Dynamic Trees in Practice

ROBERT E. TARJAN
Princeton University and HP Labs
and
RENATO F. WERNECK
Microsoft Research Silicon Valley

Dynamic tree data structures maintain forests that change over time through edge insertions and deletions. Besides maintaining connectivity information in logarithmic time, they can support aggregation of information over paths, trees, or both. We perform an experimental comparison of several versions of dynamic trees: ST-trees, ET-trees, RC-trees, and two variants of top trees (self-adjusting and worst-case). We quantify their strengths and weaknesses through tests with various workloads, most stemming from practical applications. We observe that a simple, linear-time implementation is remarkably fast for graphs of small diameter, and that worst-case and randomized data structures are best when queries are very frequent. The best overall performance, however, is achieved by self-adjusting ST-trees.

Categories and Subject Descriptors: E.1 [**Data Structures**]: Trees; G.2.1 [**Discrete Mathematics**]: Combinatorics—Combinatorial Algorithms; G.2.2 [**Discrete Mathematics**]: Graph Theory—Network Problems

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: dynamic trees, link-cut trees, top trees, experimental evaluation

1. INTRODUCTION

The *dynamic tree problem* is that of maintaining an n-vertex forest that changes over time. Edges can be added or deleted in any order, as long as no cycle is ever created. In addition, data can be associated with vertices, edges, or both. This data can be manipulated individually (one vertex or edge at a time) or in bulk, with operations that deal with an entire path or tree at a time. Typical operations include finding the minimum-cost edge on a path, adding a constant to the costs of all edges on a path, and finding the maximum-cost vertex in a tree. The dynamic tree problem has applications in network flows [Goldberg et al. 1991; Sleator and

Part of this work was done while the second author was at Princeton University. Research partially supported by the Aladdin Project, NSF Grant 112-0188-1234-12.
First author's address: Department of Computer Science, Princeton University, Princeton, NJ 08544, USA and Hewlett-Packard Laboratories, Palo Alto, CA 94034, USA. *ret@cs.princeton.edu.*
Second author's address: Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA. *renatow@microsoft.com.*

Tarjan 1983; Tarjan 1997], dynamic graphs [Cattaneo et al. 2002; Frederickson 1985; 1997a; Henzinger and King 1997; Radzik 1998; Zaroliagis 2002], and other combinatorial problems [Kaplan et al. 2003; Klein 2005; Langerman 2000].

Several well-known data structures can (at least partially) solve the dynamic tree problem in $O(\log n)$ time per operation: ST-trees [Sleator and Tarjan 1983; 1985], topology trees [Frederickson 1985; 1997a; 1997b], ET-trees [Henzinger and King 1997; Tarjan 1997], top trees [Alstrup et al. 1997; Alstrup et al. 2005; Tarjan and Werneck 2005; Werneck 2006], and RC-trees [Acar et al. 2004; Acar et al. 2005]. They all map arbitrary trees to balanced ones, but use different techniques to achieve this goal: path decomposition (ST-trees), linearization (ET-trees), and tree contraction (topology trees, top trees, and RC-trees). As a result, their relative performance depends significantly on the workload. We consider nine variants of these data structures. Section 2 presents a high-level description of each of them, including their limitations and some implementation issues. (A more comprehensive overview can be found in [Werneck 2006, Chapter 2].)

Our experimental analysis, presented in Section 3, includes three applications (maximum flows, online minimum spanning forests, and a simple shortest path algorithm), as well as randomized sequences of operations. Being relatively simple, these applications have dynamic tree operations as their bottleneck. We test different combinations of operations (queries and structural updates), as well as different types of aggregation (over paths or entire trees). The experiments allow for a comprehensive assessment of the strengths and weaknesses of each strategy and a clear separation between them. Section 4 summarizes our findings and compares them with others reported in the literature.

2. DATA STRUCTURES

2.1 ET-Trees

The simplest (and most limited) dynamic-tree data structure is the ET-tree [Henzinger and King 1997; Tarjan 1997]. This structure represents an arbitrary unrooted tree by an *Euler tour*, i.e., a tour that traverses each edge of the tree twice, once in each direction. The tour is a circular list, each element of which represents either an arc (one of the two occurrences of an edge) or a vertex of the forest. For efficiency, the list is broken at an arbitrary point and represented as a binary search tree, with the nodes (the elements of the list) appearing in symmetric (left-to-right) order. This allows edge insertions (*link*) and deletions (*cut*) to be implemented in $O(\log n)$ time as *joins* and *splits* of binary trees. Our implementation of ET-trees (denoted by ET) follows Tarjan's specification [Tarjan 1997] and uses splay trees [Sleator and Tarjan 1985]. A splay tree is a self-adjusting form of binary search tree with amortized performance guarantees. The basic operation on splay trees is *splay*, which moves a designated node to the root of its tree by doing a sequence of *rotations*, each of which is a local restructuring.

As described by Tarjan [1997], ET-trees associate values with vertices. Besides allowing queries on and updates to individual values, the ET-tree interface has operations to add a constant to all values in a tree and to find the minimum-valued vertex in a tree. These operations take $O(\log n)$ time if every non-root node stores its value in *difference form*, i.e., its value minus the value of its parent in the binary tree; this allows value changes at the root to implicitly affect all descendants. ET-trees can be adapted to support other types of queries, but they have a fundamental limitation: information can only be aggregated over trees. Efficient path-based aggregation is impossible because the two nodes representing an edge may be arbitrarily far apart in the data structure.

2.2 ST-Trees

The best-known dynamic tree data structures that support path operations are Sleator and Tarjan's ST-trees [1983; 1985]. Also known as *link-cut trees*, they predate ET-trees and were the first to support dynamic-tree operations in logarithmic time. ST-trees represent rooted trees. The basic structural operations are *link*(v, w), which creates an arc from a root v to a vertex w, and *cut*(v), which deletes the arc from v to its parent. The root can be changed by *evert*(v), which makes v the new root by reversing all arcs on the path from v to the previous root. Functions *findroot*(v) and *parent*(v) can be used to query the structure of the tree. As described in [Sleator and Tarjan 1985], ST-trees associate a cost with each vertex v, retrievable by the *findcost*(v) function. Two operations deal with the path from v to the root of its tree: *addcost*(v, x) adds x to the cost of every vertex on this path, and *findmin*(v) returns the minimum-cost vertex on the path.

The obvious implementation of the ST-tree interface is to store with each node its value and a pointer to its parent. This supports *link*, *cut*, *parent* and *findcost* in constant time, but *evert*, *findroot*, *findmin* and *addcost* must traverse the entire path to the root. Despite the linear-time worst case, this representation might be good enough for graphs with small diameter, given its simplicity. We tested two variants of this implementation, LIN-V and LIN-E; the former associates costs with vertices, the latter with edges. Both store values at nodes, but LIN-E interprets such a value as the cost of the arc to the parent. The values must therefore be moved during *evert*.

To achieve sublinear time per operation, Sleator and Tarjan propose an indirect representation that partitions the underlying tree into vertex-disjoint *solid paths* joined by *dashed edges*. Each solid path is represented by a binary search tree in which the original nodes appear in symmetric order. These binary trees are then "glued" together to create a single *shadow tree*: the root of a binary tree representing a path P becomes a *middle child* (in the shadow tree) of the parent (in the original forest) of the topmost vertex of P. See Figure 1.

To manipulate a path from v to the root, the data structure first *exposes* it, i.e., changes the partition of the tree (by a series of *joins* and *splits* of binary trees) so that the unique solid path containing the root starts at v. Standard binary tree operations can then be applied to the exposed path to implement *findroot*, *parent* and *findmin* in logarithmic time. The *evert* operation requires reversing the left/right pointers of all nodes on the exposed path, which can be done in constant time (once the path is exposed) if a *reverse* bit is stored in difference form. Similarly, *addcost* can be performed on the exposed path in constant time.

When paths are represented as splay trees, *expose* (and thus the other dynamic tree operations) take $O(\log n)$ amortized time [Sleator and Tarjan 1985]. An *expose* is done in three passes: the first does a splay within each binary tree representing a part of the path to be exposed; the second changes the partition so that the path to

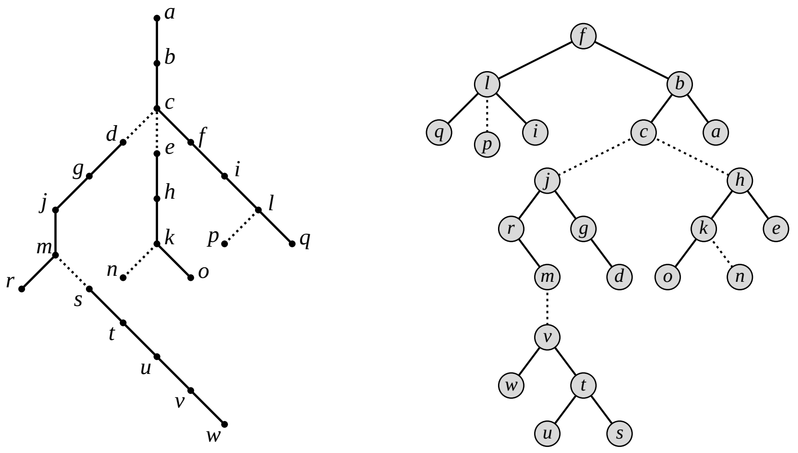

Fig. 1. An ST-tree (adapted from [Sleator and Tarjan 1985]). On the left is the original tree, rooted at a and already partitioned into vertex-disjoint paths. On the right is the corresponding shadow tree. In both cases, edges within the same path are solid, and those connecting different paths are dashed.

be exposed is a single solid path; the third does a final splay on the tree representing that path. A worst-case logarithmic bound is achievable with globally biased search trees [Sleator and Tarjan 1983]. In this case, the partition into solid paths depends only on the structure of the input tree (and not on the sequence of operations previously performed). This requires each *expose* to be followed a *conceal* operation, which restores an acceptable partition. Moreover, each node must maintain several extra fields to ensure that the tree is balanced. We have not implemented this solution, since we expect it to be much slower than the self-adjusting version (the authors themselves consider it "mainly of theoretical interest").

We call the splay-based implementation used in our experiments ST-V. It stores costs on nodes. For applications with costs on edges, we can interpret the value stored at v as the cost of the edge between v and its parent in the original tree. This is only efficient, however, if *evert* is never called, since this operation changes the parent of several nodes at once. Supporting *evert* with costs on edges requires maintaining additional nodes to represent the edges explicitly. Our implementation of this variant, ST-E, uses ST-V as the underlying data structure. More precisely, we implicitly insert one extra vertex in the middle of each original edge and represent the augmented forest with ST-V. With the costs of the original vertices set to infinity, *findmin* always returns the node associated with the appropriate edge.

ST-trees can be modified to support other types of queries, as long as information is aggregated over paths only. Aggregation over arbitrary trees would require traversing the ST-tree in a top-down fashion, which is impossible because nodes do not maintain pointers to their (potentially numerous) middle children. A solution is to apply *ternarization* to the underlying forest, which replaces each high-degree vertex by a chain of low-degree ones [Goldberg et al. 1991; Kaplan et al. 2003; Klein 2005; Langerman 2000; Radzik 1998].

2.3 Tree Contraction

A third approach used by dynamic tree data structures is to represent a *contraction* of the tree, as done by *topology trees*, *RC-trees*, and *top trees*. We concentrate on the most general, top trees, and briefly discuss the other two.

Top trees were introduced by Alstrup et al. [1997; 2005], but we borrow the notation used by Tarjan and Werneck [2005]. The data structure represents free (unrooted) trees with sorted adjacency lists (i.e., the edges adjacent to each vertex are arranged in some fixed circular order, which can be arbitrary). A *cluster* represents both a subtree and a path of the original tree. Indeed, we refer to a cluster by the endpoints of the corresponding path: cluster (v, w) represents the path between v and w. Each original edge of the graph is a *base cluster*. A *tree contraction* is a sequence of local operations that successively pair up these clusters until a single cluster remains. The *top tree* is merely the binary tree representing the contraction. If two clusters (u, v) and (v, w) share a degree-two endpoint v, they can be combined into a *compress cluster* (u, w). Also, if (w, x) is the successor of (v, x) (in the circular order around x) and v has degree one, these clusters can be combined into a *rake cluster*, also with endpoints w and x. See Figure 2. Each *rake* or *compress* cluster can be viewed as a parent that aggregates the information contained in its children. It represents both the subtree induced by its descendants and the path between its two endpoints, and can also be viewed as a higher-level edge. The root cluster of the top tree represents the entire underlying tree. Whenever there is a *link* or *cut*, the data structure merely updates the contractions affected.

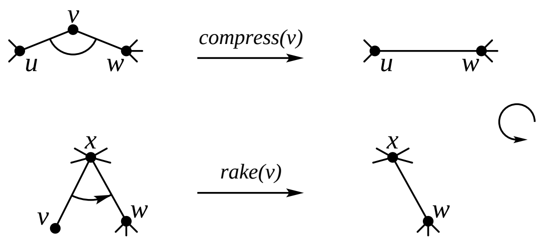

Fig. 2. Pairing up clusters in top trees: *compress* combines the edges adjacent to a degree-two vertex, and *rake* pairs a leaf edge with its successor (in counterclockwise order in the figure).

An important feature of the top tree interface is that it decouples the contraction itself from the values that must be manipulated. The data structure decides which operations (*rake* or *compress*) must be performed, but updates no values on its own. Instead, it calls user-defined *internal functions* to handle value updates whenever a pairing of clusters is performed (*join*) or undone (*split*).[1] The interface also stipulates that these calls are only made when all clusters involved are *roots* of (possibly partial) top trees—none of the input clusters has a parent. This makes the implementation of the internal functions easier, but it may hurt performance because the top tree cannot be updated in a single pass. An update performs a top-down series of *splits* followed by a bottom-up sequence of *joins*.

[1] These operations on clusters should not be confused with *joins* and *splits* of binary trees.

To perform a query, the user calls the *expose*(v, w) operation, which returns a root cluster having v and w as end vertices (or *null*, if v and w are in different trees). Note that, even if v and w are in the same tree, *expose* may need to change the contraction to ensure that the root cluster actually represents the path from v to w. In principle, the internal functions should be defined so that the cluster returned by *expose* automatically contains the answer to the user's query, so there is no need to actually traverse the tree. This makes it easy to implement the internal functions and algorithms that use top trees. As our experiments show, however, changing the tree during queries can have a significant effect on performance.

Top trees support aggregation over paths or trees directly, with no degree limitation. In particular, they naturally support applications that require both types of aggregation, such as maintaining the diameter, the center, or the median of a tree [Alstrup et al. 2005].

The first contraction-based data structures were in fact not top trees but Frederickson's *topology trees* [1985; 1997a; 1997b]. They interpret clusters as *vertices* instead of edges. This leads to a simpler contraction algorithm, but it requires all vertices in the underlying forest to have degree bounded by a constant. Although ternarization can remedy this, it is somewhat inelegant and adds an extra layer of complexity to the data structure. Recently, Acar et al. [2004; 2005] invented *RC-trees*, which can be seen as a simpler, randomized version of topology trees. RC-trees also require the underlying tree to have bounded degree, and use space proportional to the bound (following [Acar et al. 2005], we set the bound to eight in our experiments). We call the implementation (by Acar et al. [2005]) of RC-trees used in our experiments RC. The name "RC-trees" is a reference to the *rake* and *compress* operations, which were introduced by Miller and Reif [1985] in the context of parallel algorithms on trees.

RC-trees have a generic interface to separate value updates from the actual contraction. Unlike top trees, queries perform a traversal of the tree, but keep its structure unchanged. This approach makes queries faster, but requires extra implementation effort from the user and is less intuitive than a simple call to *expose* (as in top trees). In addition, the interface assumes that the underlying tree has bounded degree: with ternarization, the interface will be to the transformed tree, with dummy vertices.

2.3.1 *Implementing Top Trees.* Alstrup et al. [2005] proposed implementing top trees not as a standalone data structure, but as a layer on top of topology trees. Given the complexity of topology trees, this extra layer (which may as much as double the depth of the contraction) is undesirable. Recently, Holm, Tarjan, Thorup and Werneck proposed a direct implementation that still guarantees $O(\log n)$ worst-case time without topology trees. (Details, still unpublished, can be found in [Werneck 2006].) The data structure represents a natural contraction scheme: it works in rounds, and in each round performs a maximal set of independent pairings (i.e., no cluster participates in more than one pair). Level zero consists of all base clusters (the original edges). Level $i + 1$ contains *rake* and *compress* clusters with children at level i, with *dummy clusters* added as parents of unpaired level-i clusters. Figure 3 shows a contraction and the corresponding top tree.

After a *link*, *cut*, or *expose*, the contraction can be updated in $O(\log n)$ worst-case

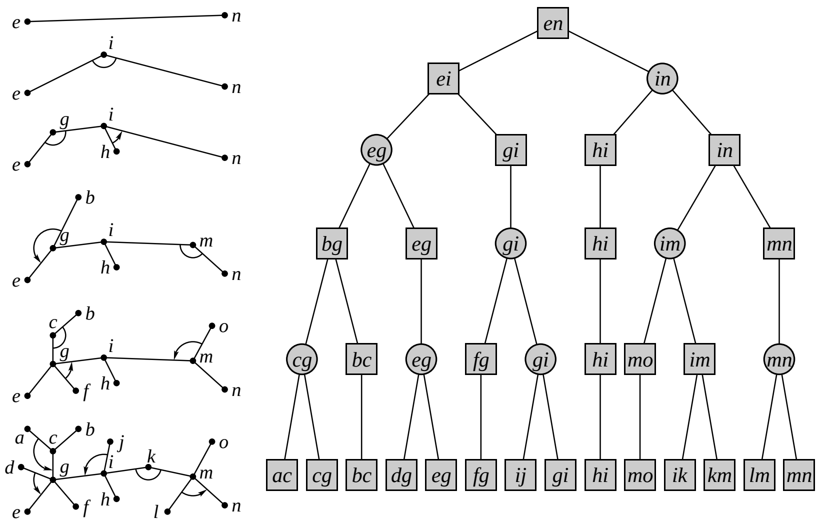

Fig. 3. A contraction (left, to be read bottom-up) and the corresponding top tree (right). Circles represent rake nodes, and squares represent base (no child), dummy (one child) or compress (two children) nodes.

time [Werneck 2006]. In practice, however, this implementation (which we refer to as TOP-W) has some important drawbacks. First, to preserve the circular order, each level maintains a linked list representing an Euler tour of its clusters, which makes updating the contraction expensive. Second, even though a "pure" top tree representing an n-vertex forest will have no more than $2n$ clusters, when dummy clusters are taken into account this number might be as large as $6n$. As a result, TOP-W uses considerably more memory than simpler data structures.

To overcome these drawbacks, Tarjan and Werneck [2005] proposed a self-adjusting implementation of top trees (which we call TOP-S) that supports all operations in $O(\log n)$ amortized time. It partitions a tree into maximal edge-disjoint paths, each represented as a *compress tree* (a binary tree of *compress* clusters with base clusters as leaves). Each subtree incident to a path is represented recursively as a binary tree of *rake* clusters (which is akin to ternarization, but transparent to the user), and its root becomes a middle child of a *compress* node. This is the same basic approach as ST-trees, but ST-trees represent vertex-disjoint paths, have no embedded ternarization (which prevents them from supporting aggregation over trees), and do not support circular adjacency lists directly.

3. EXPERIMENTAL RESULTS

3.1 Experimental Setup

This section presents an experimental comparison of the data structures discussed above (and summarized in Table I): ET-trees (ET), self-adjusting top trees (TOP-S), worst-case top trees (TOP-W), RC-trees (RC), and ST-trees implemented both with splay trees (ST-V/ST-E) and just with parent pointers (LIN-V/LIN-E). We tested these data structures on algorithms for three problems: maximum flows, minimum spanning trees, and shortest paths on arbitrary graphs. Given our emphasis on the underlying data structures, we did not test more elaborate dynamic graph algorithms, in which dynamic trees are typically just one of several components; [Zaroliagis 2002] surveys this topic.

Table I. Dynamic tree implementations compared in our study.

Implementation	Description	Bound per operation
ET	self-adjusting ET-trees	$O(\log n)$ amortized
LIN-V	node-based ST-trees with parent pointers	$O(n)$ worst-case
LIN-E	edge-based ST-trees with parent pointers	$O(n)$ worst-case
RC	RC-trees	$O(\log n)$ randomized
ST-V	node-based self-adjusting ST-trees	$O(\log n)$ amortized
ST-E	edge-based self-adjusting ST-trees	$O(\log n)$ amortized
TOP-Q	query-oriented contraction-based top trees	$O(\log n)$ worst-case
TOP-S	self-adjusting top trees	$O(\log n)$ amortized
TOP-W	contraction-based top trees	$O(\log n)$ worst-case

All algorithms were implemented in C++ and compiled with `g++` 3.4.4 with the `-O4` (full optimization) option. We ran the experiments on a Pentium 4 running Microsoft Windows XP Professional at 3.6 GHz, with 16 KB of level-one data cache, 2 MB of level-two cache, and 2 GB of RAM. Almost all data structures were implemented by the authors and are available upon request. The exception is RC-trees, available at *http://www.cs.cmu.edu/∼jvittes/rc-trees/*, and implemented by Acar, Blelloch, and Vittes [2005]. We tested RC-trees only on online minimum spanning forests, readily supported by the code provided.

CPU times were measured with the `getrusage` function, which has precision of 1/60 second. We ran each individual computation repeatedly (within a single loop in the program) until the aggregate time (measured directly) was at least two seconds, then took the average. The timed executions were preceded by a single untimed run, used to warm up the cache (thus ensuring a fair comparison among the algorithms). Running times do not include generating or reading the input data (which is done only once by the entire program), but include the time to allocate, initialize, and destroy the data structure (each done once per run within the program). For each set of parameters, we report the average results from five inputs, with different seeds for the pseudorandom generator.

To ensure uniformity among our implementations, we reused code whenever possible. In particular, routines for splaying were implemented only once (as template functions) and used by TOP-S, ST-E, ST-V, and ET. To update values, each data structure defines an inline function that is called by the splaying routine whenever there is a rotation. Also, the user-defined functions used by top trees were implemented as templates (thus allowing them to be inlined) and were shared by both top tree implementations. Values were stored as 32-bit integers. At initialization time, each data structure allocates all the memory it might need as a single block, which is managed by the data structure itself (the only exception is the RC-tree implementation, which allocates memory as needed in large blocks, and frees it all at once). All executions fit in RAM, unless specifically noted.

3.2 Maximum Flows

One of the original motivations for dynamic tree data structures was the *maximum flow problem* (see, e.g., [Ahuja et al. 1993]). Given a directed graph $G = (V, A)$ (with $n = |V|$ and $m = |A|$) with capacities on the arcs, a *source* s and a *sink* t, the goal is to send as much flow as possible from s to t. We implemented what Ahuja et al. [1993] call the *shortest augmenting path algorithm.* In each iteration, it finds a path with positive residual capacity that has the fewest residual arcs and sends as much flow as possible along this path. The algorithm stops when no such augmenting path exists. Augmenting along residual paths of fewest arcs was proposed by Edmonds and Karp [1972], who proved that the number of augmentations is $O(nm)$. In order to find shortest augmenting paths efficiently, the algorithm maintains a *distance label* $d(v)$ for each vertex v that is a lower bound on the number of arcs on a residual path from v to t. An arc (v, w) is *admissible* if $d(v) = d(w) + 1$. The algorithm grows a path of admissible arcs from s until either reaching t, when it augments along the path, or reaching a vertex v with no outgoing admissible arc, when it increases $d(v)$ and backs up. The algorithm is very similar to that of Dinic [1970] and has the same running time, $O(n^2 m)$; Dinic's algorithm differs in that it maintains exact distance labels and recomputes them globally rather than locally.

The running time can be reduced to $O(mn \log n)$ if we use logarithmic-time dynamic trees to maintain a forest of admissible arcs (with LIN-V or LIN-E, the total running time will not improve). The modified algorithm always processes the root v of the tree containing the source s. If v has an admissible outgoing arc, the arc is added to the forest (by *link*); otherwise, all incoming arcs into v are *cut* from the forest. Eventually s and t will belong to the same tree; flow is sent along the s-t path by decrementing the capacities of all arcs on the path and *cutting* the arcs whose capacities drop to zero. See [Ahuja et al. 1993] for details, including a comparison with Dinic's algorithm. Note that this algorithm does very little besides dynamic tree operations, making it ideal for comparing the data structures in our study.

The operations supported by the ST-tree interface (such as *addcost*, *findmin*, *findroot*) are, by construction, exactly those needed for this implementation. With top trees, we have to define what to store in each cluster. We chose to make each cluster $C = (v, w)$ represent both a *rooted* tree and a *directed* path between v and w. The cluster stores the root of the subtree it represents, a pointer to the minimum-capacity arc on the path between v and w (or *null*, if the root is neither v nor w), the actual capacity of this arc, and a "lazy" value to be added to the capacities of all subpaths of $v \cdots w$ (this supports the equivalent of ST-tree's *addcost*). We also implemented the full ST-tree interface on top of top trees, as suggested in [Alstrup et al. 1999], but it was up to twice as slow as the direct method. Detailed results for this variant are thus omitted.

We ran our tests on synthetic graph classes with well-defined structure, which allows for a better understanding of the behavior of each implementation. Our first experiment is on *random layer graphs* [Anderson 1993], parameterized by the number of rows (r) and columns (c). Each vertex in column i has outgoing arcs to three random vertices in column $i + 1$, with integer capacities chosen uniformly at random from $[0, 2^{16}]$. In addition, the source s is connected to all vertices in column 1, and all vertices in column c are connected to the sink t (in both cases, the arcs have infinite capacity). We used $r = 4$ (thus making all augmenting paths have $\Theta(n)$ length) and varied c from 128 to 16 384. Figure 4 reports average running times normalized with respect to ST-V.

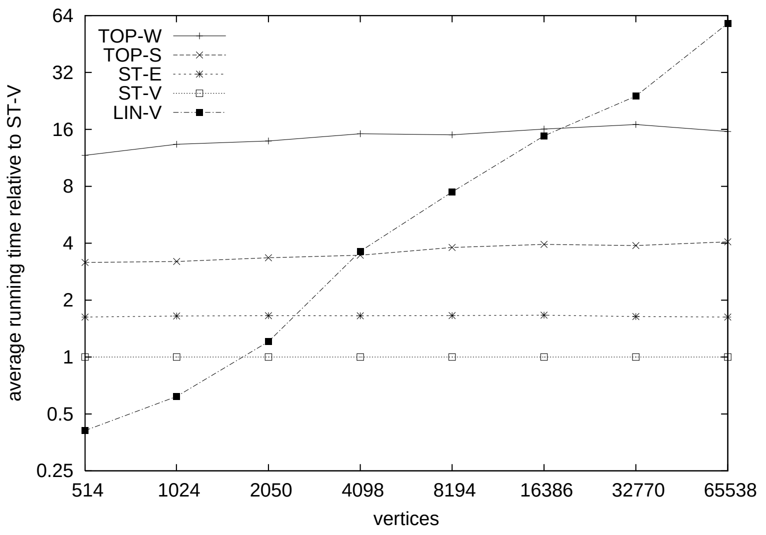

Fig. 4. Maximum flows on layer graphs with four rows and varying numbers of columns.

Our second experiment is on directed meshes, also parameterized by the number of rows (r) and columns (c). A mesh consists of a source s, a sink t, and an $r \times c$ grid. Each grid vertex has outgoing arcs to its (up to four) neighbors[2] with random integer capacities in the range $[0, 2^{16}]$. The grid is circular: the first and last rows are considered adjacent. In addition, there are infinite-capacity arcs from s to the first column, and from the last column to t. We kept the product of r and c constant at 2^{16} and varied their ratio. Figure 5 reports average running times relative to ST-V.

For both graph families, the $O(\log n)$ data structures have similar relative performance: ST-trees are the fastest, followed by self-adjusting top trees and, finally, worst-case top trees. Although there are costs (capacities) on arcs, ST-V can be used because *evert* is never called: we can interpret the cost stored at node v as the capacity of the arc to its parent. ST-E, included in the experiments for comparison, is slightly slower because it uses extra nodes to represent the arcs. With the linear-time data structure (LIN-V), the maximum flow algorithm is asymptotically worse, running in $O(n^2m)$ time. Being quite simple, the algorithm is still the fastest for small trees, but is eventually surpassed by ST-V. On random layer graphs, this happens when augmenting paths have roughly 500 vertices; for directed meshes, both

[2]A similar description was mistakenly given for the "directed meshes" tested in [Werneck 2006]; those graphs, obtained with a different generator [Anderson 1993], were actually acyclic.

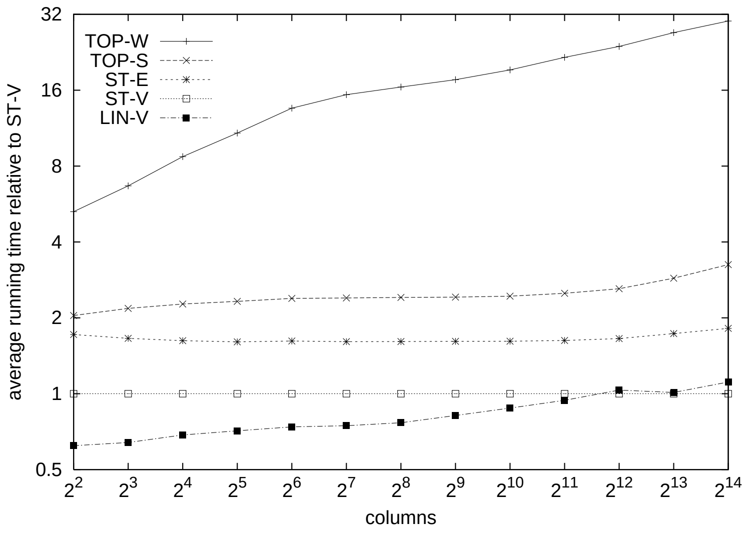

Fig. 5. Maximum flows on meshes with 65 538 vertices and varying numbers of columns (and rows).

algorithms still have comparable running times when augmenting paths have more than 16 384 vertices. With so many columns, according to our experiments the average number of *links* between augmentations is almost 9000 on directed meshes, but only four on layer graphs. This means that on directed meshes (but not layer graphs) a typical operation is on a very small path, which explains why the linear algorithm remains competitive.

With LIN-V, the maximum flow algorithm performs the same basic operations as when implemented with no dynamic tree data structure at all. Curiously, however, in early experiments we noticed that it was marginally faster with the data structure, presumably because of better cache locality: traversing a path in the data structure is cheaper than in the full graph.

In the maximum flow algorithm, every query is soon followed by a structural update (*link* or *cut*). Next, we consider an application in which queries can vastly outnumber structural updates.

3.3 Online Minimum Spanning Forests

The *online minimum spanning forest problem* is that of maintaining the minimum spanning forest (MSF) of an n-vertex graph to which m edges are added one by one. If we use dynamic trees to maintain the MSF, each new edge can be processed in $O(\log n)$ time. Let $e = (v, w)$ be a new edge added to the graph. If v and w belong to different components of the MSF, we simply add (*link*) e to it. Otherwise, we find the maximum-cost edge f on the path from v to w. If it costs more than e, we remove (*cut*) f from the forest and add (*link*) e instead. This is a straightforward application of Tarjan's "red rule": if an edge is the most expensive in *some* cycle in the graph, then it does not belong to the minimum spanning forest [Tarjan 1983].

To find the maximum-cost edge of an arbitrary path with ST-trees, we simply maintain the negative of the original edge costs and call *findmin*. Because the *evert* operation is required to maintain unrooted trees, we must use ST-E in this case. With top trees, it suffices to maintain in each cluster $C = (v, w)$ a pointer to the maximum-cost base cluster on the path from v to w, together with the maximum cost itself.

Our first experiment is on random graphs: edges are random pairs of distinct vertices with integer costs picked uniformly at random from $[1, 1000]$. We varied n from 2^{10} to 2^{20} and set $m = 8n$. With this density, we observed that roughly 37% of the edges processed by the algorithm are actually inserted; the others generate only queries. Figure 6 shows the average time necessary to process each edge. For reference, it also reports the time taken by Kruskal's algorithm, which is *offline*: it sorts all edges (with quicksort) and adds them to the solution one at a time using a union-find data structure to detect cycles.

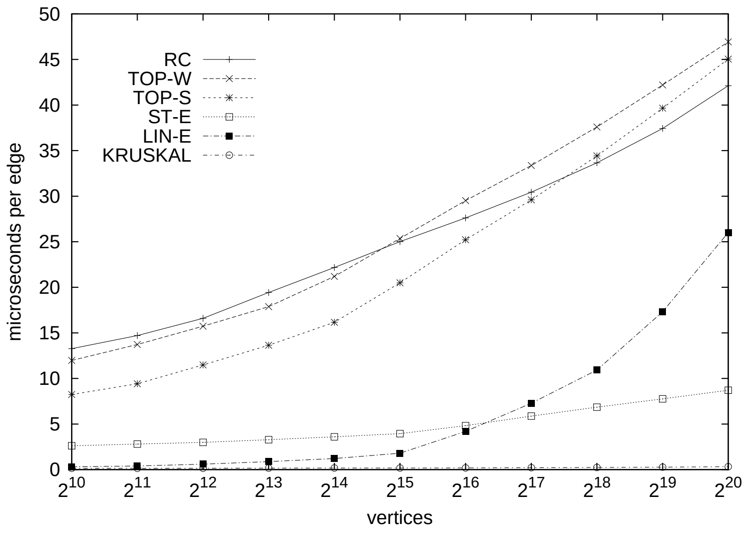

Fig. 6. Online minimum spanning forests on random graphs with average degree 16.

Being much simpler, Kruskal's algorithm is roughly 20 times faster than ST-E, the best dynamic tree data structure. Both implementations of top trees, as well as RC-trees, have worse performance.[3] Although LIN-E is faster than ST-E on small graphs, it has worse asymptotic performance because in our experiments the average path has superlogarithmic length. This is consistent with a theoretical result [Addario-Berry et al. 2006] that the expected diameter of the minimum spanning tree of a complete graph with random edge weights is $\Omega(\sqrt[3]{n})$.

[3]Recall that RC-trees do not support vertices with degree greater than eight (which are very rare but sometimes occur in our experiments). Our implementation of the MSF algorithm with RC checks if an endpoint of the edge about to be inserted already has degree eight; if it does, the edge just removed (if there is any) is reinserted instead. Although the modified algorithm may not maintain the exact MSF, its running time is mostly unaffected for the graph families we tested.

Our second experiment also involves random graphs, but now we fix $n = 2^{16}$ and vary the average vertex degree from 4 to 512 (i.e., we vary m from 2^{17} to 2^{24}). As the density increases, relatively fewer *links* and *cuts* are performed: when the average degree is 512, only roughly 2.5% of the input edges are actually inserted into the MSF. As Figure 7 shows, the average time to process an edge can actually decrease with the density, as queries dominate the running time. The speedup is more pronounced for TOP-W and RC, since the self-adjusting data structures (ST-E and TOP-S) must change the tree even during queries.

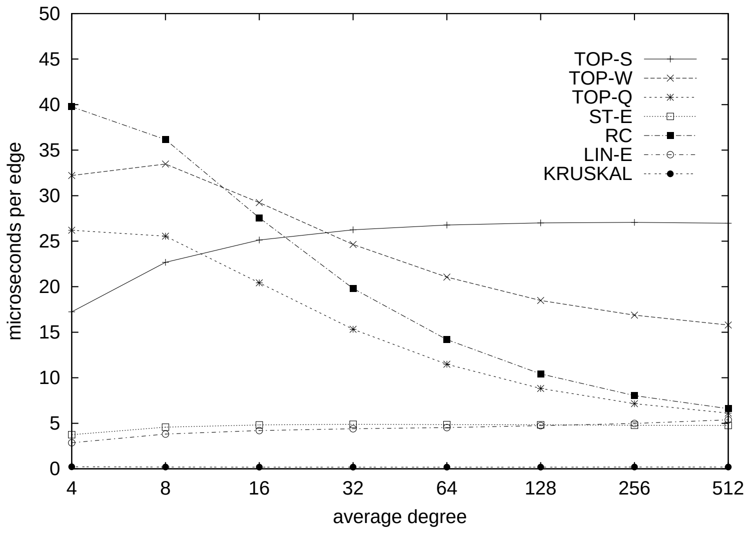

Fig. 7. Online minimum spanning forests on random graphs with 65 536 vertices.

Recall that TOP-W must change the contraction when performing queries (*expose*) to ensure that the relevant path is represented at the root of its top tree. In principle, this would make *exposes* about as expensive as *links* and *cuts*. As suggested by Alstrup et al. [2005], however, our implementation of *expose* only marks some existing top tree nodes as "invalid" and builds a temporary top tree with the $O(\log n)$ root clusters that remain. The original top tree is restored as soon as some other operation is performed. Building a temporary top tree eliminates the need for expensive updates (such as repairing the Euler tour of the clusters in each level) during queries. Fast queries help explain why TOP-W is more competitive with TOP-S for the online MSF application than it is for maximum flows. In addition, consecutive dynamic tree operations are correlated in the maximum flow application, which tends to benefit self-adjusting data structures such as TOP-S.

The RC-tree method does not modify the tree (even temporarily) during queries: instead, it traverses the tree in a bottom-up fashion, aggregating information contained in internal nodes. This explains why RC-trees become significantly faster than TOP-W as queries become more frequent. For comparison, we have implemented TOP-Q, a variant of TOP-W that explicitly traverses the tree during queries, with no calls to *expose*. Technically, TOP-Q is not an implementation of top trees, since it violates the top tree interface. As Figure 7 shows, though, it is significantly faster than TOP-W when queries are numerous, and about as fast as RC. Speed comes at a cost, however: implementing a different query algorithm for each application is much more complicated (and less intuitive) than simply calling *expose*.

To further assess query performance, we tested the algorithms on *augmented random graphs*. A graph with n vertices, m edges, and *core size* $c \leq n$ is created in four steps:

(1) Generate a random spanning tree on c vertices: starting with no edge, we add random edges one by one (discarding those that create a cycle) until a single component remains.

(2) Progressively transform the original edges into paths, until there are n vertices in total. This is done by assigning (uniformly at random) each of the $n - c + 1$ remaining vertices to an original edge. Eventually, an edge to which k vertices are assigned is split into $k + 1$ segments.

(3) Randomly permute the vertex labels (to avoid biases during the online minimum spanning tree computation).

(4) Add $m - n + 1$ random edges to the graph, each with cost n.

Costs are assigned so that only the first $n - 1$ edges (which are processed first by the online MSF algorithm, in random order) result in *links*; the remaining edges result in queries only. Figure 8 shows the performance of various algorithms with $n = 2^{13}$, $m = 2^{18}$, and c varying from 2 to 2^{13}. The average length of the paths queried is inversely proportional to the core size; when the length drops below roughly 100, LIN-E becomes the fastest online algorithm (surpassing TOP-Q, which is particularly fast because more than 95% of the operations are queries). The crossover point between LIN-E and ST-E is closer to 150; this is also true in the experiment illustrated in Figure 6.

Finally, we investigate how caching affects the performance of the data structures. We ran the MSF algorithm on graphs consisting of $32c$ vertices (for a given parameter c) randomly partitioned into c equal-sized components. Edges are inserted into the graph by first picking a random component, then a random pair of vertices within it. We stop after $128c$ edges are added, so that each component has 128 edges on average. The sequence of edges to be added is generated offline.

Since the size of each component is fixed, in theory the average time per operation should not depend on c. As Figure 9 shows, however, cache effects cause all data structures to become slower as the number of components increases. Interestingly, LIN-E has the most noticeable slowdown: almost a factor of eight, compared to around two for other data structures. It benefits the most from caching when processing very small instances, since it has the smallest footprint per node (only 8 bytes). This is significantly less than ST-E (57 bytes), TOP-S (224), TOP-W (579), and RC (roughly one kilobyte). In fact, RC-trees even ran out of RAM for the largest graph tested (this is the only case reported in this paper in which this happened—all other tests ran entirely in memory). The excessive memory usage of this particular implementation of RC helps explain why it is consistently slower than worst-case top trees, despite being somewhat simpler.

Even though Figure 9 shows an extreme case, cache effects should not be disre-

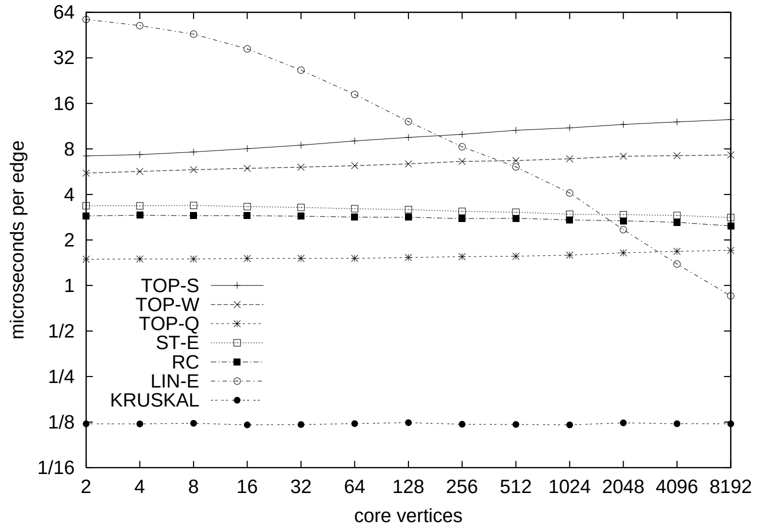

Fig. 8. Online minimum spanning forests on augmented random graphs.

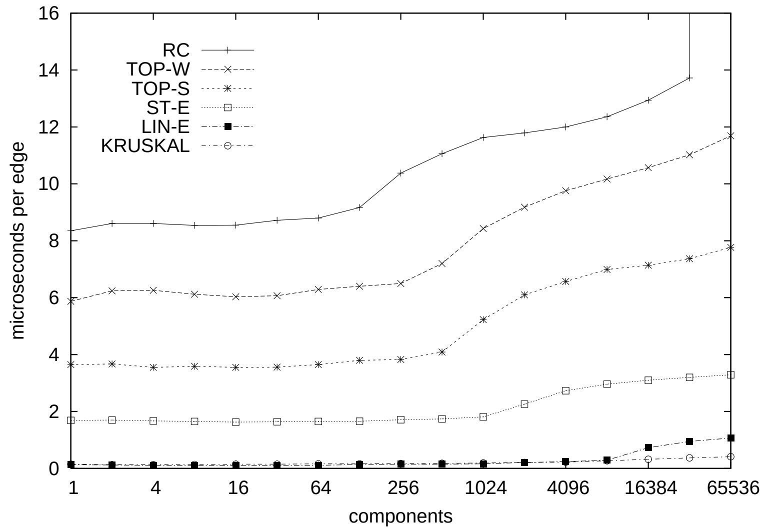

Fig. 9. Cache effects on forests with 32-vertex random components.

garded. Take, for instance, the layer graphs used in the maximum flow application. The graph generator assigns similar identifiers to adjacent vertices. This gives path traversals strong locality, since within our data structures the vertices are ordered by their identifiers. Randomly permuting the identifiers in the input slows down all algorithms, but LIN-V (which uses only 8 bytes per node) is affected the most. On layer graphs with 65 538 vertices, we observed that running times increase by 150% for LIN-V, 40% for ST-V (which uses 24 bytes per node), and only 11% for TOP-W.

3.4 Single-Source Shortest Paths

The applications considered so far require dynamic trees that aggregate information over paths. We now test an application that aggregates over trees: a single-source shortest path algorithm. Given a directed graph $G = (V, A)$ (with $|V| = n$ and $|A| = m$), a length function ℓ, and a *source* $s \in V$, it either finds the distances between s and every vertex in V or detects a negative cycle (if present).

The Bellman-Ford-Moore (BFM) algorithm [Bellman 1958; Ford 1956; Moore 1959] maintains a *distance label* $d(v)$ for every vertex v, representing an upper bound on its distance from s (initially zero for s and infinity otherwise). It also maintains a tentative shortest path tree spanning all vertices with finite distance label. The algorithm iterates over a fixed list of the arcs. If an arc $(v, w) \in A$ is such that $d(v) + \ell(v, w) < d(w)$, the algorithm *relaxes* it by setting $d(w) \leftarrow d(v) + \ell(v, w)$ and making v the parent of w in the tentative shortest path tree. The algorithm can stop after a pass in which no arc is relaxed. If this does not happen after n passes, the graph must have a negative cycle. In the worst case, BFM takes $O(mn)$ time.

After an arc (v, w) is relaxed, we could immediately decrease the distance labels of all descendants of w in the current candidate shortest path tree. The BFM algorithm will eventually do this, but it may take several iterations. With a dynamic tree data structure that supports aggregation over trees (such as ET-trees or top trees), we can perform such an update in $O(\log n)$ time. Although dynamic trees increase the worst-case complexity of the algorithm to $O(mn \log n)$, one can expect fewer iterations to be performed in practice.

We tested this algorithm on graphs consisting of a Hamiltonian circuit (corresponding to a random permutation of the vertices) augmented with random arcs. We used $m = 4n$ arcs, n of which belong to the Hamiltonian circuit. All arcs have lengths picked uniformly at random; those on the cycle have lengths in the interval $[1, 10]$, and the others have lengths in $[1, 1000]$.

Figure 10 shows the average time each method takes to process an arc. (ST-tree implementations are omitted because they cannot aggregate information over arbitrary trees.) ET-trees are much faster than both versions of top trees (TOP-W and TOP-S), and about as fast as TOP-Q, which explicitly traverses the tree during queries instead of calling *expose*. In these experiments, only 10% of the arcs tested result in structural updates. This makes TOP-W competitive with TOP-S, and TOP-Q competitive with ET (which is much simpler).

The standard version of the Bellman–Ford–Moore algorithm, which maintains a single array and consists of one tight loop over the arcs, can process an arc up to 600 times faster than ET. Even though dynamic trees reduce the number of iterations, they do so by a factor of at most four (for $n = 262\,144$, the algorithm requires 14.4 iterations to converge with dynamic trees and 56.6 without). As a result, ET is 100 to 200 times slower than BFM.

It should be noted that the only purpose of the experiment is to compare the performance of the data structures to one another. This is obviously a poor application of dynamic trees, since the straightforward algorithm is trivial and has better running time. In fact, BFM itself is not among the fastest algorithms for this problem (see e.g. [Cherkassky and Goldberg 1999; Cherkassky et al. 2008]).

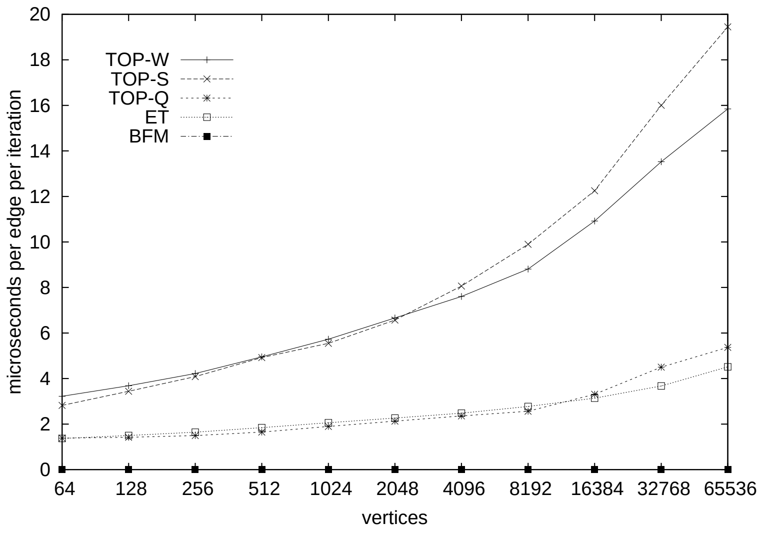

Fig. 10. Single-source shortest paths on graphs with Hamiltonian circuits.

3.5 Random Structural Updates

In order to compare all data structures at once, we consider a sequence of m operations consisting entirely of *links* and *cuts*, with no queries. We start with $n-1$ *links* that create a random spanning tree. We then execute a sequence of $m-n+1$ alternating *cuts* and *links*: we remove a random edge from the current tree and replace it with a random edge between the two resulting components. (Due to the limitations of RC, only vertices with degree smaller than eight are candidates to be endpoints of the new edge.) We fixed $m = 10n$ and varied n from 2^6 to 2^{18}. For implementations of the ST-interface, every *link* or *cut* is preceded by the *evert* of one of the endpoints. Even though there are no queries, values were still appropriately updated by the data structure (as if we were maintaining the MSF). Figure 11 shows the average time to execute each operation of the precomputed sequence. The results are in line with those observed for previous experiments: the more memory a method uses, the slower it tends to be. ST-trees are the fastest logarithmic-time data structure, followed by ET-trees (which uses 89 bytes per node), self-adjusting top trees, worst-case top trees, and RC-trees.

3.6 Additional Observations

An efficient implementation of the *evert* operation on ST-trees requires each node to store a *reverse bit* (in difference form), which implicitly swaps left and right children. Our implementation of LIN-V always supports *evert*, even in experiments in which it is not needed (such as the maximum flow algorithm). Preliminary tests show that a modified version of LIN-V with no support for *evert* is roughly 5% faster in the maximum flow application. Also, as observed by Philip Klein (personal communication), an additional speedup of at least 10% can be obtained with a more specialized implementation of splaying that delays value updates until

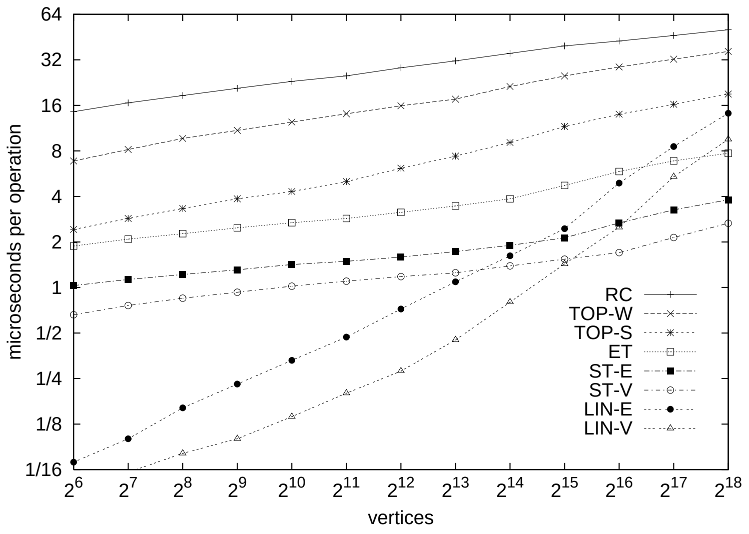

Fig. 11. Average time per operation on randomized sequences of *links* and *cuts*.

they are final (our current implementation does each rotation separately, updating all values). In an extreme case, if we do not update values at all during rotations, we observed that ST-V becomes almost 20% faster on a random sequence of *links* and *cuts*. The main reason is improved locality, since value updates require looking outside the splaying path. Such a simplified data structure could still be used to maintain connectivity information.

The performance of the data structures also depends on how much data is stored in each node. If we were to store values as 64-bit `doubles` (instead of 32-bit integers), all data structures would be slightly slower, but more compact ones would be affected the most. For random *links* and *cuts*, 64-bit values slow down ST-V by at least 10% and TOP-W by only 1%.

4. FINAL REMARKS

We have shown that the linear-time implementation of the ST-tree interface can be significantly faster than other methods when the paths queried have up to a few hundred vertices, but they become impractical as path sizes increase. Alstrup et al. [1999] observed the same for sequences of operations chosen according to a natural distribution. Recently, Ribeiro and Toso [2007] have used the linear-time data structure as a building block for a simple method to maintain fully dynamic minimum spanning trees, which can be competitive with more elaborate algorithms for some graphs.

Among the logarithmic-time data structures, the self-adjusting implementation of ST-trees is generally the fastest, especially when *links* and *cuts* are numerous. It is relatively simple and can benefit from correlation of consecutive operations, as in the maximum flow application. Self-adjusting top trees are slower than ST-trees by a factor of up to four, but often much less. These are reasonably good results, given how much more general top trees are: our implementation supports sorted adjacency lists and aggregation over trees. As explained in [Tarjan and Werneck 2005], the data structure can be simplified if these features are not required (as in the maximum flow and MSF applications). Even without simplification, the slowdown (relative to ST-trees) is arguably a reasonable price to pay for generality and ease of use. None of the logarithmic-time data structures studied is particularly easy to implement; the ability to adapt an existing implementation to different applications is a valuable asset.

When queries vastly outnumber *links* and *cuts*, worst-case and randomized data structures are competitive with self-adjusting ones. Even TOP-W, which changes the tree during queries, can be faster than self-adjusting top trees. But TOP-Q and RC prove that not making changes at all is the best strategy. Similar results were obtained by Acar et al. [2005], who observed that RC-trees are significantly slower than ST-trees for structural updates, but faster when queries are numerous. Cattaneo et al. [2002] use a randomized implementation of ET-trees in conjunction with (self-adjusting) ST-trees to speed up connectivity queries within a dynamic minimum spanning tree algorithm. Although ST-trees can easily support such queries, the authors found them too slow.

This situation is not ideal. A clear direction for future research is to create general data structures that have a more favorable trade-off between queries and structural updates. A more efficient implementation of worst-case top trees would be an important step in this direction. In addition, testing the data structures on more elaborate applications would be valuable.

Acknowledgements. We thank Umut Acar, Guy Blelloch, Adam Buchsbaum, Jakob Holm, Philip Klein, Mikkel Thorup, and Jorge Vittes for helpful discussions.

REFERENCES

ACAR, U. A., BLELLOCH, G. E., HARPER, R., VITTES, J. L., AND WOO, S. L. M. 2004. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 524–533.

ACAR, U. A., BLELLOCH, G. E., AND VITTES, J. L. 2005. An experimental analysis of change propagation in dynamic trees. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 41–54.

ADDARIO-BERRY, L., BROUTIN, N., AND REED, B. 2006. The diameter of the minimum spanning tree of a complete graph. In *Fourth Colloquium on Mathematics and Computer Science*. Discrete Mathematics and Theoretical Computer Science (DMTCS), 237–248.

AHUJA, R., MAGNANTI, T., AND ORLIN, J. 1993. *Network Flows: Theory, algorithms, and applications*. Prentice-Hall.

ALSTRUP, S., HOLM, J., DE LICHTENBERG, K., AND THORUP, M. 1997. Minimizing diameters of dynamic trees. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*. Lecture Notes in Computer Science, vol. 1256. Springer, 270–280.

ALSTRUP, S., HOLM, J., AND THORUP, M. 1999. On the power and speed of top trees. Unpublished manuscript.

ALSTRUP, S., HOLM, J., THORUP, M., AND DE LICHTENBERG, K. 2005. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms 1,* 2, 243–264.

ANDERSON, R. 1993. The `washington` graph generator. In *DIMACS Series in Discrete Mathematics and Computer Science*, D. S. Johnson and C. C. McGeoch, Eds. AMS, 580–581.

BELLMAN, R. 1958. On a routing problem. *Quarterly Mathematics 16*, 87–90.

CATTANEO, G., FARUOLO, P., FERRARO-PETRILLO, U., AND ITALIANO, G. F. 2002. Maintaining dynamic minimum spanning trees: An experimental study. In *Proceedings of the 4th Workshop* *on Algorithm Engineering and Experiments (ALENEX)*. Lecture Notes in Computer Science, vol. 2049. Springer, 111–125.

Cherkassky, B. V., Georgiadis, L., Goldberg, A. V., Tarjan, R. E., and Werneck, R. F. 2008. Shortest path feasibility algorithms: An experimental evaluation. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 118–132.

Cherkassky, B. V. and Goldberg, A. V. 1999. Negative-cycle detection algorithms. *Mathematica Programming 85*, 277–311.

Dinic, E. A. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady 11*, 1277–1280.

Edmonds, J. and Karp, R. M. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM 19*, 248–264.

Ford, Jr., L. 1956. Network flow theory. Tech. Rep. P-932, The Rand Corporation.

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

Frederickson, G. N. 1997a. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing 26,* 2, 484–538.

Frederickson, G. N. 1997b. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms 24,* 1, 37–65.

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. *Mathematical Programming 50*, 277–290.

Henzinger, M. R. and King, V. 1997. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*. 519–527.

Kaplan, H., Molad, E., and Tarjan, R. E. 2003. Dynamic rectangular intersection with priorities. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*. 639–648.

Klein, P. N. 2005. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 146–155.

Langerman, S. 2000. On the shooter location problem: Maintaining dynamic circular-arc graphs. In *Proceedings of the 12th Canadian Conference on Computational Geometry (CCCG)*. 29–35.

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

Moore, E. F. 1959. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*. Harvard University Press, 285–292.

Radzik, T. 1998. Implementation of dynamic trees with in-subtree operations. *ACM Journal of Experimental Algorithmics 3,* 9.

Ribeiro, C. C. and Toso, R. F. 2007. Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes on edge weights. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA)*. Lecture Notes in Computer Science. Springer, 393–405.

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

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

Tarjan, R. E. 1983. *Data Structures and Network Algorithms*. SIAM Press.

Tarjan, R. E. 1997. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Mathematical Programming 78*, 169–177.

Tarjan, R. E. and Werneck, R. F. 2005. Self-adjusting top trees. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 813–822.

Werneck, R. F. 2006. Design and analysis of data structures for dynamic trees. Ph.D. thesis, Princeton University.

Zaroliagis, C. D. 2002. Implementations and experimental studies of dynamic graph algorithms. In *Experimental Algorithms: From Algorithm Design to Robust and Efficient Software*,

R. Fleischer, B. Moret, and E. M. Schmidt, Eds. Lecture Notes in Computer Science. Springer, 229–278.