Maintaining Information in Fully Dynamic Trees with Top Trees

STEPHEN ALSTRUP

IT University of Copenhagen

JACOB HOLM

Improva ApS

KRISTIAN DE LICHTENBERG

AND

MIKKEL THORUP

AT&T Labs—Research

Abstract. We design top trees as a new simpler interface for data structures maintaining information in a fully dynamic forest. We demonstrate how easy and versatile they are to use on a host of different applications. For example, we show how to maintain the diameter, center, and median of each tree in the forest. The forest can be updated by insertion and deletion of edges and by changes to vertex and edge weights. Each update is supported in $O(\log n)$ time, where n is the size of the tree(s) involved in the update. Also, we show how to support nearest common ancestor queries and level ancestor queries with respect to arbitrary roots in $O(\log n)$ time. Finally, with marked and unmarked vertices, we show how to compute distances to a nearest marked vertex. The latter has applications to approximate nearest marked vertex in general graphs, and thereby to static optimization problems over shortest path metrics.

Technically speaking, top trees are easily implemented either with Frederickson's [1997a] topology trees or with Sleator and Tarjan's [1983] dynamic trees. However, we claim that the interface is simpler for many applications, and indeed our new bounds are quadratic improvements over previous bounds where they exist.

This article includes work presented at ICALP'97 [Alstrup et al. 1997] and SWAT'00 [Alstrup et al. 2000].

Authors' addresses: S. Alstrup, IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 København S, Denmark, e-mail: stephen@itu.dk; J. Holm, Improva ApS, Symbion Science Park, Fruebjergvej 3, DK-2100 København Ø, Denmark, e-mail: jh@improva.dk; M. Thorup, AT&T Labs—Research, Florham Park, NJ 07932, e-mail: mthorup@research.att.com.

Categories and Subject Descriptors: E.1 [**Data Stuctures**]—*Trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]Nonnumerical Algorithms and Problems—*Computations on discrete structures*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Fully dynamic forest

1. *Introduction*

In this article, we introduce top trees as a new simpler interface for data structures maintaining information in a fully dynamic forest. Here *fully dynamic* means that edges may be both inserted and deleted. The information could be, say, the diameter of each tree in the forest. However, if the tree is a minimum spanning tree of a dynamic graph, the information could help changing the minimum spanning tree as the graph changes.

Technically speaking, top trees are easily implemented either with Frederickson's [1997a] topology trees or with Sleator and Tarjan's [1983] dynamic trees. The contribution of top trees is the *design* of an interface, providing users with easier access to the full power of these advanced techniques.

Targeting a broad audience of potential users, the bulk of this article is like a tutorial where we demonstrate the flexibility of top trees in different types of applications:

—We rederive some of the classic applications from Frederickson [1997a], Sleator and Tarjan [1983], and Goldberg et al. [1991], for example, finding the maximum weight of a given path.

—We improve some previous bounds. More specifically, we show how to maintain the centers and medians of trees in a dynamic forest in $O(\log n)$ time per updates. The previous bounds were $O(\log^2 n)$ time [Cheng and Ng 1996; Auletta et al. 1996].

—We consider problems that appear not to have been studied before for a dynamic forest. For example, we show how to maintain the diameters of trees in a dynamic forest. We also show how to answer level ancestor and nearest common ancestor queries with respect to arbitrary roots. Finally, with marked and unmarked vertices, we show how to compute distances to a nearest marked vertex. In all of these cases, we support both updates and queries in logarithmic time. The marking result has applications to approximate nearest marked vertex in general graphs, and thereby to static optimization problems over shortest path metrics.

We note that finding medians and centers is more difficult than, for example, finding the minimum edge on a given path because they are "nonlocal" properties. Here, by a *local property* we mean that if an edge or a vertex has the property in a tree, then it has the property in all subtrees it appears in. Local properties lend themselves nicely to bottom-up computations, whereas nonlocal properties tend to be more challenging. Building on top of our top trees, we present here a quite general technique for dealing with nonlocal properties.

We implement our top trees with Frederickson's [1997a] topology trees, which we in turn implement with Sleator and Tarjan's [1983] st-trees. The implementation of topology trees with st-trees was not known. It has the interesting consequence that the simple amortized version of st-trees gives a simple amortized version of topology trees.

We note that since top trees were originally announced [Alstrup et al. 1997], they have found applications in other works [Gabow et al. 2001; Holm et al. 2001; Thorup 2001]. All these applications rely on results presented in this article. Also, our specific result for dynamic tree diameters has found its own application in Nardelli et al. [2001].

1.1. PRELIMINARIES. Most of this article concerns graphs that are forests. Thus, if vertices v and w are connected, they are connected by a unique path, which we shall denote $v \cdots w$.

When we talk about an edge (v, w), on an implementation level, we often really think of an identifier e of the undirected edge with endpoints v and w. Via arrays, the endpoints can be found from the identifier e in constant time. However, other information can also be associated with e such as its successor and predecessor in the incidence lists around v and w.

1.2. CONTENTS. The article is organized as follows. In Section 2 we introduce top trees and solve the diameter problem. In Section 3 we present our technique for nonlocal problems, and solve the center and median problems. In Section 4 we discuss the advantages and limitations of using top trees relative to other data structures for dynamic trees. In Section 5 we mention some generalizations of top trees used in later articles. In Section 6 we implement top trees with topology trees and topology trees with st-trees. Finally, we have some concluding remarks in Section 7.

2. *Top Trees*

A top tree is defined based on a pair consisting of a tree T and a set ∂T of at most two vertices from T, called *external boundary vertices*. Given $(T, \partial T)$, any subtree C of T has a set $\partial_{(T,\partial T)}C$ of *boundary vertices* which are the vertices of C that are either in ∂T or incident to an edge in T leaving C. Here, by a *subtree* of an undirected tree, we mean any connected subgraph. The subtree C is called a *cluster* of $(T, \partial T)$ if it has at least one edge and at most two boundary vertices. Then T is itself a cluster with $\partial_{(T,\partial T)}T = \partial T$. Also, if A is a subtree of C, $\partial_{(C,\partial_{(T,\partial T)}C)}A = \partial_{(T,\partial T)}A$, so A is a cluster of $(C, \partial_{(T,\partial T)}C)$ if and only if A is a cluster of $(T, \partial T)$. Since $\partial_{(T,\partial T)}$ is a canonical generalization of ∂ from T to all subtrees of T, we will use ∂ as a shorthand for $\partial_{(T,\partial T)}$ in the rest of the article.

A *top tree* $\mathcal{R}$ over $(T, \partial T)$ is a binary tree such that

(1) the nodes of $\mathcal{R}$ are clusters of $(T, \partial T)$;
(2) the leaves of $\mathcal{R}$ are the edges of T;
(3) sibling clusters are *neighbors* in the sense that they intersect in a single vertex, and then their parent cluster is their union (see Figure 1);
(4) the root of $\mathcal{R}$ is T itself;

A tree with a single vertex has an empty top tree. The basic philosophy is that clusters are induced by their edges, the vertices only being included as their endpoints. This is why clusters need at least one edge, and we note that neighboring clusters are induced by disjoint edge sets inducing a common vertex.

We will sometimes refer to the tree T as the *underlying* tree to differentiate it from the *top* tree $\mathcal{R}$.

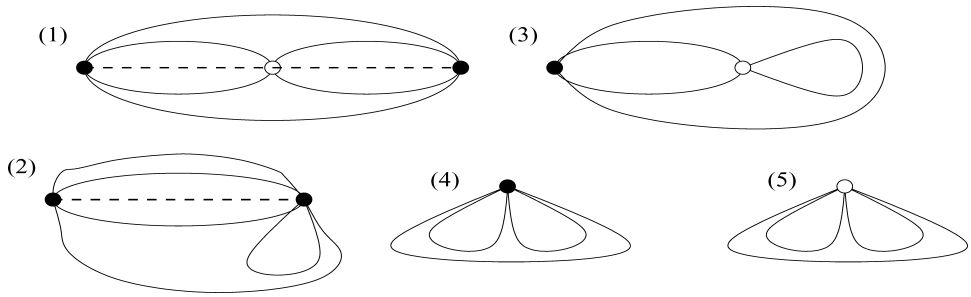

FIG. 1. The cases of joining two neighboring clusters into one. The • are the boundary vertices of the joined cluster and the ○ are the boundary vertices of the children clusters that did not become boundary vertices of the joined cluster. Finally, the dashed line is the cluster path of the joined cluster.

The top trees over the trees in our underlying forest are maintained under the following *forest updates*:

—link$((v, w))$ where v and w are in different trees, links these trees by adding the edge (v, w) to our dynamic forest.

—cut(e) removes the edge e from our dynamic forest.

—expose(v, w) where v and w are in the same tree T, makes v and w the external boundary vertices of T. Moreover, expose returns the new root cluster of the top tree over T. We can also call expose with a single vertex as argument if we only want one external boundary vertex.

In general, link and cut make the set of external boundary vertices for the resulting trees empty. To accommodate these forest updates, the top trees are changed by a sequence of local *top tree modifications* described below. During these modifications, we will temporarily accept a *partial* top tree whose root cluster may not be a whole underlying tree T but just a cluster of T.

—$e :=$ create(): creates a top tree with a single cluster e which is just an edge.

—$C :=$ join(A, B): where A and B are neighboring root clusters of two top trees $\mathcal{R}_A$ and $\mathcal{R}_B$. Creates a new cluster $C = A \cup B$ and makes it the common root of A and B, thus turning $\mathcal{R}_A$ and $\mathcal{R}_B$ into a single new top tree $\mathcal{R}_C$. Finally, the new root cluster C is returned.

—$(A, B) :=$ split(C): where C is a root cluster with children A and B in a top tree $\mathcal{R}_C$. Deletes C, thus turning $\mathcal{R}_C$ into the two top trees $\mathcal{R}_A$ and $\mathcal{R}_B$ with roots A and B. Finally, A and B are returned.

—destroy(e): eliminates the top tree consisting of edge e.

2.1. DISCIPLINE FOR MODIFYING TOP TREES. Top tree modifications have to be applied in the following order:

(1) First, top-down, we perform a sequence of splits.

(2) Then we destroy the clusters of some edges.

(3) Then we update the forest.

(4) Then we create clusters of some edges.

(5) Finally, with joins, we recreate the top tree bottom-up.

The above order implies that when we do a split or join, we know that all parts of the underlying forest are partitioned into base clusters.

It is an important rule that *a forest update may not change any current cluster*. Here, a cluster is changed by a forest update if the update changes its set of edges or its set of boundary vertices. To appreciate the latter, consider an update expose(v). This update only changes clusters with v an interior vertex. A cluster in which v is already a boundary vertex is not changed. Satisfying the rule means that when we get to the update in step 3, the previous steps 1–2 should have eliminated all clusters that would be changed by the update.

It is often natural to perform a *composite sequence of updates* in step 3. For example, if dealing with a spanning tree T, we might want to swap one tree edge (v_1, w_1) with another edge (v_2, w_2). If we do $\langle \mathsf{cut}((v_1, w_1)); \mathsf{link}((v_2, w_2))\rangle$ as a composite update rather than as two separate updates, we avoid dealing with a temporary forest when we do the top tree modifications in steps 1–2 and 4–5.

In this article, we are going to show the following result:

THEOREM 2.1. *For a dynamic forest we can maintain top trees of height* $O(\log n)$ *supporting each* link, cut, *or* expose *with a sequence of* $O(1)$ create *and* destroy, *and* $O(\log n)$ join *and* split. *These top tree modifications are identified in* $O(\log n)$ *time. The space usage of the top trees is linear in the size of the dynamic forest. For a composite sequence of* k *updates, each of the above bounds are multiplied by* k.

The proof of Theorem 2.1 is deferred to Section 6. Until then, the focus will be on applications of top trees.

2.2. TOP TREES GENERALIZE BALANCED BINARY SEARCH TREES. Put in perspective, our top trees are natural generalizations of standard balanced binary trees over dynamic collections of lists that may be concatenated and split. In the balanced binary trees, each node represents a segment of a list, which in top terminology is just a special case of a cluster. Standard implementations for balanced binary trees also ascertain that the height is $O(\log n)$, and that each concatenation and split can be done by $O(\log n)$ local modifications.

2.3. TOP TREE TERMINOLOGY. If a vertex in a cluster is not a boundary vertex, it is *internal* to that cluster. If a cluster C has two boundary vertices a and b, we call C a *path cluster* and $a \cdots b$ the *cluster path* of C, denoted $\pi(C)$. If C has only one boundary vertex a, C is called a *point cluster* and then $\pi(C) = a$. Note that if A is a child cluster of C and A shares an edge with $\pi(C)$, then $\pi(A) \subseteq \pi(C)$, and then we call A a *path child* of C. In terms of boundary vertices, if C has children A and B, A is a path child of C if and only if $|\partial C| = 2$ and either $\partial A = \partial C$ (Figure 1 (2)) or $\partial C \subset \partial A \cup \partial B$ (Figure 1 (1)).

2.4. REPRESENTATION AND USAGE OF TOP TREES. A top tree is represented as a standard binary rooted tree with parent and children pointers. The nodes used to represent the top tree are denoted *top nodes*. The top nodes of the binary tree represent the clusters, and with each top node is associated the set of at most two boundary vertices of the represented cluster. With a top leaf we store the corresponding edge. With a nonleaf top node is stored how it is decomposed into its children (cf. Figure 1). Thus, considering the information descending from a top node, we can construct the cluster it represents. Finally, from each vertex v, there

is a pointer to the smallest cluster $C(v)$ that v is internal to, or to the root cluster containing v if v is an external boundary vertex. Note that if v is internal to $C(v)$, then the ancestors of $C(v)$ represent all the clusters that v is internal to.

Following parent pointers from $C(v)$, we can find the root, *top_root*(v), of the top tree over the underlying tree T containing v. In the case of a forest, two vertices v and w are in the same underlying tree if and only if *top_root*$(v) =$ *top_root*(w). With top trees of logarithmic height as in Theorem 2.1, we identify *top_root*(v) in $O(\log n)$ time.

An *application* of the top tree data structure, such as maintaining diameters, centers, or medians, has direct access to the above representation, and will typically associate some extra information with the top nodes. The application employs an *implementation* of top trees, which is an algorithm like the one described in Theorem 2.1, converting each link, cut, or expose into a sequence of splits and joins on the top trees. In connection with each join and split, the application is notified and given pointers to the top nodes representing the involved clusters. The application can then update its information associated with these top nodes. We note that a top tree may only be modified with split and join. This discipline is important if we have several applications running over the same top trees, each maintaining its own information as splits and joins are performed. Typically, link and cut are operations imposed from the outside whereas expose typically is used internally by an application.

2.5. CONCRETE APPLICATIONS. As a first example, we can now easily derive a main result from of Sleator and Tarjan [1983]:

THEOREM 2.2 ([SLEATOR AND TARJAN 1983]). *We can maintain a dynamic collection of weighted trees in $O(\log n)$ time per* link *and* cut*, supporting queries about the maximum edge weight between any two vertices in $O(\log n)$ time.*

PROOF. For this application, with each (top node representing a) cluster C, we store as extra information the maximum weight *max_weight*(C) on the cluster path $\pi(C)$. For a point-cluster C, *max_weight*$(C) = -\infty$. If a path cluster consists of a single edge e, *max_weight*(e) is just the weight of the edge. When a path cluster C is created by a join, *max_weight*(C) is the maximum weight stored at its path children. When C is split or destroyed, we just discard the information stored with C. Now, to find the maximum weight between v and w, we set $C :=$ expose(v, w). Then $\pi(C) = v \cdots w$, and we return *max_weight*(C). Since join and split are supported in constant time, the theorem now follows from Theorem 2.1. □

In the above example, split is trivial. To see the relevance of split, we consider a simple extension.

THEOREM 2.3 ([SLEATOR AND TARJAN 1983]). *In Theorem* 2.2*, we can also add a common weight x to all edges on a given path $v \cdots w$ in $O(\log n)$ time.*

PROOF. For this extension, for each cluster C, we introduce a "lazy" weight *extra*(C) which is to be added to all edges in $\pi(C)$ in all clusters properly descending from C. We note that if C is a root cluster, *max_weight(C)* is not affected by these *extra*-values, so *max_weight(C)* is the correct maximum weight on $\pi(C)$. In particular, we can still find the maximum weight between v and w as *max_weight*(expose(v, w)).

The addition of x to $v \cdots w$ is now done by calling $C :=$ expose(v, w) and adding x to *max_weight(C)* and to *extra*(C). Then split(C) requires that, for each path child A of C, we set $max\_weight(A) := max\_weight(A) + extra(C)$ and $extra(A) := extra(A) + extra(C)$. For $C :=$ join(A, B), we set $max\_weight(C) := \max\{max\_weight(A), max\_weight(B)\}$ and $extra(C) := 0$. Finally, to find the maximum weight on the path $v \cdots w$, we set $C :=$ expose(v, w) and return *max_weight(C)*. □

We now go beyond Sleator and Tarjan [1983] with an extension of Goldberg et al. [1991].

THEOREM 2.4 ([GOLDBERG ET AL. 1991]). *In addition to the path updates from Theorem* 2.3*, we can ask for the maximum weight in the underlying tree containing a given vertex v in* $O(\log n)$ *time.*

PROOF. Elaborating on the information from the previous two proofs, for each cluster C, we will maintain a variable $max\_non\_path(C)$ denoting the maximum weight on an edge in C which is not on the cluster path. Assuming this variable, we can find the maximum weight of the underlying tree containing v, setting $C := top\_root(v)$ and returning $\max\{max\_non\_path(C), max\_weight(C)\}$.

We maintain the *max_non_path* variables as follows. When the cluster of an edge e is created, if e has two boundary vertices, we set $max\_non\_path(e) = -\infty$. Otherwise, $max\_non\_path(e)$ is set to the weight of e. When a cluster is joined as $C :=$ join(A, B), we first set $max\_non\_path(C) := \max\{max\_non\_path(A), max\_non\_path(B)\}$. If C is not a path cluster but one of its children, say A, is a path cluster (cf. Figure 1(3)), then we further have consider weights from the cluster path of A, setting $max\_non\_path(C) := \max\{max\_non\_path(C), max\_weight(A)\}$. We note here that because A was a root cluster, $max\_weight(A)$ has its correct value, not missing any *extra*-values from ascending clusters. When clusters are split or destroyed, this has no impact on the *max_non_path*-variables. □

In the rest of this article, we are more interested in distances than in maximum weights. Modifying the proof of Theorem 2.2, for each cluster C, we will maintain the length $length(C)$ of the cluster path. The length is maintained as the maximum weight except that, if C is created by a join, $length(C)$ is the sum of lengths stored with its path children. Thus we have

LEMMA 2.5. *In top trees, for each cluster C, we can maintain the length, denoted* $length(C)$*, of the cluster path in constant time per local top update, and hence in* $O(\log n)$ *time per* link *or* cut*. Then the distance between two vertices v and w can be found in* $O(\log n)$ *time as* $length(\text{expose}(v, w))$. □

As an interesting new application of top trees, we get the claimed result for dynamic diameters.

THEOREM 2.6. *We can maintain a dynamic collection of weighted trees in* $O(\log n)$ *time per* link *and* cut*, supporting queries about the diameter of the tree containing any vertex in* $O(\log n)$ *time.*

PROOF. For each cluster C, we store its diameter $diam(C)$. Moreover, for each of its boundary vertices $a \in \partial C$, we store the maximum distance $max\_dist(C, a)$ from a to any vertex in C. Finally, we maintain the cluster length from Lemma 2.5.

The variables max_dist and $length$ are auxiliary fields, needed for a fast join. Such carefully chosen extra information is often crucial in top tree applications.

When the cluster of an edge e is created, $diam(e) = weight(e)$, and for each boundary vertex v of e, $max\_dist(e, v) = weight(e)$. Now, suppose $C :=$ join(A, B), and that c is the common boundary vertex of A and B. Then we set

$$diam(C) := \max\{diam(A), diam(B), max\_dist(A, c) + max\_dist(B, c)\}.$$

Now consider any boundary vertex a of C. By symmetry, we may assume that if a is not in one of A and B, it is not in B. Let c be the intersection vertex of A and B. Then

$$max\_dist(C, a) = \max\{max\_dist(A, a), length(A) + max\_dist(B, c)\}.$$

Thus, create and join are implemented in constant time. As in the proof of Theorem 2.2, split and destroy do not require any action. Hence Theorem 2.1 implies that we can maintain the above information in $O(\log n)$ time per link or cut. To answer a diameter query for a vertex v, we set $C :=$ expose(v) and return $diam(C)$. □

Another illustrative application is the maintenance of nearest marked neighbors.

THEOREM 2.7. *We can maintain a dynamic collection of trees in $O(\log n)$ time per* link *and* cut*, or marking and unmarking of a vertex, supporting queries about the (distance to) the nearest marked vertex of any given vertex in $O(\log n)$ time.*

PROOF. Below, we just focus on finding the distance to the nearest marked vertex. This is easily extended to also providing the vertex.

For each boundary vertex a of a cluster C, we maintain the distance $mark\_dist(C, a)$ from a to the nearest marked vertex in $C \backslash \partial C$. The reason that we exclude the boundary of C from consideration is that a vertex v may appear as boundary vertex of $\Omega(n)$ clusters, and all these would be affected, if v was (un)marked. From $mark\_dist(C, a)$ we can easily compute the distance $mark\_dist^*(C, a)$ from a to the nearest marked vertex in C excluding only boundary vertices different from a. Then $mark\_dist^*(C, a) = 0$ if a is marked, and $mark\_dist^*(C, a) = mark\_dist(C, a)$ if a is unmarked. We also maintain the cluster path length, $length(C)$, as in Lemma 2.5.

Given a vertex u, to find the distance to the nearest marked vertex, we simply set $C :=$ expose(u), and return $mark\_dist^*(C, u)$.

To (un)mark a vertex v, we first expose v. As an external boundary vertex, v has no impact on any $mark\_dist$-value, so we can freely (un)mark it.

Suppose the cluster C is created as an edge (v, w). Then $mark\_dist(C, v)$ is the weight of (v, w) if w is marked and not in the boundary; otherwise, we set it to infinity.

Finally, consider $C :=$ join(A, B) with $\{c\} = A \cap B$. Let a be a boundary vertex of C. By symmetry, we can assume that a is in A. We now have $mark\_dist(C, a) =$

$$\begin{cases} \min\{mark\_dist(A, a), mark\_dist(B, a)\} & \text{if } a = c, \\ \min\{mark\_dist(A, a), length(A) + mark\_dist(B, c)\} & \text{if } a \neq c \text{ and } c \in \partial C, \\ \min\{mark\_dist(A, a), length(A) + mark\_dist^*(B, c)\} & \text{if } a \neq c \text{ and } c \notin \partial C. \end{cases}$$

Thus, we can support both join and create in constant time, and split and destroy do not require any action. By Theorem 2.1, this completes the proof of Theorem 2.7 □

COROLLARY 2.8. *For any positive integer parameter k, in a fixed undirected graph on n vertices and m edges, in $O(kmn^{1/k} \log n)$ expected time we can build an*

$O(kn^{1+1/k})$ space data structure, supporting (un)marking of vertices and queries about stretch $2k-1$ distances to a nearest marked vertex. Here stretch $2k-1$ means that the reported distance may be up to a factor $2k-1$ too long. Both queries and updates take $O(kn^{1/k}\log n)$ time.

PROOF. In Thorup and Zwick [2005], it was shown how to generate a cover of edge-induced trees within the above preprocessing bounds so that each vertex v is in $O(kn^{1/k})$ trees, and if the distance from v to w is d, there is a tree in which the distance is at most $(2k-1)d$. Now, if a vertex is marked, it is marked in all the trees containing it, and to find a stretch $2k-1$ distance to a nearest marked vertex, we find the shortest distance to a marked vertex over all the trees. □

The above corollary is interesting because in Goel et al. [2001] it was shown that several combinatorial optimization problems can be approximated efficiently on metrics with dynamic nearest neighbor. For example, in the bottleneck matching problem, where we wish to minimize the furthest distance between a pair in the matching, we now get a $4k-2$ approximation in $\tilde{O}(mn^{1/k})$ expected time. An exact solution currently requires $\tilde{O}(mn+n^{2.5})$ time [Even and Kariv 1975].

3. *Nonlocal Searching*

We are now going to build a black box on top of our top trees for maintenance of centers and medians. As discussed in the introduction, the common feature of centers and medians is that they represent nonlocal properties. Here a vertex/edge property is local if it being satisfied by a vertex/edge in a tree implies that the vertex/edge satisfies the property in all subtrees containing it. For example, being the minimum edge on a given path is a local property. Local properties lend themselves nicely to bottom-up computations whereas nonlocal properties appear to be more challenging.

For our general nonlocal searching, the application should supply a function select that, given the root cluster of a top tree, selects one of the two children. Recall here that a root cluster represents the whole underlying tree, which is important when dealing with nonlocal properties. Our black box will use select to guide a binary search after a desired edge. More precisely, the first time select is called, it is just given the root of an original top tree $\mathcal{R}$. It then selects one of the two children. In subsequent iterations, there will be some cluster C in the original top tree which is the intersection of all clusters selected so far. If C has children A and B, the black box modifies the top tree so that A and B are subsumed by different children A^* and B^* of the root. Then select is called on the root $C^* = \mathsf{join}(A^*, B^*)$. If A^* is selected, A is the new intersection of all selected clusters. Likewise, if B^* is selected, B is the new intersection of all selected clusters. This way, select is used to guide a binary search down through the original top tree $\mathcal{R}$. The formal statement of the result is as follows.

THEOREM 3.1 (NONLOCAL SEARCH). *Starting with the root cluster of a top tree of height h and at most one external boundary vertex, after $O(h)$ calls to* select, join, *and* split, *there is a unique edge (v, w) contained in all clusters chosen by* select, *and then (v, w) is returned. Subsequently, the top tree is returned to its previous state with $O(h)$ calls to* join *and* split.

If there are two external boundary vertices x and y, the above selection process will stop with a unique (v, w) edge on the path from x to y.

As stipulated in the general interface to top trees, the implementation behind Theorem 3.1 will only manipulate the top tree with join and split operations. In our applications, we will apply Theorem 3.1 to a top tree from Theorem 2.1 with height $h = O(\log n)$. Then the number of calls to join and split in Theorem 3.1 is $O(h) = O(\log n)$.

Theorem 3.1 will not be proved till Section 3.4. Before that we demonstrate applications of Theorem 3.1 in the dynamic center, median, and ancestor problems. In these applications, our general approach is to first decide the information needed for select, and second to show how to make the information available. The external boundary vertices will only play a role in the ancestor application in Section 3.3.

3.1. DYNAMIC CENTER. Let T be a tree with positive edge weights. For any vertex v in T, let $max\_dist(T, v)$ denote the maximum distance from v in T. A *center* is a vertex v minimizing $max\_dist(T, v)$.

LEMMA 3.2. *Let T be a tree with positive edge weights, and let A and B be neighboring clusters with $A \cap B = \{c\}$ and $A \cup B = T$. If $max\_dist(A, c) \geq max\_dist(B, c)$, A contains all centers.*

PROOF. Let w be a vertex in A of maximum distance to c. Then $dist(c, w) = max\_dist(A, c) = max\_dist(T, c)$. Now, for any $v \in B \setminus A$, $max\_dist(T, v) \geq dist(v, w) = dist(v, c) + dist(c, w) = dist(v, c) + max\_dist(T, c)$. Since the edge weights are positive, $dist(v, c) > 0$; thus $max\_dist(T, v) > max\_dist(T, c)$ and v cannot be a center. □

In the dynamic center problem, we maintain a forest under link and cut interspersed with queries $center(u)$ requesting the center of the current tree containing the vertex u. We use the top trees from Theorem 2.1. For each boundary vertex a of a cluster C, we maintain the maximum distance $max\_dist(C, a)$ from a in C as described in the proof of Theorem 2.6. Then link and cut take $O(\log n)$ time.

To find $center(u)$, we first set $D :=$ expose(u) so that D becomes the current root cluster over the tree containing u. The nonlocal search of Theorem 3.1 will start in D, but we need to define select given an arbitrary root cluster C with children A and B, $A \cap B = \{c\}$. If $max\_dist(A, c) \geq max\_dist(B, c)$, select picks A; otherwise it picks B. By Lemma 3.2, any cluster picked contains all centers, so, following Theorem 3.1, the returned edge (v, w) contains all centers. Moreover, select takes constant time, so (v, w) is found in $O(\log n)$ time. To find out if v or w is a center, we compute $D :=$ expose(v, w) in $O(\log n)$ time. Since D coincides with T, we can return v if $max\_dist(D, v) < max\_dist(D, w)$; w otherwise. Hence we can answer $center(u)$ in $O(\log n)$ time. Thus we conclude:

THEOREM 3.3. *The center can be maintained dynamically under* link, cut *and $center(u)$ queries in $O(\log n)$ worst case time per operation.*

3.2. DYNAMIC MEDIAN. Let T be a tree with positive vertex and edge weights. A *median* is a vertex m minimizing $\sum_{v \in V}(weight(v) \times dist(v, m))$, where $dist(v, m)$ is the distance from v to m in the tree. For any tree T, let $vert\_weight(T)$ denote the sum of the vertex weights of T. Our approach to finding medians is similar to that for centers, but for the median, it is natural to allow the application to change vertex weights. A challenge posed by these changes is that we cannot explicitly maintain the vertex weight of each cluster.

We shall use the following simple lemma of Goldman [1971]:

LEMMA 3.4 ([GOLDMAN 1971]). *Let (v, w) be an edge in the weighted tree T, and let T_v and T_w be the trees from $T \setminus \{(v, w)\}$ containing v and w, respectively. If $vert\_weight(T_v) = vert\_weight(T_w)$, v and w are the only medians in T, and if $vert\_weight(T_v) > vert\_weight(T_w)$, all medians in T are in T_v.*

COROLLARY 3.5. *Let T be a tree, and let A and B be neighboring clusters with $A \cap B = \{c\}$ and $A \cup B = T$. Then $vert\_weight(A) \geq vert\_weight(B)$ implies that A contains all medians of T.*

PROOF. Consider any edge (c, w) in B. Let T_c and T_w be defined as in Lemma 3.4. Then $T_c \subseteq A$ while $T_w \subseteq B \setminus \{c\}$. Since all vertex weights are positive, including that of c, we conclude that

$$vert\_weight(T_c) \geq vert\_weight(A) \geq vert\_weight(B) > vert\_weight(T_w).$$

Then Lemma 3.4 states that all medians of T are in T_c. However, A is the intersection over all such trees T_c, so we conclude that all medians are A. □

The above corollary suggests that we should maintain the vertex weight of each cluster. However, a single vertex may be contained in arbitrarily many clusters, and changing its weight would affect the vertex weight of all these clusters. Recall that we faced a similar problem for the *mark_dist*-values in the proof of Theorem 2.7. Again, we will resort to ignoring the boundary.

For each cluster C, we only maintain their "internal weight" $int\_weight(C) = vert\_weight(C \setminus \partial C)$. We can still derive the real weight $vert\_weight(C)$ as $int\_weight(C) + weight(\partial C)$ in constant time.

To join two clusters A and B, $A \cap B = \{c\}$ into C, we add their internal weights plus the weight of c if $c \notin \partial C$. A split has no impact on the weights. To change the weight of a *vert*ex v, we first call expose(v). Then v is not internal to any cluster, and hence no cluster information has to be updated when we change the weight of v.

We can now implement select as suggested by Corollary 3.5, choosing the child cluster minimizing *vert_weight* in constant time. Thus we get an edge (v, w) which contains all medians in $O(\log n)$ time.

To find a median among v and w, we apply Lemma 3.4. We cut the edge (v, w), and return v if the (root cluster of the) tree T_v containing v is heavier; otherwise we return w. Before returning v or w, we link (v, w) back in T. The link and cut take $O(\log n)$ time, so we conclude:

THEOREM 3.6. *The median can be maintained dynamically under* link, cut *and change of vertex weights in $O(\log n)$ worst-case-time per operation.*

3.3. NEAREST COMMON ANCESTORS AND LEVEL ANCESTORS. We will now show how to implement nearest common ancestors and level ancestors with respect to arbitrary roots. In the context of unrooted trees, this is done via the two functions $jump(x, y, d)$, returning the vertex d hops from x on the path from x to y, and $meet(x, y, z)$, returning the intersection point between the three paths connecting x, y, and z. With root r, the level ℓ ancestor of v is $jump(r, v, \ell)$, and the nearest common ancestor of u and v is $meet(u, v, r)$.

To implement $jump$ and $meet$, we assume unit edge weights and maintain the cluster path lengths $length(\cdot)$ from Lemma 2.5. To implement $jump(x, y, d)$, we first expose x and y. We now implement select as follows. Let A and B be the children of the root cluster C with $x \in A$ and $y \in B$. If $length(A) \leq d$,

we select A; otherwise we select B. Applying Theorem 3.1 with external boundary vertices x and y, the nonlocal search produces some edge between x and y. One endpoint of this edge has the desired distance d from x. We query the distance from each endpoint to x using Lemma 2.5, and return the one whose distance to x is d.

Having implemented $jump$ and using the distances from Lemma 2.5, we compute $meet(x, y, z)$ as $jump(z, x, (dist(x, z) + dist(y, z) - dist(x, y))/2)$. Thus we conclude:

THEOREM 3.7. *We can maintain a dynamic collection of weighted trees in* $O(\log n)$ *time per* link *and* cut, *supporting* $jump$ *and* $meet$ *queries in* $O(\log n)$ *time.*

3.4. NONLOCAL SEARCH IMPLEMENTATION. We will now first prove Theorem 3.1 when there are no external boundary vertices. Essentially our search will follow a path down the given top tree $\mathcal{R}$. As we search down, we will modify the top tree so as to facilitate calls to select, but we will end up restoring it in its original form. All modifications for the search are done via split and join, as stipulated in the general interface to top trees.

Our search consists of $O(\log n)$ iterations $i = 0, \ldots$. At the beginning of iteration i, there will be a "current" cluster C_i on depth i in the original top tree $\mathcal{R}$ which contains exactly the edges that have been in all clusters selected so far. Thus C_0 is the original root cluster representing an underlying tree T. If C_i is a single edge (v, w), we return (v, w). Otherwise C_i has children A_i and B_i in the original top tree. Then select will be presented a root cluster joining A_i^* and B_i^* such that $A_i \subseteq A_i^*$, $B_i \subseteq B_i^*$, and $T = A_i^* \cup B_i^*$. That is, the application-defined select will be called as select(join(A_i^*, B_i^*)). If the application selects A_i^*, we have $C_{i+1} = A_i$ for the next iteration. Otherwise $C_{i+1} = B_i$.

When iteration i starts, C_i is the root of a top tree which was a subtree of the original top tree $\mathcal{R}$. Besides, for each boundary vertex a of C_i, we have an "outside" root cluster X_a with everything from the underlying tree T that is separated from C_i by a. Also, X_a includes a. Together with C_i, the outside root clusters X_a partition the edges of T. For $C_0 = T$, we do not have any outside root clusters.

We are done when C_i is a top leaf consisting of a single edge. Otherwise, we split C_i into two children A_i and B_i.

To create A_i^*, we take the outside root clusters intersecting A_i and join them with A_i. If an outside root cluster does not intersect A_i, it intersects B_i, and is joined with B_i to create B_i^*. We then call the application-defined select on join(A_i^*, B_i^*).

We now split all the newly joined clusters so that the root clusters become A_i, B_i, and the outside root cluster for each boundary vertex of C_i from the beginning of the iteration. By symmetry, we may assume that select picked A_i. We then set $C_{i+1} := A_i$, and we join B_i with all outside root clusters intersecting B_i in a new maximal outside root cluster. Finally, we recurse on C_{i+1}.

As mentioned, the iterations stop as soon as we arrive at a C_i, which is just a single edge (v, w). Since each iteration only involves a constant number of joins and splits, we conclude that the total number of joins and splits is $O(h)$, where h is the initial height of the top tree. In the end, when we have found $C_i = (v, w)$, we just reverse all joins and splits to restore the top tree in its original form, and return the edge (v, w).

With a minor modification, the above construction also works in the presence of a single external boundary vertex. The modification is in the case where a boundary vertex a of C_i is the external boundary vertex and where a does not separate C_i from any part of the underlying tree. In that case, no outside cluster X_a is associated with a. This completes our implementation of Theorem 3.1 when there are fewer than two external boundary vertices.

3.5. TWO EXTERNAL BOUNDARY VERTICES. The nonlocal search described above works fine with fewer than two boundary vertices. However, when we have two external boundary vertices x and y in the underlying tree T, the goal of the nonlocal search is to select an edge on $x \cdots y = \pi(T)$. In the above selection process, this means that the currently selected cluster C_i should always have an edge e from $x \cdots y$. Then $e \in \pi(C_i) \subseteq \pi(T)$. Thus it follows that if a child of C_i is not a path child, then that child cannot be selected. In that case, the only path child is automatically made the next current cluster C_{i+1}. The process stops when $\pi(C_i)$ consists of a single edge, which is then returned.

In the actual implementation, since C_i has an edge in its cluster path, C_i has two distinct boundary vertices a and b with disjoint outside root clusters X_a and X_b. Each of these outside root clusters contains one of the two external boundary vertices. Let A_i and B_i be the children of C_i with $a \in A_i$ and $b \in B_i$. If A_i is not a path child, it has no edge from $x \cdots y$, so we set $X_a = \mathsf{join}(X_a, A_i)$ and $C_{i+1} = B_i$. Similarly, if B_i is not a path child, we set $X_b = \mathsf{join}(X_b, B_i)$ and $C_{i+1} = A_i$. It is only if both A_i and B_i are path children that we call the application-defined select on $\mathsf{join}(A_i^*, B_i^*)$ where $A_i^* = \mathsf{join}(X_a, A_i)$ and $B_i^* = \mathsf{join}(X_b, B_i)$.

We note that with two external boundary vertices x and y, it is necessary that we restrict select to pick edges from $x \cdots y$ as above. Otherwise, above we could end up with A_i and B_i intersecting in a vertex c outside $x \cdots y$. Since A_i^* and B_i^* intersect in c and partition the underlying tree, one of them would contain both x and y, and hence have three boundary vertices x, y, and c.

This completes our implementation of Theorem 3.1.

4. *Methodological Remarks*

Our results on diameters, centers, and medians could also have been achieved based on either Sleator and Tarjan's [1983] dynamic trees, or Frederickson's [1985, 1997a] topology trees. However, we claim that the derivation from these more classical data structures would have been more technical.

4.1. FREDERICKSON'S TOPOLOGY TREES. Top trees are very similar to Frederickson's [1985, 1997a] topology trees, from which they are derived. The essential difference is that the clusters of topology trees are not connected via vertices, but via edges. Since Frederickson's boundary consisted of edges, he could not limit the boundaries for unlimited degree trees. Thus, in applications for unbounded degrees one has to code these with ternary trees, inserting some extra edges and vertices that typically require special handling. Even if we assume we are dealing with ternary trees, topology trees still have clusters with up to three boundary edges instead of just two boundary vertices. Also topology join combines two clusters *plus* the edge between them whereas a top join just unites two neighboring clusters. Neither of these issues lead to fundamental difficulties, but, in our experience, they lead to significantly more cases.

We note that Frederickson [1997b] has already shown how Sleator and Tarjan's [1983] axiomatic interface to dynamic trees can be implemented with topology trees. Our corresponding implementation with top trees from Section 2 is inspired by that of Frederickson.

4.2. SLEATOR AND TARJAN'S DYNAMIC TREES. Sleator and Tarjan [1983] provided an axiomatic interface for their dynamic trees where an application can choose a root with a so-called evert operation, and then, for any specific vertex, add weights to all edges on the path to the root, or ask for the minimum of all weights on this path. This is basically the interface we implemented with top trees at the end of Section 2, assuming that we expose both the desired root and the specified vertex.

Before discussing limitations to the above interface, we first illustrate its generality by viewing the min-query as representing an arbitrary associative operator $\oplus$. For example, suppose as in Sleator and Tarjan [1983] that we want to implement parent pointers to the current root. We then let the weight of an edge be its pair of endpoints and define $a \oplus b = a$. Then the "min"-query returns the endpoints of the first edge on the path to the root, from which we immediately get a parent pointer. Similarly, adding x to all weights on a path could be done with any associative operator $\otimes$ that distributes over $\oplus$, that is, $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$. Instead of having $(\oplus, \otimes) = (\min, +)$, we could have, for example, $(\oplus, \otimes) = (+, \times)$.

Despite these generalizations, the axiomatic interface is still centered around paths, and it has been found too limited for many applications of dynamic trees. Instead authors have had to work directly with Sleator and Tarjan's [1983] underlying representation [Westbrook and Tarjan 1992; Battista and Tamassia 1989, 1990; La Poutré 1991; Poutré 1992, 1994; Galil and Italiano 1991; Kanevsky et al. 1991; Goldberg et al. 1991; Cohen and Tamassia 1991, 1995, 1997; Peckham 1989]. In particular, this is the case for the previous solutions to the dynamic center [Cheng and Ng 1996] and median problems [Auletta et al. 1996], and we believe part of the reason for their worse bounds and more complex solutions is difficulties in working directly with Sleator and Tarjan's [1983] underlying representation. For contrast, with top or topology trees it is easy to deal directly with the representation, and with top trees we got the right logarithmic bounds for the dynamic center and median problems in trees.

4.3. HENZINGER AND KING'S ET-TREES. For completeness, we also mention Henzinger and King's [2001] ET-trees, which they originally used to maintain connectivity in a dynamic forest. ET-trees; maintain a standard binary tree over the Euler tour of each tree in a dynamic forest. This technique is much simpler to implement than those mentioned above. As pointed out in Tarjan [1997], ET-trees provide a simple solution to the problem of maintaining the minimum weight vertex of each tree in a dynamic forest. However, it appears that there is no simple way for ET-trees to support path-oriented updates and queries. Thus, contrasting Theorem 2.4, ET-trees will not support efficient path updates, adding a value to all weights on a specified path in a tree.

5. *Generalizations of Top Trees*

In the following, to avoid confusion with leaves in the underlying trees, we refer to the leaves of a top tree as *base clusters*. At present the base clusters are just the edges

of the underlying tree, but it is sometimes important to deal with fewer but larger base clusters. For example, this is needed in classical topology tree applications such as maintaining the minimum spanning tree of a fully dynamic graph [Frederickson 1985]. Also, it is needed for a recent application of top trees maintaining minimum cuts [Thorup 2001]. For these applications, we allow the user to distribute *labels* on the vertices of the underlying tree. These labels represent application-specific information associated with the vertices. For example, if we are maintaining a minimum spanning tree, the labels represent incident ends of non tree edges.

We note that Frederickson's [1985, 1997a] topology trees do not support labels. His underlying trees have to be ternary so each application has to decide how to code high-degree vertices and other information in ternary trees.

Thus our top trees are now dealing with a labeled tree T. Each label is attached to a unique vertex, but the same vertex may have many labels attached. In many regards, the labels can be thought of as edges with a single endpoint.

In a subtree U of a labeled tree T, each vertex may have attached any subset of its labels in T. We extend the notion of boundary vertices to include vertices in U that have fewer labels attached in U than in T. That is, ∂U is now the set of vertices in U that are either external boundary vertices of T or vertices with an incident edge or attached label that is included in T but not in U.

A cluster U of T is a subtree with at most two boundary vertices containing at least an edge or a label. Thus, we now accept a single vertex as a cluster if it has an associated label in the cluster. Two clusters are neighbors if their intersection is a single vertex. They cannot have any labels or edges in common. It follows that the base clusters of a top tree form a partitioning of the edges and labels of the underlying tree. Similarly, it follows that labels, like edges, appear in exactly one cluster on each level in a top tree.

One conceptual advantage to labels is that any cluster can be reduced to an edge or a label. More precisely, we get a new labeled tree if we replace a point cluster with a label at its boundary vertex, or if we replace a path cluster with an edge between its boundary vertices.

A simple application of labels would be to attach a label $[v]$ to a vertex v. On each level of a top tree, the label $[v]$ will only appear once whereas the vertex v can participate in arbitrarily many clusters. This way, $[v]$ can be used as a distinguished representative for v in a top tree.

In addition to the original link, cut, and expose operations, we have the two new operations:

—attach(v, a): attaches a label a to the vertex v.

—detach(a): detaches the label a from whatever vertex it was attached to.

To get the full power of the generalized top trees, we allow top nodes C with a single child D, created by $C :=$ join(D). Then C and D represent exactly the same cluster. We can then get *leveled top trees* where all base clusters are on level 0, and where the parent of a level i top node is on level $i + 1$. We define the *size of a cluster or labeled tree* to be the total number of its edges and labels. We now have the following generalization of Theorem 2.1:

THEOREM 5.1. *Consider a fully dynamic forest and let Q be a positive integer parameter. For the trees in the forest, we can maintain a leveled top trees whose base clusters are of size at most Q and such that if a tree has size s, it has height*

$h = O(\log s)$ and $\lceil O(s/(Q(1+\varepsilon)^i))\rceil$ clusters on level $i \le h$. Here ε is a positive constant. Each link, cut, attach, detach, *or* expose *operation is supported with $O(1)$* create*s and* destroy*s, and $O(1)$* join*s and* split*s on each positive level. If the involved trees have total size s, this involves $O(\log s)$ top tree modifications, all of which are identified in $O(Q + \log s)$ time. For a composite sequence of k updates, each of the above bounds are multiplied by k. As a variant, if we have parameter S bounding the size of each underlying tree, then we can choose to let all top roots be on the same level $H = O(\log S)$.*

We note that Theorem 5.1 implies Theorem 2.1. More precisely, to get Theorem 2.1 from Theorem 5.1, we set $Q = 1$, use no labels, and skip all top nodes that are single children.

To appreciate Theorem 5.1, we briefly sketch Frederickson's algorithm for maintaining a minimum spanning tree of a fully dynamic graph, but using top trees instead of topology trees.

THEOREM 5.2 ([FREDERICKSON 1985]). *We can maintain a minimum spanning tree of a fully dynamic connected graph in $O(\sqrt{m})$ time per edge insertion or deletion.*[1]

PROOF. If an edge (v, w) is inserted in the graph, it should be added to the minimum spanning tree T if it is lighter than the maximum weight on the path from v to w in T. From Theorem 2.2, we already know how to support such path queries in $O(\log n)$ time.

Our challenge is to deal with the deletion of a tree edge. Our task is to find a lightest replacement edge reconnecting the tree, and we will show how to do this in $O(\sqrt{m})$ time.

We will employ leveled top trees $\mathcal{R}$ from Theorem 5.1, where the labels attached to a vertex are ends of incident nontree edges. More precisely, for each nontree edge (v, w), we have a label $[v, w]$ attached to v and a symmetric label $[w, v]$ attached w. These two labels are always attached or detached as a composite update (cf. Section 2.1) so that we never have one but not the other present in our top trees. The total size of our labeled forest is then the number m of edges in the graph.

We will use the variant of top trees in the end of Theorem 5.1 with S an upper bound on the total size m. Using standard background rebuilding, we can ensure $S = \Theta(m)$. More precisely, we can divide updates into epochs that first initiate new top trees $\mathcal{R}'$ with this $S' = 2m$ instead of the current S. During the next $S/4$ updates, we copy the current data from $\mathcal{R}$ to $\mathcal{R}'$, and switch to $\mathcal{R}'$ when done.

Now that S is fixed for the current top tree $\mathcal{R}$, we set $Q = \sqrt{S} = \Theta(\sqrt{m})$. Since we have at most two trees at any time, the number of clusters on level $i \le H = O(\log S) = O(\log m)$ is $\lceil O(S/(Q(1+\varepsilon)^i))\rceil = \lceil O(\sqrt{m}/(1+\varepsilon)^i)\rceil$.

For each pair (C, D) of clusters on the same level, we will store the lightest nontree edge *lightest*(C, D) between them. Here (v, w) goes between C and D if $[v, w]$ is a label in C and $[w, v]$ is a label in D, or vice versa. Assuming that the clusters are enumerated with numbers up to $O(\sqrt{S})$, we can implement *lightest* as a simple two-dimensional array over all cluster pairs. We can just ignore entries

[1] We note that for denser graphs, Eppstein et al. [1997] have improved the $O(\sqrt{m})$ bound to $O(\sqrt{n})$ using their general sparsification technique.

with cluster pairs on different levels. Also, since *lightest* is symmetric, we identify *lightest*(C, D) with *lightest*(D, C).

Assuming that the array *lightest* is properly maintained, if a tree edge (v, w) is deleted, we cut it, and then the desired minimum replacement edge is the minimum edge between the root clusters. More precisely, we perform the following sequence of operations:

$$\mathsf{cut}((v, w));\ C := \mathit{top\_root}(v);\ D := \mathit{top\_root}(w);\ (x, y) := \mathit{lightest}(C, D);$$
$$\langle \mathsf{detach}([x, y]);\ \mathsf{detach}([y, x]); \rangle\ \mathsf{link}((x, y));$$

We now have to show how to maintain *lightest*. Suppose a base cluster B is created. Since it has only $\sqrt{m}$ incident nontree edges, each going to a base cluster on the same level, we can easily find *lightest*(B, D) for all the $O(\sqrt{m})$ base clusters $D \in \mathcal{R}$ in $O(\sqrt{m})$ time.

Now suppose a level $i > 0$ cluster C is joined. For each of the $\lceil O(\sqrt{m}/(1+\varepsilon)^i) \rceil$ other level i clusters $D \in \mathcal{R}$, we set *lightest*(C, D) to be the lightest of *lightest*(A, B) where A is a child of C and B is a child of D. Thus we compute *lightest*(C, D) in constant time.

Finally, we note that split and destroy require no action. It follows from Theorem 5.1 that each link, cut, expose, attach, or detach operation is supported in

$$O\left(\sum_{i=0}^{O(\log n)} \lceil \sqrt{m}/(1+\varepsilon)^i \rceil\right) = O(\sqrt{m})$$

time, which is then also the time bound for finding a replacement edge. Note how the above sum uses the geometrically decreasing bound on the number of nodes in each level; otherwise the time bound would be worse by a logarithmic factor. □

A much more involved application using the generalized top trees from Theorem 5.1 is the fully dynamic algorithm for maintaining minimum cuts [Thorup 2001]. We note that Thorup [2001] assumed Theorem 5.1, which is proved below in this article by reduction to Frederickson's [1997a] trees.

6. *Implementing Top Trees*

We will now first implement the top trees of Theorem 5.1 via Frederickson's [1997a] topology trees, and thereby establish Theorem 5.1 and Theorem 2.1. Next, we implement the topology trees with Sleator and Tarjan's [1983] st-trees. The connection is interesting because topology trees and st-trees so far have been implemented with very different techniques. A nice consequence is that the simple amortized implementation of st-trees implies a simple amortized implementation of topology trees, and of top trees. Previously, no simple amortized implementation of topology trees was known. We note that, for a practical implementation, one should not follow all our reductions rigorously, but rather go for a more direct implementation. We hope to address these practical issues in future work.

6.1. Implementing Expose.

As a very first step in our reduction, we note that if we first have an implementation of top trees without expose, then later we can easily add expose. The simple point is that, in a top tree of height h, each vertex is internal to at most h clusters. To expose a and b, we simply split all the clusters

having them as internal vertices. We now have a set of $O(h)$ root clusters to be joined into one cluster. Clearly, this can require at most $O(h)$ joins, so we do not need to worry about the new height. First, as long as there is a point cluster, we join it with an arbitrary neighbor. If $a = b$, this process ends with a single point cluster, as desired. Otherwise, we end with a string of path clusters $C_1, \ldots, C_k$ with boundaries $\{c_0, c_1\}, \{c_1, c_2\}, \ldots, \{c_{k-1}, c_k\}$, where $c_0 = a$ and $c_k = b$. We can then repeatedly join neighbors in this string until a single path cluster with boundary $\{a, b\}$ remains. Before supporting any new link or cut, we simply revert all the above joins and splits, restoring the previous unexposed top tree.

Thus, in the remaining implementation, we may consider expose done, and focus on maintaining top trees of height $O(\log n)$ under link and cut as in Theorem 5.1 but without expose.

6.2. TOP TREES VIA TOPOLOGY TREES. Theorem 5.1 without expose is proved in Frederickson [1997a] in the context of topology trees with their different definition of clusters. The topology clusters are subtrees like top clusters, but in a topology tree, independent clusters are vertex-disjoint. In particular, the topology base clusters are disjoint. They partition the vertices and are connected via edges. The topology trees are only defined for ternary trees. A cluster may have at most three edges leaving it, called *boundary edges*, and if it has three edges leaving it, it may only consist of a single vertex. The topology tree is binary like a top tree. A parent cluster is the union of the two child clusters plus the edge connecting them.

Now, implementing top trees with topology trees is easy. We ternarize each vertex as follows: while there is a vertex v with degree >3, we turn v into a path with the incident edges branching off. More precisely, if v is incident to $w_0, \ldots, w_d$, $d \geq 3$, we may replace v by a path $v_1, \ldots, v_{d-1}$ with incident edges (v_1, w_0), (v_i, w_i), $i = 1, \ldots, d-1$, and (v_{d-1}, w_d). The edge (v_i, w_j) remembers that it originated from (v, w_j). In Frederickson's topology trees, the base clusters are all disjoint. To represent labels associated with a vertex v, we just add them to the above path representing v as extra vertices.

To transform a topology tree into a top tree, we essentially just take each topology cluster C and transform it into the top cluster C' induced by the vertices, edges, and labels contained in C. We note that C' has at most two boundary vertices. Clearly this is the case if C has at most two boundary edges, but if C has three boundary edges, C consists of a single vertex, which is hence the only boundary vertex. As an exception, if a topology cluster has no labels or edges from the original tree, the corresponding top cluster is considered empty and has no representative in the top tree.

The base top clusters are those derived from the base topology clusters, plus a base cluster for each edge not in a derived base cluster. Now, a topology join converts into two top joins, where first one of the topology children join with the edge between them. Next the resulting top cluster joins with the other topology child. Here a join with an empty top cluster is just skipped. Since a topology join may requires two top joins, each level in a topology tree translates into two levels in a top tree. Given the proofs for topology trees in Frederickson [1997a], pp. 486–497, we conclude that Theorem 2.1 and 5.1 hold true. The achievement with top trees is a simpler interface for high-degree trees where the ternarization is not done by each application but by the implementation via the above reduction. Also, the join has slightly fewer cases and is slightly simpler because we do not have to incorporate an edge between the clusters.

6.3. TOPOLOGY TREES VIA ST-TREES. We will now demonstrate how Sleator and Tarjan's [1983] st-trees can be used to implement topology trees whose base clusters are the vertices. Together with the previous reduction from top trees to topology trees, this provides us with a very different implementation of Theorem 2.1. Here by st-trees, we do not refer to the nice path-oriented axiomatic interface from Sleator and Tarjan [1983], but to the underlying implementation.

First, we note that the st-trees are presented for rooted trees, but on the other hand, they have an evert(v) operation, making v the root of its tree. Hence, to perform an arbitrary link(u, v), we can first evert(u), making it root of its tree, and then link(u, v), making (u, v) a parent pointer.

Since our starting point is an unrooted ternary tree, a rooted version of it is a binary tree. An exception is the root, which in principle could have three children. However, this is easily avoided. First of all, we could pick the root as a leaf in the unrooted tree with degree 1. Also, consider the situation above where we want to link(u, v) and first make u the root with evert(u). Since the result is ternary, u had degree at most two before link(u, v), so u does not get three children. The link(u, v) operation is just adding a parent pointer to u.

Sleator and Tarjan [1983] partitioned a rooted binary tree T into a set of disjoint solid paths, each a path from some vertex to a descendant of that vertex in T. They then formed an st-tree $\mathcal{T}$ as follows. They took each solid path $P = (v_1, \ldots, v_p)$ with v_1 closest to the root and v_0 the parent of v_1, and removed all parent pointers of the vertices in P. Then they made a binary tree $\mathcal{P}$ with $v_1, \ldots, v_p$ as leaves appearing in this order, and made v_0 the parent of the root. If v_1 was the root of the whole tree, the root of $\mathcal{P}$ became the root of $\mathcal{T}$, which in Sleator and Tarjan [1983] ended up with logarithmic height.

Note that some nodes of $\mathcal{T}$ are vertices in T. Each node v in $\mathcal{T}$ represents the cluster $C(v)$ of T induced by the vertices from T descending from v in $\mathcal{T}$. To see that these are clusters, we just note that, if $v \in \mathcal{P}$ above, the descendants of v from P form a segment S of P. The only potential edges incident to $C(v)$ are a parent pointer from the first vertex in S and a child pointer from the last vertex in S to a child in P.

We can now construct the topology tree as follows. The base clusters are the vertices of T. The rest of the topology tree is constructed by following $\mathcal{T}$ bottom-up. When we meet a node v which is a vertex from T, it has only one child w in $\mathcal{T}$, which was its nonsolid child in T. Then $C(v) = \text{join}(\{v\}, C(w))$. When we meet a node v' which is not a vertex from T, it has two children u and w in $\mathcal{T}$, and then $C(v') = \text{join}(C(u), C(w))$.

Thus we have established a mapping from the st-tree $\mathcal{T}$ to a topology tree $\mathcal{R}$ whose base clusters are the vertices. Since the st-tree has height $O(\log n)$, so does the topology tree. Also, the main technical result from Sleator and Tarjan [1983] is that each link, cut, and evert only affects $O(\log n)$ nodes in the st-trees, including their parents, and hence this gets translated into $O(\log n)$ splits and joins. Thus, we can derive Frederickson's [1997a] topology trees, and hence top trees from Sleator and Tarjan's [1983] st-trees. In particular this implies that the simple amortized version of st-trees [Sleator and Tarjan 1985] provides a simpler amortized version of top trees. We note here that the amortized version of st-trees uses a slightly different representation than that of the worst-case version discussed above, and that our reduction would have to be changed accordingly. Also note that when using such

an amortized version of top trees, there is no guarantee for the height of the top tree. However, if we precede each query with an expose, we will get logarithmic amortized bounds for both queries and updates.

The advantage of top trees and topology trees over st-trees is a nice, easy-to-apply interpretation of the system of solid paths replaced by binary trees in st-trees. This point is illustrated with our top tree solutions to the diameter, center, and median problems for dynamic trees, improving over previous solutions based on st-trees [Auletta et al. 1996; Cheng and Ng 1996].

7. *Concluding Remarks*

We have designed top trees as an interface providing users with easier access to the power of previous techniques for maintaining information in a fully dynamic forest. Conceptually, top trees are very similar to Frederickson's [1997a] topology trees, the subtle difference being that top clusters are joined by vertices whereas topology trees are joined via edges. This small difference has the immediate advantage that top trees work directly for trees of unbounded degrees, which with topology trees would first have to be coded as ternary trees. It also makes joins of two clusters a bit simpler in that they do not involve an intermediate edge.

Using top trees, we dealt with a variety of different applications including non-local search problems like maintaining the center or median of trees in a dynamic forest. For these two problems, we provided quadratic improvements over previous bounds. We also showed how top trees, in theory, could be implemented both with Frederickson's [1997a] topology trees, and with Sleator and Tarjan's [1983] st-trees.

A main practical challenge is now to make a good library implementation of top trees for use in different applications. We could have different implementations, for example, a worst-case implementation based on the ideas in topology trees [Frederickson 1997a], and a faster amortized implementation based on st-trees [Sleator and Tarjan 1983]. For speed, the implementations should be tuned directly for top trees and not just use our general reductions. Ideally, applications and implementations should only communicate with each other via the top tree interface, so that one can replace one implementation with another in a plug-and-play manner without a change to the applications. It is not trivial to make such generic interfaces efficient, but C++ solutions have been reported by Austern et al. [2003] for the simpler case of balanced binary search trees. We do hope to address such practical library implementations of top trees in future work.

Recent developments. Some very recent developments should be mentioned. Acar et al. [2004, 2005] have studied an alternative technique for dynamic trees based on tree contraction. As with topology trees, their underlying dynamic trees are assumed ternary. They do suggest an interesting randomized implementation that may be relevant for top trees. Also Tarjan and Werneck [2005] have found a direct implementation of amortized top trees that should be more efficient than our reduction to amortized st-trees.

ACKNOWLEDGMENT. We would like to thank Renato Werneck as well as a referee from *ACM Transactions on Algorithms* for many helpful comments.

REFERENCES

ACAR, U., BLELLOCH, G., HARPER, R., VITTES, J., AND WOO, S. 2004. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proceedings of the 15th SODA*. ACM Press, New York, NY/SIAM Press, Philadelphia, PA, 531–540.

ACAR, U., BLELLOCH, G., AND VITTES, J. 2005. Experimental analysis of change propagation in dynamic trees. In *Proceedings of the ALENEX05*. ACM, Press, New York, NY.

ALSTRUP, S., HOLM, J., DE LICHTENBERG, K., AND THORUP, M. 1997. Minimizing diameters of dynamic trees. In *Proceedings of the 24th ICALP*. Springer, Berlin, Germany, 270–280.

ALSTRUP, S., HOLM, J., AND THORUP, M. 2000. Maintaining center and median in dynamic trees. In *Proceedings of the 7th SWAT*. Lecture Notes in Computer Science, vol. 1851. Springer, Berlin, Germany, 46–56.

AULETTA, V., PARENTE, D., AND PERSIANO, G. 1996. Dynamic and static algorithms for optimal placement of resources in a tree. *Theor. Comp. Sci. 165*, 441–461.

AUSTERN, M., STROUSTRUP, B., THORUP, M., AND WILKINSON, J. 2003. Untangling the balancing and searching of balanced binary search trees. *Softw.: Pract. Exper. 33*, 13, 1273–1298.

BATTISTA, G., AND TAMASSIA, R. 1989. Incremental planarity testing. In *Proceedings of the 30th FOCS*. IEEE. 436–441.

BATTISTA, G., AND TAMASSIA, R. 1990. On-line graph algorithms with SPQR-trees. In *Proceedings of the 17th ICALP*. Lecture Notes in Computer Science, vol. 443. Springer, Berlin, Germany, 598–611.

CHENG, S., AND NG, M. 1996. Isomorphism testing and display of symmetries in dynamic trees. In *Proceedings of the 7th SODA*. ACM Press, New York, NY/SIAM Press, Philadelphia, PA, 202–211.

COHEN, R., AND TAMASSIA, R. 1995. Dynamic expression trees. *Algorithmica 13*, 3, 245–265.

COHEN, R., AND TAMASSIA, R. 1997. Combine and conquer. *Algorithmica 18*, 3, 324–362.

COHEN, R. F., AND TAMASSIA, R. 1991. Dynamic expression trees and their applications. In *Proceedings of the 2nd SODA*. ACM Press, New York, NY/SIAM Press, Philadelphia, PA, 52–61.

EPPSTEIN, D., GALIL, Z., ITALIANO, G. F., AND NISSENZWEIG, A. 1997. Sparsification—A technique for speeding up dynamic graph algorithms. *J. ACM 44*, 5, 669–696.

EVEN, S., AND KARIV, O. 1975. An $o(n^{2.5})$ algorithm for maximum matching in general graphs. In *Proceedings of the 16th FOCS*. IEEE, Piscataway, NJ, 100–112.

FREDERICKSON, G. 1985. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comp. 14*, 4, 781–798.

FREDERICKSON, G. 1997a. Ambivalent data structures for dynamic 2–edge–connectivity and k smallest spanning trees. *SIAM J. Comp. 26*, 2, 484–538. Announced at FOCS'91.

FREDERICKSON, G. 1997b. A data structure for dynamically maintaining rooted trees. *J. Algorith. 24*, 1, 37–65. Announced at SODA'93.

GABOW, H. N., KAPLAN, H., AND TARJAN, R. E. 2001. Unique maximum matching algorithms. *J. Algorith. 40*, 2, 159–183. Announced at STOC'99.

GALIL, Z., AND ITALIANO, G. 1991. Maintaining biconnected components of dynamic planar graphs. In *Proceedings of the 18th ICALP*. Lecture Notes in Computer Science, vol. 510. Springer, Berlin, Germany, 339–350.

GOEL, A., INDYK, P., AND VARADARAJAN, K. 2001. Reductions among high dimensional proximity problems. In *Proceedings of the 10th SODA*. ACM Press, New York, NY/SIAM Press, Philadelphia, PA, 769–778.

GOLDBERG, A. V., GRIGORIADIS, M. D., AND TARJAN, R. E. 1991. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Math. Programm. 50*, 277–290.

GOLDMAN, A. 1971. Optimal center location in simple networks. *Transportat. Sci. 5*, 212–221.

HENZINGER, M. R., AND KING, V. 2001. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Comput. 31*, 2, 364–374.

HOLM, J., DE LICHTENBERG, K., AND THORUP, M. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. *J. ACM 48*, 4, 723–760.

KANEVSKY, A., TAMASSIA, R., BATTISTA, G. D., AND CHEN, J. 1991. On-line maintenance of the four-connected components of a graph. In *Proceedings of the 32nd FOCS*. IEEE, Press, Los Alamitos, CA, 793–801.

LA POUTRÉ, J. A. 1991. Dynamic graph algorithms and data structures. Ph.D. dissertation, Department of Computer Science, Utrecht University, Utrecht, The Netherlands.

NARDELLI, E., PROIETTI, G., AND WIDMAYER, P. 2001. Finding all the best swaps of a minimum diameter spanning tree under transient edge failures. *J. Graph Algorithm. Appl. 5*, 5, 39–57.

PECKHAM, S. 1989. Maintaining tree projections in amortized $O(\log n)$ time. Tech. rep. TR89-1034. Computer Science Deparment, Cornell University, Ithaca, NY.

POUTRÉ, J. A. L. 1992. Maintenance of triconnected components of graphs. In *Proceedings of the 19th ICALP*. Lecture Notes in Computer Science, vol. 623. Springer, Berlin, Germany, 354–365.

POUTRÉ, J. A. L. 1994. Alpha-algorithms for incremental planarity testing. In *Proceedings of the 26th STOC*. ACM Press, New York, NY, 706–715.

SLEATOR, D., AND TARJAN, R. 1983. A data structure for dynamic trees. *J. Comput. Syst. Sci. 26*, 3, 362–391. Announced at STOC'81.

SLEATOR, D., AND TARJAN, R. 1985. Self-adjusting binary search trees. *J. ACM 32*, 652–686.

TARJAN, R. 1997. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Math. Prog. 78*, 169–177.

TARJAN, R., AND WERNECK, R. 2005. Self-adjusting top trees. In *Proceedings of the 16th SODA*. ACM Press, New York, NY/SIAM Press, Philadelphia, PA, 813–822.

THORUP, M. 2001. Fully-dynamic min-cut. In *Proceedings of the 33rd STOC*. ACM Press, New York, NY, 224–230.

THORUP, M., AND ZWICK, U. 2005. Approximate distance oracles. *J. ACM 52*, 1, 1–24. Announced at STOC'01.

WESTBROOK, J., AND TARJAN, R. 1992. Maintaining bridge-connected and biconnected components on-line. *Algorithmica 7*, 433–464.

RECEIVED OCTOBER 2004; REVISED APRIL 2005; ACCEPTED APRIL 2005