Dynamic Trees in Practice*

Robert E. Tarjan[1,2] and Renato F. Werneck[3]

[1] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA
ret@cs.princeton.edu
[2] Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304, USA
[3] Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA
renatow@microsoft.com

Abstract. 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.

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 [14,22,25], dynamic graphs [9,11,12,15,20,28], and other combinatorial problems [16,17,18].

Several well-known data structures can (at least partially) solve the dynamic tree problem in $O(\log n)$ time per operation: ST-trees [22,23], topology trees [11,12,13], ET-trees [15,25], top trees [4,6,26,27], and RC-trees [1,2]. They all map an arbitrary tree into a balanced one, 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

* 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.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 80–93, 2007.

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 [27, 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

ET-trees. The simplest (and most limited) dynamic-tree data structures are ET-trees [15,25]. They represent an *Euler tour* of an arbitrary unrooted tree, i.e., a tour that traverses each edge of the tree twice, once in each direction. It can be thought of as a circular list, with each node representing 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 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 [25] and uses splay trees [23], a self-adjusting form of binary search trees whose performance guarantees are amortized.

As described by Tarjan [25], ET-trees associate values with vertices. Besides allowing queries and updates to individual values, the ET-tree interface also 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 node stores its value in *difference form*, i.e., relative to 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.

ST-trees. The best-known dynamic tree data structures supporting path operations are Sleator and Tarjan's ST-trees (also known as *link-cut trees*). They predate ET-trees [22,23], 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 reverses 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 [23], 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 its minimum-cost vertex.

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 vertices, but LIN-E interprets such a value as the cost of the arc to the parent (the values must 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 where the original nodes appear in symmetric order. These binary trees are then "glued" together to create a single *virtual tree*: the root of a binary tree representing a path P becomes a *middle child* (in the virtual tree) of the parent (in the original forest) of the topmost vertex of P. For an efficient implementation of *addcost* and *evert*, values are stored in difference form.

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. When paths are represented as splay trees, *expose* and the other dynamic tree operations run in $O(\log n)$ amortized time [23]. A worst-case logarithmic bound is achievable with globally biased search trees [22], but this solution is too complicated to be practical.

We call the splay-based implementation used in our experiments ST-V. Although it has costs on vertices, it can also represent costs on edges as long as *evert* is never called. 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.

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 [14,16,17,18,20].

Tree contraction. A third approach 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. [4,6], but we borrow the notation used by Tarjan and Werneck [26]. 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. 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. 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 of the top tree represents the entire underlying tree. Whenever there is a *link* or *cut*, the data structure merely updates the contractions affected.

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*). The interface stipulates that these calls will be made when all clusters involved are *roots* of (maybe partial) top trees—none of the input clusters will have a parent. This makes the implementation of the call-back functions easier, but it may hurt performance: because *joins* must be called bottom-up and *splits* top-down, the tree cannot be updated in a single pass.

To perform a query, the user calls the *expose*(v, w) operation, which returns a root cluster having v and w as endpoints (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 user should define the internal functions so that the cluster returned by *expose* automatically contains the answer to the user's query; there is no need to actually search the tree. 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 [6].

The first contraction-based data structures were in fact not top trees but Frederickson's *topology trees* [11,12,13]. They interpret clusters as *vertices* instead of edges. This leads to a simpler contraction algorithm, but it requires all vertices in the 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. [1,2] 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 [2], we set the bound to 8 in our experiments). We call the implementation (by Acar et al. [2]) of RC-trees used in our experiments RC. The

name "RC-trees" is a reference to *rake* and *compress*, which were introduced by Miller and Reif [19] in the context of parallel algorithms.

RC-trees have a generic interface to separate value updates from the actual contraction. Unlike top trees, queries require a traversal of the tree. This 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.

Alstrup et al. [6] 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 the extra layer of topology trees. (Details, still unpublished, can be found in [27].) 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.

After a *link*, *cut*, or *expose*, the contraction can be updated in $O(\log n)$ worst-case time [27]. 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 $2n$ nodes, when dummy nodes 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 [26] 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, and do not support circular adjacency lists directly.

3 Experimental Results

Experimental Setup. This section presents an experimental comparison of the data structures discussed above: 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 explicitly (LIN-V/LIN-E). We tested these 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; the reader is referred to [28] for a survey on this topic.

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, 16 KB of level-one data cache, 2 MB of level-two cache, and 2 GB of RAM. With the exception of RC-trees, all data structures were implemented by the authors and are available upon request. RC-trees, available at *http://www.cs.cmu.edu/~jvittes/rc-trees/*, were implemented by Acar, Blelloch, and Vittes [2]. We only tested RC-trees 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. 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 different randomized inputs.

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 RC-trees, which allocates memory as needed in large blocks, and frees it all at once). All executions fit in RAM, unless specifically noted.

Maximum flows. One of the original motivations for dynamic tree data structures was the *maximum flow problem* (see, e.g., [3]). 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 the *shortest augmenting path* algorithm for this problem, due to Edmonds and Karp [10]. In each iteration, it finds a path with positive residual capacity that has the fewest arcs and sends as much flow as possible along it; the algorithm stops when no such *augmenting path* exists. Intuitively, the algorithm grows a path from s containing only admissible arcs (potential candidates to belong to the shortest path) until it reaches t, backtracking whenever it reaches a vertex with no outgoing admissible arcs. A direct implementation takes $O(n^2m)$ worst-case time, which can be reduced to $O(mn \log n)$ if we use dynamic trees to maintain a forest of admissible arcs. The modified algorithm always processes

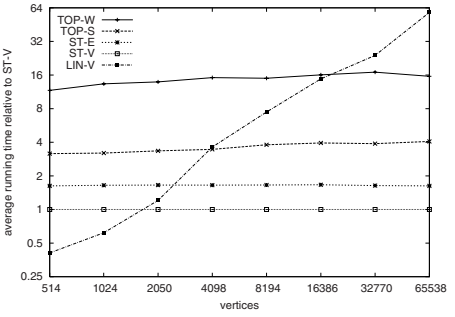

Fig. 1. Maximum flows on layer graphs with four rows and varying numbers of columns

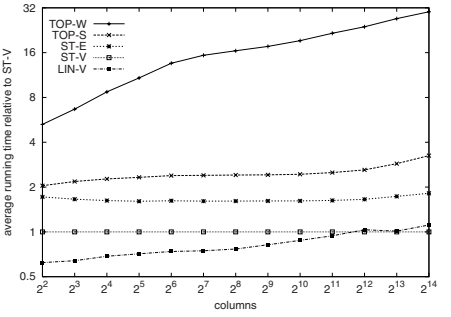

Fig. 2. Maximum flows on meshes with 65 538 vertices and varying numbers of columns (and rows)

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* those that drop to zero. See [3] for details.

The operations supported by the ST-tree interface (such as *addcost*, *findmin*, *findroot*) are, by construction, exactly those needed to implement this algorithm. With top trees, we make each cluster $C = (v, w)$ represent both a *rooted* tree and a *directed path* between its endpoints. The cluster stores the root vertex 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 $v \cdots w$ (it supports the equivalent of ST-tree's *addcost*). We also tried implementing the full ST-tree interface on top of top trees, as suggested in [5], but it was up to twice as slow as the direct method.

Our first experiment is on *random layer graphs* [7], 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 1 reports average running times normalized with respect to ST-V.

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[1] 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

[1] A similar description was mistakenly given to the "directed meshes" tested in [27]; those graphs, obtained with a different generator [7], were actually acyclic.

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. See Figure 2.

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 edges, ST-V can be used because *evert* is never called; ST-E, included in the experiments for comparison, is slightly slower. 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 with 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 algorithms still have comparable running times when augmenting paths have more than 16 384 vertices. With so many columns, the average number of *links* between augmentations is almost 9000 on directed meshes, but only four on layer graphs (which are acyclic). This explains the difference in performance.

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.

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 a dynamic tree to maintain the MSF, each new edge can be processed in $O(\log n)$ time. Suppose a new edge $e = (v, w)$ is added to the graph. If v and w belong to different components, we simply add (*link*) e to the MSF. 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 the "red rule" described by Tarjan in [24]: if an edge is the most expensive in *some* cycle in the graph, then it does not belong to the minimum spanning forest.

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, 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 3 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). Being much simpler, it is roughly 20 times faster than ST-E.

Our second experiment also involves random graphs, but we now fix $n = 2^{16}$ and vary the average vertex degree from 4 to 512 (i.e., we vary m from 2^{17} to 2^{25}). As the density increases, relatively fewer *links* and *cuts* will be performed:

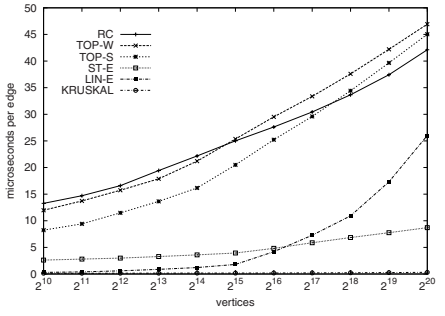

Fig. 3. Online minimum spanning forest on random graphs with average degree 16

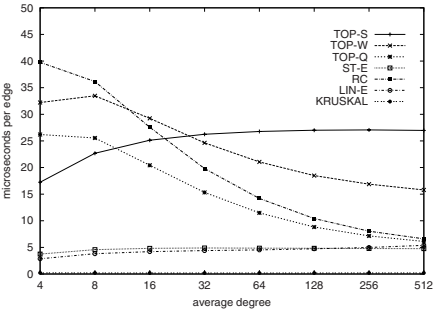

Fig. 4. Online minimum spanning forest on random graphs with 65 536 vertices

when the average degree is 512, only roughly 2.5% of the input edges are actually inserted into the MSF. As a result, as Figure 4 shows, the average time to process an edge decreases with the density, and 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.

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. [6], however, we implemented *expose* by marking some existing top tree nodes as "invalid" and building a temporary top tree with the $O(\log n)$ root clusters that remain. This eliminates expensive updates of Euler tours during queries. Fast queries help explain why TOP-W is more competitive with TOP-S for the online MSF application as compared to maximum flows. Being self-adjusting, TOP-S also benefits from the fact that consecutive dynamic tree operations are correlated in the maximum flow application.

RC-trees do not modify the tree (even temporarily) during queries: instead, they traverse the tree in a bottom-up fashion, aggregating information contained in internal nodes. 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-W is not an implementation of top trees, since it violates its well-defined interface. As Figure 4 shows, however, 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 three steps: first, generate a random spanning tree on c vertices; second, progressively transform the original edges into paths (until there are n vertices in total); finally, add $m-n+1$ random edges to the graph. 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 5 shows the performance of various algorithms with $n = 2^{13}$, $m = 2^{18}$,

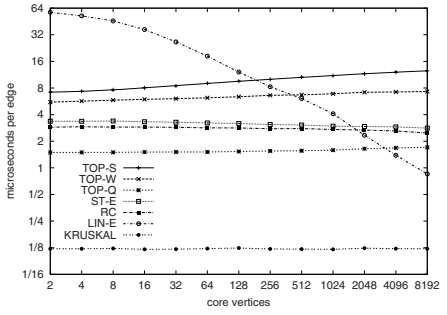

Fig. 5. Online minimum spanning forests on augmented random graphs

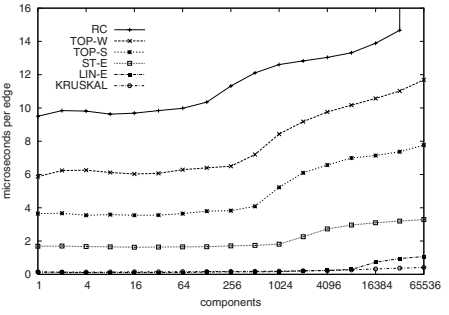

Fig. 6. Cache effects on forests with 32-vertex random components

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 (in Figure 3 as well).

Finally, we investigate the effect of caching on 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. The total number of edges added is $128c$, so that each component has 128 edges on average. Figure 6 shows that, due to cache effects, the average time to process each edge actually varies as a function of c: all methods 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 16 bytes). This is significantly less than ST-E (57 bytes), TOP-S (216), TOP-W (399), and RC (roughly one kilobyte). If fact, RC-trees even ran out of RAM for the largest graph tested (this is the only case reported in the paper in which this happened—all other tests ran entirely in memory); the excessive memory usage of this particular implementation helps explain why it is consistently slower than worst-case top trees, despite being presumably much simpler.

Even though Figure 6 shows an extreme case, cache effects should not be disregarded. Take, for instance, the layer graphs used in the maximum flow application. The graph generator assigns similar identifiers to adjacent vertices, which means that path traversals have strong locality. Randomly permuting vertex identifiers would slow down all algorithms, but LIN-V (which uses only 8 bytes per node) would be affected the most: on layer graphs with 65 538 vertices, running times would increase by 150% for LIN-V, 40% for ST-V (which uses 24 bytes per node), and only 11% for TOP-W.

Single-source shortest paths. The applications considered so far require dynamic trees to aggregate information over paths. We now test an application that

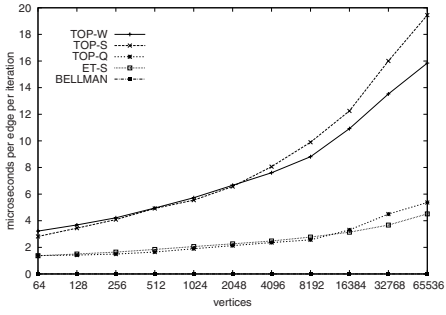

Fig. 7. Single-source shortest paths on graphs with Hamiltonian circuits

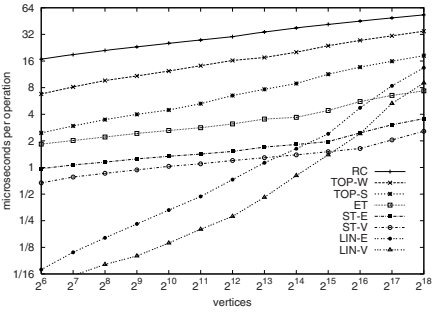

Fig. 8. Average time per operation on a randomized sequence of *links* and *cuts*

aggregates over trees: a label-correcting 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). Bellman's algorithm [8] maintains a *distance label* $d(v)$ for every vertex v, representing an upper bound on its distance from s (initially zero for s and infinite otherwise). While there exists an arc $(v, w) \in A$ such that $d(v) + \ell(v, w) < d(w)$, the algorithm *relaxes* it by setting $d(w) \leftarrow d(v) + \ell(v, w)$. After $n - 1$ passes through the list of arcs (in $O(mn)$ time) either all distance labels will be exact or a negative cycle will be found.

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. Bellman's algorithm will eventually do it, 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 7 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 Bellman's algorithm (denoted by BELLMAN), 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 do 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 Bellman's algorithm. This is obviously a poor application of dynamic trees, since the straightforward algorithm is trivial and has better running time; the only purpose of the experiment is to compare the performance of the data structures among themselves.

Random structural operations. 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. 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 8 shows the average time to execute each operation of the precomputed sequence. The results are in line with those observed for previous experiments. ST-trees are the fastest logarithmic data structure, followed by ET-trees, self-adjusting top trees, worst-case top trees, and RC-trees.

Additional observations. An efficient implementation of the *evert* operation in 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 where 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 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, ST-V becomes almost 20% faster on a random sequence of *links* and *cuts*. The main reason is better locality: value updates require looking outside the splaying path.

The performance of the data structures also depends on how much data is stored in each node. If we stored 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. [5] observed the same for randomized sequences of operations. Recently, Ribeiro and Toso [21] 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 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 [26], the data structure can be simplified if these features are not required (as in the maximum flow and MSF applications). We plan to implement restricted versions in the future, but even the current slowdown (relative to ST-trees) is arguably a reasonable price to pay for generality and ease of use. None of the logarithmic 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. [2], who observed that RC-trees are significantly slower than ST-trees for structural operations, but faster when queries are numerous. In [9], a randomized implementation of ET-trees is used 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 operations. 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

1. Acar, U.A., Blelloch, G.E., Harper, R., Vittes, J.L., Woo, S.L.M.: Dynamizing static algorithms, with applications to dynamic trees and history independence. In: Proc. 15th SODA, pp. 524–533 (2004)
2. Acar, U.A., Blelloch, G.E., Vittes, J.L.: An experimental analysis of change propagation in dynamic trees. In: Proc. 7th ALENEX, pp. 41–54 (2005)
3. Ahuja, R., Magnanti, T., Orlin, J.: Network Flows: Theory, algorithms, and applications. Prentice-Hall, Englewood Cliffs (1993)
4. Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M.: Minimizing diameters of dynamic trees. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 270–280. Springer, Heidelberg (1997)
5. Alstrup, S., Holm, J., Thorup, M.: On the power and speed of top trees. Unpublished manuscript (1999)

6. Alstrup, S., Holm, J., Thorup, M., de Lichtenberg, K.: Maintaining information in fully dynamic trees with top trees. ACM TALG 1(2), 243–264 (2005)
7. Anderson, R.: The washington graph generator. In: Johnson, D.S., McGeoch, C.C.: (eds.), DIMACS Series in Discrete Mathematics and Computer Science, pp. 580–581. AMS (1993)
8. Bellman, R.: On a routing problem. Quarterly Mathematics 16, 87–90 (1958)
9. Cattaneo, G., Faruolo, P., Ferraro-Petrillo, U., Italiano, G.F.: Maintaining dynamic minimum spanning trees: An experimental study. In: Paliouras, G., Karkaletsis, V., Spyropoulos, C.D. (eds.) Machine Learning and Its Applications. LNCS (LNAI), vol. 2049, pp. 111–125. Springer, Heidelberg (2002)
10. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. JACM 19, 248–264 (1972)
11. Frederickson, G.N.: Data structures for on-line update of minimum spanning trees, with applications. SIAM J. Comp. 14(4), 781–798 (1985)
12. Frederickson, G.N.: Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. SIAM J. Comp. 26(2), 484–538 (1997)
13. Frederickson, G.N.: A data structure for dynamically maintaining rooted trees. J. Alg. 24(1), 37–65 (1997)
14. Goldberg, A.V., Grigoriadis, M.D., Tarjan, R.E.: Use of dynamic trees in a network simplex algorithm for the maximum flow problem. Mathematical Programming 50, 277–290 (1991)
15. Henzinger, M.R., King, V.: Randomized fully dynamic graph algorithms with polylogarihmic time per operation. In: Proc. 29th STOC, pp. 519–527 (1997)
16. Kaplan, H., Molad, E., Tarjan, R.E.: Dynamic rectangular intersection with priorities. In: Proc. 35th STOC, pp. 639–648 (2003)
17. Klein, P.N.: Multiple-source shortest paths in planar graphs. In: Proc. 16th SODA, pp. 146–155 (2005)
18. Langerman, S.: On the shooter location problem: Maintaining dynamic circular-arc graphs. In: Proc. 12th CCCG, pp. 29–35 (2000)
19. Miller, G.L., Reif, J.H.: Parallel tree contraction and its applications. In: Proc. 26th FOCS, pp. 478–489 (1985)
20. Radzik, T.: Implementation of dynamic trees with in-subtree operations. ACM JEA, vol. 3(9) (1998)
21. Ribeiro, C.C., Toso, R.F.: Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes on edge weights. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 393–405. Springer, Heidelberg (2007)
22. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. JCSS 26(3), 362–391 (1983)
23. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. JACM 32(3), 652–686 (1985)
24. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM Press (1983)
25. Tarjan, R.E.: Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. Mathematical Programming 78, 169–177 (1997)
26. Tarjan, R.E., Werneck, R.F.: Self-adjusting top trees. In: Proc. 16th SODA, pp. 813–822 (2005)
27. Werneck, R.F.: Design and Analysis of Data Structures for Dynamic Trees. PhD thesis, Princeton University (2006)
28. Zaroliagis, C.D.: Implementations and experimental studies of dynamic graph algorithms. In: Fleischer, R., Moret, B., Schmidt, E.M. (eds.) Experimental Algorithms: From Algorithm Design to Robust and Efficient Software. LNCS, pp. 229–278. Springer, Heidelberg (2002)