Self-Adjusting Top Trees*

Robert E. Tarjan† Renato F. Werneck‡

Abstract

The dynamic trees problem is that of maintaining a forest that changes over time through edge insertions and deletions. We can associate data with vertices or edges, and manipulate this data individually or in bulk, with operations that deal with whole paths or trees. Efficient solutions to this problem have numerous applications, particularly in algorithms for network flows and dynamic graphs in general. Several data structures capable of logarithmic-time dynamic tree operations have been proposed. The first was Sleator and Tarjan's ST-tree [16, 17], which represents a partition of the tree into paths. Although reasonably fast in practice, adapting ST-trees to different applications is nontrivial. Topology trees [9], top trees [3], and RC-trees [1] are based on tree contractions: they progressively combine vertices or edges to obtain a hierarchical representation of the tree. This approach is more flexible in theory, but all known implementations assume the trees have bounded degree; arbitrary trees are supported only after ternarization. We show how these two approaches can be combined (with very little overhead) to produce a data structure that is as generic as any other, very easy to adapt, and as practical as ST-trees.

1 Introduction

Consider the following problem. We are given an n-vertex forest of rooted trees with costs on edges. Its structure can be modified by two basic operations: $link(v, w, c)$ adds an edge with cost c between a root v and a vertex w in a different component; $cut(v)$ removes the edge between v and its parent. At any time, we want to be able to find, for any vertex v, its parent $p(v)$ and the cost of the edge $(v, p(v))$. All these operations take constant time with an obvious implementation: for each vertex v, store a pointer to its parent and the cost of the edge between them.

Now suppose we also want to find the cheapest edge on the path from a vertex v to the root, or to add a constant c to the cost of every edge on this path. The obvious implementation can support these operations, but in time proportional to the length of the path, which could be $\Theta(n)$.

This specific problem appears in the context of network flow algorithms [16]. We are interested in its generalized version: a data structure that supports in $O(\log n)$ time queries and updates related to vertices and edges individually, and to entire trees or paths. We call this the *dynamic trees problem*. Other typical operations include adding a certain value to all vertices in a tree, or asking for the sum of all edge weights on a path. Operations such as these are needed in several solutions to the maximum flow problem [2, 10, 11, 19] and related algorithms [16]. They are also used in algorithms that maintain properties of dynamic graphs, such as minimum spanning trees and connectivity [3, 8, 12, 14]. Applications for maintaining dynamic expression trees have also been reported [6, 9].

The first data structure to support every operation in the example application in $O(\log n)$ time was Sleator and Tarjan's ST-tree [16] (also known as *link-cut tree*). This structure partitions the tree into vertex-disjoint paths and represents each one by a binary tree in which the original vertices appear in symmetric order. The binary trees are then glued together according to how the paths are connected (the root of each binary tree becomes a *middle child* of a node in another binary tree). For the algorithm to be efficient, this hierarchy must be balanced. But making each binary tree balanced is not enough—the total height of the hierarchy would be $O(\log^2 n)$. Sleator and Tarjan have shown that using globally biased search trees [5] one does achieve $O(\log n)$ worst-case time per operation. They later showed [17] how splaying greatly simplifies the data structure while still achieving the $O(\log n)$ bound (now amortized).

ST-trees can be adapted to solve other problems beyond our example, but this requires an understanding of their inner workings. In particular, Goldberg et al. [10] show how subtree-related operations (such as adding a value to all vertices in a tree) can be accomplished with an implicit ternarization of the original tree, which transforms high-degree vertices into chains of constant-degree ones. The data structure becomes more complicated, however.

*Research at Princeton University partially supported by the Aladdin project, NSF Grant no. CCR-9626862.

†Department of Computer Science, Princeton University, and HP Labs. E-mail: ret@cs.princeton.edu.

‡Department of Computer Science, Princeton University. E-mail: rwerneck@cs.princeton.edu.

A simpler and more elegant way to handle subtree-related operations stems from the observation that a tree can be represented by an Euler tour. Representing tours as standard balanced binary trees is the basis of ET-trees, proposed by Henzinger and King [12], and later simplified by Tarjan [19]. Unfortunately, these data structures cannot deal with path-related operations (such as the ones suggested in our example), so their applications are somewhat limited.

A third class of data structures is based on *tree contractions*. These structures use two operations proposed by Miller and Reif [15] in the context of parallel algorithms: *rake* (which removes leaves) and *compress* (which removes vertices of degree two). Each operation replaces the original elements (vertices and edges) by a *cluster* that aggregates information about them. The entire tree is represented by a hierarchy of clusters, which is itself a tree.

In Frederickson's *topology trees* [7, 9], the contraction works in rounds, each with a maximal set of independent rakes and compresses. Since the tree shrinks by a constant factor in each round, there are at most $O(\log n)$ rounds. More importantly, the contraction can be updated after a *link* or *cut* in $O(\log n)$ worst-case time. However, the data structure is somewhat involved, since it must maintain one tree for each level as well as the connections between these trees. In practical applications, this makes it twice as slow as ST-trees [9]. Recently, Acar et al. proposed *RC-trees* [1], a randomized variant that is conceptually simpler and runs in $O(\log n)$ expected time per operation. Both data structures view clusters as vertices, which, for technical reasons, means that the $O(\log n)$ bound only applies to trees of bounded degree. Arbitrary trees can be handled by ternarization, but this increases the tree size and adds an extra level of complexity.

An alternative is *top trees*, proposed by Alstrup et al. [3, 4]. By considering clusters to be edges (instead of vertices), they avoid the need for explicit ternarization. In addition, they provide an interface for handling data independently of the order in which rakes and compresses are performed, so one can adapt this data structure to different applications without modifying its inner workings (a similar interface was also defined for RC-trees). This not only simplifies the implementation of existing algorithms for various applications, but also makes it easier to devise new ones. In [3], however, the suggested implementation of top trees is as a layer on top of topology trees, hardly a practical solution.[1]

[1]Holm and de Lichtenberg [13] did suggest a direct implementation of top trees, but they later found their run-time analysis to be flawed (personal communication). Even if the bound is correct, the implementation is far from trivial.

In a very broad sense, all these data structures have the same ultimate goal: to map an arbitrary tree into a balanced one. ET-trees do it in a very elegant, direct way, but they cannot deal with path-related operations. ST-trees represent individual paths as binary trees, which are then glued together to represent the whole tree. This approach is ideal for path-related operations, but handling subtree queries requires ternarization. Topology trees and RC-trees represent not the tree itself, but the steps necessary to contract it. This can be viewed as a multi-level decomposition of the original tree, which lends itself naturally to applications related to dynamic graphs. These two data structures, however, can only deal with trees of bounded degree. Top trees eliminate this constraint and have the most natural interface, but they achieve this by adding an extra layer to topology trees that merely hides the ternarization. Devising a data structure that is at the same time generic, flexible, and practical has been an elusive goal.

This paper achieves this goal. We show how the principles behind Sleator and Tarjan's ST-trees can be used to implement top trees. A partition of the original free tree into edge-disjoint paths can be directly mapped onto a series of rakes and compresses, which shows that partitions and contractions are essentially equivalent. The end result is a data structure that is almost as streamlined as the original ST-trees, but as flexible as top trees (with the extra ability to handle ordered edges around each vertex).

The paper is organized as follows. Section 2 formalizes the problem and outlines the top tree interface. Our representation is described in Section 3. Section 4 shows how queries and updates are handled. Section 5 establishes the $O(\log n)$ amortized time per operation. Final remarks are made in Section 6.

2 Top Trees

Consider a collection of free (unrooted, undirected) trees whose edges are organized in circular order around each vertex. The order can be arbitrary or given by the application. We interpret compress and rake as follows. A degree-two vertex v is *compressed* if the two edges incident to it, (u, v) and (v, w), are replaced by a single edge (u, w). A degree-one vertex v with neighbor x is *raked* if the edge (v, x) and its successor (w, x) around x are replaced by a single edge, also with endpoints w and x. (We also say that the edge (v, x) is raked onto (w, x).) See Figure 1. We note that our interpretation differs slightly from the one proposed by Alstrup et al., which assumes that there is no order among the edges incident to the same vertex.

A *top tree* is a binary tree that embodies a contraction of a tree into a single edge via a sequence of

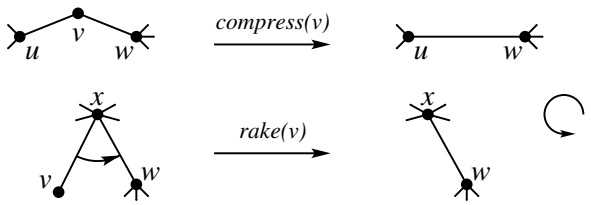

Figure 1: Basic operations.

rake and compress operations. Rakes and compresses are viewed as manipulating clusters. Each leaf of the top tree is a *base cluster* representing an original edge, and each internal node is either a rake cluster or a compress cluster. A node aggregates information pertaining to all descendants; in particular, the entire original tree is represented at the root of the top tree. When an edge is deleted or inserted, there is no need to recompute a new contraction from scratch: it is enough to update the affected top trees to make them consistent with the new underlying forest. Since changes to the leaves propagate to the root, only sequences of rakes and compresses that produce *balanced* top trees can provide an $O(\log n)$ solution to the dynamic trees problem.

Alstrup et al. [3] show that top trees can maintain a variety of properties efficiently (such as shortest paths between any two vertices, tree center, and tree diameter) as long as the appropriate information is kept in each cluster. Quite sensibly, they suggest making the structural changes independent of the data. Regardless of the order of rakes and compresses, the data structure just calls four user-defined functions to update values appropriately: JOIN, SPLIT, CREATE, and DESTROY. The first two are called when a rake or compress is performed or undone; the last two are the equivalents for base clusters. All four functions deal only with root clusters.

For example, to compute the sum of the costs of all edges in a tree, we store a single value in each cluster. CREATE initializes this value as the cost of the corresponding edge. JOIN stores in the new cluster the sum of the values in the children. Because children always keep their original values, SPLIT need not do anything. DESTROY also does nothing.

The user does not call any of these four functions directly. She is limited to three basic operations: $link(v, w)$ adds an edge to the forest; $cut(v, w)$ removes an edge; and $expose(v, w)$ ensures that v and w are endpoints of the root cluster, the only cluster the user can manipulate directly. Our goal is to implement these three operations efficiently.

3 Representation

To represent a free tree, we first pick a degree-one vertex as the *root* and direct all edges towards it. We call this (a directed tree whose root has degree one) a *unit tree*. We then partition the tree into non-crossing edge-disjoint paths that begin at a leaf and end at another path. The only exception is the *root path* (or *exposed path*), which ends at the root. (A complete example illustrating this and other aspects of the representation is given in the appendix.)

Our goal is to create a cluster to represent the whole unit tree. Any internal vertex v of the root path p has exactly two neighbors on the path and zero or more *outer neighbors*. Since edges around a vertex are arranged in circular order, the outer neighbors of v are divided by p into two (possibly empty) subsequences (see Figure 2). Each element of these subsequences is a unit tree rooted at v, and therefore can be recursively represented by a single cluster. Clusters in the same subsequence are progressively paired up to create a *rake tree*: its root represents the entire subsequence, leaves represent unit trees, and each internal node is the rake of the left onto the right child.

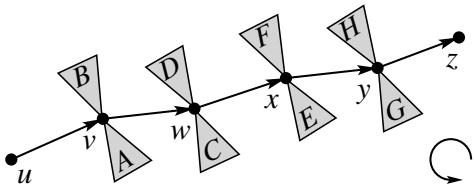

Figure 2: A unit tree rooted at z.

We are now left with a path containing some k base clusters and at most $2k-2$ incident *outer clusters*, each representing a subtree (empty subtrees will have no associated cluster). Ignoring the outer clusters, we could represent the path by a *compress tree*, a binary tree whose leaves are base clusters, and whose internal nodes represent compresses of adjacent clusters. Each node in the compress tree represents a subpath of the original path (leaves represent original edges, internal nodes represent nontrivial paths).

We deal with the outer clusters by raking them onto the root path. This is done as late as possible: an outer cluster incident to vertex v is raked just before v is compressed. In the data structure, it will become a *foster child* of the node representing $compress(v)$ (the two original children are *proper children*). The left foster child is raked onto the proper left child, and the right foster child onto the proper right child. The resulting clusters are then compressed.

Figure 3 is a possible representation of the unit tree in Figure 2. Shaded nodes belong to the compress tree.

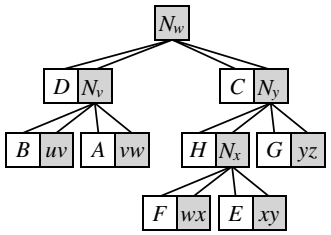

Figure 3: A top tree corresponding to Figure 2.

Internal nodes are labeled after the vertices compressed (e.g., N_y represents $compress(y)$). Each internal node has up to four children and represents at most three clusters: two rakes (one for each foster child) and one compress (of the clusters generated by the rakes). Foster children, shown in white, are actually binary (rake) trees whose leaves represent unit trees.

Summing up, we represent a unit tree as follows:

- Recursively compute clusters to represent each unit tree incident to the root path p.
- Create rake trees to represent each contiguous sequence of unit trees incident to p.
- Create a binary tree of compress nodes to represent the root path, with the rake trees appearing as foster children.

This method works for the entire tree, which is itself a unit tree. The end result is a hierarchy of alternating rake and compress trees.

Order within binary trees. We have seen that a rake tree represents a contiguous sequence of clusters in symmetric order. The leaves of a compress tree represent the edges of a path, and in principle could also appear in symmetric order. To handle path reversals efficiently, however, we use a more relaxed condition. Given a node representing $compress(v)$ with endpoints u and w, one of its subtrees must represent the path $u \cdots v$, and the other $v \cdots w$. Left and right subtrees can be interchanged freely.[2] (The "correct" order among children can be retrieved from the cluster endpoints.) ST-trees use a similar technique to support the *evert* (change root) operation.

Handles. Top tree operations (*link*, *cut*, and *expose*) are defined in terms of vertices, but the top tree itself is a hierarchy of clusters and nodes, which can be viewed as edges or paths—not vertices. Therefore, we associate to each vertex v a *handle* N_v. If the degree of v is at least two, N_v is the node representing $compress(v)$. If the degree is one, N_v is the topmost non-rake node that has v as an endpoint (this is either the root of the entire tree or a leaf of a rake tree). Isolated vertices have no handle. A node may be the handle of as many as three vertices; in Figure 3, for example, the root (N_w) is the handle of u, w, and z. The map from vertices to handles is maintained explicitly.

[2] The example in the appendix illustrates this.

4 Updates

Before operating on a path, the top tree interface mandates that we first *expose* it, i.e., make it represented at the root node. The representation described in Section 3 requires both endpoints of the root path to have degree one. To handle an arbitrary path $v \cdots w$, we first pick a root path that contains $v \cdots w$ as a subpath, then we temporarily convert up to two compress nodes into rake nodes. We call the first step a *soft expose* of vertices v and w, and the second a *hard expose* of the path $v \cdots w$. We discuss each in turn, in Sections 4.1 and 4.2. Cuts and links are discussed in Sections 4.3 and 4.4, and additional implementation issues in Section 4.5.

4.1 Soft Expose. The outcome of $soft\_expose(v, w)$ depends on how v and w relate. If the vertices are isolated, nothing is done. If $v = w$ or v and w are in different components, N_v (v's handle) and N_w (w's handle) are simply brought to the root of their components. When v and w are different vertices in the same component, $soft\_expose(v, w)$ ensures that cluster vw (representing the path $v \cdots w$) is close to the root of the top tree. It works by first making N_w the root, then bringing N_v as close to the root as possible (preserving N_w). When both v and w have degree two or more, all three nodes (N_w, N_v, and vw) are different, and the outcome is the one depicted in Figure 4. In degenerate cases, we may have $N_w = N_v$ or $N_v = vw$ (or both). To simplify hard expose, we require both N_v and vw to be right children (unless they coincide with the root N_w).

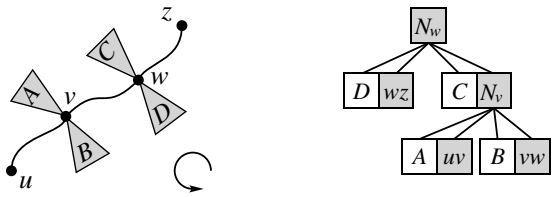

Figure 4: Configuration after $soft\_expose(v, w)$.

The *soft_expose* operation uses the same basic tools as the amortized version of ST-trees [17]: *splay* and *splice*.

Splaying. Splaying [17] is a heuristic for rebalancing binary trees using rotations. After a node x is accessed,

it is rotated up to the root. The precise nature of each rotation depends on the relative positions of x, its current parent p, and its current grandparent g. If x and p are both right (or both left) children (*zig-zig* case), we first rotate edge (p, g), then (x, p). If the edges alternate (*zig-zag* case) we rotate (x, p) first, then (x, g). When p is the root (*zig* case), we just rotate (p, x). In our case, rotations (and splaying) happen only within individual rake or compress trees. We therefore perform *guarded* splays, which stop when x becomes the child of a reference node (the *guard*). Ordinary splays are guarded by *null*.

Figure 5 shows a rotation within a compress tree. A key observation is that foster children are not affected: they always keep the same parents. Rotations in rake trees are similar (with no foster children). Splaying is performed for balancing purposes only: it just changes the order in which different moves of the same type occur, preserving the original partition into paths.

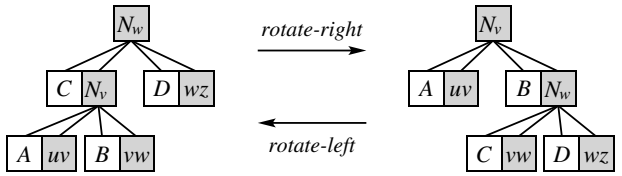

Figure 5: Rotation in compress trees.

Splaying requires the left and right children of all nodes visited to appear in a consistent (symmetric) order. Before splaying on a node N, we must therefore *rectify* all compress nodes in the path from N to the root of its top tree (rake nodes always have the correct order). If a node X has parent N_y and grandparent N_z, then X, N_y, and N_z must form a zig-zag if and only if the endpoints of X are y and z. We ensure this by flipping the children of N_y appropriately. Rectification also guarantees that every compress node it visits has the subpath that is farthest from the root represented in its left child. Rectification is performed in a top-down fashion. To preserve the circular order when proper children are flipped, we flip the foster children as well.

Splice. The operation that changes the partition of the original tree into paths is *splice*. A vertex v that is internal to a path divides this path into two segments. Splice replaces the segment that is farthest from the root with one of the outside paths incident to v. In Figure 6, $x \cdots v$ is replaced by $y \cdots v$.

Figure 7 shows a possible configuration of the corresponding top tree (circles represent internal nodes of rake trees). The original proper children of N_v (v's handle) are vx and vz, representing $x \cdots v$ and $v \cdots z$.

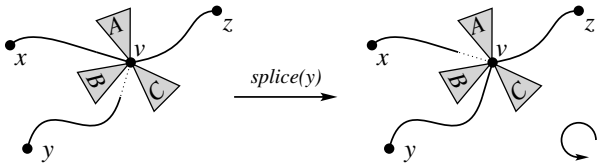

Figure 6: Splice: $y \cdots v \cdots z$ replaces $x \cdots v \cdots z$ as the exposed path.

We shall see that splices only occur after a series of local splays (within compress and rake trees). N_v will be the root of a compress tree, and there will be at most two rake nodes between vy and N_v. Splice makes vy the left child of N_v and incorporates the former left child (vx) into a rake tree, where it appears between A and B as required by the circular order.

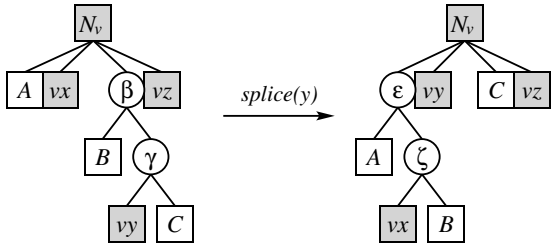

Figure 7: Splice: Top trees corresponding to Figure 6.

Figures 6 and 7 represent only the most generic out of several possible cases for splice. The precise outcome depends on which foster child contains vy (the subpath to be exposed) and on whether some of the rake subtrees (A, B, or C) are absent. We must always replace the left child of N_v (which represents the subpath that is farthest from the root) while ensuring that the circular order of the up to six relevant subtrees rooted at v (denoted by A, vx, B, vy, C, and vz in Figure 7) is preserved.

Exposing the target. Now that we have the necessary building blocks, we return to the implementation of *soft_expose*(v, w). Its first step is to expose the target vertex w, making its handle the root node.[3] The function starts from N_w itself, and works in three passes:

1. (Local Splay) Splay within each compress and rake subtree in the path from N_w to the root.
2. (Splice) Perform a series of splices from N_w to the root, making N_w part of the topmost compress subtree.
3. (Global Splay) Splay on N_w, making it the root of the entire tree.

[3] We refer to w as the "target" for convenience; the path $v \cdots w$ is actually undirected.

The last two passes are straightforward, so we only discuss the first in detail. It is divided into several subpasses, each starting from a different compress tree. Let N be a node of this tree (initially, $N = N_w$). Splay on it, making N the root of its compress tree and a leaf of a rake tree. If the rake tree contains other nodes besides N, splay on N's parent, P, within the rake tree. If N ends up with a new parent P', splay on P' with P as a guard. This concludes the subpass.

The left of Figure 7 shows a possible configuration after a subpass associated with node vy. It becomes the root of a compress tree (not shown), and between itself and the closest compress ancestor (N_v), there are at most two rake nodes (γ and β). In fact, we splay twice on the rake tree precisely to divide it into three subsequences: B, vy, and C. An alternative is to perform a special "splaying split" of the rake tree, which is similar to splaying on vy directly.

Once the subpass is concluded, we advance to the first compress ancestor of N and repeat the procedure there. We stop upon reaching the root of the entire tree.

Exposing the source. Now consider how to expose the source v. At this point, N_w will be the root. If v is an endpoint of N_w or if N_w represents *compress*(v), we are done: N_w is v's handle as well. Otherwise, we must bring v's handle (N_v) as close to the root as possible.

The exact procedure depends on the degree of w. If it is one, we apply to N_v the same three-pass procedure applied to N_w; this makes N_v the root, with w as one of its endpoints (N_v and N_w coincide). If w has degree two or more, then N_w will represent *compress*(w). To expose v, we apply to N_v a procedure similar to that applied to N_w, but with all splays guarded by N_w. This ensures that no node will replace N_w at the root, so either N_v will end up as N_w's child (when v has degree at least two) or v will become an endpoint of N_w (and $N_w = N_v$ will be the root).

To follow the specification of *soft_expose*, we may need to flip the children of N_w and N_v. If $N_v \neq N_w$, N_v must be N_w's right child; if the node representing $v \cdots w$ is different from N_v, it must be its right child. If v and w turn out to be in different components, the procedure above will end up exposing N_v as if it were the target, as required by the specification.

4.2 Hard Expose. In general, *soft_expose*(v, w) does not make v and w the endpoints of the root node. Instead, as Figure 4 shows, the root node will represent some path $u \cdots z$, with vw as the rightmost grandchild. To fix this, the *hard_expose* operation temporarily converts to rake the (at most two) compress ancestors of vw. In the figure, N_v and N_w would be affected. Before another pair of vertices is exposed, these modifications must be undone to bring the tree back to its "normalized" form.

4.3 Cuts. To cut an edge (v, w), we first execute *soft_expose*(v, w), making N_w the root. As Figure 8 shows, in the general case (when both v and w have degree at least two) N_w's right child will be N_v, and N_v's right child will be the base node representing (v, w).

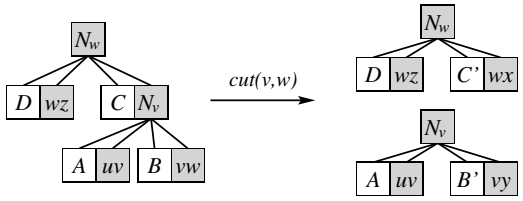

Figure 8: Cutting edge (v, w).

We must destroy the base node and reorganize the remaining nodes into two valid top trees. In the original top tree, N_w represents some unit tree with root path $u \cdots z$. The right subtree represents a unit tree rooted at w with (v, w) as the topmost edge. If we remove the link between N_v and N_w, N_w will be the root of a tree containing only the vertices in w's component. Similarly, if we remove the right child of N_v, only vertices in v's component will remain.

In both cases, the original right child must be replaced. We detail only how to process N_w; the case of N_v is similar. To preserve the circular order, the replacement must be either the immediate successor of the edge removed (the leftmost leaf of the left foster subtree of N_w) or the immediate predecessor (the rightmost leaf of the right foster subtree of N_w). To extract the appropriate leaf, we simply splay on its parent. If N_w has no foster child, it is deleted and its left child becomes the new root.

4.4 Links To insert an edge (v, w) as the successor of (a, v) around v and of (b, w) around w, we first perform *soft_expose*(a, v) and *soft_expose*(b, w).[4] If all vertices have degree greater than one, we will have the two top trees shown on the left of Figure 9. To link them, we do the opposite of cut: we first replace the right child of N_v with vw, making the old right child (N_a) the rightmost leaf of the right foster subtree of N_v; then we do the same for N_w, making N_v its new right child.

We follow similar procedures when v or w have degree zero or one. In particular, when v has degree one, it is an endpoint of N_v; to add (v, w) to the tree,

[4] $a \cdots v$ and $b \cdots w$ can also be arbitrary paths.

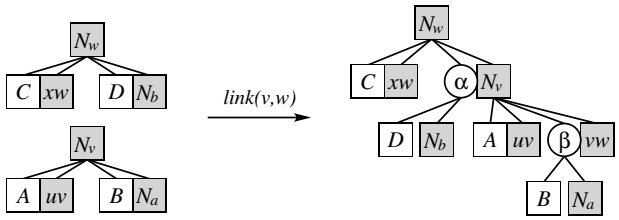

Figure 9: Linking v and w, general case.

we create a new compress node and make the old N_v and the base node representing (v, w) its children. A new compress node is also necessary when w has degree one.

4.5 Implementation Issues

Handling data. So far, we have discussed only structural changes to the top trees. To update the values in each cluster, we use the user-defined functions CREATE, DESTROY, JOIN, and SPLIT. Rotations and splices can be easily expressed in terms of a series of SPLITs and JOINs. Since the top tree is modified in a bottom-up fashion, these functions cannot be called as we go (recall that they can only be applied to root clusters). Instead, we first just mark all affected clusters, then SPLIT and DESTROY them in a top-down fashion. Only then do we perform all structural modifications. A simple recursive function unmarks all clusters and calls CREATE and JOIN in a bottom-up fashion.

Data on vertices. Since top tree clusters correspond to edges, representing edge-related information is trivial. In many applications, however, we must associate data with *vertices* instead. Alstrup et al. [3] suggest attaching to each vertex v a special edge (a *label*) to store vertex-related data; one of its endpoints is v, and the other a dummy vertex with no other incident edge. Although this approach is generic, adding extra edges is an undesirable overhead. For most applications, it is enough to keep the data associated with the vertices in a separate array to which the internal functions (JOIN, SPLIT, CREATE, and DESTROY) have access. Vertex information would be explicit for exposed vertices (i.e., those that are isolated or endpoints of root clusters), and implicit for internal ones; therefore, only the values of exposed vertices could be accessed directly.

Suppose, for example, that we want to maintain the total weight of the vertices in a tree. The auxiliary array keeps individual weights, while each cluster stores a single value w corresponding to the total weight of the subtree it represents. CREATE initializes this value as the sum of the weights of its endpoints, while JOIN sets it to the sum of the values in its children minus the weight of the disappearing vertex (to avoid double-counting).

Non-local search. Alstrup et al. [3] observed that certain applications (such as maintaining the tree median) require performing a binary search within the top tree to find an edge with some specific property. According to the top tree interface, however, the user should not be required to traverse the tree directly. Instead, the authors propose a routine that gradually transforms the original top tree into another, with the target edge represented at the root. A fifth user-defined internal function, SELECT, is used to guide this construction. The total running time is proportional to the original depth of the base node representing the target edge. As soon as the desired query is completed, the original tree is restored. This operation can be applied here, as long as it is followed by a call to $expose(v, w)$, necessary to amortize its cost appropriately.

Roots. The example application we presented in the introduction assumes the trees are rooted, whereas top trees deal with unrooted trees. It is not hard to see that these two variants are essentially equivalent. A data structure that deals primarily with rooted trees can deal with unrooted trees as long as it supports *evert*, an operation to change the root. This is the case for both ST-trees and topology trees. RC-trees and top trees, although implicitly rooting the tree, have interfaces that assume that the underlying tree is not rooted. To support rooted trees, we add to each cluster the label of the root of the tree that contains it; this value (as any other) is updated by the internal top tree operations.

Nodes and clusters. As described, our representation does not implement top trees directly. We have nodes with up to four children, whereas top trees are binary. For a direct implementation, it suffices to replace each compress node by the three corresponding clusters (one compress, two rakes). The splaying procedure must then be modified to account for the fact that compress trees now have interspersed rake nodes.

5 Analysis

The run-time analysis of our algorithm does not consider the top tree itself, but an equivalent *phantom tree*. In the top tree, compress nodes have up to four children; in the phantom tree, up to three (left, middle, and right). To convert a four-child top tree node to a phantom tree node, we create an *articulation node* (the new middle child) and make it the parent of the original

foster children. The articulation node is inserted only when there are two foster children. While a tree with n vertices may be represented by top trees of varying sizes, phantom trees will have exactly $n-1$ nodes; this greatly simplifies the analysis. Phantom trees are used in the analysis only; they need not be implemented.

We extend Sleator and Tarjan's analysis of ST-trees [17]. The *rank* of a node N is defined as $r(N) = \log s(N)$, where $s(N)$, the *size* of N, is the number of leaves descending from N. (Note that the rank is at most $\log n$.) The *potential* of the phantom tree is defined as q times the sum of the ranks of all nodes, where q is a constant to be chosen later. The *amortized cost* of the i-th operation in a sequence is defined as $a_i = c_i + \phi_{i+1} - \phi_i$, where c_i is the *actual cost* of the operation and ϕ_i and ϕ_{i+1} the potentials before and after it is performed. A bound on the total amortized time translates into a bound on the actual time [18].

In general, the operations we perform take an *active* node and move it upwards in the tree (the active node changes during splice). Each basic operation (rotation, double rotation, or splice) deals with a constant number of nodes, and therefore takes constant time. We define the actual cost of the i-th operation (c_i) as the amount by which the depth of the active node is reduced.

Rotations within rake and compress trees follow Sleator and Tarjan's analysis [17]. The amortized time for zig-zig or zig-zag on a node N is $3q(r'(N) - r(N))$, where $r(N)$ and $r'(N)$ denote the rank of N before and after the operation. A zig move has $3q(r'(N)-r(N))+1$ amortized cost.[5] As for splices, Figure 10 shows how they work on phantom trees: just as in Figure 7, with articulation nodes (α and δ) added where necessary. The active node is vy before the operation, and N_v after.

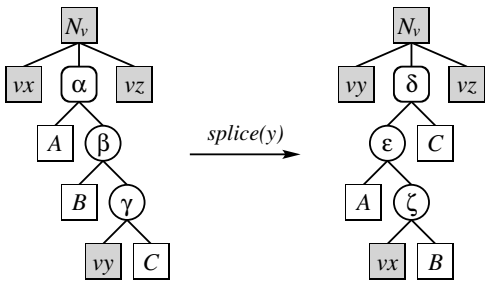

Figure 10: Splicing on the phantom tree corresponding to the tree on Figure 7.

LEMMA 5.1. *The amortized cost of a splice is at most* $3q(r'(N')-r(N))+4$, *where* N *is the active node before the operation, and* N' *the one after.*

[5] In Sleator and Tarjan's analysis of ST-trees, $q = 2$.

Proof. The actual cost of the operation is at most 4, an upper bound on the amount by which the depth of the active node is reduced (see Figure 10). The only nodes whose ranks change are those labeled with Greek letters in the figure. All three affected nodes on the left are ancestors of the original active node $N = vy$, so their combined rank is at least $3r(N)$; the affected nodes on the right are all descendants of the final active node $N' = N_v$, which means their combined rank is at most $3r'(N')$. The amortized cost a of the operation is

$$a = c + \phi' - \phi \leq 4 + q(3r'(N')) - q(3r(N)),$$

as claimed. A similar analysis holds if there are fewer than three nodes between N and N'. □

We now bound the amortized cost of *soft_expose*(v, w). It is enough to bound the time to expose the target vertex w; the same bound applies to the source v. We analyze each pass (local splays, splices, and global splay) in turn.

If k is the number of compress trees in the path from N_w to the root of the top tree, we will splay within k compress trees and up to $k-1$ rake trees (twice in each). Adding the amortized costs of the rotations, we can bound the total amortized cost of the splays by $6q \log n + (3k - 2)$. The first term results from two telescoping sums: one accounts for all rotations within compress trees and during "primary rake" splays, and the other for rotations during "secondary rake" splays. The additive term is an upper bound on the number of zig moves.

The second pass performs $k - 1$ splices. From Lemma 5.1, the total amortized cost is at most $3q \log n + 4(k-1)$ (the sum of ranks telescopes).

The splay on the third pass reduces the depth of the active node from $k-1$ to 0 with $k-1$ rotations. The total amortized cost of the step is $3q \log n + 1$. Our definition of potential charges q time units per rotation; since the actual cost is one, this leaves us $(q-1)(k-1)$ extra units. Setting $q = 8$, we will be only one unit short of fully paying for the extra $3k - 2$ rotations in the first pass and for the $4k - 4$ unpaid moves in the second pass.

This bounds the cost of the exposing the target vertex at $12q \log n + 2 = 96 \log n + 2 = O(\log n)$. The same applies to the source.

We claim both *link* and *cut* take $O(\log n)$ amortized time. This is certainly a lower bound, since both begin with *soft_expose*. *Link* goes on to perform a constant number of operations, all close to the root; this increases the potential by at most $O(\log n)$. *Cut* is slightly more complicated, since it performs one or two additional splays (in a foster subtree of each new root). Since

each splay takes $O(\log n)$ amortized time, the claim still holds. We have thus proved the following:

THEOREM 5.1. *Self-adjusting top trees support* link, cut, *and* expose *in* $O(\log n)$ *amortized time.*

6 Final Remarks

Our data structure demonstrates that the two main approaches used to represent dynamic trees are equivalent. ST-trees represent a partition of the trees into disjoint paths. Topology trees, RC-trees, and top trees are based on tree contraction. Frederickson [9] noticed that partitions and contractions have similarities, and Alstrup et al. [3] even showed that topology trees can be implemented using ST-trees. However, the transformation is far from direct, as the authors themselves observe.

We use a more direct mapping. Any sequence of rakes and compresses can be translated into a (unique) partition of the tree into edge-disjoint paths. Conversely, any partition into paths can be translated into a sequence of rakes and compresses, although not necessarily a unique one. In our case, tree edges will be in the same compress tree if and only if they belong to the same path in the partition. The result is a top tree representation with very little overhead, as a single binary tree with one node per cluster. Topology trees, in contrast, must represent each tree that results from a parallel set of rakes and compresses, as well as the relationships between these trees.

We have an early implementation of our data structure, and the results are encouraging. An implementation of the basic ST-tree interface [16] on top of it is roughly twice as slow as a direct (self-adjusting) implementation, which is remarkable considering how much more general our data structure is. It supports subtree operations on trees of unbounded degree and ordered incidence lists, and neither of these features is required by this particular application.

In such cases (when its full power is not needed), our data structure can indeed be simplified. When there is no order among the edges around each vertex, we can replace the two foster children of a compress node by a single middle child (as in phantom trees), which greatly simplifies splice, *cut*, and *link*. Further simplification is possible for applications (such as our initial example) where rakes have no effect on the target cluster. Then we can completely eliminate rake trees and link the root node of each compress tree directly to the compress tree above, mimicking the dashed edges of ST-trees [16]. In fact, one can think of our data structure as a generalized version of ST-trees.

We observe that the constant factors obtained in our analysis are rather large. Although a more careful analysis might reduce them, we are also investigating whether there is further room for simplification in the full version of the data structure. In particular, it might be interesting to relax some of the constraints imposed on the structure of the top tree, such as performing all rakes in a subsequence at once.

Another question is whether we can obtain other practical implementations of top trees. In particular, we are interested in a worst-case version. Alstrup et al. have shown that top trees can be implemented as a layer on top of topology trees [3]; this proves that an $O(\log n)$ worst-case solution does exist. Using some of the ideas in [13], we have devised (in joint work with S. Alstrup, J. Holm, and M. Thorup) an alternative version based on an explicit representation of the contraction: rakes and compresses are executed in rounds, and updates to the contraction happen in a bottom-up fashion. This implementation requires keeping one tree per level as well as the links between them, which makes it just as complicated as topology trees. Another approach would be to adapt the algorithms presented here to work with globally biased binary trees instead of splay trees, but this solution is unlikely to be practical. We are currently looking for something simpler. Finally, we are also considering randomization as a tool to simplify the data structure.

Acknowledgements. We thank Umut Acar, Guy Blelloch, and Mikkel Thorup for helpful discussions.

References

[1] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proceedings of the Fifteenth ACM-SIAM Symposium on Discrete Algorithms*, pages 524–533, 2004.

[2] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.

[3] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining diameter, center, and median of fully-dynamic trees with top trees. `http://arxiv.org/abs/cs/0310065`, 2003.

[4] S. Altrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1256 of *Lecture Notes in Comp. Science*, pages 270–280. Springer-Verlag, 1997.

[5] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14(3):545–568, 1985.

[6] R. F. Cohen and R. Tamassia. Dynamic expression trees. *Algorithmica*, 13:329–346, 1995.

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

[8] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal of Computing*, 26(2):484–538, 1997.

[9] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24:37–65, 1997.

[10] A. V. Goldberg, M. D. Grigoriadis, and R. E. Tarjan. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Math. Programming*, 50:277–290, 1991.

[11] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[12] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[13] J. Holm and K. de Lichtenberg. Top-trees and dynamic graph algorithms. Technical Report DIKU-TR-98/17, Department of Computer Science, University of Copenhagen, 1998.

[14] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

[15] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

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

[17] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[18] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, 1985.

[19] R. E. Tarjan. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78(2):169–177, 1997.

Appendix: An Example

The left part of Figure 11 shows an example of a free tree. Edges are arranged in counter-clockwise order around each vertex. To represent it, we first pick a degree-one vertex as the root and direct all edges towards it. Then, we divide the tree into maximal non-crossing edge-disjoint paths, all starting at some leaf. The right part of Figure 11 shows a possible partition with root z. A top tree corresponding to this partition is shown in Figure 12. Base nodes are shaded rectangles, compress nodes are white rectangles, and rake nodes are circles. Note that some nodes appear in pairs; in such cases, the foster child is on the left, and the proper child on the right. Non-paired nodes are always proper children.

The root path is $abcwpz$. It is represented by the top compress tree in Figure 12, which has N_b, N_c, N_w, and N_p as internal nodes, and ab, bc, cw, pw, and pz as leaves. Although the leaves represent the edges in the path, they need not appear in symmetric order.

The largest rake tree in the example has three internal nodes (β, δ, and ε) and four leaves (N_d, N_e, cf and N_g). The leaves all represent unit trees that are rooted at c and occur between (b, c) and (c, w) in the circular order (these are edges of the path that contains c). The first (leftmost) unit tree is composed by edges (d, r) and (c, d); the rightmost contains (g, h), (g, j), and (c, g).

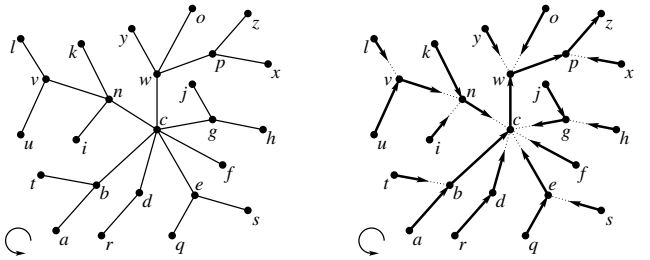

Figure 11: Example: Original tree and directed version (rooted at z and partitioned).

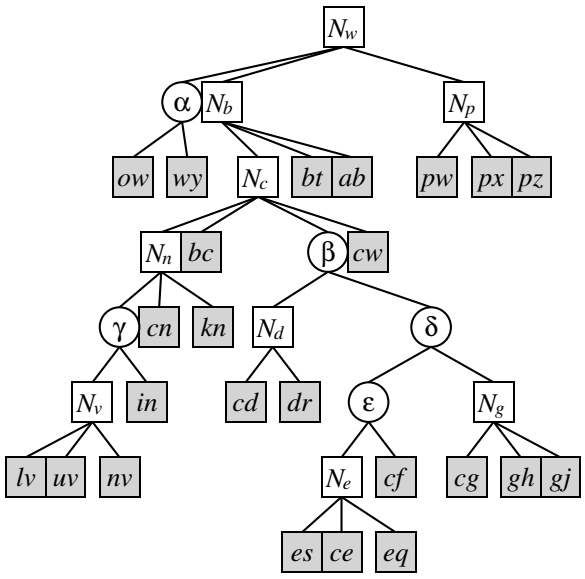

Figure 12: Corresponding top tree. Base nodes are shaded rectangles, compress nodes are white rectangles, rake nodes are circles.