Maintaining Dynamic Minimum Spanning Trees: An Experimental Study*

Giuseppe Cattaneo[1], Pompeo Faruolo[1], Umberto Ferraro Petrillo[1], and Giuseppe F. Italiano[2]

[1] Dipartimento di Informatica e Applicazioni, Università di Salerno, Salerno, Italy
{cattaneo,umbfer,pomfar}@dia.unisa.it
[2] Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Roma, Italy
italiano@info.uniroma2.it
http://www.info.uniroma2.it/~Italiano

Abstract. We report our findings on an extensive empirical study on several algorithms for maintaining minimum spanning trees in dynamic graphs. In particular, we have implemented and tested a variant of the polylogarithmic algorithm by Holm *et al.*, sparsification on top of Frederickson's algorithm, and compared them to 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.

1 Introduction

In this paper we consider *fully dynamic graph algorithms*, namely algorithms that maintain a certain property on 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 operations (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 [11,12,14,18,20,21] and for directed graphs [9,10,19,23,24]. One of the most studied dynamic graph problem is perhaps the fully dynamic maintenance of a minimum spanning tree (MST)

* This work has been partially supported by the IST Programme of the EU under contract n. IST-1999-14186 (ALCOM-FT), by the Italian Ministry of University and Scientific Research (Project "Algorithms for Large Data Sets: Science and Engineering") and by CNR, the Italian National Research Council, under contract n. 01.00690.CT26.

of a graph [3,13,11,14,15,20,21]. 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 [14,15], the sparsification technique by Eppstein *et al.* [11] and the logarithmic decomposition by Holm *et al.* [21].

Many researchers have been complementing this wealth of theoretical results on dynamic graphs with thorough empirical studies, in the effort of bridging the gap between the design and theoretical analysis and the actual implementation, experimental tuning and practical performance evaluation of dynamic graph algorithms. In particular, Alberts *et al.* [1] implemented and tested algorithms for fully dynamic connectivity problems: the randomized algorithm of Henzinger and King [18], and sparsification [11] on top of a simple static algorithm. Amato *et al.* [5] proposed and analyzed efficient implementations of dynamic MST algorithms: the partitions and topology trees of Frederickson [14, 15], and sparsification on top of dynamic algorithms [11]. Miller *et al.* [17] proposed efficient implementations of dynamic transitive closure algorithms, while Frigioni *et al.* [16] and later Demetrescu *et al.* [8] conducted an empirical study of dynamic shortest path algorithms. Most of these implementations have been wrapped up in a software package for dynamic graph algorithms [2]. Finally, Iyer *et al.* [22] implemented and evaluated experimentally the recent fully dynamic connectivity algorithm of Holm *et al.* [21], thus greatly enhancing our knowledge on the practical performance of dynamic connectivity algorithms.

The objective of this paper is to advance our knowledge on dynamic MST algorithms by following up the recent theoretical progress of Holm *et al.* [21] with a thorough empirical study. In particular: (1) we present and experiment with efficient implementations of the dynamic MST algorithm of Holm *et al.* [21], (2) we propose new simple algorithms for dynamic MST, which are not as asymptotically efficient as [21], but nevertheless seem quite fast in practice, and (3) we compare all these new implementations with previously known algorithmic codes for dynamic MST [5], such as the partitions and topology trees of Frederickson [14,15], and sparsification on top of dynamic algorithms [11].

We found the implementations contained in [22] 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. With this bulk of implementations, we performed extensive tests under several variations of graph and update parameters in order to gain a deeper understanding on 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 both 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.

2 The Algorithm by Holm *et al.*

In this section we quickly review the algorithm by Holm *et al.* for fully dynamic MST. We will start with their algorithm for handling deletions only, and then sketch on how to transform this deletion-only algorithm into a fully dynamic one. The details of the method can be found in [21].

2.1 Decremental Minimum Spanning Tree

We maintain a minimum spanning forest F over a graph G having n nodes and m edges. All the edges belonging to F will be referred as tree-edges. The main idea behind the algorithm is to partition the edges of G into different levels. Roughly speaking, whenever we delete edge (x, y), we start looking for a replacement edge at the same level as (x, y). If this search fails, we consider edges at the previous level and so on until a replacement edge is found. This strategy is effective if we could 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 having level at least i. Thus, $F = F_0 \supseteq F_1 \supseteq ... \supseteq F_L$. The following invariants are maintained throughout the sequence of updates:

1. F is a maximum (w.r.t. ℓ) spanning forest of G, that is, if (v, w) is a non-tree edge, v and w are connected in $F_{l(v,w)}$
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.
3. If e is the heaviest edge on a cycle C, then e has the lowest level on C.

We 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 then 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, 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 there is no replacement edge on level $> i$, finds a replacement edge of the highest level $\leq i$, if any. The algorithm works as follows. Let T_v and T_w be the tree 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. First, we move all edges of level i of T_v to level $(i+1)$, then, we start 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 and the search stops. If the search fails, we call Replace$((u, w), i - 1)$. When the search fails also at level 0, we stop as there is no replacement edge for (v, w).

2.2 The Fully Dynamic Algorithm

Starting from the deletions-only algorithm, Holm *et al.* obtained a fully dynamic algorithm by using a clever refinement of a technique by Henzinger and King [20] for developing fully dynamic data structures starting from deletions-only data structures.

We maintain the set of data structures $\mathcal{A} = A_1, .., A_{s-1}, A_s$, $s = \lceil(\log n)\rceil$, where each A_i is a subgraph of G. We denote by F_i the local spanning forest maintained on each A_i. We will refer to edges in F_i as *local tree edges* while we will refer to edges in F as *global tree edges.* All edges in G will be in at least one A_i, so we have $F \subseteq \bigcup_i F_i$. During the algorithm we maintain the following invariant:

1. For each global non-tree edge $f \in G \backslash F$, there is exactly one i such that $f \in A_i \backslash F_i$ and if $f \in F_j$, then $j > i$.

Without loss of generality assume that the graph is connected, so that we will talk about MST rather than minimum spanning forest. At this point, we use a dynamic tree of Sleator and Tarjan [28] 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 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 call the update procedure on $\mathcal{A}$.
Delete(e) We delete e from all the A_i and we collect in a set R all the replacement edges returned from each deletions-only data structure. Then, we check if e is F, if so we search in R the minimum cost edge reconnecting F. Finally, we update $\mathcal{A}$ using R.
Update A with edge set D We find the smallest j such that $|(D \bigcup_{h \le j}(A_h \setminus F_h)) \setminus F| \le 2^j$. Then we set

$$A_j = F \cup D \cup \bigcup\nolimits_{h \le j}(A_h \setminus F_h)),$$

and we initialize A_j as a MST deletions-only data structure. Finally we set $A_h = 0$ for all $h < j$.

The initialization required by the updates is one of the crucial points in the algorithm by Holm *et al.* To perform this task efficiently, they use a particular compression of some subpaths, which allows one to bound the initialization work at each update. This compression is carried out with the help of top trees: details can be found in [21].

2.3 Our Implementation

Our implementation follows exactly the specifications of Holm *et al.* except for the use of the compression technique described in [21]. Indeed, our first experience with the compression and top trees was rather discouraging from the experimental viewpoint. In particular, the memory requirement was substantial so that we could not experiment with medium to large size graphs (order of thousands of vertices). We thus engineered a different implementation of the compression technique using the dynamic trees of Sleator and Tarjan [28] in place of the top trees; this yielded a consistent gain on the memory requirements of the resulting algorithm with respect to our original implementation. We are currently working on a more sophisticated implementation of top trees that should allow us to implement in a faster way the original compression.

3 Simple Algorithms

In this section we describe two simple algorithms for dynamic MST that are used in our experiments. They are basically a fast "dynamization" of the static algorithm by Kruskal (see e.g., [7,25]). We recall here that Kruskal's algorithm grows a forest by scanning all the graph edges by increasing cost: if an edge (u, v) joins two different trees in the forest, (u, v) is kept and the two trees are joined. Otherwise, u and v are already in a same tree, and (u, v) is discarded.

3.1 ST-Based Dynamic Algorithm

Our dynamization of the algorithm by Kruskal uses the following ideas. Throughout the sequence of updates, we keep the following data structures: the minimum spanning tree is maintained with a dynamic tree of Sleator and Tarjan [28], say MST, while non-tree edges are maintained sorted in a binary search tree, say NT.

When a new edge (x, y) is to be inserted, we check in $O(\log n)$ time with the help of dynamic trees whether (x, y) will become part of the solution. If this is the case, we insert (x, y) into MST: the swapped edge will be deleted from MST and inserted into NT. Otherwise, (x, y) will not be a tree edge and we simply insert it into NT.

When edge (x, y) has to be deleted, we distinguish two cases: if (x, y) is a non-tree edge, then we simply delete it from NT in $O(\log n)$. If (x, y) is a tree edge, its deletion disconnects the minimum spanning tree into T_x (containing x) and T_y (containing y), and we have to look for a replacement edge for (x, y). We examine non-tree edges in NT by increasing costs and try to apply the scanning strategy of Kruskal on T_x and T_y: namely, for each non-tree edge $e = (u, v)$, in increasing order, we check whether e reconnects T_x and T_y: this can be done via findroot(u) and findroot(v) operations in Sleator and Tarjan's trees. Whenever such an edge is found, we insert it into MST and stop our scanning. The total time required by a deletion is $O(k \cdot \log n)$, where k is the total number of non-tree edges scanned. In the worst case, this is $O(m \log n)$.

We refer to this algorithm as `ST`. Note that `ST` requires few lines of code, and fast data structures, such as the dynamic trees [28]. We therefore expect it to be very fast in practice, especially in update sequences containing few tree edge deletions or in cases when, after deleting tree edge (x, y), the two trees T_x and T_y get easily reconnected (e.g., the cut defined by (x, y) contains edges with relatively small costs).

3.2 ET-Based Dynamic Algorithm

Our first experiments with `ST` showed that it was indeed fast in many situations. However, the most difficult cases for `ST` were on sparse graphs, i.e., graphs with around n vertices. In particular, the theory of random graphs [6] tells us that when $m = n/2$ the graph consists of a few large components of size $O(n^{2/3})$ and smaller components of size $O(\log n)$. For $m = 2n$, the graph contains a giant component of size $O(n)$ and smaller components of size $O(\log n)$. In these random cases, 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. Indeed, a careful profiling showed that most of the time in these cases was spent by `ST` in executing findroot operations on Sleator and Tarjan's trees (ST trees).

We thus designed another variant of this algorithm, referred to as `ET`, which uses Euler Tour trees (ET trees) in addition to ST trees. The ET-tree (see e.g., [18] for more details) is a balanced search tree used to efficiently maintain 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 [4] to support the ET-trees.

In particular, we keep information about tree edges both with an ST tree and with an ET tree. The only place were we use the ET-tree is in the deletion algorithm, i.e., where we check whether a non-tree edge $e = (u, v)$ reconnects the minimum spanning tree. We still need ST-trees, however, as they give a fast method for handling edge insertions. Note that `ET` has the same asymptotical update bounds as `ST`.

The main difference between the two implementations is that we expect findroot operations on randomized search trees to be faster than on Sleator and Tarjan's trees, and consequently `ET` to be faster than `ST` on sparse graphs. However, when findroot operations are no longer the bottleneck, we also expect that the overhead of maintaining both ET-trees and ST-trees may become significant. This was exactly confirmed by our experiments, as shown in Fig. 3.2, which illustrates an experiment on random graphs with 2,000 vertices and different densities.

3.3 Algorithms Tested

We have developed, tested and engineered many variants of our implementations. All the codes are written in `C++` with the support of the LEDA [26] algorithmic software library, and are homogeneously written by the same people, i.e., with the same algorithmic and programming skills. In this extended abstract, we will report only on the following four implementations:

`HDT`	Our implementation of the algorithm proposed by Holm *et al.* with the original deletions-only fully-dynamic transformation proposed by Henzinger and King. It uses randomized balanced ET Trees from [1].
`Spars`	Simple sparsification run on top of Frederickson's light partition of order $\lceil m^{2/3} \rceil$ and on lazy updates, as described in [5].

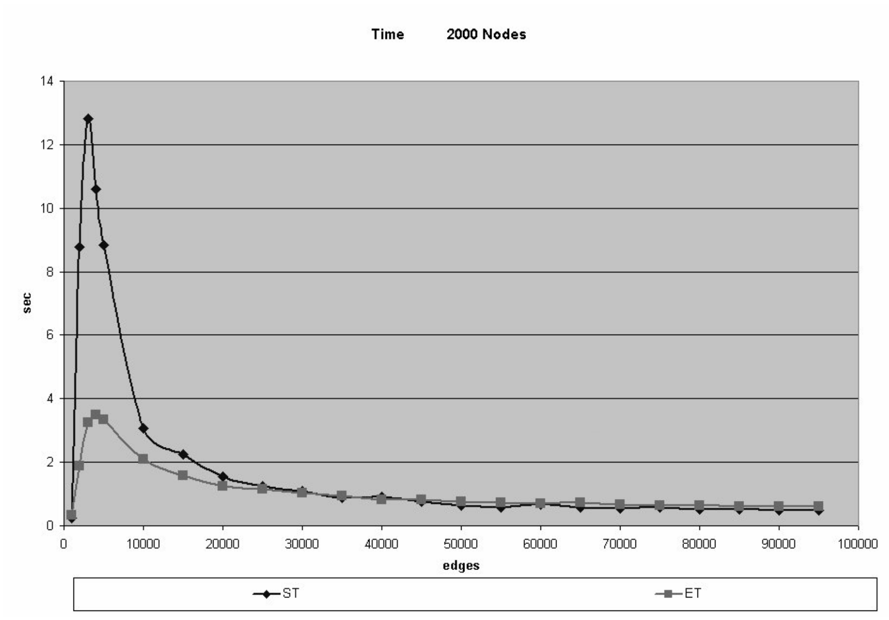

Fig. 1. ET and ST on random graphs with 2,000 vertices and different densities. Update sequences were random and contained 10,000 edge insertions and 10,000 edge deletions.

ST	The implementation of the algorithm described in Sect. 3.1.
ET	The implementation of the algorithm described in Sect. 3.2.

4 Experimental Settings

We have conducted our tests on a Pentium III (650 MHz) under Linux 2.2 with 768 MB of physical RAM, 16 KB L1-cache and 256 KB L2-cache. We choose this platform as it was the one with largest RAM available to us, so that we could experiment with large graphs without incurring in external memory swap problems. Our codes were compiled with a `Gnu C++ 2.9.2` compiler with the `-O` flag. The metrics we have used to rate the algorithms' performance were the total amount of CPU time in user mode spent by each process and the maximum amount of memory allocated during the algorithms' execution.

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 experiments. Building on previous experimental work on dynamic graph algorithms [1,5,22], we considered the following test sets:

Random Graphs. The initial graph G is generated randomly according to a pair of input parameters (n, m), where n is the initial number of nodes and m the initial number of 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.

Semirandom Graphs. We generate a random graph, and choose a fix number of candidate edges E. All the edge costs are randomly chosen. The updates here contain random insertions or random deletions from E only. As pointed out in [22], this semirandom model seems slightly more realistic than true random graphs in the application of maintaining a network when links fail and recover.

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 those cliques with $2k$ randomly chosen inter-clique edges. As before, all the edge costs are randomly chosen. Note that any spanning tree of this graph consists of intra-clique trees connected by inter-clique edges. For these k-clique graphs, we considered different types of updates. On the 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 operations 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 [22], 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 terrains, ranging from worst-case inputs (inter-clique updates only) to more mixed inputs (both inter- and intra-clique updates).

Worst-case inputs. For sake of completeness, we adapted to minimum spanning trees also the worst-case inputs introduced by Iyer *at al.* [22] for connectivity. These are inputs that try to force a bad sequence of updates for `HDT`, and in particular try to promote as many edges as possible through the levels of its data structures. We refer the interested reader to reference [22] for the full details. For the sake of completeness, we only mention here that in these test sets there are no non-tree edges, and only tree edge deletions are supported: throughout these updates, `HDT` is foiled by unneeded tree edge movements among levels.

5 Experimental Results

Random inputs. Not surprisingly, on random graphs and random updates, the fastest algorithms were `ST` and `ET`. This can be easily explained, as in this case edge updates are not very likely to change the MST, especially on dense graphs: `ST` and `ET` are very simple algorithms, can be coded in few lines, and therefore likely to be superior in such a simple scenario. The only interesting issue was perhaps the case of sparse random graphs, where a large number of updates can force changes in the solution. As already explained in Sect. 3.2, `ST` does not perform well in such a scenario, while `ET` exhibits a more stable behavior. As already observed in [22], the decomposition into edge levels of [21] helps `HDT` "learn" about and adapt to the structure of the underlying graph. A random graph has obviously no particular structure, and thus all the machinery of `HDT` would not seem particularly helpful in this case. It was interesting to notice, however, that even in this experiment, `HDT` was slower than `ET` and `ST` only by a factor of 5. As far as `Spars` is concerned, it was particularly appealing for sparse graphs, when the sparsification tree had a very small height. As the number of edges increased, the overhead of the sparsification tree became more significant. Figure 2 illustrates the results of our experiments with random graphs with 2, 000 vertices and different densities. Other graph sizes exhibited a similar behavior.

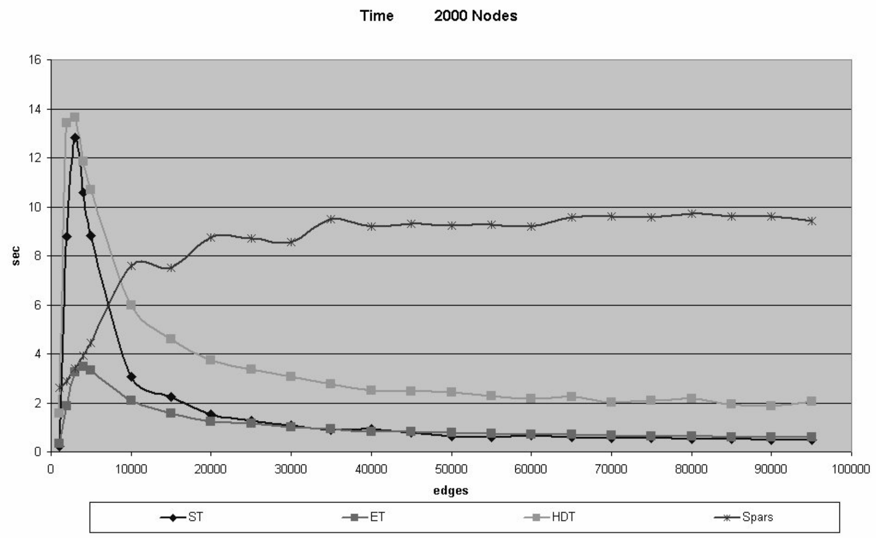

Fig. 2. Experiments on random graphs with 2,000 vertices and different densities. Update sequences contained 10,000 insertions and 10,000 deletions.

We also experimented with the operations by changing the sequence mix: to force more work to the algorithms, we increased the percentage of tree edge deletions. Indeed, this seems to be the hardest operation to support, as non-tree deletion is trivial, and tree insertions can be easily handled with ST trees. On random graphs, `ET` remained consistently stable even with evenly mixed update sequences (50% insertions, 50% deletions) containing 45% of tree edge deletions. `ST` was penalized more in this, because of the higher cost of findroot on ST-trees. Figure 3 reports the results of such an experiment.

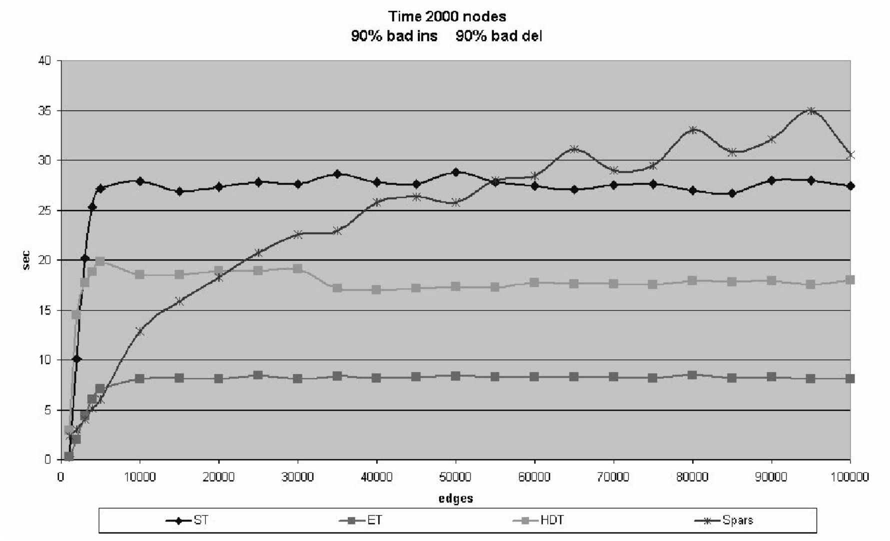

Fig. 3. Experiments on random graphs with 2,000 vertices and different densities. Update sequences contained 10,000 insertions and 10,000 deletions: 45% of the operations were tree edge deletions.

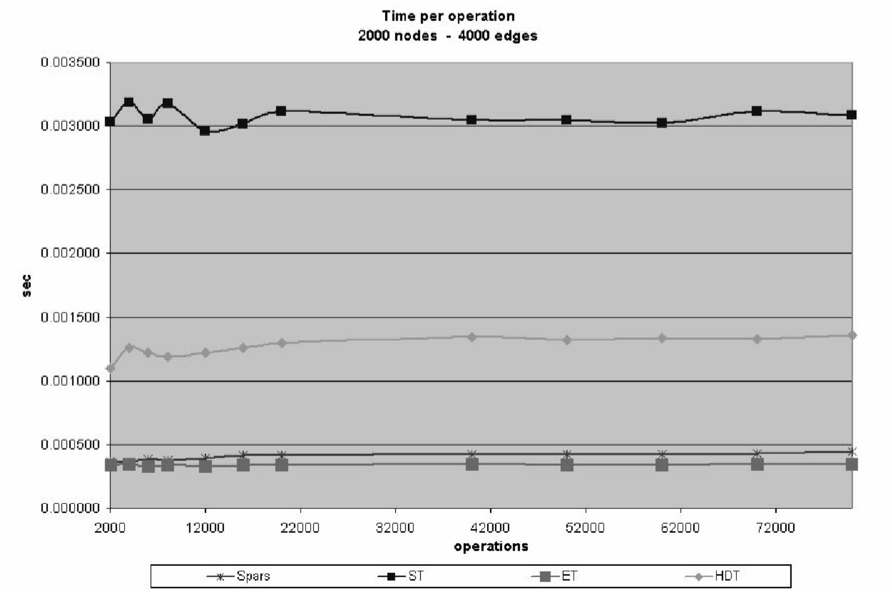

Fig. 4. Experiments on semirandom graphs with 2,000 vertices and 4,000 edges. The number of operations ranges from 2,000 to 80,000.

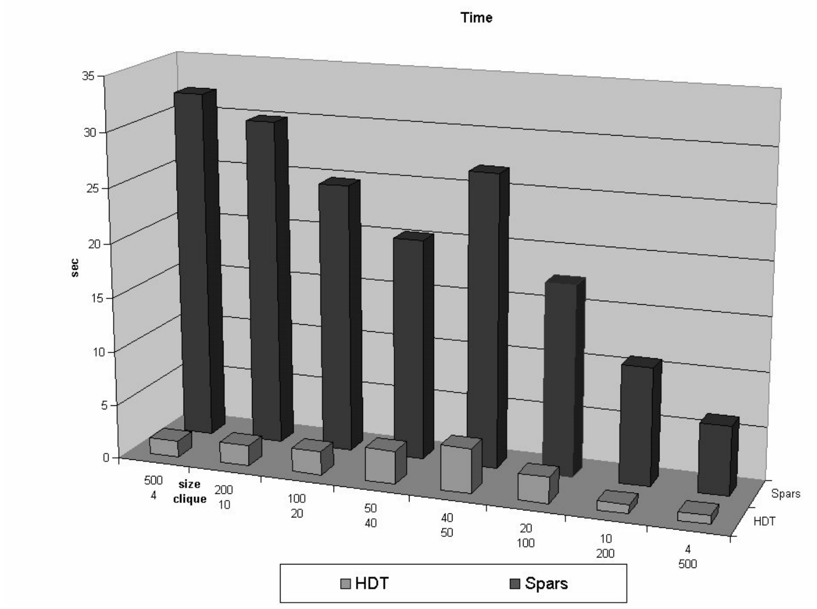

Fig. 5. Experiments on k-cliques graphs with inter-clique operations only.

Semirandom inputs. We have made several experiments with semirandom inputs using random graphs with 2,000 vertices and a number of operations ranging from 2,000 to 8,000. In the first experiment we have chosen a subset E of 1,000 edges. As already pointed out by Holm *et al* in [22], in this case we have many disconnected components that will never be joined again since the edge set E is fixed. The performance of ST and ET seems quite good since very few non-tree edges are candidates for being replacement edges; the same behavior seems to apply to HDT. On the other hand, here Spars performs very badly compared to the other algorithms: although the graphs we are experimenting with are very sparse, in this case there is a high overhead due to the underlying partition of Frederickson. This situation totally changes when we increase the size of E. In our second and third experiment we have fixed the E size respectively to 2,000 and 4,000 edges. According to our results illustrated in Fig. 4, ET still remained the fastest algorithm together with Spars that seemed to be the most stable algorithm among the one we tested for this kind of experiments. ST and HDT suffered a significant performance loss. In the case of ST, its behavior has been very similar to the one we measured in the Random experiments and is probably due to the overhead of findroots.

k-Clique inputs. For these tests, we considered two kinds of update sequences, depending on whether all update sequences were related only to inter-clique edges or could be related to intra-clique edges as well. In the first case, where only inter-clique edges were involved HDT was by far the quickest implementation. In

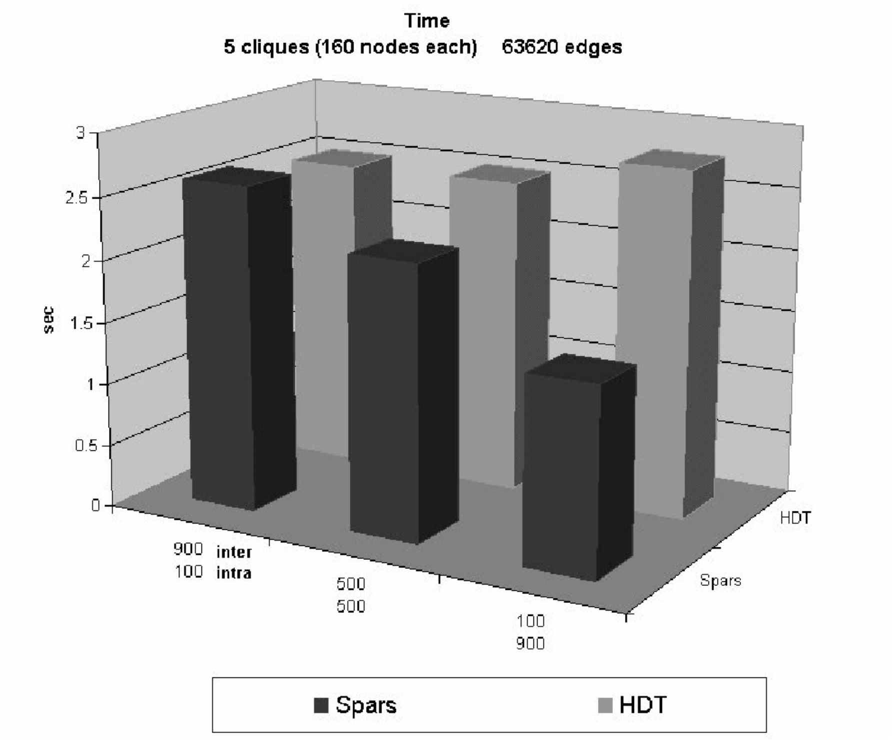

Fig. 6. Experiments on k-cliques graphs with a different mix of inter- and intra-clique operations.

fact, stimulated by inter-clique updates, HDT was quite fast in learning the 2-level structure of the graph, and in organizing its level decomposition accordingly, as shown in Fig. 5.

When updates could involve also intra-clique edges as well, however, the random structure of the update sequence was somehow capable of hiding from HDT the graph structure, thus hitting its level decomposition strategy. Indeed, as it can be seen from Fig. 6, as the number of operations involving intra-clique edges increased, the performance of Spars improved (updates on intra-clique edges are not likely to change the MST and thus will not propagate all the way up to the sparsification tree root), while on the contrary the performance of HDT slightly deteriorated.

In both cases, ET and ST were not competitive on these test sets, as the deletion of an inter-clique edge, for which the set of replacement edges is very small, could have a disastrous impact on those algorithms.

Worst-case inputs. Figure 7 illustrates the results of these tests on graphs with up to 32,768 vertices. As expected, HDT is tricked by the update sequence and spends a lot of time in (unnecessary!) promotions of edges among levels of the

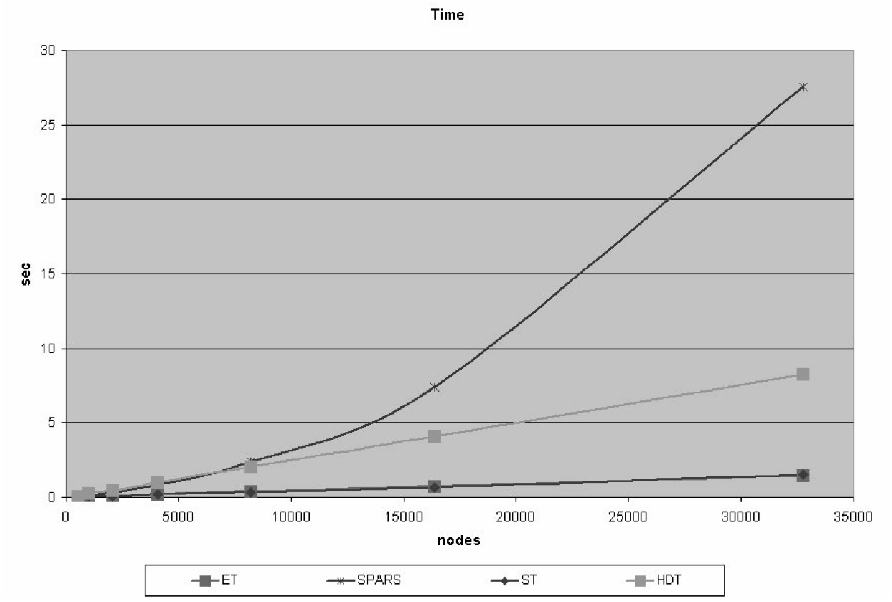

Fig. 7. Experiments on worst-case inputs on graphs with different number of vertices.

data structures. `ET` and `ST` achieve their best case of $O(\log n)$ time, as there are no non-tree edges to consider. `Spars` is also hit pretty badly by these test sets: indeed each tree edge deletion suffers from the overhead of the underlying implementation of Frederickson's light partition of order $\lceil m^{2/3} \rceil$.

References

1. D. Alberts, G. Cattaneo, G. F. Italiano, "An empirical study of dynamic graph algorithms", *ACM Journal on Experimental Algorithmics*, vol. 2 (1997).
2. D. Alberts, G. Cattaneo, G. F. Italiano, U. Nanni, C. D. Zaroliagis "A Software Library of Dynamic Graph Algorithms". *Proc. Algorithms and Experiments* (ALEX 98), Trento, Italy, February 9–11, 1998, R. Battiti and A. A. Bertossi (Eds), pp. 129–136.
3. D. Alberts, M. R. Henzinger, "Average Case Analysis of Dynamic Graph Algorithms", *Proc. 6th Symp. on Discrete Algorithms* (1995), 312–321.
4. C.R. Aragon, R. Seidel, "Randomized search trees", *Proc. 30th Annual Symp. on Foundations of Computer Science* (FOCS 89), 1989, pp. 540–545.
5. G. Amato, G. Cattaneo, G. F. Italiano, "Experimental Analysis of Dynamic Minimum Spanning Tree Algorithms", *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, (1997), 5–7.
6. B. Bollobás. *Random Graphs.* Academic Press, London, 1985.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms.* Mac-Graw Hill, NY, 1990.

8. C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, U. Nanni. "Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study." *Procs. 3rd Workshop on Algorithm Engineering* (WAE2000). Lecture Notes in Computer Science, Springer-Verlag, 2001.
9. C. Demetrescu and G.F. Italiano. Fully dynamic transitive closure: Breaking through the O(n2) barrier. *Proc. of the 41st IEEE Annual Symposium on Foundations of Computer Science* (FOCS'00), pages 381-389, 2000.
10. C. Demetrescu, G. F. Italiano, "Fully Dynamic All Pairs Shortest Paths with Real Edge Weights". *Proc. 42nd IEEE Annual Symp. on Foundations of Computer Science* (FOCS 2001), Las Vegas, NV, U.S.A., October 14-17, 2001.
11. D. Eppstein, Z. Galil, G. F. Italiano and A. Nissenzweig, "*Sparsification – A technique for speeding up dynamic graph algorithms*". In *Journal of ACM*, Vol. 44, 1997, pp. 669-696.
12. D. Eppstein, Z. Galil, G. F. Italiano, T. H. Spencer, "Separator based sparsification for dynamic planar graph algorithms", *Proc. 25th ACM Symposium on Theory of Computing* (1993), 208–217.
13. D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung, Maintenance of a minimum spanning forest in a dynamic plane graph, *J. Algorithms*, 13:33–54, 1992.
14. G.N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications", *SIAM J. Comput.* 14 (1985), 781–798.
15. G.N. Frederickson, "Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees", *Proc. 32nd IEEE Symp. Foundations of Computer Science* (1991), 632–641.
16. 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*, vol. 3 (1998), Article 5.
17. D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Schaefer, and C. Zaroliagis "An experimental study of dynamic algorithms for directed graphs", in *Proc. 6th European Symp. on Algorithms*, Lecture Notes in Computer Science 1461 (Springer-Verlag, 1998), pp.368-380.
18. M. R. Henzinger and V. King, Randomized dynamic graph algorithms with polylogarithmic time per operation,*Proc. 27th Symp. on Theory of Computing*, 1995, 519–527.
19. M. R. Henzinger and V. King, Fully dynamic biconnectivity and transitive closure, *Proc. 36th IEEE Symp. Foundations of Computer Science*, pages 664–672, 1995.
20. M. R. Henzinger and V. King, Maintainig minimum spanning trees in dynamic graphs, *Proc. 24th Int. Coll. Automata, Languages and Programming* (ICALP 97), pages 594–604, 1997.
21. J. Holm, K. de Lichtenberg, and M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Proc. 30th Symp. on Theory of Computing* (1998), pp. 79-89.
22. R. Iyer, D. R. Karger, H. S. Rahul, and M. Thorup, An Experimental Study of Poly-Logarithmic Fully-Dynamic Connectivity Algorithms, *Proc. Workshop on Algorithm Engineering and Experimentation*, 2000.
23. V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. *Proc. 40th IEEE Symposium on Foundations of Computer Science* (FOCS'99), 1999.
24. V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Proc. 31st ACM Symposium on Theory of Computing* (STOC'99), pages 492-498, 1999.

25. J. B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, *Proc. Amer. Math. Soc. 7*, (1956), 48–50.
26. K. Melhorn and S. Näher, “*LEDA, A platform for combinatorial and geometric computing*”. *Comm. ACM*, (1995), 38(1): pp.96-102.
27. M. Rauch, “Improved data structures for fully dynamic biconnectivity”, *Proc. 26th Symp. on Theory of Computing* (1994), 686–695.
28. D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *J. Comp. Syst. Sci.*, 24:362–381, 1983.
29. R. E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.*, 31:245–281, 1984.