Splay top trees and 2-edge connectivity

Andreas Nikolaj Valkær, au617767, 201804962

Master's Thesis, Computer Science
June, 2023
Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

The main goal of this thesis is to consider data structures and algorithms for dynamic graph problems using splay top trees as the central data structure. This is done through the theory of splay trees, splay top trees and a dynamic 2-edge connectivity algorithm, which are considered in detail. An amortized analysis of splay trees is provided to gain insight into concepts reused for splay top trees. The correctness and amortized runtime of splay top trees are then considered, with the addition of user-defined operations skipped in the publication of splay top trees. The user-defined operations are heavily used in the applications of top trees and are mentioned in the original paper on top trees. The 2-edge connectivity algorithm uses top trees as a black box. Thus, no modifications were required directly in the algorithm but relied on reintroducing user-defined operations in splay top trees.

The Splay top tree data structure was implemented and verified using an incremental minimum spanning trees algorithm compared to the static Kruskal algorithm. Finally, 2-edge connectivity was implemented and verified against a static algorithm and the theoretical results through experiments. The experiments suggested that the modified analysis of splay top trees is correct. An experimental comparison of two different implementations behaved differently than expected, as the advanced version showed no improvements even though it was conjectured as a significant speed-up.

Acknowledgements

I wish to express my sincere gratitude towards my supervisor Gerth Stølting Brodal for making this project possible. He always had time to answer my questions, and was a huge help in discussing problems, solutions and ideas during the weekly meetings.

Andreas Nikolaj Valkær,
Aarhus, June 13, 2023.

Contents

1 Introduction

Dynamic graph algorithms is an extensively researched area within computer science that considers a graph $G = (V, E)$ that changes, e.g. through edge insertions and deletions. Dynamic algorithms aim to maintain enough information about a given problem so that queries, i.e., connectivity, 2-edge connectivity, bi-connectivity, shortest path, and many more, can be efficiently answered without recomputing the result from scratch[5, 8]. This calls for dynamic data structures that can be updated efficiently, e.g., top trees[1], where the following operations are usually considered:

- PREPROCESS: Preprocess the graph G
- INSERT: Insert an edge or vertex in the graph G
- DELETE: Delete an edge or vertex in the graph G
- QUERY: Query the data structure for some property

Generally, we are interested in three different measurements for efficiency: (i) The initial preprocess time; (ii) the update time for insertions and deletions, which can differ between the two, and (iii) the query time. The runtimes can be expected, worst-case or amortized depending on the algorithm and analysis applied. Additionally, we may require the algorithm to be deterministic, which can negatively impact the achieved runtimes or significantly increase the complexity.

Dynamic graph algorithms can be sorted into one of three groups depending on which update operations it supports. It may be fully dynamic, supporting both insertion and deletion; Incremental only supporting insertions or decremental, supporting only deletion, with the last 2 also being referred to as semi-dynamic.

Throughout this thesis, the focus will be on fully dynamic data structures and algorithms. First, splay trees are used to introduce amortized analysis and the splaying concept[17]. The concept of splaying was recently reused by Holm et al. to give a direct proof for the top tree data structure, used in many state-of-the-art graph algorithms, achieving amortized runtimes of $O(\log n)$ matching the asymptotic bounds of current implementations. Top trees were initially proved by a reduction to topology trees by Alstrup et al. [1], with many implementations still being based on topology trees today[9].

The top tree data structure allows the user to store user-defined information about subsets of the spanning tree it is built upon, which are used to answer queries quickly. As the top tree data structure dynamically changes, so does the stored subsets, thus, requiring a recomputation of the information stored. These recomputations happen through user-defined operations COMBINE and SPLIT, skipped by Holm et al. in their pseudocode and analysis of splay top trees[9]. The user-defined operations have been readded and considered throughout the analysis. Splay top trees were implemented, and to verify the implementation, an incremental algorithm for maintaining spanning trees was implemented and compared to Kruskal.

A dynamic 2-edge connectivity algorithm by Holm et al. using top trees is then considered in detail, reproducing its correctness and amortized analysis. The dynamic algorithm considered takes $O(\log^4 n)$ amortized time for the INSERT and DELETE operations, and $O(\log^3 n)$ amortized time for QUERY with a space usage of $(m + n \log^2 n)$, with n being the number of vertices and m the number of edges[6, 7, 8]. The algorithm has been improved multiple times over the years, achieving better update and query times with a smaller space usage [8, 10, 19]. The original algorithm proposed by Holm et al. has been implemented using splay top trees and compared to a static 2-edge connectivity algorithm to verify the correctness of the implementation. Finally, an

experimental evaluation of the implementation was made to verify the theoretical observations. The experiments supported the theory, except for a conjectured result by Holm et al.[9], where a more advanced version of splay top trees provided no improvement.

2 Splay Trees

In this section, splay trees and their properties are considered. Splay trees are a version of binary search trees(BST) that take recently accessed elements and rotate them up the tree so they are located closer to the root. The advantage of this is that repeated access to the same elements is faster because they are located near the root, but it can also lead to worse worst-case times than a standard balanced binary search tree as seen in Table 1. The reason for the worse worst-case times is the fact splay trees may be unbalanced. Thus, in the worst case, the splay tree can have a height and runtime of $O(n)$[17].

Operation	Amortized	Worst case
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(n)$

Table 1: Splay tree runtimes

2.1 Splay tree operations

This section considers the SEARCH, INSERT and DELETE operations on splay trees. The splay tree operations can be implemented in multiple ways, but can all be implemented as an extension of their BST implementations, an example of which can be seen in Algorithm 1. The INSERT operation is implemented by performing the BST insertion, followed by an operation which moves the newly inserted node to the root. This is handled by the SPLAY operation, which is discussed in detail in Section 2.1.1. SEARCH and DELETE can be implemented in a similar fashion as seen in Algorithm 2 and Algorithm 3.

As all three operations are based on the SPLAY operation, this will be the initial focus of the analysis in Section 2.2. The runtimes of INSERT, SEARCH and DELETE is then considered in Section 2.2.1.

Algorithm 1 INSERT(x)

```
1: BST insertion(x)
2: SPLAY(x)
```

Algorithm 2 SEARCH(x)

```
1: BST search(x)
2: if x ∈ T then
3:     SPLAY(x)
4: else
5:     SPLAY(last accessed node in BST search)
```

Algorithm 3 DELETE(x)

1: BST delete(x)
2: SPLAY(parent of removed node)

2.1.1 Splaying

The splay operation moves the desired node x to the root by repeated applications of rotations through the zig, zig-zig and zig-zag operations described below. Let $p(x)$ denote the parent of x in splay tree T.

- Zig: This single rotation only happens if $p(x)$ is the root. The rotation is performed on the parent $p(x)$. An example of this can be seen in Figure 1.

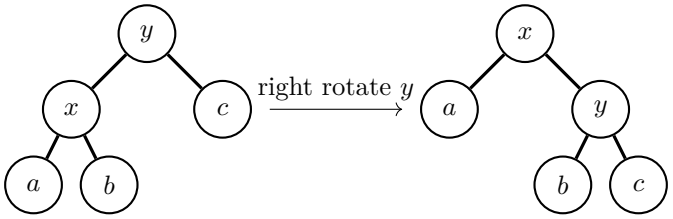


Figure 1: zig(x) example

- Zig-zig: This operation is performed if the element x and $p(x)$ are both left(right) children on the path to the root. In the case of x and $p(x)$ both being left children, we perform two right rotations, first on $p(p(x))$ and then on $p(x)$. An example of this can be seen in Figure 2.

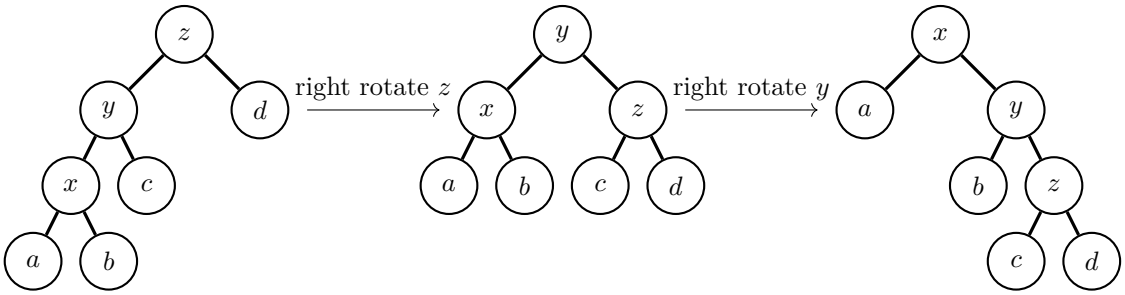


Figure 2: zig-zig(x) example

- Zig-zag: This is performed if the element x is a right child and $p(x)$ is a left child (or swapped). Here we first perform a left rotation on $p(x)$, followed by a right rotation on $p(p(x))$. An example of this can be seen in Figure 3.

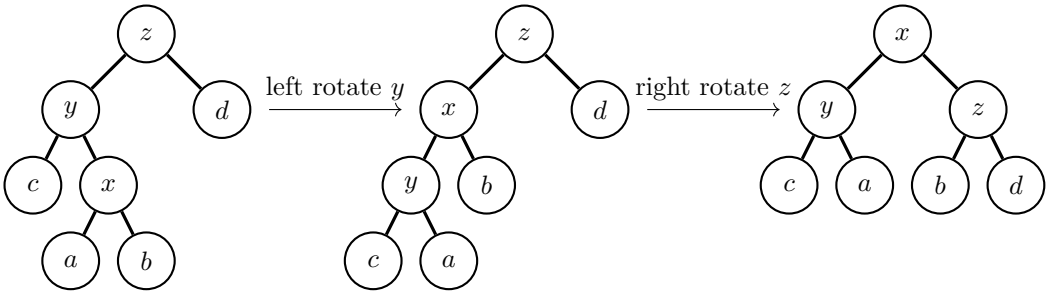


Figure 3: Zig-zag(x) example

Mirrored versions, zag-zag and zag-zig, are also required; these are, however, handled identically to zig-zig and zig-zag. Therefore, these will be ignored throughout.

2.2 Analysis

We wish to consider the amortized runtime of zig, zig-zig and zig-zag. To do so, a potential function $\phi(\cdot)$ is needed. The amortized runtime a is then equal to the actual time t and the difference in potential before and after the operation, e.g. $a = t + \phi' - \phi$, where ϕ and ϕ' denote the potential before and after respectively.

The amortized analysis can be performed on a weighted tree, where each node is assigned some positive weight $w(x) > 0$. But for this analysis, assume the weight of every node is 1. The size $s(x)$ is then defined to be the size of the subtree rooted at x. From the size, we then define the rank of a node to be $r(x) = \log_2(s(x))$. The potential function of a splay tree T is then defined to be the sum of all ranks $\phi(T) = \sum_{x \in T} r(x)$.

The total runtime of a sequence of m operations is then, $\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i + \phi_{i-1} - \phi_i) = \phi_0 - \phi_m + \sum_{i=1}^{m} a_i$. The potential function ϕ is ≥ 0 for any splay tree, with $\phi = 0$ for an empty tree. This means that $\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i$ if the sequence of operations starts on an empty tree, which allows us to give an upper bound on the actual runtime by computing the amortized time of each operation.

One downside of using amortized analysis is that even though the average runtime of the operation might be low, there may be some worst-case occurrences that are very slow. This may be an issue in some applications if they depend on quick worst-case response times.

Lemma 2.1. *(Lemma 1 in [17]) Amortized runtime to splay x in a tree T with root node x' is at most $3(r(x') - r(x)) + 1$.*

Proof.

Case zig(x):

This is the simplest case, as only two elements change rank x and its parent $y = p(x)$ as seen in Figure 1. Let $r(x)$ and $r'(x)$ denote the rank before and after the zig operation. Similarly, let $s(x)$ and $s'(x)$ denote the tree size before and after. We note that only a single rotation is performed. Thus, let the actual time be equal to 1.

$$
\begin{aligned}
a &= t + \phi' - \phi && \text{definition} \\
&= 1 + r'(y) + r'(x) - r(y) - r(x) \\
&\leq 1 + r'(x) - r(x) && \text{follows from } r(y) \geq r'(y) \\
&\leq 1 + 3(r'(x) - r(x)) && \text{follows from } r'(x) \geq r(x)
\end{aligned}
$$

The final step is performed only to get the constant of 3, to get some similarity with the zig-zig and zig-zag result.

Case zig-zig(x)**:** Zig-zig performs two rotations. As such, the actual time is 2. Let $x, y = p(x)$ and $z = p(y)$ be the elements whose rank changes as seen in Figure 2.

$$
\begin{aligned}
a &= t + \phi' - \phi && \text{definition} \\
&= 2 + r'(z) + r'(y) + r'(x) - r(z) - r(y) - r(x) \\
&= 2 + r'(z) + r'(y) - r(y) - r(x) && r'(x) = r(z) \\
&\leq 2 + r'(z) + r'(x) - 2r(x) && r'(x) \geq r'(y) \text{ and } r(y) \geq r(x) \\
&\leq 2 + (2r'(x) - r(x) - 2) + r'(x) - 2r(x) && \text{See concavity argument below} \\
&= 3(r'(x) - r(x))
\end{aligned}
$$

The third equality follows from the fact that the new root of this subtree is x, which now has the same size as the subtree rooted at z had before the rotation, i.e. $s'(x) = s(z)$ as seen in Figure 2.

The last inequality follows from the concavity of the log function, i.e. $\frac{\log(a)+\log(b)}{2} \leq \log \frac{a+b}{2}$. Which gives the following inequality

$$
\begin{aligned}
\frac{r(x) + r'(z)}{2} &= \frac{\log(s(x)) + \log(s'(z))}{2} \\
&\leq \log \frac{s(x) + s'(z)}{2} \\
&\leq \log \frac{s(x) + s'(z)}{2} && \text{Since } s(x) + s'(z) \leq s'(x) \\
&\leq \log \frac{s'(x)}{2} \\
&= r'(x) - 1
\end{aligned}
$$

The second inequality follows directly from Figure 2. We see that $s(x) + s'(z) = w(x) + s(a) + s(b) + w(z) + s(c) + s(d)$ and $s'(x) = w(x) + w(y) + w(z) + s(a) + s(b) + s(c) + s(d)$, thus $s'(x) - s(x) - s'(z) = w(y)$, which means $s(x) + s'(z) \leq s'(x)$.

Thus, by the concavity of the log function, we have $r'(z) \leq 2r'(x) - r(x) - 2$.

Case zig-zag(x)**:** The actual time is 2 for the zig-zag case. Let $x, y = p(x)$ and $z = p(y)$ be the elements whose rank changes as seen in Figure 3.

$$
\begin{aligned}
a &= 2 + r'(z) + r'(y) + r'(x) - r(z) - r(y) - r(x) && \text{definition} \\
&= 2 + r'(z) + r'(y) - r(y) - r(x) && r'(x) = r(z) \\
&\leq 2 + r'(z) + r'(y) - 2r(x) && r(y) \geq r(x) \\
&\leq 2(r'(x) - r(x)) && \text{Concavity arguement} \\
&\leq 3(r'(x) - r(x)) && r'(x) > r(x)
\end{aligned}
$$

The second to last inequality follows from the concavity of the log function again, where $r'(z) \leq 2r'(x) - r'(y) - 2$ can be obtained through a similar analysis.

With the above three cases, the desired result can be shown using a telescoping sum, adding up all the individual splay-step costs from moving an element to the root.

$$
\begin{aligned}
\text{amortized cost} &= \sum_i cost(splay\_step_i) \\
&\leq \sum_i (3r'(x)) - r(x)) + 1 && \text{Note: } +1 \text{ from the potential zig step} \\
&= 3(r(root) - r(x)) + 1
\end{aligned}
$$

□

When considering an unweighted tree, $r(root) = \log(n)$, and $r(x) \leq \log(n)$. Which gives an amortized cost for splaying an element to the root of $\leq 3\log(n) + 1 = O(\log(n))$. With this, it remains to be argued that we can also perform the INSERT, DELETE and SEARCH operations in $O(\log(n))$ amortized time.

2.2.1 runtime analysis of search, insert and delete operations

Let us consider the operations described in Section 2.1 and their amortized runtimes. Because the operation is defined as their BST version followed by splaying, we can consider their amortized cost as follows:

$$
\begin{aligned}
a &= \text{cost of splay} + \text{BST operation cost} + \text{potential change from BST operation} \\
&= O(\log n) + \text{BST operation cost} + \text{potential change from BST operation} \\
&= O(\log n) + \text{potential change from BST operation}
\end{aligned}
$$

When splaying an element to the root, some potential is freed up to pay for some of the rotations performed in the zig-zig and zig-zag operations. The same potential can be used to pay for the additional levels searched throughout the SEARCH, INSERT and DELETE operations in BST that occur due to an unbalanced tree, as these operations are dominated by traversing down the tree to the location of the node. The insertion and deletion change a few pointers, which takes $O(1)$ time, and can thus be ignored. Therefore, the cost of the BST operations is at most $O(\log n)$ as well.

Thus, the only part missing is the potential change from the operations.

- Search ⇒ Does not change the tree. Thus there is no potential change besides from the splaying
- Delete ⇒ Decreases the potential (The ranks of all ancestors become smaller), thus potential change from operation ≤ 0.
- Insert ⇒ Increases the potential of possibly n ancestors, and thus we need to handle this case.

Lemma 2.2. *The potential increases by at most $O(\log n)$ when a new node is inserted.*

Proof. When inserting an element, the potential of all ancestors increases. Let $p = p_0 \to p_1 \to \ldots \to p_l$ be the path to the root. Each has size $s(p_i)$ before the insertion and $s'(p_i)$ afterwards. As only a single element is inserted, we know that $s'(p_i) = s(p_i) + 1$, and the potential change of the entire tree is then:

$$\begin{aligned}
\sum_{i=0}^{l} r'(p_i) - \sum_{i=1}^{l} r(p_i) &= \sum_{i=1}^{l} r'(p_i) - r(p_i) \\
&= \sum_{i=1}^{l} \log(s'(p_i)) - \log(s(p_i) \\
&= \sum_{i=1}^{l} \log\left(\frac{s'(p_i)}{s(p_i)}\right) \\
&= \sum_{i=1}^{l} \log\left(\frac{s(p_i)+1}{s(p_i)}\right) \\
&\leq \sum_{i=1}^{l} \log\left(\frac{i+1}{i}\right) \\
&= O(\log n)
\end{aligned}$$

When a new element is inserted, it always becomes a leaf, and thus the new node p_0 has $s'(p_0) = 1$ and $r'(s(p_0)) = 0$ and therefore can be ignored in the sums above. □

Given this, we have an amortized runtime for INSERT, SEARCH and DELETE of $O(\log n)$.

2.3 Alternative versions of splay trees

Depending on your application's needs and requirements, different splay trees exist. Specifically, we may require a space-efficient version that sacrifices some runtime. Such a version with only two pointers per node is shown in Figure 4[17].

Likewise, one may not wish to move the latest accessed element to the root, as this may require a lot of restructuring of the splay tree. Thus, a new concept was introduced called semi-splaying. The semi-splaying heuristic is to only perform a single rotation during zig-zig, namely, the first step of rotating with $p(p(x))$ and then continuing the splaying from $y = p(x)$, essentially the zig-zig step stops at the middle figure in Figure 2. This ensures that the depth of x is at most half the starting depth after splaying, as the remaining operations are left alone[17].

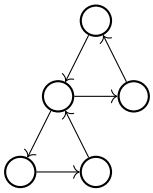


Figure 4: Memory efficient splay tree

Whether semi-splaying was an improvement to fully-splaying was assumed to be dependent on the access pattern. Intuitively, the main difference is how long it takes to adapt to new access patterns. Semi-splaying takes longer to adjust, performing fewer rotations and writings to memory. It has later been experimentally tested, and it turns out that for a wide variety of access patterns semi splaying practically outperforms splaying the node the entire way and can, thus, be used as a replacement in many cases[3].

3 Top trees

This section gives a detailed description of splay top trees with the user-defined operations reintroduced. An analysis of the amortized runtime of the splay top tree data structure is given too. Some of the lemmas in this section are restated from the original paper on splay top trees with additional proof details.

The top tree $\mathcal{T}$ data structure is a rooted binary tree that can be combined with any explicitly maintained spanning tree T. Top trees are designed to be used in various graph algorithms, i.e. minimum spanning tree, 2-edge connectivity and bi-connectivity [8], thus, demanding a certain amount of versatility. This is accomplished by storing user-defined information in the top tree, which can be tailored to the problem.

Let a cluster be a connected set of edges in T. A summary is stored for each cluster, containing the user-defined information. These summaries are then stored in the nodes of $\mathcal{T}$, representing the clusters of T. A cluster C can contain up to 2 boundary vertices, either flagged as a boundary by the user or incident to something outside the cluster. If two clusters share a boundary vertex, they can be combined to define a larger cluster. An example of this can be seen in Figure 5, where the clusters A and B can be combined.

A vertex flagged as a boundary vertex is also called exposed, and the importance of these should become clear throughout.

In the rest of this paper, nodes will be used to address elements in the top tree and vertices for elements in the spanning tree. The relationship between nodes is expressed using parent, children, sibling, or uncle. There is a one-to-one mapping between clusters and nodes. Thus, let these terms be used interchangeably.

Cluster information: Let leaf nodes in $\mathcal{T}$ represent a single edge in the spanning tree. These are the smallest clusters, with summary information which only depends on a single edge.

The internal nodes of $\mathcal{T}$ represent a union of its children, thus requiring them to share a boundary vertex. The summary information may now depend on multiple edges. Recomputing the summary from scratch for each node may be an expensive operation. Therefore, the user-defined COMBINE operation is reintroduced. COMBINE computes the cluster information of an internal

node by combining the cluster information of its children. The COMBINE operation is not a novelty, as it plays an essential part in the 2-edge connectivity algorithm considered in Section 4 and any other algorithm using top trees. However, likely for simplicity reasons COMBINE was ignored in the analysis of splay top trees by Holm et al., even though it was used in the implementation they provided [9].

The COMBINE operation varies depending on the problem, with examples of COMBINE shown in Table 2. A slightly more in-depth example is given to gain familiarity with the COMBINE operation. Let $\pi(C)$ define the path between boundary vertices of C in T. For clusters with fewer than 2 boundary vertices, $\pi(C)$ is undefined. Now consider the problem of maintaining the maximum edge weight of $\pi(C)$ in a spanning tree as the summary information.

Let C be a cluster with 0 or 1 boundary vertices, called a point cluster. Since $\pi(C)$ is undefined, let $C.maxWeight$ be undefined. Instead, let C be a cluster with 2 boundary vertices, called a path cluster. Since leaf nodes only represent a single edge e in their cluster C, $\pi(C)$ can only contain e. Therefore, let $C.maxWeight = e.weight$.

Let an internal node represent cluster C. We can now consider how COMBINE should operate in different cases. Let COMBINE be called on C with boundary vertices a and b, and children A and B. Let A be a path cluster with boundaries a and b, and B a point cluster with boundary b as seen in Figure 5. In this case, we can set $C.maxWeight = A.maxWeight$ as the boundary vertices of C and A are the same, which means $\pi(C) = \pi(A)$.

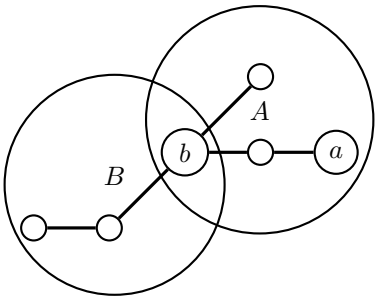


Figure 5: Spanning tree T with a clusters A and B with boundary vertices a, b and b respectively

Let COMBINE be called on cluster C with boundary vertices a and b, with children A and B, as seen in Figure 6. Let A and B be path clusters with boundary vertices a, c and b, c respectively. $C.maxweight$ can then be computed as, $C.maxWeight = \max(A.maxWeight, B.maxWeight)$.

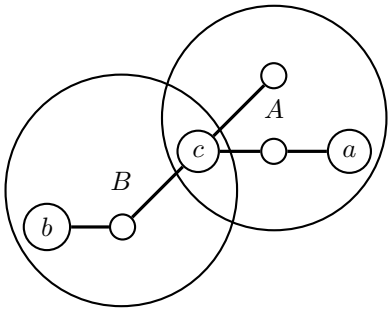


Figure 6: Spanning tree T with a clusters A and $B$with boundary vertices a, c and b, c respectively

The attentive reader may ask, what if A and B are point clusters? When this is the case, it is impossible for C to be a path cluster. Since A and B are children of C, they must share a boundary vertex c, which means C can only have c as a potential boundary vertex. Thus, $\pi(C)$ is always undefined for this case. All cases for COMBINE can be seen in Table 3.

Without the ability for users to explicitly mark vertices as boundary vertices, the max edge weight problem is rather dull. The cluster represented by the top trees root node contains all vertices in the spanning tree. Thus, the only boundary vertices of this cluster are vertices exposed by the user. By exposing vertices u and v, the root will contain the maximum edge weight between these vertices. Thus, the user can ask for the maximum edge weight on the path between any two vertices in the spanning tree.

Not all problems require two boundary vertices in the root to answer queries. But generally, with 2 exposed vertices, we answer queries about the path between them, i.e., the maximum edge weight. If we expose a single vertex, we could view the spanning tree as rooted in this vertex and answer queries about its height. Finally, if we don't expose any vertices, the queries are often unrelated to specific vertices, i.e., the size or diameter of the spanning tree.

Problem	Case handled in Table 3	Computation
Size of spanning tree	1,2,3,4 and 5	$C.size = A.size + B.size$
Maximum edge weight	1	$C.maxWeight = \max\{A.maxWeight, B.maxWeight\}$
Maximum edge weight	2	$C.maxWeight = A.maxWeight$
Maximum edge weight	3,4 and 5	$C.maxWeight = undefined$
Diameter of spanning tree	1	$d_{ab} = A.d_{ac} + B.d_{bc}$ $d_a = \max(A, d_a, A.d_{ac} + B.d_c)$ $d_b = \max(B.d_b, B.d_{bc} + A.d_c)$ $d = \max(A.d, B.d, A.d_c + B.d_c)$
Diameter of spanning tree	2	$d_{ab} = A.d_{ab}$ $d_a = \max(A, d_a, A.d_{ab} + B.d_b)$ $d_b = \max(B.d_b, A.d_b)$ $d = \max(A.d, B.d, A.d_c + B.d_c)$
Diameter of spanning tree	3	$d_{ab} = d_b = undefined$ $d_a = \max(A, d_a, A.d_{ab} + B.d_b)$ $d = \max(A.d, B.d, A.d_c + B.d_c)$
Diameter of spanning tree	4	$d_{ab} = d_b = undefined$ $d_a = \max(A, d_a, B.d_a)$ $d = \max(A.d, B.d, A.d_c + B.d_c)$
Diameter of spanning tree	5	$d_{ab} = d_a = d_b = undefined$ $d = \max(A.d, B.d, A.d_c + B.d_c)$

Table 2: Examples of COMBINE computations for different problems for the cases presented in Table 3

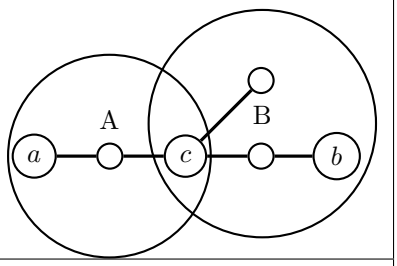
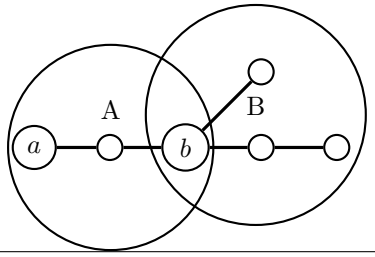
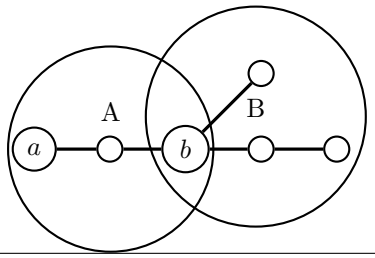
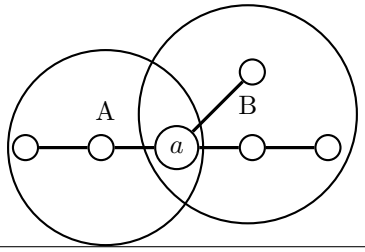
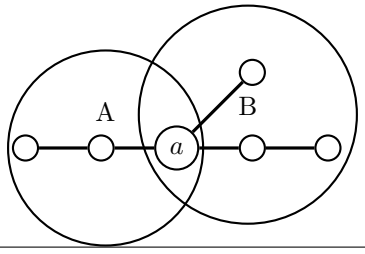
Case number	Description	Drawing
1	C is a path cluster with boundary vertices a, b and children A, B. A and B are path clusters with a, c and b, c as boundary vertices respectively	
2	C is a path cluster with boundary vertices a, b and children A, B. A is a path clusters with boundaries a, b and B is a point cluster with b as boundary vertex	
3	C is a point cluster with boundary vertices a and children A, B. A is a path clusters with boundaries a, b and B is a point cluster with b as boundary vertex	
4	C is a point cluster with boundary vertices a and children A, B. A and B are point clusters with a as their boundary vertex.	
5	C is a point cluster with no boundary vertex and children A, B. A and B are point clusters with a as their boundary vertex.	

Table 3: All cases COMBINE can come across when computing the cluster information

Some problems, i.e. 2-edge connectivity and bi-connectivity, require large amounts of information stored in each summary along with a more complex COMBINE operation, which we will see in Sec-

tion 4 when the problem of 2-edge connectivity is considered. These problems may also require that information be passed down the tree. This is supported in the original top tree implementation with the user-defined operation SPLIT, used by Holm et al. in their 2-edge connectivity algorithm[8]. Thus, throughout the analysis, COMBINE and SPLIT have been reintroduced in preparation for the 2-edge connectivity algorithm.

3.1 Orientation invariant

In this section, an orientation invariant is introduced. The orientation invariant simplifies the analysis, reducing the number of cases that need to be considered.

When considering the children of an internal node representing C, it is convenient to consider one child as the left and the other as the right. This allows us to treat boundary vertices as left, middle or right boundary vertices of C by considering which child it comes from. A left boundary vertex of a C must also be a boundary vertex of the cluster represented by the left child, with a similar definition for right boundary vertices. A boundary vertex can only be considered a middle boundary vertex if it is the central vertex shared between both children.

From the above orientation on boundary vertices, we can define a cluster to have a leftmost boundary vertex if a left or middle boundary vertex exists. If both a left and middle boundary vertex exists, the leftmost boundary vertex is the left boundary vertex. Let the definition of the rightmost boundary vertices be similar.

The orientation of boundary vertices in leaf nodes with only 1 edge e is a bit special. Since there are no children, the left and rightmost boundary vertices are defined based on the endpoints of e. Let the left and right boundary vertex be the left and right endpoint of e, respectively. Finally, middle boundary vertices are undefined for leaf nodes.

With the above defined, the orientation invariant proposed by Holm et al. is "*For any internal node C, the leftmost boundary vertex of the right child B and the rightmost boundary vertex of the left child A must both exist and be equal to the central vertex of C*"[9]. Throughout this thesis, this orientation invariant will be used as well. For an example of a top tree satisfying the orientation invariant, see Figure 7. Leaf nodes a and b must be connected through vertex w, the rightmost boundary vertex of a, and the leftmost boundary vertex of b. The cluster A must have a rightmost boundary vertex, which is the middle boundary vertex w since the other endpoint x of b has no other incident edge. With A having a middle boundary vertex, it also has a leftmost boundary vertex. Thus, A could be the right child as well.

(a) Top tree (b) Spanning tree

Figure 7: Orientation invariant example

3.2 Data structure

This section outlines the required information stored for nodes in the top tree and some requirements for the spanning tree.

The data structure for the top tree should must the following information in each node.

- A pointer to the parent node
- For internal nodes, pointers to its children
- For leaf nodes, a pointer to the edge
- A counter storing the number of boundary vertices
- A boolean or flip bit storing whether the subtree should be flipped.
- User-defined cluster information

Finally, the spanning tree itself must be stored and support a set of operations. It should be possible to insert and delete edges in the spanning tree, retrieve an arbitrary edge incident to a vertex v, and a way to determine if $degree(v) \geq 2$. These operations can be supported in $O(1)$ time.

3.3 Interface of top tree

This section first outlines external operations for top trees and then presents the internal operations needed to implement the external ones. The format for presenting the operations is heavily inspired by Holm et al. [9].

Let T_v refer to the spanning tree containing vertex v, $\mathcal{T}_v$ refer to the top tree built on T_v and e an edge with endpoints u and v. The interface for interacting with top trees consists of the following operation.

- EXPOSE(v): Mark v as exposed and returns the root of $\mathcal{T}_v$. Requires that T_v have at most 1 exposed vertex and v is not currently exposed.

  This operation may cause a restructuring of $\mathcal{T}_v$ to ensure no cluster contains more than 2 boundary vertices.

- DEEXPOSE(v): Mark v as not exposed and returns the root of $\mathcal{T}_v$. Requires that v is currently exposed.

  This operation may cause a restructuring of $\mathcal{T}_v$.

- LINK(u, v): u and v are newly connected in the spanning forest by edge e. LINK creates a leaf node representing e, and returns the root of the new tree. Requires 0 exposed vertices in both T_u and T_v, which needs to be different trees.

- CUT(e): Deletes the leaf node representing edge e, which is newly deleted in the spanning tree. Requires that the tree $T_u = T_v$ contains no exposed vertices.

  In the original top tree interface, CUT returns the roots of $\mathcal{T}_u$ and $\mathcal{T}_v$. This is not required for 2-edge connectivity. Thus, let CUT return nothing or both roots.

All external operations may force some clusters to be recomputed by COMBINE. Thus, all of them should ensure that any information stored further up the tree is propagated downwards using split beforehand. The EXPOSE, DEEXPOSE and LINK immediately call the same operation

findConsumingNode, which will be defined soon. For simplicity, let findConsumingNode handle the split calls for those operations.

cut also causes recomputations of some clusters. Let cut ensure all required information is correctly propagated downwards for itself.

Consider the following internal operations. Those defined from hasLeftBoundary and onwards will be considered in greater detail throughout the analysis. Let *node* be some node in $\mathcal{T}_v$.

- isPoint(*node*) / isPath(*node*): Returns information about whether *node* is a point or path cluster.
- flip(*node*): To maintain the orientation invariant when the structure changes, we may sometimes flip entire subtrees. This can be implemented in many ways, but for efficiency, let it be represented by the flip bit or boolean, allowing the subtree to be considered flipped without actually flipping it.
- pushFlip(*node*): This ensures that the flip bit or boolean is false. The flip is performed by swapping the children and calling flip on both of them. This can be performed in $O(1)$ time.
- hasLeftBoundary(*node*)/ hasMiddleBoundary(*node*)/ hasRightBoundary(*node*): Returns whether a *node* has a left, middle or right boundary vertex respectively.
- rotateUp(*node*): This rotation operation differs from the rotation seen as part of splay trees. We only allow rotateUp to be called if it results in a valid top tree, which requires all clusters to be valid. A cluster is considered valid if it is a connected set of edges, with at most 2 boundary vertices.

  rotateUp also makes no guarantees that the ordering of leaves is preserved.

  Finally, the rotation itself is defined differently. rotateUp swaps the position of *node* and *sibling*(*parent*(*node*)) as seen in Figure 8 and adjusts other necessary values and pointers to ensure everything is satisfied.

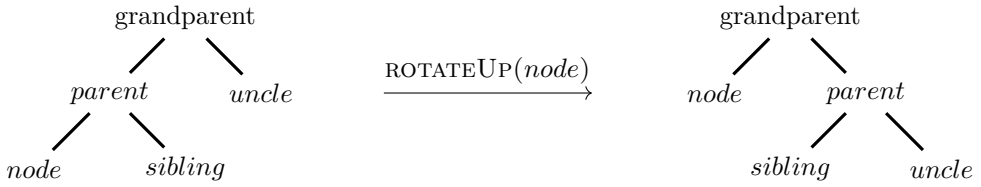


Figure 8: Example of rotation

- semiSplayStep(*node*): This operation calls rotateUp($\cdot$) one or two times to restructure the top tree reducing the depth of *node* by 1. Finally, it returns the root of the changed subtree.

  This operation is a central part of splay trees. Conceptually, it can be thought of as the rotation, as it decreases the depth of *node* by 1. As part of the analysis, it is shown that if $depth(node) \geq 5$, this operation always succeeds and reduces the depth.

  With the result of semiSplayStep always succeeding if $depth(node) \geq 5$, it is possible to implement splay-like behaviour. Intuitively, if we repeatedly call semiSplayStep(*node*),

node is moved towards the root. Unfortunately, similar to splay trees, more caution is required to make the amortized analysis work.

- SEMISPLAY(*node*): This operation is inspired by the semi-splay concept of splay trees. The depth of *node* is reduced by some constant by repeated calls to SEMISPLAYSTEP.

  The potential change is $O(\log n) - \Omega(depth(node))$, where $depth(node)$ is the original depth of *node*. Thus, this operation allows for $O(depth)$ work to be performed by other operations for free.

- FULLSPLAY(*node*): This operation has the same potential change as SEMISPLAY, but guarentees the $depth(node)$ is reduced to ≤ 4.

  This operation considers two calls to SEMISPLAYSTEP at a time, which can be compared to how zig-zig and zig-zag are used in splay trees.

- FINDCONSUMINGNODE(v): Returns the consuming node of the vertex v, defined as the first cluster containing all edges incident to v.

  Recall that this operation has been assigned the additional task of calling SPLIT as it tries to locate the consuming node. This was chosen as all operations calling this can force recomputations on the path between the consuming node and the root.

3.4 Detecting boundary vertices

In this section, a correctness argument for HASLEFTBOUNDARY, HASMIDDLEBOUNDARY and HASRIGHTBOUNDARY is given, as these operations are central for maintaining the orientation invariant.

The process of detecting boundary vertices is identical for the left and right vertices. Thus, only HASLEFTBOUNDARY(*node*) is included.

Let *node* be a leaf node and recall that left boundary vertices were defined as the left endpoint being exposed or incident to something else. Thus, we can check if the left endpoint of *node* satisfies one of these requirements. The operation should consider if *node* is conceptually flipped by its flip bit/boolean.

Consider the case where *node* is an internal node. *node* can only have a left boundary vertex if the left child is a path cluster. For contradiction, assume the left child was a point cluster and a left boundary vertex existed. By the orientation invariant, we require a rightmost boundary vertex to exist for the left child and be shared with its sibling. Thus, a contradiction is reached, as the only boundary vertex was assumed to be a left boundary vertex, which can not be the rightmost boundary vertex. Thus, a left boundary vertex can not exist if the left child is a point cluster.

Assume the left child is a path cluster. Then the rightmost boundary vertex is again shared with its sibling. Thus, the remaining boundary vertex must be a left boundary vertex for *node* as it comes from the left child.

Algorithm 4 hasLeftBoundary(*node*)

```
1: if isLeafNode(node) then
2:     if node.flip then
3:         endpoint = node.edge.endpoints[1]
4:         return endpoint.exposed || degree(endpoint) ≥ 2
5:     else
6:         endpoint = node.edge.endpoints[0]
7:         return endpoint.exposed || degree(endpoint) ≥ 2
8: else
9:     if node.flip then
10:        return isPath(node.children[1])
11:    else
12:        return isPath(node.children[0])
```

Let *node* be a leaf node or an internal node with 0 boundary vertices. In both these cases, no middle boundary vertices can exist. Thus, consider the case where *node* contains at least one boundary vertex. Each path child of *node* contributes a non-middle boundary vertex by the same argument used for left boundary vertices in internal nodes.

Thus, by subtracting 1 from *node.numBoundary* for each path child, we can check for the existence of a middle boundary node using the following expression, with $false = 0$ and $true = 1$.

$$node.numBoundary - \text{isPath}(left\ child) - \text{isPath}(right\ child) > 0$$

Algorithm 5 hasMiddleBoundary(*node*)

```
1: if node.isLeaf || node.numBoundary == 0 then
2:     return false
3: else
4:     return node.numBoundary - isPath(left child) - isPath(right child) > 0
```

By a quick observation of Algorithm 4 and Algorithm 5, the operations do not change the top tree and run in $O(1)$ actual time. Thus, the amortized cost of these operations is $O(1)$ with no potential change.

3.5 Rotations

In this section, the rotateUp operation is first considered through a correctness argument, followed by an analysis of when rotations are valid in the top tree. The correctness of rotateUp is only shown for rotations proven valid in Section 3.5.3

rotateUp plays a significant role in top trees, allowing other internal operations to modify the top tree. These modifications are performed by swapping a node with its parent's sibling. As briefly stated in the introduction to the rotateUp operation, it is only allowed when all clusters remain valid, with the only new cluster being $sibling(node) \cup sibling(parent(node))$. Finally, the ordering of nodes may shift throughout the modification of the top tree. Thus, there is no guarantee for the orientations of the nodes after a call to rotateUp, except that they satisfy the orientation invariant.

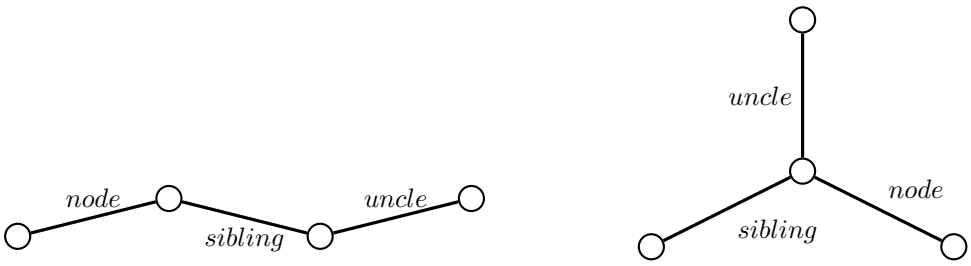
Algorithm 6 rotateUp(*node*)

```
1: parent = node.parent
2: grandparent = parent.parent
3: sibling = sibling(node)
4: uncle = sibling(parent)
5:
6: pushFlip(grandparent)
7: pushFlip(parent)
8:
9: uncle_is_left_child = grandparent.children[0] == uncle
10: sibling_is_left_child = parent.children[0] == sibling
11: to_same_side = uncle_is_left_child == sibling_is_left_child
12: sibling_is_path = isPath(sibling)
13: uncle_is_path = isPath(uncle)
14: grandparent_is_path = isPath(grandparent)
15:
16: if to_same_side and sibling_is_path then
17: // Rotation on path
18:     grandparent_middle = hasMiddleBoundary(grandparent)
19:     new_parent_is_path = grandparent_middle or uncle_is_path
20:     flip_new_parent = false
21:     new_parent_is_path = false
22:     if grandparent_middle and !grandparent_is_path then
23:         ggp = grandparent.parent
24:         if ggp != null then
25:             gp_is_left_child = ggp.children[0] == grandparent
26:             flip_grandparent = gp_is_left_child == uncle_is_left_child
27: else
28: // Rotation on star
29:     if !to_same_side then
30:         new_parent_is_path = sibling_is_path or uncle_is_path
31:         flip_new_parent = sibling_is_path
32:         flip_grandparent = sibling_is_path
33:         node.flip = !node.flip
34:     else
35:         new_parent_is_path = uncle_is_path
36:         flip_new_parent = false
37:         flip_grandparent = false
38:         sibling.flip = !sibling.flip
39: // Update pointers
40: if uncle_is_left_child then
41:     parent.children[1] = sibling
42:     parent.children[0] = uncle
43:     grandparent.children[1] = node
44:     grandparent.children[0] = parent
45: else
46:     parent.children[0] = sibling
47:     parent.children[1] = uncle
48:     grandparent.children[0] = node
49:     grandparent.children[1] = parent
50: parent.flip = flip_new_parent
51: parent.boundary = if new_parent_is_path {2} else {1}
52:
53: grandparent.flip = flip_grandparent
54:
55: node.parent = grandparent
56: uncle.parent = parent
57:
58: combine(parent)
```

3.5.1 Correctness

Let the configurations presented in Figure 9 and Figure 10 be those considered for the rotateUp algorithm. Let the top tree representations be considered as they are, meaning no nodes are considered flipped when calling rotateUp. The goal is to argue how an implementation of rotateUp could handle combinations of these configurations. The combinations of these configurations will be handled on a case-by-case basis throughout this section.

The combination of Figure 9a and Figure 10b is impossible.



(a) Rotation on a path (b) Rotation on a star

Figure 9: Spanning tree configurations that can be observed during a rotation. Each edge may be a larger cluster consisting of multiple edges

(a) Sibling and uncle to the same side (b) Sibling and uncle to different sides

Figure 10: Top tree configurations that can be observed during rotations.

When performing a rotation, rotateUp needs to determine how many boundary vertices the new cluster $sibling(node) \cup sibling(parent(node))$ has and maintain the orientation invariant of all nodes. Finally, the parent and children pointers should be updated accordingly.

Impossibility of combining Figure 9a and Figure 10b

Consider the combination of Figure 9a and Figure 10b. This case is impossible, as *node* and *uncle* can not avoid being connected. Assume *node* is a path cluster, then *uncle* must connect to the rightmost boundary vertex of *parent*, which comes from *node*. Instead, let *node* be a point cluster. In this case, the *parent* must have a middle boundary vertex, which the uncle is connected to because the middle boundary vertex is the rightmost boundary vertex of *parent*. The middle boundary vertex of *parent* is the node shared between *node* and *sibling*, which means *node* and *uncle* are connected.

Path structure

Case 1: This case considers the combination of Figure 9a and Figure 10a. Let Figure 11 show how the rotation impacts the top tree.

Assume *sibling* was a point cluster. Then both *node* and *uncle* must be connected to the same endpoint of *sibling*. However, this contradicts the layout assumed in Figure 9a. Thus, *sibling* must be a path cluster in this case.

With this knowledge of *sibling*, consider how many boundary vertices the new *parent′* cluster has. Because *sibling* is a path cluster, it must contribute a boundary vertex to *parent′*. If *uncle* is a path cluster, it also contributes a boundary vertex. Finally, if *uncle* is a point cluster, the vertex v shared by *sibling* and *uncle* may be a boundary vertex if it is incident to something outside the cluster. This is only the case if v was a middle boundary vertex of *grandparent* before the rotation. Thus, *parent′* is a path cluster if the *uncle* is or if *grandparent* had a middle boundary vertex before the rotation.

Finally, rotateUp should maintain the orientation invariant. The order of *node*, *sibling* and *uncle* was maintained. Thus, the orientation of *node*, *sibling*, *uncle* and *parent′* is maintained without flipping any of them.

Assume the *grandparent* had a single middle boundary vertex before the rotation, the vertex shared by *sibling* and *uncle*. After the rotation, the central vertex of *grandparent* is shared by *node* and *sibling*. Thus, the boundary vertex of *grandparent* is now either a left or right boundary vertex. Therefore, rotateUp must ensure that a leftmost boundary vertex exists if *grandparent* is a right child; otherwise, a rightmost boundary vertex should exist.

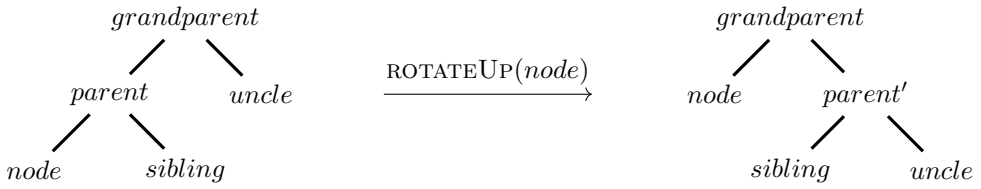


Figure 11: Rotation with *sibling* and *uncle* to the same side

Star structure

The analysis for the spanning tree configuration shown in Figure 9b is split into three separate cases.

Case 2: Consider first the case where the top tree looks like Figure 10a. The impact on the top tree is visualized in Figure 11.

If *sibling* were a path cluster, *node* and *uncle* would connect to different endpoints giving a path structure instead. Thus, let *sibling* be a point cluster.

The vertex shared by *node*, *sibling* and *uncle* is a middle boundary vertex of the new *parent′*, as *node* is located outside the cluster. Only the other endpoint of *uncle* can contribute an additional boundary vertex to *parent′*, which only happens if *uncle* is a path cluster. Thus *parent′* is a path cluster exactly when *uncle* is.

The order of *node*, *sibling* and *uncle* is maintained. However, *sibling* was a right child before the rotation, which meant it was connected to *node* through its leftmost boundary vertex. After

the rotation, it is a left child and should be connected through its rightmost boundary vertex. This is solved by flipping *sibling*.

To satisfy the orientation invariant, the new *parent*′ must have a leftmost boundary vertex as it is a right child connected to *node*, however, since it has a *middle* boundary vertex, both a leftmost and rightmost boundary vertex exist. The orientation invariant of the *grandparent* remains satisfied, as the central vertex of *grandparent* remains the same. The other possible boundary vertices come from *node* and *uncle*, which are left and right children, respectively, before and after the rotation.

Case 3: Consider a combination of Figure 9b and Figure 10b, the performed rotation can be seen in Figure 12. This case is split into two, and the first assumes *sibling* is a point cluster.

Assume *node* was a path cluster, which means *sibling* and *uncle* must connect to different endpoints, causing a path-like structure. Thus, both *node* and *sibling* are point clusters. The new *parent*′ is a path cluster only if *uncle* is, as they share boundary vertices.

node changes from a right to a left child. Thus, it must be flipped to maintain the orientation invariant. *sibling* and *uncle* are left and right children, respectively, before and after the rotation. Therefore, the orientation invariant remains satisfied for them.

The new parent must have a leftmost boundary vertex, which it does, as it is guaranteed to have a middle boundary vertex. Finally, the orientation invariant of *grandparent* is maintained, as the endpoints of *uncle* remains in the same position. The left endpoint is the middle boundary before and after the rotation, with the right endpoint being the right boundary vertex, if it exists.

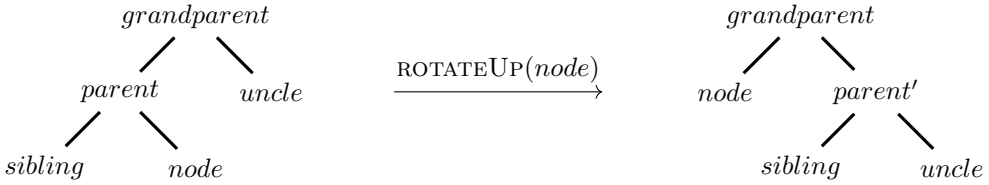


Figure 12: Rotation with *sibling* and *uncle* to different sides

Case 4: Consider the same combination of Figure 9b and Figure 10b, with *sibling* as a path cluster. The rotation is visualized in Figure 12.

node must be a point cluster to avoid *sibling* and *uncle* connecting to different endpoints. If *uncle* is a path cluster, the new *parent*′ has 3 boundary vertices. Thus, let *uncle* be a point cluster as well.

With *node* and *uncle* being point clusters, the new *parent*′ has the same boundary vertices as *sibling*, meaning *parent*′ is a path cluster when *sibling* is.

Similarly to case 3, we must flip *node* to maintain the orientation invariant. As the boundary vertices of *grandparent* only come from endpoints of *sibling*, these endpoints must remain in the same place. However, as seen in Figure 12, this isn't the case, but by flipping both *parent*′ and *grandparent*, the desired result is achieved. The effect of flipping both *parent*′ and *grandparent* can be seen in Figure 13

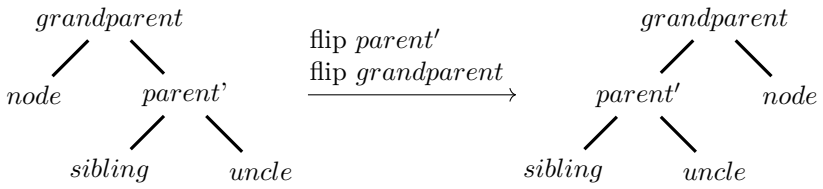
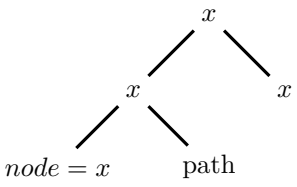


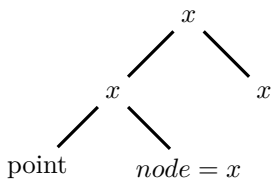
Figure 13: Flip *grandparent* and *parent'*

3.5.2 Supported rotations

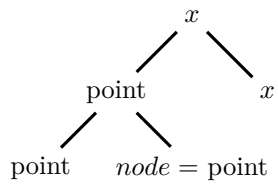
The rotations supported by ROTATEUP are summed up in Figure 14, where all the supported top tree configurations are visualised, except for mirrored ones, and which case supports them. Let x denote support for both point and path clusters in those positions.



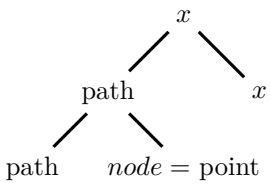
(a) Case 1



(b) Case 2



(c) Case 3



(d) Case 4

Figure 14: The top tree configurations supported by ROTATEUP, let x denote support for both point and path clusters

3.5.3 Analysis for when rotations are allowed

As mentioned at the start of Section 3.5, rotations are only valid if all clusters remain valid. As stated, the only new cluster is $sibling(node) \cup sibling(parent(node))$. The *grandparent* is still the union of *node*, *sibling* and *uncle* and thus remains valid. However, explicitly checking if the new cluster is valid is inconvenient. Therefore, Holm et al. showed a few top tree configurations that lead to valid clusters[9].

Before these configurations are considered, a few helping lemmas are proven.

Lemma 3.1. *(Lemma 4.1 from [9]) If A and B are valid clusters of a top tree whose intersection is a single vertex v, then the cluster $A \cup B$ is invalid if and only if the following three conditions hold: A is a path cluster, B is a path cluster, and v is a boundary vertex of $A \cup B$.*

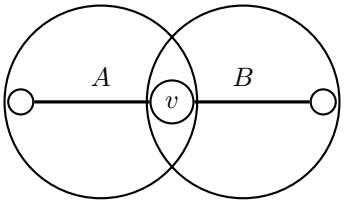


Figure 15: The spanning tree representation of Lemma 3.1

Proof. The boundary vertices of an internal node can be expressed as the union of boundary vertices of its children, with or without the central vertex, depending on the scenario.

First, consider why both A and B must be path clusters. Assuming A is a point cluster. Then it must have a single boundary vertex shared with B. Thus, the union of $A \cup B$ can contain at most 2 boundary vertices if the shared vertex v is included.

Therefore, it must be the case that both A and B are path clusters. v must be a boundary vertex of the internal node as well. Otherwise, at most, 1 boundary vertex is contributed from each child. □

Lemma 3.2. *(Lemma 4.2 from [9]) If A, B, and C are valid clusters with C the parent of A and B. If A is a point cluster, then all boundary vertices of C are also boundary vertices of B.*

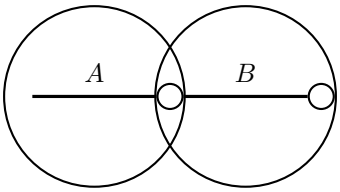
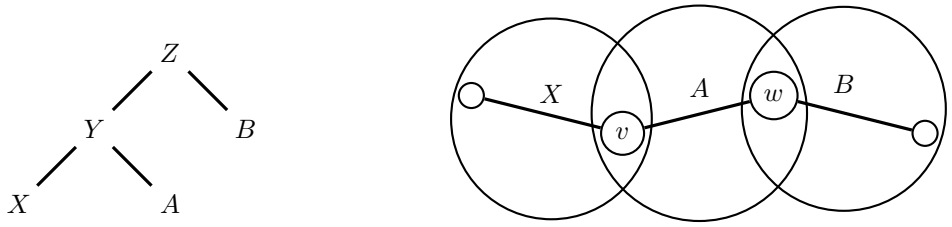


Figure 16: The spanning tree representation of Lemma 3.2

Proof. If A is a point cluster, it only contains a single boundary vertex that it shares with its sibling B. Thus, the boundary vertex of A is also a boundary vertex of B. Since the boundary vertices of C must also be boundary vertices of one of its children, all of them must be part of B. □

Lemma 3.3. *(Lemma 4.3 from [9]) Let X, Y, Z, A, B be valid clusters in a top tree with Y the parent of X and A, and Z the parent of Y and B. If A is a path cluster and X, Y are both left or right children, then $X \cup B$ is not a connected set of edges.*



(a) Top tree configuration of Lemma 3.3 (b) Spanning tree configuration of Lemma 3.3

Figure 17: One possible configuration for Lemma 3.3

Proof. WLOG assume the case shown in Figure 17. Because A is a path cluster with two boundary vertices, then by the orientation invariant, it must be the case that X and B are connected to different vertices. Let v be the vertex shared by X and A, and w the vertex shared by A and B.

Now assume $X \cup B$ was a connected set of edges, implying that a vertex u exists that was shared between X and B. However, this would create a cycle in the spanning tree, as we could move from A to B through w, to X by u and back to A with v. Thus, no such u can exist, and $X \cup B$ is not a connected set of edges. □

Lemma 3.4. *(Lemma 4.4 from [9]) Let X be a valid cluster in a top tree. If X and its grandparent Z are both point clusters, then it is valid to call rotateUp(X).*

(a) Configuration 1 (b) Configuration 2

Figure 18: Top tree representations of Lemma 3.4

Proof. When ignoring mirrored versions, the tree must be structured in one of the two ways pictured in Figure 18. The goal is to prove $A \cup B$ is a valid cluster. Recall a valid cluster is a connected set of edges with at most 2 boundary vertices.

Because X is a point cluster, then by Lemma 3.2, all boundary vertices of $Y = X \cup A$ are also boundary vertices of A. Y and B share a boundary vertex as they are siblings. Thus, A shares a vertex with B, meaning $A \cup B$ is a connected set of edges.

What remains is to show that $A \cup B$ has at most 2 boundary vertices. Point cluster X has a single boundary vertex that it shares with A. This boundary vertex would be shared with $A \cup B$ post rotation. Thus $A \cup B$ can have at most 1 boundary vertex more than $X \cup (A \cup B)$, which is the point cluster Z. Thus, $A \cup B$ can have at most 2 boundary vertices. □

Lemma 3.5. *(Lemma 4.5 from [9]) Let X, Y, Z, A, B be valid clusters in a top tree with Z the parent of Y and B and Y the parent of X and A. If Y is a path cluster, with both X and Y being left or right children, then it is valid to call rotateUp(X).*

Postcondition 1: If Z is a path cluster and not the root, then $A \cup B$ and Z are both left or right children if and only if they both were left or right children before the rotation.
Postcondition 2: If Z is a point cluster, then $A \cup B$ is a point cluster after the rotation.

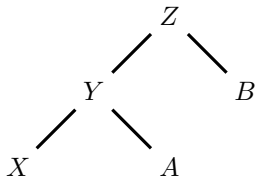


Figure 19: Top tree representation of Lemma 3.5

Proof. Without loss of generality, let A and B be right children, thus both X and Y being left children. An example of this configuration can be seen in Figure 19. First, we wish to prove that $A \cup B$ is connected. Let v be the boundary vertex between Y and B and the central vertex of Z. v is the rightmost boundary vertex of Y, which means that A and B are connected.

Because Y is a path cluster, it has another boundary vertex w, which due to the orientation invariant and the assumption about X and Y being left children, must be a boundary vertex of X.

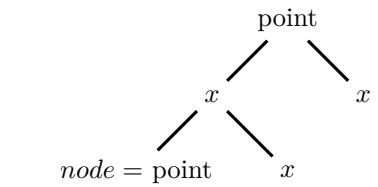
It remains to show that $A \cup B$ has at most 2 boundary vertices. Assume either A or B is a point cluster, then the union is a valid cluster by Lemma 3.1. Thus, assuming both A and B are path clusters, then Lemma 3.1 implies that the shared vertex v of A and B is not a boundary vertex. Since $Z = (A \cup B) \cup X$ is a valid cluster, the only way v can be a boundary vertex of $A \cup B$ is if X and B share this vertex. If v were shared with something outside the cluster, Z would have three boundary vertices as Y and B are path clusters. However, X and B are disconnected by Lemma 3.3. Thus, v is not a boundary vertex of $A \cup B$; thus, the cluster is valid.

Postcondition 1: To maintain the orientation invariant for Z when it is not the root, the sibling and parent of Z may need to be considered. Thus, assume Z is a path cluster with sibling C. By the assumption that Z is a path cluster, it has a boundary vertex which is not a non-middle boundary vertex. Assume, for simplicity, this was a left boundary vertex u, shared with C before the rotation. This boundary vertex should remain the left boundary vertex after the rotation to maintain the orientation invariant. If u was not shared with C, it must remain a left boundary vertex, so the rightmost boundary vertex of Z does not change. The non-middle vertex can never originate solely from a as its boundary vertices are connected to x and b before and after the rotation. Thus, any left or right boundary vertex must come exclusively from x or b, meaning x and $a \cup b$ must be children to the same side as $x \cup a$ and b were before.

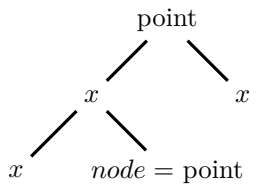
Postcondition 2: Assuming Z is a point cluster with X and Y both being left or right children, the boundary vertex of Z must be w, which is not the vertex shared by Y and B. As w is not the central vertex of Z, it must come exclusively from Y and be a boundary vertex of X. If $A \cup B$ is a path cluster after the rotation, then Z must also be a path cluster. □

By comparing the valid rotations seen in Figure 20 to the ones supported by ROTATEUP visualised in Figure 14, we can see which cases handle the valid rotations argued in Lemma 3.4 and Lemma 3.5. Let x denote support for both point and path clusters.

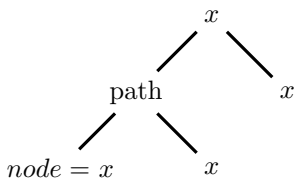
Lemma 3.4 and Lemma 3.5 will be used used in Section 3.6.1 and Section 3.8.1 to argue that only valid rotations are performed.



(a) Lemma 3.4 handled by case 1 and 2



(b) Lemma 3.4 handled by case 3 and 4



(c) Lemma 3.5 handled by case 1 and 2

Figure 20: The valid top tree configurations for rotations

3.6 Analysis of splay operations

This section analyses the amortized cost of SEMISPLAY and FULLSPLAY. SEMISPLAY and FULLSPLAY make a series of calls to SEMISPLAYSTEP, which searches for a valid top tree structure to call ROTATEUP on. Thus, the analysis initially focuses on SEMISPLAYSTEP, where it is shown that rotations can be made sufficiently often that only a constant number of nodes needs to be considered. Additionally, it is argued that all rotations performed are valid according to Lemma 3.4 and Lemma 3.5.

The amortized cost of SEMISPLAY and FULLSPLAY are equivalent when ignoring constants. The stronger depth guarantees from FULLSPLAY are sometimes required. Otherwise, SEMISPLAY is used whenever possible as Holm et al. conjecture the usage of SEMISPLAY to be a significant speed-up[9].

For the amortized analysis, the following potential function is used, $\phi = \sum_{\mathcal{T}} \sum_{x \in \mathcal{T}} r(x)$, with $r(x) = \log_2(s(x))$ and $s(x)$ as the number of leaves in the subtree rooted in x.

For the simplicity of the analysis, let each potential be able to pay for a constant number of COMBINE and SPLIT operations. Unfortunately, to handle this throughout the analysis, a factor of $O(COST(\text{COMBINE}) + COST(\text{SPLIT}))$ should be carried along in the amortized cost of each operation. To simplify these expressions significantly, assume for the analysis that COMBINE and SPLIT runs in $O(1)$.

3.6.1 semiSplayStep

This operation is designed to handle rotations when splaying a node, as calling ROTATEUP is only allowed sometimes. Thus, the entire goal of this operation is to detect when such calls are allowed, moving the node one level up in the tree.

Algorithm 7 semiSplayStep(b_0)

```
1: b1 = b0.parent
2: b2 = b1.parent
3: if b1 is null OR b2 is null then // No rotation possible, uncle can not exist
4:     return null
5: if isPoint(b0) AND isPoint(b2) then
6:     rotateUp(b0)
7:     return b2
8: b3 = b2.parent
9: if b3 is null then
10:    return null
11: if isPath(b1) AND (isPath(b2) OR isPoint(b3)) then
12:    pushFlip(b2)
13:    pushFlip(b1)
14:    nodeIsLeft = b1.children[0] == b0
15:    parentIsLeft = b2.children[0] == b1
16:    gparentIsLeft = b3.children[0] == b2
17:    if nodeIsLeft == parentIsLeft then
18:        rotateUp(b0)
19:        return b2
20:    if parentIsLeft == gparentIsLeft then
21:        rotateUp(b1)
22:        return b3
23: // At this point nodeIsLeft == gparentIsLeft is true
24:    rotateUp(sibling(b0)) // swaps sibling(b0) and sibling(b1)
25:    rotateUp(b1)
26:    return b3
27: return semiSplayStep(b1)
```

The semiSplayStep operation will follow the path $b_0, b_1, b_2, \ldots, b_k$ where b_k is the root, while it tries to match the current structure of the tree to a set of allowed patterns for rotations. Once it finds a match, it calls rotateUp and returns the root of the modified subtree.

The operation prefers to rotate nodes further down the tree, thus, patterns of b_0, b_1, b_2 are chosen over b_1, b_2, b_3. The patterns checked for are:

1. b_0, b_1, b_2 = point, path, point (line 5)
2. b_0, b_1, b_2 = point, point, point (line 5)
3. b_1, b_2, b_3 = path, path, point (line 11)
4. b_1, b_2, b_3 = path, point, point (line 11)
5. b_1, b_2, b_3 = path, path, path (line 11)

If none of these patterns matches semiSplayStep calls itself with $b_0 = b_1$ and tries again.

Lemma 3.6. *(Lemma 5.1 from [9]) The semiSplayStep operation only makes valid calls to rotateUp.*

Proof. For simplicity, we will consider the RotateUp calls made throughout the Algorithm 7 and argue why they are valid.

Consider the RotateUp call made on line 6. This only happens if b_0 and b_2 are point clusters and is a legal call by Lemma 3.4.

The RotateUp call made on line 18 of Algorithm 7 only happens if b_1 is a path cluster with b_0 and b_1 both being left or right children. Thus, calling RotateUp on b_0 is allowed by Lemma 3.5.

The RotateUp call on line 21 is slightly more complicated. If both b_1 and b_2 are left or right children, then it must be the case that b_2 is a path cluster, thus allowing RotateUp to be called on b_1 by Lemma 3.5. Assume for contradiction that b_2 is a point cluster, and both b_1 and b_2 are left children as visualised in Figure 21. The only boundary vertex of b_2 must be the rightmost boundary vertex it shares with $sibling(b_2)$. However, since b_1 is a path cluster, it contributes a left boundary vertex to b_2. Thus, reaching a contradiction as b_2 must have 2 boundary vertices and, thus, be a path cluster.

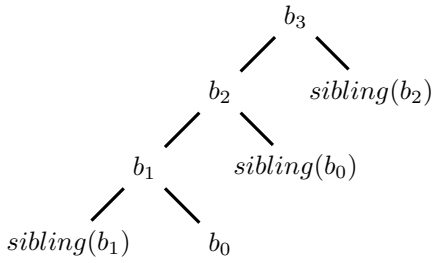


Figure 21: Configuration where both b_1 and b_2 are left children

Finally, consider the rotations on lines 24 and 25, where both b_0 and b_2 are left or right children. An example of this case is visualised in the left part of Figure 22. Since b_1 is a path cluster, with b_1 and $sibling(b_0)$ both being right children, calling RotateUp on $sibling(b_0)$ is valid per Lemma 3.5. If b_2 is a path cluster, the first post-condition of Lemma 3.5 states that b_1 and b_2 are both left or right children after the rotation, as seen in Figure 22. Since b_2 is a path cluster, with both b_1 and b_2 as left or right children, it is valid to call RotateUp with b_1 by Lemma 3.5.

Otherwise, if b_2 is a point cluster, then b_1' is a point cluster after the rotation by the second post-condition of Lemma 3.5. The only way to reach this is for b_3 to be a point cluster to satisfy the check on line 11. Thus, calling RotateUp with b_1 by Lemma 3.4 is valid.

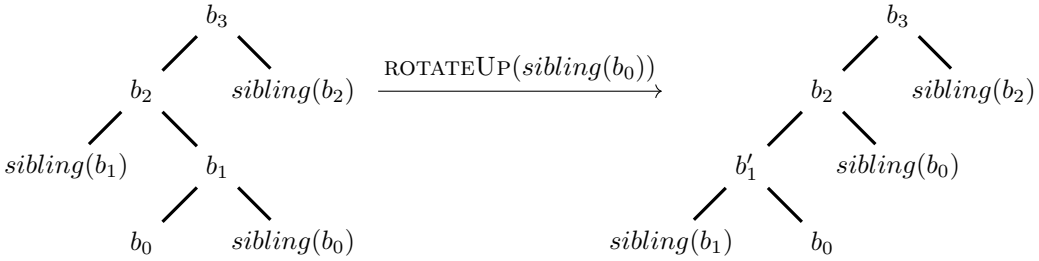


Figure 22: One configuration where both b_0 and b_2 are left or right children

□

Lemma 3.7. *(Lemma 5.2 from [9]) If $d = depth(b_0) \geq 5$, then the* semiSplayStep *always succeeds in finding a valid rotation and reduces the height of b_0 by 1, and returns a node b' with depth $d - 5 \geq depth(b') \geq d - 2$ such that all modified nodes are in the subtree rooted in b'.*

Proof. Consider the path $b_0, b_1, b_2, b_3, b_4, b_5$ on the way to the root. The goal is to argue that a pattern is matched on b_3, b_4, b_5 at the latest.

In Table 4, the patterns checked by the initial and first recursive call to semiSplayStep are shown, with the double lines separating the two iterations. Notice, all 5 patterns will be checked on b_1, b_2, b_3 and onward due to the recursive nature of semiSplayStep. This implies that both b_3 and b_4 must be path clusters to avoid matching any patterns, as everything ending in a point cluster is valid. Finally, we notice both path, path, path and path, path, point patterns are accepted. Thus, independently of what b_5 is b_3, b_4, b_5 is accepted, and thus, a match is found no later than this.

b_0	b_1	b_2	b_3	b_4	b_5
point	point	point			
point	path	point			
	path	path	point		
	path	point	point		
	path	path	path		
	point	point	point		
	point	path	point		
		path	path	point	
		path	point	point	
		path	path	path	

Table 4: The patterns matched through two iterations of semiSplayStep, with the double line separating the two iterations.

To argue that the depth of the returned node has the desired depth, consider the algorithm shown in Algorithm 7. Patterns matched on b_0, b_1, b_2 all return b_2. Whereas patterns for b_1, b_2, b_3 return either b_2 or b_3, thus we must return either b_2, b_3, b_4 or b_5. All of which have a depth satisfying the requirement.

Finally, to argue that the depth of the node is reduced by 1, we may consider the different rotations performed. The rotations performed directly on *node* swap it with its uncle, located one layer up the tree. The other cases call rotation on an ancestor of *node*, which moves the subtree in which the *node* is located one layer up, achieving the same result. □

Lemma 3.8. *(Lemma 5.3 from [9]) If* semiSplayStep*(b_0) fails to find a matching pattern then $depth(b_0) \leq 4$. Furthermore, if b_0 is a point cluster, then the $depth(b_0) \leq 3$. If the root is a point cluster, then the $depth(b_0) \leq 2$, and if both b_0 and the root are a point cluster, then $depth(b_0) \leq 1$.*

Proof. From Lemma 3.7, we are given that for any node b_0 with $depth(b_0) \geq 5$ semiSplayStep would succeed. Thus we have $depth(b_0) \leq 4$.

Consider the case where b_0 is a point cluster, and the root is a path cluster. We want to find a scenario where b_4 exists, and it fails to find a matching pattern. Because b_0 is a point cluster, b_2 must be a path cluster not to match one of the patterns on b_0, b_1, b_2. The same goes for b_3, which means both b_2 and b_3 are path clusters, and thus, if b_4 is a point or path cluster, it will match one of the checked patterns. Therefore, b_4 can not exist and $depth(b_0) \leq 3$ in this case.

Because we match all patterns ending in a point cluster, failure to find a match when the root is a point cluster must be because b_3 does not exist. Thus, $depth(b_0) \leq 2$. If both the b_0 and the root is a point cluster, then the two first patterns will match for any b_1, which means b_2 does not exist, meaning $depth(b_0) \leq 1$. □

Lemma 3.9. *(Lemma 5.4 from [9]) Consider the operation* semiSplayStep$(p_k(x))$ *with* $p_i(x)$ *the ith ancestor of* x *or the root if* $i \geq depth(x)$*. Let* x' *be the node* x *after the operation. Let* $0 \leq k < b$ *and* $0 \leq a \leq k+1$ *be such that* semiSplayStep$(p_k(x))$ *returns* $p_l(x')$ *for some* $l \leq b$*, which is the same node as* $p_{l+1}(x)$*. Then the semi-splay step changes the amortization potential by* $\Delta\phi = r(sibling(p_{l-1}(x'))) + \sum_{i=a}^{b-1} r(p_i(x')) - \sum_{i=a}^{b} r(p_i(x))$.

Proof. The semiSplayStep operation performs one or two rotations depending on the matched pattern. If the semiSplayStep rotate once as seen in Figure 23, where semiSplayStep found a matching pattern calling rotateUp(b_3). semiSplayStep returns b_5 the direct ancestor of b_3'. This means that $l = 1$, the potential of b_4 has been replaced with the b_4', which can be written as $\Delta\phi = r(sibling(p_{l-1}(b_3')) - r(p_l(b_3))$. This is the same expression as the one in the lemma with $l \leq b$ and $0 \leq a \leq b$.

Let $k = 0$ in $0 \leq k \leq b$ and $0 \leq a \leq k+1$ because the call to semiSplayStep went through 0 recursions. Choose $b = 1$ and $a = 0$, satisfying the inequalities to get the following.

$$\begin{aligned}
\Delta\phi &= r(sibling(p_{l-1}(b_3'))) + \sum_{i=a}^{b-1} r(p_i(b_3')) - \sum_{i=a}^{b} r(p_i(b_3)) \\
&= r(sibling(p_0(b_3'))) + \sum_{i=0}^{0} r(p_i(b_3')) - \sum_{i=0}^{1} r(p_i(b_3)) \\
&= r(sibling(p_0(b_3'))) + r(p_0(b_3')) - r(p_0(b_3)) - r(p_1(b_3)) \\
&= r(sibling(p_0(b_3'))) - r(p_1(b_3))
\end{aligned}$$

The terms $r(p_0(b_3')) - r(p_0(b_3))$ cancel out as the rotation did not change anything about the subtree rooted in b_3. Thus, $a = 1$ would obtain the same result. Simiarly, $r(p_c(b_3'))$ cancels out with $r(p_{c+1}(b_3))$ for any $c \geq l$, which can be observed in Figure 23 too, where the leaves of b_5 and b_6 are the same before and after the rotation. The only difference is that b_5 is the parent of b_3' but the grandparent of b_3. Thus, choosing a greater b will not change anything.

If semiSplayStep did not immediately find a matching pattern, or the rotateUp call was made on an ancestor to b_3, the expression would still be the same, only swapping out b_3 for the ancestor rotateUp was called on.

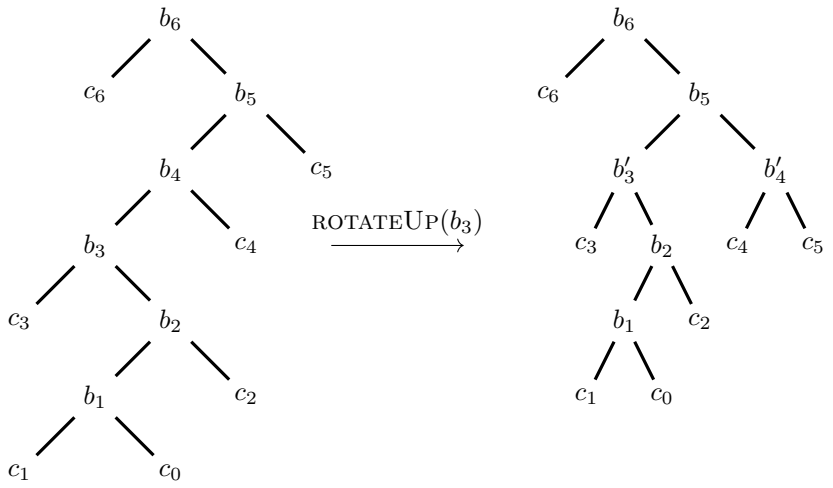


Figure 23: semiSplayStep(b_3) matches on the pattern point,point,point as b_3, b_4 and b_5 are point clusters. The top tree example is taken from [9], and the corresponding spanning tree can be seen in Figure 42 in Appendix D

If instead, two rotates are performed as seen in Figure 24 where semiSplayStep(b_1) was called. Then the outputted node is b_4, the 2nd ancestor of b'_1 in the final tree. Thus, let $l = 2$ and observe that the potential change is $\Delta\phi = r(sibling(p_1(b'_1))) + r(p_1(b'_1)) - r(p_2(b_1)) - r(p_1(b_1))$ by observing the tree. This is equivalent to the expression in the lemma with $l \leq b$. The semiSplayStep finds a match in the first iteration. Thus, $k = 0$. WLOG let $a = 1$ and $b = 2$.

$$\begin{aligned}\Delta\phi &= r(sibling(p_{l-1}(b'_1))) + \sum_{i=1}^{1} r(p_i(b'_1)) - \sum_{i=1}^{2} r(p_i(b_1)) \\ &= r(sibling(p_{l-1}(b'_1))) + r(p_1(b'_1)) - r(p_2(b_1)) - r(p_1(b_1)) \qquad \text{writing out the sums}\end{aligned}$$

If a larger b is chosen, the terms cancel out. The same goes for choosing $a = 0$.

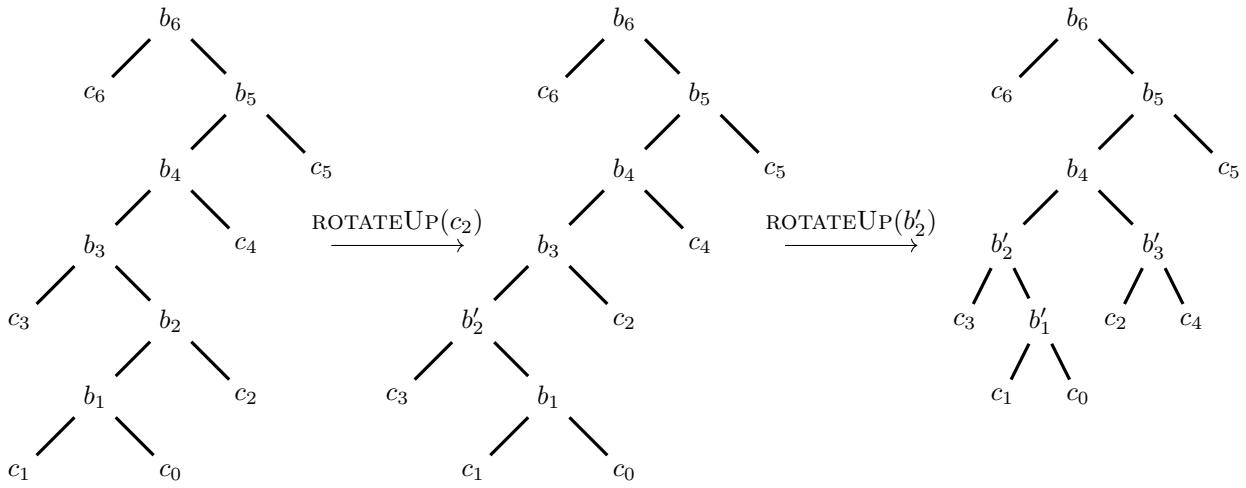


Figure 24: semiSplayStep(b_1) matches the pattern on path,point,point because b_2 is path, and both b_3 and b_4 are point clusters. The top tree example is taken from [9], and the corresponding spanning tree can be seen in Figure 42 in Appendix D

□

3.6.2 semiSplay

In this section, the potential change and amortized cost of the semiSplay operation is considered, along with the reduction in depth it provides for the splayed node. It reduces the depth of a node x by at least $\frac{1}{5}depth(x)$ by repeatably calling semiSplayStep.

Algorithm 8 semiSplay(x)

```
1: top = x
2: while top is not null do
3:     top = semiSplayStep(top)
```

A supporting result is required before considering the amortized cost of semiSplay.

Lemma 3.10. *(Lemma 5.5 from [9]) Let $a, b, c > 0$, if $a + b \leq c$ then $\log_2(a) + \log_2(b) \leq 2\log_2(c) - 2$.*

Proof. By the arithmetic-geometric mean inequality we have $\sqrt{ab} \leq \frac{a+b}{2} \leq \frac{c}{2}$, this gives us $ab \leq (\frac{c}{2})^2$ and $\log_2(a) + \log_2(b) = \log_2(ab) \leq 2\log_2(\frac{c}{2}) = 2(\log_2(c) - 1) = 2\log_2(c) - 2$. □

Lemma 3.11. *(Lemma 5.6 from [9]) Calling semiSplay on a node x does $O(depth(x))$ work, changes the potential by $\Delta\phi \leq O(1 + r(root(x)) - r(x)) - \Omega(depth(x))$ and reduces the depth of x to at most $\lceil \frac{4}{5}depth(x) \rceil$.*

Proof. Consider a single successful iteration of semiSplay which calls semiSplayStep with node x. The semiSplayStep returns $p_l(x')$, with x' being x after the rotation. semiSplayStep

always returns a node higher in the tree. Thus we get $1 \leq l$, and due to Lemma 3.7, the returned node of semiSplayStep can be at most 4 levels higher in the tree compared to x'. Thus, we know that $1 \leq l \leq 4$. Choose $a = 0$ and $b = l$ to satisfy the requirements of Lemma 3.9, which gives the following potential change for a single iteration.

$$\Delta\phi = r(sibling(p_{l-1}(x'))) + \sum_{i=0}^{l-1} r(p_i(x')) - \sum_{i=0}^{l} r(p_i(x))$$

The expression is simplified before considering the potential change of the entire semiSplay operation. First, recall that $r(x') = \log(s(x'))$, with $s(x')$ being the number of leaves in the subtree rooted in x'. Lemma 3.10 states that $r(sibling(p_{l-1}(x'))) + r(p_{l-1}(x')) \leq 2r(p_l(x')) - 2$, because $sibling(p_{l-1}(x'))$ and $p_{l-1}(x')$ have disjoint subtrees, which means $s(sibling(p_{l-1}(x'))) + s(p_{l-1}(x')) \leq s(p_l(x'))$.

$$\begin{aligned}\Delta\phi &= r(sibling(p_{l-1}(x'))) + \sum_{i=0}^{l-1} r(p_i(x')) - \sum_{i=0}^{l} r(p_i(x)) \\ &\leq 2r(p_l(x')) - 2 + \sum_{i=0}^{l-2} r(p_i(x')) - \sum_{i=0}^{l} r(p_i(x))\end{aligned}$$

Then by analysing the possible values for l, we can rewrite the sums. If $l = 1$, the first sum was already consumed for Lemma 3.10. The second sum has two terms, which can be lower bounded by $r(x)$ as all parents of x must contain more leaves.

$$\Delta\phi \leq 2r(p_l(x')) - 2r(x) - 2 \leq 4r(p_l(x')) - 4r(x) - 2$$

If $l \geq 2$, the terms $p_0(x') = p_0(x)$ cancel out as the children of x are unchanged. This leaves us with 1 or 2 terms in the first sum and 3 or 4 in the second, respectively. The first sum is upper bounded by $r(p_l(x'))$, and the second sum is lower bounded by $r(x)$.

$$\begin{aligned}\Delta\phi &\leq 2r(p_l(x')) - 2 + \sum_{i=0}^{l-2} r(p_i(x')) - \sum_{i=0}^{l} r(p_i(x)) \\ &\leq 4r(p_l(x')) - 4r(x) - 2\end{aligned}$$

The analysis can likely be improved to $2r(p_l(x')) - 2r(x) - 2$ by carefully considering which subtrees are modified.

The next iteration of the while loop in Algorithm 8 uses the returned node $p_l(x')$ for the next call to semiSplayStep. Thus, the $4r(p_l(x'))$ gets cancelled if the next call succeeds.

Now consider the full potential change by semiSplay. First, by considering that any call to semiSplayStep is successful if the depth is at least 5 per Lemma 3.7, there must be at least $\lfloor \frac{1}{5} depth(x) \rfloor \geq \frac{1}{5}(depth(x) - 4)$ successful iterations. By summing over all iterations, the change

in potential is, at most:

$$\begin{aligned}\Delta\phi &\leq 4(r(root(x)) - r(x)) - \frac{2}{5}(depth(x) - 4) \\ &= 4(r(root(x)) - r(x)) - \frac{2}{5}depth(x) + \frac{8}{5} \\ &= 4(\frac{8}{20} + r(root(x)) - r(x)) - \frac{2}{5}depth(x) \\ &= O(1 + r(root(x)) - r(x)) - \Omega(depth(x)) \\ &= O(\log n) - \Omega(depth(x))\end{aligned}$$

The SEMISPLAY operation itself has $O(depth(x))$ iterations. Where the cost of each iteration is constant except for the calls to COMBINE during the rotations, for each SEMISPLAYSTEP, a constant number of calls to combine are made, and thus, we make at most $O(depth(x))$ COMBINE calls. Recall each freed potential can pay for some constant amount of COMBINE calls. Thus, achieving an amortized cost of $O(\log n) - \Omega(depth(x))$.

Any call to SEMISPLAYSTEP reduces the depth of x by 1. Thus, the new depth of x is at most $depth(x) - \lfloor \frac{1}{5}depth(x) \rfloor = \lceil \frac{4}{5}depth(x) \rceil$. □

3.6.3 fullSplay

In this section, the potential change and amortized cost of the FULLSPLAY operation is considered, along with the reduction in depth it provides for the splayed node. When FULLSPLAY(x) terminates $depth(x)$ satisfies the same bounds as Lemma 3.8.

If the only requirement was to reduce the depth of x within a constant of the root, repeatably calling SEMISPLAYSTEP(x) suffices. This strategy can not achieve the desired runtime.

Algorithm 9 FULLSPLAY(x)

```
1: while true do
2:     top = SEMISPLAYSTEP(x)
3:     if top is null then
4:         return
5:     SEMISPLAYSTEP(top)
```

Lemma 3.12. *(Lemma 5.7 from [9]) Calling FULLSPLAY on a node x does $O(depth(x))$ work, changes the potential by $\Delta\phi \leq O(\log n) - \Omega(depth(x))$, and reduces the depth of x so it satisfies the same bounds as in Lemma 3.8*

Proof. Consider an iteration of FULLSPLAY where both SEMISPLAYSTEPS succeeds and let x denote the node before the first SEMISPLAYSTEP, x' and x'' the same node after the first and second call, respectively. Let $p_{l_1-1}(x')$ and $p_{l_2-1}(x'')$ denote the nodes returned from the SEMISPLAYSTEP calls. From the proof of Lemma 3.11, we have $1 \leq l_1 \leq 4$, and through similar reasoning, we have $l_1 \leq l_2 \leq 8$. Let $\Delta\phi_1$ and $\Delta\phi_2$ represent the potential change induced by the

first and second semiSplayStep call, respectively.

$$\Delta\phi_1 = r(sibling(p_{l_1-1}(x'))) + \sum_{i=a}^{b-1} r(p_i(x')) - \sum_{i=a}^{b} r(p_i(x))$$
$$\Delta\phi_2 = r(sibling(p_{l_2-1}(x''))) + \sum_{i=c}^{d-1} r(p_i(x'')) - \sum_{i=c}^{d} r(p_i(x'))$$

Thus, the potential change for a single iteration where both semiSplayStep calls succeed is

$$\begin{aligned}\Delta\phi_{double} &= \Delta\phi_1 + \Delta\phi_2 \\ &= r(sibling(p_{l_1-1}(x'))) + \sum_{i=a}^{b-1} r(p_i(x')) - \sum_{i=a}^{b} r(p_i(x)) + \\ &\quad r(sibling(p_{l_2-1}(x''))) + \sum_{i=c}^{d-1} r(p_i(x'')) - \sum_{i=c}^{d} r(p_i(x'))\end{aligned}$$

Let $d = b - 1$ and $a = c$.

$$\Delta\phi_{double} = r(sibling(p_{l_1-1}(x'))) + r(sibling(p_{l_2-1}(x''))) + \sum_{i=a}^{b-2} r(p_i(x'')) - \sum_{i=a}^{b} r(p_i(x))$$

By Lemma 3.9 we have $l_2 \leq d \Rightarrow d \geq 8$, therefore let $d = 8$ and $a = 1$ with $b = 9$ satisfying $d = b - 1$.

$$\Delta\phi_{double} = r(sibling(p_{l_1-1}(x'))) + r(sibling(p_{l_2-1}(x''))) + \sum_{i=1}^{7} r(p_i(x'')) - \sum_{i=1}^{9} r(p_i(x))$$

$sibling(p_{l_1-1}(x'))$ and $sibling(p_{l_2-1}(x''))$ have disjoint subtrees because $p_{l_2-1}(x'')$ is an ancestor of $p_{l_1-1}(x')$ and $sibling(p_{l_1-1}(x'))$. Since both $p_{l_2-1}(x'')$ and $p_{l_1-1}(x')$ are decendents of $p_8(x'')$, we have $s(sibling(p_{l_2-1}(x''))) + s(sibling(p_{l_1-1}(x'))) \leq s(p_8(x''))$. Thus, by Lemma 3.10 $r(sibling(p_{l_2-1}(x''))) + r(sibling(p_{l_1-1}(x'))) \leq 2r(p_8(x'')) - 2$.

$$\begin{aligned}\Delta\phi_{double} &\leq 2r(p_8(x'')) - 2 + \sum_{i=1}^{7} r(p_i(x'')) - \sum_{i=1}^{9} r(p_i(x)) \\ &\leq -2 + \sum_{i=1}^{9} r(p_i(x'')) - \sum_{i=1}^{9} r(p_i(x))\end{aligned}$$

The final inequality follows from $2r(p_8(x'')) \leq r(p_8(x'')) + r(p_9(x''))$.

Now consider the case where only a single semiSplayStep succeeds, then the following change

in potential is observed. Let $a = 1$ and $b = 9$ satisfying Lemma 3.9.

$$\begin{aligned}\Delta\phi_{single} &= \Delta\phi_1 \\ &= r(sibling(p_{l_1-1}(x'))) + \sum_{i=a}^{b-1} r(p_i(x')) - \sum_{i=a}^{b} r(p_i(x)) \\ &= r(sibling(p_{l_1-1}(x'))) + \sum_{i=1}^{8} r(p_i(x')) - \sum_{i=1}^{9} r(p_i(x)) \\ &= \sum_{i=1}^{9} r(p_i(x')) - \sum_{i=1}^{9} r(p_i(x))\end{aligned}$$

The final equality follows from $(sibling(p_{l_1-1}(x'))) \leq r(p_9(x'))$.

When both semiSplaySteps succeed, the depth of x is reduced by 2. Thus, there must be at least $\frac{1}{2}(depth(x) - 9)$ iterations where this occurs.

Now consider how many iterations where both semiSplaySteps succeed, there must at least $\frac{1}{2}(depth(x) - 9)$. This gives the following potential change when considering these iterations.

$$\Delta\phi \leq \sum_{j=0}^{\frac{1}{2}(depth(x)-9)} \Big(-2 + \sum_{i=1}^{9} r(p_i(x'')) - \sum_{i=1}^{9} r(p_i(x)) \Big)$$

For the next iteration of the while loop, $x = x''$. Thus, all but the first and last terms cancel out. Finally, a few iterations may occur with only 1 succeeding semiSplayStep, which cancels out similarly.

When semiSplayStep terminates, let x be the node before any rotations and x^* be the same node after all rotations.

$$\begin{aligned}\Delta\phi &\leq \sum_{i=1}^{9} r(p_i(x^*)) - \sum_{i=1}^{9} r(p_i(x)) - depth(x) + 9 \\ &\leq 9(1 + r(root(x)) - r(x)) - depth(x) \\ &= O(\log n) - \Omega(depth(x))\end{aligned}$$

The second inequality follows by upper bounding and lower bounding the sums.

The fullSplay operation has $O(depth(x))$ iterations, where each iteration requires constant work, except for the $O(1)$ calls to combine. Thus, similarly to semiSplay, $O(depth(x))$ calls to combine is made, which is paid off by the freed potential. This gives an amortized cost of $O(\log n) - \Omega(depth(x))$.

The depth of x is constant and satisfies Lemma 3.8, as fullSplay terminates when the semiSplayStep on line 2 returns null, which only happens when $depth(x) \leq 5$ as argued in Lemma 3.7 and Lemma 3.8. □

3.7 Finding the consuming node of vertex

This section considers the correctness and amortized cost of findConsumingNode.

Let the consuming node of a vertex v be the lowest cluster in the top tree that contains all edges adjacent to v. If v is not exposed, then the consuming node is the smallest cluster that contains

v without having it as a boundary vertex. When combining two clusters, their boundary vertices can only become non-boundary if they are the central vertex and, thus, shared between both clusters. Therefore, the operation should return the first cluster C with v as the central vertex without a middle boundary vertex.

C is also the largest cluster with the v as a central vertex. Assume for contradiction that an ancestor D of C had v as a central vertex. Then it must be the case that v is shared between the decedents of D, which means that v was still a boundary vertex in C. If v is exposed, it will remain a boundary vertex; thus, the largest cluster where v is the central vertex is returned.

Recall that this operation had been assigned the task of handling SPLIT calls for EXPOSE, DEEXPOSE and LINK.

Algorithm 10 FINDCONSUMINGNODE(v)

```
1: if v has no neighbors then
2:     return null
3: node = any edge incident to v
4: x = root(node)
5: while x is not node do
6:     SPLIT(x)
7:     x = next node on path between x and node
8: SEMISPLAY(node)
9: if v has at most one incident edge then
10:    return node
11: // Where is v a boundary vertex in node?
12: isLeft = (is v left endpoint of node) != node.flip
13: isMiddle = false
14: isRight = (is v right endpoint of node) != node.flip
15: lastMiddleNode = null
16: while node is not root do
17:    parent = node.parent
18:    isLeftChild = parent.children[0] == node
19: // Compute where v is in the parent, takings the parents flip into account
20:    if isLeftChild then
21:        isMiddle = isRight || (isMiddle AND !HASRIGHTBOUNDARY(node))
22:    else
23:        isMiddle = isLeft || (isMiddle AND !HASLEFTBOUNDRY(node))
24:    isLeft = (isLeftChild != parent.flip) AND !isMiddle
25:    isRight = (isLeftChild == parent.flip) AND !isMiddle
26:    node = parent
27:    if isMiddle then
28:        if !HASMIDDLEBOUNDARY(node) then
29: // Only happens if the vertex is not exposed
30:            return node
31:        lastMiddleNode = node
    // This only happens when the vertex is exposed
32: return lastMiddleNode
```

First, consider which SPLIT calls are required. The operations EXPOSE, DEEXPOSE and LINK

only makes changes to the consuming node and above. However, SEMISPLAY(*node*) may cause recomputations of any cluster between *node* and the root. Thus, SPLIT has to be called on the entire path between *node* and the root, which includes the consuming node and its ancestors.

Now let us consider the correctness of the returned node. If v has no neighbours, it is an unconnected vertex with no edges represented in $\mathcal{T}$. Therefore, no node can be returned. A special case is when v only has a single edge adjacent to it, as the leaf node representing this edge contains all edges adjacent to v, but it has no central vertex and, thus, needs to be handled independently. Algorithm 10 handles this by checking the degree of v if it is 1, the leaf node is returned.

The operation checks when v becomes the central vertex by tracking whether it is a left, middle or right boundary vertex at each level in $\mathcal{T}$. This is done with respect to the parent's orientation; thus, left and right may be flipped if $parent.flip$ is true. Initially, v can not be the middle boundary vertex, as a leaf node only consists of a single edge.

The while loops recompute whether v is a left, middle or right boundary vertex in the *parent* for each iteration. v can only become the central vertex and middle boundary of *parent*, if *node* is the left child and v is the rightmost boundary vertex of *node*. Similarly, if *node* is the right child, v must be the leftmost boundary vertex.

v is a left boundary vertex if and only if it comes from the left child and is not the central vertex. The same goes for v being a right boundary vertex.

Lastly, the returned value is correct when v is not exposed, as line 23 of Algorithm 10 checks that v is the central vertex and not a boundary vertex. If this never happens, it must be the case that v is exposed, and we return the last and largest cluster in which v was a central node.

Lemma 3.13. *The amortized cost of* FINDCONSUMINGNODE *is* $O(\log n) - \Omega(depth(node^*))$*, where* $node^*$ *is the returned node.*

Proof. The actual work done by SEMISPLAY is $O(depth(node))$, and it changes the potential by $O(\log n) - \Omega(depth(node))$ per Lemma 3.11. The second while loop runs for at most $depth(node')$, where $node'$ is the *node* after the SEMISPLAY call. Each iteration requires a constant amount of work. Thus, let the freed potential pay for the $O(depth(node))$ SPLIT calls along with the work performed in SEMISPLAY and the second while loop.

Let $node^*$ be the output of FINDCONSUMINGNODE, with $depth(node^*) \leq depth(node)$. This means that the potential freed can continue paying for additional work required for nodes on the path between $node^*$ and the root by choosing a sufficiently large constant for the potential.

Thus, the actual cost of FINDCONSUMINGNODE is $O(depth(node))$. But with the sufficiently large constant for the potential, the amortized cost is $O(\log n) - \Omega(depth(node^*))$. □

The amortized cost must be $O(\log n) - \Omega(depth(node^*))$, as it allows for $depth(node^*)$ work to be performed later for free. This is used in both EXPOSE and DEEXPOSE to obtain the desired runtimes.

3.8 External operations

In this section, the external top tree operations are considered. Initially, EXPOSE and DEEXPOSE are analysed, as they are used in the implementations of LINK and CUT.

3.8.1 expose

This section shows the correctness of the simple EXPOSE operation proposed by Holm et al. is shown. The amortized cost of EXPOSE is also considered[9].

The EXPOSE operation takes a vertex v and makes it a boundary vertex for all clusters it is part of. As v may be part of clusters that already have 2 boundary vertices, a few steps must be taken to avoid creating invalid clusters.

First, we are only interested in increasing the number of boundary vertices in clusters containing v, which does not already have v as a boundary vertex. The consuming node is the first cluster which contains v without having it as a boundary vertex. The final step is to use rotations to ensure no node between the consuming node and the root contains more than 2 boundary vertices. Once this has been prepared, we can finally expose v and recompute the information stored in the affected clusters.

A requirement of EXPOSE is for $\mathcal{T}$ to have < 2 exposed vertices currently.

Holm et al. also propose an advanced version of EXPOSE, which avoids using FULLSPLAY. This version is conjectured to be a significant speed-up in practice. Theoretically, they provide the same guarantees; thus, the simpler version is considered here[9]. An experimental comparison between the two EXPOSE operations is included in Section 5.

Algorithm 11 EXPOSE(v)

```
1:  node = findConsumingNode(v)
2:  if node is null then // v has degree zero
3:      v.expose = true
4:      return null
5:  while isPath(node) do
6:      parent = node.parent
7:      pushFlip(node)
8:      nodeIDX = index of node in parent.children
9:      rotateUp(node.children[nodeIDX])
10:     node = parent
    fullSplay(node)
11: // Now depth(node) ≤ 1, and node is the consuming point cluster
12: root = null
13: while node is not null do
14:     root = node
15:     root.numBoundaryVertices += 1
16:     combine(root)
17:     node = root.parent
18: vertex.exposed = true
19: return root
```

Let the starting point for the EXPOSE operation be the consuming node *node* by calling findConsumingNode. The while loop on line $5 - 10$ ensures that *node* is a point cluster by performing a series of rotations. The rotations are called on a child of *node* to ensure the rotation is legal with respect to Lemma 3.5. Once a rotation has been performed, the parent of *node* is the new consuming node of v, as it is the lowest common ancestor of *node*s children as visualised in

Figure 25.

The consuming node *node* has to become a point cluster for two reasons. First, it is a requirement before increasing the number of boundary vertices. Secondly, FULLSPLAY provides a better guarantee when called on a point cluster. In particular, when both the *node* and root are point clusters, FULLSPLAY ensures $depth(node) \leq 1$, which allows us to expose v and increase the number of boundaries in both nodes.

Finally, we must argue that the orientation invariant is not broken when exposing v. The rotations are designed to preserve the orientation invariant as argued in Section 3.5. Thus, only the orientation invariant of the root can be broken by exposing v. After the FULLSPLAY call, *node* is a point cluster with v as its central node, which means it connects to $sibling(node)$ by a left or right boundary vertex. Assume *node* is a left child with a rightmost boundary vertex. By making v a middle boundary vertex, the rightmost boundary vertex is unchanged; thus, the root's orientation invariant remains satisfied.

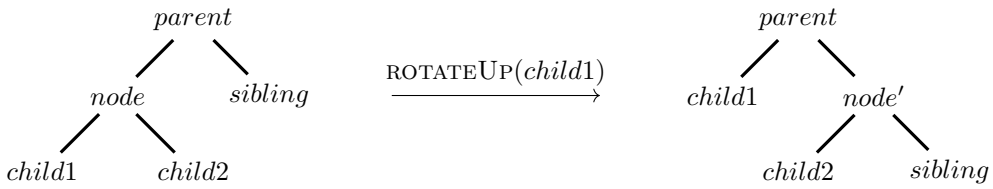


Figure 25: One possible rotation called in the while loop

Lemma 3.14. *The amortized cost of* EXPOSE *is* $O(\log n)$.

Proof. The call to FINDCONSUMINGNODE has an amortized cost of $O(\log n) - \Omega(depth(node))$, where *node* is the returned node, per Lemma 3.13.

The while loops run for at most $depth(node)$ iterations, as the consuming node of v moves 1 layer up $\mathcal{T}$ for each rotation. Each iteration of the while loop calls ROTATEUP, making $O(1)$ work, except for the COMBINE calls, which can be paid for by the potential freed during FINDCONSUMINGNODE.

Each while loop iteration also changes the potential of $\mathcal{T}$ through the rotations. The change in each iteration can be observed in Figure 25 and is the following.

$$r(node') - r(node) \leq r(parent) - r(node)$$

The first expression is upper bounded by the *parent* to make the telescoping sum work. In the next iteration of the while loop, we have $node = parent$. Thus, the terms cancel out. This gives a total potential change of $r(node^*) - r(node) \leq \log n$ where *node* and $node^*$ are the consuming node before and after the while loop.

Thus, up until the FULLSPLAY, the amortized cost is $O(\log n)$. As stated in Lemma 3.12, the actual runtime of FULLSPLAY($node$) is $O(depth(node))$, with a potential change of $O(\log n) - \Omega(depth(node))$. The final while loop runs for at most 2 iterations, as $depth(node) \leq 1$ after the full splay, this adds a constant number of COMBINE calls which can be paid off by either SEMISPLAY or FULLSPLAY.

Thus, the amortized cost for the entire EXPOSE operation is $O(\log n)$. □

3.8.2 deExpose

In this section, the correctness of the deExpose operation is shown along with its amortized cost.

This operation takes an exposed vertex v and ensures v only counts as a boundary vertex in clusters where v is incident to something outside the cluster.

As discussed in Section 3.8.1, v is only a boundary in the consuming node and above if it is exposed. Thus, updating the number of boundary vertices in these clusters and recomputing the cluster information is everything Algorithm 12 has to do. The orientation invariant of the changed clusters remains satisfied, as the vertex shared with a sibling can never be v as that would imply the consuming node is found further up the tree.

A requirement of deExpose is that the provided vertex is currently exposed.

Algorithm 12 deExpose(v)

```
1: root = null
2: node = findConsumingNode(v)
3: while node is not null do
4:     root = node
5:     root.numBoundaryVertices -= 1
6:     combine(root)
7:     node = root.parent
8: v.exposed = false
9: return root
```

Lemma 3.15. *deExpose takes $O(\log n)$ amortized time.*

Proof. The call to findConsumingNode has an amortized cost of $O(\log n) - \Omega(\mathit{depth}(\mathit{node}))$ as described in Lemma 3.13, with *node* being the node returned by findConsumingNode. The rest of the operation runs in $\mathit{depth}(\mathit{node})$ time. Thus, the amortized cost for deExpose is $O(\log n)$. □

3.8.3 link

This section shows the correctness of the link operation along with its amortized cost.

The link operation takes two vertices u and v, newly connected in the spanning forest, and combines $\mathcal{T}_u$ and $\mathcal{T}_v$ into a single top tree.

If u was an unconnected vertex in the spanning forest before adding (u, v), it is added to the top tree of $\mathcal{T}_v$, similarly for v. If both u and v are unconnected vertices, we create a new top tree with only one node representing the new edge.

It is a requirement that no vertices are currently exposed in T_u and T_v.

The expose ensures that a vertex is considered a boundary vertex of all clusters it is part of and returns the tree's root. At first, consider expose(u), all clusters containing u now consider it a boundary vertex. This is desired as the edge (u, v) is outside these clusters. Thus, when a leaf node representing (u, v) is added alongside an internal node at the top of $\mathcal{T}_u$, u should be a boundary vertex of every cluster it is part of, as the consuming node of u is the new root. This

also allows us to efficiently de-expose u by setting $u.exposed = false$ and manually updating the number of boundary vertices in the root. The same idea is used for v, which is merged with the new root.

To ensure that the orientation invariant is maintained, we make sure to flip $\mathcal{T}_u$ such that u is a right boundary vertex of the root in $\mathcal{T}_u$ as it will be the left child of the new root, and u will be the shared vertex with the right child. Similarly, v is ensured to be a left boundary vertex for the same reason.

Algorithm 13 LINK(u, v)

```
1: tu = EXPOSE(u)
2: if if tu is not null AND HASLEFTBOUNDARY(tu) then
3:     tu.flip = !tu.flip
4: u.exposed = false
5: tv = EXPOSE(v)
6: if tv is not null AND HASTRIGHTBOUNDARY(tv) then
7:     tv.flip = !tv.flip
8: v.exposed = false
9: T = new leaf node corresponding to the edge (u,v)
10: T.numBoundaryVertices = (tu is not null) + (tv is not null)
11: combine(T)
12: if tu is not null then
13:     T = new node with children tu and T
14:     T.numBoundaryVertices = (tv is not null)
15:     combine(T)
16: if tv is not null then
17:     T = new node with children T and tv
18:     T.numBoundaryVertices = 0
19:     combine(T)
20: return T
```

Lemma 3.16. *The amortized runtime of* LINK *is* $O(\log n)$ *while increasing the by* $O(\log n)$.

Proof. We make two calls to EXPOSE, which have an amortized runtime of $O(\log n)$ as stated in Lemma 3.14; the rest of the operation runs in constant time, except for the $O(1)$ COMBINE calls. The COMBINE calls can be paid for by the potential change induced by SEMISPLAY or FULLSPLAY in EXPOSE.

The potential change of LINK is the addition of up to three new nodes. The new leaf node changes the potential by $r(1) = \log 1 = 0$, while the two internal nodes are upper bounded by $r(root(\mathcal{T}_{new})) = \log n$. Thus, the potential change is $\Delta\phi \leq \log 1 + 2(\log n) \leq O(\log n)$.

Thus, the amortized cost of LINK is $O(\log n)$. □

3.8.4 cut

In this section, the correctness of the CUT operation is shown along with its amortized cost.

The cut operation takes an edge $e = (u, v)$, which was recently removed or is being removed from the spanning forest, and updates the top tree data structure such that the top tree $\mathcal{T}$ containing

u and v is split into $\mathcal{T}_u$ and $\mathcal{T}_v$. If u becomes an unconnected vertex once (u, v) is removed, $\mathcal{T}_u$ is an empty tree. Similarly, for v and $\mathcal{T}_v$.

We require that no vertices are exposed in $\mathcal{T}$ when calling the CUT operation.

A supporting operation DELETEALLANCESTORS is defined in Algorithm 14. It takes a node and deletes the ancestors starting with the root.

Algorithm 14 DELETEALLANCESTORS(*node*)

```
1: p = node.parent
2: if p is not null then
3:     s = sibling(node)
4:     deleteAllAncestors(p)
5:     s.parent = null
6: delete node
```

Algorithm 15 CUT(e)

```
1: node = leaf node representing e
2: x = root(node)
3: while x is not node do
4:     split(x)
5:     x = next node on path between x and node
6: u = e.vertices[0]
7: v = e.vertices[1]
8: fullSplay(node)
9: // now depth(e) ≤ 2, if u and v are boundary vertices of the leaf node, otherwise depth(e) ≤ 1
10: deleteAllAncestors(e)
11: u.exposed = true
12: v.exposed = true
13: tu = deExpose(u)
14: tv = deExpose(v)
```

As part of the CUT operation, FULLSPLAY is called on *node*, which is the leaf node representing (u, v). As FULLSPLAY may cause cluster information to be recomputed, any information currently stored further up in the tree should be propagated downwards. Thus, let CUT start with a sequence of SPLIT calls to handle this exact scenario.

With no exposed vertices in $\mathcal{T}$, the root must be a point cluster, which means that FULLSPLAY(*node*) ensured $depth(node) \leq 2$. Specifically, the depth is 2 if *node* is a path cluster; otherwise, it is 1 per Lemma 3.8.

If *node* is a path cluster, both u and v must be connected to something else than (u, v), thus, CUT should generate $\mathcal{T}_u$ and $\mathcal{T}_v$. A node is left without a parent for each of the two parents deleted in DELETEALLANCESTORS. These nodes become the new roots for $\mathcal{T}_u$ and $\mathcal{T}_v$ as they are now in disjoint top trees.

In case *node* is a point cluster, only a single parent is deleted in DELETEALLANCESTORS, which is the new root for $\mathcal{T}_u$ or $\mathcal{T}_v$ depending on which vertex is connected to something else. The

other vertex becomes unconnected, and since leaf nodes in the top tree only represent edges, an unconnected vertex isn't part of any top tree. Thus, the CUT operation only generates a single top tree $\mathcal{T}$ corresponding to the spanning tree $T \setminus (u, v)$.

When deleting (u, v) after calling FULLSPLAY on *node*, u and v may be incorrectly counted as a boundary vertex of clusters in which they are not a boundary vertex anymore. Thus, set $u.exposed$ and $v.exposed$ to true, and call DEEXPOSE on them to correctly update the boundary vertices of these clusters.

Lemma 3.17. *The amortized runtime of* CUT *is* $O(\log n)$*, and the potential decreases.*

Proof. Let *node* be the leaf node representing e, then $depth(node)$ SPLIT calls are made, which can be paid for by the potential freed in the FULLSPLAY call.

The CUT operation decreases the potential by deleting a few nodes. At most, 2 internal nodes with at most n leaves are deleted alongside a leaf node. Thus, $O(\log n)$ potential is freed throughout the CUT operation.

The rest of the runtime follows directly from Lemma 3.12 and Lemma 3.15. □

Theorem 3.18. *The external operations of splay top trees run in* $O(\log n \cdot (COST(\text{COMBINE}) + COST(\text{SPLIT})))$ *amortized time.*

Proof. Recall that throughout the analysis, COMBINE and SPLIT were assumed to run in $O(1)$ time to avoid a factor of $O(COST(\text{COMBINE}) + COST(\text{SPLIT}))$ in the amortized cost for each operation. Thus, by multiplying $COST(\text{COMBINE}) + COST(\text{SPLIT})$ on the achieved amortized cost of each operation, the amortized cost for non-constant COMBINE and SPLIT is achieved. Thus, the rest follows directly from the results of Lemma 3.14, Lemma 3.15, Lemma 3.16 and Lemma 3.17. □

3.9 Implementation

The implementation is written in Java and took inspiration from the C implementation provided by Holm et al. in their paper on splay top trees [9]. Their implementation included an operation which checked if the orientation invariant was maintained in the top tree. Expanding this idea, additional checks were implemented, i.e. to ensure EXPOSE was never called on the same node twice. Finally, a method was added to check that no cluster information stored was incorrect due to a restructuring of the tree. This was particularly useful when debugging the 2-edge connectivity algorithm presented in Section 4.

The correctness of the Java implementation was validated by implementing different problems using the top tree and verifying against static solutions, e.g. a minimum spanning tree implementation inspired by Holm et al.[9], which was compared against a Kruskal implementation found on geeksforgeeks.org [2].

4 2-edge Connectivity

In this section, we will consider the problem of 2-edge connectivity for undirected graphs G. The issue boils down to whether a graph remains connected if one edge is removed. In Figure 26, two graphs are visualised. The leftmost graph is 2-edge connected, while the rightmost graph is not. The graph on the right has a bridge marked in red, which is a single edge that can be removed to generate two disjoint components. Thus, detecting whether a graph contains bridges solves the problem.

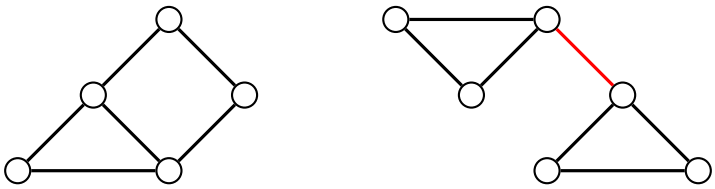


Figure 26: Example of a 2-edge connected graph and one which is not 2-edge connected

Deciding whether an edge e is a bridge can be done by removing e and running a Breadth-first-search(BFS) or Depth-first-search(DFS) on the remaining graph. If all vertices in the graph can be reached, e was not a bridge. While this is a simple solution, it is far from optimal, as determining whether a graph is 2-edge connected would take $O(|E| \cdot (|V| + |E|))$, which for all interesting graphs is $O(|E|^2)$, as a graph can never be 2-edge connected if $|E| < |V|$. Improved bridge detection algorithms have been proposed, such as the linear time algorithm by Robert Tarjan in 1974 that finds bridges in $O(V + E)$ time [18]. Both of these solutions, however, are static. Thus, any change to the graph and the answer must be recomputed entirely from scratch.

A generalised version of this problem exists called k-edge connectivity, where a graph is k-edge connected if removing up to $k - 1$ edges leaves the graph connected. Another version of this problem asks whether two vertices u and v in the graph are 2-edge connected instead of the entire graph, allowing bridges to exist as long as u and v are not connected through a bridge.

This problem is closely related to bi-connectivity, where a graph should remain connected even if a vertex and all its edges are deleted.

4.1 Dynamic solutions

The fully dynamic algorithm using top trees was initially published by Holm et al. at STOC98 [7]. In the same year, Jacob Holm and Kristian de Lichtenberg completed their masters with the title 'Top-trees and dynamic graph algorithms', where additional details were given on the 2-edge connectivity algorithm [6]. In 2001 Holm et al. published a new paper with additional information and improvements over the STOC98 paper[8], e.g. achieving a reduced memory footprint.

The dynamic 2-edge connectivity algorithm can be used to answer whether two vertices are 2-edge connected or if any bridges exist in the top tree data structure. The focus throughout this section will be to answer queries about two vertices.

The goal of Section 4.1.1, Section 4.1.2 and Section 4.1.3 is to prove Theorem 4.1 and provide additional details required to implement the algorithm.

Theorem 4.1. *(Theorem 17 from [8]) A deterministic, fully dynamic algorithm exists for maintaining 2-edge connectivity in a graph, using $O(\log^4 n)$ amortized time per update operation, and $O(\log^3 n)$ for queries.*

4.1.1 High-level overview

In this section, a theoretical version of the dynamic 2-edge connectivity algorithm is proven. The implementation requires additional details to achieve the desired runtimes, which will be considered in Section 4.1.2.

The 2-edge connectivity algorithm uses a top tree $\mathcal{T}$ to store information about clusters in the dynamic graph G. Using a top tree requires an explicitly maintained spanning tree T of the graph, as mentioned in Section 3. Let tree edges refer to edges in T, and non-tree edges refer to edges in G. By the nature of spanning trees only being a subset of edges in a graph, one of the major challenges will be maintaining the spanning tree under edge deletions. Let vertices u and v be connected through edge e in the spanning tree. When deleting e from T and G, the graph may remain connected, and when this is the case, T should reflect this, and a replacement edge should be inserted into T.

To efficiently find these replacement edges, the algorithm stores them as part of the cluster information. Consider the graph G and spanning tree T shown in Figure 27, (u, v) can not be inserted into T as a cycle would be formed. Therefore, (u, v) can instead be used as a replacement edge for any of $(y, u), (y, w)$ and (w, v), so let an edge e be covered by another edge e', if the insertion of e' would create a cycle O in T, with $e \in O$. Let e' be a cover edge of e.

Recall the $\pi(C)$ definition as the path between the boundary vertices of C. $\pi(C)$ is only defined for path clusters, as point clusters only have a single boundary vertex. The cover edge is stored in the cluster information as $C.coverEdge$ if C is a leaf node or all edges on $\pi(C)$ are covered by another edge.

(a) Dynamic graph G (b) Spanning tree T of G

Figure 27: An example of a graph with a possible spanning tree

The addition of cover information makes the deletions of edges more complicated. If an edge e not part of T is used as a cover edge, e.g. (u, v) in Figure 27, it may be stored as the $coverEdge$ in some clusters. These clusters should be updated to reflect the removal of e, as the edges of the cluster are no longer covered by e. Let C be a cluster previously covered by e; $C.coverEdge$ must be updated with a new edge e' if possible. The e' used to cover C may not cover all edges that e did. Thus, multiple edges may have to be considered, which may be expensive if too many edges are considered before finding all the new cover edges. The cost of considering multiple edges will be paid for by introducing a new concept of levels to edges only in G. Let $l(e)$ be the level of e, with $0 \leq l(e) \leq l_{max} = \lceil \log_2 n \rceil$. Now let $G = G_0 \supseteq G_1 \supseteq \ldots \supseteq G_{l_{max}} \supseteq T$ where

G_i denote a subgraph of edges f with $l(f) \geq i$. The following invariant should be maintained in each graph.

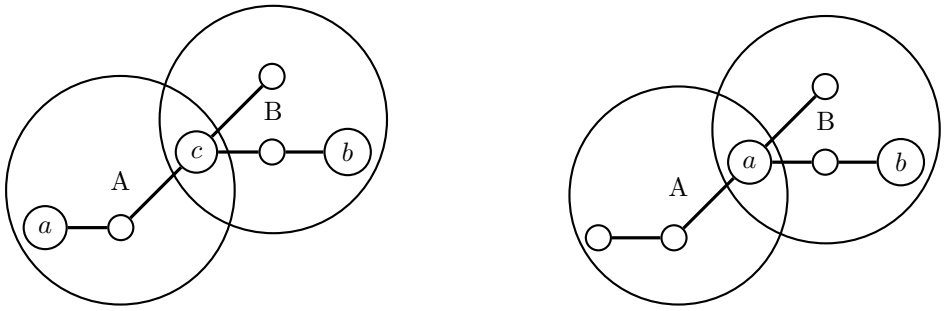
1. The maximum number of vertices of a 2-edge connected component of G_i is $\lceil n/2^i \rceil$.

The goal is to amortize the cost of finding new cover edges over level increases. We can amortize the cost of all but 2 of the edges considered with careful selection.

With the additional information about edge levels, update the cluster information stored in C to also include $C.cover = l(C.coverEdge)$, where $C.coverEdge$ is the edge $e \in \pi(C)$ with the lowest level. Let $C.cover = -1$ indicate that C isn't covered by any edge. $C.cover$ and $C.coverEdge$ is the only information required for the theoretical version of 2-edge connectivity.

Let COMBINE and SPLIT be the first operations considered as the top tree calls them. The theoretical version of 2-edge connectivity does not require the SPLIT operation, as other methods will handle the propagation of information for simplicity. However, to achieve the desired amortized runtimes, SPLIT is introduced in Section 4.1.2. Thus, consider the COMBINE operation, which ensures the top tree operations update the cluster information as needed throughout any restructuring.

Since only path clusters store $C.cover$ and $C.coverEdge$ information, the number of cases we must consider for COMBINE is reduced. Let C be a path cluster with children A and B and boundary vertices a and b. The cases that need to be considered are visualised in Figure 28.



(a) Case 1 (b) Case 2

Figure 28: COMBINE cases

Case 1: Let A and B be a path clusters, with boundary vertices a, c and b, c respectively as seen in Figure 28a. If A and B are covered, then the entire path $a, \ldots, c, \ldots, b$ must be covered, which means C is covered. If either A or B is uncovered, then so is C. Therefore, we can set $C.cover = \min(A.cover, B.cover)$ and update $C.coverEdge = A.coverEdge$ if $A.cover < B.cover$ otherwise $C.coverEdge = B.coverEdge$.

Case 2: Let A be a point cluster with boundary vertex a. It must be the case that B is a path cluster with boundary vertices a, b; otherwise, C can not be a path cluster. In this case, C is covered if and only if B is. This can easily be seen in Figure 28b, where $\pi(B) = \pi(C)$. The case is similar for B as a point cluster. Set $C.cover = B.cover$ and $C.coverEdge = B.coverEdge$.

Finally, if C is a point cluster set, $C.cover = -1$ and $C.coverEdge = \text{nil}$.

Algorithm 16 COMBINE(C)

```
1: A = left child of C
2: B = right child of C
3: if isPath(C) then
4:     if isPoint(B) then
5:         C.cover = A.cover
6:         C.coverEdge = A.coverEdge
7:     else if isPoint(A) then
8:         C.cover = B.cover
9:         C.coverEdge = B.coverEdge
10:    else
11:        C.cover = min(A.cover, B.cover)
12:        if A.cover < B.cover then
13:            C.coverEdge = A.coverEdge
14:        else
15:            C.coverEdge = B.coverEdge
16: else
17:    C.cover = −1
18:    C.coverEdge = nil
```

With the COMBINE method handled, we can start considering operations specific to the 2-edge connectivity algorithm, beginning with the query. To compute whether u and v are 2-edge connected, expose both and check if $C.cover \geq 0$. This gives the desired result as $C.cover \geq 0$ if and only if all edges $e \in \pi(C)$ are covered.

Algorithm 17 TWOEDGECONNECTIVITY(u, v)

```
1: EXPOSE(u)
2: C = EXPOSE(v)
3: result = C.cover ≥ 0
4: DEEXPOSE(u)
5: DEEXPOSE(v)
6: return result
```

What remains to be shown is how edges are inserted and deleted from the graph while correctly updating the cluster information.

Inserts

A few operations are required before considering the INSERT operation shown in Algorithm 20. Let COVER(C, i, e) be used to update *cover* and *coverEdge* for edges in $\pi(C)$. The new information is only inserted into C and must then be propagated downwards to all path clusters. In Section 4.1.2, this will be done using SPLIT, but for simplicity, use the RECURSIVECOVER operation shown in Algorithm 19, which propagates the information immediately. By propagating the information to all path clusters, the edges along $\pi(C)$ are updated, while the cover information of all other edges in C is left unchanged.

Algorithm 18 COVER(u, v, i)

```
1: EXPOSE(u)
2: C = EXPOSE(v)
3: e = edge between u and v
4: if C.cover < i then
5:     C.cover = i
6:     C.coverEdge = e
7: RECURSIVECOVER(C, i, e)
8: DEEXPOSE(v)
9: DEEXPOSE(u)
```

The RECURSIVECOVER operation takes a starting cluster C, along with a cover level i and edge e. It then checks if path descendent D has $D.cover < i$; if so, update $D.cover = i$ and $D.coverEdge = e$. If $D.cover \geq i$, then D is already covered at level i, and we don't have to update it. Finally, recursively call RECURSIVECOVER(D, i, e), so path descendants of D are also updated.

Algorithm 19 RECURSIVECOVER(C, i, e)

```
1: if C.isLeaf then
2:     return
3: else
4:     for each path descendent D of C do
5:         if D.cover < i then
6:             D.cover = i
7:             D.coverEdge = e
8:         RECURSIVECOVER(D, i, e)
```

Let $e = (u, v)$ be a newly inserted edge in G; then, there are two scenarios for INSERT to handle. If the endpoints of e are in two different components of T, insert e in T and call LINK(u, v). The call to LINK handles the creation of a new leaf node along with the merger of $\mathcal{T}_u$ and $\mathcal{T}_v$. The cover information is also updated for the new nodes created as part of LINK, none of which are covered, as they all contain the newly inserted uncovered edge.

If u and v are already connected in T, the new edge e can be used as a cover edge for all edges in the cycle generated by inserting e into T. Let u and v be exposed with root cluster C. The cycle generated by (u, v) goes through the edges in $\{\pi(C) \cup (u, v)\}$, thus by calling COVER on C, the cover information of the desired edges are updated.

Finally, the graph invariant should be maintained. Notably, the newly inserted edge has $l(e) = 0$. By the orientation invariant, G_0 can contain all vertices in a 2-edge connected component.

Algorithm 20 INSERT(u, v)

```
1: if u and v in same spanning tree then
2:     COVER(u, v, 0)
3: else
4:     add (u, v) to T
5:     LINK(u, v)
```

Deletes

The most difficult part of dynamic 2-edge connectivity comes from deleting edges. In particular, difficulties arise when deleting covered spanning tree edges or edges currently used to cover edges in the spanning tree.

Let $c(e)$ be defined for any tree edge e as the level e is covered at. Then consider the problem of deleting an uncovered tree edge e from T. Since $c(e) = -1$, no edge $e' \in G$ exists, which can replace e such that T remains a connected component. Thus, we can remove e from T and call CUT to split $\mathcal{T}$ correctly into multiple top trees.

If $c(e) \geq 0$, some edge e' can replace e in T. By swapping e' and e, the problem is reduced to deleting a non-tree edge in G, which may have been used to cover edges in T. Deleting the now non-tree edge e from the graph can be done by first uncovering any edges in T that may have been covered by e. This may leave some of the edges with cover levels that are incorrect. Thus, a recovery phase is required to restore the cover levels to the correct values. Finally, the edge e should be deleted from the graphs G_i with $i \leq l(e)$, handled by a DELETEEDGE operation.

The SWAP, UNCOVER, and RECOVER operations will be explored in greater detail throughout the rest of this section.

Algorithm 21 DELETE(u, v)

```
1: if (u,v) is an edge in the spanning tree then
2:     if (u,v) is a bridge then
3:         remove (u, v) from T
4:         CUT(u,v)
5:     else
6:         SWAP(u,v)
7:         i = l(u, v)
8:         UNCOVER(u,v,i)
9:         DELETEEDGE(u,v)
10:        RECOVER(u,v,i)
11: else
12:    i = l(u,v)
13:    UNCOVER(u,v,i)
14:    DELETEEDGE(u,v)
15:    RECOVER(u,v,i)
```

The SWAP operation finds the edge e' that covers (u, v) and use it to replace (u, v) in T. The cover edge is found by exposing u and v, with path cluster C being the new root. With $(u, v) \in T$,

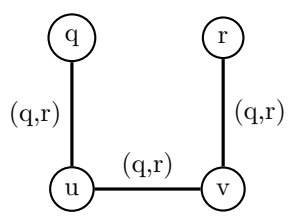
the only edge in $\pi(C)$ is (u, v) itself, thus, $C.coverEdge$ can be used as e'. The swap of (u, v) and e' can be implemented by calling CUT and LINK and using (u, v) to cover the new edge.

Algorithm 22 SWAP(u, v)

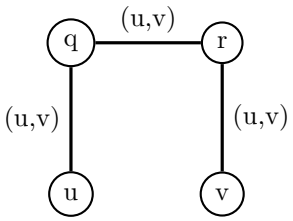
```
1: EXPOSE(u)
2: C =EXPOSE(v)
3: if C.Cover ≥ 0 then
4:     e' = C.coverEdge
5:     swap (u, v) with e' in T
6:     CUT(u, v)
7:     LINK(e')
8:     COVER(u, v, C.cover)
```

SWAP should correctly update the cover information of T, and maintain the graph invariant. Consider the graph invariant first, before the SWAP call it is satisfied, which means that no 2-edge connected component in G_j contains more than $\lceil n/2^j \rceil$ vertices. The invariant must be maintained if all the 2-edge connected components stay the same. Let i be the cover level of the swapped edge. Then for $j \leq i$, both edges are already in G_j, which means the connected components do not change. For $j > i$, only the edge (u, v) is in G_j and is a bridge. When swapping the edges, e' becomes a bridge in G_j, and since bridges do not impact 2-edge connected components, they remain unchanged in G_j.

Finally, it should be shown that the cover levels are correctly updated. If an edge was covered by (q, r) at level i before the swap, it is covered by $e = (u, v)$ afterwards. The call to COVER(u, v, i) updates the cover edges of the relevant cluster. The call also updates the cover edge and level of the newly inserted edge e'. Thus, all the cover levels are correctly updated. An example of how the cover edges are updated can be seen in Figure 29.



(a) T before SWAP with cover edges



(b) T after SWAP with cover edges

Figure 29: Example of SWAP, with cover edges shown

The next operation is UNCOVER. Let e be a non-tree edge, eg. the edge $e = (u, v)$ in Figure 29b. Before deleting e, UNCOVER must ensure e is not used as the cover edge for any clusters. This is done by removing all cover information for edges e' between u and v if $c(e') \leq i$ with $i = l(e)$. This may leave some edges with $cover = -1$ and $coverEdge = nil$, even though some edge in G_j covers them. The cover levels of these edges will be recovered in the recovery phase.

Similarly to COVER, the information is deleted in the root C and propagated down to its path descendants. The information is propagated downwards using RECURSIVEUNCOVER, which works

similarly to RESURSIVECOVER. The graph invariant can never be broken when removing edges from G_i since it can only decrease the size of the 2-edge connected components.

Algorithm 23 UNCOVER(u, v, i)

```
1: EXPOSE(u)
2: C =EXPOSE(v)
3: if C.cover ≤ i then
4:     C.cover = −1
5:     C.coverEdge = nil
6: RECURSIVEUNCOVER(C, i)
7: DEEXPOSE(v)
8: DEEXPOSE(u)
```

Algorithm 24 RECURSIVEUNCOVER(C, i)

```
1: if c.isLeaf then
2:     return
3: else
4:     for each path descendent D of C do
5:         if D.cover ≤ i then
6:             D.cover = −1
7:             D.coverEdge = nil
8:         RECURSIVEUNCOVER(D, i)
```

Before considering the recovery phase, a new definition is required. Let C be a cluster with boundary vertices u and v, then $\pi(C) = u, \ldots, v$ is considered fine on level i if the cover value is correct for edges $e \in \pi(C) = u, \ldots, v$, that are covered by an edge e' with $l(e') \geq i$. If e is only covered by an edge e' with $l(e') < i$, $c(e)$ may have an incorrect lower value. Additionally, let $meet(u, v, w)$ return the unique vertex, where the paths of $u, \ldots, v$, $u, \ldots, w$ and $v, \ldots, w$ meets.

The goal for RECOVER(u, v, i) is then to restore $u, \ldots, v$ to be fine on level i after the deletion of edge (u, v). Unfortunately, this means after a call to RECOVER, only level i is corrected, and thus, $u, \ldots, v$ might not be fine on $i-1$. Therefore, RECOVER(u, v, j) must be called for all values $0 \leq j \leq i$, starting at $j = i$, then $j = i - 1$ and so on.

The most straightforward implementation of RECOVER would be to consider all edges $f = (q, r)$ of level i and call COVER(q, r, i). However, by considering all edges, the RECOVER operation will be very expensive; instead, a more complex RECOVER operation is used.

The RECOVER algorithm is two-phased. The first phase starts with $w = u$, while the second starts with $w = v$. Then consider all edges $f = (q, r)$ only in G, with $meet(u, v, q) == w$, and $c(e) == i$. For each of these edges, check if the level of f can be increased without breaking the graph invariant. If so, do it and call COVER$(q, r, i + 1)$. If not, call COVER(q, r, i) and exit the phase. If all edges with $meet(u, v, q) == w$ were increased, let w be the next vertex on the path between u and v and repeat.

Algorithm 25 RECOVER(u, v, i)

```
1:  // Phase 1
2:  for Vertex w on the path from u to v do
3:      for Edge f =(q, r) with meet(u, v, q) == w and c(e) == i do
4:          if increasing the level of e is legal then
5:              l(e) = i + 1
6:              COVER(q, r, i + 1)
7:          else
8:              COVER(q, r, i)
9:              Exit both for loops
10: // Phase 2
11: for Vertex w on the path from v to u do
12: // Same as phase 1, but starting from v
13:     ⋮
```

What remains to be shown is that RECOVER correctly restores the cover information without breaking the graph invariant. The stopping condition of each phase in RECOVER is exactly when increasing the level would break the graph invariant. Thus, the invariant is satisfied at the end of all the RECOVER calls.

Lemma 4.2. *(Lemma 16 from [8]) Assume $u, \ldots, v$ is fine on level $i+1$. Then after a call to RECOVER(v, u, i), it is fine on level i.*

Proof. First, consider the impact RECOVER may have on level $i+1$. By the assumption, all edges which should be covered at $i+1$ contain correct information already. Throughout the RECOVER operation, COVER may be called with i or $i+1$. Calls with i can not impact level $i+1$. Finally, calling COVER with $i+1$ updates the cover levels of edges covered by the newly increased edge. Thus, $u, \ldots, v$ remains fine on level $i+1$.

Let a relevant edge be defined as follows. Let (q, r) be an edge in G_i, but not in T shown in Figure 30. (q, r) is relevant if $q, \ldots, r \cap u, \ldots, v \neq \emptyset$. The intersected edge has been marked in red. By calling either COVER(q, r, i) or COVER$(q, r, i+1)$, the cover level of the red edge would be corrected.

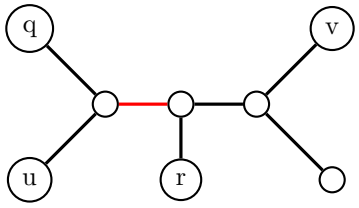


Figure 30: Showcasing of a relevant edge (q, r) in T

To argue that $u, \ldots, v$ is fine on level i when RECOVER terminates, we must show that for all edges along $u, \ldots, v$ whose cover level should be $\geq i$ is updated to reflect that.

Consider at first that phase 1 does not stop, then all relevant non-tree edges have been considered, and thus, the cover levels of edges on the path between u and v have been updated to $i+1$ if

possible. The edges whose cover level was unchanged are not covered by any edges at level i and have $cover < i$ as the path was fine on level $i+1$ before the RECOVER call. Thus, the path between u and v is fine on level i if the phase does not stop.

Now consider the case where phase 1 stops. Let w_1 be the last vertex considered along the path, and let (q_1, r_1) be the edge whose level could not be increased. If phase 1 stops, then phase 2 must also stop at some vertex w_2. If phase 2 completes, it must have gone through w_1 and increased the level of (q_1, r_1), which was illegal. Thus, (q_2, r_2) is the last edge considered in phase 2.

The part of phases 1 and 2 that are completed ensures that the cover level of $u, \ldots, w_1$ are correct, similarly with $w_2, \ldots, v$. Thus, we have to show that $w_1, \ldots, w_2$ are fine on level i.

Due to the illegality of increasing the level of (q_1, r_1), the 2-edge connected component H_1 containing q_1 in $G_{i+1} \cup (q_1, r_1)$ must have $> \lceil n/2^{i+1} \rceil$ vertices. The same goes for $H_2 = G_{i+1} \cup (q_2, r_2)$. Before deleting (u, v), H_1 and H_2 were part of the same 2-edge connected component H in G_i. H had at most $\lceil n/2^i \rceil$ vertices, thus $H_1 \cap G_2 \neq \emptyset$. This means H_1 and H_2 shares vertices and must be contained in the same 2-edge connected component H' of $G_{i+1} \cup (q_1, r_1) \cup (q_2, r_2)$. Since the covering has been performed for $i+1$ edges, the calls to COVER(q_1, r_1, i) and COVER(q_2, r_2, i) ensures that edges in H' are covered with $cover \geq i$.

Finally, we have that $T \subseteq G_{l_{max}} \subseteq \ldots \subseteq G_i$. If q_1 and r_1 are connected through w_1 in T, which must be the case for phase 1 to end on (q_1, r_1), it must be the case that $w_1 \in H_1$. The same goes for $w_2 \in H_2$, which implies that $w_1, \ldots, w_2 \in H'$, which means all edges on the path between w_1 and w_2 have the desired cover level of $\geq i$. □

To reiterate, it requires multiple calls to RECOVER(u, v, j) to correct the cover information that may have been left incorrectly by UNCOVER. When a RECOVER operation terminates, there is only a guarantee that the path between u and v is fine on level j. This is solved by calling REOVER with all values from $0 \leq j \leq i$, then only values < 0 can be incorrect. But with -1 being the lowest value, all cover values are correct, and so is the theoretical 2-edge connectivity algorithm.

4.1.2 Implementation details

In this section, the operations and cluster information discussed in Section 4.1.1 is modified to support an efficient implementation of 2-edge connectivity.

Consider the COVER and UNCOVER operations described in Algorithm 18 and Algorithm 23, respectively. The new information was only inserted at the root and then propagated down the tree. This propagation may be expensive if most of the edges in T are on the path between the exposed vertices. Therefore, the propagation is delayed until the cluster information changes by adding lazy information in COVER and UNCOVER. The delayed propagation is handled by implementing the top tree operation SPLIT[8].

The following information, $cover^+, cover^-, coverEdge^+$, is added to the cluster information. Similarly to *cover* and *coverEdge*, they are only defined for path clusters C, as they still depend on the path $\pi(C)$. The $cover^+$ and $coverEdge^+$ are used as lazy cover information, while $cover^-$ is used for uncovering. Let SPLIT be defined to propagate information to the direct path descendants D of C in a similar way to RECURSIVECOVER and RECURSIVEUNCOVER used in Section 4.1.1. The biggest difference is the removal of the recursive call. Thus, if $D.cover \leq \max\{C.cover^-, C.cover^+\}$ set $D.cover = C.cover^+$ and $D.coverEdge = C.coverEdge^+$, and

update the lazy cover information. The simplest implementation is done by reusing parts of the soon-to-be-updated COVER and UNCOVER operations. Let COVERINNER and UNCOVERINNER be operations which handled the updating of cluster information during COVER and UNCOVER calls.

If $cover^+ = cover^- = -1$, the lazy information can be ignored, as the cluster holds no information about new coverings or uncoverings that is yet to be propagated down.

Algorithm 26 SPLIT(C)

```
1: for child D of C do
2:     if isPath(D) then
3:         uncoverInner(D, C.cover⁻)
4:         coverInner(D, C.cover⁺, C.coverEdge⁺)
5: C.cover⁺ = −1
6: C.cover⁻ = −1
7: C.coverEdge⁺ = nil
```

The lazy cover information allows for efficient implementations of COVER and UNCOVER. The RECOVER operation remains slow for finding relevant edges and checking if the graph invariant is broken if an edge level is increased. To solve these challenges, add two types of counters, *incident* and *size*. These counters will be stored in vertices and clusters, with different information stored at each place. For each vertex and point cluster, $O(\log n)$ counters are introduced, whereas, for each path cluster, $O(\log^2 n)$ counters are added.

First, for any vertex v and level i, introduce $incident_{v,i}$, which is the number of non-tree edges in G_i incident to v. For simplicity, $size_{v,i}$ is also defined but is always 1.

Let C be a point cluster, and let $I_{C,v,i}$ be the set of internal vertices u in C, reachable by a path P with $c(e) \geq i$ for all edges $e \in P$. The following counters are then stored.

- $size_{C,v,i} = |I_{C,v,i}|$
  $size_{C,v,i}$ denotes the size of the 2-edge connected component of C in G_i containing v.
- $incident_{C,v,i} = \sum_{w \in I_{C,v,i}} incident_{w,i}$
  $incident_{C,v,i}$ denotes the number of non-tree edges (q, r) with $q \in I_{C,v,i}$.

Now consider C as a path cluster. Let $I_{C,v,i,j}$ be the set of internal vertices that are reachable by some path P in T, where P starts in v, with $c(e) \geq i$ for edges e in $P \cap \pi(C)$ and $c(e) \geq j$ for edges e in $P \setminus \pi(C)$.

Finally, for the path clusters store:

- $size_{C,v,i,j} = |I_{C,v,i,j}|$
  $size_{C,v,i,j}$ denotes the size of the 2-edge connected components in C containing v.
- $incident_{C,v,i,j} = \sum_{w \in I_{C,v,i,j}} incident_{w,i}$
  $incident_{C,v,i,j}$ denotes the number of non-tree edges (q, r) with $q \in I_{C,v,i,j}$

There is a set of size and incident counters stored for each boundary vertex v of the path cluster.

Updated operations: We are ready to revisit the operations with the added cluster information. At first, consider the operations that change the spanning tree and graphs, INSERT, DELETE, RECOVER and SWAP. These operations should update the incident counters of vertices,

as they can add edges, remove edges or both from the graphs. For INSERT, DELETE and SWAP, these are the only changes. Thus, they have been left out of this section.

Instead, consider COVER, for reusability parts of the COVER operation are extracted into COVERINNER. The COVER operation shown in Algorithm 27 updates the cluster information of C obtained by exposing u and v and calling COVERINNER. $C.cover$ and $C.coverEdge$ is updated similarly to Algorithm 18. Thus, the focus will be on maintaining the newly introduced lazy information and counters.

Algorithm 27 COVER(u, v, i)

```
1: EXPOSE(u)
2: C =EXPOSE(v)
3: e = edge between (u, v)
4: COVERINNER(C, i, e)
5: DEEXPOSE(u)
6: DEEXPOSE(v)
```

Algorithm 28 COVERINNER(C, i, e)

```
1: if C.cover < i then
2:     C.cover = i
3:     C.coverEdge = e
4: if i < C.cover+ then // Do nothing
5: if C.cover+ ≤ i ≤ C.cover− then
6:     C.cover+ = i
7:     C.coverEdge+ = e
8: if C.cover− < i then
9:     C.cover− = i
10:    C.cover+ = i
11:    C.coverEdge+ = e
12: for −1 ≤ k ≤ i do
13:    for −1 ≤ j ≤ l_max do
14:        for v in boundaryVertices(C) do
15:            size_{C,v,k,j} = size_{C,v,−1,j}
16:            incident_{C,v,k,j} = incident_{C,v,−1,j}
```

Let e be the new cover edge with $i = l(e)$. If $i < C.cover^+$, C is already covered by some edge e' with $i < l(e')$. If $C.cover^+ \leq i \leq C.cover^-$, $C.cover^+$ and $C.coverEdge^+$ is updated to reflect the new cover edge with cover level i. Finally, if $C.cover^- < i$, the $C.cover^-$ value can effectively be ignored by setting $C.cover^- = C.cover^+ = i$, as the new edge covers all clusters whose level should have been decreased.

Consider the value $size_{C,v,-1,j}$ to see that the counter values are correctly updated. All vertices on $\pi(C)$ are included in the internal set of reachable vertices $I_{C,v,-1,j}$ as all edges are covered at ≥ -1 at all times. Thus, when increasing the cover levels of all edges on $\pi(C)$, the same vertices should be included in $I_{C,v,i,j}$ and all values between -1 and i. This means $size_{C,v,-1,j}$ and $incident_{C,v,-1,j}$ can be used directly instead of recomputing anything.

Thus, all the information remains correct after a call to COVER.

Lemma 4.3. *The amortized cost of* $O(\log n(COST(\textsc{combine}) + COST(\textsc{split})) + \log^2 n)$ *is achieved for* COVER.

Proof. The runtime follows directly from the calls to EXPOSE, DEEXPOSE and updating of the counters. In the worst case, $i = l_{max}$, which makes it $O(\log^2 n)$ counters to update. □

Similarly to COVER, parts of UNCOVER are extracted. The UNCOVER operation shown in Algorithm 29 updates the cluster information of C obtained by exposing u and v and calling UNCOVERINNER. $C.cover$ and $C.coverEdge$ is updated similarly to Algorithm 23. Thus, the focus will be on maintaining the newly introduced lazy information and counters.

Algorithm 29 UNCOVER(u, v, i)

```
1: EXPOSE(u)
2: C = EXPOSE(v)
3: UNCOVERINNER(C, i)
4: DEEXPOSE(u)
5: DEEXPOSE(v)
```

Algorithm 30 UNCOVERINNER(C, i)

```
1: if C.cover ≤ i then
2:     C.cover = −1
3:     C.coverEdge = nil
4: if i < C.cover⁺ then
5:     do nothing
6: if C.cover⁺ ≤ i then
7:     C.cover⁺ = −1
8:     C.cover⁻ = max(C.cover⁻, i)
9:     C.coverEdge⁺ = nil
10: for 0 ≤ k ≤ i do
11:     for 0 ≤ j ≤ l_max do
12:         for v in c.boundaryVertices do
13:             size_{C,v,k,j} = size_{C,v,i+1,j}
14:             incident_{C,v,k,j} = incident_{C,v,i+1,j}
```

If $i < C.cover^+$, the cluster is covered by some edge e with $c(e) > i$, and thus, the lazy information should still reflect that e is covering this cluster. Otherwise, if $C.cover^+ \leq i$, the cluster should be considered uncovered.

Similarly to COVER, the counters should be updated when UNCOVER is called. For any value, $0 \leq j \leq i$, the edges are now uncovered if the cluster was previously covered at i or below; in this case, the values should reflect those stored in $size_{C,v,i+1,k}$ and $incident_{C,v,i+1,k}$. Importantly, the values for $j = -1$ are not updated, as those values should include all vertices even if nothing is covered. In the original paper, these values are incorrectly updated as well [8].

Thus, all the information remains correct after a call to UNCOVER.

Lemma 4.4. *The amortized cost of* $O(\log n(COST(\textsc{combine}) + COST(\textsc{split})) + \log^2 n)$ *is achieved for* UNCOVER.

Proof. The runtime follows directly from the calls to EXPOSE, DEEXPOSE and updating of the counters. In the worst case, $i = l_{max}$, which makes it $O(\log^2 n)$ counters to update. □

Before considering the updated RECOVER operation. Let us define a FIND operation used to find relevant edges in the top tree. Assume $incident_{C,w,-1,i} > 0$, then some internal vertex u is reachable through a path P, with $incident_{u,i} > 0$. The goal is the find the non-tree edge incident to u efficiently.

The reason -1 is used in $incident_{C,w,-1,i}$ is to account for FIND only being called in RECOVER. When working with the counter values in RECOVER, they may be incorrect due to the UNCOVER call. If $incident_{C,w,i,i}$ were used, only vertices reachable by edges covered at $i + 1$ before the latest deletion would be included.

Algorithm 31 FIND(w, C, i)

```
1: if incident_{w,i} > 0 then
2:     return non-tree edge e incident to u with c(e) = i
3: SPLIT(C)
4: Let A and B be children of C
5: Let w be a boundary of A
6: if ISPATH(A) and incident_{A,w,-1,i} > 0 then
7:     FIND(w, A, i)
8: else if ISPOINT(A) and incident_{A,w,i} > 0 then
9:     FIND(w, A, i)
10: else
11:     Let v be the boundary vertex closest to w in B
12:     FIND(v, B, i)
```

If $incident_{w,i} > 0$, w is incident to some edge e with $c(e) = i$ and FIND can immediately return e.

Thus, consider the case where $incident_{w,i} = 0$, but $incident_{C,w,-1,i} > 0$. To locate one of the edges e with $c(e) = i$, let w be a boundary vertex of A and A a child of C. Check $incident_{A,w,i} > 0$ and $incident_{A,w,-1,i} > 0$ for point and path clusters respectively and call FIND(A, w, i) to narrow the search space if either is true. At some point, $incident_{A,w,i} = 0$ or $incident_{A,w,-1,i} = 0$ is true, which means the incident edge is located in B, the other child of C. In this case, call FIND(B, b, i) with b being the boundary vertex closest to w in B, by continuing this recursion at some point a boundary vertex b is found with $incident_{b,i} > 0$, and an edge is returned.

Lemma 4.5. *The amortized cost of* FIND *is* $O(depth(d) \cdot \log^2 n)$*, where* d *where the recursion depth terminated and an edge* e *is found.*

Proof. Each recursive call takes at most $O(\log^2 n)$ time due to the SPLIT call and moves one level down the tree. □

The updated RECOVER operation is also two-phased. The first phase corresponds to running Algorithm 32 with $w = u$, and the second phase with $w = v$. RECOVER still has to be called for all values of $-1 \leq j \leq i$, starting with $j = i$.

Algorithm 32 RECOVER(u, v, w, i)

```
1: EXPOSE(u)
2: C = EXPOSE(v)
3: while incident_{C,w,-1,i} + indident_{u,i} > 0 do
4:     (q, r) = FIND(w, C, i) // returns an edge
5:     DEEXPOSE(v)
6:     DEEXPOSE(u)
7:     EXPOSE(q)
8:     D = EXPOSE(r)
9:     if size_{D,q,-1,i+1} + 2 > ⌈n/2^{i+1}⌉ then // +2 is the external boundary vertices
10:        COVERINNER(D, i, (q, e))
11:        break
12:    else
13:        l(q, r) = i + 1
14:        update incident counters of q and r at level i and i + 1
15:        COVERINNER(D, i + 1, (q, r))
16:    DEEXPOSE(r)
17:    DEEXPOSE(q)
18:    EXPOSE(u)
19:    C = EXPOSE(v)
20: DEEXPOSE(v)
21: DEEXPOSE(u)
```

RECOVER does not change the cluster information directly, as all cluster information is updated through COVERINNER, EXPOSE and DEEXPOSE. Only the vertex counters are updated directly on line 14. Therefore, assume the user-defined information is updated correctly, and focus on maintaining the invariant.

Recall from the DELETE operation in Algorithm 21, RECOVER is only called after uncovering the path between u and v. Thus, the $incident_{C,w,j,k}$ counters may have been affected for any $j \leq i$ if $C.cover$ was $\leq i$ before the UNCOVER call. But the goal is to recover the cover level of $u, \ldots, v$ to i; thus, $incident_{C,w,-1,i}$ is used, as this value correctly reflects the ideal world where $u, \ldots, v$ was covered at level i. In particular, $incident_{C,w,-1,i}$ tells us if any internal vertex, with a non-tree edge e with $l(e) = i$, could be reached through a path P covered at i before the UNCOVER call. If this is the case, the FIND operation is used to find and return such an edge.

Once an edge e has been found, the RECOVER operations call DEEXPOSE and EXPOSE to restructure the tree such that the endpoints of e are exposed. Let D be the new root, and $size_{D,q,-1,i+1}$ denote the size of the 2-edge connected component containing q in G_{i+1} created by increasing the level of e to $l(e) = i + 1$. If $size_{D,q,-1,i+1} + 2 \leq \lceil n/2^{i+1} \rceil$, the level of e can be increased without breaking the graph invariant, followed by a call to COVERINNER. 2 must be added to account for the boundary vertices of D, as $size$ only includes the internal vertices. If $size_{D,q,-1,i+1} + 2 > \lceil n/2^{i+1} \rceil$, immediately call COVERINNER and terminate the phase.

The correctness of stopping after failing to increase $l(e)$ follows through a similar proof as Lemma 4.2.

Lemma 4.6. *The amortized cost of* RECOVER *is at most* $O(\log n + t \cdot \log n) \cdot (COST(\text{COMBINE}) + COST(\text{SPLIT}))$*, where* t *is the number of edges whose level was increased.*

Proof. Independently of whether any iteration of the while loop is run, the RECOVER operation makes calls to EXPOSE and DEEXPOSE. These operations take $O(\log n \cdot (COST(\text{COMBINE}) + COST(\text{SPLIT}))$ each.

The while loop runs until it is stopped or no relevant edges exist. To achieve the desired runtime, some of the iterations must be paid for by the edge-level increases. Therefore, any iteration that increases the level of an edge is counted and paid for separately. This means, at most, a single iteration has to be paid for directly in each RECOVER call.

Each while loop iteration makes multiple calls to EXPOSE and DEEXPOSE. The call to FIND takes $O(depth(x) \cdot \log^2 n)$ time, where x is the node in which the relevant edge (q, r) is found. However, the top tree operations can pay for FIND, as both the SPLIT and COMBINE operations take $O(\log^2 n)$ time in 2-edge connectivity(The updated COMBINE operation is considered next). Thus, by the assumption that each free potential can pay for $c(COST(\text{COMBINE}+COST(\text{SPLIT})))$ work each, the $\Omega(depth(node))$ potential freed by the calls to FULLSPLAY in EXPOSE can pay for the FIND operation. To support this claim, a small experiment was performed to ensure *node* had a greater depth than the recursion depth of FIND. □

Combine

The only remaining function to update is the COMBINE(C) operation. The COMBINE operation updates *cover* and *coverEdge* similarly to the version considered in Section 4.1.1. Thus, let the focus be on how the counters are updated for path and point clusters, respectively. The lazy information is ignored completely, as it is only used to push information down the tree, which it should already have been before COMBINE is called on a node.

The counters will be considered independently for point and path clusters as they are computed differently and update different counters. Noteworthy, the computations for *size* and *incident* counters are equal in all cases. Thus, only the computation for *size* is included.

Point Let C be a point cluster with boundary vertices a and child clusters A and B. Consider the case where A is a path cluster with boundary vertices a, b, and B is a point cluster with boundary vertex b. This case can be seen in Figure 31, where the boundary vertices of A and B are marked with a and b.

Let $size_{C,a,j}$ be the current counter to update. $size_{A,a,j,j}$ contains the number of internal vertices reachable from a by only traversing edges covered at level j. Thus, we can use this as a baseline value. However, if $A.cover \geq j$, all edges on $\pi(A)$ are covered at level j. Thus, it is possible to reach vertices in B as well. The number of reachable vertices in B is already stored in $size_{B,b,j}$; thus, we can add this onto our baseline. Finally, $size_{b,j}$ must be added as it isn't considered internal in either A or B.

Therefore, the final computation is

$$size_{C,a,j} = size_{A,a,j,j} + (size_{B,b,j} + size_{b,j} \text{ if } A.cover \geq j)$$

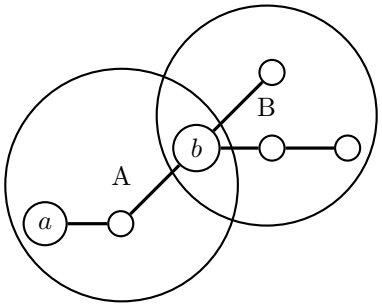


Figure 31: Combine cluster example

The case above and the others with C being a point cluster have been inserted into Table 5.

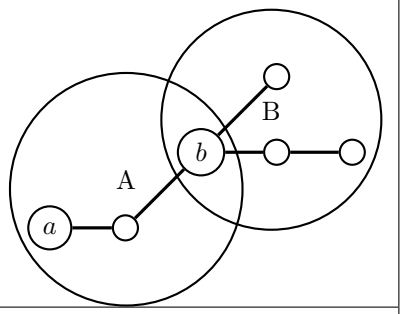
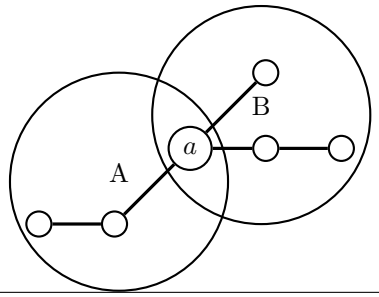
Case	Computation	Drawing
Let A be a path cluster with boundary vertices a, b and B a point cluster with boundary b	$size_{C,a,j} = size_{A,a,j,j} + (size_{B,b,j} + size_{b,j}$ *if* $A.cover \geq j)$	
Let A amd B be point clusters with boundary vertex a	$size_{C,a,j} = size_{A,a,j} + size_{B,b,j}$	

Table 5: COMBINE computations of point cluster C

Path Let C be a path cluster with boundary vertices a and b and child clusters A and B. The possible cases for path clusters can be seen in Table 6. The cases are only shown for computation of $size_{C,a,i,j}$. However, they are equivalent for $size_{C,b,i,j}$.

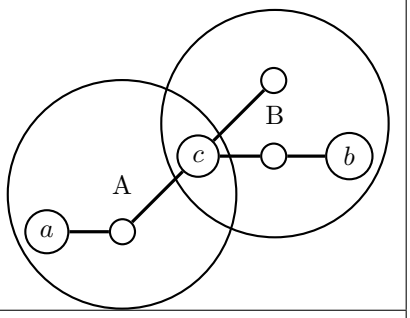
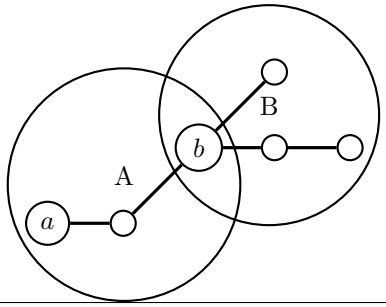
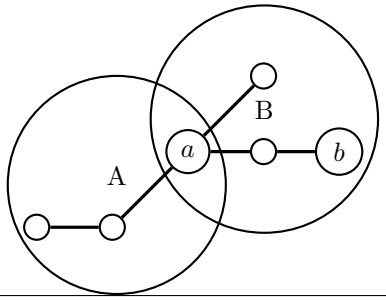
Case	Computation	Drawing
Let both A and B be path clusters with boundary vertices $a, c \in A$ and $b, c \in B$	$size_{C,a,i,j} = size_{A,a,i,j} + (size_{B,c,i,j} + size_{c,i}$ *if* $A.cover \geq i)$	
Let A be a path cluster with $a, b \in A$, and B a point cluster with $b \in B$	$size_{C,a,i,j} = size_{A,a,i,j} + (size_{B,c,i,j}$ *if* $A.cover \geq i)$	
Let A be a point cluster with $a \in A$, and B a path cluster with $a, b \in B$	$size_{C,a,i,j} = size_{A,a,j} + size_{B,a,i,j}$	

Table 6: COMBINE computations of path cluster C

4.1.3 Runtime analysis

Theorem 4.1. *(Theorem 17 from [8]) A deterministic, fully dynamic algorithm exists for maintaining 2-edge connectivity in a graph, using $O(\log^4 n)$ amortized time per update operation, and $O(\log^3 n)$ for queries.*

Proof. Let us consider the runtime of the top tree operations CUT, LINK, EXPOSE and DEEXPOSE first. They all have a runtime of $O(\log^3 n)$ when factoring in the COMBINE and SPLIT runtimes of $O(\log^2 n)$.

Throughout the DELETE operations, calls may be made to DELETEEDGE, CUT, SWAP, UNCOVER and RECOVER. The DELETEEDGE operation takes $O(1)$ time if an edge e with $l(e) = i$ is only stored in G_i. The amortized cost of CUT, SWAP and UNCOVER is $O(\log^3 n)$. Finally, RECOVER may be called $O(\log n)$ times with an amortized cost of $O(\log^3 n + t \log^3 n)$, where t denotes the number of edges whose level was increased. Because each edge can only be increased $\lceil \log n \rceil$

times, at most $O(\log^4 n)$ time can be spent increasing a single edge between its insertions and deletion. Thus, if INSERT pays for $O(\log^4 n)$ work up front, the edge increases can be paid off separately. Thus, the RECOVER calls contribute at most $O(\log^4 n)$ to the actual runtime of DELETE. Thus, the amortized cost of DELETE is $O(\log^4 n)$.

The actual runtime of INSERT is $O(\log^3 n)$ as a call is made to either COVER or LINK. Thus, the advanced payment for level increases is the dominant factor of INSERT. Thus, an amortized cost of $O(\log^4 n)$ is achieved for INSERT.

The TWOEDGECONNECTIVITY query has an amortized runtime of $O(\log^3 n)$ due to the EXPOSE and DEEXPOSE calls. □

4.1.4 Space analysis

Theorem 4.7. *The space usage of the 2-edge connectivity graph is* $O(m + n \log^2 n)$

Proof. Consider at first the space used by the top tree. Each node may store at most $O(\log^2 n)$ counters, with $O(n)$ nodes. Therefore, the top tree uses $O(n \log^2 n)$ space.

Besides the top tree, the spanning tree and the graphs G_i must be stored. The spanning tree takes up $O(n)$ space, compared to the graphs whose total size is $O(m + n \log n)$. The space complexity is achieved by only storing each edge e in G_i for $i = l(e)$.

Finally, each vertex stores $O(\log n)$ counters for an additional $O(n \log n)$ counters. Thus, the total space usage for the 2-edge connectivity graph is $O(n \log^2 n + m + n \log n + n \log n + n) = O(m + n \log^2 n)$. □

4.1.5 Improvements

Shortly after the first publication, Thorup improved the runtime from $O(\log^4 n)$ to $O(\log^3 n \log \log n)$. This speedup is achieved by observing that it suffices to approximate the *size* and *incident* counters stored in each cluster. Each approximate could be stored in only $O(\log \log n)$ bits compared to $O(\log n)$ bits, which meant $\Omega(\log n / \log \log n)$ counters could fit into a single $\Omega(\log n)$ bit word. This reduced the cost of adding two vectors of approximate values from $O(\log n)$ to $O(\log \log n)$. This trick also reduced the space usage to $O(m + n \log n \log \log n)$ [19].

Holm et al. improved the space complexity to $O(m + n \log n)$ by using a more complicated COMBINE operation. Unfortunately, this improvement could not be combined with the improvement above. They briefly mention an improved query time of $O(\log n)$ as well[8].

Holm et al. improved the update time to $O(\log^2 n \log \log^2)$, along with a $O(\log n / \log \log n)$ query time. This is achieved through multiple tricks. They first show a new approach to obtain a $O(\log^3 n \log \log n)$ update time algorithm by splitting the top tree data structure into three, one for the cover level and one for the size. The final data structure was used to find candidates for replacement edges efficiently by introducing a new concept of labels. Each operation on these data structures costs at most $O(\log^2 \log \log n)$ with at most $O(\log n)$ repeated calls throughout a call to INSERT or DELETE. Essentially, this improved the worst-case situation of $O(\log n)$ calls to the RECOVER operation running in $O(\log^3 n)$ [10].

The update time was further improved by the same trick Thorup initially used, which gave the final update time of $O(\log^2 n \log \log^2 n)$. The application of this trick was mentioned as a major challenge in the development of this result.

Finally, by splitting the data structures, they also reduced the number of counters stored, providing an improvement over $O(\log^2 n)$ counters stored in the original. Along with this reduction in counters, they presented a general technique to reduce the space usage of top trees, which reduced the total space usage to $O(m + n)$.

5 Experiment

This section explores the runtime of the dynamic 2-edge connectivity implementation. The runtime is also compared when using the simple and advanced versions of EXPOSE. Finally, a brief discussion of when the dynamic algorithm may outperform the static version.

The static 2-edge connectivity algorithm, taken from GeekForGeeks, could not run on large graphs due to recursion depth issues in Java[16]. This means the static algorithm would fail on graphs with 5000 vertices and 25000 edges. The dynamic algorithm managed to run on larger graphs, but the runtime grew rather quickly. Thus, a direct comparison of runtimes has been omitted.

5.1 Experimental setup

The algorithms were implemented and tested using Java and tested on randomly generated undirected graphs. The edge generation was created such that no duplicate edges were added.

This simple graph generation strategy should give edges evenly spread among the vertices, which may not reflect all real-world usages. Thus, the experiments would benefit from running on real-world graphs such as a dynamic road network or social media connections.

The data fit in RAM for all experiments, which positively affects the runtimes. This was chosen to shift the dominating factors around, to make calculations the dominating factor instead of I/O delay.

All experiments were run on OpenJDK 11.0.10 on version 10.0.19045 of Windows 10 Home, with an Intel® Core™ i7-10750H @ 2.60GHz and 16 GB of 2933 MHz ram.

5.2 2-edge connectivity experiments

This section performs a series of experiments on the dynamic 2-edge connectivity algorithm. First, the average runtime for a sequence of INSERT and DELETE operations are considered. This is followed by an individual assessment of the average runtime for insertions, deletions and queries. Finally, a very crude lower bound is found for the required hidden constant in the O notation.

The 2-edge connectivity algorithm was tested on graphs of varying size, with the largest consisting of 25600 vertices and 102400. The first experiment is used to verify the $O(\log^4 n)$ amortized runtime for a sequence of INSERT and DELETE operations. The average runtime over a sequence of operations should trend towards $O(\log^4 n)$. The total runtime is divided by $t \log^4 n$ in Figure 32, where t denotes the total number of operations.

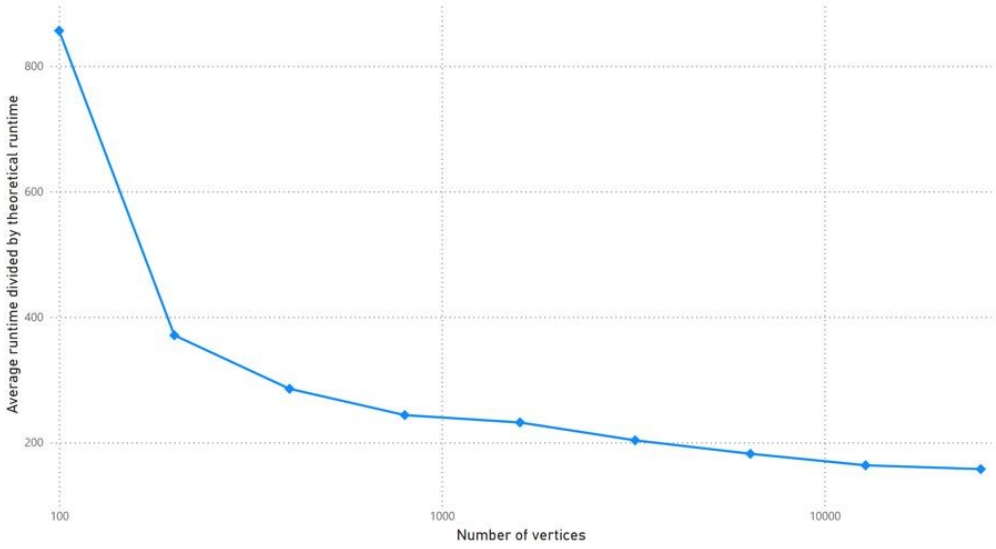


Figure 32: Average runtime divided by $O(\log^4 n)$ for sequence of $4n$ INSERTs, and n DELETEs

Importantly, the INSERT operations actual runtime is only $O(\log^3 n)$, which may skew the experiment, as some edges may not be increased the $\log n$ times which are paid upfront by the $O(\log^4 n)$ amortized cost of INSERT. Thus, consider the runtime of DELETE visualised in Figure 33. The runtime of DELETE also trends towards $O(\log^4 n)$. Theoretically, even if the level of all edges was increased $\log n$ times the RECOVER operations add at most $c \cdot 4n \log^4 n = O(n \log^4 n)$ work split over the n DELETE operations.

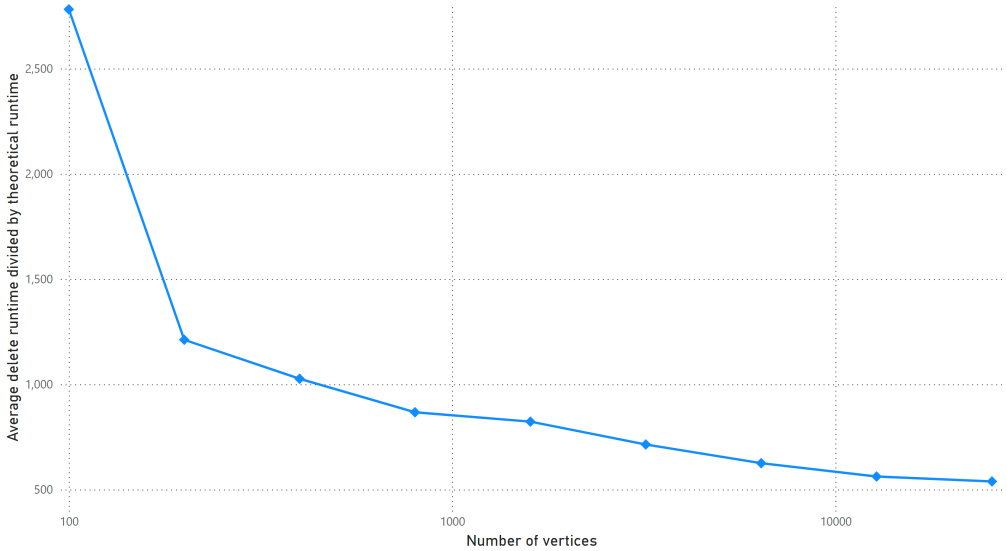


Figure 33: Average runtime of DELETE divided by $O(\log^4 n)$ for sequence of n DELETEs

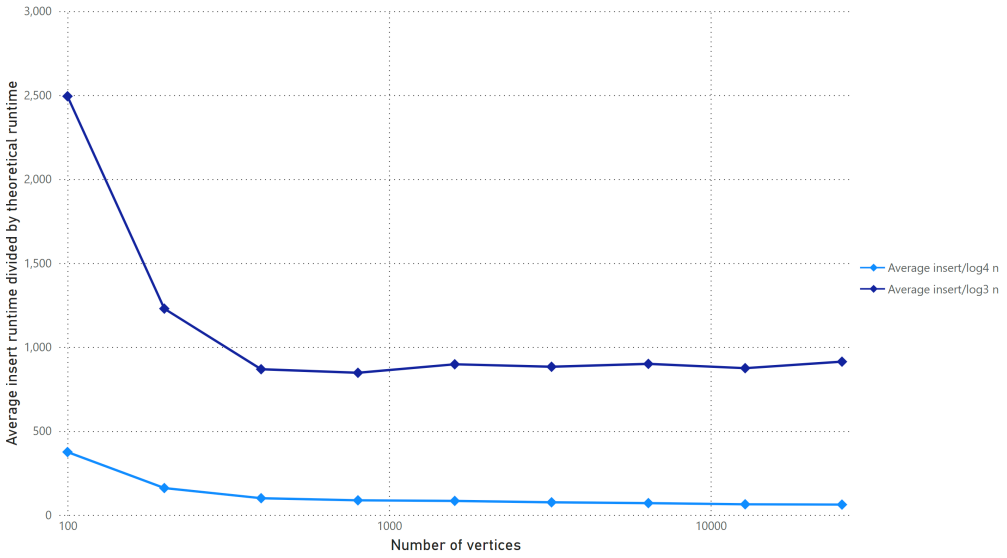


Figure 34: Average runtime of INSERT divided by $\log^3 n$ and $\log^4 n$ respectively for sequence of $4n$ INSERTs

Consider the runtime of INSERT individually as well. The actual runtime of the operation is $O(\log^3 n)$, while the amortized cost is $O(\log^4 n)$. The average runtime divided by both has been plotted in Figure 34, where both trend towards a constant. The difference between the constants is $\log n$, which for the values of n considered here is in the range of $\approx [7, 15]$.

Lastly, the average runtime of n random queries in the dynamic 2-edge connected data structure trends towards the expected $O(\log^3 n)$ as visualised in Figure 35.

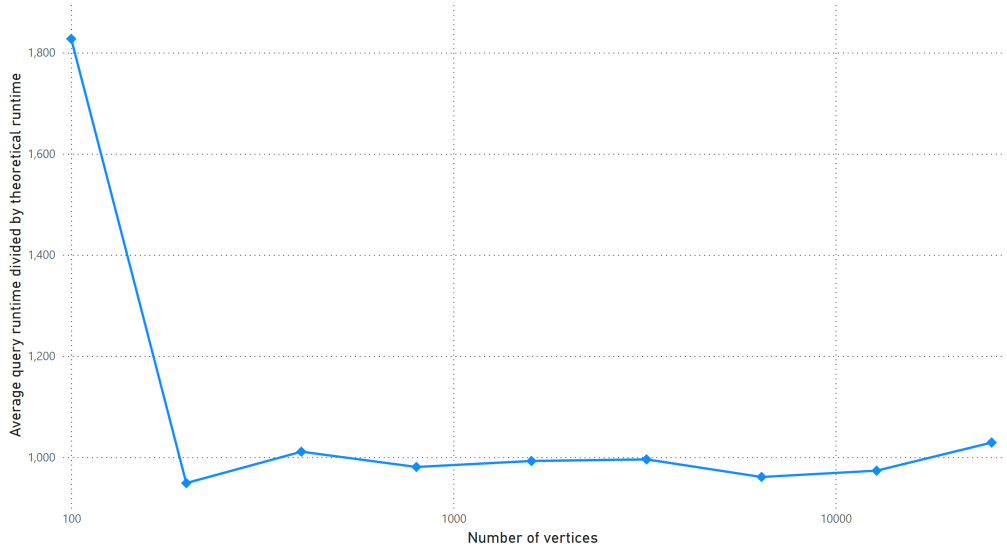


Figure 35: Average runtime of queries divided by $\log^3 n$ for a sequence of n

The average runtimes of the operations indicate that the amortized analysis is correct. However, it does not state anything about the worst-case performance of a single operation. Therefore, consider a graph with 1600 vertices and the runtime of 6400 insertions visualised in Figure 36. The early insertions are performed on mostly empty trees. Thus, their runtimes are quicker compared to those occurring later.

A few insertions run considerably slower compared to the rest, likely due to loads of clusters being recomputed throughout the EXPOSE calls made in INSERT, as it was consistent over multiple runs on the same graph, which was checked to ensure it was not due to background interference.

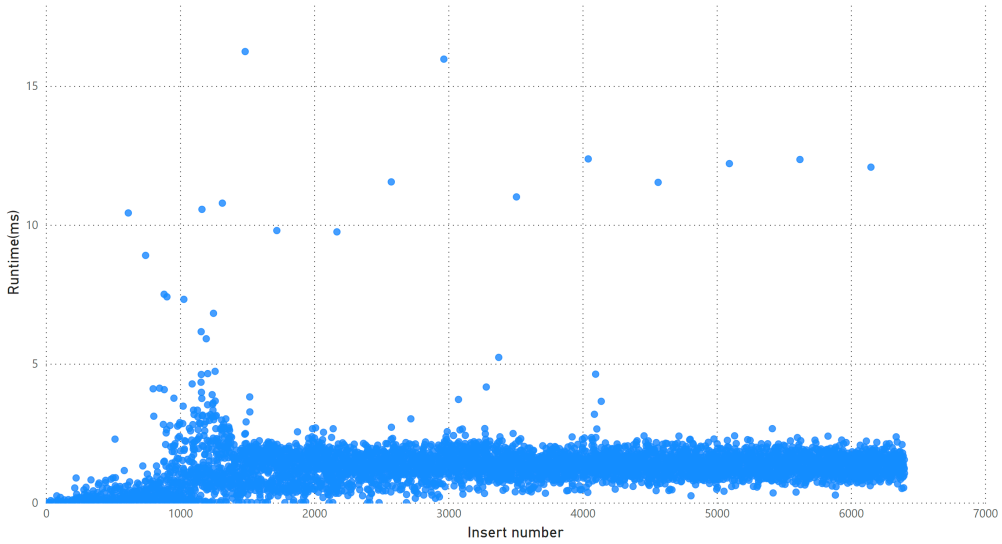


Figure 36: Runtime (ms) of the INSERT operation

When observing the runtime of the DELETE operation, the runtime of the first deletions stands out in Figure 37. They are substantially slower than the later deletions caused by the RECOVER operation. When the first edge is deleted, all edges e has $l(e) = 0$, increasing many edges to the next level. The runtime seems to stabilize quickly, supported by theory, as the edges can only be increased at most $\log n$ times. Furthermore, as edges are spread across different levels, increasing them without breaking the graph invariant becomes increasingly hard.

Theoretically, the slow DELETE operation can happen multiple times throughout the lifetime of a top tree. If most edges are deleted, followed by insertions of new edges, a similar state can be achieved where nearly all edges have a low level. However, it is also possible if a large batch of new edges were inserted that the next RECOVER call for level 0 find loads of edges whose endpoints belong to the same 2-edge connected component of G_1. However, it is rather unlikely that the RECOVER algorithm doesn't stumble upon an edge which would increase the size of the 2-edge connected component breaking the invariant. The final claim is supported by Figure 38, where 6400 edges were inserted after the 100th delete without a huge increase in runtime for the following deletes.

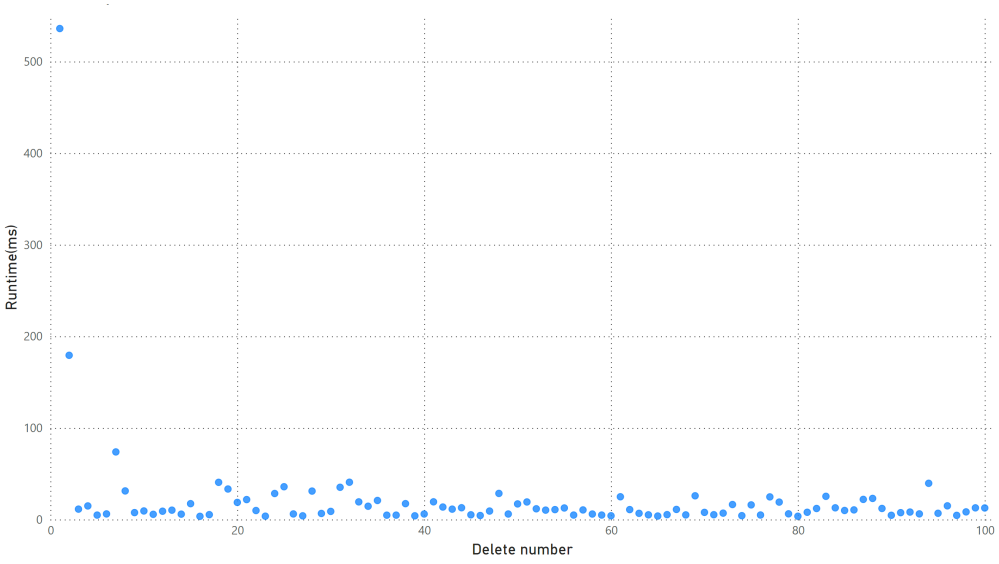


Figure 37: Runtime (ms) of the DELETE operation

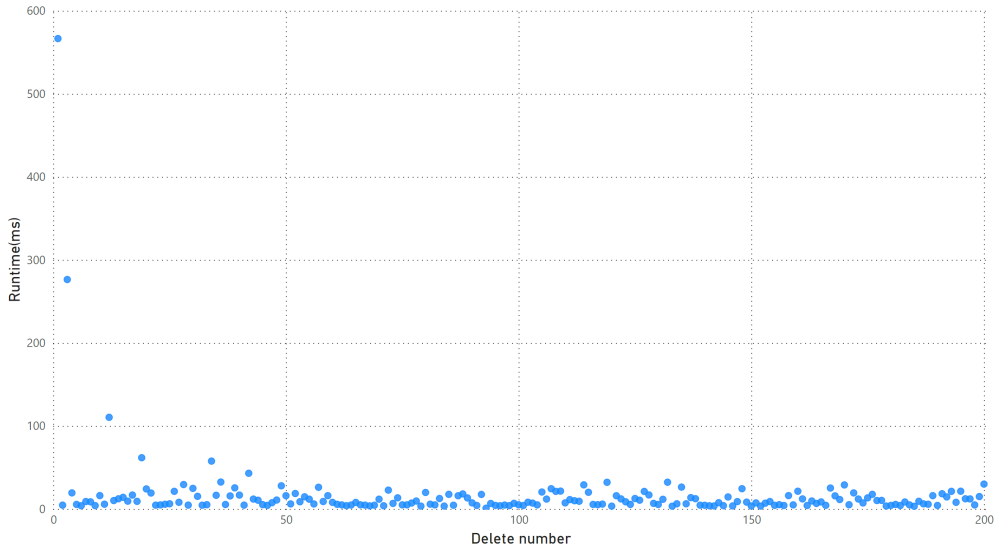


Figure 38: Runtime (ms) of the DELETE operation. 6400 insertions → 100 deletions → 6400 insertions → 100 deletions

One could consider a sequence of operations in an attempt to find a crude lower bound on the constant required for the $O(\log^4 n)$ amortized runtime of INSERT and DELETE of the 2-edge connectivity algorithm. The goal would be finding a constant c such that $i \cdot \log^4 n \cdot c$ is greater than the accumulative runtime t_i of the sequence after i operations. The constant can be computed as the $c \geq \max_i \frac{t_i}{i \log^4 n}$. Unfortunately, a different sequence of operations could impact the constant. Thus, this is only provide a slight insight into which operations are expensive.

Consider a sequence of 6400 insertions followed by 6400 deletions. The constant required for the INSERT operation seems to stabilise just short of 100, as seen in Figure 39. Unfortunately, the DELETE operations require a higher constant, making it rise quickly. This is then followed by a steady increase, which peaks at ≈ 350. The required constant then starts dropping, which happens as the DELETE operations become quicker when the spanning tree becomes disconnected as < 1600 edges are left in the graph.

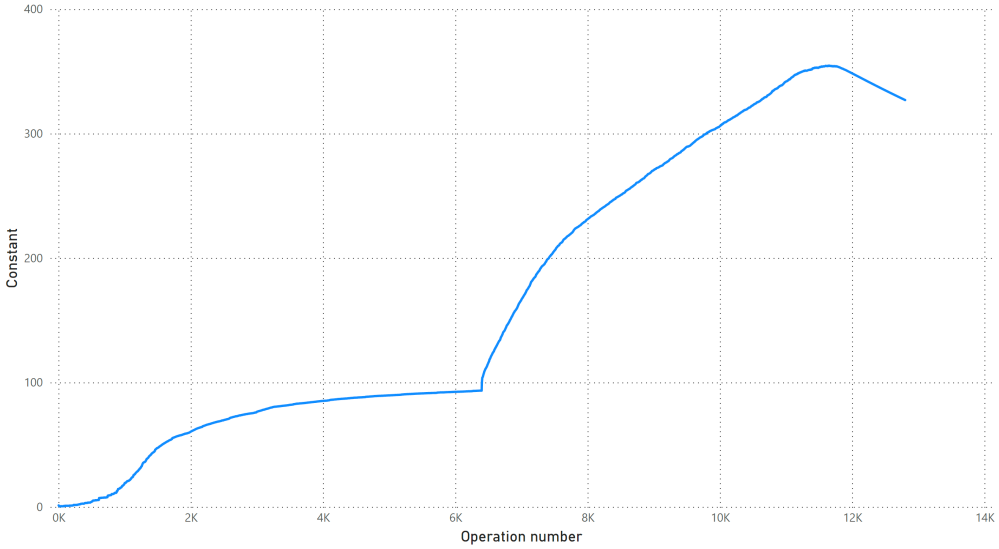


Figure 39: Lower bound for c in dynamic 2-edge connectivity, using 1600 vertices with 6400 insertions followed by 6400 deletions

The analysis made in Section 3 and Section 4 provides no constant as O-notation is heavily used to simplify expressions. Even if the analysis provided a constant for the lower bound, it would hardly be comparable to the measured constant, which is based on runtime. Thus, the result of this experiment is very unreliable, as no comparison can be made with the theory. Furthermore, both the implementation and language used can heavily impact the runtime. Thus, this is only a very crude lower bound.

Holm et al. present two different EXPOSE implementations[9]. The simpler version considered in Section 3.8.1 uses FULLSPLAY, with the more advanced version omitting this call. They conjectured the advanced version to be a significant speed in practice. However, as observed in Figure 40, the constants required in $O(\log^4 n)$ are very similar for a sequence of INSERT and DELETE.

This similarity was also observed when comparing the average runtime of the operations for different graph sizes as visualised in Figure 41.

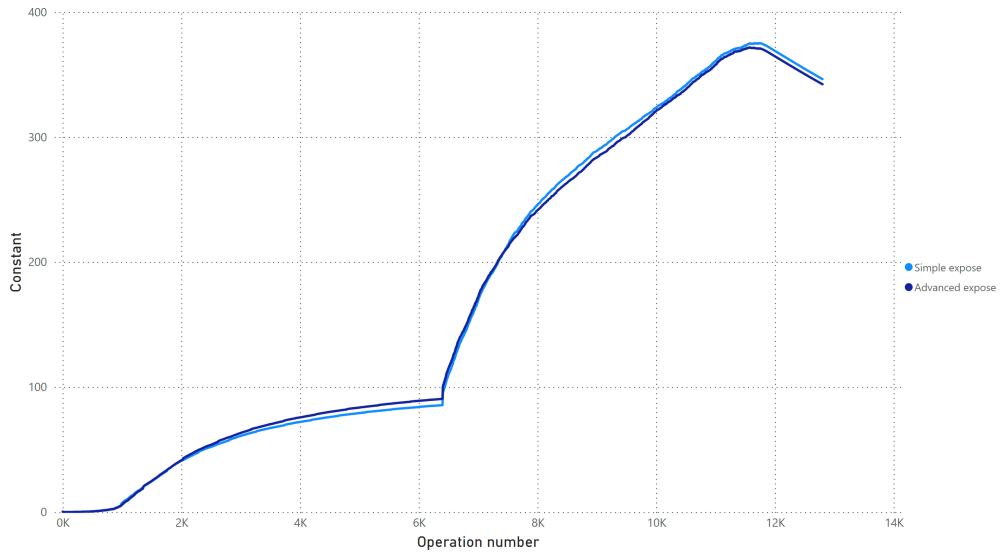


Figure 40: Comparison of EXPOSE operations for a graph with 1600 vertices. 6400 insertions followed by 6400 deletions

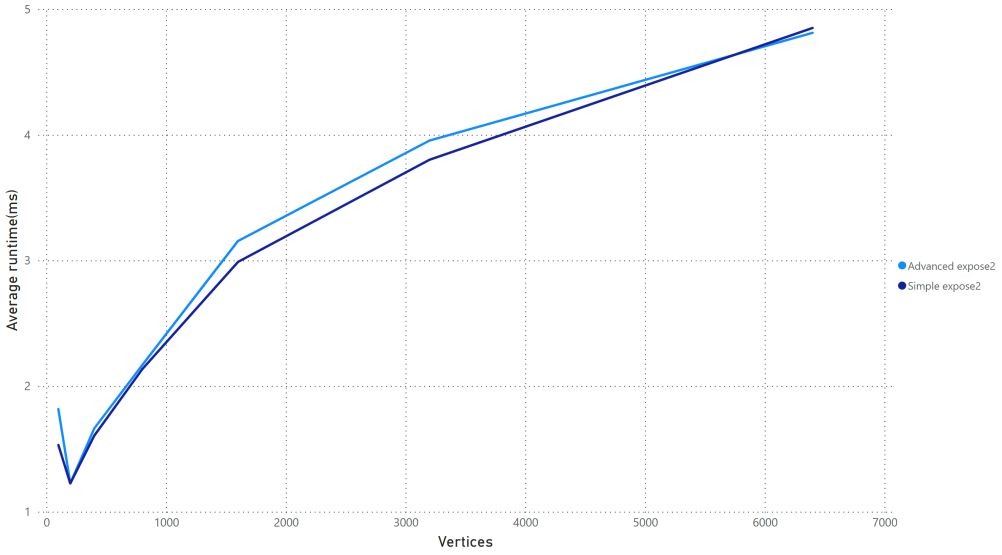


Figure 41: Comparison of EXPOSE operations for graphs with varying size. $4n$ insertions followed by n deletions

Even if a comparison between the performance of the static and dynamic 2-edge connectivity algorithms were performed, the results would be highly situational. The static algorithm proposed by Tarjan has a theoretical query runtime of $O(m+n)$, with $O(1)$ update time[18]. This is compared to the dynamic algorithm's $O(\log^4 n)$ update times and $O(\log^3 n)$ query time. Consider a graph with loads of edge insertions. As they are inserted, the query time of the static

algorithm becomes slower, whereas the query time of the dynamic algorithm is unaffected. The runtime of the entire sequence for the dynamic algorithm is instead affected by the slow edge insertions. As the algorithms are impacted differently, it becomes hard to say one solution is generally preferred over the other. For example, consider the static query run on a graph with $n = 1000$ and $m = 4000$; theoretically, this should take the same time as $n = 2000$ and $m = 3000$. Now consider the same graphs for the dynamic algorithm, where both the update and query time of these graphs differs. Here it is extremely unlikely that the runtime of the entire sequence takes the same amount of time, as the theoretical runtime of insertions only is $4000(\log^4 1000) \approx 40000$ and $3000(\log^4 2000) \approx 33000$ when ignoring the constants, which would require a rather specific number of queries for these graphs to perform equivalently over the entire sequence. Thus, even if the dynamic solution experimentally outperformed the static solution for a sequence of operations performed on a graph, generalising it is hard as even other sequences on the same graph may lead to another result. A query-heavy sequence of operations is more likely to run faster on the dynamic 2-edge connectivity than a sequence of mostly insertions and deletions.

The above considerations also make a theoretical comparison hard. Even though it may be possible to estimate how often queries have to be run in a sequence of operations for a graph of a given size, it is hardly useful as it requires a pretty specific use case.

6 Conclusion

In this thesis, we wanted to study data structures and algorithms for dynamic graph problems. We looked at the theory behind splay top trees, first by considering splay trees to gain familiarity with the concepts of splaying and amortized analysis. Splay top trees were then analysed, incorporating the user-defined operations COMBINE and SPLIT since they are central to all dynamic graph algorithms using top trees. This lead to an amortized cost of $O(\log n(COST(\text{COMBINE}) + COST(\text{SPLIT}))$ compared to $O(\log n)$ by Holm et al. for the interface operations EXPOSE, DEEXPOSE, CUT and LINK.

Splay top trees were implemented using Java, and the implementation's correctness was verified through assertions and experiments. A simple incremental minimum spanning tree was implemented and checked against Kruskal, verifying the correct result was computed every time. No experimental evaluation was made to compare the incremental minimum spanning tree algorithm's performance to Kruskal.

A dynamic 2-edge connectivity algorithm by Holm et al. was chosen as a more advanced algorithm using top trees, as it required both COMBINE and SPLIT, compared to the incremental minimum spanning tree, which only used COMBINE. The 2-edge connectivity algorithm was verified against a static version to ensure correctness. Unfortunately, the off-the-shelf static algorithm could not run on larger graphs making it impossible to get any relevant runtime comparisons. Thus, only the theoretical runtimes of 2-edge connectivity were tested. The experiments supported the theoretical runtimes of $O(\log^4 n)$ for updates and $O(\log^3 n)$ for queries. The experiments are only run on randomly generated graphs. Ideally, this was expanded to real use cases like road networks or social media connections. Holm et al. proposed two different EXPOSE operations, conjecturing one of them to be a significant speed-up in practice [9]. This was not supported by the experiments where both versions had nearly identical runtimes for the 2-edge connected algorithm.

6.1 Future work

A direct continuation of this thesis would be to expand the experiments to include real use cases with a comparison between the static and dynamic algorithms. The comparisons could also be made for other dynamic algorithms using top trees, such as Bi-connectivity, closely related to the 2-edge connectivity algorithm. Ideally, these comparisons would be run on optimised implementations written in a low-level language such as C or C++. The current implementation may suffer from early sub-optimal decisions, which were not rectified throughout the optimisations ahead of the experimental process.

Holm et al. improved the 2-edge connectivity algorithm by using degree d top trees, which could be another continuation of this thesis[10]. To my knowledge, splay top trees have not been modified to support degree d internal nodes. If generalising splay top trees is possible, it may require a significant amount of work, i.e. the orientation invariant is heavily dependent on only two children per node. The same goes for internal operations such as ROTATEUP operation.

The generalisation of splay top trees would ideally end with implementing degree d splay top trees and the improved 2-edge connectivity algorithm.

A small survey was made on dynamic graph algorithms as part of the early stages of planning and preparing for this project. The goal for this survey was to both gain an overview of current results and to determine a specific topic to explore in further detail, hopefully, find something interesting to write about. Shortly before the official start of the thesis, I became aware of the newly published paper on splay top trees by Holm et al.[9], which became the focus. The unfinished survey is included in Section 6.1, and a continuation of this project could be researching the extent top trees are used in state-of-the-art graph algorithms.

Lastly, it would be interesting to explore if quantum algorithms can achieve any significant theoretical improvements for dynamic graph algorithms. When performing a brief survey of the field, I was unable to find any results on quantum algorithms for dynamic graph problems. However, some static graph problems have been improved using quantum algorithms, e.g. computing a minimum spanning tree in $O(\sqrt{nm})$ using quantum computing, which is an improvement over the randomised classical algorithm, which runs in $O(n)$[4, 13]. However, in recent times the interest in quantum computing has increased. Thus, the likelihood of discovering new quantum graph algorithms has also increased.

References

[1] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. doi:10.1145/1103963.1103966.

[2] Aashish Barnwal and GeeksforGeeks. Kruskal's minimum spanning tree (mst) algorithm, 2023. visited on 2023-06-11. URL: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/.

[3] Gunnar Brinkmann, Jan Degraer, and Karel De Loof. Rehabilitation of an unloved child: semi-splaying. *Softw. Pract. Exp.*, 39(1):33–45, 2009. doi:10.1002/spe.886.

[4] Christoph Dürr, Mark Heiligman, Peter Høyer, and Mehdi Mhalla. Quantum query complexity of some graph problems. *SIAM J. Comput.*, 35(6):1310–1328, 2006. doi:10.1137/050644719.

[5] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Fully-dynamic all-pairs shortest paths: Improved worst-case time and space bounds. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2562–2574. SIAM, 2020. doi:10.1137/1.9781611975994.156.

[6] Jacob Holm and Kristian de Lichtenberg. Top-trees and dynamic graph algorithms. Master's thesis, University of Copenhagen, 1998. URL: https://di.ku.dk/forskning/Publikationer/tekniske_rapporter/tekniske-rapporter-1998/98-17.pdf.

[7] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 79–89. ACM, 1998. doi:10.1145/276698.276715.

[8] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

[9] Jacob Holm, Eva Rotenberg, and Alice Ryhl. Splay top trees. In Telikepalli Kavitha and Kurt Mehlhorn, editors, *2023 Symposium on Simplicity in Algorithms, SOSA 2023, Florence, Italy, January 23-25, 2023*, pages 305–331. SIAM, 2023. doi:10.1137/1.9781611977585.ch28.

[10] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 35–52. SIAM, 2018. doi:10.1137/1.9781611975031.3.

[11] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, volume 9294 of *Lecture Notes in Computer Science*, pages 742–753. Springer, 2015. doi:10.1007/978-3-662-48350-3_62.

[12] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In Philip N. Klein, editor,

Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, pages 510–520. SIAM, 2017. doi:10.1137/1.9781611974782.32.

[13] David R. Karger, Philip N. Klein, and Robert Endre Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995. doi:10.1145/201019.201022.

[14] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 950–961. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.92.

[15] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008. doi:10.1137/060650271.

[16] Seth and GeeksforGeeks. Check if a given graph is 2-edge connected or not, 2023. visited on 2023-06-11. URL: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/.

[17] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.

[18] Robert Endre Tarjan. A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2(6):160–161, 1974. doi:10.1016/0020-0190(74)90003-9.

[19] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 343–350. ACM, 2000. doi:10.1145/335305.335345.

[20] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Encyclopedia of Algorithms*, pages 738–741. 2016. doi:10.1007/978-1-4939-2864-4_569.

[21] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143. ACM, 2017. doi:10.1145/3055399.3055415.

List of Figures

List of Tables

List of Algorithms

Appendix

Appendix A: Symbols and Dictionary

Notation/Symbol	Description
$G(V, E)$	Used to describe a graph with vertex set V and edge set E
n	Number of vertices in the spanning tree/graph
m	Number of edges in the graph
Nodes	Used to describe elements in the top tree
Vertex/Vertices	Used to describe elements in the spanning tree
Tree edges	Used to describe edges in the spanning tree
Non-tree edges	Used to describe edges in the graph
$p(u)$ or $parent(u)$	Used to describe parent node of u
$sibling(u)$	Used to describe the sibling of u
ϕ	The potential function
$s(u)$	Used to describe the size or number of leaves in the subtree rooted in u
$r(u)$	$r(u) = \log_2(s(u))$ used in the potential function of both splay trees and the splay top tree
Cluster C	A cluster is a connected set of edges in the underlying spanning tree/forest with at most 2 boundary vertices. The cluster represented in the top tree as nodes. Leaf nodes represent clusters of size 1, while internal nodes represent the union of its children.
Point cluster	A cluster with 0 or 1 boundary vertices
Path cluster	A cluster with 2 boundary vertices
$\pi(C)$	Used to describe the path between the boundary vertices of C in the top tree
Exposed vertex	A vertex can be marked as exposed through the top tree interface, when done it is considered boundary for all clusters it is in
Boundary vertices	A vertex is a boundary vertex for a cluster if it is incident to something outside the cluster or it is exposed
T	This notation is used to refer to a spanning tree or splay tree
T_v	is used to refer to the spanning tree or splay tree containing vertex v
$\mathcal{T}$	This notation is used for top trees
$\mathcal{T}_v$	is used to refer to a top tree with vertex v
$boundary(u)$	The set of boundary vertices of u
Left/Right boundary vertex	These boundary vertices come exclusively from the left or right child
Central vertex	The vertex shared between the children of an internal node
Middle boundary vertex	Only the central node can be a middle boundary vertex.
Leftmost boundary vertex	The leftmost boundary vertex is the left boundary vertex if it exists, otherwise, it is the middle boundary. If neither exists, the leftmost boundary vertex is undefined.
Rightmost boundary vertex	See definition above.
Orientation invariant	*For any internal node C, the leftmost boundary vertex of the right child B and the rightmost boundary vertex of the left child A must both exist and be equal to the central vertex of C*[9]

Table 7: Notation/Symbol with descriptions

Appendix B: Code

The code is accessible in the following GitHub repository.
https://github.com/Andr9172/au-masters

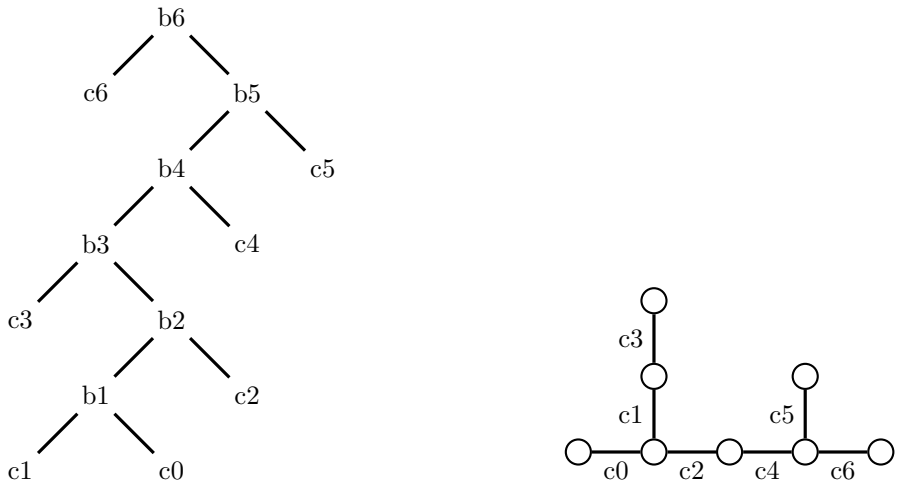
Appendix C: Graph algorithms survey table

Directed graphs	Update	Query	Space
Reachability Deterministic [15]	Amortized: $O(m\sqrt{n})$	Worst case: $O(\sqrt{n}) = O(n^{1/2})$	?
Reachability Probabilistic[15]	Amortized: $O(m^{0.58}n)$	Worst case: $O(m^{0.43})$	?
APSP uni-cost Deterministic [5]	Worst case: $O(n^{2+2/3}(\log n)^{2/3})$	?	$\tilde{O}(n^2)$
APSP weighted distance Deterministic [5]	Worst case: $O(n^{2+3/4}(\log n)^{2/3})$	?	$\tilde{O}(n^2)$
APSP uni-cost Probabilistic[1] [5]	Worst case: $O(n^{2+1/2}(\log n)^3)$	?	$\tilde{O}(n^2)$
APSP weighted distance Probabilistic[1] [5]	Worst case: $O(n^{2+2/3}(\log n)^3)$	?	$\tilde{O}(n^2)$
General undirected Graph	Update	Query	Space
Randomized Connnectivity [12]	Expected Amortized: $O(\log n(\log\log n)^2)$	$\log(n)/\log\log\log(n)$	?
Deterministic Connnectivity	Amortized [20]$O(\log^2 n/\log\log n)$	$\log(n)/\log\log(n)$	?
Minimum spanning trees/forest randomized [11]	Amortized: $O(\log^4 n/\log\log n)$	?	?
Minimum spanning trees/forest Deterministic [11]	Amortized: $O(\frac{\log^4 n \log\log\log n}{\log\log n})$	?	?
Minimum spanning trees/forest randomized [2] [21]	worst case: $O(n^{1/2-c})$	worst case: $O(1)$	?
Minimum spanning trees/forest randomized[2] [14]	worst case: $O(n^{O(1)})$[3]	?	?

[1] the ranndomized APSP is Las vegas style, against a non-oblivious adversary
[2] Las vegas style
[3] $O(1)$ hides a $O(\log\log\log n/\log\log n)$ factor

Table 8: Fully dynamic algorithms

Appendix D: Top tree and Spanning tree



(a) Top tree (b) Spanning tree

Figure 42: Top tree and the spanning tree