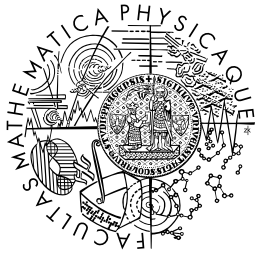

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Lukáš Ondráček

Worst case driver for Top trees

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Vladan Majerech, Dr.
Study programme:	Computer Science
Study branch:	Theoretical Computer Science

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

I thank my supervisor, Vladan Majerech, for introducing me to the topic and for his guidance during the research.

Title: Worst case driver for Top trees

Author: Lukáš Ondráček

Department: Department of Theoretical Computer Science and Mathematical Logic

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

Abstract: A top tree data structure solves one of the most general variants of a well-studied dynamic trees problem consisting in maintenance of a tree along with some aggregated information on paths or in individual trees, possibly in a mutable way, under operations of inserting and removing edges. It provides a simple interface separated from both an internal top tree structure representing a hierarchical partitioning of the graph, and a driver ensuring its depth to be logarithmic, which has a crucial role for the efficiency of the data structure. The driver proposed in this thesis is based on biased trees, combining techniques used in the worst-case version of link/cut trees and in the amortized driver for top trees: An input forest is decomposed into heavy paths and interleaving vertices, all of them being represented by biased trees connected together to form exactly the top tree structure. The driver is meant to be a more efficient alternative to the originally proposed one, and a comparably efficient alternative to the driver proposed by Werneck; there is a room for their experimental comparison.

Keywords: top tree, biased tree

Contents

Introduction

There are many applications of data structures which maintain trees along with some other data under operations of inserting and removing edges, it is so called dynamic trees problem. A typical example of such a usage is finding blocking flow in Dinic's algorithm, where we are repeatedly searching for a minimum-weight edge on a path, decreasing weights of all edges on the path, and inserting other edges; each of the operations can be done in logarithmic time using link/cut trees [ST81], which led to an asymptotic improvement of the original algorithm. Data structures of different level of generality were proposed for the problem, including Euler tour trees, link/cut trees (or ST-trees), topology trees, and top trees. See Werneck's dissertation [Wer06] for details of the data structures and other examples of their use.

The top tree data structure was originally defined by Alstrup et al. [AHLT05]; we will use a slightly different definition of Tarjan and Werneck [TW05]. It is based on a hierarchical partitioning of the *underlying tree* – the tree we want to maintain – into so called *clusters*. A cluster is a connected subgraph with exactly two *boundary* vertices, which can be shared with adjacent clusters; we can think of it as of a generalized edge with the boundary vertices as its endpoints. Each cluster is either a *base* cluster, representing an edge, or it is composed out of two clusters which are its partitioning. The hierarchy of the clusters yields a binary *top tree*: each node of the tree stands for a cluster and its children for clusters partitioning it; its root represents the whole underlying tree and leaves correspond to its edges. In fact, we usually store a forest in the data structure, which leads to maintaining a forest of top trees; further modifications may involve splitting or joining the trees.

There is a nice interface separating application from the internal top tree structure and from a *driver* (or *implementation*), which ensures the depth of the tree to be logarithmic and manages all of its restructurings. An application can access only data in the root cluster; they can be associated either with the whole tree, or with the path connecting the boundary vertices of the cluster. To access data associated with another path, the path has to be *exposed*, which sets its endpoints to become the boundary vertices of the root cluster via restructuring the tree; other restructurings occur if an edge is being added or removed. Each time the top tree is being modified, driver calls user-defined routines to change the data associated with nodes appropriately. The data can be aggregated in the bottom-up fashion or lazily propagated in the opposite direction. The time complexity of provided operations depends on the efficiency of the used driver.

The original driver proposed by Alstrup et al. [AHLT05], served primarily as a proof that the worst-case logarithmic time bounds are reachable. It consisted in reduction from Frederickson's topology tree [Fre83], which had to be managed alongside the top tree causing the construction to be complicated.

Tarjan and Werneck later described much simpler amortized driver [TW05] based on splay trees using similar techniques as in link/cut trees. It is currently the fastest top tree driver according to the experimental measurements [TW10, Set18]. Its only drawback is that all the time and tree depth bounds are only amortized.

Having a fast worst-case driver may be useful, even if we are not interested in persistence of the data structure. Exposing a path involves many structural changes in the top tree and also many calls of user-defined routines, which may be expensive. If we are, for example, performing expose only to read some data and then we want to recover the previously exposed path, we can simply undo all the changes without calling user-defined routines again. In another example, we use top tree for multiple applications with the same graph, all of them manage its own data in the tree and one of the applications now wants to expose a path and make some changes. We can simply disable calling routines of the others, expose the path, let the application update its data, and rollback all the structural changes calling only the routines of the specific application. As a result, the changes required by the application were made, and the data of all the other applications are still consistent without any change. We note that this cannot be done in the case of an amortized driver, because reverting the previous structure breaks its amortization argument and exposing the originally exposed path can result in a different structure from the original one.

Another driver described by Werneck [Wer06, Chapter 3] has worst-case logarithmic bounds and is more straightforward than the originally proposed one, though still more complicated than the amortized version. Its basic idea is similar to that of Frederickson's topology trees: It proceeds by levels; adjacent clusters are paired, so that the pairing is maximal; the pairs form clusters of the higher level, and the procedure is being repeated until there is just one cluster per tree.

In this thesis, we propose a new worst-case driver based on biased trees [BST85]. We combine techniques used in the amortized top tree driver [TW05] and the worst-case version of link/cut tree [ST81], following one of the directions mentioned in the Final Remarks of Werneck's thesis [Wer06]. This driver may be an alternative to the Werneck's worst-case one.

In Section 1, we formally define top tree and describe its parts which are independent of our driver. In Section 2, we decompose the underlying tree into paths, which form a partition of its edges. In Section 3, we describe globally biased binary tree. In Section 4, we transform the decomposition from Section 2 into binary *compress trees* and *rake trees*, which composed together create the top tree structure. A compress tree represents one path of the decomposition; leaves of the compress tree of an isolated path are just the base clusters of the edges of the path in the correct order. A rake tree groups multiple paths ending at the same vertex; leaves in this tree are the roots of the compress trees of the paths. In general, base clusters in a compress tree can be interleaved by the roots of rake trees. Both compress and rake trees are balanced as biased trees. In Section 5, we connect all the pieces together and describe how adding and removing edges work.

1. Top Tree

We now define top tree, its interface, and operations *expose* and *non-local search*, which are independent of the used driver.

1.1 Definition

We follow the definition of Tarjan and Werneck [TW05] with clusters of just two endpoints, instead of the original definition [AHLT05] with clusters of up to two endpoints.

Definition 1 (clusterization)**.** A *cluster* of a tree is its connected subgraph with just two vertices marked as *boundary*; all other cluster vertices are said to be *internal*. A *clusterization* of a tree is a partitioning of its edges into clusters sharing only their boundary vertices.

We can think of a cluster as of a generalized edge connecting its boundary vertices. We have two operations: *compressing* a vertex and *raking* one cluster onto another one. Having a vertex v and its only neighbours u and w, compressing v means replacing the edges uv and vw with one edge uw and removing v. Having v with neighbours u and a leaf w, raking the edge vw onto uv means replacing them with one edge uv and removing w. We use these two operations to combine two adjacent clusters into one, which has the removed vertex as internal and the other two as boundary.

Definition 2 (top tree)**.** A *top tree* is a binary tree representing a hierarchical clusterization of a given underlying tree. Each node corresponds to a cluster with explicitly specified boundary vertices and its children correspond to its clusterization. Nodes are of three types:

- A *base node* stands for an edge of the original tree; it is a leaf in the top tree.
- A *compress node* stands for compression of a shared vertex of subclusters; the shared vertex disappears from the new boundary.
- A *rake node* stands for raking one of its children onto the other one resulting in the boundary vertices being the same as of the second child.

The root of the top tree corresponds to the whole underlying tree and its boundary vertices are called *external*. Clusters of the three types of nodes are illustrated in Figure 1.1.

Each node represents both the whole subtree of the cluster and the path connecting its boundary vertices, which is also part of the cluster. The root cluster thus represents the whole tree and the *exposed* path connecting its external boundary vertices. Top tree provides operations link to add a new edge, cut to remove an edge, expose to change the boundary vertices, and non-local search, which we will describe below; all of them involve some structural changes in top trees.

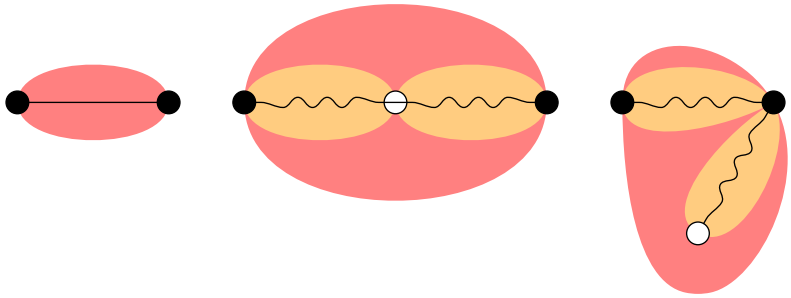

Figure 1.1: The types of top clusters. From left to right: base cluster, compress cluster, and rake cluster. The black vertices are the boundary vertices of the topmost (red) clusters, the white vertices are boundary only in the children (orange) clusters.

An application can store some data associated with subtrees or paths in the corresponding nodes. The interface of top tree allows accessing only the data in root nodes and calls user-defined routines create, join, split, and destroy when the structure of top trees changes. After creating a new base node create is called to initialize the user data in it; before destroying a node destroy is called. The join routine is called if two root nodes are being joined together by attaching them as children of a new root node, and split when a root node is being destroyed yielding its children as root nodes. Every time tree is being restructured, operations are done in the following order: splitting nodes, then destroying nodes, then creating new nodes, and finally joining nodes together.

Let us describe one example of using such an interface. We would like to maintain a forest with weights on edges and be able to find a minimum on a given path. In each cluster, we store a minimum on the path between its boundary vertices. We need to set the value when create or join is called. When creating a base cluster, the minimum is just the weight of the edge. When joining two clusters, we need to distinguish compressing, in which case minimum of the subclusters is used, and raking, which requires just copying the value of the subcluster with the same boundary vertices. Now we can create the forest by a sequence of link and cut operations and then simply expose an arbitrary path and read its minimum from the root node.

We can easily extend the previous example to allow modifying the weights. Specifically, we are interested in increasing/decreasing all the weights on a path by a constant. To achieve this, we modify the minimum value in root and store there the value of increment which should be added to all weights on the path. If split is called, we read the value and modify minima and the increments in subclusters (or just one of them if they were raked). Now we can modify all the weights on an exposed path without any asymptotic slowdown and they will be lazily propagated when necessary. To update an individual edge, we can expose its endpoints and update the minimum on this one-edge path.

For more examples, see [AHLT05] or [Wer06].

1.2 Representation, Expose, Non-local Search

In each node t, we store the pointers to its parent and children, and to the boundary vertices top_boundary(t). Each vertex v of the underlying graph contains a pointer top_node(v) to the bottommost node with v being its inner vertex if such a node exists, the pointer leads to the root cluster of the tree containing v otherwise.

In time proportional to the depth of the tree, we can find the root top_root(t) of the top tree containing the node t. Vertices u and v are in the same component if and only if top_root(top_node(u)) = top_root(top_node(v)).

We use a straightforward expose implementation described in [AHLT05]. This temporarily increases depth of the tree and breaks path decomposition used by our driver, so we need to unroll all the structural changes before performing other expose/link/cut operation. To restore the original structure, we just perform all the operations reversed, including calling joins instead of splits and vice versa; modifications of user data are thus propagated as they should be. In the following sections, we always assume the data structure in its unexposed form. The implementation of expose is the following:

expose(vertex u, vertex v):
1. Split top_node(u) and top_node(v) and all their ancestors top-down.
2. Compress and rake all nodes outside uv-path
   and rake them onto uv-path nodes (in any order).
3. Compress the whole uv-path (in any order).

Let us note that the order of splits has to be always top-down and the order of joins bottom-up, because we need to apply these two operations only on root nodes. This is needed for proper aggregation and propagation of user data. To ease understanding, we will not take care of the order in the following text; it is always possible to rearrange the operations to satisfy this requirement.

Claim 1. *The time complexity of expose excluding calls of user-defined routines and the number of the calls are $\mathcal{O}(d)$, where d is the original depth of the top tree. After performing expose, the depth is at most thrice as large as originally.*

Proof. There are at most $2d$ splits in Step 1 leading to at most $2d + 1$ root clusters of the maximal depth d. We need at most $2d$ joins in the remaining steps. Performing the joins in any order, we obtain a tree of depth at most $3d$. □

Finally, we mention the *non-local search* operation, which was also described in [AHLT05]. This operation is used to search for an edge or a vertex in the underlying tree; for example, its center or median. After invoking the operation, a user is repeatedly given a root cluster with a question whether the wanted object is in its left or right subcluster, then the tree is restructured and the user asked again. At time of asking, the cluster always represents the whole tree in spite of changes in its hierarchical clusterization. The time complexity of this operation is asymptotically the same as of expose.

The operations link and cut are part of our driver and will be described later, as well as the operation construct building the whole top tree in linear time.

2. Decomposition into Paths

2.1 The Decomposition

Our path decomposition is similar to heavy-light decomposition used in [ST81] with two main differences: All edges are covered by the paths – they are connected by vertices instead of light edges; and we use a little different heavy edge property extending all paths up to leaves.

The following definition is more general to correspond to all possible decompositions representable by our data structure. Then we describe which decompositions we are interested in and how they can be maintained under the operations link and cut.

Definition 3 (path decomposition)**.** A *path decomposition* of a tree is a set of edge-disjoint paths covering the whole tree, one of which is marked as the *main* path. It has to be possible to create the decomposition iteratively in the following way: The main path can be chosen arbitrarily; every other path connects any vertex of an earlier created path with any other vertex. The endpoint of a path closer to the main path is called its *top* and the other one is its *bottom*; the orientation of the main path in terms of top and bottom may be undefined.

A decomposition path, for example, cannot connect inner vertices of other decomposition paths. For every vertex, at most one path can pass through it or have there its bottom; any number of paths can have there their tops. The terminology of top (and bottom) endpoint originates from Frederickson's topology trees [Fre83] and has no relation to the name of top tree. Tarjan and Werneck in [TW05] use terms tail and head instead.

Definition 4 (rooted path decomposition)**.** A path decomposition is *rooted* at a vertex r if it can be created by the following process:

1. Mark r as the only visited vertex and all edges as uncovered.
2. Repeat until all edges are covered:
3. $v \leftarrow$ any visited vertex incident to an uncovered edge
4. $p \leftarrow$ a new path containing only v (the first such path is the main path)
5. While v is incident to an uncovered edge:
6. $e \leftarrow$ any such edge leading to a subtree with maximal number of edges
7. Extend p with e.
8. Mark e covered.
9. $v \leftarrow$ the other endpoint of e
10. Mark v visited.

Step 6 implies that the decomposition is not necessarily unique. If we are referring to top of the main path, we mean the root of the decomposition; bottom is the other endpoint. However, we usually want our path decomposition to be a little more restrictive and rooted at two vertices at the same time, which makes the orientation of the main path unclear again.

Definition 5 (symmetric path decomposition)**.** A path decomposition with the main path uv is *symmetric* if it is rooted at both u and v.

The main goal of rooted and symmetric path decompositions is the following:

Claim 2. *A path between any two vertices u and v in a tree intersects at most logarithmic number of paths of a decomposition rooted at an arbitrary vertex w.r.t. the size of the tree.*

Proof. We begin at the vertex of uv-path closest to the root and walk towards u or v counting decomposition paths. Each time, we use an edge of other decomposition path than previously, we descent to a subtree which has at most the same number of edges as the subtree of the following edge in the previous path. Each such step at least halves the size of the subtree determined by the current vertex, which cannot occur more than logarithmic number of times. □

2.2 Reroot and Symmetrize

To change one decomposition into another, we use an operation called splice. It divides the path (if it exists) passing through a given vertex v into two paths, one having there its bottom and the other its top; then it may connect the path with bottom at v with any path with top at v. If the original path is the main path, its part containing its top possibly extended by another path stays the main path; we have to determine (or change) its orientation in such a case. We thus change the main path by cutting off its part and extending it via different one; it is possible that we need to cut it just at its top and extend it with another path, so that the decomposition paths are not changed except for the information which one is the main path. Each path has at least one edge before and after splice.

We measure the difference between path decompositions in the number of splices needed to create one from the other; we are not taking into account changes in the orientation of the main path during such a transition (their number is not higher than the number of splices).

To ease dealing with different decompositions rooted at the same vertex we prove the following lemma:

Lemma 3. *For two rooted path decompositions D_1 and D_2 sharing part of their main paths, there exists a path decomposition D_2' such that*

- *it is rooted at the same vertex as D_2,*
- *it has the same main path as D_2, and*
- *it differs from D_1 only in splices on the main paths of D_1 and D_2.*

Proof. We create D_2' by the algorithm in Definition 4 according to how D_1 was created. We visit all edges outside the main paths of D_1 and D_2 in the same direction in both processes and so their subtrees can be created in the same way. □

To prove that a rooted path decomposition is symmetric, it is now sufficient to show that we can create the main path also from its other end using our algorithm. Then there exists a decomposition rooted there which doesn't differ on the main path and also anywhere else. We use it in the following claim:

Claim 4. *The transition between symmetric path decomposition and path decomposition rooted at a given vertex requires at most logarithmic number of splices on the main path of the rooted one and no splices elsewhere.*

Proof. By Lemma 3, it suffices to describe creation of the main paths of the decompositions. We will describe both transitions independently; see Figure 2.1.

Let D_r be a path decomposition rooted at v with the other end u of the main path. We create a symmetric path decomposition D_s as rooted at u in such a way, that we follow the original main path as long as possible while creating the new main path. We either end again at v, or we leave the main path of D_r at a vertex z continuing to a leaf w. The former case is easy, we have just discovered that D_r is already symmetric. Let us concentrate on the latter case. We denote G_v, G_u, G_w some of the components of the original tree after removing z containing the respective vertices v, u, w (we suppose G_v to be empty if $v = z$). We know $|V(G_u)| \geq |V(G_w)|$ from D_r and $|V(G_w)| > |V(G_v)|$ from D_s, therefore $|V(G_u)| > |V(G_v)|$. Beginning the main path of a new decomposition rooted at the vertex w, we have to go to z and then we can continue to u, which proves that D_s is symmetric.

Let D_s be a symmetric path decomposition with u and w the endpoints of the main path, v the root of a wanted decomposition D_r and z the closest vertex to v on the uw-path. Starting at v, we can reach the vertex z, because after removing it, either G_u or G_w has to have the maximal weight of all components, no matter where on the uw-path z was. Then we can surely continue along one of the original main path's branches up to u or w; without loss of generality to u to match the figure.

We know that all the necessary splices are on the main paths. Moreover, we can observe that only one splice is needed on the main path of D_s, in the vertex z: The path zw is part of the main path in D_s rooted at u and is just besides the main path in D_r, it can be thus created in the same way in both of them and there is no need for splices outside of the main path of D_r. The number of splices is at most logarithmic, because the uv-path cannot use more than logarithmic number of the decomposition paths of D_s according to Claim 2. □

The claim yields operations reroot and symmetrize. The former transforms a symmetric decomposition into rooted at a given vertex and the latter transforms a decomposition rooted at a given vertex into a symmetric one:

```
reroot(vertex v):
1   Let u, w be the endpoints of the main path.
2   If v is on the uw-path:
3     Split the uw-path at v using splice; set heavier subpath as the main path.
4   Else:
5     If there is a path passing through v, split it at v using splice.
6     p ← the path with bottom at v
7     v ← the top of p
8     While v is not on the uw-path:
9       Perform splice in v to extend p.
10      v ← the top of p
11    Perform splice in v to extend the heavier branch of the uw-path with p.
```

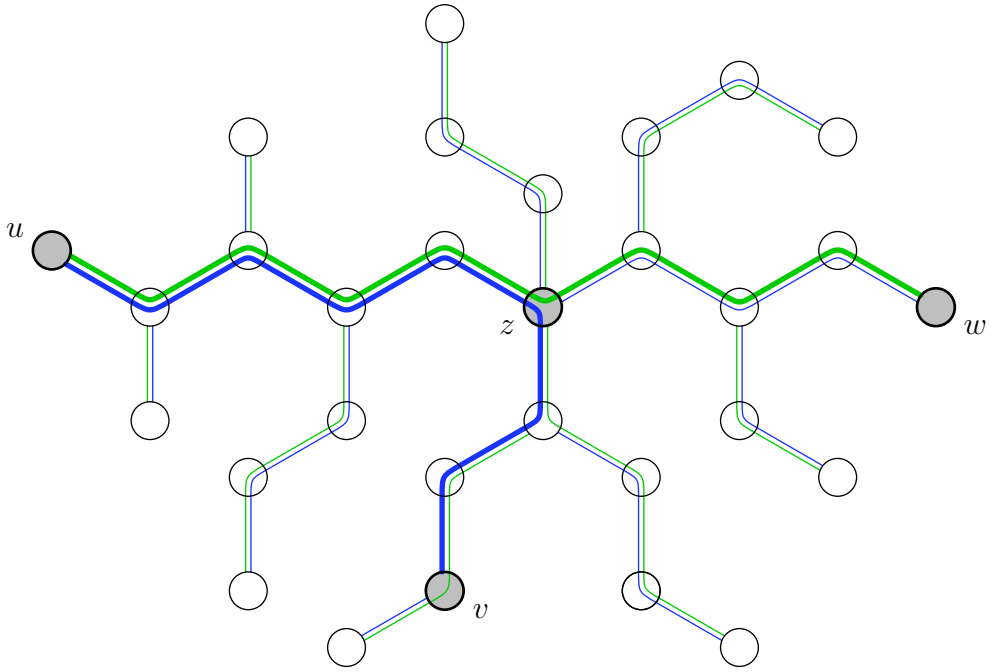

Figure 2.1: An example of transition between a symmetric path decomposition with roots u and w, composed out of the green paths; and a path decomposition rooted at v, composed out of the blue paths. Main paths are drawn in bold. The two differs only in splices on the v-rooted main path.

symmetrize(vertex v):
1. Let u be the other endpoint of the main path, the new root.
2. $z \leftarrow$ the closest vertex to u on the uv-path where it continues wrong way (end if it doesn't exist)
3. Perform splice in z to fix the main path.
4. Repeat:
5. $p \leftarrow$ the path containing v
6. $v' \leftarrow$ the closest vertex to the top of p, where p continues wrong way (break if v' doesn't exist)
7. Perform splice in v' to fix p.

2.3 Link and Cut

Now we can describe link and cut operations in terms of reroot and symmetrize. We assume the path decomposition in a symmetric form before and after link/cut operation.

link(vertices u, v):
1. Swap $u \leftrightarrow v$ if u is in the lighter component.
2. reroot(u), reroot(v)
3. $w \leftarrow$ the bottom of the path containing v
4. Add a new edge uv.
5. symmetrize(w)

After adding the edge in Step 4, we obtain a correct decomposition rooted at w, because the main path from w has to connect it with the heavier component through the new edge and continue there.

```
cut(edge uv):
1   If uv is not on the main path:
2       Swap u ↔ v if u is farer from the main path.
3       w ← the bottom of the path containing v
4       reroot(w)
5   Remove the edge uv.
6   symmetrize(u), symmetrize(v)
```

In Step 4, we have created a main path passing through the edge uv. After removing the edge, we obtain correct decompositions rooted at u and v.

We will define all the operations splice, reroot, symmetrize, link, and cut in more detail in Section 5.

3. Globally Biased Binary Tree

Biased trees were proposed by Bent, Sleator, and Tarjan in [BST85]. Here we only sketch their basic idea and properties of globally biased binary trees, which we use for balancing the compress and rake trees. For a complete description see the original paper.

A biased tree is a search tree with values in leaves, whose depths depend on their weights. The depth of a leaf i with a weight w_i is $\mathcal{O}(\log(W/w_i))$, where W is the sum of weights of all leaves. In addition to the find operation, split and join are provided; they can be either two-way or three-way. The two-way split divides the tree into just two parts, while its three-way variant returns also the element we are splitting at as an independent one-node tree. The two variants of join does the inverse. We can easily provide also insert and delete by combining a two-way split with the three-way join, and vice versa. All the operations cost linear time w.r.t. the depth of the leaf (or leaves) we search, or split or join at (or between).

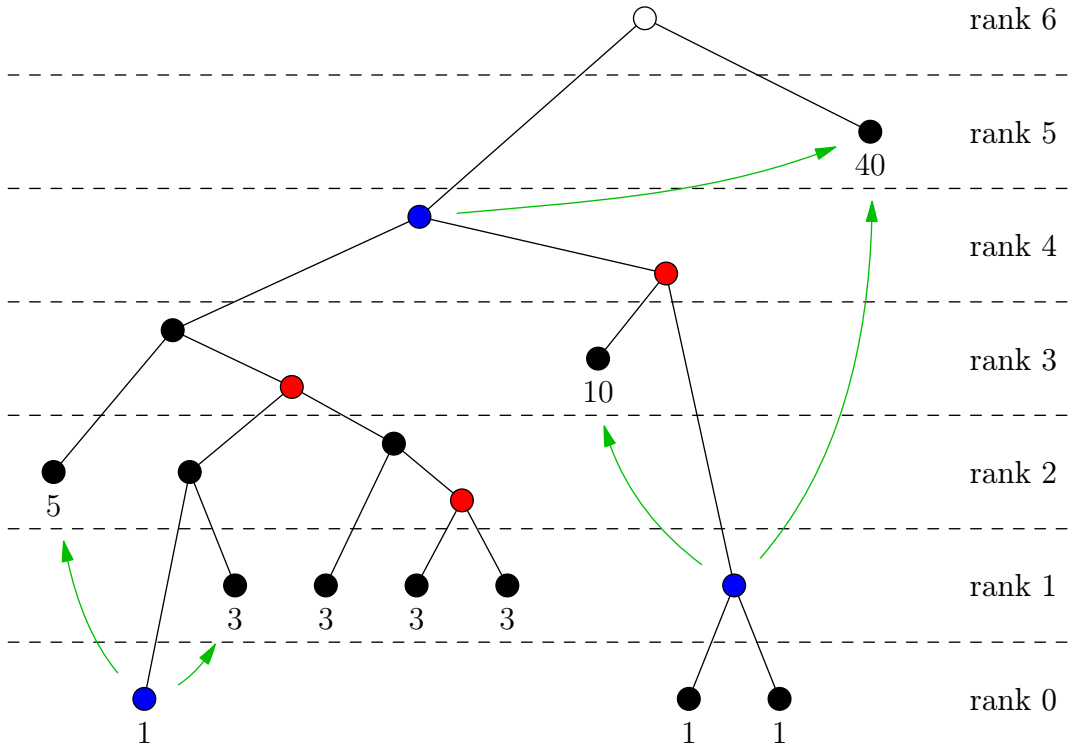

Figure 3.1: An example of a globally biased tree. The individual nodes are vertically partitioned by their ranks and leaves are labeled by their weights. The ranks of leaves are calculated from their weights w_i as $\lfloor \log_2 w_i \rfloor$, the ranks of the other nodes depend on their children according to the mentioned invariants. The last invariant, the global bias property, is highlighted by the green arrows, leading from the minor nodes to the corresponding neighbouring leaves.

Balancing invariants of the tree are expressed w.r.t. the weights $w(x)$ of leaves x and integer ranks $s(x)$ of nodes x derived from the weights. To ease understanding, we can also translate them to the red-black tree terminology, or more precisely generalized red-black-blue tree. The translation is, however, not lossless; we thus need to keep the original formulation in mind. A node y

with the parent x has red colour if $s(x) = s(y)$, black colour if $s(y) = s(x) - 1$, and blue colour if $s(y) < s(x) - 1$. We say that red and black nodes are *major* while blue nodes are *minor*. We further define a *neighbouring leaf* of an inner node to be a leaf adjacent to the whole subtree of the node. The invariants are:

1. For a leaf x, $s(x) = \lfloor \log_2 w(x) \rfloor$.
2. For x parent of y, $s(y) \leq s(x)$ (each node except root has a colour); if y is a leaf, $s(y) \leq s(x) - 1$ (leaves are black or blue).
3. For x grandparent of y, $s(y) \leq s(x) - 1$ (child of a red node cannot be red).
4. For minor y with the parent x and a neighbouring leaf z, $s(z) \geq s(x) - 1$ (so called global bias property).

See Figure 3.1 for an example. In the original paper, biased trees were first described as biased 2,4-trees, which can be obtained from the binary trees by merging all adjacent nodes with the same rank. There were also described locally biased trees, whose time bounds are only amortized.

Globally biased trees have the following properties:

Claim 5. *Let W be the sum of weights of all leaves, then it holds:*

a) The depth of a leaf i with weight w_i is $\mathcal{O}(\log(W/w_i) + 1)$.

b) The time complexity of three-way join and split is $\mathcal{O}(\log(W/w_i) + 1)$, where w_i is the weight of the leaf we are splitting or joining at.

c) The time complexity of two-way join and split is $\mathcal{O}(\log(W/(w_- + w_+)) + 1)$, where w_- and w_+ are the weights of the adjacent leaves we are splitting or joining between.

d) The whole tree can be constructed in linear time by inserting the elements in the required order and using a little modified join operation.

e) The operations join and split can be easily modified, so that: In each inner node, we can store immutable data derived from the immutable data in its children; and we can store mutable data in a node to be lazily propagated to its descendants.

Proof. We omit proofs of a), b), c), d) referring to the original paper [BST85]. We have just modified the depth and time complexities by adding an additive constants to handle even the special cases like one-node trees.

For storing bottom-up aggregated data, we need to know that every time a node is being created subtrees of its children already exist and will not be changed until the node is no longer needed or we have enough time to update the node if necessary. For storing top-down propagated data, we need to ensure the propagation of the data from nodes that will be destroyed; specially bottom-up split has to be preceded by top-down propagation on the same path. □

4. Compress and Rake Trees

We compose top tree out of compress trees and rake trees. The main idea of connected compress and rake trees originates from Tarjan and Weneck [TW05], but the exact realisation differs. A compress tree represents one path of the decomposition. Its leaves are base clusters, standing for edges, and roots of rake trees, standing for vertices, which are present only if there are some other paths having tops at the vertices. The leaves are ordered from the bottom of the path to its top. A rake tree groups paths with their tops at the same vertex, its leaves are the roots of the compress trees of the paths. The root of a top tree is the root of the compress tree of the main path. We often don't distinguish between a compress or rake tree and its root.

The leftmost and rightmost leaves in a compress tree are usually base nodes. There can be a rake tree at the bottom of a path, but only temporarily, since the bottoms of paths in rooted decompositions are leaves. A rake tree at the top of a path is allowed only in the root path; otherwise, there already exists a rake tree for the same vertex closer to the root of the top tree, which should be used instead. Hence there are no rake trees at the ends of paths if the decomposition is symmetric.

The types of nodes (base node, rake node, compress node) in compress and rake trees are not as evident as one may expect. An inner node in a rake tree is always a rake node; a leaf can be either a compress node, or a base node if the compress tree is one-node, or even a rake node if the compress tree contains only one edge and one rake tree (at the bottom). The father of a rake tree root in a compress tree is always a rake node; all other inner nodes in a compress tree are compress nodes; a leaf can be either a base node, or any other type of node, since the rake tree can be also one-node.

We assign a weight to each node in top tree; it equals the number of edges in the cluster. In other words, base nodes have weight 1 and weights of all other nodes are sums of weights of their children. In each individual compress or rake tree, we use the weights of its leaves for balancing it as a biased tree.

See Figure 4.1 for an example of a clustered underlying tree and the corresponding top tree.

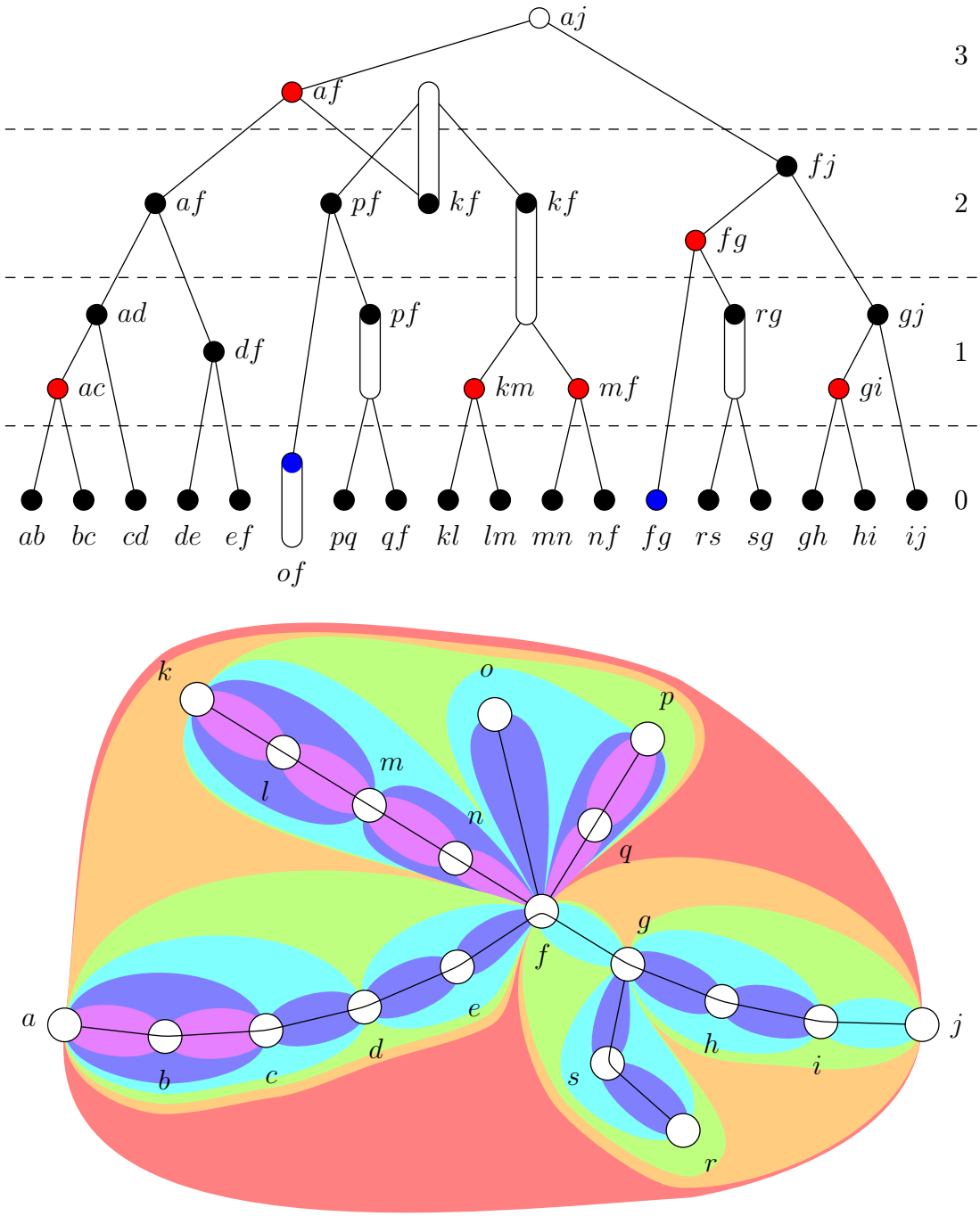

Figure 4.1: An example of a tree clusterization and the corresponding top tree. The example path-decomposed and clustered underlying tree is drawn at the bottom and its top tree composed out of the compress and rake biased trees above it. The path decomposition of the underlying graph can be seen in its vertices; the hierarchically organized clusters are drawn as coloured regions having exactly two vertices at their boundaries. The leaves of the top tree correspond to the edges of the underlying tree; all nodes are labeled by the boundary vertices of the corresponding clusters. The nodes are also vertically partitioned by their ranks in the biased trees, which are written on the right-hand side. A node being simultaneously a root in one biased tree and a leaf in another one is drawn vertically stretched with the coloured endpoint representing the leaf; it can have a slightly different ranks in the two biased trees.

4.1 Rake Trees

We order the leaves in a rake tree ascending according to their weights. It allows us to perform two-way split/join in the same asymptotic time as the three-way variants, as we will see later.

Besides its weight, every inner node of a rake tree contains also the maximum of weights of its leaf descendants. We can therefore discover the maximum weight of path with the top at a specified vertex directly from the root of the corresponding rake tree. It will be useful for finding the vertices of splices in the symmetrize routine.

We define the following operations:

- rake_heaviest$(R) \to C$
  traverses the tree and returns the rightmost leaf (with the maximal weight) in the rake tree R.
- rake_root$(R) \to R'$
  traverses the tree and returns the root of the rake tree containing R.
- rake_join$(R_1, C, R_2) \to R$
  joins the rake tree R_1, a new leaf containing the compress tree C, and the rake tree R_2. If C is nil, the two-way join of R_1 and R_2 is performed. If R_1 or R_2 is nil, it is considered to be an empty tree.
- rake_split$(R, C) \to R_1, C', R_2$
  splits the rake tree R at C into rake trees R_1 and R_2. If C is a compress tree being a leaf of R, it is returned as C' and excluded from R_1 and R_2; if C is a compress tree not included in R, a two-way split is performed at an inner node where C belongs according to its weight and C' is nil; if C is nil, $(R, \mathsf{nil}, \mathsf{nil})$ is returned.

4.2 Compress Trees

Compress trees are a little more complicated. Each inner node or base node represents a subpath of the decomposition path. We need to know which of the endpoints in its top_boundary is at the top and which at the bottom, while we have to be able to reverse the whole path. We also need to be able to find a vertex in which splice should be performed in the symmetrize routine. Namely, we define:

- compress_top$(C) \to C'$; compress_bottom$(C) \to C'$
  return endpoints of the subpath represented by the node C.
- compress_reverse(C)
  reverses the path in constant time, C has to be the root node of a top tree.
- compress_heavier$(C) \to R$
  returns a rake tree root R, a leaf descendant of C, which contains a heavier path continuation than the current one from R up to the bottom of the path; the closer one to the root if ambiguous; nil if no such node exists. Its time complexity is linear in the depth of R in C, or constant if R is nil.

- compress_root(C) $\to C'$
  traverses the tree and returns the root of a compress tree containing C; the closer one to the root of the top tree if ambiguous. Such an ambiguity may occur if C is the only node of a rake tree, it is then a root in one compress tree and a leaf in another one.

- compress_join(C_1, R, C_2) $\to C$
  joins the compress tree C_1, a new leaf containing the rake tree or edge R, and the compress tree C_2 (in this order). If R is nil, the two-way join of C_1 and C_2 is performed. If C_1 or C_2 is nil, it is considered to be an empty tree; C can be also nil.

- compress_split(C, R) $\to C_1, R', C_2$
  splits the compress tree C at its node R into compress trees C_1 and C_2. If R is a rake tree or an edge being a leaf of C, it is excluded from C_1 and C_2 and returned as R'; if R is an inner node of C, R' is nil; if R is nil, $(C, \mathsf{nil}, \mathsf{nil})$ is returned.

In each node, we store its bottom and top endpoints. They are just the boundary vertices, which have to be already pointed to as top_boundary, but we need to know which ends they represent. Recognizing compress_top from compress_bottom is then as hard as recognizing the right child from the left one.

To handle reversals, we store one bit in each node saying whether the meaning of left and right children, and bottom and top pointers is swapped in the whole subtree of the compress tree. To reverse the whole path via compress_reverse, we just need to flip the bit. We do it only in the root node of a top tree, which doesn't break the biased tree invariants.

To be able to decide, what is the current orientation of the path in a node, we need to descent to the node from the root of the compress tree calculating xor of all the seen bits. We have to keep this in mind when going from leaf to root and visit the root in advance. It has no effect on the asymptotic time complexity of our data structure, because we always need to visit the root at some time and having all the bits accumulated, we can xor them in the backward order when going up. We will assume that all other methods including compress_bottom, compress_top, compress_join, and compress_split take these bits into account.

To search for a rake tree with heavier path continuation in it, we will use the same technique as in the worst-case implementation of ST-trees [ST81]. We consider the following difference of weights for a rake tree R on a path P:

$$w((R)\cdots\mathsf{compress\_bottom}(P)) - w(\mathsf{rake\_heaviest}(R)), \tag{4.1}$$

where the first weight includes all the base clusters and rake trees on the subpath from R to the bottom excluding R. The expression is less than zero if and only if R corresponds to one of the wanted vertices. Thus, being able to find such a rake tree closest to the top of the path is what we need.

Descending from the root to a given rake tree, we can easily calculate the value by summing the weights of all left children we don't visit, and finally, subtracting the greatest weight stored in the root of the rake tree. However, to seek the correct rake tree in compress_heavier, we need to know the minimum value of (4.1) in

the subtree of each node. Then we can in each node just choose the child with the negative minimum, or the closer one to top if both minima are negative, or return nil if there is no such child.

We store the minimum value in the root of compress tree directly. In all other nodes, we store the difference of the minimum in the node and in its parent. Summing these values while descending from root, we can find out the required minima.

When splitting the path, the values of (4.1) in the subtree containing the original bottom stay the same and the values in the other subtree all have to be lowered by the same additive constant (the weight of the first subtree). We can do it by changing the minimum in the new root of the tree. There are also minor local changes of minima while restructuring the tree. Joining two paths works similarly.

In fact, we need to store two values in each node to be able to reverse the path – one for each path direction. Both of them have to be being updated, but only one is being read while seeking.

We will use *(decomposition) path* in the meaning of the corresponding compress tree or its root. A vertex on a path is determined either by the rake tree grouping paths with their tops there if it exists, or by the inner node of the compress tree representing compression of the vertex; we use it in the context of splitting and joining compress trees, in which we usually don't need to refer to the boundary vertices of paths.

5. The Driver

In the previous section, we have described independent compress trees and rake trees; here we analyze the composed top tree as a whole and describe detailed versions of the operations splice, symmetrize, reroot, link, and cut.

Claim 6. *The depth of a top tree of an underlying tree with n edges is $\mathcal{O}(\log n)$.*

Proof. For any base cluster x, let us denote $x_0, \ldots, x_{t-1}$ roots of the compress and rake trees on the path in the top tree from its root x_0 to the leaf $x = x_t$, and $W = w_0, \ldots, w_t = 1$ the corresponding weights. For a compress or rake tree with the root x_i the depth of its leaf x_{i+1} is $\mathcal{O}(\log(w_i/w_{i+1}) + 1)$ according to Claim 5. The number t of such trees is $O(\log n)$ by Claim 2. We can thus derive the following upper bound on the depth:

$$\mathcal{O}\left(\sum_{i=0}^{t-1}\left(\log\frac{w_i}{w_{i+1}}+1\right)\right) = \mathcal{O}\left(\log\frac{w_0}{w_t}+t\right) = \mathcal{O}(\log n).$$

The bound holds for a tree in its non-exposed form; after exposing, the depth can be at most thrice as large according to Claim 1 and every such expose has to be unrolled before other expose. □

5.1 Splice

The splice routine takes three arguments: A compress tree T of the original path, a rake tree or an inner node R in T determining the vertex of splice on the path, and a leaf S in R representing the new continuation of the path; S may be nil if R should be the bottom the of the new path. We split the path T at R into the bottom subpath T_1 and the top subpath T_2, replace S with T_1 in R, and join the paths S and T_2:

splice(top nodes T, R, S):

1. T_1, R, T_2 $\leftarrow$ compress_split(T, R)
2. R_1, S, R_2 $\leftarrow$ rake_split(R, S)
3. R' $\leftarrow$ rake_join(R_1, nil, R_2)
4. R'_1, nil, R'_2 $\leftarrow$ rake_split(R', T_1)
5. R'' $\leftarrow$ rake_join(R'_1, T_1, R'_2)
6. T' $\leftarrow$ compress_join(S, R'', T_2)
7. T, R $\leftarrow T', R''$
8. Return T_1.

We need to perform split and join twice in the rake tree in order to keep its leaves sorted. We therefore do the two-way join in Step 3 followed by the two-way split, which can divide the tree elsewhere. We remind that the split in Step 4 is performed just between the leaves where T_1 belongs according to its weight.

Lemma 7. *The time complexity of splice excluding calls of user-defined routines and the number of the calls are*

$$\mathcal{O}\left(\log\frac{w(T)}{\min(w(S), w(T_1))}+1\right)$$

in a general case;

$$\mathcal{O}\left(\log \frac{w(T)}{w(S)} + 1\right) \text{ or } \mathcal{O}\left(\log \frac{w(T)}{w(T_1)} + 1\right)$$

if T_1 or S is missing, respectively; and

$$\mathcal{O}(\log w(T) + 1)$$

if the rake tree is missing before or after the operation.

Proof. In the general case, each split or join costs $\mathcal{O}(\log(w_A/w_B) + 1)$, where w_A is the weight of the whole tree A (before splitting or after joining) and w_B is the weight of the node B we are splitting or joining at. It holds by Claim 5 if B is a leaf in A. For the other case, the same claim provides an upper bound with w_B replaced by the sum of the adjacent leaves we are splitting or joining between; one of them has a higher weight than B, since this happens only in weight-ordered rake trees. We can therefore use the same upper bound with w_B. If there is no leaf with a higher weight than B, then no split/join is needed and we recognize it in the root by discovering the maximum weight of its leaf descendant.

The weights of the tree T before splice and T' after it are equal and higher than weights of all other nodes. The complexity is

$$\begin{aligned}\mathcal{O}\left(\log \frac{w(T)}{w(R)} + \log \frac{w(R)}{w(S)} + \log \frac{w(R')}{w(S)} + \log \frac{w(R')}{w(T_1)} + \log \frac{w(R'')}{w(T_1)} + \log \frac{w(T')}{w(R'')} + 1\right) \\ = \mathcal{O}\left(\log \frac{w(T)}{w(S)} + \log \frac{w(T')}{w(T_1)} + 1\right) = \mathcal{O}\left(\log \frac{w(T)}{\min(w(S), w(T_1))} + 1\right).\end{aligned}$$

In the first equality, we have summed the first two logarithms and the last two logarithms; their sum is the upper bound on the sum of the omitted middle two logarithms.

If S is missing, T_1 should be moved into the rake tree, which is kept at the bottom of the path. Steps 2 and 3 have no effect. The complexity is

$$\mathcal{O}\left(\log \frac{w(T)}{w(R)} + \log \frac{w(R)}{w(T_1)} + \log \frac{w(R'')}{w(T_1)} + \log \frac{w(T')}{w(R'')} + 1\right) = \mathcal{O}\left(\log \frac{w(T)}{w(T_1)} + 1\right).$$

If T_1 is missing, Steps 4 and 5 have no effect, which is symmetric to the previous case. The complexity is

$$\mathcal{O}\left(\log \frac{w(T)}{w(S)} + 1\right).$$

If both S and the rake tree are missing (R is now an inner node of T), the rake tree should be created with the only leaf T_1. Step 1 acts as two-way split, which costs $\mathcal{O}(\log w(T) + 1)$; Step 6 costs at most the same. Steps 2, 3, and 4 have no effect; Step 5 just creates a new one-node tree. The complexity is

$$\mathcal{O}(\log w(T) + 1).$$

The case with missing T_1 and S being the only leaf in R, is symmetric to the previous one and has the same time complexity.

□

5.2 Reroot and Symmetrize

In reroot, we need to distinguish three cases: Either the new root v is already a boundary vertex of the main path; or it is the bottom of another path, which we have to find in a rake tree; or it is an inner vertex of a path and we need to find the corresponding rake tree, if it exists. The pseudocode follows:

```
reroot(vertex v):
1   T ← top_node(v)
2   If v ∈ top_boundary(T):
3     compress_reverse(T) if compress_top(T) ≠ v
4   Else:
5     If T is a rake node:
6       S ← the child of T such that v is its boundary vertex
7       While S is not a rake tree leaf:
8         S ← the child of S such that v is its boundary vertex
9       R ← rake_root(S)
10      T ← compress_root(R)
11    Else:
12      R ← T
13      T ← compress_root(T) if T is not a compress tree root
14      For each child R' of R:
15        While R' is not a compress tree leaf:
16          R' ← a child of R' with v in its boundary, prefer rake tree root
17        If R' is a rake tree root:
18          R ← R'
19          Break.
20      S ← nil
21    While T is not the main path:
22      splice(T, R, S)
23      S ← T
24      R ← rake_root(S)
25      T ← compress_root(R)
26    If the bottom subpath of T divided at R is heavier than the top one:
27      compress_reverse(T)
28    splice(T, R, S)
29    compress_reverse(T)
```

In the first case, v is already an endpoint of the main path; we may only need to reverse it in Step 3.

In the other cases, splices are necessary and we need to find the initial T, R, S triple for the bottommost splice, then we will perform splices traversing the tree up in Step 21.

In the second case, top_node(v) is a rake node (in Step 5) and v is the bottom of a path in the corresponding rake tree. We need to traverse down the rake tree to find the path and assign it to S, then it is easy to find R and T. The first hop in Step 6 is unconditional, because T might be an inner node of a compress tree; after it, S has to be a node in the corresponding rake tree.

In the third case (the branch in Step 11), v is an inner vertex on a path. We thus need to cut the path by the first splice, so S has to be nil. The top_node(v) is now an inner node of a compress tree. We need to find the root of that tree, keeping in mind that if T is already a compress root, compress_root(T) may lead to a wrong compress tree with T as a leaf. Next, we have to check whether there exists a rake tree R corresponding to the vertex v, otherwise R stays the inner node. We find the rake tree by traversing down the compress tree up to the at most three adjacent leaves with v in their boundaries, the rake tree root has to be one of them if it exists. The ambiguity in Step 16 occurs just before reaching the rake tree root.

Lemma 8. *The time complexity of* reroot *excluding calls of user-defined routines and the number of the calls are logarithmic w.r.t. the number of edges in the underlying tree.*

Proof. The ith splice takes $\mathcal{O}(\log(w(T_i)/w(S_i)) + 1)$ by Lemma 7, because S_i cannot be heavier than the current path continuation; the only exception may be the bottommost one taking $\mathcal{O}(\log w(T_i) + 1)$. The number of splices is at most logarithmic by Claim 4 and $S_{i+1} = T_i$; they all thus sum to $\mathcal{O}(\log n)$, where n is the number of base clusters in the top tree. While traversing the top tree, we change the direction only constant number of times, so the complexity of them is also bounded by the depth of the tree. □

The implementation of symmetrize is straightforward:

```
symmetrize(vertex v):
1   T ← top_root(top_node(v))
2   compress_reverse(T) if v ≠ compress_bottom(T)
3   Repeat:
4     R ← compress_heavier(T)
5     Break if R = nil.
6     S ← rake_heaviest(R)
7     T ← splice(T, R, S)
```

We first set the original root v to be the bottom of the main path; then we traverse the tree downwards repeatedly fixing the path with v to be a correct path of a decomposition rooted at the new top of the main path.

Lemma 9. *The time complexity of* symmetrize *excluding calls of user-defined routines and the number of the calls are logarithmic w.r.t. the number of edges in the underlying tree.*

Proof. The complexity of Step 1 is bounded by the depth of the top tree.

In Step 7, we always take a heavier subpath from the rake tree R and place lighter one there, which is then returned from the splice; its time complexity is thus $\mathcal{O}(\log(w(T)/w(T_{\text{returned}})) + 1)$, except for the last splice, whose complexity may be $\mathcal{O}(\log w(T) + 1)$. They sum to $\mathcal{O}(\log n)$, where n is the number of base clusters in the top tree.

Steps 4 and 6 take $\mathcal{O}(\log(w(T)/w(R)))$ and $\mathcal{O}(\log(w(R)/w(S)))$, respectively, which sum to at most the complexity of the following splice. In the case of breaking the cycle, compress_heavier has constant complexity. □

5.3 Link and Cut

The routines *link* and *cut* are very similar to their previous versions in Section 2.3:

```
link(vertices u, v):
1   reroot(u)
2   reroot(v)
3   P_1 ← top_root(top_node(u))
4   P_2 ← top_root(top_node(v))
5   If w(P_1) < w(P_2):
6     Swap u ↔ v, P_1 ↔ P_2.
7   compress_reverse(P_2)
8   e ← a new edge uv
9   P ← compress_join(P_1, e, P_2)
10  symmetrize(compress_top(P))
```

```
cut(vertices u, v):
1   e ← the edge uv as a leaf of a compress tree:
2     C ← top_node(u) or top_node(v) with the other vertex in the boundary
3     While C is not a base node:
4       C ← the child of C with the boundary {u, v}
5     e ← C
6   P ← compress_root(e) if e is not a compress tree root
7   If P is not the main path:
8     reroot(compress_bottom(P))
9     P ← compress_root(e)
10  compress_split(P, e)
11  symmetrize(u)
12  symmetrize(v)
```

The only possibly unclear step is seeking the base node in cut. Let us begin at the base node and walk upwards. At some point, one of the vertices disappears from the boundary while the other preserves; that node is the one we obtain in Step 2. We just go in the opposite direction in the algorithm. The other possibility is that the wanted edge is the only one in the tree and both top_node(u) and top_node(v) lead directly to the base cluster.

Claim 10. *The time complexity of link and cut excluding calls of user-defined routines and the number of the calls are $\mathcal{O}(\log n)$, where n is the number of edges in the underlying tree.*

Proof. The cost of traversing the tree is bounded by its depth. The other non-trivial steps take at most logarithmic time and require at most logarithmic number of calls of user-defined routines by Lemmas 8 and 9 and Claim 5. □

5.4 Linear Time Construction

Finally, we present a linear time construction of the top tree.

In Definition 4, we have described creation of a rooted path decomposition and according to the proof of Claim 4, we can easily create a symmetric one. By Claim 5, we can construct a biased tree in linear time by inserting the elements in the required order.

The basic idea is the following: We choose an arbitrary vertex and simulate creation of a main path rooted there to obtain the other endpoint, the real root. Then we can start creation of the symmetric path decomposition, but we proceed recursively; i.e., we always process first the lighter edges and their paths to create the whole rake tree for the insertion into the compress tree.

The only difficulty is sorting the leaves of the rake trees by their weights not to break linear time of the construction. To accomplish this, we use bucket sorting of all the vertices in one round to create a sorted list of children for each vertex: For n edges in the tree, we create $n + 1$ buckets. Then we perform depth-first search from the root adding the vertices in post-order to the buckets according to the weights of their subtrees. Finally, we take the vertices from the buckets one by one ascending by their weights and add each of them to the list of children of its parent.

Here is the whole algorithm:

```
construct:
1   v ← an arbitrary vertex of the tree
2   Calculate weights of all subtrees via DFS from v.
3   Repeat:
4     v ← the other endpoint of an edge from v to the heaviest subtree
5     Break if v is a leaf.
6   r ← v
7   bucket(0), ..., bucket(n) ← empty lists
8   DFS from r:
9     pre-order (v ← the current vertex):
10      parent(v) ← the vertex we came from
11      children(v) ← an empty list
12      w(v) ← 0
13    post-order:
14      For each neighbour v' of v except parent(v):
15        w(v) ← w(v) + w(v') + 1
16      Add v to bucket(w(v)).
17  For i ← 0, ..., n:
18    For each vertex v in bucket(i):
19      Add v at the end of children(parent(v)).
We continue by the recursive part on the following page...
```

```
... we continue construct having the weight-ordered lists of children:
20  P ← a new compress tree
21  Complete the path P as rooted at v ← r:
22    While children(v) is not empty:
23      If |children(v)| > 1:
24        R ← a new rake tree
25        While |children(v)| > 1:
26          v′ ← pop the first item from children(v)
27          P′ ← a new compress tree
28          Insert vv′-edge as a new base node at the end of P′.
29          Complete recursively path P′ as rooted at v′.
30          Insert P′ at the end of R.
31        Insert R at the end of P.
32      v′ ← pop the only item from children(v)
33      Insert vv′-edge as a new base node at the end of P.
34      v ← v′
35  Return P.
```

Claim 11. *The time complexity of* construct *excluding calls of user-defined routines and the number of the calls are $\mathcal{O}(n)$, where n is the number of edges in the underlying tree.*

Proof. Both depth-first searches take linear time. The number of the buckets is also linear and so costs of their initialization and reading. By walking in the tree, we visit every edge only constant number of times. All the operations of inserting nodes into biased trees summed can be done in linear time by Claim 5 with linear number of calls of user-defined routines. □

5.5 Final Remarks

In this thesis, we have assumed that a top tree is directly composed out of compress and rake trees. It is possible, but we need a way of handling expose and non-local search operations. They temporarily modify the structure of the top tree breaking our decomposition of compress and rake trees; we need to be able to repair them while restoring the original structure. The other difficulty is in ordering calls of the user-defined routines split, destroy, create, join, as mentioned in Section 1. The order presented in the algorithms is more straightforward and so easier to understand, but different to the required one.

A simple solution to handle these is having two different structures: one consisting of rake and compress trees and the other representing the top tree. Changes in the first one can be easily applied to the second one in the right order and temporal changes made in the second one will not influence the first one.

Another solution may be obtained by rearranging the algorithms presented here to correspond to the required order of events and by being careful while performing and reverting expose and non-local search.

Conclusion

We have described a top tree driver based on biased trees and proved its logarithmic depth and time complexities in the worst-case. There is now a room for its experimental comparison with the other existing drivers, primarily with the Werneck's worst-case one. We will mention some facets of the driver in comparison with some other drivers, namely with the original driver of Alstrup et al. [AHLT05] based on Frederickson's topology trees [Fre83], the worst-case driver of Renato Werneck [Wer06], and the amortized self-adjusting driver of Tarjan and Werneck [TW05].

A top tree composed out of our compress and rake trees is somewhat more restrictive than by the original definition. For example, if we have a rake tree with two paths of very different lengths, it may be better to rake the lighter one deeper in the top tree; in our driver, roots of both compress trees have to be in the same depth. We also restrict the order of leaves in the rake trees. Sorting the leaves by their weights can be viewed as an attempt to make the tree structure similar to the Huffman tree, having optimal depths of leaves; it would however need more sorting steps to reach it.

On the other hand, the restriction is still a little lower than in the amortized driver, which requires roots of rake trees to be raked onto another nodes just before the compression of the corresponding vertex. In our driver, the depth of the rake tree is determined by biased tree and can be arbitrarily higher than that of the compress node.

Our driver also doesn't need any preprocessing of the tree nor the transformation of the internal data structure into the top tree. A typical example of such a preprocessing is ternarization – lowering the degree of the vertices in graph to at most three –, which is needed in Frederickson's topology trees. The amortized top tree driver internally uses compress trees with vertices of degree up to four, which has to be transformed into binary trees. And the Werneck's worst-case driver uses so called *dummy* nodes with only one child, which has to be skipped while working with the top tree. In comparison, the composition of our compress and rake trees exactly corresponds to the structure of the top tree.

Top tree in its original definition doesn't maintain a circular order of the edges leading from a given vertex. It is, however, often required by a driver to choose and maintain such an order. Frederickson's topology trees order the edges in ternarization, Tarjan and Werneck's amortized driver preserves the order in the rake trees, and Werneck's worst-case driver maintain it via Euler tours. In our driver, the order is determined by the weights of the subtrees and so it cannot maintain an explicitly specified circular order.

The main aim of our driver is to serve as an alternative to the Werneck's worst-case one. The Werneck's driver creates a top tree by levels and needs dummy nodes, where a cluster hasn't changed between two consecutive levels; in addition, each level has to maintain an Euler tour in the correspondingly contracted tree, which is also a little overhead. Even though, the driver seems to be relatively straightforward compared to ours. The complexity of our driver is partially hidden in the biased trees, which we haven't described in detail. We need several splits and joins per splice, all of them maintaining also the weights

and their different sums in nodes. The main idea of both drivers is, however, totally different and so it is difficult to compare them only theoretically.

The experimental comparison should measure several different properties: the depth of the resulting top trees, which affects the complexity of expose and non-local search in the number of calls of user-defined routines; the number of such calls during link and cut; and the actual complexity of these two operations.

Bibliography

[AHLT05] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, October 2005.

[BST85] Samuel W Bent, Daniel D Sleator, and Robert E Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.

[Fre83] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 252–257, New York, NY, USA, 1983. ACM.

[Set18] Jiří Setnička. Comparison of top trees implementations. Master's thesis, Faculty of Mathematics and Physics, Charles University, 2018.

[ST81] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 114–122, New York, NY, USA, 1981. ACM.

[TW05] Robert E. Tarjan and Renato F. Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 813–822, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[TW10] Robert E. Tarjan and Renato F. Werneck. Dynamic trees in practice. *J. Exp. Algorithmics*, 14:5:4.5–5:4.23, January 2010.

[Wer06] Renato F. Werneck. *Design and Analysis of Data Structures for Dynamic Trees*. PhD thesis, Princeton University, Princeton, NJ, USA, 2006. AAI3214596.

List of Figures