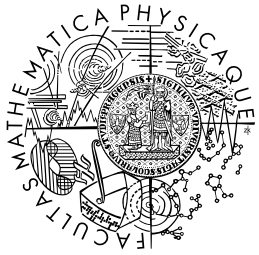

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jiří Setnička

Comparison of Top trees implementations

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Vladan Majerech, Dr.
Study programme:	Computer Science
Study branch:	Discrete Models and Algorithms

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Comparison of Top trees implementations

Author: Jiří Setnička

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Vladan Majerech, Dr., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Definition and description of Top trees and introduction of problems solvable by them including problem of edge 2-connectivity. Definition and description of Topology trees used as one of the drivers for Top trees. After the initial descriptions the two top trees implementations are introduced: one based on self adjusting trees, second based on topology trees. Comparison of these implementations is done by two experiments. Measurements are discussed in conclusion – results corresponds with initial estimates but with different multiplicative constant than expected.

Keywords: Top Trees, Complexity, Implementation

I would like to thank my supervisor Vladan Majerech for bringing such interesting topic, although the topic proved to be more difficult and more challenging than I thought. Also I would like to thank my family and friends for patience with me during writing this thesis.

Contents

Introduction

Main aim of this thesis is to provide two different *Top Trees* implementations and to compare them in different situations. Both implementation were written from scratch in C++ to provide comparable results.

Top Trees are not so well known data structure which could be used to maintain information of some dynamically updated collection of trees. User of this data structure defines four basic operations, which are used internally when Top Trees structure is changing. When there occurs some cutting or joining on underlying trees the structure updates internally stored information using these user functions.

This data structure could be used for example to dynamically maintain diameter, center or median (minimizing weighted distance from all other vertices) of given tree in time $\mathcal{O}(\log N)$ (where N denotes the number of vertices).

Because it is essential to understand how the Top Trees structure works, some basic principles of the Top Trees structure are introduced in the Chapter 1 and some basic principles of Topology trees used in one of the implementations are introduced in the Chapter 2. Some examples of problems, which could Top Trees handle quickly, are listed in Chapter 3 of this thesis.

Basic usage of both implementations and some technical details are the contents of the Chapter 4. Chapter 5 and Chapter 6 describes details of both implementations.

First implementation of the Top Trees structure is based on article *Self-Adjusting Top Trees* [1] by Tarjan and Werneck. This implementation promises quick amortized time per operation (with small constant), but it does not guarantee these times in worst case. This implementation is described in Chapter 5 of this thesis.

Second implementation is based on article *Maintaining Information in Fully-Dynamic Trees with Top Trees* [2] by Alstrup, Holm, Lichtenberg and Thorup and uses Topology trees introduced by Frederickson in [3]. This implementation promises time $\mathcal{O}(\log N)$ in worst-case but with much larger multiplicative constant. This implementation is described in Chapter 6 of this thesis.

To compare both implementations it was necessary to perform some experiments on different problems on different graphs with different sizes. Experiments were performed on problem of *maximum edge weight in tree with interval updates* (described in section 3.2) and on problem of *edge 2-connectivity* (described in section 3.3. Details of these experiments and their setup are described in Chapter 7.

We expected that the first implementations would have smaller multiplicative constant than the second one. This expectation turned out to be right and multiplicative constant for both implementations was measured in Chapter 8 together with some results for turning out unnecessary updates during some operation in the second implementation.

1. Top Trees

Top Trees are data structure intended to maintain informations of underlying dynamically updated forest. They were introduced by Alstrup, Holm, Lichtenberg and Thorup in *Minimizing Diamaters of Dynamic Trees* [4] in 1997 as variant of the Topology Trees and they were extended by the same authors in *Maintaining Information in Fully-Dynamic Trees with Top Trees* [2] in 2003.

1.1 Definition

Top Trees structure acts as driver for underlying forest. It represents underlying trees as collection of generalized edges called *clusters*. Each *Cluster* represents some subtree in the underlying forest. Only some of them called *root clusters* (which represents whole trees of the underlying forest) could be directly accessed by the user.

User defines format of the data stored in these clusters and four basic *user functions* CREATE, DESTROY, JOIN and SPLIT used to manipulate with clusters data. Above that user could define fifth function CHOOSE which is needed for non-local search but it is not needed for basic usage.

Then user controls the Top Trees structure by using *operations* CUT(u, v), LINK(u, v) and EXPOSE(u, v). Last of them makes cluster representing the path between vertices u and v a root cluster (because root clusters are the only clusters of the top tree, which could be accessed by the user). The Top Trees structure dynamically updates stored data in clusters by using user defined functions.

Notation: We will use capitalize form to denote situations where we refer directly to the defined user functions or to the top trees operations called by users. When referring to the generic process of joining, splitting or to the internal procedures related to these processes we will use normal font style.

1.2 Clusters

As has been said *Clusters* are generalized edges. Each cluster has two *boundary vertices* and represents part of the underlying forest between these vertices. We denote two clusters as *connected* if they are edge disjoint and they share one boundary vertex.

A *Clusterization* is division of the underlying forest into clusters such that each edge is in exactly one cluster. As we mentioned above *roots clusters* are the ones that represent whole trees in the underlying forest (all edges of their underlying tree are contracted inside and there are no outgoing edges – this means that in a clusterization both their boundary vertices are not connected with any other cluster).

Another special clusters are *leaf clusters*. We denote a cluster as a *leaf cluster* in some clusterization if only one of its boundary vertices is connected to another cluster.

Clusters in the Top Trees structure are organized into binary trees (called *top*

trees) where each leaf represents one edge of the underlying forest and each inner vertex represents contraction of its children. More about this structure will be discussed later in the *Cluster model* subsection. Before that we need to introduce types of clusters. There are three types of clusters:

- **Base cluster** – represents one edge of the underlying forest (and each edge of the underlying forest has exactly one base cluster, it is 1:1 mapping), boundary vertices are endpoints of the edge. This cluster could appear only as leaf in the Top Trees structure.

- **Rake cluster** – represents one way how to contract two clusters with one common boundary vertex. Let's have two clusters $C_1(u, v)$ and $C_2(v, w)$ next to each other around common boundary vertex v (and let the C_1 be the left one of them in some topological order given for example by indices of the edges or by some planar embedding).

  If the left cluster (C_1) is a leaf cluster then we can construct *left rake cluster* by *raking* the left cluster (C_1) on the right one (C_2). The resulting cluster would have the same boundary vertices as the cluster C_2.

  If the right cluster (C_2) is a leaf cluster then we can, similarly to the previous case, construct *right rake cluster* by *raking* the right cluster (C_2) on the left one (C_1). The resulting cluster would have the same boundary vertices as the cluster C_1.

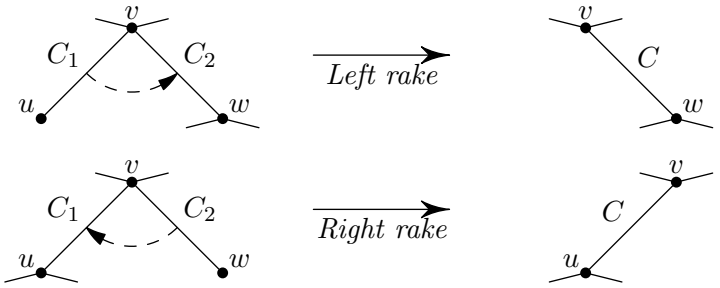

Figure 1.1: Left and right rake clusters

- **Compress cluster** – represents other contraction of the two clusters with one common boundary vertex v into one cluster by attaching first cluster after the other. Right before compressing the common vertex v must have degree (number of incident clusters) exactly two. If there are other clusters attached to the same common boundary vertex they must be firstly *raked* onto one of the compressed clusters.

  If boundary vertices of the cluster C_1 were (u, v) and boundary vertices of the cluster C_2 were (v, w), the cluster $C = compress(C_1, C_2)$ would have boundary vertices (u, w) (and we will call it *compress cluster of vertex* v and the operation *compressing around vertex* v). This cluster also in some way represents the vertex v and we will use this cluster as *handle* of the v.

Figure 1.2: Compress cluster

1.2.1 Clusters model

Clusters in the Top Trees structure are organized into binary trees. Leaves of these trees (Base clusters) represent edges of the underlying trees and each inner vertex represents contraction of two child clusters into one.

Compress and rake clusters have each of them two children, base clusters are childless. Each cluster represent subtree of the underlying forest. By sequence of clusters contractions we could represent each underlying tree as one *root cluster*. This whole binary tree of cluster contractions leading to the one root cluster is called *top tree*.

Compress clusters are used to represent paths in the underlying tree – each path could be compressed into one *compress tree* consisting only of compress clusters. If there are branches separating from this path, they are firstly recursively represented as single clusters (*rake trees*) and then they are *raked onto* clusters in the path.

Because there are M base clusters for an underlying tree with M edges and each inner vertex of the corresponding top tree joins two adjacent clusters into one, there will be $M - 1$ inner clusters for representing this underlying tree.

Underlying tree could have (and usually have) many different divisions into paths and so the underlying tree have many different representations. Crucial part of the top trees structure is to maintain this representation in some nice form during updates.

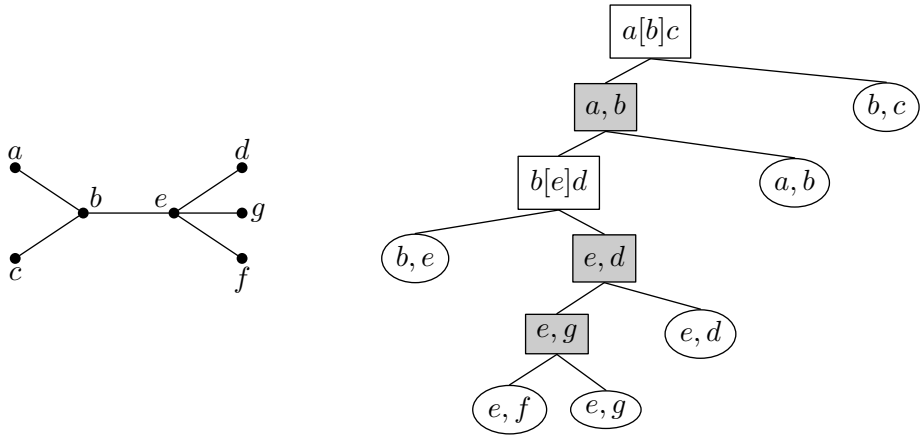

Figure 1.3: Original tree and corresponding top tree (rake clusters are grey)

1.2.2 Extended clusters model

Tarjan and Werneck in [1] suggested that in some cases it may be useful to modify structure of the clusters and they introduced *foster children* for *compress clusters*. In their suggestion a compress cluster could have up to four descendants – two normal children and up to two foster children.

Normal children of a compress cluster are clusters from the compressed path and foster children are clusters originating from the separating branches. In normal cluster model they would be raked onto clusters from path and the path would be compression of these rake clusters.

In this extended model the clusters originating from the separating branches

are firstly combined in *rake trees* – there are maximally two rake trees around each path vertex, one of them is raked from branches on one side of the path and the second one is raked from branches on the other side of the path. And these rake trees are connected as left and right foster child of the compress cluster constructed from this part of the path.

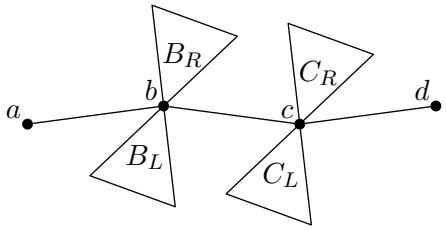

Figure 1.4: Rake trees (triangles) around a path, they can be connected as foster children to compress clusters

During computation (Join and Split operations) there is need to use virtual rake clusters, but it takes only $\mathcal{O}(1)$ time per one compress cluster. We will discuss it later in the first implementation for which this extended model is used.

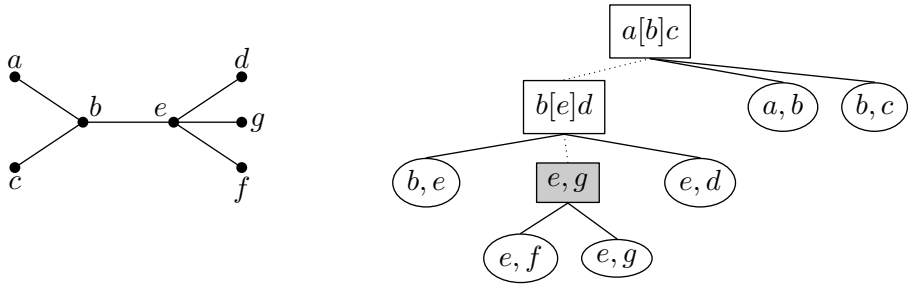

Figure 1.5: Original tree and corresponding top tree with extended clusters model (foster children are connected by dotted edges)

1.3 User defined functions

There are four basic functions to manipulate the clusters data which have to be implemented by user of the Top Trees structure. Then user uses public Top Trees structure operations and these user functions are used internally when constructing, destroying or reorganizing clusters.

If user wants to use the Search operation, he has to implement fifth user function Choose to traverse around the path by choosing children.

Examples of the functions and related problems are given in the Chapter 3.

1.3.1 Create

This function is called when new base cluster is created. It gets reference to the underlying edge and to the newly created base cluster, populates base cluster's data based on the underlying edge and runs other user defined operations according to logic of given problem.

1.3.2 Destroy

Opposite of the Create function. This function is called just before deleting base cluster. It gets reference to the underlying edge and to the base cluster which would be destroyed and it could perform some end-of-life operations (like saving computed data from the cluster).

1.3.3 Join

This function is called during contraction of two clusters into one (compress or rake) cluster. In the general view it should populate parent cluster with the data aggregated from contracted child clusters or perform other join-related operations according to logic of given problem.

It gets references to both of the contracted clusters and to the newly created parent cluster with information about their boundary vertices. From boundary vertices of the parent and both children can be clearly determined if it is compress or rake cluster (and which one of the children in rake cluster is raked onto the another one) – user may or may not use this information according to the logic of given problem.

1.3.4 Split

Opposite of the Join function. It is called just before removing connection among parent cluster and its children. This function gets references to the parent cluster and both of its children with information about their boundary vertices. It should distribute data from the parent into children – notice that no data could be stored in the parent cluster after the Split operation (because the parent cluster will be deleted after this operation).

The Join and Split functions are frequently called during reorganization of the Top trees structure – common pattern is to split everything around changed path in the top-down manner, reorganize the structure and then join everything in the bottom-up manner.

1.3.5 Choose

This operation for given root cluster selects one of its child clusters. It gets reference to the cluster and its children and returns reference to one of them. It is used internally by the Search operation.

1.4 Top Trees operations

These are the only operations which could user use to manipulate the Top Trees structure. In addition to that, user could access root clusters and read informations from them.

Normalized shape

Depending on implementation there could be defined a normalized shape of the Top Trees structure. All operations expect the Top Trees structure in this normalized shape.

Some operations may corrupt this normalized shape and in that case the correct shape must be restored prior to the next operation. For both following implementations an example of such operation is the EXPOSE operation (see implementation details).

We cannot restore the correct shape right after finishing the operation because user may need to interact with the Top Trees structure in this corrupted state. Therefore we have to record that the structure is in corrupted state and we have to do check and eventually restoration at beginning of each operation. See the following *Restore* operation for more details.

Handles

Following operations are defined for pair of vertices of the underlying forest, but the Top Trees structure operates on (generalized) edges. We need to map these vertices to clusters.

We want to choose clusters whose in some way represent operations with vertices. Every cluster represents some path and for given vertex we want to choose cluster which has this vertex in its path. Also we want that the chosen cluster could be easily transformed around this vertex. This means that we want cluster that has chosen vertex as its boundary vertex or common vertex (for compress clusters).

To accomplish this mapping we define *handle* for each vertex of the underlying forest in this way:

- Isolated vertex has no handle.
- If the vertex is a leaf of the underlying tree the handle for this vertex is the topmost compress (or base) cluster having this vertex as one of its boundary vertices (rake clusters cannot be handles).
- Root cluster is handle for its boundary vertices regardless of their degree in the underlying tree.
- And finally if the vertex has degree at least two the compress cluster of this vertex (compress cluster having this vertex as the common boundary vertex) is the handle of this vertex.

One node could be handle for at most three vertices – two as endpoints and one as common boundary vertex. To mark handle of a vertex v we will use notation N_v.

With handles we could transform operations with vertices into operations with clusters.

1.4.1 Expose

Expose is one of the most basic operations. Calling Expose(u, v) will result in exposing the $u \dots v$ path (if exists) in the root cluster, which the user could modify.

Implementation of the expose slightly differs in the first and the second implementation (first implementation uses splays and splices and the second one does expose through several splits and joins), but result of both is the same root cluster (but with possibly different decomposition to subclusters).

See details of both implementations for more information.

1.4.2 Restore

As we mentioned before we may need to keep the Top Trees structure in some normalized shape. Purpose of this operation is to restore the structure to such form. Because some operation may corrupt this normalized shape we call the Restore operation at beginning of all other operations.

To allow the user to modify user functions (Split, Join, ...) for each operation we will make the Restore callable by the user independently.

In one of the experiments we will use this functionality to turn off expensive updates during the Expose operation for implementation with topology trees. After finishing Expose we will Restore the structure back to the correct form and before next operation we will turn the expensive updates back on.

1.4.3 Cut

Operation Cut(u, v) deletes edge between vertices u and v and reorganizes the Top Trees structure to reflect this change. Precondition for this operation is that $u \neq v$ and there exists edge (u, v).

Both implementations use different approaches, but the result is the same – they remove the (u, v) edge and return roots of two new top trees.

1.4.4 Link

The Link operation is an opposite to the Cut operation. Calling Link(u, v) on two disconnected vertices joins them by the new edge (u, v). Precondition for this operation is that u and v are disconnected.

Both implementations use different approaches, but the result is the same – both top trees are joined by the new edge (u, v) and the cluster of resulting top tree is returned.

1.4.5 Search

When defined the Choose user function this operation could be used to find and return specific base cluster. Search guided by Choose functions splits clusters on the way from given root cluster to an edge cluster that was in all clusters chosen by Choose (similar to binary search on a normal tree).

2. Topology Trees

The Topology Trees data structure introduced by Frederickson [3] is used as the basic building block in the second implementation. We devote this chapter to basic understanding of how this data structure works and what must be done to use it in our case.

Basic idea of the Topology Trees is to divide a tree (or in general a forest) with maximal degree of three into recursive *clusters*. Trees with higher degree has to be firstly *ternarized* – their vertices have to be splitted. We will discuss the ternarization later, let's suppose we already have ternarized tree.

A collection of rooted binary trees (called *topology trees*) is built from these recursive clusters, one for each tree in the original forest.

Cluster in topology tree is a set of at most two nodes (vertices of the original tree or other topology tree clusters) with edge between them (but without edges to neighbors) and with at most three neighbors, better definition will conclude below. Nodes not connected by edge cannot be in common cluster, thus clusters represents some contracted subtrees of the original tree.

2.1 User interaction

User of the topology trees could interact with them by using Cut and Link operations like in the Top trees structure, but there is no Expose operation. To use it as base for the Top trees structure we will have to implement Expose differently.

Also note that clusters in topology trees are slightly different than clusters in top trees. In top trees clusters are generalized edges (with two endpoints) representing contracted subtree between two vertices, but clusters in topology trees are generalized vertices (with at most three outgoing edges).

We will build the Top trees structure based on topology trees in the Chapter 6, now let's introduce details of topology trees.

Notation: Like in the first chapter we will use capitalize form of the Cut *and* Link *operations to denote that they are called directly by the user. Also we will use terms internal cut and internal link to denote internal operations that works on a ternarized tree.*

2.2 Definition and properties

2.2.1 Topology clusters and clusterization

A *cluster of order k* is a set of k nodes connected by edges (there must be a path from each to each). A *clusterization of order k* of graph is division of this graph into clusters that:

- Each cluster has order at most k.
- Each vertex of the original graph is contained in exactly one cluster.

The *Topology clusterization* of a tree (with maximal degree 3) is a clusterization of order 2 whose clusters (called *topology clusters*) must meet these conditions:

1. **Neighbors limit**: Every topology cluster has degree (number of outgoing edges to neighbors) at most 3.
2. **Simple crossroads**: Clusters with three neighbors must have order 1 (they consist of only one vertex).
3. **Minimality**: There is no cluster that could be merged with its neighbor without violation these rules.

Because of that there are only 8 types of valid topology clusters:

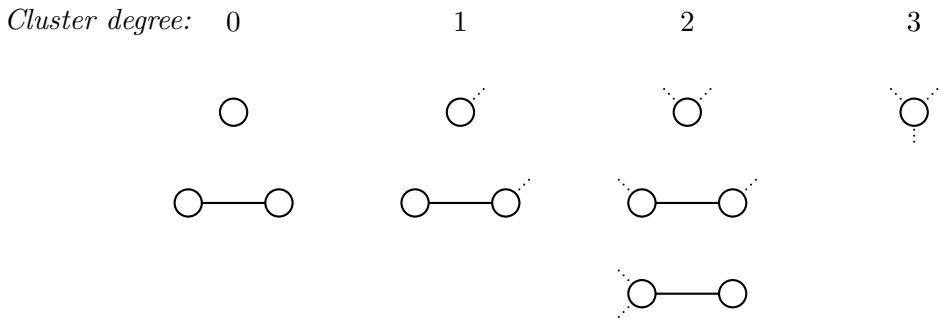

Figure 2.1: All 8 types of valid topology clusters

2.2.2 Topology tree

The *topology tree* is a rooted binary tree where each level of it represents some topology clusterization of the original tree. On the lowest level each of the original vertices is an independent basic cluster. These clusters are joined in the first level of topology clusterization, resulting clusters are contracted and acts as nodes for above level and so on.

Example of topology tree construction and yielding topology tree is on following figures 2.2 and 2.3.

Each inner cluster has at most two children on the level below (the clusters from which is it contracted) and at most one parent (topology clusters without parent are roots of topology trees). Among these tree edges each topology cluster is connected with its neighbors on the same level – each topology cluster has at most three outgoing edges.

2.2.3 Height of a topology tree

Frederickson in [3] proved that each level of topology clusterization has at most 5/6 clusters of the previous level. Number of clusters on the lowest level corresponds to the number of vertices of the original tree (denote it as N) and at k-th level there are at most $N \cdot (5/6)^k$ clusters. Therefore the height of the topology tree for tree with N vertices could be estimated as $\mathcal{O}(\log N)$.

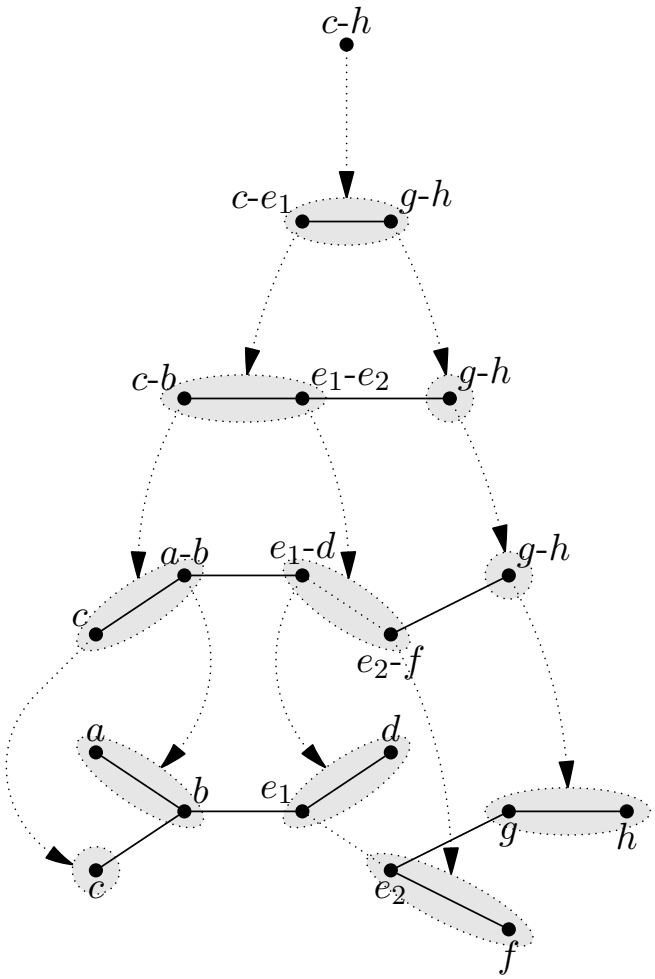

Figure 2.2: Example of the topology tree construction. On the bottom level there is original underlying tree with ternarized vertex e (ternarization has no effect on the construction, for more details see subsection Ternarization).

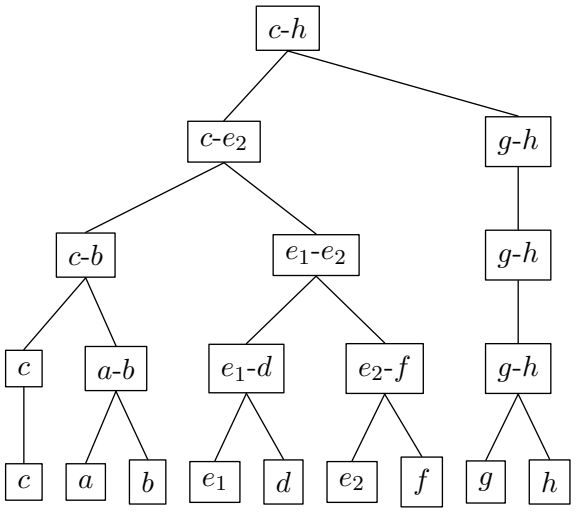

Figure 2.3: Resulting topology tree from the previous construction.

2.3 Updates – internal cuts and links

In this section we will assume that we work on a ternarized tree, generic Cut and Link operations will be discussed below in the ternarization section.

After a change in the underlying forest (adding or removing an edge) we must update the whole topology trees structure – after *link* (adding edge) two topology trees are joined into one and after *cut* (removing edge) one topology tree is splitted into two topology trees.

This change has to be propagated on all levels of a topology tree and corresponding topology clusters (and their neighbors) have to be updated. The main idea is that this operation should do constant work on each level which leads to the time $\mathcal{O}(\log N)$ for each operation.

Update process was originally described by Frederickson in [3] but my implementation is based on process described in Martin Mareš's master thesis [5] (in Czech). The process is the same but it was described more clearly in the second source.

2.3.1 Update process

Both link and cut starts by modifying the original forest which will break some constraints in the lowest level of our topology trees. Whole update process goes from the lowest level up and repairs broken constraints.

Update process work in phases (one phase for level in topology tree) and uses three lists listed below:

- **Delete list** – clusters to be deleted, initially it is empty.
- **Change list** – clusters which are changed and needs recomputation, initially it contains basic clusters at endpoints of the added/removed edge.
- **Abandon list** – clusters which are (for some reason) without parent and needs some, initially it is empty.

In each phase update process processes all clusters from all three lists and prepares lists of clusters from above level for the next phase. When there are no clusters to process it ends.

Firstly for all clusters from delete list: We delete this cluster (and disconnect all outer edges) and if it is the only child of its parent we add parent to the delete list for the next phase, otherwise we add the second child into current change list, because we need to recompute its neighbors (and that transitively ensures that parent would get into new change list).

Next for all clusters in change list: If they have sibling (the second child of their parent) and they are connected with him by edge then everything is correct – we only update their neighbors (based on its children neighbors) and add the parent into next change list (we need to update parent's neighbors too).

If cluster from change list has sibling but they are not connected by edge (for example the initial state after cut operation) their parent is no longer a valid cluster – therefore we add their parent into next delete list and move both clusters into abandon list (because they are without valid parent).

And finally for all other clusters (rest of change list and abandon list) we do this process:

- *When cluster has no neighbors we found a new root cluster* → we just save it into list of root clusters.
- *When cluster has three neighbors and one of the neighbors is single cluster (cluster with only one neighbor)* → we join them together under one parent, resulting cluster would have two neighbors. Depending on parents of both clusters there are several options:
  - If one of the clusters has parent, we reuse it and we add this parent into next changed list.
  - If both clusters have parent, we choose one, reuse it (adding it into next change list) and we add second one into next delete list.
  - If both clusters were without parent we have to create a new one and add this new parent into next abandon list.
- *When cluster has three neighbors but no neighbor is single cluster* → we cannot join cluster with any of its neighbors, therefore we only ensure there is parent of this cluster (if parent exists we add it into next change list, otherwise we create it and add it into next abandon list).
- *If cluster has degree (number of neighbors) at most 2 and it has neighbor without sibling such that degree of cluster plus degree of this neighbor is at most 4* → We join them together. The resulting parent cluster will have degree 2 (because we encapsulate the common edge, which we count twice, into parent cluster). Ensuring the parent cluster is similar as in the second case. If there is no such neighbor we only ensure that parent of this cluster exists (similar as in the third case).

This process takes $\mathcal{O}(1)$ per level and $\mathcal{O}(\log N)$ for the whole topology tree.

2.4 Ternarization of a tree

One of the things we have to deal with when using Topology trees as an executive layer for Top trees are degrees of vertices – the Top Trees structure have to work on trees of any degree but Topology trees work only on trees with max degree 3.

We have to *ternarize* each vertex – turning vertex with higher degree into chain of multiple vertices with degree 3 – and keep this ternarization during cuts and joins.

Let's have a vertex of degree $D \geq 4$. It could be splitted into chain of multiple vertices with maximal degree 3 by these simple steps:

- Create $D-2$ *subvertices* and set their *superior vertex* as the original vertex.
- Connect subvertices by edges (called *subvertice edges*) into one chain (first and last vertex will have one edge used, inner will have two edges used)

- Disconnect neighbors from the original vertex and connect them to these subvertices (there are exactly the right number of free neighbor slots).

Because the Cut and Link operations in the Top trees works with the original vertices, we have to build some mapping on internal Topology trees cuts and link operations.

2.4.1 Ternarization during Cut operation

Both endpoints of the Cut(u, v) could be processed independently. When cutting on vertex with degree at most 3 nothing had to be done and we may simply call internal cut operation.

When cutting on vertex which is splitted into several subvertices the operation is more difficult. Firstly we need to find right subvertex which incidents with the edge. This could be done easily by some list of pointers. After finding the right subvertex and edge we call the internal cut on this edge, but this operation decreases degree of this subvertex. According to situation we have to do several repair steps:

- *If this is inner subvertex of the subvertice chain:* We remove it from the chain, we need two internal cuts on edges to both neighbors and one internal link to directly reconnect neighbors.
- *If this is outer vertex of the subvertices chain and there is at least one inner vertex of the chain (when degree of the original vertex was at least 5):* We "steal" one outer edge from the neighbor (which is an inner vertex of the subvertice chain) by one internal cut and one internal link and then we continue as in the first case by removing this inner vertex.
- *If this is outer vertex of the subvertice chain and there is no inner vertex:* We have to join subvertices into the original vertex. We cut each edge to neighbor and link it back to the original vertex. Finally we cut the edge between both subvertices and delete these subvertices.

  Because degree of the original vertex is 3 after this operation we do exactly 4 internal cuts and 3 internal links.

In each case we have to do only constant number of internal cuts and links. All these inner operations work in the $\mathcal{O}(\log N)$ time so the time complexity of the whole Cut operation is $\mathcal{O}(\log N)$ (but the multiplicative constant could be really large).

2.4.2 Ternarization during Link operation

As in the Cut operation both endpoints of the Link(u, v) could be processed independently. We firstly need to ensure that both endpoints have degree at most 2. When linking such vertices we don't need to do anything and we just simply call internal link operation.

When linking vertex with degree 3 or more we need to split it into subvertices (or add a new subvertex to the existing subvertices chain).

When degree of the vertex was exactly 3, it is not splitted yet and we have to split it – as in the procedure above we create two subvertices, we link them by one edge and then for all existing neighbors we cut them from the original vertex and link them to one of the subvertices. One subvertex ends with degree only 2 and we use this subvertex as an endpoint for the main link operation.

Otherwise when vertex is already splitted into subvertices we have to create a new subvertex and insert it into the subvertices chain. Easiest is to do cut between the first and the second subvertex in the chain and then two links – between the first and the new subvertex and between the new subvertex and the second subvertex. Then we use the new subvertex as an endpoint for the main link operation

In each case we have to do only constant number of inner cuts and links. All these inner operations works in the $\mathcal{O}(\log N)$ time so the time complexity of the whole Link operation is $\mathcal{O}(\log N)$ (but as in the Cut's case the multiplicative constant could be really large).

3. Examples of problems and user functions

In this section there is a list of several problems which could be solved by Top Trees with small time complexity.

3.1 Finding distance between two vertices

This problem was originally solved in *A Data Structure for Dynamic Trees* [6] by Sleator and Tarjan in 1983 and then it was adapted for Top Trees by Alstrup, Holm, Lichtenberg and Thorup in [2].

Theorem: Lets have dynamic collection of weighted trees with link and cut operations. We could find length of the path between any two vertices (or find that they are not connected) in $\mathcal{O}(\log N)$ time.

Proof: We will maintain length of the cluster path in every cluster.

- CREATE creates cluster with length equivalent to the length of the underlying edge.
- JOIN of clusters C_1 and C_2 into C depends on the type of the C:
  - If C is a compress cluster of C_1 and C_2: Set length of the C's path as sum of the lengths of C_1 and C_2.
  - If C is a rake cluster and C_1 is raked onto C_2: Set length of the C's path as length of the C_2's path (and vice versa if C_2 is raked onto C_1).
- SPLIT and DESTROY does nothing.

After that we could easily get length of the (u, v)-path by calling EXPOSE(u, v) and reading length from this cluster. Because operations in user functions are in constant time and EXPOSE takes $\mathcal{O}(\log N)$ operations, we could answer on any such question in $\mathcal{O}(\log N)$ time.

3.2 Maximum edge weight between given vertices with interval update

Similarly to the previous problem this problem was originally solved in [6] and then it was adapted for Top Trees in [2].

Theorem: Let's have dynamic collection of weighted trees with operation of linking, cutting, updating edge weight and updating edge weights on given path. We could find maximum edge weight between any two vertices (or find that they are not connected) in $\mathcal{O}(\log N)$ time.

Proof: We can maintain w_{max} in each cluster as maximum weight on this cluster's path and w_{extra} as weight added to each edge on this path.

- CREATE creates cluster with $w_{max} = w(e)$ where e is the edge for which is this cluster created. There is no extra weight yet, so $w_{extra} = 0$.
- JOIN of clusters C_1 and C_2 into C depends on the type of the C:
  - If C is a compress cluster of C_1 and C_2: Set w_{max} as maximum of w_{max} from clusters C_1 and C_2. There is no extra weight, so $w_{extra} = 0$.
  - If C is a rake cluster and C_1 is raked onto C_2: Copy w_{max} from the C_2. There is no extra weigh, so $w_{extra} = 0$.
- SPLIT have to distribute w_{extra} to the children. For C_1, C_2, children of splitted C will operations depend on the type of the C:
  - If C is a compress cluster of C_1 and C_2:
    - $w_{extra}(C_i) = w_{extra}(C_i) + w_{extra}(C)$ for $i \in 1, 2$
    - $w_{max}(C_i) = w_{max}(C_i) + w_{extra}(C)$ for $i \in 1, 2$
  - If C is a rake cluster and C_1 is raked onto C_2: Apply above operation only for C_2 (and vice versa only for C_1 if C_2 is raked onto C_1).
- DESTROY sets weight of the underlying edge: $w(e) = w_{max} + w_{extra}$.

Then we could just call EXPOSE(u, v) and read w_{max} or add both to w_{extra} and w_{max} of the root cluster representing the (u, v)-path. Everything in time complexity of an EXPOSE operation, which is $\mathcal{O}(\log N)$.

3.3 Edge 2-connectivity

This is an example of more complex problem, where is the Top tree structure used as only part of the whole algorithm. It was introduced by Holm, Lichtenberg and Thorup in 2001 in [7]. A complete description of the problem is in the mentioned article, there we will only recall some basic principles (most of this section is a simplified citation from the [7]).

Theorem: Problem of 2-edge connectivity on dynamically updated graph with N edges could be solved in amortized time $\mathcal{O}(\log^4 N)$ per one operation (addition or deletion of an edge and test if given vertices are 2-edge connected).

3.3.1 Basic principle

We will maintain a spanning forest F of graph G and we will say, that tree edge e is *covered* by nontree edge v, w if $e \in v \dots w$ (if there is a tree path from v to w and e is on this path).

Frederickson showed that two vertices x and y are 2-edge connected if and only if they are connected in F and all edges in path $x \dots y$ are covered.

So we will maintain a spanning forest F together with a set C of covering edges for each non-bridge edge in F. Two vertices are 2-edge connected in G if and only if they are 2-edge connected in $F \cup C$.

During updates, when we will delete an edge from F we will need to find an replacement in C. And for every deleted edge from C we may need to add several

replacement edges into C. By carefully choosing replacement edges we will be able to amortize the time to meet $\mathcal{O}(\log^4 N)$ per operation.

3.3.2 Brief overview of details

We do not discuss all the details here, they are described in [7]. We only briefly overview some basic principles to make the big picture. For details see the mentioned article.

Algorithm associates with every tree edge a nontree edge that covers it. When we add u, v edge and u and v are already connected in the F we should add u, v edge as cover edge for all tree edges on the $u \dots v$ path. It would be slow to directly update all the affected edges so we will distribute this information with lazy updates through the top tree clusters – each cluster has its own cover information which it distributes to its path children when the cluster is splitted.

Deletion of an uncovered tree edge is quite easy, we just Cut it from the F. Otherwise we need to find some replacement edges for edges that stops to be covered. To accomplish easy finding of replacement edges the algorithm associates with every nontree edge a level (number $< \log N$) – each nontree edge starts with level 0 and levels may be only increased.

For each i we will define a graph G_i as graph induced by only edges of level at least i together with the F. For each tree edge $e \in F$ we will maintain a cover level as maximal level of nontree edge covering it (thus maximal level i for which e is in a 2-edge connected component of G_i). When some vertices are 2-edge connected on level i they 2-edge connected on all levels $\leq i$.

When updating the cover information after deleting some edge we use these levels as guide for finding new cover edges. Major part of the time complexity of the whole algorithm is the time needed for updating data structures with nontree egdes incident to some vertex on some level. These data structures are used for finding replacement edges. This part is described in detail in the [7].

3.3.3 Operations

User could control the algorithm with three simple operations:

- **Inserting an u, v edge** – When given vertices are not connected in the F just connect them by Link, otherwise add a nontree edge and do cover on the path cluster for path $u \dots v$.
- **Deleting an u, v edge** – When the u, v edge is a tree edge and it is a bridge (not covered by any other edge) just Cut it. If it is covered tree edge, swap it with its cover nontree edge and delete it like other nontree edges. Nontree edge is deleted firstly by *uncovering* $u \dots v$ path (removing all cover informations that it may generate on the $u \dots v$ path) and then by *recovering* this path (because by *uncovering* we may remove too much cover information).
- **Query** if u and v are 2-edge connected – This is an easy operation when we just need to Expose $u \dots v$ path and read the cover information from the returned root cluster.

3.3.4 Speed up query by disabling expensive updates

This idea came from the original article [7] where they analyzed time complexity and postpones some enhancements.

Covering and uncovering on graph with N edges takes $\mathcal{O}(\log^2 N)$ and therefore one call to the JOIN takes $\mathcal{O}(\log^2 N)$ and LINK, CUT and EXPOSE operations takes $\mathcal{O}(\log^3 N)$. Recover operations takes $\mathcal{O}(\log^3 N)$ for every increase of and edge level by one and because edge level is increased at most $\mathcal{O}(\log N)$ times, we spent at most $\mathcal{O}(\log^4 N)$ time for each edge between its insertion and deletion.

The query in this original algorithm takes $\mathcal{O}(\log^3 N)$, but it could be speed up if some conditions are met. During query we don't need any updated incident informations, we need only cover informations and these informations could be joined in $\mathcal{O}(1)$ time.

If the top trees implementation could preserve the original shape of the clusters and restore to this original form after EXPOSE, we may turn off expensive updates in the JOIN operation during the query and complete the query in $\mathcal{O}(\log N)$ time.

This condition is met for the second implementation using topology trees, but we cannot disable the updates in the first implementation, because the shape of the top tree is different before and after the EXPOSE and we would lost some incident information.

3.4 Vertex 2-connectivity

Vertex 2-connectivity problem is only a more complicated version of an edge 2-connectivity problem discussed in the previous subsection.

Solution using top trees was introduced by Holm, Lichtenberg and Thorup 2001 in [7] and this solution works in time $\mathcal{O}(\log^5 N)$ per operation. For more details see the mentioned article.

4. Implementation and usage

As has beeen mentioned in the Introduction both implementation are written in C++. More precisely they are written in C++14 with frequent use of smart pointers introduced in C++11 and enhanced in C++14, which helped a lot with memory handling.

Source code of both implementations is attached to this thesis including Makefile for easier compiling and testing. Source code is also published on Github, which may be more pleasant way to explore it or use it in other projects:

`https://github.com/setnicka/top-trees`

4.1 Interface of the Top Trees structure

Both implementations share the same interface which makes them easily interchangeable.

Firstly user needs to include `TopTreesInterface.hpp` and define classes for holding data in vertices and edges. These classes must inherit from generic classes for edge and vertex data defined in the `TopTree` namespace.

Then user has to choose one implementation and init the Top Tree. After that user could use Cut, Link and Expose methods to manipulate with the structure.

4.1.1 User data structures

If user want to store data in edges and vertices he has to define classes that inherits from default `TopTree::EdgeData` and `TopTree::VertexData` classes.

```
class MyEdgeData: public TopTree::EdgeData {
public:
    int weight;
    std::string label;
    virtual std::ostream& ToString(std::ostream& o) const {
        return o << label;
    }
};

class MyVertexData: public TopTree::VertexData {
public:
    std::string label;
    virtual std::ostream& ToString(std::ostream& o) const {
        return o << label;
    }
};
```

Figure 4.1: Example of classes for edge and vertex data

When using `DEBUG` options (see below) classes for edges and vertices must implement `std::ostream& ToString(std::ostream&)` method, it is used for debug printing. How to enable debug printing will be showed later.

As second step user needs to define structure for cluster data. This structure must inherit from generic structure `TopTree::ClusterData`. For initialization of this structure user has to provide function `TopTree::InitClusterdata` and function `TopTree::CopyClusterData` to do a deep copy of of given data to the another cluster. Example of all definitions follows.

```
struct MyData: public TopTree::ClusterData {
public:
    int max_weight;
};

std::shared_ptr<TopTree::ClusterData> TopTree::InitClusterData() {
    return std::make_shared<MyData>();
}

void TopTree::CopyClusterData(
    std::shared_ptr<ICluster> from,
    std::shared_ptr<ICluster> to
) {
    auto fromData = std::dynamic_pointer_cast<MyData>(from->data);
    auto toData = std::dynamic_pointer_cast<MyData>(to->data);

    toData->max_weight = fromData->max_weight;
}
```

Figure 4.2: Example of struct for holding cluster data

4.1.2 User functions

Necessary for operating with Top Trees are definitions of user functions. User must define user functions in the `TopTrees` namespace, without these functions the program cannot be even compiled.

Both JOIN and SPLIT user functions takes shared pointers to three clusters (generic interface `TopTree::ICluster`), two for child clusters and one for parent cluster. CREATE and DESTROY functions takes shared pointer to the created/destroyed cluster and shared pointer to the underlying edge's data (data is passed as generic type `TopTree::EdgeData` and it have to be casted to the user's data class as previously defined).

```
void TopTree::Join(
    std::shared_ptr<TopTree::ICluster> left,
    std::shared_ptr<TopTree::ICluster> right,
    std::shared_ptr<TopTree::ICluster> parent
) {
    auto l = std::dynamic_pointer_cast<MyData>(left->data);
    auto r = std::dynamic_pointer_cast<MyData>(right->data);
    auto p = std::dynamic_pointer_cast<MyData>(parent->data);
    p->max_weight = max(r->max_weight + l->max_weight);
}

void TopTree::Split(
    std::shared_ptr<TopTree::ICluster> left,
    std::shared_ptr<TopTree::ICluster> right,
    std::shared_ptr<TopTree::ICluster> parent
) {} // Nothing to do, but we must define the function

void TopTree::Create(
    std::shared_ptr<ICluster> cluster,
    std::shared_ptr<EdgeData> edge
) {
    auto data = std::dynamic_pointer_cast<MyData>(cluster->data);
    auto edge_data = std::dynamic_pointer_cast<MyEdgeData>(edge);
    data->w_max = edge_data->weight;
}

void TopTree::Destroy(
    std::shared_ptr<ICluster> cluster,
    std::shared_ptr<EdgeData> edge
) {} // Nothing to do, but we must define the function
```

Figure 4.3: Example of user functions

4.1.3 Choosing top tree implementation and initialization

Both top tree implementations share interface `TopTree::ITopTree` which defines `Expose`, `Cut`, `Link` and `Restore` functions.

To use one of the implementations just initialize it:

```
// For Sleator-Tarjan self-adjusting trees implementation:
auto TT = new TopTree::STTopTree();

// For topology trees implementation:
auto TT = new TopTree::TopologyTopTree();
```

Top tree could be initialized from scratch or it could be initialized from some underlying tree. For this underlying tree there exists `TopTree::BaseTree` class with `AddVertex`, `AddEdge` and `AddLeaf` functions (last one is only shortcut for `AddVertex` and `AddEdge` in one step).

To initialize from some underlying tree user could call `InitFromBaseTree`.

```
auto baseTree = std::make_shared<TopTree::BaseTree>();

auto a = baseTree->AddVertex(std::make_shared<MyVertexData>("a"));
auto b = baseTree->AddLeaf(a, std::make_shared<MyEdgeData>(15),
                           std::make_shared<MyVertexData>("b"));
auto c = baseTree->AddLeaf(a, std::make_shared<MyEdgeData>(3),
                           std::make_shared<MyVertexData>("c"));
auto d = baseTree->AddLeaf(b, std::make_shared<MyEdgeData>(7),
                           std::make_shared<MyVertexData>("d"));
auto e = baseTree->AddLeaf(b, std::make_shared<MyEdgeData>(4),
                           std::make_shared<MyVertexData>("e"));

auto TT = new TopTree::STTopTree();
TT->InitFromBaseTree(baseTree);
```

Figure 4.4: Example of initialization from underlying base tree

4.1.4 User methods

After initialization user could control the structure by Link, Cut and Expose methods:

- `Cut(v, w)` – Cut edge between given vertices (given by integers returned from `AddVertex` calls). It returns tuple with pointers on the root clusters of both resulting top trees and pointer to the edge data from cutted edge. When vertices are not connected by an edge (when not connected at all or when connected by some longer path) it will return `NULL`.
- `Link(v, w, edge_data)` – Link given vertices with a new edge with given data. When vertices are already in the same tree it will return `NULL` (otherwise it will return pointer on root cluster of resulting top tree).
- `Expose(v, w)` – Expose given $v \ldots w$ path and return pointer to the cluster covering this path. When vertices are not connected it will return `NULL`.
- `Restore()` – Restore the structure into valid form (useful only for Topology trees implementation when user wants to disable some expensive updates during Expose).

4.2 Debug and Graphviz output

Both implementations have debug output which can be used to trace their processing steps and optionally to output their current state in Graphviz format. Debug messages goes on their error output (stderr) and Graphviz on their standard output (stdout).

Graphviz is open source program which is able to visualize several types of graphs (tree structures included). As we mentioned above both implementations are able to print Graphviz script at their standard output and this script could be translated by Graphviz program to a PDF image.

Because of Graphviz problem with outputting multipage PDF at the time of writing this thesis better approach is to translate them to PostScript and to convert this PostScript to the PDF manually:

```
./top_trees > output.dot
dot -Tps2 output.dot -o output.ps
ps2ps output.ps output-fixed.ps
ps2pdf output-fixed.ps output.pdf
```

There is also Makefile rule so inside the project directory you can call only `make output.pdf` to convert `output.dot` file.

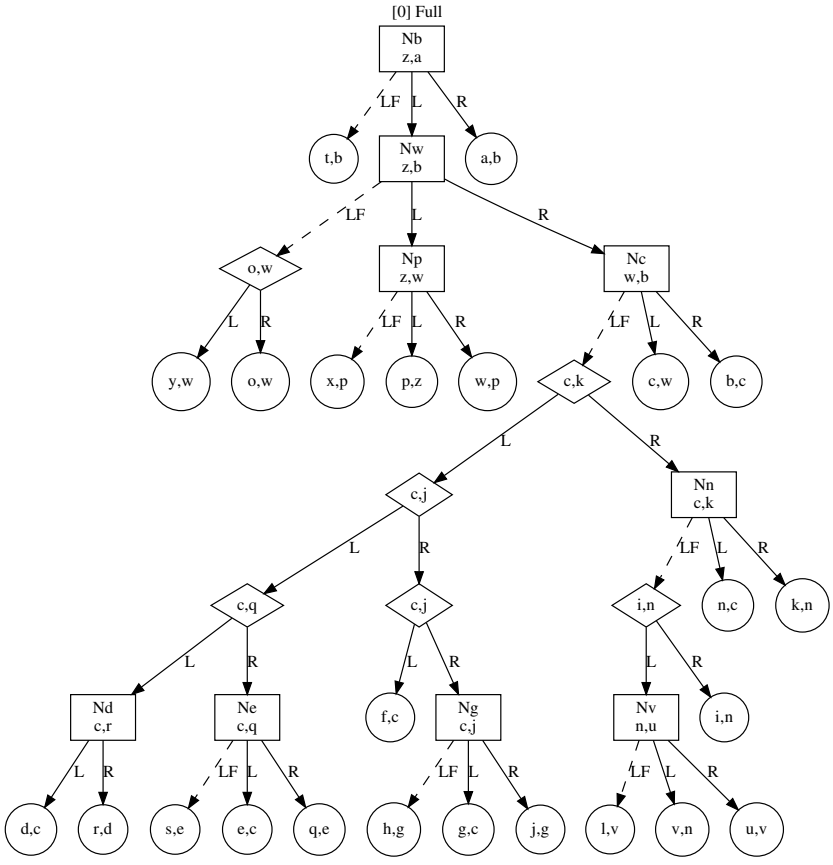

Figure 4.5: Example of Graphviz showing top tree from the first implementation

4.2.1 Enabling debug and Graphviz output

User only needs to uncomment lines at the top of `.cpp` files and compile the program again. Possible options: `#define DEBUG` for debug output on stderr, `#define DEBUG_GRAPHVIZ` for graphviz output on stdout and `#define WARNINGS` for warnings about incorrect usage like linking vertex to itself on stderr.

5. Implementation of Top Trees using self adjusting trees

This implementation is based on article *Self-Adjusting Top Trees* [1] by Tarjan and Werneck and uses the extended clusters model with foster children discussed in the Chapter 1.

5.1 Construction

Tarjan and Werneck in [1] suggested this construction:

1. Choose root r as a vertex with degree one.
2. Orient all edges in the tree containing vertex r towards the vertex r.
3. Divide tree into paths starting in some leaf and continuing along the direction of the edges – the first path will end in the root r and became the *root path*, other paths end up being connected to some existing path.
4. Recursively compute clusters to represent each path incident to the root path and create *rake trees* from these incident paths.
5. Create binary tree of compress clusters to represent the root path and connect rake trees as foster children.
6. If there are some unused vertices of degree one, start the process again from any of these vertices to construct another top tree.

In the implementation we choose equivalent construction but in more recursive manner. We started the same way by choosing the root r as vertex with degree one, but we don't divide the tree into paths.

Starting from the second vertex we choose one neighbor as continuation of the path and recursively called the same function on all other neighbors. Recursion returns clusters representing each of the subtrees and then they could be raked into left and right rake trees and saved into this vertex for future use.

When compressing the path into compress clusters we just look into the common vertex of compressed clusters and if there are saved rake trees we connect them as left and right foster children.

This construction is easier to implement and gives us ability to better control the shape of the resulting top tree. By choosing neighbors instead of directing paths from leafs we could prefer longer paths by choosing neighbors with deepest subtree (we firstly run DFS[1] to obtain depths). Longer paths are better contracted in binary tree structure of compress clusters to obtain lower top tree.

[1]Deep-first search – common known algorithm to search graph $G = (V, E)$ in time $\mathcal{O}(|V| + |E|)$

5.2 Expose

Expose in this implementation is based on splaying and splicing which are used to bring handles of given vertices to the top of theirs top trees.

Implementation of the EXPOSE operation has two parts: soft and hard expose. Soft expose is used internally by other operations, hard expose is used only in the EXPOSE itself.

Before moving forward we will recall some internal structure of top trees with extended clusters model.

Compress and rake trees

Because of extended clusters model each top tree consist from independent *compress trees* (only compress clusters as internal nodes) and *rake trees* (only rake clusters as internal nodes).

Whole top tree is one compress tree (which represents the *root path*). It has base clusters as leafs and roots of rake trees as foster children (these foster children are other paths connected to the root path). These rake trees have rake clusters as theirs internal nodes and base or compress clusters as theirs leaf. And so on.

This division of the top tree into smaller blocks could be used to expose given pair of vertices in the root of the top tree. We will use operations of *splay* and *splice* introduced by Sleator and Tarjan in [8].

Split and Join operations

Before doing any operation that changes shape of a top tree, all nodes involved in this operation must be splitted (including all their parents on the way to the root of this tree). This is crucial because after changing shape of the top tree a data stored in these nodes may be changed (for example depth of subtree bellow this node).

Split operations have to be done in top-down manner (starting from the root). The easiest way how to accomplish this is to have flag in each node if it is splitted and recursively split parent before splitting current node. All splitted nodes should be logged into some list to easily join all of them after completing current operation.

Joining is done in opposite direction, in bottom-up manner (ending in the root). We will assume that before doing anything with any node during splaying and splicing operations we firstly split this node and after completing the entire expose operation we will call join on all splitted nodes.

5.2.1 Splaying

Splaying is originally a heuristic for balancing binary trees which uses an idea that often used nodes should be near the root of the tree. Each operation (find, delete, ...) on a vertex in splay tree is preceded by splaying this vertex which moves this vertex to the root of the tree.

Splaying is done by rotations or double rotations (which are called *zig-zig* or

zig-zag rotations). That moves target vertex up by one or two levels preserving all vertices in the right order.

Although some not so often used vertices may be in $\mathcal{O}(n)$ distance from the root, Sleator and Tarjan in [8] proved that all operations work in amortized time $\mathcal{O}(\log n)$ per operation.

In the Top Trees structure we will use *guarded splays* that works exactly the same way as normal splays, but it stops splaying when they reach a guard (some node). Normal splay has as guard root of the whole tree, but we want to do splays limited only inside compress or rake tree (not to mix compress and rake clusters).

Implementation of the splay is straightforward. Only noticeable detail (which Tarjan and Werneck mentioned in the [1]) is that foster children are not affected by any rotations, they always keep the same parents.

5.2.2 Splicing

Only by splaying we are not able to expose a path containing both given vertices. And we are not even able to carry compress clusters over rake trees (because rake trees are connected as foster children under compress trees). For that we need to change partitioning of the top tree into paths.

Lets take some vertex v which is internal to some path $a \dots b$ (so there is compress cluster around this vertex) and which has node representing path $c \dots v$ in its rake tree. We may change partitioning into paths by removing one half of the original path (for example $v \dots b$), pushing cluster with this path into rake tree, removing node representing path $c \dots v$ from the rake tree and changing the compress cluster that it will represent path $a \dots c$.

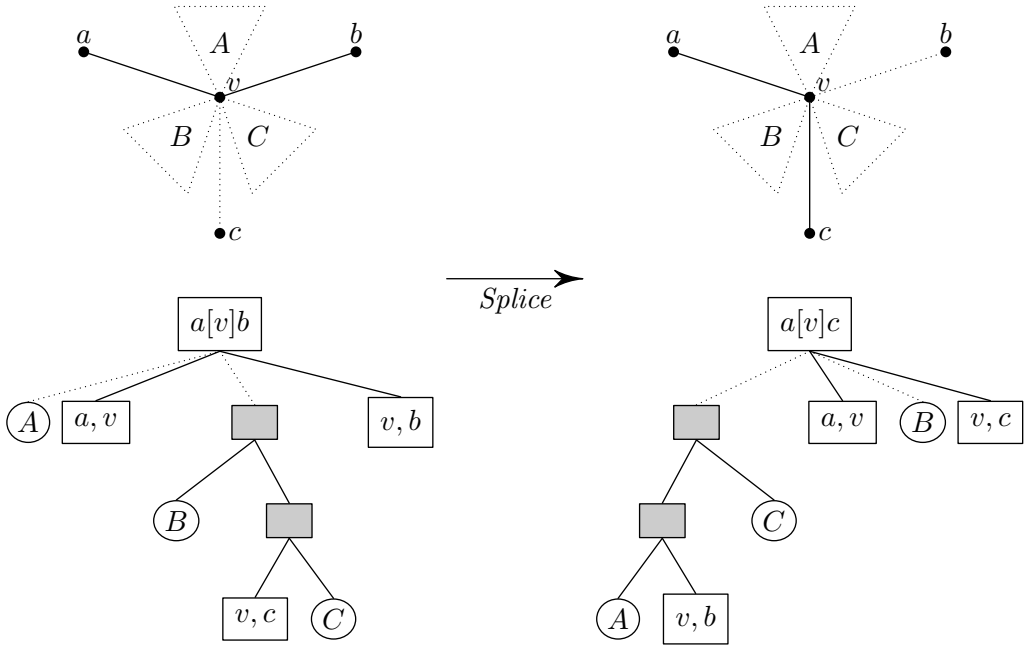

Figure 5.1: Splice around vertex v and changing $v \dots b$ for $v \dots c$ (grayed clusters are rake clusters).

Splicing is used after splaying and its task is to move cluster, which is leaf of some rake tree, to the compress tree above it.

Tarjan and Werneck in the [1] only described the main idea in the general case, but implementation details had to be worked out. In this work we decided to do *left splice* (replacing left child of the compress cluster), but the procedure would work the same way if we choose to replace the right child.

Our implementation does splicing in this way:

1. Prepare empty lists of left and right neighbors.
2. Starting from given node N go up until reaching node N_c in compress tree. During that add left and right neighbors into neighbors lists and destroy old internal nodes of these rake trees.
3. Add left child of the N_c to the appropriate list of neighbors and connect N as the new left child of the N_c (that makes N part of the above compress tree).
4. Construct new left and right foster children (rake trees) from left and right neighbors lists respecting their order. New internal rake nodes have to be constructed (and added into list of nodes for joining).

To assure that we do not broke connectivity of N_c with above clusters by replacing left child we need to check children of the N_c. If the above cluster (parent) is compress cluster, one of boundary vertices is common vertex of the parent. And if the parent is rake cluster, one of boundary vertices is the vertex around which is the parent raked. If this connection is in the left child we need to flip children (otherwise child with the connection would be moved into rake tree and connection would broke).

This detail was not discussed by Tarjan and Werneck – they only suggested transforming top trees into some normalized form which would work for both splaying and splicing. But they did not mention some corner cases where such checking and flipping is needed.

5.2.3 Soft expose

Soft expose is the first part of the EXPOSE operation. When it is called as *soft_expose(u,v)* its target is to bring path $u \dots v$ as subpath into the root cluster of corresponding top tree (so after soft expose there should be some path $a \dots u \dots v \dots b$ in the root cluster). Truncating the root cluster to contain only $u \dots v$ path is quest for the hard expose operation.

Soft expose takes handles of both vertices and brings them to the top of their top trees. If both vertices are in different components (they are not connected by a path) both the handles of u and v are brought to the roots of corresponding top trees using series of local changes in the top trees (similarly if $u = v$).

When they are in the same component (they are connected by a path) firstly the handle of u is brought to the root of corresponding top tree. If the current root cluster is also handle of v we are done, otherwise the handle of v is brought as close to the root as possible (but not replacing the handle of u as root).

Algorithm

Exact procedure for one vertex (as described by Tarjan and Werneck in [1]) is following:

1. **Local Splays** – Starting from the handle of given vertex:
   (a) Splay current node inside compress tree (that makes it root of that compress tree)
   (b) If reaching a root cluster (there is no parent) → stop the cycle.
   (c) If parent of the current node is rake cluster → set this parent as current node and splay it inside its rake tree.
   (d) Take parent of the current node (compress cluster having current node as foster child) and repeat the cycle.
2. **Splices** – Starting from the handle of given vertex splice current vertex, move to its parent and repeat. This procedure moves in every step handle of the given vertex across one rake tree to the above compress tree. Finally it moves this handle into the root compress tree.
3. **Global splay** – Perform splay on the handle of given vertex (now in the root compress tree) to make it root of this top tree.

Firstly we expose $handle(u)$ using above procedure. After that we expose $handle(v)$ with $handle(u)$ as the guard (to assure that it remains at the top). If both vertices (and so both handles) are in different top trees both of them ends as roots of theirs top trees – this situation is not interesting anymore, so we will assume that they are in the same top tree.

Tarjan and Werneck discussed special situation when one of the vertices has degree one (and its handle is not a compress cluster around this vertex), in this case they realized that mentioned procedure leads to exposing handle of the second vertex with first vertex as one of its endpoints.

Another special case, which they did not discussed, is situation when both vertices have degree one – in that case before starting procedure for the second vertex we have to ensure that the base cluster of the first vertex is not the left child of root cluster (otherwise it would be moved to rake tree when splicing the handle of the second vertex, what we do not want).

If handles for both vertices are the same there is nothing to do (both handles are occupied by the root cluster). When they are different the second handle will end as one of the first handle children. To make hard expose easier we will assume that this is the right children (otherwise we just flip left and right children).

And where is path $u \dots v$ located? There are three possibilities (notice that we are still operating with extended clusters model, where rake trees are connected as foster children – in standard top tree model there could be intermediate rake clusters and the cluster with path $u \dots v$ could be located deeper):

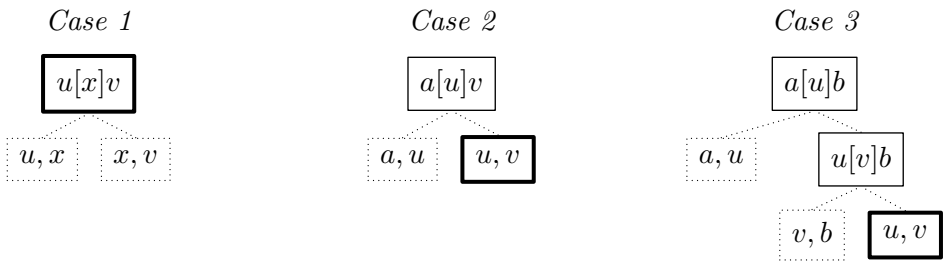

Figure 5.2: Three possible cases where cluster with endpoints u, v could be located after soft_expose(u,v).

1. Root cluster itself – When both vertices have degree one, they are endpoints of the root cluster and so their handles are represented in the root cluster.

2. Child of the root cluster – When one vertex has degree at least two its handle is compress cluster. In that case the root cluster represents some path $a \dots u \dots v$ or $u \dots v \dots b$ and desired path $u \dots v$ is represented in its child (assume that it is the right child, otherwise flip them).

   Another case when path is in child of the root cluster is when u and v are connected directly by an edge. In this case the path is represented by base cluster and so it might be child of some compress cluster.

3. Grandchild of the root cluster – When both vertices have degree at least two, both handles are represented by compress clusters. Child of the root cluster is bounded from one side (by common vertex of the root compress cluster) and it is again a compress cluster (the second handle). The grandchild of the root cluster is bounded from by the second cluster and so it represents our path (let again assume it is the right child, otherwise flip children).

To ensure that the root cluster represents only the path $u \dots v$ we need the hard expose operation.

5.2.4 Hard expose

Hard expose is the second part of the EXPOSE operation, it follows the soft expose and its job is to truncate the path in the root cluster that it contains only the exposed subpath.

Ideal situation is when u and v are endpoints of the root cluster, it is possible when both of them have degree one or when all other clusters incident to the root cluster are raked onto the root cluster.

But in general root cluster could represent some path $x \dots y$ with path $u \dots v$ as subpath. In this case we need to temporarily convert ends of this path (paths $x \dots u$ and $v \dots y$) into rake clusters so the compress tree would represent the path $u \dots v$ with these ends raked onto this path.

As we discussed at the end of the soft expose operation, wanted subpath may be located in the root cluster itself, in its (right) child or in its (rightmost) grandchild.

Tarjan and Werneck (in [1]) came with simple trick – temporarily convert compress clusters above wanted cluster (at most two compress clusters as we

realized above) to rake clusters (*rakerizing* them). Because we are using left rake clusters the resulting rake cluster will have the same boundaries as its right child – this is the reason why we needed to have cluster with the wanted path on the right side.

After rakerizing clusters (we have to not forget to split them before the operation and join them after it) we have the $u \dots v$ path represented in the root cluster, expose procedure is finished.

But we brought the top tree in some corrupted form by rakerizing (at most two) compress clusters. Prior the beginning of the next operation (Expose, Link, Cut or Search) we have to undone this and return the top tree into its original form, otherwise the amortization arguments would not work.

When rakerizing we save all rakerized compress clusters into some list and before any other operation (Cut, Link or Expose) we call the Restore operation. The Restore operation checks this list and if there are some vertices it changes them back to compress clusters. All that we could do in constant time at start of every operation.

5.3 Cut

Implementing Cut operation is quite easy thanks to the soft expose operation. First step of the Cut(u, v) operation is to do $soft\_expose(u, v)$ which brings the top tree into state described at the end of the soft expose subsection – depending on the degrees of u and v the cluster representing (u, v) edge will be the rightmost child or grandchild of the root cluster of the corresponding top tree.

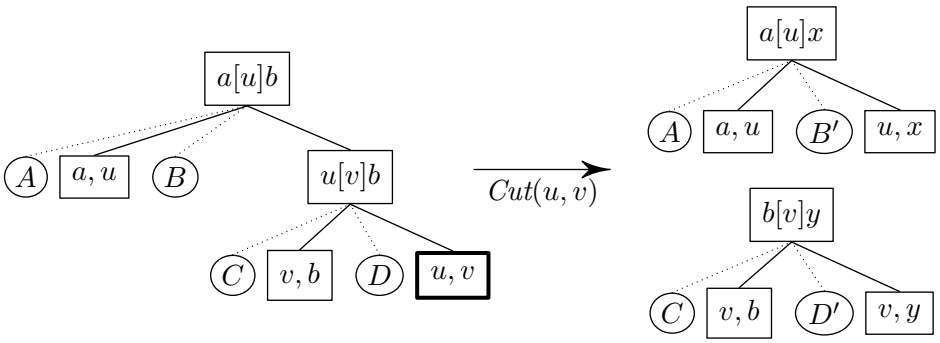

Figure 5.3: Example of cut. Cluster (u, x) is the rightmost cluster from B and B' is B without this cluster (cluster (v, y) and D' in the same way).

We have to destroy the base node representing (u, v) and remove connection between these two handles. After that we have to reorganize clusters to ensure that all clusters have both children.

When one of the vertices have degree only one we are removing leaf edge and there will be only one resulting top tree (or even no resulting top tree when both vertices have degree only one and the whole top tree consists only from this edge). Otherwise we have to split the top tree in the middle points of the two compress clusters which we bring to the root by soft expose.

Starting from the root we detach the right child of the current cluster – but we could not leave the cluster in this form, we need to find new right child.

If there are some nodes in left or right foster children (rake trees) we could take one leaf from the rake tree (base or compress cluster) and make it the new right child of the current cluster. When there is only one cluster in the rake tree it is easy, but what to do if there are multiple clusters?

When taking cluster from the left rake tree we want to get the leftmost one (or the rightmost one in the right rake tree) to maintain order. We splay on the chosen clusters parent (which is rake cluster) to make it the root of this rake tree – chosen cluster will be the left (right) child of this root and the rest of the rake tree will be the second child. Now we can simply remove this root rake cluster, use our chosen cluster as the right child and rest of the rake tree as the new left (right) foster child.

The last case is when the current compress cluster have no foster children. In that case we simply remove this compress cluster and replace it by its left child.

This whole procedure is done at most twice during the Cut operation (third level is the u, v base cluster itself). It produces (two) new root clusters clusters of the resulting top trees after cutting the u, v edge.

5.4 Link

Link is similar to the Cut operation but in the opposite way. First step during the Link(u, v) operation is bringing both u and v to the top of corresponding top trees. This could be simply done by calling $soft\_expose(u, v)$.

Special case is when we are joining solitary vertices. In that case we simply construct new base cluster and return it as the new top tree. Otherwise we choose one vertex (and its handle) as the root of the final top tree, for easier construction we choose the vertex with bigger degree. Then starting from the second cluster (handle of the second vertex) we firstly move its right child into one of its foster children (rake trees). If there is no rake tree it is simple, otherwise we just construct new rake cluster connecting the existing rake tree as one of its child and the former right child of the current compress cluster as the another.

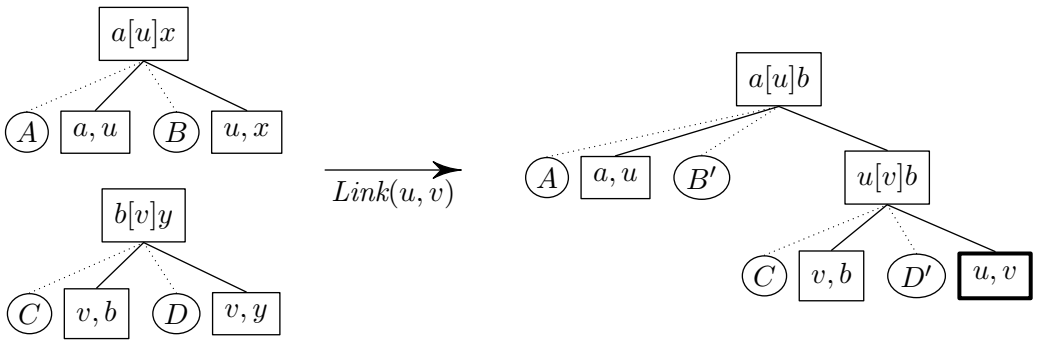

Figure 5.4: Example of link. Subtree B' is B with the (u, x) cluster connected as its rightmost child. Subtree D' and cluster (v, y) similarly.

After this move we simply connect the new u, v base cluster as right child and we are done with this (lower) compress cluster. For the root compress cluster we will do similar procedure with the only difference – instead of the u, v base cluster we will connect modified compress cluster from the previous step as the right child. This leads to constructing the final top tree.

6. Implementation of Top Trees using Topology Trees

This implementation is based on the article *Maintaining Information in Fully-Dynamic Trees with Top Trees* [2] by Alstrup, Holm, Lichtenberg and Thorup. It builds Top trees structure on the base of topology trees introduced in the Chapter 2. The update process of topology trees and some basic overview of top clusters mapping was discussed in that chapter, there we will introduce some details of joining, splitting and the EXPOSE operation (LINK and CUT operations have been described in the mentioned chapter).

6.1 Mapping top trees clusters

Usage of Topology trees as backend for Top trees was described by Alstrup, Holm, Lichtenberg and Thorup in [2]. They described the need of ternarization and how to transform operations with the topology clusters to SPLIT and JOIN functions used in the Top trees structure.

Outgoing edges acts differently in topology clusters and top trees clusters – in topology clusters outgoing edges are not parts of topology clusters, but top trees clusters are based on these edges. Even that this is a major difference the mapping could be done quite easily.

But before mapping we have to deal with fake subvertices and subvertice edges added by ternarization.

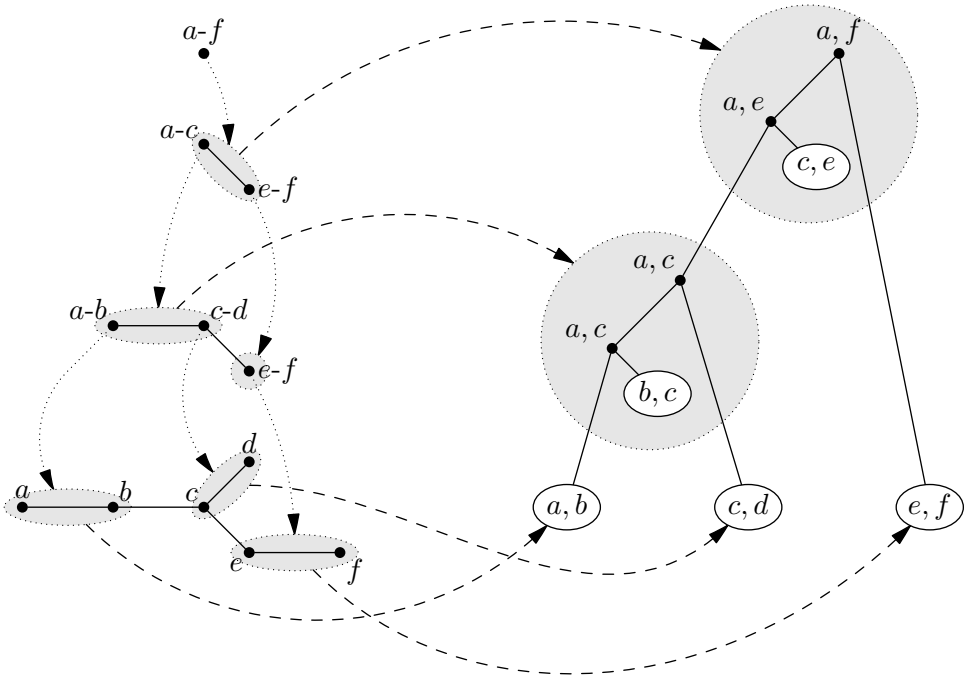

Figure 6.1: Example of mapping topology tree clusters onto top clusters – on base level there are three base clusters and on other levels there is always one topology cluster which combines edge cluster with two clusters from lower levels.

6.1.1 Subvertices and subvertice edges from the Top trees perspective

During ternarization in the Chapter 2 we added additional subvertices and subvertice edges into the graph. That is needed by topology trees but it may be problem for the top trees operations.

Firstly how to deal with subvertices: When performing top trees Join or Split and joined cluster has subvertex as its endpoint, we use the superior vertex of this endpoint instead of the original endpoint when passing endpoint to the Join/Split user function. From the user's points of view all subvertices are represented by the original superior vertex.

And how to deal with subvertice edges? Just ignore them – there are no top clusters associated with them. When joining topology cluster with subvertice edge and two children, we just rake these children ignoring the edge.

6.1.2 Associated top clusters

With each topology cluster may be associated at most three top clusters:

- *Edge cluster* – when there is an normal edge inside the topology cluster.
- *Combined edge cluster* – joined *edge cluster* and cluster from the first child (when the first child has its own top cluster).
- *Top cluster* for the whole topology cluster (joining *combined edge cluster* and cluster from the second child if there is any).

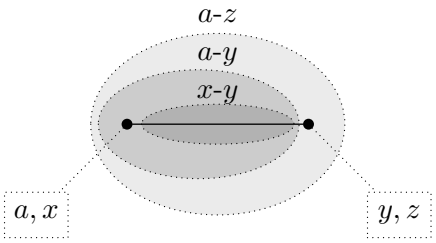

Figure 6.2: Combination of clusters a-x and y-z with edge xy: Firstly edge cluster x-y is created, then it is combined with the first child cluster into a-y cluster and finally this cluster is combined with the second child cluster into a-z cluster.

If topology cluster (or recursive its children) contains at least one normal edge we call it *topology top cluster* (because it has associated top cluster). Otherwise we will call it *empty topology cluster*. Originally all vertices of the original tree are empty topology clusters and by joining them we create topology top clusters.

6.2 Joins and Splits

Join and Split (and Create and Destroy for base clusters) are user defined functions that are called on top clusters with defined endpoints. We cannot

call them directly on topology clusters, so we need to split topology clusters to associated top clusters and call user defined function on these top clusters.

The external view of the topology clusters during splitting and joining is similar as in the first implementation – when operating on some topology cluster we have to firstly ensure that this cluster and all of its parents are splitted. Splitting is done recursively in the top-down manner and all splitted clusters have to be joined after completing all operations in the bottom-up manner. This is done by logging all splitted topology clusters into some list and joining all of them after completing current operation.

6.2.1 Joining

We will recall that if the topology cluster C is not base cluster on the lowest level it could have one or two children (mark them as C_1 and C_2). When joining the C we have to do these operations:

1. If C has only one child $\rightarrow$ Just copy C_1's *top cluster* (with endpoints) into the C's *top cluster* and end.
2. If there is a normal edge between C_1 and C_2 $\rightarrow$ CREATE new *edge cluster* from this edge (otherwise initialize dummy one).
3. If the C_1 is a topology top cluster:
   - If we created an *edge cluster* $\rightarrow$ JOIN C_1's *top cluster* with the *edge cluster* into *combined egde cluster* (depending on C_1's shape set new cluster's endpoints as rake or compress cluster).
   - Otherwise copy C_1's *top cluster* into *combined edge cluster* (with updating endpoints).

   Otherwise just copy *edge cluster* into *combined edge cluster*.
4. If the C_2 is a topology top cluster:
   - If there is valid *combined edge cluster* (if C_1 is a topology top cluster or there is a normal edge) $\rightarrow$ JOIN C_2's *top cluster* with the *combined edge cluster* into C's *top cluster* (depending on C_2's shape set new cluster's endpoints as rake or compress cluster).
   - Otherwise copy C_2's *top cluster* into C's *top cluster*.

   Otherwise just copy *combined edge cluster* into C's *top cluster*.

We have done at most two calls to the JOIN user function and one to the CREATE user function.

6.2.2 Splitting

Split procedure is opposite to the join procedure. Endpoints of all top clusters are correctly set by the join procedure so we have to only do SPLIT and DESTROY operations in the opposite way.

If there is a normal edge in the topology cluster we firstly SPLIT C's *top cluster* into *combined edge cluster* and C_2's *top cluster* and then SPLIT *combined edge cluster* into *edge cluster* and C_1's *top cluster*. Finally just DESTROY the *edge cluster*.

When there is a subvertice edge just Split C's *top cluster* into children *top clusters*. If some of the children is not a topology top cluster we just do copy instead of SPLIT (like in the join procedure).

We have done at most two calls to the SPLIT user function and one to the CREATE user function.

6.3 Expose

The mechanism of the EXPOSE(u, v) is slightly tricky. Basic idea of the EXPOSE in the Top trees structure based on topology trees is to leave the topology trees intact. We only need to split some topology clusters and then take their inner top clusters and join them into a new structure.

This procedure was basically described by Alstrup et al. in [2] but without any details. Here we describe it will all necessary details.

We split all topology clusters in the paths from both exposed vertices to the root of corresponding topology tree, which gave us $\mathcal{O}(\log N)$ top clusters splitted around this path (because of the maximal height of a topology tree for N vertices).

Then we join all these top clusters to construct *auxiliary top tree* such that the root cluster of this auxiliary top tree would be a compress cluster with two given vertices as its endpoints (and all other clusters raked on its path).

After finishing EXPOSE we serve the root cluster to the user and he could do modifications on this root cluster. When it is over we split the auxiliary top tree, which distributes information from the modified root cluster into splitted clusters in the original topology tree. Finally we join all splitted clusters in the topology tree and the operation is finished.

6.3.1 Splitting during expose

Firstly we start with recursive splitting from both given vertices – better said from base topology clusters that contains these vertices. We need to split all clusters that contain them but not as *external vertices* and to save these splitted vertices into *chain* which we will join in the next step.

We have to make clear that *boundary vertices* (or endpoints) that we defined for top trees may be different than *external vertices*. More precisely external vertices are subset of boundary vertices. We defined that each top cluster has two boundary vertices because we defined top clusters as generalized edges. But external vertices are only those boundary vertices that are connected to other clusters. Computation of external vertices may be done in $\mathcal{O}(1)$ from children's external vertices because number of outgoing edges for both children is limited.

With definition of external vertices the splitting part is straightforward – for each cluster (starting from given vertex's base cluster) we check that given vertex is external vertex (by checking outgoing edges) and if no we split this cluster.

Notice that all clusters above the first splitted clusters will be splitted too, because once the vertex stops be external it will never be external again. To have the full coverage of the tree we have to add the last not splitted cluster to the chain too (it will be the first cluster in the chain).

Splitted cluster is divided (by at most two call to the SPLIT user function and one to the DESTROY user function) into at most two topology clusters (one from which we climbed up and second as its sibling) and one edge.

If there is sibling cluster, this sibling has its top cluster and there is a normal edge, we CREATE top cluster for this edge, JOIN this top cluster with sibling's top cluster (according to previous situation of both siblings we join them as rake or compress cluster) and then we add them into chain of top clusters for the next operation. If there is only edge or only sibling's top cluster (when there is only a subvertice edge) we just add it into chain directly without joining.

We only need to care not to add sibling that is on the second vertex's path. For the second vertex the only difference would be that we will stop this process when we reach already splitted cluster (otherwise we would add some clusters twice).

Example of splitted clusters

We will take topology tree from the figure 2.2 (page 14) and let us EXPOSE(b,f). We will start splitting clusters from base clusters containing these vertices as you can see on the figure 6.3.

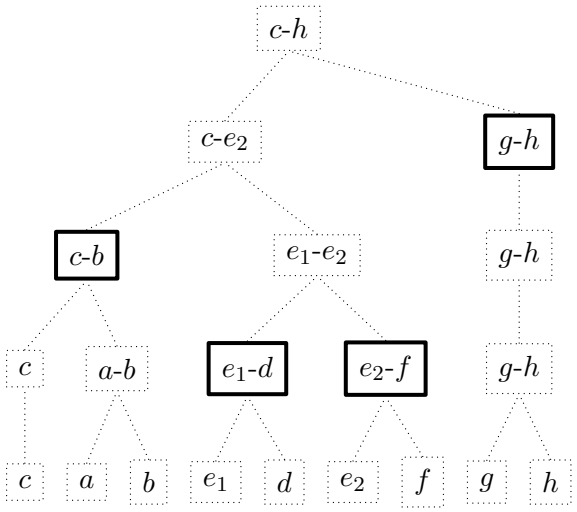

Figure 6.3: Splitting from vertices b and f. Marked clusters will be added into chain – notice that they cover all vertices of the underlying tree and with connecting edges they cover all edges too.

Firstly we split all clusters above vertices b and f. Then starting from the vertex b we continue up to the moment, when b stops be an external vertex – it is in the cluster c-e_2 where we add the previous cluster (c-b) into the chain. Also we should add our sibling e_1-e_2, but this cluster is splitted (because it is on the path from f to the root) so we add only the edge cluster b-e_1.

Continuing above we add sibling cluster g-h combined with edge (so we will

push cluster e_2-h into the chain) and that is all for the first vertex. For the second vertex we continue up until reaching cluster e_1-e_2, where f stops be an external vertex. We add previous cluster e_2-f and sibling e_1-d to the chain (notice that we ignore subvertice edge because it is not a cluster). Above this cluster we hit the path processed by the first run and thus we end.

Starting from both vertices we ended with two chains of clusters that have to be joined together: [c-b; b-e_1; e_2-h] and [f-e_2; e_1-d]. Subvertices of the same vertex acts from this point of view as the same vertex and therefore we may act like all occurrences of the e_1 or e_2 are e (from this point we will not act with topology tree structure and ternarization is not needed).

6.3.2 Chain joining

During splitting we saved splitted clusters along the paths from both given vertices and we saved those splitted vertices into two chains. Now we need to join them. In [2] Alstrup, Holm, Lichtenberg and Thorup suggested some joining based on joining point clusters with arbitrary neighbors and then compressing all remaining clusters on path.

After several tries we decided to do different approach. All clusters in chains acts like generalized edges so we can build a tree from them and then run a DFS (depth first search) on this tree.

Our DFS will return computed top cluster for given subtree and will have three parameters: a current vertex, target vertex and parent cluster (cluster which was used to move to the current vertex). Target vertex will guide joining of clusters so that if subtree under current vertex contains target vertex, the returned cluster will have target vertex as its endpoint.

When joining clusters during Expose(u, v) we call DFS with parameters $chainJoin(u, v, NULL)$.

For each vertex $chainJoin(v, target, parent)$ will do:

- If there is no outgoing cluster (except the *parent*) return only the *parent*.
- For all outgoing clusters C (except the *parent*) with endpoints (v, x) recursive call
$$childCluster = chainJoin(x, target, C)$$
- Rake join all child clusters into *cluster*. If there was a child cluster with *target* endpoint, all clusters will be raked onto this cluster (otherwise rake arbitrarily).
- If v is *target*, rake *cluster* onto *parent* and return this cluster. Otherwise compress *cluster* with *parent* and return this cluster.

The last cluster is returned to the user. Because paths from both given vertices to the root cluster has length at most $\mathcal{O}(\log N)$ there were at most $\mathcal{O}(\log N)$ clusters. Thus the tree used for DFS run has $\mathcal{O}(\log N)$ edges and there were at most $\mathcal{O}(\log N)$ calls to the Join user function.

Now the structure is in a degraded form and before any other operation it must be restored.

6.3.3 Restore

Restore is called manually by the user or automatically by all others operations. If the structure was modified by the Expose there is a topology tree in the right shape, but with splitted clusters. And also there is an auxiliary top tree which could contain modified information that should be distributed into original clusters.

Firstly we need to Split all the clusters in the auxiliary top tree (there is at most $\mathcal{O}(\log N)$ clusters). Clusters on the last level of the auxiliary top tree distribute information to the clusters in the original tree.

Now we can delete the auxiliary top tree and then join all the splitted clusters in the original tree. Because the initial splitting takes $\mathcal{O}(\log N)$ calls to the Split user function the joining would take the same amount of calls to the Join user function.

6.3.4 Keeping original clusters during Expose

Because the original shape of the underlying topology tree remains intact during the Expose operation we may use it for our benefit. From the user's point of view we Split some clusters, then we build from them a new top tree and finally we destroy this new top tree and Join again the same clusters as were before.

In some use cases there are a lot of informations in the clusters used for structural changes in the underlying clusters – for example in the 2-edge connectivity that we will use as one of our experiments. In these cases it may be good to turn off some expensive updates during Join/Split and to turn them on after the top tree is returned into its original shape.

For this behavior we need to remember the original values somewhere. User may do this on it's own (for example by saving this information in edges), but there is an better approach. Because our interface for all user functions pass them pointers to the clusters (and the data is stored inside them) we may keep these original clusters intact after the Split and then pass them to the Join (instead of newly created empty clusters). We will use this functionality in our experiment with the 2-edge connectivity.

7. Experiments

Comparison of both implementations is an important part of the whole thesis and for objective results multiple tests were needed.

I choose two different problems mentioned in the chapter 3. First of them is a problem of maximum edge weight between given vertices with interval updates (described in section 3.2), which uses Top trees directly and aims to work in time $\mathcal{O}(\log N)$ per operation. Second problem is an edge 2-connectivity (described in section 3.3), which uses Top trees "under the hood" and aims to work in time $\mathcal{O}(\log^4 N)$ per operation. Both problems uses Top trees in different ways.

Another necessary condition of good comparison is to have various input data. In our case of investigating Top trees behavior with given problems input data consist of two things:

- Size and type of the underlying graph (number of edges, degrees of vertices)
- Strategy of data structure usage (portion of affected edges, proportion of operation types, ...)

7.1 Experiments strategy

Common scenario for all experiments was introduced. Each experiment was done for both implementations with the same input data and with increasing input size. Common scenario:

- Choose graph size (number of vertices and edges).
- For each chosen graph size choose multiple random seeds to generate initial edges and sequence of operations.
- For each implementation and every generated input run the test and measure elapsed time.
- Compute average elapsed time and standard deviation for each input size.

Python wrapper was used to generate random seeds and to execute testing utilities. Testing utilities written in C++ firstly generates all the input data and sequence of operations, initializes data structures and then began to measure time and execute the operations.

This procedure was chosen to minimize influence of test functions and to measure only the time used by Top trees operations.

7.2 Maximum edge weight experiment

This problem was described in section 3.2 and it operates directly on the underlying tree.

For given size N of the tree and number of operations the initial tree with N edges and $N + 1$ vertices was generated. After that the list of operations was generated (every operation with the same chance) from following operations list:

- Add edge – Choose random two vertices and execute Link operation (when we choose two vertices in the same top tree the Top trees structure should return an error).
- Remove edge – If there are at least $\frac{7}{10}N$ edges choose random edge from list of edges and execute Cut operation (we will maintain array of added edges so every remove operation should be successful), otherwise skip this operation.
- Add weight on path – Execute Expose operation and modify content of the returned root cluster.
- Get weight on path – Execute Expose operation and read content of the returned root cluster.

Every operation executes $\mathcal{O}(1)$ top trees operations and it should take time $\mathcal{O}(\log N)$, where N is a number of edges. Number of edges during execution was maintained between N and $\frac{7}{10}N$ to maintain the same asymptotic complexity.

Both implementations were tested on the same input data. For each implementation the initial edges and list of operations were generated from the same random seed. Then the implementation was initialized from the initial edges and time of this initialization was measured. And finally all the operations were executed and running time of all operations was measured.

Source code of the library for maximum edge weight is located in the header file `include/examples/maximum_edge_weight.hpp` and source code of the experiment itself is in the `src/experiment_edge_weight.cpp` file.

Results from this experiment follows in the section 8.1.

7.3 Edge 2-connectivity experiment

This problem was described in the section 3.3 and it uses top trees as helper data structure for more complex operations.

Cover informations with incident edges are stored in top clusters and as a supplement to the top trees structure it holds informations about all nontree edges and some global counters.

7.3.1 Stored data

Information stored in each top cluster:

- Cover level of the cluster (maximal level on which is the cluster path covered), cover edge of this level.
- Cover informations to be distributed into children on splitting (lazy propagation of *cover* and *uncover* operations).
- Arrays with incident information for both endpoints of this cluster, size of these arrays is $\mathcal{O}(\log^2 N)$.

Outside of the top clusters there are structures associated to the each edge (because edge may change from tree to nontree and vice versa we need to associate this structure with each edge) which holds information about level of the edge and if it is covered by other edges (like the cover information in top clusters). Each edge have $\mathcal{O}(1)$ information associated with it.

And finally there are global incident counters for each vertex, they takes $\mathcal{O}(\log^2 N)$ space.

Because there are a lot of other data structures outside of the top trees structure we cannot use the top trees structure initialization from underlying base graph, but we must initialize the structure by inserting all the edges one by one. During experiments the initialization time was measured independently.

7.3.2 Generating initial graph and choosing number of edges

Important decision during generating graph was number of edges. For graph on N vertices we could have up to $\mathcal{O}(N^2)$ edges, but such graph wouldn't be much interesting for the question of edge 2-connectivity.

We want to have graphs which are not fully 2-connected, but whose have large 2-connected components. During testing graphs with sizes cN^2, $N\sqrt{N}$, $N \log N$ and cN were tested (where N is number of vertices and c some small constant) and size $3N$ was chosen as an optimal size with the best ratio of positive and negative queries (graphs with larger number of edges has more unbalanced ratio of positive and negative queries).

Initial edges of the graph was generated uniformly randomly as pairs of vertices. During experiment the number of edges of the underlying graph was kept between $\frac{7}{10}N \log N$ and $\frac{13}{10}N \log N$.

7.3.3 Test scenarios

Possible operations were described in the section 3.3. For measuring time two scenarios were introduced. In the first one 70 % of operations were queries, 15 % of them insertions and last 15 % were deletions. This experiment aims to simulate some average structure usage with all operations.

Second scenario does only queries to properly measure difference between normal variant of the second implementation and variant of the second implementation with disabled expensive updates during queries.

Source code of the library for maximum edge weight is located in the header file `include/examples/double_edge_connectivity.hpp` and source code of the experiment itself is in the `src/experiment_double_edge_connectivity.cpp` file.

Results from this experiment follows in the section 8.2.

The problem of edge 2-connectivity and its solution is much larger problem than the problem in the first experiment. I tried to implement it in the best way but my aim was the comparison of the two top trees implementations, not building universal solution for edge 2-connectivity.

The experiment still has some undefined behavior in some edge cases (for some random seeds), but it had no effect on comparison – when some failure occurred another random seed was chosen.

8. Results

Measured data for all experiment together with specification of computers used for measure them is part of the attachment of this thesis. Here we will discuss the results.

8.1 Maximum edge weight experiment results

Experiment described in section 7.2 was performed on both implementations and construction time and execution time of operations were measured. Execution time per one operation is showed in the following chart.

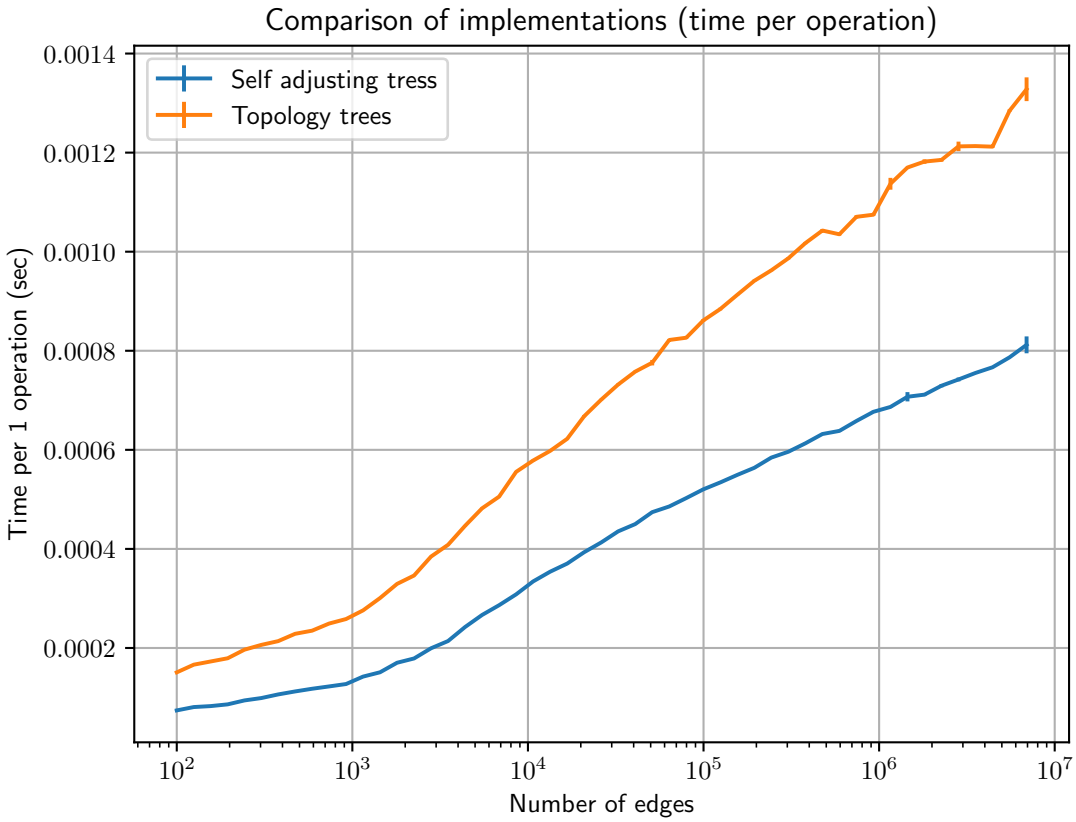

Figure 8.1: Chart showing time per operation in the maximum edge weight experiment

Results show that both implementations have the same asymptotic time complexity but as have been expected implementation with topology trees have larger multiplicative constant.

Despite expectations this multiplicative constant is relatively small – according to measured data the implementation with topology trees for larger inputs (more than 10^4 edges) is only 1.58 to 1.68 times slower than the implementation which uses self adjusting trees.

Running time of construction

Initial construction time measured per one edge of the initial underlying tree is showed in the following figure. It shows some anomaly for small number of edges (see bigger standard deviation in the beginning) but from tree size of 10^4 edges it stabilizes.

It shows that the implementation based on self adjusting trees could initialize in approximately half time than the implementation based on topology trees, which corresponds to time per operation in the previous chart.

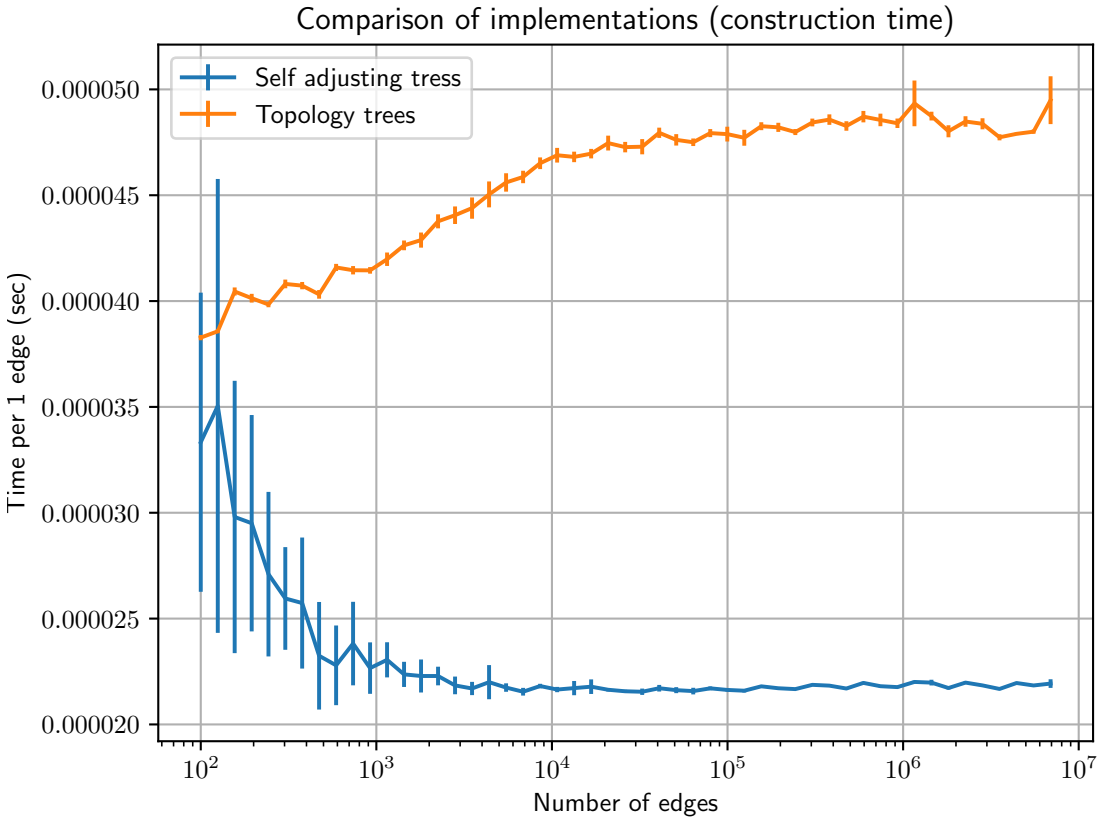

Figure 8.2: Chart showing construction time per one edge in the maximum edge weight experiment

8.2 Edge 2-connectivity experiment results

Similarly to the previous experiment the experiment described in the section 7.3 was performed on both implementations (on second with normal updates and with expensive updates turned off during query). Construction time (time to insert all initial edges) and execution time of all operations and execution time on only query operations were measured.

Running time of all operations

Firstly we analyze the running time of normal operations. As you can see on the following chart, there are some anomalies depending on size of the graph, but mean curve of the measured times shows that all implementations have the same asymptotic complexity (they should operate in $\mathcal{O}(\log^4 N)$). The implementation which uses self adjusting trees has the lowest multiplicative constant as have been expected.

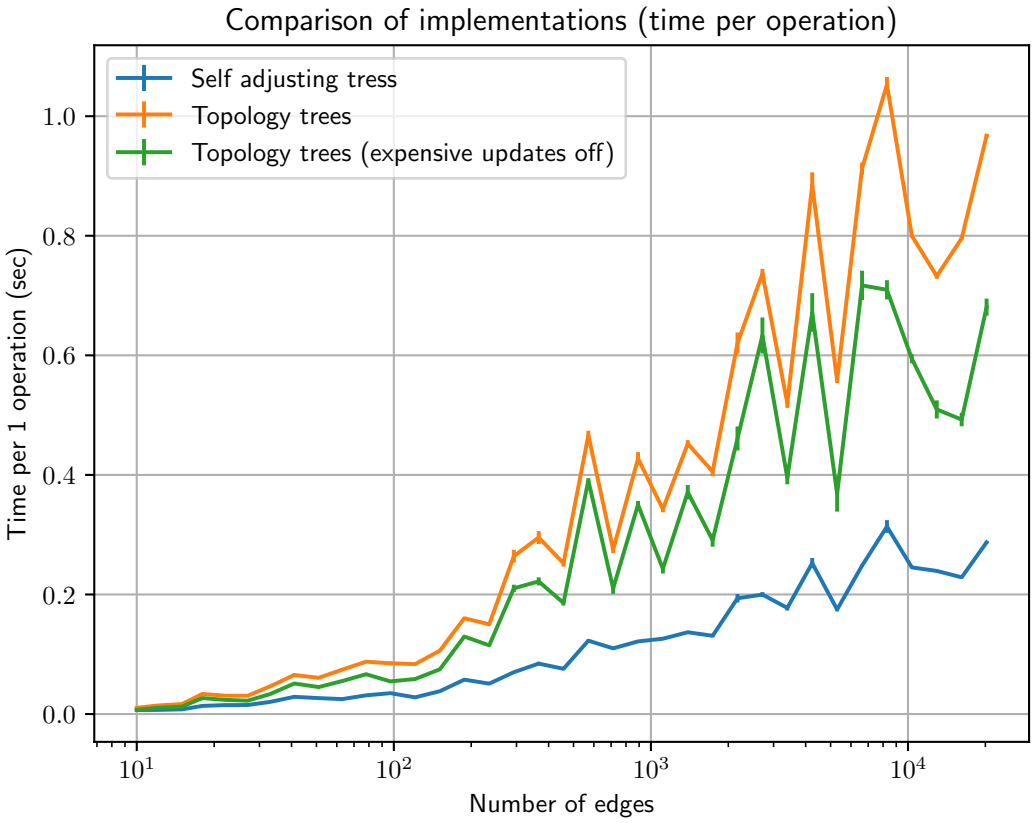

Figure 8.3: Chart showing time per any operation in the edge 2-connectivity experiment

The implementation with topology trees is 2.2 to 3.5 times slower than the implementation which uses self adjusting trees. This multiplicative constant is larger than the same multiplicative constant in the first experiment.

Mark multiplicative constant from the experiment 8.1 as C. Expected value

for the multiplicative constant in this experiment would be approximately C^3 (because operations in this experiment have asymptotic time complexity of $\mathcal{O}(\log^3 N)$ or $\mathcal{O}(\log^4 N)$), but measured multiplicative constant for this experiment is lower. The difference against the expectation might be caused by processor caches – updates in the edge 2-connectivity experiment operates on arrays in the clusters in sequential order which causes less cache misses.

Another interesting observation is that turning of expensive updates during queries have lowered the running time only about 30 % although the ratio of queries was 70 % in all the experiments – so the majority of time is spent on the insertions and deletions of edges.

Running time of only queries

In some uses it might be crucial to quickly answer on queries and other updates may be slower. The second experiment executed only queries and measured their running time.

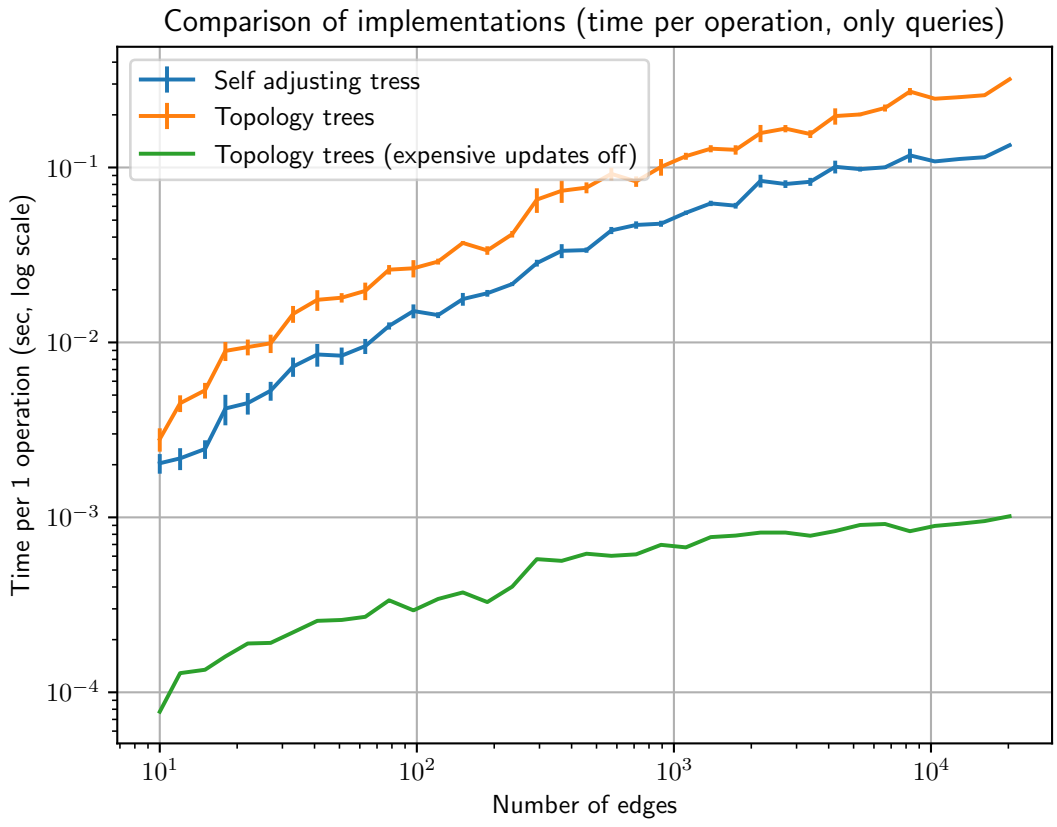

Figure 8.4: Chart showing time per query in the edge 2-connectivity experiment

As you can see on the above chart (it has logarithmic scale on the time axis) the implementation where we could turn off expensive updates during query – topology trees implementations – is the absolute winner. Time complexity of the implementation without expensive updates is asymptotically lower than the time complexity of any other implementation.

Running time of construction

Construction in this problem cannot be done in one step from some underlying tree, but it must be done by sequentially inserting all the edges from the underlying graph. Thus time of construction of graph with M edges is time of sequential insertion of M edges into the empty structure.

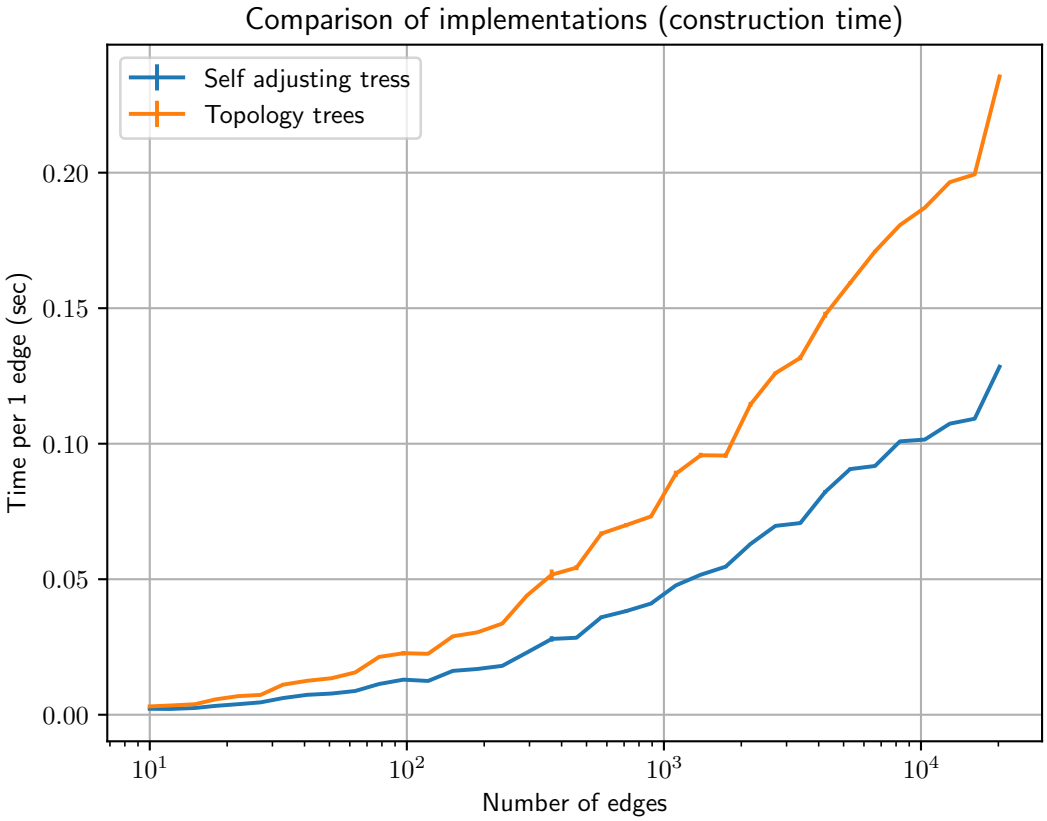

Figure 8.5: Chart showing construction time per one edge in the edge 2-connectivity experiment

As you can see the construction time is similar and as have been expected the implementation which uses topology trees is slower. According to measurement it is 1.7 to 1.9 times slower than the construction of the implementation which uses self adjusting trees.

Not so good result is that the construction time per one edge grows with the graph size and from some size it starts to be unsuitable for real use. For production use it would need some construction method like the maximum edge weight problem.

Conclusion

There are two main products of this thesis: Implementations of Top trees structure and results of experiments.

Both implementations were written from scratch in C++ and they are publicly available for any usage – as part of this thesis or as repository on Github (mentioned in the chapter 4). They both shares the same generic interface, which allows to easily interchange them, so each user of them can choose the one which better fits to his needs.

Also some other works may build upon this codebase and expand it (code is shared under public licence). Both implementations were written with efficiency in mind, but there are still places, where they could be tweaked for even better performance. I will be pleased if there will be active users of this piece of code.

Second product of this thesis are results of experiments discussed in the last chapter. They showed that in most cases the implementation which uses self adjusting trees is better due to larger multiplicative constant of the implementation with topology trees.

Only when Join or Split functions are time consuming and some updates could be turned of during Expose operation the second implementation could be considered as interesting.

The experiment with edge 2-connectivity showed, that when Expose operations are significant but not the only part of all operations, the implementation with topology trees is still slower than implementation with self adjusting trees. But if we want to ensure quick Expose operations or number of Expose operations is asymptotically larger than number of other operations, we may use the second implementation with topology trees and with expensive updates turned off during Expose. In these circumstances it could works much quicker for Expose operation.

These results may help users of provided implementations to choose the right implementation to fit their needs. Also other researchers in the field of Top Trees may build upon these results.

And finally, important result of this thesis is a new knowledge which I earned during working on it. It was an interesting journey through this complex data structure, it teach me a lot of things about dynamic data structures. I hope, that readers of this thesis would be at least half as much pleased as I am.

Bibliography

[1] Robert E. Tarjan and Renato F. Werneck. **Self-adjusting Top Trees**. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 813–822, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[2] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. **Maintaining Information in Fully-Dynamic Trees with Top Trees**. *Computing Research Repository*, cs.DS/0310065, 2003.

[3] Greg N. Frederickson. **A Data Structure for Dynamically Maintaining Rooted Trees**. 24(1):37–65, 1997.

[4] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. **Minimizing Diameters of Dynamic Trees**. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, ICALP '97, pages 270–280, London, UK, UK, 1997. Springer-Verlag.

[5] Martin Mareš. **Dynamické grafové algoritmy**. Master's thesis, Charles University in Prague, Prague, 2000.

[6] Daniel D. Sleator and Robert Endre Tarjan. **A Data Structure for Dynamic Trees**. *J. Comput. Syst. Sci.*, 26(3):362–391, June 1983.

[7] 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, July 2001.

[8] Daniel D. Sleator and Robert Endre Tarjan. **Self-adjusting Binary Search Trees**. *Journal of the ACM*, 32(3):652–686, 1985.

List of Figures

Attachments

Attachment 1: Source code of the two implementations (including experimental data and generators)

Attachment 2: Measured data from experiments and generated charts