Discrete Applied Mathematics 158 (2010) 404–425

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Maintaining dynamic minimum spanning trees: An experimental study☆

G. Cattaneo [a], P. Faruolo [a], U. Ferraro Petrillo [b,*], G.F. Italiano [c]

[a] Dipartimento di Informatica ed Applicazioni, Università degli Studi di Salerno, Baronissi (Salerno), Italy

[b] Dipartimento di Statistica, Probabilità e Statistiche Applicate, "Sapienza" - Università di Roma, Roma, Italy

[c] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Roma, Italy

ARTICLE INFO

Article history:
Received 15 February 2005
Received in revised form 24 April 2009
Accepted 17 October 2009
Available online 7 November 2009

Keywords:
Experimental analysis
Minimum spanning tree algorithms
Dynamic graphs

ABSTRACT

We report our findings on an extensive empirical study on the performance of several algorithms for maintaining minimum spanning trees in dynamic graphs. In particular, we have implemented and tested several variants of the polylogarithmic algorithm by Holm et al., sparsification on top of Frederickson's algorithm, and other (less sophisticated) dynamic algorithms. In our experiments, we considered as test sets several random, semi-random and worst-case inputs previously considered in the literature together with inputs arising from real-world applications (e.g., a graph of the Internet Autonomous Systems).

1. Introduction

In this paper we consider *fully dynamic graph algorithms*, namely algorithms that maintain a certain property of a graph that is changing dynamically. Usually, dynamic changes include the insertion of a new edge, the deletion of an existing edge, or an edge cost change; the key operations are edge insertions and deletions, however, as an edge cost change can be supported by an edge deletion followed by an edge insertion. The goal of a dynamic graph algorithm is to update the underlying property efficiently in response to dynamic changes. We say that a problem is *fully dynamic* if both insertions and deletions of edges are allowed, and we say that it is *partially dynamic* if only one type of operation (i.e., either insertions or deletions, but not both) is allowed. This research area has been blossoming in the last decade, and it has produced a large body of algorithmic techniques both for undirected graphs [18–20,22,29,28,27] and for directed graphs [9,13,14,16,17,26,31,32]. One of the most studied dynamic graph problems is perhaps the fully dynamic maintenance of a minimum spanning tree (MST) of a graph [4,18,21–23,29,28]. This problem is important on its own, and it finds applications to other problems as well, including many dynamic vertex and edge connectivity problems, and computing the k best spanning trees. Most of the dynamic MST algorithms proposed in the literature introduced novel and rather general dynamic graph techniques, such as the partitions and topology trees of Frederickson [22,23], the sparsification technique by Eppstein et al. [18–20] and the logarithmic decomposition by Holm et al. [29].

There is a general effort to bridge the gap between the design and theoretical analysis of dynamic graph algorithms, and their actual implementation, experimental tuning and practical performance evaluation. Towards this end, many

☆ This work has been partially supported by MIUR, the Italian Ministry for University and Research, under Project MAINSTREAM "Algorithms for Massive Information Systems and Data Streams" and by the University of Salerno under project "Sicurezza Dati, Computazione Distribuita e Compressione Dati". A preliminary version of this paper appeared in G. Cattaneo, P. Faruolo, U. Ferraro Petrillo, G.F. Italiano (2002) [10].

* Corresponding author. Tel.: +39 06 49910513; fax: +39 06 4959241.

E-mail addresses: cattaneo@dia.unisa.it (G. Cattaneo), pomfar@dia.unisa.it (P. Faruolo), Umberto.Ferraro@uniroma1.it (U.F. Petrillo), italiano@disp.uniroma2.it (G.F. Italiano).

URL: http://www.disp.uniroma2.it/users/italiano (G.F. Italiano).

0166-218X/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.dam.2009.10.005

researchers have been complementing this wealth of theoretical results on dynamic graphs with thorough empirical studies. In particular, Alberts et al. [2] implemented and tested algorithms for fully dynamic connectivity problems: the randomized algorithm by Henzinger and King [27], and sparsification [18] on top of a simple static algorithm. Amato et al. [6] and Ribeiro and Toso [35] proposed and analyzed efficient implementations of dynamic MST algorithms: the partitions and topology trees of Frederickson [22,23], and sparsification on top of dynamic algorithms [18]. Frigioni et al. [25] proposed efficient implementations of dynamic transitive closure algorithms, while Frigioni et al. [24], Demetrescu et al. [12,15] and Buriol et al. [9] conducted empirical studies of dynamic shortest path algorithms. In addition, Tarjan and Werneck [39] performed an extensive experimental analysis of several variants of dynamic trees that are commonly used in dynamic graph algorithms. Most of these implementations [2,6,18,22,23,26] have been wrapped up in a software package for dynamic graph algorithms [3]. Finally, Iyer et al. [30] implemented and evaluated experimentally the fully dynamic connectivity algorithm by Holm et al. [29], thus greatly enhancing our knowledge on the practical performance of dynamic connectivity algorithms.

The objective of this paper is to advance our knowledge of dynamic MST algorithms by following up the theoretical progress of Holm et al. [29] with a thorough empirical study. In particular:

(1) we present and experiment with efficient implementations of the dynamic MST algorithm by Holm et al. [29];
(2) we propose new simple algorithms for dynamic MST, which are not as asymptotically efficient as [29], but nevertheless seem quite fast in practice;
(3) we compare all these new implementations with previously known algorithmic codes for dynamic MST [6], such as the partitions and topology trees of Frederickson [22,23], and sparsification on top of dynamic algorithms [18].

We found that the implementations contained in [30] were particularly targeted and engineered for dynamic connectivity, so that an extension of this code to dynamic MST appeared to be a difficult task. After some preliminary tests, we decided to produce a completely new implementation of the algorithm by Holm et al., more oriented towards dynamic MST. We decided to not include in our experimentation the dynamic MST algorithm by Henzinger and King [28] as the algorithm by Holm et al. represents an evolution of this algorithm, and thus the approach it follows is likely to deliver faster performance. With this bulk of implementations, we performed extensive tests under several variations of graph and update parameters in order to gain a deeper understanding of the experimental behavior of these algorithms. To this end, we produced a rather general framework in which the dynamic graph algorithms available in the literature can be implemented and tested. Our experiments were run on randomly generated graphs and update sequences, and on more structured (non-random) graphs and update sequences, which tried to enforce bad update sequences on the algorithms. We also considered, as an instance of a real-world graph, a graph of the Internet Autonomous Systems.

1.1. Organization of the paper

The remainder of the paper is organized as follows. In Section 2 we describe the experimental settings and data sets for our implementations. In Section 3 we present a few simple algorithms that we have implemented for dynamic minimum spanning trees and discuss their experimental behavior. In Section 4 we describe and discuss the experimental behavior of more sophisticated implementations for dynamic minimum spanning trees, such as the algorithms by Frederickson [22,23], by Eppstein et al. [18] and by Holm et al. [29]. Section 5 reports more experimental results on all our implementations. Finally, we list some concluding remarks in Section 6.

2. Experimental settings

We have conducted our tests on a Pentium IV (1.3 GHz) with 1 GB of physical RAM, 16 KB L1-cache and 256 KB L2-cache under Linux 2.2. All our implementations are coded in C++ with the support of the LEDA [34] algorithmic software library, and are homogeneously written by the same people, i.e., with the same algorithmic and programming skills. We compiled our codes with the `GNU C++ 2.9.2` compiler with all the optimizations turned on (i.e., `-O` optimization flag). The algorithms' performances have been measured according to two distinct profiles. A general profile has been obtained by sampling the overall running time and the maximum amount of memory allocated during the algorithms' execution. On the other hand, we obtained a fine-grained profile by properly instrumenting the algorithms' source code so as to measure the total amount of time and the total number of calls required by each function.

In all our experiments, we generated a weighted graph $G = (V, E)$, and a sequence σ of update operations featuring i edge insertions and d edge deletions on this input graph G. We then fed each algorithm to be tested with G and the sequence σ. All the collected data were averaged on ten different runs. Building on previous experimental work on dynamic graph algorithms [2,6,30], we considered the following test sets:

- **Random graphs.** The initial graph G is generated randomly according to a pair of input parameters (n, m) reporting the initial number of vertices and edges of G. To generate the update sequence, we choose at random an edge to insert from the edges not currently in the graph or an edge to delete, again at random, from the set of edges currently in the graph. All the edge costs are randomly chosen using a uniform distribution in the range $[0, 2^{31} - 1]$. In addition, we also considered update sequences with a fixed percentage of tree updates (i.e., operations that change the MST).

- *k*-**Clique graphs.** These tests define a two-level hierarchy in the input graph G: we generate k cliques, each of size c, for a total of $n = k \cdot c$ vertices. We next connect these cliques with $2k$ randomly chosen inter-clique edges. In order to force more work on the implementations considered in this paper, we assign to inter-clique edges weights that are larger than the weights of intra-clique edges. This should make it more difficult to replace a deleted inter-clique edge when using algorithms that search for a replacement by scanning edges in non-decreasing order. For these k-clique graphs, we considered different types of update. On one side, we considered operations involving inter-clique edges only (i.e., deleting and inserting inter-clique tree edges). Since the set of replacement edges for an inter-clique tree edge is very small, this sequence seems particularly challenging for dynamic MST algorithms. The second type of update operation involved the set of edges inside a clique (intra-clique) as well, and considered several kinds of mix between inter-clique and intra-clique updates.

  As reported in [30], this family of graphs seems interesting for many reasons. First, it has a natural hierarchical structure, common in many applications, and not found in random graphs. Furthermore, investigating several combinations of inter-clique/intra-clique updates on the same k-clique graph can stress algorithms on different scenarios, ranging from worst-case inputs (inter-clique updates only) to more mixed inputs (both inter- and intra-clique updates).
- **Real-world graphs.** We also experimented on real-world graphs describing part of the Internet topology. We recall here that the Internet is essentially organized in two levels: the outer level is made of several interconnected Autonomous Systems, with each Autonomous System usually being a single administrative domain including several groups of IP network. In our tests with real-world graphs, we considered the graph of the Internet Autonomous Systems. The initial graph G, containing approximately 16,000 vertices and 31,000 edges, reflects a large portion of the graph of the Autonomous Systems existing in Internet at the beginning of September 2003. Then, the update sequence that we use is obtained by following the real updates which occurred to the graph of the Autonomous Systems throughout September 2003.

  From a practical point of view, these data have been extrapolated by processing periodical snapshots of the graph of the Internet Autonomous Systems collected by the "Route-Views" project [36]. We parsed this data by associating to every Autonomous System a distinct vertex and by associating an edge for every pair of adjacent Autonomous Systems. Since no information was available about the capacity of each link we assigned random weights to all edges. Finally, we assembled the update sequence by comparing pairs of subsequent snapshots and encoding the resulting changes by means of edge insertions and deletions.
- **Iyer et al. Worst-case inputs.** For sake of completeness, we adapted to minimum spanning trees the worst-case inputs introduced by Iyer et al. [30] for connectivity. These are inputs that try to force a bad sequence of updates for the algorithm by Holm et al. [29], and in particular try to promote as many edges as possible through the levels of its data structures. This is accomplished by generating an input graph containing a line of 2^k vertices connected by $(2^k - 1)$ edges. The update sequence consist of deletions only. The first deletion removes the middle edge of input graph, generating two lines of 2^{k-1} vertices. Then, the sequence continues by recursively deleting middle edges. In this way, every deletion will split the original spanning tree component into two different components having approximately the same size, thus maximizing the number of tree edges promotions. We refer the interested reader to reference [30] for the full details.

3. Simple algorithms

In this section we describe some simple algorithms that we have implemented for dynamic minimum spanning trees. These algorithms are basically a fast "dynamization" of the static algorithm by Kruskal (see e.g., [11,33]). We recall here that Kruskal's algorithm grows a forest by scanning all the graph edges by increasing cost: if an edge (v, w) joins two different trees in the forest, (v, w) is kept and the two trees are joined. Otherwise, v and w are already in the same tree, and (v, w) is discarded. The first algorithm, described in Section 3.1 uses the dynamic trees of Sleator and Tarjan [37] based on splay trees [38]. The second algorithm, described in Section 3.2, uses Euler trees [27] in addition to dynamic trees.

3.1. ST-based dynamic algorithm

Our dynamization of the algorithm by Kruskal is based on very simple ideas. Throughout the sequence of updates, we keep the following data structures: the minimum spanning forest, say *MST*, is maintained by using the dynamic tree data structure of Sleator and Tarjan [37] implemented with splay trees [38] (in short, ST-trees), while non-tree edges are maintained in an AVL balanced binary search tree [1], say *NT*, according to their costs. When a new edge (v, w) is to be inserted, we check with the help of dynamic trees whether (v, w) will become part of the solution. If this is the case, we insert (v, w) into *MST*, and then the edge with maximum cost in the cycle induced by the insertion of (v, w) will be deleted from *MST* and inserted into *NT*. Otherwise, (v, w) will not be a tree edge and we simply insert it into *NT*. All these operations require $O(\log n)$ amortized time.

When edge (v, w) has to be deleted, we distinguish two cases: if (v, w) is a non-tree edge, then we simply delete it from *NT* in $O(\log n)$ time. If (v, w) is a tree edge, its deletion disconnects the minimum spanning tree into T_v (containing v) and T_w (containing w), and we have to look for a replacement edge for (v, w). We examine non-tree edges in *NT* by increasing costs and try to apply the scanning strategy of Kruskal on T_v and T_w: namely, for each non-tree edge $e = (x, y)$, in increasing

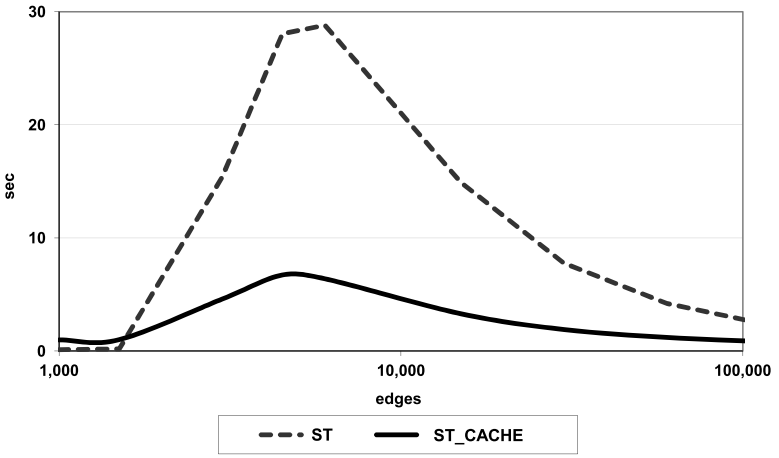


Fig. 1. Performance of ST and of ST_CACHE on random graphs with 3000 vertices and different densities. Update sequences contained 10,000 random edge insertions and 10,000 random edge deletions.

order, we check whether e reconnects T_v and T_w: this can be done via findroot(x) and findroot(y) operations in Sleator and Tarjan's trees. Whenever such an edge is found, we insert it into *MST* and stop our scanning. Note that a tree edge deletion can be the most expensive operation: its worst case running time can be as high as $O(m \log n)$, while all the other operations can be implemented in $O(\log n)$. We refer to this algorithm as ST.

It is not difficult to see that findroots are the bottleneck of this algorithm. Indeed, a tree edge deletion in the graph pushes dynamic trees into a particular work area, as they are subject to many findroots. In more detail, a tree edge deletion triggers the following operations on the dynamic trees: first, one cut operation that removes the deleted edge, and breaks the original tree into two smaller trees; next, a batch of findroots on the two trees to find the replacement edge; finally, one link operation to reconnect the two trees via the replacement edge.

One simple heuristic that we used to speed up tree edge deletions was caching findroots during the search for a replacement edge: namely, during a tree edge deletion, we stored the results of the findroot operations performed, and reused them whenever possible. This way, subsequent findroots issued on the same node could be carried out in $O(1)$ time by a simple table lookup. We refer to this algorithm as ST_CACHE. The total time required by a tree edge deletion is thus $O(m + k \cdot \log n)$ in the worst case, where k is the total number of findroot operations issued on different vertices during the deletion. Since $k \leq n$, the worst case for ST_CACHE is $O(m + n \log n)$. Despite their large asymptotic running times, we note that both ST and ST_CACHE are much simpler than the other algorithms considered in this work, and use fast data structures, such as the dynamic trees [38]. We therefore expected them to be very fast in practice, especially in update sequences containing few tree edge deletions or in cases when, after deleting tree edge (v, w), the two trees T_v and T_w are easily reconnected (e.g., the cut defined by (v, w) contains edges with small costs). This was indeed confirmed by our experiments.

3.1.1. Experiments with ST and ST_CACHE

In our first test we ran both ST and ST_CACHE on random graphs with different densities. To this end, we generated a sequence of 20,000 random operations, evenly mixed between insertions and deletions. As documented in Fig. 1, our results showed that ST and ST_CACHE were indeed fast in many situations. However, the most difficult cases for them were on sparse graphs, and, in particular, on graphs with around $2n$ edges. The theory of random graphs [8] tells us that, when $m = 2n$, the graph contains a giant component of size $O(n)$ and smaller components of size $O(\log n)$. In this case, a random edge deletion is likely to disconnect the minimum spanning forest and to cause the scanning of many non-tree edges in the quest for a replacement. As the graph density increases, a random update operation will less frequently involve a tree edge: this explains the improved performance of the two algorithms on dense graphs.

As illustrated in Fig. 1, by caching findroots we are able to improve the overall performance of ST, with the only exception of very sparse graphs. In this case, indeed, the number of non-tree edges (and thus potential candidates for edge replacements) is very small, and consequently caching findroot operations seems to introduce more overhead than savings. The gain of ST_CACHE over ST becomes especially significant in the case of sparse graphs, i.e., graphs with around $2n$ edges: in this case tree edge deletions are more frequent and consequently there are more findroots involved.

This gain is even more clear in Fig. 2 and 3, which report the results of an experiment with a high percentage of tree edge insertions and deletions: the more tree edge deletions, the more findroot caching yields substantial savings. In this experiment we distinguished between the time spent in findroots and the remaining execution time. As is shown in Figs. 2 and 3, the performance of ST and ST_CACHE is dominated by the findroot costs, shown as squared area in both figures. In addition, it is interesting to note that the running times of both algorithms on sequences with a high percentage of tree edge

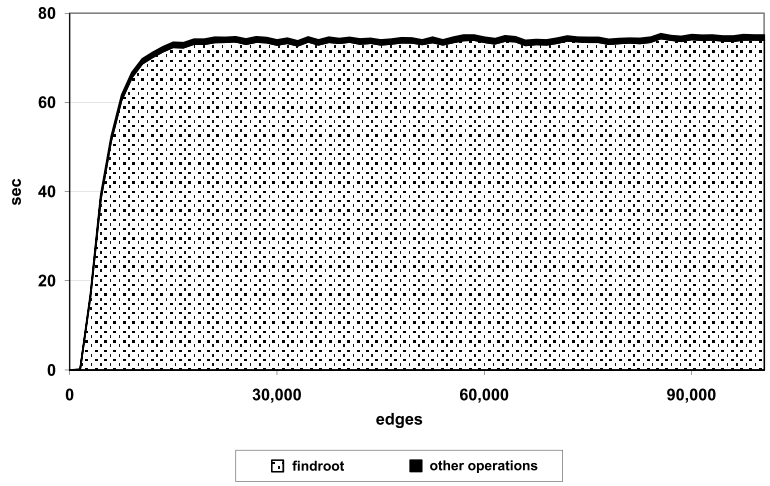


Fig. 2. Performance of ST on random graphs with 3000 vertices and different densities. The squared area reports the percentage of work done to process findroot operations. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 45% of the operations were tree edge insertions and 45% of the operations were tree edge deletions.

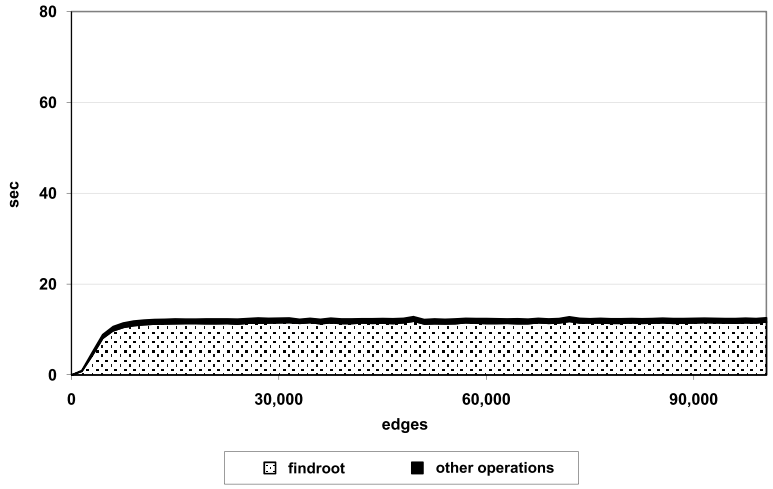


Fig. 3. Performance of ST_CACHE on random graphs with 3000 vertices and different densities. The squared area reports the percentage of work done to process findroot operations. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 45% of the operations were tree edge insertions and 45% of the operations were tree edge deletions.

insertions and deletions does not decrease with the graph density. This is due to the fact that on random graphs both ST and ST_CACHE tend to find a replacement for a deleted tree edge in a relatively few steps, independently of the graphs' density.

The findroot caching strategy becomes even more effective in the case of more structured input sets, such as k-clique graphs. We recall here that these input sets consist of k cliques, each of size c, connected by $2k$ inter-clique edges. Moreover, inter-clique edges are assigned weights that are larger than those assigned to intra-clique edges. Thus, the deletion of an inter-clique tree edge in a k-clique graph will force both ST and ST_CACHE to exhaustively scan the whole set of intra-clique non-tree edges before finding a replacement edge (if one exists). In such a context, the caching performance gain is even more significant than for random input sets as we have to perform many more findroots. This is clearly shown in Fig. 4 where we report the results of an experiment on k-clique graphs. Namely, we compared the performance of both ST and ST_CACHE on different k-cliques input sets when processing a batch of 20,000 operations featuring a vast majority of inter-clique tree edges updates. The worst performance on k-clique input sets is achieved in case of few big cliques as there is a larger number of non-tree edges that need to be considered in the quest for a replacement.

In summary, our experiments with ST and ST_CACHE confirmed our theoretical intuition. They are both fast in handling edge insertions, edge deletions that do not change the solution, and tree edge deletions where the replacement edge can be found in a few steps. Furthermore, it seems that caching findroots always pays off, as ST_CACHE is always faster than ST except for very sparse graphs.

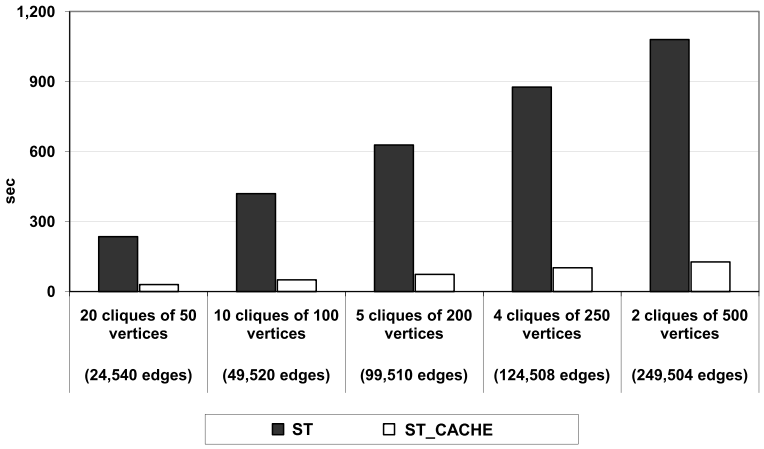


Fig. 4. Running times of ST and ST_CACHE on cliques of different sizes. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 45% of the operations were inter-clique edge insertions and 45% of the operations were inter-clique edge deletions.

3.2. ET-based dynamic algorithm

Since the bottleneck of ST_CACHE is related to findroots on dynamic trees triggered by tree edge deletions, we explored whether data structures which are simpler than dynamic trees could be of any benefit. In particular, we designed another algorithm, referred to as ET_CACHE, which uses Euler Tour trees (in short, ET-trees) in addition to ST-trees. We recall here that a ET-tree (see, e.g., [27] for more details) is a balanced search tree used to maintain efficiently the Euler Tour of a given tree. ET-trees have some interesting properties that are very useful in dynamic graph algorithms. In our implementation, we used the randomized search tree of Aragon and Seidel [7] to support the ET-trees.

In ET_CACHE we keep information about tree edges both with an ST-tree and with an ET-tree: we use ST-trees for handling edge insertions, and use ET-trees for handling edge deletions, i.e., for checking whether a non-tree edge $e = (x, y)$ reconnects the minimum spanning tree. This is the only difference with algorithm ST_CACHE. In particular, we use a table to cache the results of findroots during the search for a replacement edge. Note that there is no difference in the asymptotic time bounds for ST_CACHE and ET_CACHE. From the practical viewpoint, however, we would expect findroots to be faster on ET-trees than on ST-trees: in the latter case indeed a findroot would imply a change (splay) in the data structure. When findroot operations are no longer the bottleneck, however, we expect that the overhead of maintaining two data structures in tandem (i.e., both ET-trees and ST-trees) may become significant.

Stimulated by the simplicity of findroots on ET-trees, we decided to try to exploit more the instruction-level parallelism in the source code. In particular, rather than issuing two findroots in sequence, we tried to force their execution in parallel in the CPU (instruction-level parallelism). This was simply done by properly interleaving the code for the two findroots into one function which checks whether two nodes belong to the same tree. In particular, we use a function, named `common_root_ancestor`, which checks whether two nodes belong to the same tree by putting into one conditional loop the code needed to traverse the two trees simultaneously. If either traversal reaches the root, then we continue by only searching for the root of the other vertex, and return at the end of both searches the comparison between the roots. The C++ code of `common_root_ancestor` is shown in Fig. 5. In the assembler code resulting from the compilation of this function, the operation of traversing each level of the two trees consists of two distinct sets of assembler instructions that can be executed in parallel by modern CPUs because they do not share any common data. We call the resulting algorithm ET_CA.

The advantage of this code over ET_CACHE, which issues two separate findroots in sequence, should be clear. Indeed, we expect ET_CA to exploit more parallelism at the instruction level, by feeding the processor with more instructions and thus producing fewer CPU stalls due to cache misses. On the other side, interleaving findroots into one single function interferes with the findroot caching mechanism: in particular, with caching in place, we are not always able to interleave the two findroots. This raises the question on whether the instruction-level parallelism could be worth giving up the findroot caching strategy. The intuition is that the denser the input graph, the more likely it is that many findroots are performed on the same node during a tree edge deletion, and thus that findroot caching (ET_CACHE) should become preferable to the instruction-level parallelism (ET_CA). On the contrary, we expect that the sparser the input graph, the more likely is that instruction-level parallelism yields faster code than findroot caching.

3.2.1. Experiments with ET_CA, ET_CACHE and ST_CACHE

From the experimental viewpoint, ET_CACHE was comparable to ST_CACHE: namely, the benefits of issuing findroots on ET-trees seem to be canceled out by the cost of keeping the ET-trees and ST-trees data structures in tandem.

As we expected, the instruction-level parallelism of ET_CA yields slightly better results than the findroot caching strategy of ET_CACHE and ST_CACHE when applied to random sparse graphs. In this case, the probability for a findroot operation to

```
bool common_ancestor(node x, node y){
  if ( x==nil || y==nil ) return false;
  if ( x == y ) return true;
  while( 1 ){
       if (x->parent == nil){
            while (y->parent != nil)
              y = y->parent;
            return (y==x);
       }
       if (y->parent == nil){
            while (x->parent != nil)
              x= x->parent;
            return (y==x);
       }
       x = x->parent;
       y = y->parent;
  }
}
```

Fig. 5. Source code of `common_root_ancestor`.

be issued several times on the same node is very low, thus reducing the benefits of the caching strategy. The situation changes when turning to denser graphs, as in this case several findroots will likely be issued on a same node. However, since in this case replacement edges are typically found in few steps, `ET_CA` is only slightly slower than `ET_CACHE` and `ST_CACHE`. This is illustrated in Fig. 6 where we compare the time needed by `ET_CACHE`, `ET_CA` and `ST_CACHE` to process sequences of 20,000 updates on random graphs. Update operations contained 45% of tree edge deletions. We expect the performance gap between `ST_CACHE` and `ET_CACHE` on one side and `ET_CA` on the other side to become larger when increasing the number of findroot operations to be issued per tree edge deletion. This has been confirmed by our experiments on k-clique input sets. We tried several different cliques sizes together with sequence of operations having 90% inter-clique tree updates evenly mixed between insertions and deletions. This choice will force a larger number of findroots to be issued. This is a scenario where findroot caching seems to get a big advantage as illustrated in Fig. 7. This becomes more evident on graphs having a few big cliques as in this case there is a larger number of non-tree edges.

We conclude this section by mentioning that, in order to keep the advantages of both findroot caching and instruction level parallelism, we tried to implement a hybrid version of `ET` capable of switching between the two heuristics. Our experiments with this hybrid algorithm have been rather discouraging, however. This is probably due to the overhead needed for a good selection of either heuristic.

4. Sophisticated algorithms

4.1. The algorithms by Frederickson

Let $G = (V, E)$ be a graph, with a minimum spanning tree T. The first ingredient of the algorithms of Frederickson [22,23] is *clustering*. Namely, the algorithm maintains a suitable partition of the vertex set V into connected subtrees in T, so that each subtree is only adjacent to a few other subtrees. Basically, this partition divides G evenly into $O(m/z)$ clusters of size z each, with $z \geq 0$ being an integer denoting the number of edges in the cluster, and m being the number of edges of G. A *topology tree* is a representation of the minimum spanning tree T using a different encoding: namely, it is a tree with logarithmic depth, formed by recursive clustering. A 2-*dimensional topology tree* is formed from pairs of nodes in a topology tree, and allows us to maintain efficiently information about the edges in $E - T$. Frederickson gives three different algorithms for maintaining the minimum spanning tree of a graph. Algorithm `Fred1` is based on clustering only and obtains time bounds of $O(m^{2/3})$ per update. If the partition is applied recursively and a topology tree is associated with each cluster, we get `Fred2`, which yields better $O((m \log n)^{1/2})$ time bounds per update. Finally, algorithm `Fred3` uses 2-dimensional topology trees to achieve a time bound of $O(m^{1/2})$ per update. Frederickson defined actually two different partitions (one in [22] and the other in [23]), which have similar theoretical behavior. These two different partitions give rise to six different algorithms: `Fred1-85`, `Fred2-85`, and `Fred3-85` based on the partition defined in [22], and `Fred1-91`, `Fred2-91`, and `Fred3-91` based on the partition defined in [23].

The experiments presented in [6] have shown that `Fred1-85` is generally faster in practice than all the other variants. This seems to be mainly due to several factors. First of all, the average number of clusters affected by an update when using the partitioning scheme defined in [23] is much larger than when using the partitioning scheme defined in [22], thus resulting in longer execution times. Second, the overhead needed to maintain and to update the standard topology trees and the 2-dimensional topology trees seems to cancel the better time bounds of `Fred2` and `Fred3` over `Fred1`. Starting from these considerations, the authors of [6] developed another variant of `Fred1-85`, based on the usage of *light partitions*

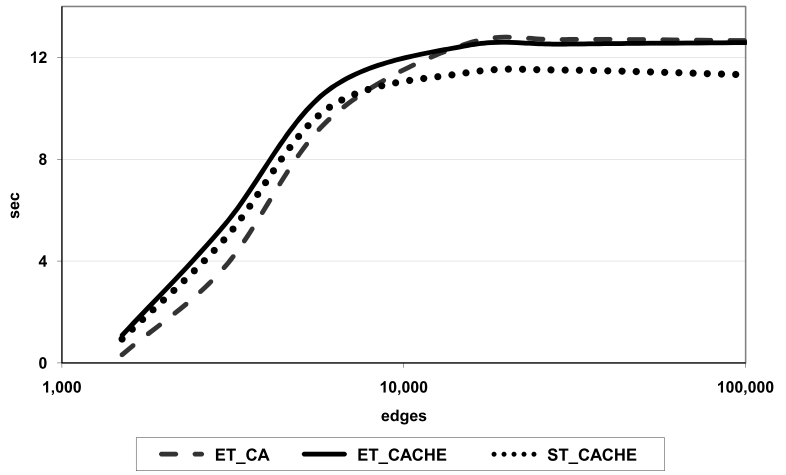


Fig. 6. Performance of ET_CACHE, ET_CA and ST_CACHE on random graphs with 3000 vertices and different densities. We used a logarithmic scale on the horizontal axis in order to highlight the behavior of the three algorithms on sparse graphs. Update sequences featured 20,000 operations: 45% of the operations were tree edge insertions and 45% of the operations were tree edge deletions.

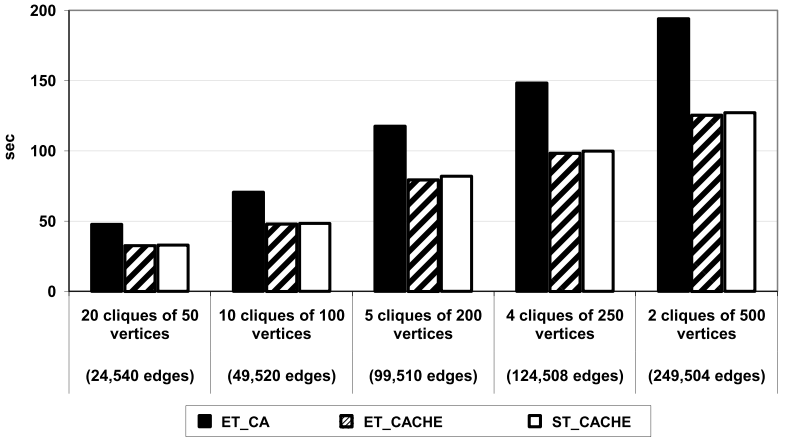


Fig. 7. ET, ET_CA and ST_CACHE total running times on cliques of different sizes. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 45% of the operations were inter-clique edge insertions and 45% of the operations were inter-clique edge deletions.

of order $\lceil m^{2/3} \rceil$ instead of balanced partitions, which performed better than `Fred1-85` both on random and non-random input sets.

4.2. Sparsification

Sparsification [18–20] is a general technique that applies to a wide variety of dynamic graph problems. It is used on top of a given algorithm, in order to speed it up, and can be used as a *black box*. Let $\mathcal{A}$ be an algorithm that solves a certain problem in time $f(n, m)$ on a graph with n vertices and m edges. There are two versions of sparsification: *simple sparsification* and *improved sparsification* [18]. When simple sparsification is applied to $\mathcal{A}$, it produces a better bound of $O(f(n, O(n)) \log(m/n))$[1] for the same problem. This is achieved by means of a suitable graph decomposition into smaller subgraphs so that $\mathcal{A}$ is called at most $O(\log(m/n))$ times on graphs with $O(n)$ edges (rather than once on a graph with m edges). Improved sparsification uses a more sophisticated decomposition and yields a $O(f(n, O(n)))$ time bound.

Sparsification is based on the concept of a *certificate*, which can be formalized as follows. Let $\mathcal{P}$ be any given graph property, G be a graph, and $c > 0$ be a given constant. A *sparse certificate* for $\mathcal{P}$ in G is a graph G' on the same vertex set as G such that the following is true:

(i) for any H, $G \cup H$ has property $\mathcal{P}$ if and only if $G' \cup H$ has the property; and
(ii) G' has no more than cn edges.

[1] We let $\log x$ stand for $\max\{1, \log_2 x\}$, so $\log x$ is never smaller than 1 even when $x < 2$.

Let G be a graph with m edges and n vertices. Simple sparsification works as follows: we partition the edges of G into a collection of $O(m/n)$ subgraphs with n vertices and $O(n)$ edges each. In our implementation, we choose subgraphs with no more than $4n$ edges. The information relevant for each subgraph is summarized in a *sparse certificate*. We next merge certificates in pairs, producing larger subgraphs which are made smaller by again computing their certificate. This is applied recursively, yielding a balanced binary tree in which each node represents a sparse certificate. This tree has $O(m/n)$ leaves, and hence its height is $O(\log(m/n))$. The total number of tree nodes is $O(m/n)$ and each node requires $O(n)$ space for the sparse certificate: thus, the sparsification method requires a total of $O(m+n)$ space and does not incur any asymptotic space blow up.

Each update involves examining the certificates contained in at most two tree paths from a leaf up to the tree root: this results in considering at most $O(\log(m/n))$ small graphs with $O(n)$ edges each, instead of one large graph with m edges, and explains how a $f(n, m)$ time bound can be sped up to $O(f(n, O(n)) \log(m/n))$. We refer the reader to [18] for the full details of the method.

The sparsification technique has been used in [6] to speed-up the `Fred1-85` algorithm by Frederickson. This implementation, referred to as `Spars(1-85)`, defines the minimum spanning tree itself as the certificate. This yields an algorithm that maintains a minimum spanning tree of a dynamic graph in $O(n^{2/3} \log(m/n))$ worst–case time per update. The same technique has also been applied to `Fred1-Mod`, the variant of `Fred1-Mod` using light partitions. According to the results presented in [6], this last variant was generally the fastest in practice with respect to the other variants of Fredericksons's algorithms, both on random and non-random inputs. This is probably due to the ability of light partitions to force less work on the algorithm's data structures than balanced partitions. We did not include in our experimentation the improved sparsification technique as it is mainly of theoretical interest. It uses a much more sophisticated graph decomposition than simple sparsification in order to cut the logarithmic factor, and it is likely to provide small benefits in practice. So, in the rest of this paper we will consider only the implementation of the simple sparsification technique using light partitions, referring to it as to `Spars`.

4.3. The algorithm by Holm et al.

In this section, we quickly review the algorithm by Holm et al. for the dynamic minimum spanning tree problem; here the algorithm is HDT. We start with the deletions-only algorithm, and then sketch how to transform this into a fully dynamic algorithm. The full details of the method can be found in [29]. Next, we present several heuristics we developed for HDT together with their experimental evaluation.

4.3.1. The decremental minimum spanning tree algorithm

We maintain a minimum spanning forest F over a graph G having n vertices and m edges. All the edges belonging to F will be referred to as *tree edges*. Note that when a tree edge is removed, it is replaced (when a replacement exists) by a heavier edge. The main idea behind the algorithm is to obtain a better time bound by only searching heavier edges. Thus, the algorithm partitions the edges of G into different levels, where the edges with smaller weights are placed in higher levels, and those with larger weights are placed in lower levels. Now, whenever we delete tree edge (v, w), we start looking for a replacement edge at level $\ell(v, w)$. If this search fails, we consider edges at level $\ell(v, w) - 1$ and so on until a replacement edge is found or there are no more levels to consider. This strategy is effective if we can arrange the edge levels so that replacement edges can be found quickly. To achieve this task, we promote to a higher level all the edges unsuccessfully considered for a replacement.

To be more precise, we associate to each edge e of G a level $\ell(e) \leq L = \lfloor \log n \rfloor$. For each i, we denote by F_i the sub-forest containing all the edges of F having level at least i. Thus, $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$. The following invariants are maintained throughout the sequence of updates:

(1) If (v, w) is a non-tree edge, v and w are connected in $F_{\ell(v,w)}$
(2) The maximum number of nodes in a tree in F_i is $\lfloor n/2^i \rfloor$. Thus, the maximum relevant level is $L = \lfloor \log n \rfloor$.
(3) If e is the heaviest edge on a cycle C in G, then e has the lowest level on C.

We now briefly define the two operations Delete and Replace needed to support deletions.

Delete(e). If e is not a tree edge then it is simply deleted. If e is a tree edge, first we delete it. Next, we have to find a replacement edge that maintains the invariants listed above and keeps a minimum spanning forest F. Since F was a minimum spanning forest before, by invariant (3) we have that a candidate replacement edge for e cannot be at a level greater than $\ell(e)$, so we start searching for a candidate at level $\ell(e)$ by invoking operation Replace$(e, \ell(e))$.

Replace$((v, w), i)$. Assuming that there is no replacement edge on level $> i$, it finds a replacement edge in the highest level $\leq i$, if any. The algorithm works as follows. Let T_v and T_w be the trees in F_i containing v and w, respectively. After deleting edge (v, w), we have to find the minimum cost edge connecting again T_v and T_w. Suppose without any loss of generality that T_v has a smaller size than T_w. First, we move all edges of level i of T_v to level $(i + 1)$, then, we start by considering all edges on level i incident to T_v in a non-decreasing weight order. Let f be the edge currently considered: if f does not connect T_v and T_w, then we promote f to level $(i + 1)$ and we continue the search. If f

connects T_v and T_w, then it is inserted as a replacement edge (without changing its level) and the search stops. If the search fails, we call Replace($(v, w), i-1$). When the search fails also at level 0, we stop as there is no replacement edge for (v, w).

4.3.2. The fully dynamic algorithm

Starting from the deletions-only algorithm, Holm et al. obtained a fully dynamic algorithm by using a refinement of a technique by Henzinger and King [28] for developing fully dynamic data structures starting from deletions-only data structures. To obtain such a reduction, they combine this technique with a contraction technique by Henzinger and King [28]. The result is a fully dynamic algorithm which supports edge insertions and deletions in $O(\log^4 n)$.

4.3.2.1. Reduction. The original reduction technique by Henzinger and King [28] makes it possible to obtain a fully-dynamic minimum spanning tree data structures starting from a deletions-only data structures. Insertions are supported by means of a "batch insertion" operation which rebuilds some of the deletions-only data structures in order to implement the insertion of a set of edges. The refined technique by Holm et al. does not use a batch insertion operation. Instead, edge insertions are supported by means of an *Update* operation which rebuilds some of the deletions-only data structures every time a single insertion changes the structure of a local or global spanning forest.

In the refined technique, we maintain a series of graphs $\mathcal{A} = A_1, \ldots, A_{s-1}, A_s, s = \lceil \log m \rceil$, where each A_i is a subgraph of G containing at most 2^i non-tree edges. Note that A_s is not necessarily G, and that there are not necessarily any containment relationships among different subgraphs A_i. The A_i's at the start of the sequence are empty, but one of them may be rebuilt following an *Update* operation. An A_i just rebuilt will contain a copy of a minimum spanning forest F plus some extra edges. Throughout the algorithm, all edges of G will be in at least one A_i, although the same edge may appear in several A_i.

We denote by LF_i the local spanning forest maintained on each A_i. We will refer to edges in LF_i as *local tree edges* while we will refer to edges in F as *global tree edges*. Since all edges in G will be in at least one A_i, $F \subseteq \bigcup_i LF_i$. During the algorithm we maintain the following invariant:

(1) For each global non-tree edge f, there is exactly one i such that f is a local non-tree edge in A_i and if $f \in LF_j$ then it must be $j > i$.

Without loss of generality assume that the graph is connected, so that we will have an MST rather than a minimum spanning forest. At this point, we use a dynamic tree of Sleator and Tarjan [37] to maintain the global MST and to check if update operations will change the solution. Here is a brief explanation of the update procedures:

Insert(e). Let be $e = (v, w)$. If v and w are not connected in F by any edge then we add e to F. Otherwise, we compare the weight of e with the heaviest edge f on the path from v to w. If e is heavier, we just update $\mathcal{A}$ with e, otherwise we replace f with e in F and we update $\mathcal{A}$ using f.

Delete(e). We delete e from all the A_i and we collect in a set R the replacement edge, if any, returned from applying to each LF_i the Delete algorithm of the deletions-only data structure. Then, we check whether e is in F. If so, we search in R for the minimum cost edge reconnecting F. If such an edge exists we add it to F. Finally, we update $\mathcal{A}$ using R.

Update $\mathcal{A}$ with edge set. D We find the smallest j such that $|(D \bigcup_{h \le j}(A_h \setminus LF_h)) \setminus F| \le 2^j$. Then we set

$$A_j = F \cup D \cup \bigcup_{h \le j}(A_h \setminus LF_h)$$

and we initialize A_j as a MST deletions-only data structure. Finally, we set $A_h = \emptyset$ for all $h < j$.

4.3.2.2. Contraction. The initializations required by the update operation are one of the crucial points in HDT. In order to bound the number of local tree edges to be initialized at any update, the algorithm deploys a compression of some subpaths made of local tree edges only into one single *super edge*. Instead of adding F to A_j in an update, the algorithm will add a forest F' of super edges *ep*, each representing a path P in F. This compression allows one to bound by a constant factor the number of local tree edges to be initialized at each update.

F' is built by taking the end-points of non-tree edges in A_j one at time: each end-point is added as a *super vertex* to the tree of super edges. Then, the tree paths connecting pairs of super vertices, which we refer to as *super paths*, are contracted into super edges. The insertion of a vertex v into the tree of super edges is handled as follows. First, we mark v as super vertex. If v is not connected in G to any other super vertex, we are done. Otherwise, we find the nearest vertex x on some path in F'. If x is a super vertex, we simply add the new super path $v \ldots x$ to the tree of super edges. Otherwise, since the super paths only intersect in the super vertices, x must be internal to a unique current super path $a \ldots b$. Now we mark x as a new super vertex, we delete super path $a \ldots b$ and we add the three super paths $v \ldots x$,$a \ldots x$ and $b \ldots x$. To mark efficiently a super path with a super edge, Holm et al. extended the top trees of Alstrup et al. [5]. This can be done by maintaining extra path information on each vertex of the top tree, such as the path the vertex belongs to and the nearest vertex on this path.

Let F_j be a local copy of the global spanning tree associated to A_j maintained using a top tree. This copy has to be updated to F each time A_j is rebuilt. To this end, we keep for each A_j a log of all changes occurred to the global MST since last A_j's reinitialization. In order to update F_j, we delete from it all edges deleted from F since last initialization, then we insert in F_j all the edges inserted in F since last initialization.

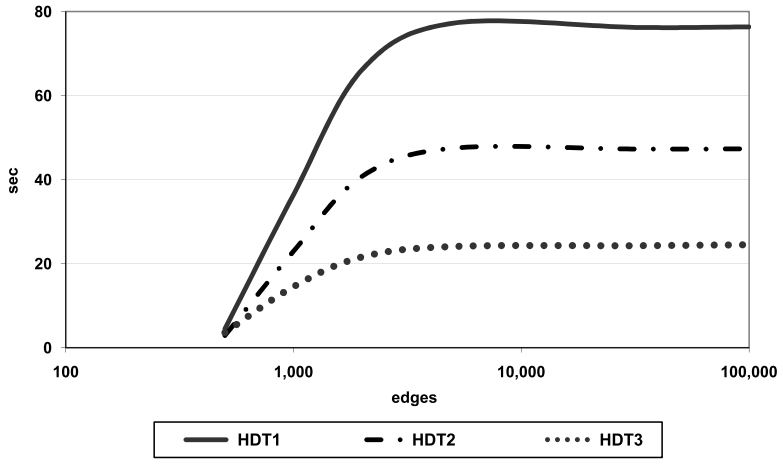


Fig. 8. Running times of HDT1, HDT2 and HDT3 on random graphs with 1000 vertices. We used a logarithmic scale on the horizontal axis in order to highlight the behavior of the three algorithms on sparse graphs. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 45% of the operations were tree edge deletions and 45% of the operations were tree edge insertions.

4.3.3. Path compression

During our computational study of HDT we observed that one of the most complex and expensive tasks is maintaining and updating super paths. For this reason, we performed some tuning and developed three implementations of HDT that differ in the strategy adopted for compressing super paths. The first implementation, here referred to as HDT1, follows closely the algorithm in the original paper and uses a complete implementation of the extended top trees we developed to this end.

The second implementation, here referred to as HDT2, adopts the compression technique used by the 2-edge connectivity algorithm presented by Henzinger and King in [27], which is briefly outlined in the following. Let F' be the compressed forest to be built starting from an existing deletions-only data structure A_j. We consider every non-tree edge (v, w) in A_j, we mark the path (v, w) in F and call the resulting marked subforest F_j^m. The *super vertices* in F' are all vertices which are leaves and all vertices which have degree at least three in F_j^m. A *super edge* (v, w) is in F' if and only if v and w are super vertices, the path between v and w is in F_j^m and there are no other super vertices which are on the path between v and w. We use dynamic trees [37,38] in order to maintain and to look for the super edge covering a certain edge or a certain vertex.

The third implementation, here referred to as HDT3, does not implement any path compression at all. Instead, every deletions-only data structure maintains an uncompressed copy of the global spanning tree F. The problem that arises here is to bound the reinitialization cost of the deletions-only data structures. That is, if path compression is not used then HDT3 should pay an $O(n)$ cost every time a deletions-only data structure A_j is rebuilt in order to insert into A_j all global tree edges. We reduced this cost by adopting the rollback technique described by Henzinger and King in [28]. Namely, we keep a list of all changes occurred to each deletions-only data structure A_j since the last reinitialization. When A_j has to be rebuilt we rollback all the changes occurred in it since last reinitialization (i.e., inserted edges are deleted while deleted edges are reinserted at level 0). Then, we update A_j by committing in it all global MST modifications occurred since last reinitialization.

4.3.4. Experiments with HDT1, HDT2 and HDT3

Our first experiments aimed at measuring the cost of maintaining compressed paths by comparing HDT1, HDT2 and HDT3 on random input sets. The main contributions to this cost are the time spent in updating the compressed local spanning forest maintained by each deletions-only data structure with respect to the global MST modifications (here referred to as *sync* time) and the time spent by these data structures in compressing paths made of tree edges into super edges (here referred to as *compress* time). Note that the number of these paths is proportional to the number of non-tree edges used to rebuild a deletions-only data structure (see Section 4.3.2).

To begin with, we considered random graphs with a fixed number of vertices ($n = 1000$) and different densities. In order to force as much work as possible on the path compression, we used sequences of operations with a high percentage of tree updates (i.e., operations that change the MST). In such a setting, dense graphs will be more challenging for the algorithm. Indeed, the average number of candidate replacement edges returned by a tree edge deletion on a sparse graph will be in average lower than on a dense graph having the same number of vertices. Therefore, the resulting *compress* time is expected to be smaller. In addition, the frequency of MST disconnections (i.e., tree edge deletions that cannot be replaced) on sparse random graphs is expected to be much higher than on dense graphs, thus reducing the number of MST modifications. This has been confirmed by our experimental results (see Fig. 8). On this input set the performance of both HDT1 and HDT2 seem to be dominated by the cost of path compression and they are significantly slower than HDT3. We also note that a compression based on dynamic trees (HDT2) seems to be more efficient than a compression based on top trees (HDT1).

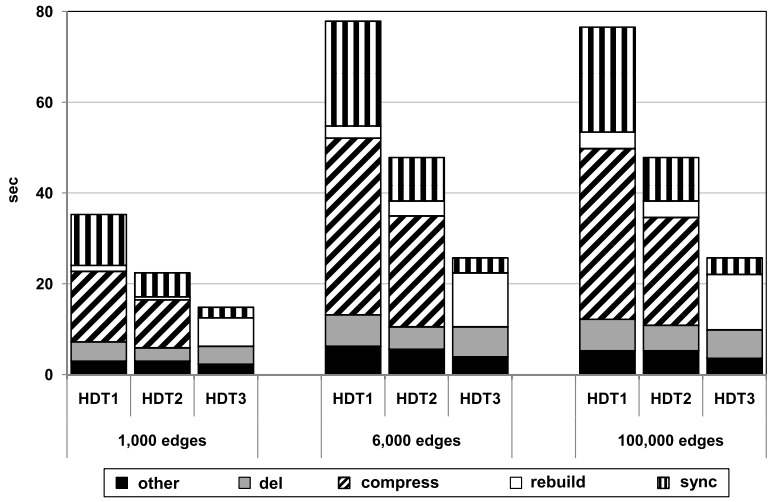


Fig. 9. HDT1, HDT2 and HDT3 running time details on random graphs with 1000 vertices. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 45% of the operations were tree edge deletions and 45% of the operations were tree edge insertions.

In order to further investigate the behavior of HDT on random graphs we analyzed in detail the execution time of the three HDT variants on some of the graphs used in this experiment: Random 1 ($m = n$), Random 2 ($m = 6n$) and Random 3 ($m = 100n$). It can be seen in Fig. 9 that HDT1 and HDT2 are able to rebuild deletions-only data structures (*rebuild* time) much faster than HDT3, thanks to the smaller number of tree edges implied by the path compression. However, such an advantage comes out at a significant cost as both algorithms spend the most part of their execution time in maintaining the local compressed trees aligned with respect to the global MST (*sync* time) and in contracting super paths into super edges (*compress* time). These costs are much smaller for HDT2 than for HDT1 perhaps due to the better efficiency of dynamic trees in this particular scenario. It should be noted also that HDT2 is indeed the algorithm capable of processing edge deletions (*del*time) more efficiently. Its deletions-only data structures contain fewer tree edges than HDT3 thanks to the compression while the cost it pays for accessing compressed trees is smaller than HDT1. The remaining execution time (*other* time), mostly due to the cost of processing edges insertions and handling internal data structures, is smaller for HDT3 because of the use of simpler data structures. As a result, it seems that for this experiment the advantage achieved in rebuilding the trees (*rebuild* time) does not pay off the overhead of using the compression technique (*compress* and *sync* time).

We now report the results of our experiments on k-clique graphs. The most challenging operation to support on this kind of graph seems to be the deletion of an inter-clique tree edge. Indeed in this case, the set of candidate replacement edges is very small with respect to the whole set of non-tree edges because a deleted inter-clique tree edge can only be replaced by another inter-clique edge. Despite this, HDT was able to process these updates faster than random tree updates on random graphs having the same number of vertices and edges. We observe that the deletion of an edge having a very small set of candidate replacement edges has two side-effects. On the one hand, it may require HDT to scan a large number of non-tree edges before finding a replacement. We expect this effect to be amortized by the edge decomposition strategy used by HDT. On the other hand, it will significantly reduce the average number of candidate replacement edges because, in this case, many deletions-only data structures will return no replacement edges at all. This will reduce accordingly the total number of super paths to be contracted when updating $\mathcal{A}$ (*compress* time).

This has been validated by our experiments on k-clique input sets presented in Fig. 10. We fixed the number of vertices to 1000 (same number of vertices as in the experiment presented in Fig. 9) while changing the number and the size of the cliques. Moreover, we generated a sequence of operations containing a high percentage of inter-clique tree updates so as to obtain the same number of tree updates enforced on random graphs in our previous experiment.

The comparison of these results with the ones obtained on random graphs having approximately the same size ($m =$ 100,000) and reported in the Fig. 9 shows a consistent performance gain of all the three HDT variants mostly coming from the smaller path compression cost.

We had similar *sync* times in both cases (Figs. 9 and 10) since the number of MST updates was approximately the same. We observe that, even in the case of Fig. 10, the cost of maintaining the compressed paths affects the overall performance of HDT1 and HDT2 making them slower than HDT3.

4.3.5. Level decomposition strategy

Iyer et al. argued in their experimental study on fully-dynamic connectivity algorithms [30] that the level decomposition strategy is not worth being used on sparse random input sets. Indeed, in this case typically only few edges have to be tested before finding a replacement. These considerations apply to the HDT dynamic MST algorithm as well, as it is essentially a specialization of the original connectivity algorithm with the main difference being that candidate replacement edges

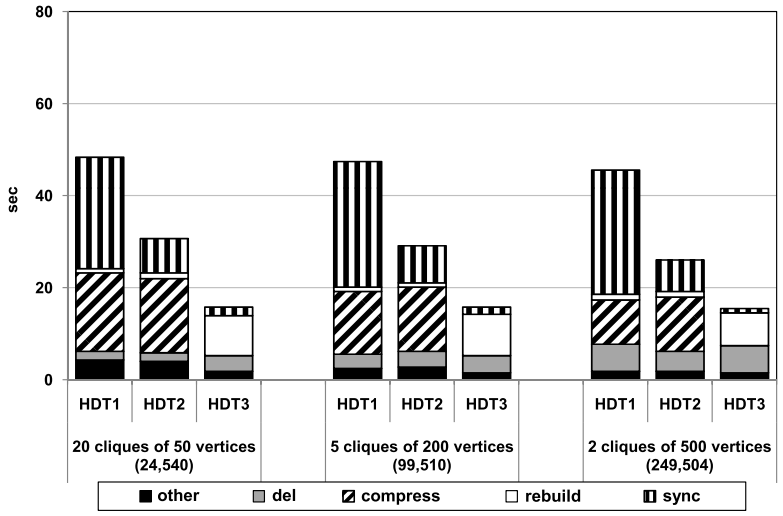


Fig. 10. HDT1, HDT2 and HDT3 running times details on k-clique graphs. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 90% of the operations were inter-clique edges tree updates.

are checked in a non-decreasing weight orders. The level decomposition strategy seems ineffective even when processing random tree edge deletions on random dense graphs. Also in such a case, the average number of non-tree edges to be checked before finding a replacement is very small. This affects the level decomposition strategy as there are very few non-tree edges promotions.

To explain this we recall the steps performed by HDT3 when processing the deletion of a random tree edge $e = (v, w)$ on an input random dense graph G. To begin with, the deletion of e will split, in the worst case, the original MST in two distinct tree components, T_v and T_w, having the same size. Suppose, without any loss of generality, that we search for a replacement among non-tree edges incident to T_v. The first step performed by HDT3 is to promote all level $\ell(e)$ tree edges in T_v to level $\ell(e) + 1$. Then, since candidate replacement edges are considered in a non-decreasing weight order, we are just interested in finding the first edge at level $\ell(e)$ and reconnecting again the two components T_v and T_w. Every non-tree edge unsuccessfully considered for a replacement is promoted to level $\ell(e) + 1$. When T_v and T_w have the same size, the number of non-tree edges crossing the cut defined by (v, w) will be likely to be approximately the same as the number of non-tree edges having both endpoints in T_v. Therefore, by considering candidate replacement edges among edges incident to T_v we will probably find a replacement in a few checks. So, the number of non-tree edges unsuccessfully considered for a replacement and then promoted will be very small. This implies that a long sequence of random tree edge deletions will force HDT3 to repeatedly promote almost all tree edges to the upper levels while most of non-tree edges will likely remain at level 0. In such a situation, the deletion of a tree edge (probably found at a level higher than 0) may require HDT3 to unsuccessfully traverse several levels before finding a replacement edge (likely to be found at level 0, if it exists).

We now consider k-clique graphs: we recall that these graphs are structured on two different levels with typically a large number of intra-clique edges and a small number of inter-clique edges. The level decomposition strategy would be useful here to distinguish among inter-clique edges and intra-clique edges; this would allow HDT3 to significantly restrict the set of candidate edges that are searched for replacements after a tree edge deletion. This is especially the case of the deletion of inter-clique tree edges. As we have already said, this is the hardest operation for HDT3 on k-clique graphs as the set of candidate replacement edges in this case is only a small fraction of all non-tree edges. As already seen in previous experiments, HDT3 should be able to efficiently process these operations because of the smaller path-compression cost they require: a deleted inter-clique edge has, on average, very few candidate replacement edges, thus implying a small overhead for rebuilding the deletions-only data structures.

We now analyze whether the level decomposition helps HDT3 in finding such replacements efficiently. Suppose we delete a tree edge $e = (v, w)$ assigned to level $\ell(e)$. Two cases are possible. If e was an inter-clique edge, then it can only be replaced by another inter-clique edge. However, since edges are considered in a non-decreasing weight order and inter-clique edges have larger weights than intra-clique edges, HDT3 will promote all intra-clique edges existing at level $\ell(e)$ and incident to T_v before considering any candidate replacement edge. Instead, if e were an intra-clique edge, then it would be replaced without promoting any inter-clique non-tree edges at all because, during the search for a replacement, they are considered after intra-clique edges. It is easy to see that, after processing a sequence of tree edge deletions, most of inter-clique non-tree edges will be likely placed at lower levels than intra-clique non-tree edges. In such a situation, HDT3 would likely be able to search a replacement for a deleted inter-clique tree edge without considering any intra-clique edge.

In summary, we expect that the level decomposition strategy used by HDT3 becomes effective especially on structured input sets. To experimentally assess its merits we also implemented a variant of HDT3 named HDT3_NOLEVEL. This variant does not use level decomposition at all; instead, when a tree edge is deleted, it looks for a replacement edge among all the

Table 1
Statistics describing a level usage of HDT on several different random and k-clique graphs. The *scan* value denotes the average number of edges to be checked before finding a replacement, when it exists. The *scanFail* value denotes the average number of edges to be checked when no replacement edge exists.

	Random 1 (Random graph $n = 1000, m = 1000$ 90% tree updates)	Random 2 (Random graph $n = 1000, m = 6000$ 90% tree updates)	Random 3 (Random graph $n = 1000, m = 100{,}000$ 90% tree updates)	Clique1 (K-clique graph $k = 5, c = 200$ 90% inter-clique tree updates)	Clique2 (K-clique graph $k = 5, c = 200$ 10% inter-clique tree updates)
scan	1.04	1.02	1.02	9.4	89.1
scanFail	0.01	0.01	0.01	3.7	16.8
Levels where edges were deleted (*searchStartLevel*)					
Level 0	19.09%	21.74%	21.41%	87.86%	70.97%
Level 1	36.41%	39.56%	39.82%	11.91%	17.45%
Level 2	31.88%	29.07%	29.12%	0.20%	10.67%
Level 3	10.86%	8.22%	8.56%	0.02%	0.89%
Level 4	1.67%	1.29%	1.06%	0%	0.02%
Level 5	0.09%	0.11%	0.03%	0%	0%
Level 6	0%	0.01%	0%	0%	0%
Levels where replacement edges were found (*foundLevel*)					
Level 0	98.70%	99.25%	99.35%	96.54%	80.19%
Level 1	1.17%	0.63%	0.56%	3.43%	15.11%
Level 2	0.13%	0.11%	0.07%	0.04%	4.70%
Level 3	0%	0.01%	0.02%	0%	0.14%
Level 4	0%	0.01%	0%	0%	0%

edges outgoing from the smallest component resulting from the deletion. We compared these two implementations on both structured and non-structured input sets.

4.3.5.1. Experiments with `HDT3` and `HDT3_NOLEVEL`. We have considered in our tests with the level decomposition strategy some of the input sets analyzed in the previous experiments with random graphs: Random 1 ($n = 1000, m = 1000$), Random 2 ($n = 1000, m = 6000$) and Random 3 ($n = 1000, m = 100{,}000$). For all these inputs we used a random sequence of 20,000 operations containing 45% of tree edge deletions in order to enforce many level promotions. Moreover, we considered also two additional input sets for experimenting with k-clique graphs: Clique1 ($k = 5, c = 200$, 90% inter-clique tree updates) and Clique2 ($k = 5, c = 200$, 10% inter-clique tree updates).

As presented in Table 1, we evaluated the efficiency of HDT3 in processing tree edge deletions by measuring the average number of edges to be checked in all the deletions-only data structures either before finding a replacement (*scan*) or before concluding that such a replacement does not exists (*scanFail*). These values can also be used to estimate the average number of non-tree edge promotions. Our experimental results show that both on random sparse graphs and dense graphs, HDT3 is able to replace deleted tree edges or to conclude that no replacement edge exists with a very small number of checks (*scan* and *scanFail*), thus performing few level promotions. The average number of candidate replacement edges is even smaller when the replacement edge does not exist (*scanFail*). In this case, the algorithm unsuccessfully searches a replacement edge in the smaller component resulting from the deletion of tree edge. This component will likely have very few incident non-tree edges both in the sparse case (because of the very small number of non-tree edges existing in the graph) and in the dense case (in this case, a component being disconnected by a random edge deletion which cannot be reconnected by a replacement edge will likely be very small).

The low number of checks both in the *scan* and in the *scanFail* cases reveals that, on these inputs sets, HDT3 will mostly promote tree edges while non-tree edges will likely stay at level 0. In order to investigate this behavior we measured the effectiveness of level decomposition by keeping a count of levels where tree edges were deleted (*searchStartLevel*) and of levels where replacement edges were found (*foundLevel*). As expected, the results show that the search for a replacement in our random input sets frequently starts at the upper levels (e.g., approximately 80% of the tree edges being deleted were located in a level higher than 0) while a replacement is likely to be found at level 0 (e.g., about 98% of the replacement edges). Therefore, in this case, the search for a replacement edge at the upper levels does not turn out to be very useful.

These results suggest that it is not worthwhile to use the level decomposition strategy on random inputs. Following these considerations, we tried HDT3_NOLEVEL. As expected, the performance gain on random input sets is evident. This is illustrated in Fig. 11, where we report the result of an experiment on a random graph with 1000 vertices and an increasing number of edges. The sequence of operations contained 90% of tree updates.

We now turn to the experimental analysis of the HDT3 level decomposition strategy when applied to k-clique input sets. HDT3 is able to take advantage of this strategy especially when processing a sequence of updates with a high percentage of inter-clique updates. In this case, the initial inter-clique tree deletions will force the deletions-only data structures to promote all intra-clique edges they contain. Then, the following inter-clique tree deletions will be processed more quickly as the search for a replacement will likely not take into account intra-clique non-tree edges. This phenomenon appears every time a deletions-only data structure is emptied and then rebuilt.

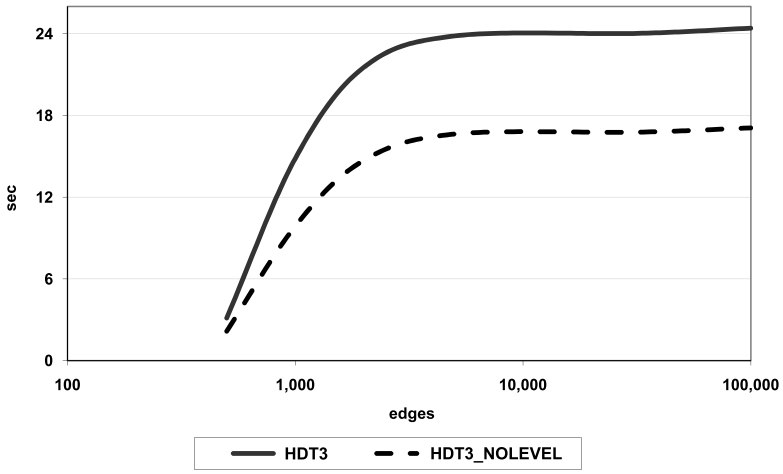


Fig. 11. HDT3 running time with and without level decomposition on a random graph with an increasing number of edges plotted using a logarithmic scale. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 45% of the operations were tree edge deletions and 45% of the operations were tree edge insertions.

This is revealed by our experiments on k-clique graphs with 1000 vertices decomposed in 5 cliques illustrated in Table 1. In the first of these experiments, Clique1, we used a sequence of operations with a high percentage of inter-clique tree updates. The average number of non-tree edges considered when searching for one replacement (*scan* and *scanFail*) is higher than in the case of random graphs because of the periodical mass promotion of intra-clique non-tree edges. Moreover, since inter-clique edges are few and are deleted and reinserted very often, they will likely stay at level 0. This implies that, for the most part, the deleted edges and their replacements are located at level 0. Indeed, about 87% of the searches start at level 0 (*searchStartLevel*), compared to 20% for the random case, and approximately 96% of the replacement edge is found at level 0 (*foundLevel*). Therefore the overhead due to unsuccessfully searches for replacement edges at the upper levels is smaller than in the random case.

If we increase the percentage of intra-clique updates, such as in Clique2, the set of candidate replacement edges for an edge deletion becomes much larger. As a consequence, HDT3 needs to check a much higher number of non-tree edges (*scan* and *scanFail*) before finding a replacement, if one exists. This will increase the number of promotions and, consequently, the number of replacement edges found at a level higher than 0 (about 20% of the replacement edges). If we turn off the level decomposition then we could expect that the cost of processing a tree update on a k-clique input becomes much higher. This is especially true for inter-clique tree deletions as, in this case, every deletions-only data structure is forced to scan the whole set of intra-clique non-tree edges it contains before considering any candidate replacement edge. This is illustrated in Fig. 12, where we compare the performance of HDT3 with and without level decomposition on Clique1 and Clique2.

4.3.6. Batching

One more performance bottleneck for HDT is the reinitialization of deletions-only data structures. As already described in Section 4.3.2, these data structures form a hierarchy of subgraphs of G, with each deletions-only data structure A_i containing at most 2^i edges. The reinitialization of a deletions-only data structure A_i consists of two steps. First, it removes all edges in the deletions-only data structures A_j, with $j < i$. Second, these edges are used to rebuild A_i together with the edges forming the current global MST and a provided set of non-tree edges D. It is a very frequent operation as it is invoked either when the deletion of an edge returns at least one replacement or when a new edge is inserted (except when it reconnects two distinct MST components). The resulting overhead turns out to be especially high due to the frequent reinitializations of the smaller deletions-only data structures.

In order to minimize this cost we applied the batching technique described by Henzinger and King in [28]. The idea is to delay as much as possible update operations using an edge list B as a temporary buffer. To this end, we redefine properly the HDT *Delete* and *Insert* operations. In the first case, every time an edge $e = (v, w)$ is deleted, the set of candidate replacement edges returned by the deletions-only data structures will be added to B. Then, if e was in the global MST, the replacement edge will be searched among all the edges stored in B; otherwise, the operation ends. In the second case, when we insert an edge $e = (v, w)$, we check if v and w are connected in F. If not, we just add e to F; otherwise, we compare the weight of e with the heaviest edge f on the path from v to w in F. If e is heavier, we add e to B; otherwise, we replace f with e and we add f to B. In both cases, all the modifications to the global MST will be annotated and then committed to the local spanning tree F_i when the deletions-only data structure A_i is rebuilt. Finally, after performing either insertions or deletions we check for the size of B and, if the threshold s is reached, we rebuild $\mathcal{A}$ using the non-tree edges contained in B.

We expect many advantages from this technique. The first is that it allows us to save many rebuildings of deletions-only data structures. In addition, the total number of deletions-only data structures to be maintained and queried for replacement

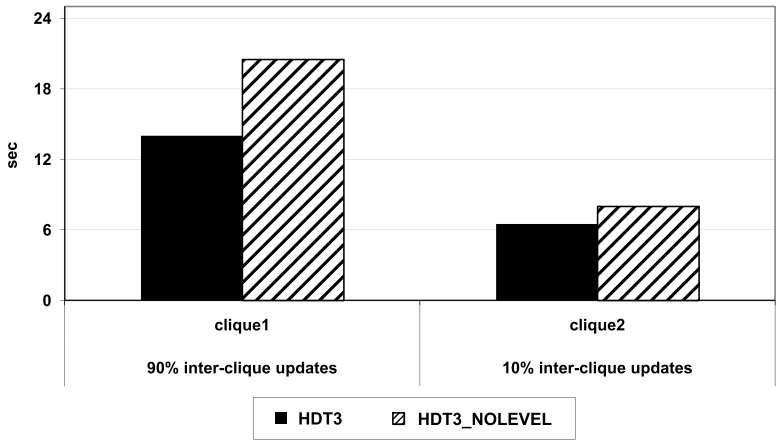


Fig. 12. HDT3 time required to perform update operations with and without level decomposition on an input graph made of 5 cliques each having 200 vertices. Update sequences contained 10,000 edge insertions and 10,000 edge deletions and use two different percentage of inter-clique updates. Clique1 uses a 90% of inter-clique updates and Clique2 uses a 10% of inter-clique updates.

edges are reduced since data structures smaller than s will never be used. Another important advantage is that a non-tree edge inserted in B (either by an *Insert* operation or as a candidate replacement edge) could be deleted by a subsequent *Delete* operation before the batch is actually being processed and emptied. This will in turn delay the filling of B while reducing the overhead involved with the deletions-only data structures reinitializations.

An important point in the experimentation of the batching technique is the tuning of the threshold s (i.e., the size of the edge list B). Indeed, this value should grow with the number of edges in the input graph as the number of deletions-only data structures used by HDT3 (and, consequently, the maximum number of non-tree edges that could be returned by an edge deletion) increases with the input graph density. On the one hand, a very high value would drastically reduce the frequency of reinitializations of $\mathcal{A}$. On the other hand, it will slow down deletions as these operations search for replacements by checking every non-tree edge stored in B. We tried several different threshold values for s: here we report only on $s = \sqrt[3]{m}$ (HDT3_S3), $s = \sqrt[2]{m}$ (HDT3_S2), $s = \sqrt[3]{m^2}$ (HDT3_S32).

4.3.6.1. Experiments with HDT3, HDT3_S2, HDT3_S3 and HDT3_S32. Our experiments with HDT3 indicated that a good batch size for random sparse graphs seems to be $\sqrt[3]{m^2}$ (S32). In this case, smaller thresholds such as $\sqrt[2]{m}$ (S2) and $\sqrt[3]{m}$ (S3) are less effective because the buffer B gets quickly filled up forcing a high number of reinitializations.

However, the situation is quite different if we increase the density of the input graph: in this case the buffer B used by HDT3_S32 ($s = \sqrt[3]{m^2}$) easily becomes very large. As it can be observed in Fig. 13, the HDT3_S32_NOLEVEL execution time grows with the graph density and thus seems dominated by the cost required to scan non-tree edges in B while processing tree edge deletions. On the contrary, the slower growth of the buffer B size for both S2 and S3 allows them to achieve a better trade-off between the overhead due to the batching and the resulting performance gain.

As expected, the batching technique seems more effective when the algorithm has to process a long sequence of operations using a small set of candidate edges. This can be seen on the experiment on k-clique input sets with a high percentage of inter-clique edge updates shown in Fig. 14. In such a case we have only a small set of inter-clique edges that are inserted and then deleted very often. As a consequence, the buffer B will be filled mostly by edges returned by intra-clique updates. Thus, the number of reinitializations is much smaller than in the random case. However, the large number of non-tree edges in k-clique inputs implies very large buffer sizes thus strongly influencing the time required to scan for replacements in B. This is clearly visible in the HDT3_S2 and the HDT3_S32 cases. This overhead is not dominant in the case of HDT3_S3 because of the smaller batch size.

4.3.7. Concluding remarks

In all the experiments we conducted on the HDT algorithm, the variant using no path-compression revealed to be the fastest. In particular, HDT3_S3_NOLEVEL was the fastest on non-structured inputs, while HDT3_S3 was the fastest on structured inputs. We will use these as reference implementations in the rest of the paper.

5. Overall experimental results

5.1. Random inputs

Not surprisingly, in our experiments with random graphs subject to random updates, ST_CACHE was faster than both HDT3_S3_NOLEVEL and Spars. This is mainly due to the fact that ST_CACHE is a very simple algorithm and performs very

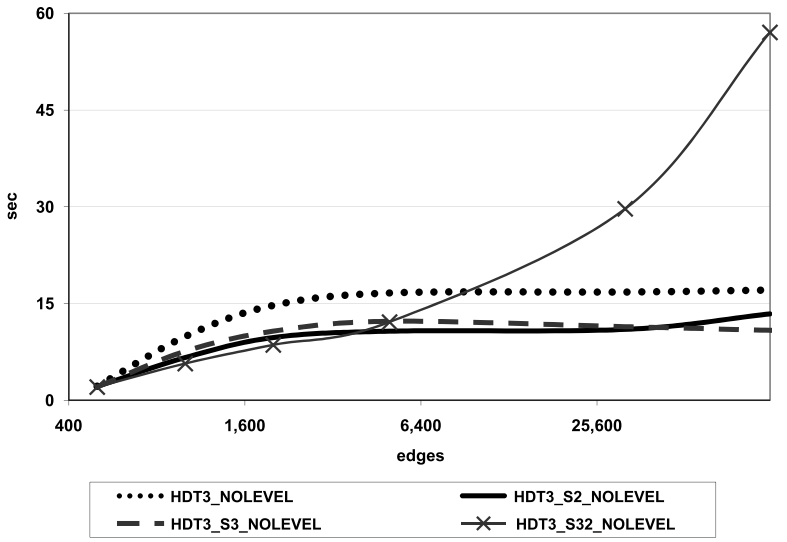


Fig. 13. HDT3 running times on random graphs compared with its variants using the batching technique: S2 ($s = \sqrt[2]{m}$), S3($s = \sqrt[3]{m}$), S32($s = \sqrt[3]{m^2}$). The horizontal axis uses a logarithmic scale. Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 90% of the operations were tree updates.

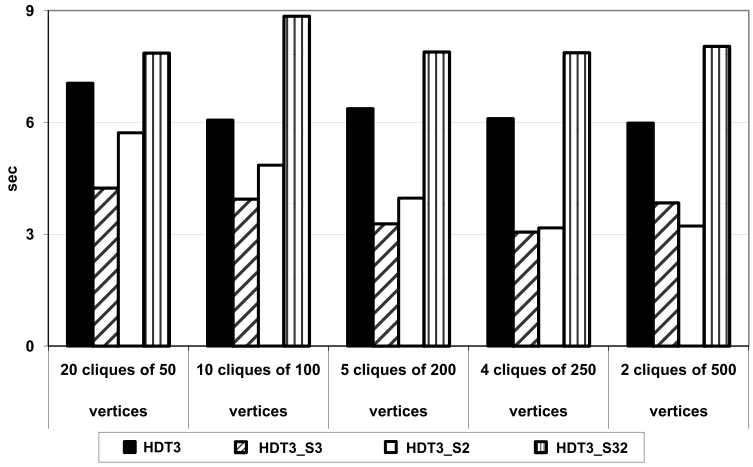


Fig. 14. HDT3 running times on k-clique graphs compared with its variants using the batching technique: S2 ($s = \sqrt[2]{m}$), S3($s = \sqrt[3]{m}$), S32($s = \sqrt[3]{m^2}$). Update sequences contained 10,000 edge insertions and 10,000 edge deletions: 90% of the operations were inter-clique edge updates.

little work, except in the case of tree edge deletions: in this case, the algorithm might scan many non-tree edges in the quest for a replacement. In random sequences of updates, however, tree edge deletions do not happen very frequently, especially for dense graphs. Note that, as opposed to ST_CACHE, both Spars and HDT3_S3_NOLEVEL perform some substantial work even in the case of updates that do not affect the current MST: Spars might have to update the certificates in a sparsification tree path (even though the update does not necessarily reaches the sparsification tree root), while HDT3_S3_NOLEVEL might have to rebuild some deletions-only data structures.

Fig. 15 illustrates the results of an experiment on random graphs with 3000 vertices and different densities. As can be seen from the figure, ST_CACHE was always faster than both Spars and HDT3_S3_NOLEVEL except for two cases: sparse graphs, where Spars delivered the best performance, and very sparse graphs, where HDT3_S3_NOLEVEL was slightly faster than ST_CACHE. This can be explained by recalling some observations from Section 3.1.1, as follows. We recall that sparse graphs (and in particular graphs with around $m = 2n$ edges) are the worst possible input for ST_CACHE during a sequence of random updates. In this case, a random edge deletion is likely to be a tree edge deletion, which can make ST_CACHE slower than other more sophisticated algorithms, such as Spars. We now turn to very sparse graphs. As already illustrated in Fig. 1, in this case the caching policy of ST_CACHE does not appear to be very effective: when the number of non-tree edges (and thus potential candidates for edge replacements) is very small, caching findroot operations seems to introduce more overhead than savings. This makes HDT3_S3_NOLEVEL more competitive for very sparse graphs.

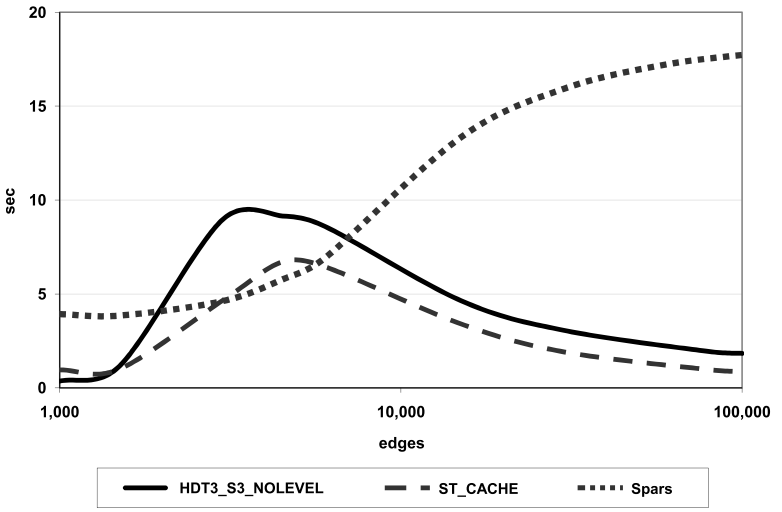


Fig. 15. Experiments on random graphs with 3000 vertices and different densities, plotted using on the horizontal axis a logarithmic scale. Update sequences contained 10,000 random insertions and 10,000 random deletions.

We also experimented with the operations by changing the sequence mix: to force more work on to the simpler algorithms, we increased the percentage of tree updates. We expected HDT3_S3_NOLEVEL to be able to find replacements for tree updates much faster than ST_CACHE. But, such an advantage can be significantly reduced by the cost required to maintain and to rebuild deletions-only data structures. This is clearly visible in Fig. 16: ST_CACHE is still the fastest algorithm thanks to its ability to quickly process random inputs. However, we observe that in this case its performance is very close to HDT3_S3_NOLEVEL. We remark here that all the experiments presented so far used a fixed number of vertices (i.e., 3000). This size has been chosen in such a way to permit us to investigate the behavior of the algorithms on very dense graphs without exceeding our available physical memory.

We also conducted additional experiments on larger graphs. As one could expect, we noticed that when the graph size increases, more sophisticated algorithms, such as HDT3_S3_NOLEVEL, are likely to become substantially faster than simpler algorithms, such as ST_CACHE. Fig. 17 illustrates the results of such an experiment on random graphs with vertices ranging from 1000 to 8000, and a proportional number of edges (i.e., $m = 50n$).

5.2. *k*-Clique inputs

We recall that on k-clique inputs we can choose whether to perform inter-clique or intra-clique edges update operations. By properly selecting the mix between these two options, we are able to create a wider range of situations going from highly structured inputs (mostly inter-clique updates) to non-structured random inputs (mostly intra-clique updates).

In our experiments on k-clique inputs, illustrated in Fig. 18, we considered several kinds of update sequences, depending on whether the majority of update sequences were related to inter-clique edges or to intra-clique edges. For all cases we have used graphs decomposed in 50 cliques each having 20 vertices. We were not surprised to discover that, when the majority of updates were inter-clique tree updates, HDT3_S3 outperformed Spars. As already observed in Section 4.3.6, in such a setting HDT3_S3 is able to process inter-clique edge updates by just inserting and deleting edges in the buffer B while performing a very small number of deletions-only data structures reinitializations.

When updates involve more intra-clique edges, the number of modifications to the MST tend to decrease since random intra-clique operations will not likely be tree updates. Thus we expect that the speed-up coming from the batching technique will be greatly reduced. This holds because the number of intra-clique edges randomly inserted and then deleted between two different reinitializations will be much smaller than when processing mostly inter-clique updates, thus increasing the frequency of the reinitializations and affecting the performance of HDT3_S3. On the other hand, we expect that the increase in the percentage of intra-clique updates has a positive effect on Spars: in such a case, updates are not likely to change the MST and thus the change will not be propagated all the way up in the sparsification tree. Finally, when the majority of updates involve intra-clique edges both HDT3_S3 and Spars achieve their best performance as there almost no tree updates to carry out.

In all the experiments that we have conducted on these input sets, the simpler algorithms, such as ST_CACHE, were not competitive at all. This is because they handle tree edge deletions by exhaustively scanning of all the existing non-tree edges searching for a replacement edge.

5.3. Iyer et al. Worst-case inputs

On the worst-case inputs introduced by Iyer et al. [30], ST achieves its best case of $O(\log n)$ time, as there are no non-tree edges to consider. Fig. 19 illustrates the results of these tests on graphs with up to 32,768 vertices. The variant of

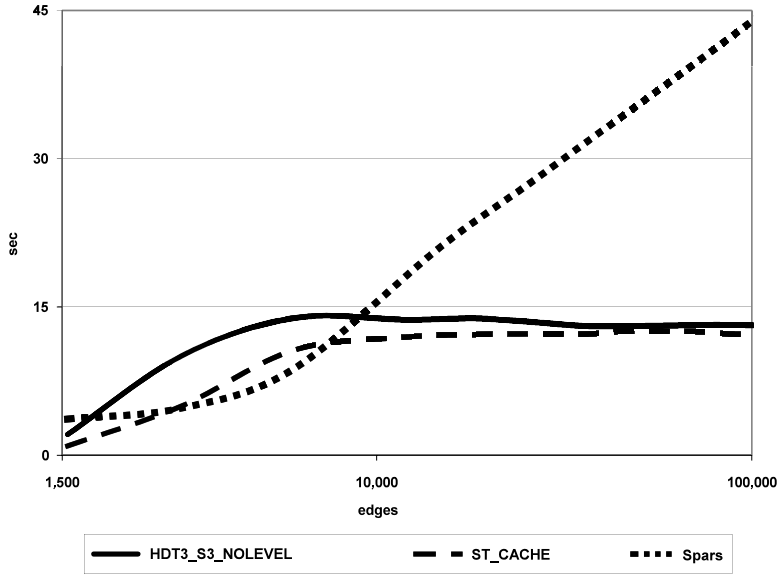


Fig. 16. Experiments on random graphs with 3000 vertices and different densities plotted using on the horizontal axis a logarithmic scale. Update sequences contained 10,000 insertions and 10,000 deletions: 45% of the operations were tree edge deletions and 45% of the operations were tree edge insertions.

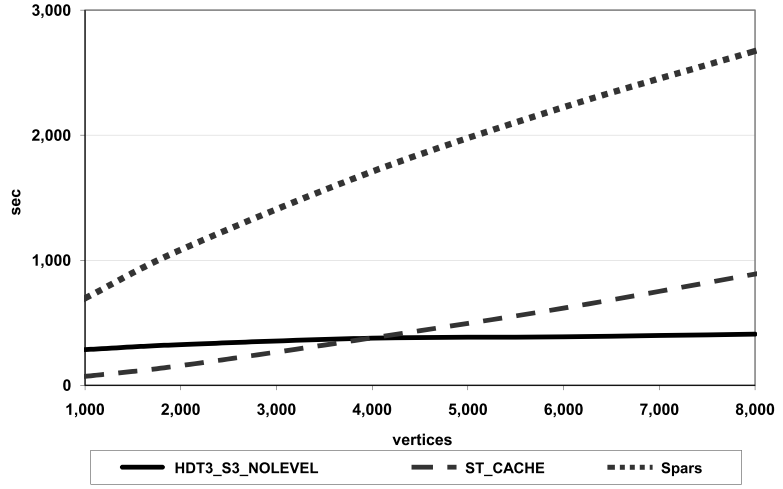


Fig. 17. Experiments on random graphs with an increasing number of vertices and a proportional number of edges ($m = 50n$). Update sequences contained 200,000 insertions and 200,000 deletions: 45% of the operations were tree edge deletions and 45% of the operations were tree edge insertions.

ST caching findroots is much slower than ST because of the cost to be paid for reinitializing the cache at any update: for this reason we did not include it in our results. As expected, `HDT3_S3` is tricked by the update sequence and has to perform a lot of unnecessary promotions of edges among levels of the data structures. This does not happen in the variant of `HDT3_S3_NOLEVEL` since it uses no level decomposition at all. Finally, `Spars` is hit pretty badly, as its performance is up to fifty times slower than `HDT3_S3_NOLEVEL`: indeed each tree edge deletion suffers from the overhead of keeping the underlying implementation of Frederickson's light partition, that has order $\lceil m^{2/3} \rceil$.

5.4. Real-world graphs

In our experimentation with the graph of the Internet Autonomous Systems, we considered a sequence of snapshots taken at regular intervals of two hours in the range from the beginning of September 2003 to the end of September 2003. This gave rise to a graph having 16,287 vertices and 31,451 edges. The overall number of edge insertions and edge deletions was 26,357 and 25,595, respectively. Due to the small density of the input graph, a large number of these updates was tree-updates. In this scenario, we expect `HDT3_S3_NOLEVEL` to be fast, since it searches for a replacement only among the edges that are incident to the smallest tree component resulting from the deletion of a tree edge. Since the input graph is not very

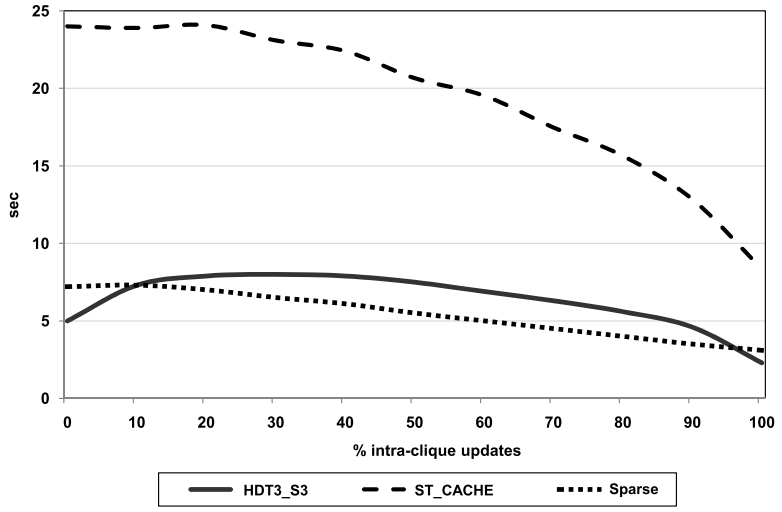
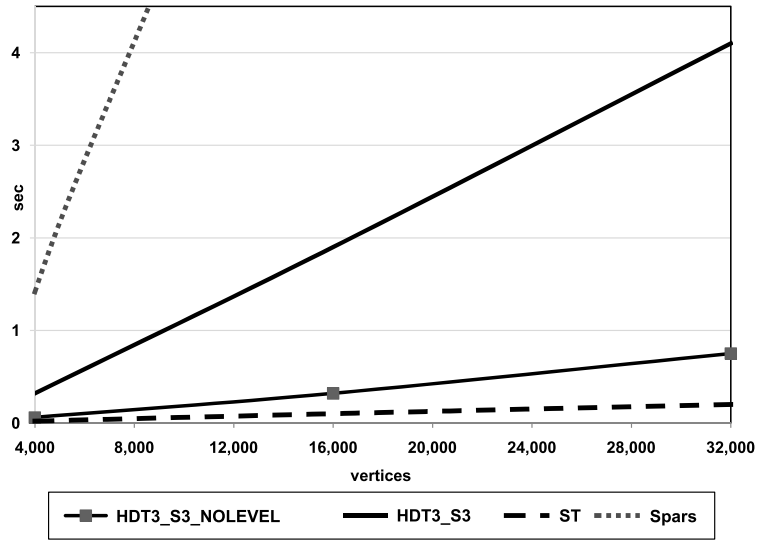


Fig. 18. Experiments on a graph decomposed in 50 cliques each having 20 vertices. Operations contained an increasing number of intra-clique updates.



Vertices	HDT3_S3_NOLEVEL	ST	HDT3_S3	Spars
4,096	0.06	0.02	0.32	1.4
16,384	0.32	0.1	1.9	11.22
32,768	0.75	0.2	4.1	42.88

Fig. 19. Experiments and execution times on Iyer et al. Worst-case inputs on graphs with different number of vertices.

dense, this will result in a small number of candidate replacement edges to check. On the contrary, we expect ST_CACHE to be not particularly efficient on this input as every tree edge deletion forces this algorithm to search for replacements in the whole set of non-tree edges. As we have already reported in our previous experiments with random inputs, even if the input graph is not very dense, when the overall number of edges to be considered is large then the execution time of ST_CACHE is dominated by the cost required for scanning non-tree edges (while looking for a replacement). This has been confirmed by our results on the graph of the Internet Autonomous Systems shown in Table 2.

6. Conclusions

In this paper we have presented the results of our study on the performance of several algorithms for maintaining minimum spanning trees in dynamic graphs. In particular, we have implemented and tested the algorithm by Holm et al. [29]

Table 2
Execution time of ST, HDT3_S3_NOLEVEL and Spars when processing updates occurring in the graph of the Internet Autonomous Systems in the month of September 2003.

Algorithm	Update time
HDT3_S3_NOLEVEL	11.6
Spars	39.6
ST_CACHE	90.2

and by Eppstein et al. [18] together with several simpler algorithms. Our experiments were aimed at characterizing the algorithms' behavior when applied to different kinds of input sets. To this end, we conducted our study on randomly generated graphs, real-world graphs, and on more structured (non–random) graphs and update sequences, which tried to enforce bad sequences on the algorithms.

The two simple algorithms that we developed were obtained as a derivation of the static algorithm by Kruskal (see e.g., [11,33]). They were competitive in all situations where edge deletions are not likely to change the MST, such as for random inputs, and in cases where deleted tree edges have a small set of candidate replacement edges. In such situations, these algorithms have very little work to do and so they perform very efficiently.

As far as the algorithm by Holm et al. [29] is concerned, our experimental study showed that a "straight-from-the-paper" implementation of this algorithm was not very efficient, in large part due to the technique used to maintain compressed tree paths on deletions-only data structures. We engineered this algorithm in many ways: path compression, level decomposition and a batching technique. We implemented, as a first variant, a different path compression technique by Henzinger and King [27] using dynamic trees in place of top-trees. This variant performed substantially better than the original algorithm. Another alternative that we tried did not use path compression at all, while adopting, instead, the rollback technique by Henzinger and King [28] in order to perform deletions-only data structures reinitializations efficiently. This variant was the fastest in practice on many inputs. Moreover, we have found that level decomposition is not very effective on random input sets. Here, the HDT deletions-only data structures are able to process tree edge deletions very efficiently without exploiting the level decomposition strategy. On these input sets, a variant of the algorithm by Holm et al. [29] using no level decomposition exhibited better performance than its original counterpart.

The original algorithm by Holm et al. [29] is able to exploit the level decomposition strategy on more structured input sets. However, the resulting speed-up in searching for replacement edges is mostly canceled out by the cost of promoting edges. Finally, we experimented with the batching technique introduced by Henzinger and King [28] in order to decrease the frequency and the cost of deletions-only data structures reinitializations. The outcoming performance gain is significant, especially on input sets with a long sequence of operations involving only a small set of edges.

In summary, our experimental study shows that on completely random input sets, simple algorithms are the ones able to achieve the best performance thanks to their ability to process non-tree updates very fast. The complex algorithms suffer the cost of being paid for maintaining their internal data structures updated even when processing non-tree updates. Nevertheless, when the number of modifications in the MST on a random graph increases, the simpler algorithms are outperformed by the variant of the algorithm by Holm et al. [29] using no level decomposition and the batching technique. In particular, this algorithm was also the fastest when processing the updates on the network graph of the Internet Autonomous Systems. This network is a sparse graph with around $2n$ edges. In such a setting, the frequency of MST updates is very high, so that it strongly affects the performance of simpler algorithms.

In contrast, the algorithm by Holm et al. [29] performs badly on the worst-case inputs introduced by Iyer et al. [30]. The fastest algorithms for this kind of input sets are the simpler ones since, in this setting, the set of non-tree edges is always empty and thus the search for a replacement edge following a tree edge deletion takes very little time.

Finally, on the more structured input sets that we tried, k-clique inputs, the fastest algorithm was Spars when using update operations involving both inter-clique and intra-clique edges. On the other hand, the algorithm by Holm et al. [29] using level decomposition was the best when update operations included a large majority of just inter-clique updates or of just intra-clique updates.

Acknowledgements

We are indebted to the anonymous referees for their useful and valuable comments.

References

[1] G.M. Adelson-Veslkii, Y.M. Landis, An algorithm for the organization of information, Doklady Akademi Nauk 146 (1962) 263–266.

[2] D. Alberts, G. Cattaneo, G.F. Italiano, An empirical study of dynamic graph algorithms, ACM Journal on Experimental Algorithmics 2 (5) (1997).

[3] D. Alberts, G. Cattaneo, G.F. Italiano, U. Nanni, C.D. Zaroliagis, A software library of dynamic graph algorithms, in: R. Battiti, A. Bertossi (Eds.), Proc. Algorithms and Experiments, ALEX 98, Trento, Italy, February 1998, pp. 129–136.

[4] D. Alberts, M.R. Henzinger, Average case analysis of dynamic graph algorithms, Algorithmica 20 (1998) 31–60.
[5] S. Alstrup, J. Holm, K. de Lichtenberg, M. Thorup, Maintaining information in fully dynamic trees with top trees, ACM Transactions on Algorithms 1 (2) (2005) 243–264.
[6] G. Amato, G. Cattaneo, G.F. Italiano, Experimental analysis of dynamic minimum spanning tree algorithms, in: Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 97, 1997, pp. 314–323.
[7] C.R. Aragon, R. Seidel, Randomized search trees, Algorithmica 16 (1996) 464–497.
[8] B. Bollobás, Random Graphs, 2nd ed., Cambridge University Press, 2001.
[9] L.S. Buriol, M.G.C. Resende, M. Thorup, Speeding up dynamic shortest path algorithms, INFORMS Journal on Computing 20 (2) (2008) 191–204.
[10] G. Cattaneo, P. Faruolo, U. Ferraro Petrillo, G.F. Italiano, Maintaining dynamic minimum spanning trees: An experimental study, in: David M. Mount, Clifford Stein (Eds.), Proc. 4th Workshop on Algorithm Engineering and Experiments, ALENEX 02, in: Lecture Notes in Computer Science, vol. 2409, Springer-Verlag, 2002, pp. 111–125.
[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd ed., MIT Press, 2001.
[12] C. Demetrescu, G.F. Italiano, Experimental analysis of dynamic all-pairs shortest path algorithms, ACM Transactions on Algorithms 2 (4) (2006) 578–601.
[13] C. Demetrescu, G.F. Italiano, Fully dynamic all pairs shortest paths with real edge weights, Journal of Computer and System Sciences 75 (5) (2006) 813–837.
[14] C. Demetrescu, P. Faruolo, G.F. Italiano, M. Thorup, Does path cleaning help in dynamic all-pairs shortest paths? in: Yossi Azar, Thomas Erlebach (Eds.), Proc. 14th Annual European Symposium, ESA06, in: Lecture Notes in Computer Science, vol. 4168, Springer-Verlag, 2006, pp. 732–743.
[15] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, U. Nanni, Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study, in: Stefan Näer, Dorothea Wagner (Eds.), Proc. 3rd Workshop on Algorithm Engineering, WAE 2000, in: Lecture Notes in Computer Science, vol. 1982, Springer-Verlag, 2001, pp. 218–229.
[16] C. Demetrescu, G.F. Italiano, Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier, in: Proc. of the 41st IEEE Annual Symposium on Foundations of Computer Science, FOCS'00, 2000, pp. 381–389.
[17] C. Demetrescu, G.F. Italiano, A new approach to dynamic all pairs shortest paths, Journal of the ACM 51 (6) (2004) 968–992.
[18] D. Eppstein, Z. Galil, G.F. Italiano, A. Nissenzweig, Sparsification—A technique for speeding up dynamic graph algorithms, Journal of ACM 44 (5) (1997) 669–696.
[19] D. Eppstein, Z. Galil, G.F. Italiano, T.H. Spencer, Separator based sparsification. I. Planary testing and minimum spanning trees, Journal of Computer and System Sciences 52 (1) (1996) 3–27.
[20] D. Eppstein, Z. Galil, G.F. Italiano, T.H. Spencer, Separator-based sparsification II: Edge and vertex connectivity, SIAM Journal of Computing 28 (1) (1998) 341–381.
[21] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, M. Yung, Maintenance of a minimum spanning forest in a dynamic plane graph, Journal of Algorithms 13 (1992) 33–54.
[22] G.N. Frederickson, Data structures for on-line updating of minimum spanning trees, with applications, SIAM Journal of Computing 14 (1985) 781–798.
[23] G.N. Frederickson, Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees, SIAM Journal of Computing 26 (2) (1997) 484–538.
[24] D. Frigioni, M. Ioffreda, U. Nanni, G. Pasqualone, Experimental analysis of dynamic algorithms for the single source shortest path problem, ACM Journal on Experimental Algorithmics 3 (5) (1998).
[25] D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Schaefer, C. Zaroliagis, An experimental study of dynamic algorithms for directed graphs, in: Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, Geppino Pucci (Eds.), Proc. 6th European Symp. on Algorithms, in: Lecture Notes in Computer Science, vol. 1461, Springer-Verlag, 1998, pp. 368–380.
[26] M.R. Henzinger, V. King, Fully dynamic biconnectivity and transitive closure, in: Proc. 36th IEEE Symp. Foundations of Computer Science, 1995, pp. 664–672.
[27] M.R. Henzinger, V. King, Randomized fully dynamic graph algorithms with polylogarithmic time per operation, Journal of the ACM 46 (4) (1999) 502–516.
[28] M.R. Henzinger, V. King, Maintaining minimum spanning forests in dynamic graphs, SIAM Journal on Computing 31 (2) (2001) 364–374.
[29] J. Holm, K. de Lichtenberg, M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity, Journal of the ACM 48 (4) (2001) 723–760.
[30] R. Iyer, D.R. Karger, H.S. Rahul, M. Thorup, An experimental study of poly-logarithmic fully-dynamic connectivity algorithms, ACM Journal on Experimental Algorithmics 6 (2001).
[31] V. King, Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs, in: Proc. 40th IEEE Symposium on Foundations of Computer Science, FOCS'99, 1999, pp. 81–91.
[32] V. King, G. Sagert, A fully dynamic algorithm for maintaining the transitive closure, Journal of Computer and System Sciences 65 (1) (2002) 150–167.
[33] J.B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, Proceedings of the American Mathematical Society 7 (1956) 48–50.
[34] K. Melhorn, S. Näher, LEDA: A platform for combinatorial and geometric computing, Cambridge University Press, February 2000.
[35] C.C. Ribeiro, R.F. Toso, Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes on edge weights, in: Proc. 6th International Workshop on Experimental Algorithms, WEA'07, June 2007, pp. 393–405.
[36] Route views project, University of Oregon. http://www.routeviews.org.
[37] D.D. Sleator, R.E. Tarjan, A data structure for dynamic trees, Journal of Computer and System Sciences 26 (3) (1983) 362–391.
[38] D.D. Sleator, R.E. Tarjan, Self-adjusting binary search trees, Journal of the ACM 32 (3) (1985) 652–686.
[39] R.E. Tarjan, R.F.F. Werneck, Dynamic trees in practice, in: Proc. 6th International Workshop on Experimental Algorithms, WEA'07, June 2007, pp. 80–93.