as follows. Let $C = \mathsf{expose}(u,v)$. Then

$$\mathsf{FindSize}(u,v,i) = size(C)_i.$$

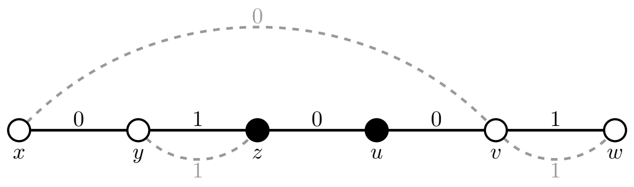


Figure 3.4: A cluster C with exposed vertices z and u marked by ●. Here $pointset(C,z) = \begin{bmatrix} \{x,y,z\} \\ \{y,z\} \end{bmatrix}$ and $pointsize(C,z) = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ while $pointset(C,u) = \begin{bmatrix} \{u,v,w\} \\ \{u\} \end{bmatrix}$ and $pointsize(C,u) = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$ and thus $size(C) = \begin{bmatrix} 6 \\ 3 \end{bmatrix}$.

We now seek to explain how to maintain the *size* vector in our top tree. In most cases it will be quite simple to maintain *size*, however, in one case it will be necessary to store some additional information.

Define first $partpath(C,u)$ of a cluster C with boundary vertices $u \in \partial C$. Then $partpath(C,u)$ is a $(l_{max}+2)$ sized vector of sets indexed from -1 to l_{max} given by

$$partpath(C,u)_i = \{v \in \pi(C) \mid \mathsf{CoverLevel}(u,v) = i\}$$

Now define $partsize(C,u)$ as a matrix of size $(l_{max}+2) \times l_{max}$, with rows indexed from -1 to l_{max} and columns indexed from 0 to $l_{max} - 1$. Then $partsize(C,u)$ is given by

$$partsize(C,u)_{i,j} = \sum_{v \in partpath(C,u)_i} pointsize(C,v)_j$$

The *partpath* and *partsize* definitions are illustrated in Figure 3.5.

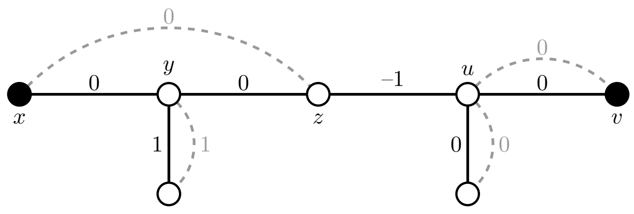


Figure 3.5: A cluster C with exposed vertices x and v marked by ●. Here $partpath(C,x) = \begin{bmatrix} \{u,v\} \\ \{y,z\} \\ \{\} \\ \{x\} \end{bmatrix}$ and $partsize(C,x) = \begin{bmatrix} 3 & 2 \\ 3 & 3 \\ 0 & 0 \\ 1 & 1 \end{bmatrix}$ while $partpath(C,v) = \begin{bmatrix} \{x,y,z\} \\ \{u\} \\ \{\} \\ \{v\} \end{bmatrix}$ and $partsize(C,v) = \begin{bmatrix} 4 & 4 \\ 2 & 1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix}$.

With the size and partsize of a cluster defined, can we proceed to describe how these can be maintained in the top tree. Suppose first we have some cluster C, which is being `create`d from an edge e with cover level $c(e)$.

We have three cases to consider, that is, when C has 0, 1 or 2 boundary vertices. Suppose $|\partial C| = 0$, then there is no cluster path, and hence the size is 0. Also there are no boundary vertices, so no partsizes exist. Hence we only set

$$size(C)_i = 0.$$

If however $|\partial C| = 1$, with $u \in \partial C$, we note that the cluster path consists of exactly u, and the single part of the cluster path is the l_{max} part containing only u itself. Thus

$$size(C)_i = \begin{cases} 2 & \text{if } i \leq c(e) \\ 1 & \text{otherwise} \end{cases}$$

$$partsize(C,u)_{i,j} = \begin{cases} size(C)_j & \text{if } i = l_{max} \\ 0 & \text{otherwise} \end{cases}$$

At last if $|\partial C| = 2$, with $u \in \partial C$ then the size is the sum of pointsizes of the 2 boundary vertices, i.e. 2 for all levels, and there exists a l_{max} and a $c(e)$ part for both of the boundary vertices. Thus

$$size(C)_i = 2$$

$$partsize(C,u)_{i,j} = \begin{cases} 1 & \text{if } i = l_{max} \vee i = c(e) \\ 0 & \text{otherwise} \end{cases}$$

With `create` out of the way, will we move on to `merge`. Assume we are merging child clusters A and B into their parent cluster C. By Lemma 3 we have five cases to consider. Case (*i*) is trivial as $|\partial C| = 0$ giving $size(C)_i = 0$, and no partsizes to compute. We now cover the cases (*ii*) to (*iv*). These cases are not difficult as $\partial A \cup \partial B \subseteq \pi(C)$. That is,

$\pi(A) \cup \pi(B) = \pi(C)$, and hence all $\mathit{pointsize}(C, v)$ for $v \in \pi(C)$ are counted in either the size vector for A or B. Let c be the central vertex of C, i.e. $\{c\} = \partial A \cap \partial B$. Then when $|\partial C| = 1$ let $a = c$. Otherwise $|\partial C| = 2$ and let $\{a, b\} = \partial C$ with $a \in \partial A$, $b \in \partial B$. Then

$$\mathit{size}(C)_i = \mathit{size}(A)_i + \mathit{size}(B)_i - 1$$

$$\mathit{partsize}(C,a)_{i,j} = \begin{cases} \mathit{partsize}(B,c)_{i,j} & \text{if } i < \mathit{cover}(A) \\ \mathit{partsize}(A,a)_{i,j} + \left(\sum_{k=i}^{l_{max}} \mathit{partsize}(B,c)_{k,j}\right) - 1 & \text{if } i = \mathit{cover}(A) \\ \mathit{partsize}(A,a)_{i,j} & \text{if } i > \mathit{cover}(A) \end{cases}$$

We note, that it is necessary to subtract 1, as the central vertex c is counted in both $\mathit{size}(A)$ and $\mathit{size}(B)$. This applies similarly to the partsizes.

We now proceed to the last case, i.e. case (v). Without loss of generality assume that $\partial A = \{a, c\}$ and $\partial B = \{c\}$. Now to find $\mathit{size}(C)_i$, we need to count the number of vertices u, with $\mathsf{CoverLevel}(a, u) \geq i$. To do this, we define

$$\mathit{diagsize}(A,a)_{i,j} = \begin{cases} \mathit{partsize}(A,a)_{i,j} & \text{if } i \leq j \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{reachsize}(B,c)_i = \begin{cases} \mathit{size}(B)_i - 1 & \text{if } i \leq \mathit{cover}(A) \\ 0 & \text{otherwise} \end{cases}$$

Observe that $\mathit{reachsize}(B, c)_i$ counts the number of vertices $u \in B \backslash \{c\}$, with $\mathsf{CoverLevel}(a, u) \geq i$. Note that we don't want to count c in $\mathit{reachsize}(B, c)$ as we count it in $\mathit{diagsize}(A, a)$. Note also that $\mathit{diagsize}(A, a)_i$ counts the size of each part, without vertices of levels not reachable due to a prior lower cover level. Hence

$$\mathit{size}(C)_i = \left(\sum_{j=-1}^{l_{max}} \mathit{diagsize}(A,a)_{j,i}\right) + \mathit{reachsize}(B,c)_i$$

$$\mathit{partsize}(C,a)_{i,j} = \begin{cases} \mathit{size}(C)_j & \text{if } i = l_{max} \\ 0 & \text{otherwise} \end{cases}$$

Thus we have shown how to maintain the size vector for a cluster and hence also how to answer operation *12.*.

Analysis

To maintain $\mathit{size}(C)$ we use an integer array of size l_{max} and to maintain $\mathit{partsize}$ we use at most two matrices of size $(l_{max} + 2) \times l_{max}$. Since $l_{max} = O(\log n)$ we use $O(\log^2 n)$ space per cluster, increasing the total space usage to $O(n \log^2 n)$. Both matrix and vector can be updated by simple iteration through the indices. This takes $O(\log^2 n)$ time.

We note that it is possible to update *partsize* in $O(\log n)$ by representing it as a binary tree keyed by the index as explained in [6], however we have chosen not to implement this due to time constraints. Additionally it is possible to further reduce this time by using a bit-trick by Thorup which we won't cover here.

3.3.6 Supporting find first label

We now show how to support operation *13.*, that is, $\mathsf{FindFirstLabel}(u, v, i)$. Recall $\mathsf{FindFirstLabel}(u, v, i)$, searches for a user label e with level i, associated with vertex w, where $\mathsf{CoverLevel}(w, meet(u, v, w)) \geq i$. Note that we want to minimize the distance $u \dots meet(u, v, w)$ if multiple results exist. We say w is the *best* vertex label, if the distance $u \dots meet(u, v, w)$ is minimized. We shall use a recursive binary search, through the clusters of the top tree, until we arrive at some leaf node of the top tree, where the corresponding edge has an endpoint with an existing user label at level i. Let $HasLabels(u, i)$ be $true$ when u has a user label at level i and $false$ otherwise. Also let $GetLabel(u, i)$ access a label from u at level i, if one such exists.

To support this search, we define a bit vector *incident* of size l_{max} indexed from 0 to $l_{max} - 1$ for all clusters. $incident(C)_i$ is 1 if a label w with $\mathsf{CoverLevel}(w, meet(u, v, w)) \geq i$ exists and 0 otherwise. We now give a formal definition. Let *pointincident* be a bit vector of size l_{max}, indexed from 0 to $l_{max} - 1$ for each vertex on the cluster path.

$$pointincident(C, u)_i = \begin{cases} 1 & \text{if } \exists v \in pointset(C, u)_i : HasLabels(v, i) \\ 0 & \text{otherwise} \end{cases}$$

$$incident(C)_i = \bigvee_{u \in \pi(C)} pointincident(C, u)_i$$

With the *incident* vector defined, can we answer the $\mathsf{FindFirstLabel}(u, v, i)$ operation as follows. Let $C = \mathsf{expose}(u, v)$. Then

$$\mathsf{FindFirstLabel}(u, v, i) = \mathsf{FindFirstLabel}(C, u, v, i)$$

with the recursive $\mathsf{FindFirstLabel}(C, u, v, i)$ defined as in Algorithm 3.

This procedure works as follows. If there are no user labels on level i in C then either $\mathsf{AccessLabel}(C, u, v, i)$ (if leaf node) returns $null$ or $incident(A)_i = 0$ and $incident(B)_i = 0$ and hence we return $null$, thus we never find a label if there are no incident labels. If there is an incident label and we are in a leaf node, we know that a label must exist in at least one of the endpoints u or v. Thus we return $GetLabel(u, i)$ if $HasLabels(u, i)$. Otherwise $HasLabels(v, i)$ must be true and we return $GetLabel(v, i)$.

If we are in an internal node, we must recursively search for a label in the child containing the best vertex label. By Lemma 3 we have a few cases to consider. Assume first

Algorithm 3 FindFirstLabel

```
function FindFirstLabel($C$,$u$,$v$,$i$)
    if $C$ is a leaf node then
        return AccessLabel($C, u, v, i$)
    $A, B \leftarrow$ left and right child of $C$
    $a_l \leftarrow$ leftmost boundary vertex of $A$
    $b_r \leftarrow$ rightmost boundary vertex of $B$
    if $A$ has a left boundary vertex and $a_l = u$ then
        $CloseChild \leftarrow A$
        $FarChild \leftarrow B$
    else if $B$ has a right boundary vertex and $b_r = u$ then
        $CloseChild \leftarrow B$
        $FarChild \leftarrow A$
    else
        if $|\partial A| = 1$ then
            $CloseChild \leftarrow A$
            $FarChild \leftarrow B$
        else
            $CloseChild \leftarrow B$
            $FarChild \leftarrow A$
    if $incident(CloseChild)_i = 1$ then
        $v' \leftarrow$ boundary vertex of $CloseChild$ not equal to $u$
        return FindFirstLabel($CloseChild, u, v', i$)
    else if $incident(FarChild)_i = 1$ then
        $u' \leftarrow$ boundary vertex of $FarChild$ closest to $u$
        $v' \leftarrow$ boundary vertex of $FarChild$ farthest from $u$
        return FindFirstLabel($FarChild, u', v', i$)
    return $null$

function AccessLabel($C$,$u$,$v$,$i$)
    if $HasLabels(u, i)$ then
        return $GetLabel(v, i)$
    else if $HasLabels(v, i)$ then
        return $GetLabel(v, i)$
    return $null$
```

that u is the left boundary vertex of the left child A of C. Clearly, we must search A if $incident(A)_i = 1$, as A must contain the best vertex label if it has one, otherwise we search B if $incident(B)_i = 1$. This applies similarly if u is the right boundary vertex of the right child.

If however u is in the middle we know that one of the children is a point cluster, say A, and thus any vertex label in A is best. If both children of C are point clusters, we can choose arbitrarily as they will both be best.

Note that it is necessary to perform a full_splay on the node which is visited last, to pay for our search through the splay top tree.

It only remains to show how one can maintain the *incident* vector in the top tree. We will include the approach for completeness, but it should be noted that the approach is close to identical to the approach used to maintain size. Therefore we will not comment on the approach further.

We introduce the matrix *partincident* of size $(l_{max}+2) \times l_{max}$, with rows indexed from -1 to l_{max} and columns indexed from 0 to $l_{max} - 1$. This matrix should help merge in case (v) of Lemma 3, similarly to how *partsize* was used. Define for all clusters C, and its boundary vertices $u \in \partial C$

$$partincident(C,u)_{i,j} = \bigvee_{v \in partpath(C,u)_i} pointincident(C,v)_j$$

Now suppose we create a cluster C from an edge $e = (u,v)$. If $|\partial C| = 0$, we have

$$incident(C)_i = 0.$$

If however $|\partial C| = 1$, with $u \in \partial C$ we have

$$incident(C)_i = \begin{cases} 1 & \text{if } HasLabels(u,i) \vee (HasLabels(v,i) \wedge c(e) \geq i) \\ 0 & \text{otherwise} \end{cases}$$

$$partincident(C,u)_{i,j} = \begin{cases} incident(C)_j & \text{if } i = l_{max} \\ 0 & \text{otherwise} \end{cases}$$

At last if $|\partial C| = 2$ with $u, v \in \partial C$ we have

$$incident(C)_i = \begin{cases} 1 & \text{if } HasLabels(u,i) \vee HasLabels(v,i). \\ 0 & \text{otherwise} \end{cases}$$

$$partincident(C,u)_{i,j} = \begin{cases} 1 & \text{if } (i = l_{max} \wedge HasLabels(u,i)) \vee (i = c(e) \wedge HasLabels(v,i)) \\ 0 & \text{otherwise} \end{cases}$$

Note that $partincident(C,v)$ is similarly computed.

Suppose now that we merge two clusters A, B into their parent cluster C. Recall the five cases of Lemma 3. Case (i) is trivial as $|\partial C| = 0$ and hence $incident(C)_i = 0$, with no partincidents to compute. For the cases (ii) to (iv), let c be the central vertex of C, i.e. $\{c\} = \partial A \cap \partial B$. Then when $|\partial C| = 1$ let $a = c$. Otherwise $|\partial C| = 2$ and let $\{a,b\} = \partial C$ with $a \in \partial A$, $b \in \partial B$. Then $a \in \partial A$ and $a \neq c$. Then

$$incident(C)_i = incident(A)_i \vee incident(B)_i$$

$$partincident(C,u)_{i,j} = \begin{cases} partincident(B,c)_{i,j} & \text{if } i < cover(A) \\ partincident(A,a)_{i,j} + \bigvee_{k=i}^{l_{max}} partincident(B,c)_{k,j} & \text{if } i = cover(A) \\ partincident(A,a)_{i,j} & \text{if } i > cover(A) \end{cases}$$

We cover now the last case, i.e. case (v). Assume w.l.o.g. that $\{a,c\} = \partial A$ and $\{c\} = \partial B$.

Then we define

$$diagincident(A,a)_{i,j} = \begin{cases} partincident(A,a)_{i,j} & \text{if } i \leq j \\ 0 & \text{otherwise} \end{cases}$$

$$reachincident(B,c)_{i,j} = \begin{cases} incident(B)_i & \text{if } i \leq cover(A) \\ 0 & \text{otherwise} \end{cases}$$

and update as follows

$$incident(C)_i = \left(\bigvee_{j=-1}^{l_{max}} diagincident_{j,i} \right) \vee reachincident(B,c)_i$$

$$partincident(C,a)_{i,j} = \begin{cases} size(C)_j & \text{if } i = l_{max} \\ 0 & \text{otherwise} \end{cases}$$

Observe that the approach is the exact same as in section 3.3.5 and thus the reasoning has been left out in this section.

Analysis

By representing *incident*(C) as a vector of boolean values and *partincident* as matrices of boolean values we achieve a result analogous to the result for the *size* and *partsize* data. In practice however it is possible to do better by noticing that it is almost impossible for l_{max} to be larger than the word length on traditional computer architectures. In this case we can pack these boolean values into a single word, meaning that *incident*(C) can be represented by a single integer and *partincident* can be represented by integer vectors of length $l_{max}+2$. This gives a space usage of $O(\log n)$ per cluster and an update time of $O(\log n)$

3.3.7 Supporting cover and uncover

We now show how to support operations *6.* and *7.*. We have for sections 3.3.3 to 3.3.6 not needed to split or destroy clusters, as all information in clusters were calculated bottom up, i.e. from leaf nodes and merged towards the root. This will however change as we want to support operations Cover(u,v,i) and Uncover(u,v,i). Naturally, we want to implement these operations utilizing our top trees, and hence we are looking to expose u and v and change some value in the root. Recall that we do not have the time to propagate information all the way down to leaf nodes. Hence we choose to store the information lazily and only propagate it downward, when necessary. For this, we introduce two auxiliary values of a cluster C, $\mathit{cover}^+(C)$ and $\mathit{cover}^-(C)$, which represent the following

$$\mathit{cover}^+(C) : \text{maximum level of a pending } \mathsf{Cover}\text{, or } -1,$$
$$\mathit{cover}^-(C) : \text{maximum level of a pending } \mathsf{Uncover}\text{, or } -1.$$

With these auxiliary values in place, will we implement $\mathsf{Cover}(u, v, i)$ by letting $C = \mathsf{expose}(u, v)$ and then calling $\mathsf{Cover}(C, i)$ which is defined below. That is

$$\mathsf{Cover}(u, v, i) = \mathsf{Cover}(C, i).$$

We implement $\mathsf{Uncover}(u, v, i)$ in a very similar way. Let $C = \mathsf{expose}(u, v)$. Then

$$\mathsf{Uncover}(u, v, i) = \mathsf{Uncover}(C, i).$$

The introduction of the values cover^+ and cover^- introduces information to our top tree which is not computed bottom-up but which instead should be propagated lazily down the top tree. This naturally introduces the need to extend the split and $\mathsf{destroy}$ operations. We note that $\mathsf{Uncover}(u, v, i)$ and $\mathsf{Cover}(u, v, i)$ operations change cover levels of edges on the cluster path of $C = \mathsf{expose}(u, v)$. Recall that changes in the cover level of edges on the cluster path affect *cover*, as this describes the minimum cover level on the cluster part. Also *partsize* and *partincident* are affected as the parts of the cluster path change. Note also that *size* and *incident* does not change, as these are independent of cover levels of edges on the cluster path.

Hence the $\mathsf{Cover}(u, v, i)$ and $\mathsf{Uncover}(u, v, i)$ operations can be propagated to the children D of C with boundary vertices $w \in \partial D$ by updating $\mathit{cover}(D)$, $\mathit{partsize}(D, w)$ and $\mathit{partincident}(D, w)$ according to $\mathit{cover}^+(C)$ and $\mathit{cover}^-(C)$, when a cluster C is split.

Before we describe split, we give a definition of $\mathsf{Cover}(C, l)$ and $\mathsf{Uncover}(C, l)$. For $\mathsf{Cover}(C, l)$ for all $u \in \partial C$ perform the following: if $l \geq \mathit{cover}^+(C)$ and $\mathit{cover}(C) < l$ update

$$\mathit{partsize}(C, u)_{i,j} = \begin{cases} 0 & \text{if } -1 \leq i < l \\ \sum_{k=-1}^{i-1} \mathit{partsize}(C, u)_{k,j} & \text{if } i = l \\ \mathit{partsize}(C, u)_{i,j} & \text{otherwise} \end{cases}$$

$$\mathit{partincident}(C, u)_{i,j} = \begin{cases} 0 & \text{if } -1 \leq i < l \\ \bigvee_{k=-1}^{i-1} \mathit{partincident}(C, u)_{k,j} & \text{if } i = l \\ \mathit{partincident}(C, u)_{i,j} & \text{otherwise} \end{cases}$$

Also update

$$\begin{aligned} cover(C) &= \max(cover(C), l) \\ cover^+(C) &= \max(cover^+(C), l) \\ cover^-(C) &= \begin{cases} -1 & \text{if } cover^-(C) \le l \\ cover^-(C) & \text{otherwise} \end{cases} \end{aligned}$$

This raises the cover level of C to l if it is not already higher. We note that this affects *partsize* and *partincident* as the parts of the cluster path change. When the cluster path is covered on level l, all levels $< l$ parts become a part of the level l part, and hence the need to sum and *or* the earlier *partsize* and *partincident* rows into row l.

Now we give the description of Uncover(C, l) with $u \in \partial C$. If $cover^+(C) \le l$ and $cover(C) \le l$ update

$$\begin{aligned} partsize(C,u)_{i,j} &= \begin{cases} \sum_{k=0}^{l} partsize(C,u)_{k,j} & \text{if } i = -1 \\ 0 & \text{if } 0 \le i \le l \\ partsize(C,u)_{i,j} & \text{otherwise} \end{cases} \\ partincident(C,u)_{i,j} &= \begin{cases} \bigvee_{k=0}^{l} partincident(C,u)_{k,j} & \text{if } i = -1 \\ 0 & \text{if } 0 \le i \le l \\ partincident(C,u)_{i,j} & \text{otherwise} \end{cases} \end{aligned}$$

and if $cover^+(C) > l$ update

$$\begin{aligned} cover^+(C) &= -1 \\ cover(C) &= \begin{cases} -1 & \text{if } cover(C) \le l \\ cover(C) & \text{otherwise} \end{cases} \\ cover^-(C) &= \begin{cases} -1 & \text{if } cover^-(C) \le l \\ cover^-(C) & \text{otherwise} \end{cases} \end{aligned}$$

Again we note that Uncover(C, l) lowers the cover level of all edges on the cluster path to -1 if their level is $\le l$. This makes all parts of level $\le l$ become a single level -1 part and hence the need to sum and *or* into the -1 row. Note that neither Cover nor Uncover changes *minpathedge*, *globalcover* and *minglobaledge* as these are not affected by the changes of cover levels on the cluster path.

Now we are ready to give the final description of split. In order to determine how we split, we note that Cover and Uncover operations only affect cover levels on the cluster path and hence only path clusters are affected. This leads to the following description of split.

Suppose C is split into its children A and B. If C is a path cluster, then for every path cluster $D \in \{A, B\}$ perform first Uncover$(D, cover^{-}(C))$ and then Cover$(D, cover^{+}(C))$.

It only remains to give a description of destroy. It is necessary when a path cluster C is destroyed into its edge e, to propagate the cover level into the edge if it was changed from above. Hence if C is path cluster, set

$$c(e) = cover(C).$$

And thus we showed how to support operations *6.* and *7.*, i.e. Cover(u, v, i) and Uncover(u, v, i).

Analysis

The addition of two integers makes for a constant addition of space. In the worst case, a Cover or Uncover must update the entire *partsize* matrix for both children which takes $O(\log^2 n)$ time.

3.3.8 Adding and removing edges

As mentioned in section 3.3.2, we also need to support operations *1.* to *4.*, i.e. Link, Cut, AddLabel and RemoveLabel. These operations add and remove tree and non-tree edges respectively.

To support Link, we use the standard operation link on the top tree, and set the cover level of the linked edge to -1. To support Cut, we use the standard cut on the top tree. Supporting AddLabel and RemoveLabel are more challenging.

When updating the associated non-tree edges of a given vertex label u, we note that the *incident* bit vectors of the clusters containing u might change. Hence when adding or removing a label, we must make sure to maintain the top tree accordingly. Let C be the leaf node consisting only of e, owning u. Then use the following procedure.

(i) First, full_splay C.

(ii) Then split top-down, on the root to C path.

(iii) Then destroy C.

(iv) Then update, add or remove relevant non-tree edges of u

(v) Then create C from e.

(vi) Finally, merge bottom-up on the C to root path.

And thus we have shown how to support operations *1.* to *4.*.

Analysis

As proved in [5] a full splay of a given cluster C takes $O(x \log n)$ amortized time and guarantees that the cluster has depth at most 4. As both merge, create split and destory are dominated by a factor of $O(\log^2 n)$ the update of the root-to-leaf path after a full splay takes $O(\log^2 n)$ meaning that an update of a vertex label owned by C takes $O(\log^3 n)+O(\log^2 n) = O(\log^3 n)$

3.3.9 Analysis of the algorithm

By combining the create, merge, split and destroy extensions from the previous sections we are ready to analyze the running time of the top tree procedures used in the algorithm. As merge, split use $O(\log^2 n)$ amortized time we get by Theorem 4 that link, cut, expose and deexpose take $O(\log^3 n)$ amortized time. This means that operations *1.* to *13.* can be implemented in $O(\log^3 n)$ amortized time using splay top trees.

Since TwoEdgeConnected and FindBridge can be implemented using a constant number of $O(\log^3 n)$ amortized time operations, these also take $O(\log^3 n)$ amortized time. This leaves us with the analysis of Insert and Delete.

However in order to do this we will first analyze Swap and Recover. Both of these procedures consider a number of non-tree edges, using $O(\log^3 n)$ amortized time to raise the level of the non-tree edges considered. Thus Swap and Recover take $O(\log^3 n) + k \cdot O(\log^3 n)$ amortized where k is the number of non-tree edges considered. We notice that a non-tree edge can be considered (and have its level raised) at most $l_{max} - 1 = O(\log n)$ times. This means that a given edge uses at most $O(\log^4 n)$ time in various Swap and Recover phases during the time between its insertion and deletion. We will amortize this by letting every edge pay for the $O(\log n)$ considerations used in Swap and Recover when it is inserted making the amortized cost of Swap and Recover $O(\log^3 n)$.

We now consider Insert. This procedure can be implemented in $O(\log^3 n)$ amortized time, however as we let this procedure pay for the edge's future $O(\log n)$ considerations in Swap and Recover making the amortized cost $O(\log^4 n)$. Delete also uses a constant number of $O(\log^3 n)$ amortized operations until the last step where Recover is called on at most $l_{max} - 1$ levels which takes at most $O(\log^4 n)$ amortized time in total.

Therefore we conclude that we can support TwoEdgeConnected and FindBridge in $O(\log^3 n)$ amortized time and Insert and Delete in $O(\log^4 n)$ amortized time.

3.4 Implementation

The implementation of the 2-edge connectivity algorithm corresponds more or less exactly to the description in section 3.3. Thus we comment only on the choice of data structures, design choices and main challenges in this section.

Layout of data in clusters

As we associate with each cluster a matrix of a fixed size we could have implemented these as two-dimensional `Arrays`. However, we decided to use a vector-based approach instead as two-dimensional setups and memory management is easier with these. This introduces a level of indirection on each cluster and probably contributes to many cache misses especially as we iterate over all elements reasonably often. A similar issue is that of where to store l_{max} for a given graph. In our implementation we chose to have a static integer associated with the cluster class, however, an unintended consequence of this is that only one `TwoEdgeConnected` data structure can exist at any given time as. An alternative is to simply store the value of l_{max} in each cluster, however, this was not done to save memory. This extra memory is probably not a problem, at least not compared to a potential cache miss of accessing the value of l_{max}. This, however, is all conjecture, and further testing would have to be done to judge the best approach.

A difference from the description of the 2-edge connectivity algorithm compared to [6] is that the cover level, find size and find first label extensions are merged into one large data structure instead of three smaller and separate ones. This was mainly done to not juggle three top trees at the same time and having to expose and deexpose each tree correctly.

In every vertex we needed to store a list of edges on each level incident to that vertex. A natural choice of data structure would be a linked list as this supports deletion in the middle in constant time. However as the ordering of the edges on each level was not important we opted instead to use vectors as this allows for easier memory management and less room for error compared to manually updating and deleting raw pointers.

Using our library

Originally, we designed the top tree library to support user-defined data, in clusters, edges and vertices, through the use of generics. This approach works great for both clusters and edges, and the implementation on top of this was quite simple.

What we did not consider was the custom vertex data, and how this is stored in a top tree. The designed library works great for non-changing vertex data, but when vertex data changes, e.g. with the addition of the AddLabel and RemoveLabel operations, we run into problems. A vertex can be part of many leaf nodes, possibly all throughout the top tree, and thus a change in data, introduces the need to recompute the entire top tree, compared to a single leaf-to-root path.

In this algorithm, we solved this problem, by letting a single leaf node own the vertex and then letting the vertex data only affect the owning cluster. This solution was not complicated itself, but as the concept of changing vertex data, might not be rare for more advanced algorithms, our top tree library would benefit greatly from a solution supporting dynamically changing vertex data.

4 | Testing

With the splay top tree library and algorithms described and implemented will we now cover the testing of our implementation. In this chapter, we give a brief comment on the test of correctness, and will then benchmark our implementation against already existing implementations.

4.1 Correctness

As our top tree library functions as a single unit, we have not tested individual methods like rotations and splaying in isolated testing environments. We have not tested these small components, due to the difficulty of accessing non-public methods, requiring us, to bypass the encapsulation and set up our data structure by accessing its data directly which is, as far as we know, bad practice.

Therefore, we chose to test the top tree as a single unit using only the public interface. We did this by implementing a couple of simple algorithms for which we could generate problems and solutions easily by hand. These include the maximum weight problem and the diameter problem. We also extended the maximum weight problem to answer queries about the maximum weight in a given component. In addition to these tests, we also used asserts to verify that selected invariants of the splay top tree are satisfied.

During the development of the 2-edge connectivity algorithm, we tested smaller cluster extensions seperately (including cover level, find size and find first label), using the top tree directly. The tests were performed to make sure the individual extensions behaved as expected before continuing development. Afterwards we tested only the external operations of the 2-edge connectivity algorithm, on various small examples generated by hand. For the 2-edge connectivity algorithm, we also generated larger tests of three types.

(*i*) Large cycle: A large cycle was generated and it was verified that a deletion of any edge would make the graph not 2-edge connected. It was also verified that reinsertion of said edge would make the graph 2-edge connected.

(*ii*) Large components: Two large disconnected completely connected graphs were generated and it was verified that connecting these with two edges would make them 2-edge connected. It was also verified that deletion of only one edge makes them not 2-edge connected,

(*iii*) Large components 2: Again two large disconnected completely connected graphs were generated and a number of edges were made between them. It was verified that the components were 2-edge connected. The edges were them removed one by one and it was verified that the two components remained 2-edge connected until the second to last edge was removed.

At last it is relevant to mention the usage of Valgrind, throughout the development of the splay top tree, and the algorithms upon it. Valgrind runs a given program in a controlled environment keeping track of allocated memory and variables among other things. This allowed us to handle and rewrite code with undefined behaviour and memory leaks.

4.2 Benchmarking

In this section, we will present and discuss an experimental comparison of different top tree implementations. We will compare our implementation of top trees, to two different top tree implementaions made by Jiři Setnika in [8]. Unfortunately, we were not able to obtain the top tree implementations from [11]. Thus the implementations we have tested are

(*i*) Splay Top Trees (Implemented in this paper based on [5])

(*ii*) Self-adjusting Top Trees (implemented in [8] based on [12])

(*iii*) Topology Tree Top Trees (implemented in [8] based on [1])

All implementations are written in `C++` and compiled with `g++` 12.2.1 with `-O3` optimization enabled. All experiments have been run on the DTU HPC cluster with an Intel Xeon 2660v3 with 512 KB of L1 cache, 2 MB of L2 cache, 25 MB of L3 cache and 16GB of DDR4 memory. The source code is publicly available at `https://github.com/MagnusSiegumfeldt/top-trees`. The test suite is also publicly available at `https://github.com/Inocxh/top-trees-bench`.The source code is publicly available at

4.2.1 Methodology

Each of the three above-mentioned top tree data structures has been tested on the two algorithms covered in sections 3.1 and 3.3, i.e. the maximum weight and 2-edge connectivity problems. The test suite designed in [8] is used as a basis of our tests. Though we have adjusted the testing methodology to suit our needs. As the running time of a single program can be quite volatile on a modern system we have the following procedure when measuring the running time for a specific program.

Each measurement is preceded by a number of warmup executions, to make sure that the effects of caching behavior and hardware acceleration are mitigated in the measurement. We allow an initialization phase which is not timed and is used for building graphs and generating operation sequences. Each measurement is performed three times and the median of the results is recorded into the final dataset. This hopefully filters out outliers where e.g. a background process interrupts the measurement. Multiple data points are gathered for each size in order to account for lucky and unlucky inputs. The average running time per operation, for each size is plotted.

Maximum weight

This experiment starts with a randomly generated tree of n vertices and a randomly generated sequence of link, cut, AddWeight[1] and MaxWeight operations. AddWeight and MaxWeight both have a 25% probability while the ratio of link and cut operations will vary. We then measure the running time of each implementation with the same initial tree and operation sequence. We start our tests at a large graph (450 vertices for 1000 operations), such that we do not end with an empty tree, if we get an unlucky sequence of Insert and Deletes.

The final results are from a run starting from $n = 450$ which increments n by a factor of 1.25 until each data point takes longer than one hour to collect. We do 5 warmup rounds per measurement and we collect 8 data points per size of n.

2-edge connectivity

The experiment starts with a randomly generated graph of n vertices and $3n$ edges and a randomly generated sequence of Insert, Delete and TwoEdgeConnected. Similar to the maximum weight tests, we will vary the Insert/Delete ratio in different tests. We were not able to test the algorithms on as large graphs as with maximum weight, as some implementations of this algorithm, were to slow to allow this. We did however also test our algorithm separately, allowing for larger tests.

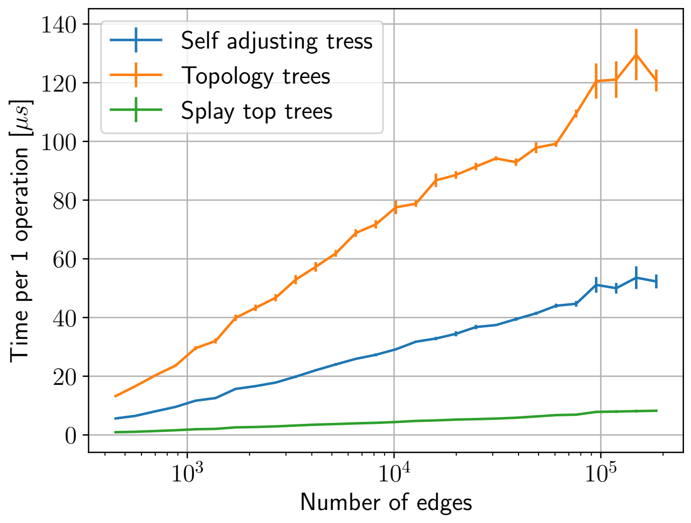
The results below are from a run starting with $n = 100$ and again incrementing n with a factor of 1.25. As each execution takes a long time, we decided to limit the number of warmup rounds to 1 per measurement and 6 data points per size of n.

4.2.2 Results

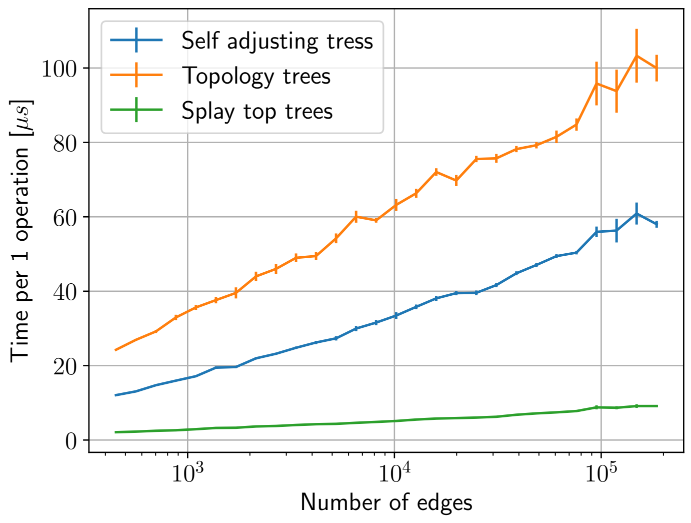
Maximum weight

The experiments described above were run and we now show the results in Figure 4.1. Our results show that the splay top trees perform very well compared to both the topology tree and self-adjusting tree based approaches. Interestingly splay top trees and self-adjusting trees perform best on workloads with a lot of cuts. We suspect that this is due to the smaller tree size in general. However, this is not the case for topology which has the best performance on sequences with a lot of links. It seems that the splay top tree and self-adjusting tree approaches are more robust against different workloads, not varying much across different ratios. The splay top tree is the faster choice of the two in our tests.

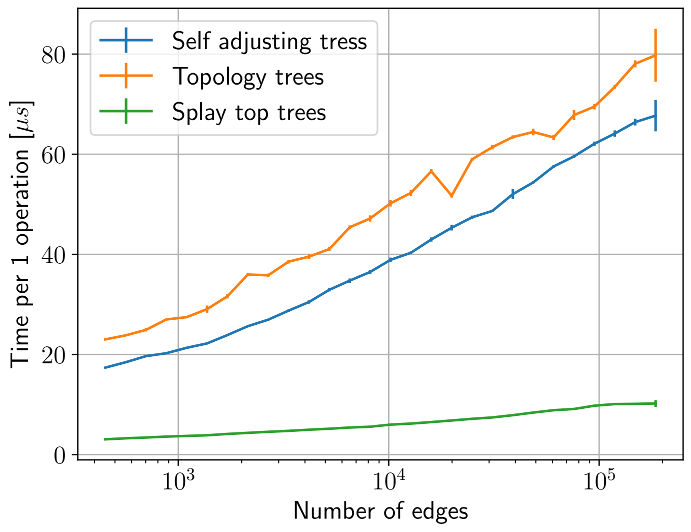
[1] AddWeight is not discussed in this project, but is implemented by introducing simple split and destroy extensions, covered in Theorem 3 of [1].



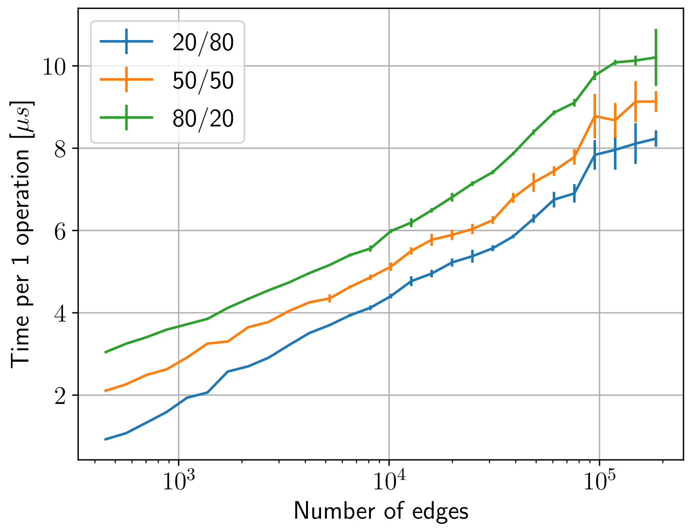
(a) Comparison of top tree implementations on 20/80 ratios.



(b) Comparison of top tree implementations on 50/50 ratios.



(c) Comparison of top tree implementations on 80/20 ratios.



(d) Comparison of 20/80, 50/50 and 80/20 ratios on splay top trees.

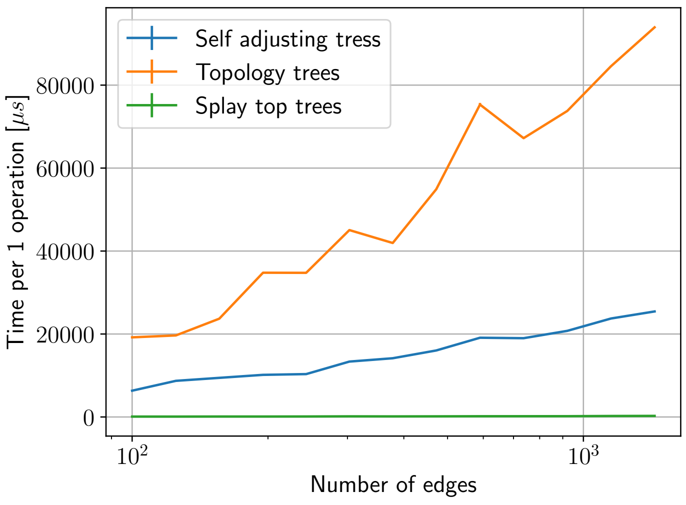
Figure 4.1: Maximum weight experiments with different link/cut ratios.

2-edge connectivity

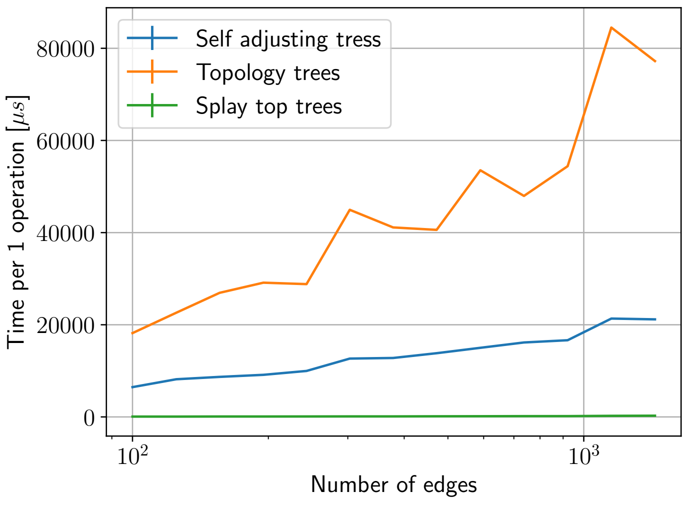
We come now to the comparison of the 2-edge connectivity implementations. We want to stress from the start, that we could not compare the three implementations as thoroughly as we intended, as both the topology tree implementation and the self-adjusting tree implementation, compute incorrect results and throw errors for larger inputs. As we are not certain what state the two data structures are in, and what computations are performed, it is difficult to draw reasonable conclusions from the comparisons. The results we were able to obtain are shown in Figure 4.2. We believe the tests, need more operations per measurement and more measurements in general, in order to mitigate the large jumps occurring throughout the tests. This will however not make the results more comparable, as they by all means need to compute correct results. Pursuit of more samples for each size was not done as the current setup was already taking around 6 hours per run. All this aside and under the assumption that the running time is somewhat representative to that of a correct implementation, we see that the splay top tree implementation outperforms the topology trees and the self-adjusting trees by a significant margin.

Testing the splay top tree against itself with differing ratios of Insert and Delete yields some more reliable results. Unlike the maximum weight tests we see that this implementation performs best when there are more Inserts. This is most likely because the potential has less chance of being paid during subsequent Deletes. Due to the nature of the testing setup we most likely never see edges with a cover level close to l_{max}, in further work we would need to test with more operations for larger graphs in order to see the full effect of the amortization.

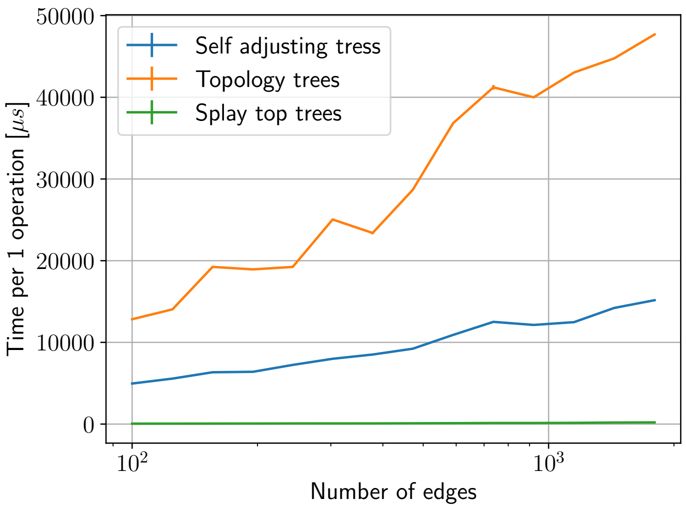
Our single valid measurement is the one, comparing our implementation at different ratios of Insert and Delete operations. Unlike the maximum weight tests, we note that our implementation performs best when a lot of Inserts occur, especially for smaller graphs. This can possibly be explained, by the time used on Swap and Recover.



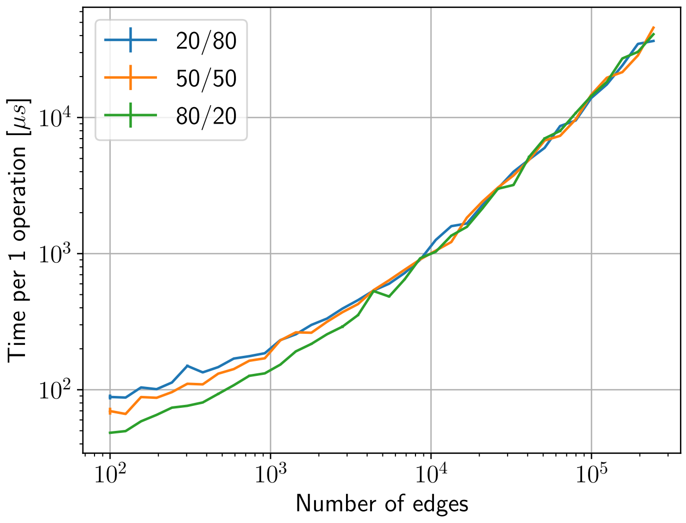
(a) Comparison of top tree implementations on 20/80 ratios.



(b) Comparison of top tree implementations on 50/50 ratios.



(c) Comparison of top tree implementations on 80/20 ratios.



(d) Comparison of 20/80, 50/50 and 80/20 ratios on splay top trees. Note the log-log axis.

Figure 4.2: Two-edge connectivity experiments with different Insert/Delete ratios.

5 | Conclusion

In this project we have implemented a splay top tree library and shown that it is easy to implement a range of different algorithms using the simple interface the library provides. Although the splay top tree library, does provide a lot of features, there are still areas, which could be improved. These areas include dynamically changing vertex data, allowing inheritance on the user-defined clusters and providing better error messages.

Throughout the project we covered two simpler algorithms: maximum edge weight, and diameter, which were proved and analyzed. Furthermore, we gave an in-depth description of a 2-edge connectivity algorithm, which translates easily to an implementation. We managed to implement the algorithm using $O(\log^4 n)$ amortized time per operation. There exist a few optimizations which could be interesting to implement and test if we had more time.

The performance of our maximum weight implementation and the designed splay top tree library performs better in our tests, compared to implementations of topology tree and self-adjusting tree based top tree implementations. This is the most accurate comparison of each top tree implementation as the user-defined operations do next to no work leaving the majority of the running time to top tree operations. Unfortunately we were not able to draw many certain conclusions regarding our 2-edge connectivity algorithm, due to the uncertainty of the self-adjusting tree and topology tree-based 2-edge algorithms, which throw errors on larger graphs.

If time had allowed it we would have liked to test the algorithms on more structured graphs, instead of the highly variable and completely random sequence of operations in this. Furthermore it would have been interesting to see a comparasion to a working implemetation of the 2-edge connectivity algorithm. [5] also describes an alternative way to implement `expose` which avoids a full splay. This could also be interesting to look into.

References

[1] Stephen Alstrup et al. *Maintaining Information in Fully-Dynamic Trees with Top Trees.* 2003. DOI: 10.48550/ARXIV.CS/0310065. URL: https://arxiv.org/abs/cs/0310065.

[2] Stephen Alstrup et al. "Minimizing diameters of dynamic trees". In: *Automata, Languages and Programming.* Ed. by Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 270–280. ISBN: 978-3-540-69194-5.

[3] Greg N. Frederickson. "Data Structures for On-Line Updating of Minimum Spanning Trees". In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing.* STOC '83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 252–257. ISBN: 0897910990. DOI: 10.1145/800061.808754. URL: https://doi.org/10.1145/800061.808754.

[4] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *J. ACM* 48 (July 2001), pp. 723–760. DOI: 10.1145/276698.276715.

[5] Jacob Holm, Eva Rotenberg, and Alice Ryhl. "Splay Top Trees". In: *Symposium on Simplicity in Algorithms (SOSA).* Society for Industrial and Applied Mathematics, Jan. 2023, pp. 305–331. DOI: 10.1137/1.9781611977585.ch28. URL: https://doi.org/10.1137%5C%2F1.9781611977585.ch28.

[6] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. *Dynamic Bridge-Finding in $\tilde{O}(\log^2 n)$ Amortized Time.* 2017. DOI: 10.48550/ARXIV.1707.06311. URL: https://arxiv.org/abs/1707.06311.

[7] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. "Faster Fully-Dynamic Minimum Spanning Forest". In: *CoRR* abs/1407.6832 (2014). arXiv: 1407.6832. URL: http://arxiv.org/abs/1407.6832.

[8] Jiří Setnika. *Comparison of Top trees implementations.* 2018. URL: https://dspace.cuni.cz/handle/20.500.11956/98673.

[9] Daniel D. Sleator and Robert Endre Tarjan. "A Data Structure for Dynamic Trees". In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing.* STOC '81. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 1981, pp. 114–122. ISBN: 9781450373920. DOI: 10.1145/800076.802464. URL: https://doi.org/10.1145/800076.802464.

[10] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-Adjusting Binary Search Trees". In: *J. ACM* 32.3 (July 1985), pp. 652–686. ISSN: 0004-5411. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835.

[11] Robert E. Tarjan and Renato F. Werneck. "Dynamic Trees in Practice". In: *Experimental Algorithms.* Ed. by Camil Demetrescu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 80–93. ISBN: 978-3-540-72845-0.

[12] Robert E. Tarjan and Renato F. Werneck. "Self-Adjusting Top Trees". In: *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA '05. Vancouver,

British Columbia: Society for Industrial and Applied Mathematics, 2005, pp. 813–822. isbn: 0898715857.

A | **Swap** and **Recover** implementation

Algorithm 4 Swap and Recover implementations.

```
function Swap($e$):
    $(u, v) \leftarrow e$
    $\alpha \leftarrow$ CoverLevel$(u, v)$
    $Cut(e)$
    $e' \leftarrow$ FindReplacement$(u, v, \alpha)$
    $(x, y) \leftarrow e'$
    RemoveLabel$(x, e)$
    RemoveLabel$(y, e)$
    Link$(x, y)$
    $l(e) \leftarrow \alpha$
    AddLabel$(u, e)$
    AddLabel$(v, e)$
    Cover$(u, v, \alpha)$

function Recover($u, v, i$):
    $s \leftarrow \lfloor$FindSize$(u, v, i)/2\rfloor$
    RecoverPhase$(u, v, i, s)$
    RecoverPhase$(v, u, i, s)$

function FindReplacement($u, v, i$):
    $s_u \leftarrow$ FindSize$(u, u, i)$
    $s_v \leftarrow$ FindSize$(v, v, i)$
    if $s_u \leq s_v$ then
        return RecoverPhase$(u, u, i, s_u)$
    else
        return RecoverPhase$(v, v, i, s_v)$

function RecoverPhase($u, v, i$):
    $e \leftarrow$ FindFirstLabel$(u, v, i)$
    while $e \neq null$ do
        $(q, r) \leftarrow e$
        if $\neg$Connected$(q, r)$ then
            return $e$
        if FindSize$(q, r, i + 1) \leq s$ then
            RemoveLabel$(q, e)$
            RemoveLabel$(r, e)$
            $l(e) \leftarrow i + 1$
            AddLabel$(q, e)$
            AddLabel$(r, e)$
            Cover$(q, r, i + 1)$
        else
```

```
            $\textsc{Cover}(q, r, i)$
            **return** *null*
        $e \leftarrow \textsc{FindFirstLabel}(u, v, i)$
    **return** *null*
```