Experimental Analysis of Algorithms for Updating Minimum Spanning Trees on Graphs Subject to Changes on Edge Weights

Celso C. Ribeiro[⋆] and Rodrigo F. Toso[⋆⋆]

Institute of Computing, Universidade Federal Fluminense
Rua Passo da Pátria 156, Niterói, RJ 24210-240, Brazil
celso@inf.puc-rio.br, rtoso@rutcor.rutgers.edu

Abstract. We consider the problem of maintaining a minimum spanning tree of a dynamically changing graph, subject to changes on edge weights. We propose an on-line fully-dynamic algorithm that runs in time $O(|E|)$ when the easy-to-implement DRD-trees data structure for dynamic trees is used. Numerical experiments illustrate the efficiency of the approach.

Keywords: Minimum spanning trees, dynamic graph algorithms, experimental analysis, algorithms, data structures, DRD-trees.

1 Introduction

The dynamic minimum spanning tree problem is that of maintaining a minimum spanning tree (MST) of a dynamically changing graph $G = (V, E)$, where changes can be vertex insertions and deletions, edge insertions and deletions, or edge weight modifications. The problem is said to be *fully dynamic* if insertion and deletion operations are allowed (or if the edge weights can increase and decrease). The problem is said to be *partially dynamic* if only one kind of operation is allowed (either deletions or insertions, either weight increases or weight decreases). The problem is said to be *on-line* if the dynamic changes must be processed in real time (i.e., there is no preprocessing and the updates are performed one at a time) [1].

There are several variants of the dynamic minimum spanning tree problem. Spira and Pan [16] proposed algorithms for vertex insertion and vertex deletion variants. Zaroliagis [18] surveyed experimental studies on dynamic graph problems subject to edge insertions and deletions. See also [7,8,9,10,11] for algorithms and [1,5] for experimental studies involving edge insertions and edge deletions.

In this work, we make a step toward the experimental evaluation of algorithms to update a minimum spanning tree after edge weight changes. Such algorithms are needful in the implementation of local search heuristics for solving broadcast

[⋆] Supported by CNPq and FAPERJ research grants.
[⋆⋆] Supported by a CNPq scholarship.

and design problems in communication networks, similarly to the algorithms for dynamic shortest path problems studied by Buriol et al. [2,3,4] in the context of the weight setting problem in OSPF/IS-IS routing.

In the next section, we describe and evaluate a new and easy-to-implement supporting data structure for dynamic trees representation, called DRD-trees. A new fully-dynamic algorithm for updating a minimum spanning tree after edge weight changes is proposed in Section 3. Complexity issues are also considered in this section. An experimental evaluation of several algorithms is carried out in Section 4 on a comprehensive set of test instances, showing that our approach outperforms the fastest algorithms for real-size instances. Concluding remarks are drawn in the last section.

2 Data Structures: Doubly-Linked Reversed Dynamic Trees

We are given a dynamic graph $G = (V, E)$ with vertex set V, edge set E, and non-negative weights $w(i, j)$ associated with each edge $(i, j) \in E$. Let $T = (V, E')$ be a dynamic minimum weight spanning tree of G, i.e., a minimum spanning tree subject to structural changes caused by modifications in the edge weights.

The dynamic trees problem [15] consists in maintaining a forest of disjoint trees that change over time through edge insertions and deletions. For example, one may want to link two trees by adding an edge or to cut a tree by removing an edge. We denote by `Link`(i, j, w) (resp. `Cut`(i, j)) the operation of inserting (resp. deleting) edge (i, j) into (resp. from) tree $T = (V, E')$. A simple way to represent a forest of disjoint trees is through a set of rooted, directed trees. To manipulate these trees, the following operations are also made available:

- `Root`(i): returns the root of the tree containing vertex i;
- `Evert`(i): makes vertex i the root of its tree; and
- `Find_Max`(i): returns the max-weight edge in the path from i to `Root`(i).

Operations `Link`(i, j, w) and `Cut`(i, j) can be implemented in constant time using *rooted reversed dynamic trees* (RD-trees): each vertex $i \in V$ stores its parent and the weight of the edge between them. Operations `Evert`(i), `Root`(i) and `Find_Max`(i) run in linear time, since they depend on the length of the path from vertex i to `Root`(i). However, a link operation may require the execution of an evert operation in the case of undirected trees, therefore resulting in linear running time. Sleator and Tarjan [15] (resp. Henzinger and King [10] and Werneck and Tarjan [17]) proposed the ST-trees (resp. ET-trees and self-adjusting top trees), designed to support all the above operations in logarithmic time (in fact, ET-trees do not support `Find_Max`(i)).

We propose an extension of the RD-trees by building doubly-linked trees (DRD-trees), instead of simply reversed trees. This can be accomplished by an additional list associated with each vertex $v \in V$, storing each of its children.

The motivation for this extension comes from the need to detect if an arbitrary edge $(x, y) \in E$ reconnects the two disjoint subtrees T_i and T_j resulting from the removal of an edge $(i, j) \in E'$, where T_i (resp. T_j) is the resulting subtree containing vertex i (resp. j). This *connectivity query* can be answered by checking if x and y belong to different subtrees (i.e., if the roots of their subtrees are different). However, storing all vertices adjacent to j and assuming (without loss of generality) that vertex i is the parent of j, one may apply a depth-first search starting from j and label all reachable vertices (i.e., those in T_j) with one, zero otherwise. These labels can then be used to answer the above connectivity query in amortized constant time when the number of queries is $O(n)$, in contrast with the root-based data structures, which depend on the implementation of `Root(`i`)` and take at least logarithmic time.

3 Fully-Dynamic Algorithm

We present a new fully dynamic algorithm to update the minimum spanning tree of a graph subject to edge weight changes. It makes use of DRD-trees, which combine easy implementation with efficiency. Two cases are considered separately: edge weight decreases are considered in Section 3.1, while edge weight increases are handled in Section 3.2.

Externally to both algorithms, we maintain an edge list A sorted from left to right by the non-decreasing order of edge weights. Whenever an edge weight is increased or decreased, this list is updated to reflect the new ordering. Let $A(k)$ be the k-th edge in the ordered list A, with $k = 1, \ldots, |E|$, and $A(i, j)$ be the position of edge $(i, j) \in E$ in the ordered list A. List A is stored as a skip list [14] for improved efficiency. The general framework used to update the minimum spanning tree, maintaining list A correctly ordered, is shown in Algorithm 1..

Algorithm 1. Updates a minimum spanning tree subject to edge weight changes

```
Input: Graph $G = (V, E)$, weights $w$.
 1: Build a list $A$ with all $(i, j) \in E$;
 2: Sort list $A$ by non-decreasing order of weights;
 3: Use list $A$ to compute the MST $T = (V, E')$;
 4: while there is an update to be processed do
 5:    Let $(i, j) \in E$ be the edge whose weight will change to $w^{new}$;
 6:    $s \leftarrow A(i, j)$;
 7:    Save the old edge weight: $w^{old} \leftarrow w(i, j)$;
 8:    Set the new edge weight: $w(i, j) \leftarrow w^{new}$;
 9:    Reorder list $A$ by non-decreasing edge weights;
10:    $f \leftarrow A(i, j)$;
11:    if $w^{new} < w^{old}$ and $(i, j) \notin E'$ then
12:       Apply Algorithm 2. with parameters $T$, $(i, j)$, and $w$;
13:    else if $w^{new} > w^{old}$ and $(i, j) \in E'$ then
14:       Apply Algorithm 3. with parameters $T$, $(i, j)$, $s$, $f$, and $w$;
15:    end if
16: end while
```

The ordered list A is initialized in lines 1 and 2. A minimal spanning tree $T = (V, E')$ is computed in line 3 using Kruskal's algorithm [12]. The loop in lines 4 to 16 is performed until all updates have been processed. The edge to be updated and its new weight are read in line 5. The position s of edge (i, j) in the current list A is saved in line 6, before the list is reordered. The old weight w^{old} of edge (i, j) is saved in line 7, while the new weight w^{new} is set in line 8. The list A is reordered in line 9 after the change of the weight $w(i, j)$ of edge (i, j). To ensure the correctness of the decremental updates, in case the new weight w^{new} of (i, j) is equal to the weight of other edges, then edge (i, j) should be placed in the last position among all edges with the same weight. The position f of edge (i, j) in the reordered list A is saved in line 10. If the comparison in line 11 (resp. in line 13) determines that the new weight of edge (i, j) is smaller (resp. larger) than the old one and (i, j) is a non-tree (resp. tree) edge, then Algorithm 2. (resp. Algorithm 3.) is applied in line 12 (resp. line 14) to update the current minimum spanning tree, since decreasing the weight of a tree edge (resp. increasing the weight of a non-tree edge) does not change the latter. We notice that updates in the weight function are reflected both in list A and in the data structure used to store the minimum spanning tree T.

3.1 Weight Decreases

Whenever the weight of a non-tree edge (i, j) is decreased, one has to find the maximum weight edge (x, y) along the path from i to j in T and to remove it if $w(x, y) > w(i, j)$ [16]. This can be accomplished by using any dynamic tree data structure to store the MST.

The procedure to update the minimum spanning tree is described in Algorithm 2.. Vertex i is made the new root of the tree in lines 1 to 3. The maximum weight edge (x, y) in the path from j to i is computed in line 4. If the weight of edge (x, y) is larger than that of edge (i, j) (comparison in line 5), then the former is removed from the tree by the `Cut`(x, y) operation in line 6 and the new edge (i, j) is inserted by the `Link`$(i, j, w(i, j))$ operation in line 7. The updated MST is returned in line 9.

The efficiency of the above computations depend on the underlying structure used to maintain the MST and to implement the path operations. If RD-trees or DRD-trees are used, then Algorithm 2. runs in time $O(|V|)$. It runs in time $O(\log |V|)$ if a more complex implementation (such as ST-trees) is used.

3.2 Weight Increases

We now face the hardest part of the problem. If the weight of a tree edge (i, j) is increased, the latter may have to be removed from the current minimum spanning tree. In this case, one has to find the minimum weight edge connecting the two resulting disjoint subtrees (namely, $T_i = (V_i, E'_i)$ and $T_j = (V_j, E'_j)$) to be inserted in the new minimum spanning tree. There are up to $O(n^2)$ such candidate edges [16].

The procedure to update the minimum spanning tree is described in Algorithm 3.. Vertex j is assumed to be a child of vertex i and a depth-first search

Algorithm 2. Non-tree edge weight decreases

```
Input: MST $T = (V, E')$, non-tree edge $(i, j)$ subject to a weight decrease, weights $w$.
 1: if $i \neq$ Root($i$) then
 2:    Evert($i$);
 3: end if
 4: $(x, y) \leftarrow$ Find_Max($j$);
 5: if $w(x, y) > w(i, j)$ then
 6:    Cut($x, y$);
 7:    Link($i, j, w(i, j)$);
 8: end if
 9: return updated minimum spanning tree $T$;
```

is applied to the current MST from vertex j in line 1. Vertices in T_j are those reachable from j by a depth-first search in T. The position k of the first candidate edge to replace (i, j) is set in line 2. The loop in lines 3 to 11 scans all edges between positions s and f of list A. The next edge (x, y) to be investigated is set in line 4 as that in position k of list A. If vertices x and y are in different subtrees (comparison in line 5), then edge (i, j) is eliminated from the current tree in line 6 and the two subtrees T_i and T_j are linked by edge (x, y) in line 7. The updated minimum spanning tree is returned in line 8. Otherwise, the current position in list A is incremented by one in line 10 and a new edge is examined. If no improving edge can be found to replace edge (i, j), then the unchanged current minimum spanning tree is returned in line 12.

Algorithm 3. Tree edge weight increases

```
Input: MST $T = (V, E')$, tree edge $(i, j)$ subject to a weight increase, positions $s$ and
    $f$, weights $w$.
 1: Assume $j$ as child of $i$ and perform DFS($j$);
 2: $k \leftarrow s$;
 3: while $k < f$ do
 4:    $(x, y) \leftarrow A(k)$;
 5:    if Label($x$) $\neq$ Label($y$) then
 6:       Cut($i, j$);
 7:       Link($x, y, w(x, y)$);
 8:       return updated minimum spanning tree $T$;
 9:    end if
10:    $k \leftarrow k + 1$;
11: end while
12: return unchanged minimum spanning tree $T$;
```

Algorithm 3. can be adapted to make use of data structures based on the tree roots, such as ST-trees or RD-trees.

The edge list A, which is externally reordered, can be updated in $O(\log |E|)$ expected time if a skip list is used. The worst case occurs when edge (i, j) is shifted from the first to the last position of A. In this case, $|E|$ edges may have to be considered for replacement. If DRD-trees (resp. ST-trees or RD-trees) are used, each `Label(`v`)` (resp. `Root(`v`)`) operation takes time $O(1)$ (resp. $O(\log |V|)$ or $O(|V|)$). Therefore, the overall complexity of Algorithm 3. is $O(|E|)$ if DRD-trees are used, $O(|E| \log |V|)$ if ST-trees are used, and $O(|E||V|)$ if RD-trees are used. The following theorem establishes the correctness of the approach:

Theorem 1. *Algorithms 1., 2., and 3. correctly update a minimum spanning tree.*

Proof. The structure of list A is such that if an edge (i, j) belongs to the current minimum spanning tree, then it is the first from left to right in the ordered list A connecting T_i and T_j. This is initially ensured by Kruskal's algorithm. It also holds for Algorithm 3., since by construction the latter selects the left-most minimum weight edge connecting T_i and T_j between positions s and f of list A.

We now show that letting the weight decreased edge be the right-most edge between those with the same weight preserves the structure of list A, ensuring the correctness of Algorithm 2.. Assume edge (x, y) is the candidate edge to be replaced by (i, j) in the updated MST. If $w(x, y) > w(i, j)$, then edge (i, j) replaces (x, y) in the tree and becomes the left-most edge in list A connecting T_i and T_j, since otherwise (x, y) would not be in the current MST. In case $w(x, y) = w(i, j)$, then edge (i, j) does not replace (x, y). Due to the condition imposed by each reordering procedure, edge (i, j) is to the right of (x, y). Thus, (x, y) is the left-most in list A connecting T_i and T_j, and, therefore, the structure of list A associated with the updated minimum spanning tree is preserved. □

4 Computational Experiments

The computational experiments are presented in three sections. We first present experiments concerning the ability of DRD-trees to answer connectivity queries when compared to existing data structures. The next sections contain the experimental analysis of algorithms for updating minimum spanning trees of dynamic graphs, with numerical results for synthetic and realistic large graph instances.

The experiments were performed on a Pentium 4 processor with a 2.4 GHz clock and 768 Mbytes of RAM under GNU/Linux 2.6.16. The algorithms were coded in C++ and compiled with the GNU `g++` compiler version 4.1, using the optimization flag `-O2`. Although some codes were obtained from different authors, all algorithms and data structures were revised and optimized for this study. All processing times are average results over 100 instances of each size (ten different trees or graphs subject to ten different update sequences). Both random and structured sequences of updates are considered.

4.1 Dynamic Trees

We evaluate the behavior of DRD-trees regarding its efficiency to answer connectivity queries. Figure 1 depicts how fast the data structures can answer 100,000 connectivity queries in random trees containing from 2,000 to 400,000 vertices.

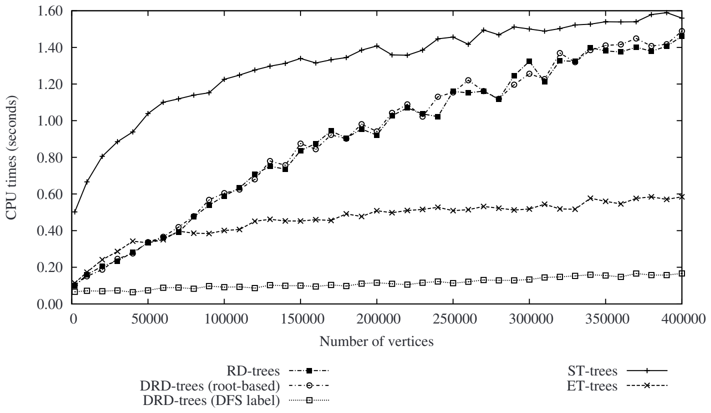

Fig. 1. CPU times for processing 100,000 connectivity queries in random trees

DRD-trees using the depth-first search labeling technique run in amortized constant time if the labels are used to process the connectivity queries, while RD-trees and DRD-trees run in linear time and ET-trees and ST-trees in logarithmic time (all of them using the root-based approach). DRD-trees are faster than the other data structures considered in this study.

4.2 Algorithms

This section presents a computational study addressing efficient algorithms for updating minimum spanning trees on dynamic graphs.

Cattaneo et al. [5] have shown that, except for very particular instances, a simple $O(m \log n)$-time algorithm has the best performance in practice and runs substantially faster than the poly-logarithmic algorithm HDT of Holm et al. [11]: the latter was faster than the first for only one out of five different classes of instances, namely those in k-clique graphs, in which small cliques are connected by some inter-clique edges and all updates involve only the inter-clique edges (see Section 4.2(a) below). Despite its theoretical running time of $O(\log^4 n)$ amortized, the complex chain of data structures supporting algorithm HDT does not seem to lead to fast implementations and may require too much memory. However, since Cattaneo et al. [5] have not applied it to grids and road networks, the performance of HDT on the instances with large memory requirements considered in Section 4.2(b) remains an open question.

We present the implemented algorithms together with their complexities in Table 1. The names of algorithms from [5] start by `C`, while those of variants of our approach start by `RT`. We indicate inside brackets the data structures used to maintain the minimum spanning tree. The following algorithms were compared:

`C(ET+ST)`, `C(DRD+ST)`, `RT(DRD)`, `RT(ET+ST)`, and `RT(DRD+ST)`. Results for algorithms `RT(RD)` and `RT(ST)` are not reported, since their computation times are too large when compared to the others. Algorithms `RT` had the ordered list A implemented as a skip list with probability $p = 0.25$. The algorithms were applied to the same benchmark instances considered by other authors [5]. Processing times are average results over 100 instances of each size.

Table 1. Algorithms and running times per update

	Update type	
Algorithm	weight decreases	weight increases
`RT(DRD)`	$O(n)$	$O(m)$ amortized
`RT(ET+ST)`	$O(\log n)$	$O(m \log n)$
`C(ET+ST)`	$O(\log n)$	$O(m \log n)$
`RT(DRD+ST)`	$O(\log n)$	$O(m)$ amortized
`C(DRD+ST)`	$O(\log n)$	$O(m)$ amortized

(a) Synthetic Inputs: Results for random sequences of 20,000 updates applied to randomly generated graphs with 4,000 vertices and 8,000 to 100,000 edges are displayed in Figure 2. The results show that algorithm `RT(DRD+ST)` is slightly faster than the other approaches for random update sequences. Also, DRD-trees can be used to significantly improve the computation times of algorithm `C(ET+ST)`.

As we are randomly selecting edges to update, the probability of increasing the weight of a tree edge is $(|V| - 1)/|E|$, decreasing with the increase of $|E|$ when $|V|$ is unchanged. Therefore, there are relatively fewer tree edge weights to be updated when the total number of edges increases and, consequently, the computation times become smaller. For all other cases (weight decreases and non-tree edge weight increases), we described algorithms that work in logarithmic time. The next experiments are more focused on the increase of tree edge weights, which can be more interesting than just randomly selecting any edge to update.

Results for structured sequences of 20,000 updates applied to randomly generated graphs with 4,000 vertices and 8,000 to 100,000 edges are shown in Figure 3. Here, 90% of the updates are tree edge weight increases and 10% are non-tree edge weight decreases. As predicted, the behavior in this case is the opposite to that in the previous situation. These are hard instances, since increasing the weight of tree edges and decreasing the weight of non-tree edges are the situations where Algorithms 2. and 3. are really called and used. In this context, the overall running time is dominated by the algorithms handling weight updates. The best options are variants `RT(DRD)` and `RT(DRD+ST)`, since DRD-trees perform better when a large number of connectivity queries has to be processed. Moreover, algorithms `C(DRD+ST)` and `C(ET+ST)` did not perform well in this case, showing their inability to deal efficiently with hard update sequences.

Figure 4 presents results for graphs composed by isolated cliques connected by a few inter-clique edges, named k-clique graphs [5]. The updates are only incremental, applied exclusively to inter-clique edges. These are very hard instances,

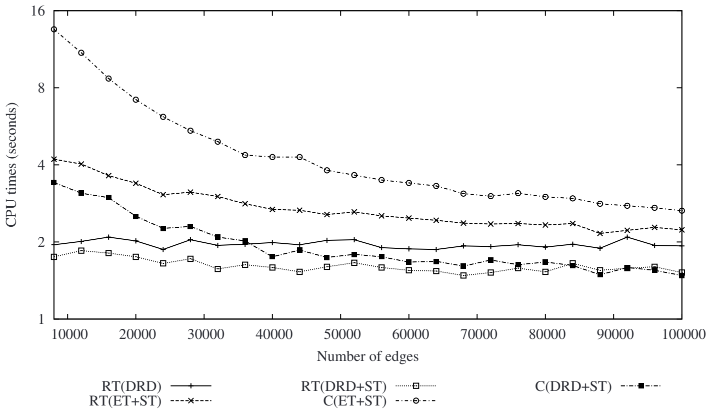

Fig. 2. CPU times for 20,000 random updates on randomly generated graphs

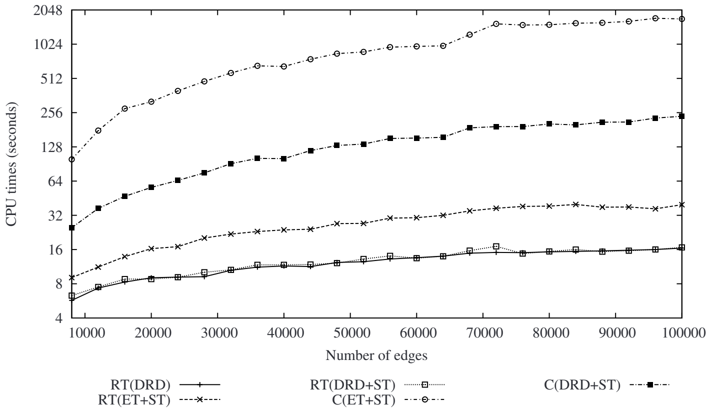

Fig. 3. CPU times for 20,000 structured updates on randomly generated graphs

since the set of candidates to replace the edge whose weight is increased is very small. The number of edges in this experiment ranges from 4,000 to 499,008, while the number of vertices is fixed at 2,000. The behavior of the algorithms is similar to that in Figure 3. The combination of edge weight increases with highly structured graphs resulted in the largest computation times among all instances considered in this section. Variants `RT(DRD)` and `RT(DRD+ST)` are the best options in most cases. However, the latter is more robust, since Algorithm 2. has logarithmic complexity due to the use of ST-trees (as one may want to have good performance for both incremental and decremental updates).

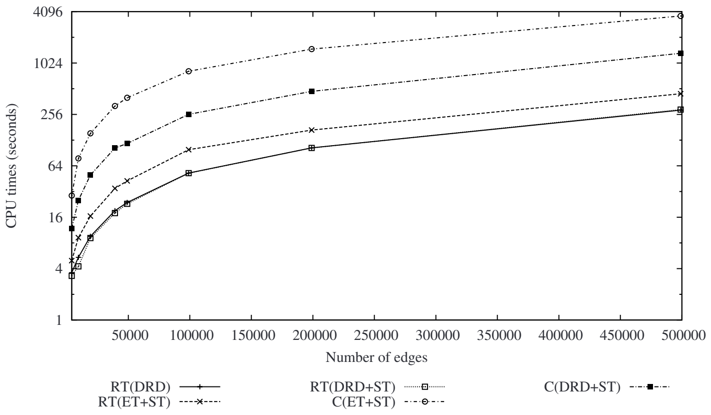

Fig. 4. CPU times for 20,000 incremental updates on k-clique instances

We conclude this section by reporting the number of connectivity queries performed by each approach – `C` and `RT` – on the k-clique instances corresponding to Figure 4. These numbers give an insight regarding the computation times obtained in all experiments above. Figure 5 shows that algorithms from [5] perform many more connectivity queries (per instance) than the proposed approach, what lead to the numerical advantage of algorithms `RT` along the experiments.

(b) Realistic Large Inputs: We now provide results for more realistic graphs from the DIMACS Implementation Challenge [6]. Table 2 displays results for four different types of graphs. Random4-n correspond to randomly generated graphs with $|E| = 4|V|$. Square-n graphs are generated in a two-dimensional square grid, with a small number of connections, while Long-n graphs are built on rectangular grids with long paths. Last, we present results for real-world instances named USA-road-d, derived from USA road networks. All update sequences are composed by 90% of weight increases and 10% of weight decreases.

The results in Table 2 show that variant `RT(DRD+ST)` performed better, providing fast algorithms for weight increases (using DRD-trees) and weight decreases (using ST-trees). This implementation was up to 110 times faster than `C(ET+ST)`, as observed for instance Random4-n.18.0. For road networks, variant `RT(DRD+ST)` also presented the best performance, achieving speedups of up to 51 times when compared to `C(ET+ST)`, as observed for instance USA-road-d.NY.

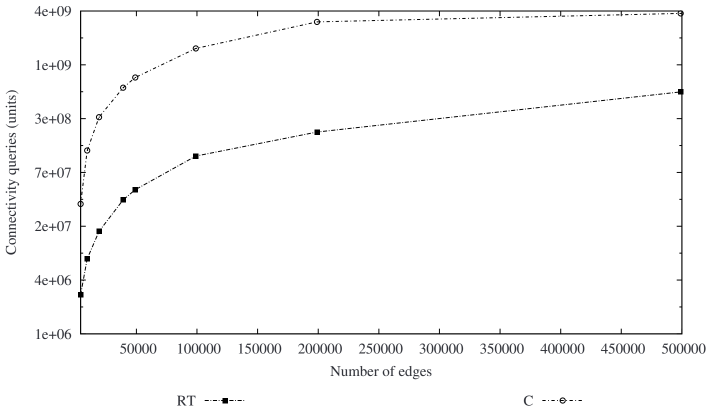

Fig. 5. Number of connectivity queries performed during the execution of 20,000 incremental updates on k-clique instances

Table 2. CPU times (seconds) for 20,000 structured updates on large instances

Instance	$\|V\|$	$\|E\|$	RT(DRD)	RT(ET+ST)	C(ET+ST)	RT(DRD+ST)	C(DRD+ST)
Random4-n.15.0	32,767	131,048	5.80	95.00	480.70	4.96	19.30
Random4-n.16.0	65,535	262,129	16.26	241.20	1,200.95	17.32	47.11
Random4-n.17.0	131,071	524,275	25.08	365.73	2,858.58	28.47	86.50
Random4-n.18.0	262,143	1,048,558	56.98	563.46	6,669.22	60.43	175.87
Random4-n.19.0	524,287	2,097,131	171.51	813.40	15,455.19	170.59	434.93
Square-n.15.0	32,760	65,156	4.87	32.60	141.13	4.19	10.33
Square-n.16.0	65,535	130,556	12.66	63.47	315.50	14.40	24.36
Square-n.17.0	131,043	261,360	28.23	101.89	847.71	30.42	51.03
Square-n.18.0	262,143	523,260	67.47	161.26	1,850.70	73.53	112.28
Square-n.19.0	524,175	1,046,900	132.73	324.63	4,321.13	140.09	229.80
Long-n.15.0	32,767	63,468	5.05	28.63	127.24	4.72	10.01
Long-n.16.0	65,535	126,956	15.03	45.25	313.92	16.17	26.86
Long-n.17.0	131,071	253,932	23.77	98.42	745.92	26.29	46.65
Long-n.18.0	262,143	507,884	59.35	184.20	1,683.73	62.95	103.20
Long-n.19.0	524,287	1,015,788	168.55	314.63	3,874.98	168.93	263.42
USA-road-d.NY	264,346	365,048	141.08	420.57	6,672.47	129.80	374.20
USA-road-d.BAY	321,270	397,414	188.44	674.85	6,832.83	174.84	416.60
USA-road-d.COL	435,666	521,199	233.69	889.54	7,351.65	216.42	560.23
USA-road-d.NW	1,207,945	1,410,384	589.47	3,196.67	23,843.47	566.80	1,518.80
USA-road-d.NE	1,524,453	1,934,008	752.20	4,112.73	47,990.40	737.60	2,152.56

(c) Dynamic vs. Non-Dynamic Algorithms: In this last section, we present results comparing the dynamic algorithms with the static ones. We considered the same instances used in the experiments reported in Table 2. In that situation, algorithm `RT(DRD+ST)` processed 20,000 updates for each instance, using the amount of time showed in Table 2. We let Kruskal's and Prim's [13] algorithms run for the same time `RT(DRD+ST)` have run. On average, Prim's and Kruskal's algorithms were able to compute from scratch only 379 and 851 minimum spanning trees, respectively.

The dynamic approach was, in average, 52 (resp. 23) times faster than Prim's (resp. Kruskal's) algorithm. Thus, even though the proposed approach has the same theoretical complexity of other classical algorithms, these results emphasize the performance and the usefulness of dynamic algorithms.

5 Concluding Remarks

We proposed a new framework for the implementation of dynamic algorithms for updating the minimum spanning tree of a graph subject to edge weight changes. An extensive empirical analysis of different algorithms and variants has shown that the techniques presented in this paper are very suitable to hard update sequences and outperform the fastest algorithm in the literature.

We also proposed an easy-to-implement data structure for the linking and cutting trees problem. While DRD-trees provide linear time implementations for almost all operations, they are considerably faster when used to handle a large amount of connectivity queries. The experimental analysis showed that this structure not only reduced the computation times observed for the algorithm of Cattaneo et al. [5], but also contributed to the fastest algorithms in the computational experiments: `RT(DRD)` and `RT(DRD+ST)`.

Acknowledgments

We acknowledge Renato F. Werneck for making available his implementation of ST-trees and Giuseppe F. Italiano for his implementation of algorithm `C(ET+ST)`, both used in the computational experiments performed in this paper.

References

1. Amato, G., Cattaneo, G., Italiano, G.F.: Experimental analysis of dynamic minimum spanning tree algorithms (extended abstract). In: Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 314–323, New Orleans (1997)
2. Buriol, L.S., Resende, M.G.C., Ribeiro, C.C., Thorup, M.: A memetic algorithm for OSPF routing. In: Proceedings of the 6th INFORMS Telecom, pp. 187–188, Boca Raton (2002)
3. Buriol, L.S., Resende, M.G.C., Ribeiro, C.C., Thorup, M.: A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing. Networks 46, 36–56 (2005)
4. Buriol, L.S., Resende, M.G.C., Thorup, M.: Speeding up shortest path algorithms. Technical Report TD-5RJ8B, AT&T Labs Research (September 2003)

5. Cattaneo, G., Faruolo, P., Ferraro-Petrillo, U., Italiano, G.F.: Maintaining dynamic minimum spanning trees: An experimental study. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 111–125. Springer, Heidelberg (2002)
6. Demetrescu, C., Goldberg, A., Johnson, D.: Ninth DIMACS implementation challenge – shortest paths, 2006. On-line reference at `http://www.dis.uniroma1.it/~challenge9/`, last visited in June 23 (2006)
7. Eppstein, D., Galil, Z., Italiano, G.F., Nissemzweig, A.: Sparsification – A technique for speeding up dynamic graph algorithms. Journal of the ACM 44, 669–696 (1997)
8. Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. SIAM Journal on Computing 14, 781–798 (1985)
9. Frederickson, G.N.: Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In: Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, pp. 632–641, San Juan (1991)
10. Henzinger, M.H., King, V.: Maintaining minimum spanning trees in dynamic graphs. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 594–604. Springer, Heidelberg (1997)
11. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In: Proceedings of the 30th ACM Symposium on Theory of Computing, pp. 79–89, Dallas (1998)
12. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. In: Proc. of the American Mathematical Society, vol. 7, pp. 48–50 (1956)
13. Prim, R.C.: Shortest connection networks and some generalizations. Bell Systems Technical Journal 36, 1389–1401 (1957)
14. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM 33, 668–676 (1990)
15. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. Journal of Computer and System Sciences 26, 362–391 (1983)
16. Spira, P.M., Pan, A.: On finding and updating spanning trees and shortest paths. SIAM Journal on Computing 4, 375–380 (1975)
17. Werneck, R.F., Tarjan, R.E.: Self-adjusting top trees. In: Proc. of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 813–822, Vancouver (2005)
18. Zaroliagis, C.D.: Implementations and experimental studies of dynamic graph algorithms. In: Fleischer, R., Moret, B.M.E., Schmidt, E.M. (eds.) Experimental Algorithmics. LNCS, vol. 2547, pp. 229–278. Springer, Heidelberg (2002)