Experimental Analysis of Dynamic Minimum Spanning Tree Algorithms (Extended Abstract)

Giuseppe Amato* Giuseppe Cattaneo† Giuseppe F. Italiano‡

Abstract

We conduct an extensive empirical study on the performance of several algorithms for maintaining the minimum spanning tree of a dynamic graph. In particular, we implemented and tested Frederickson's algorithms, and sparsification on top of Frederickson's algorithms, and compared them to other dynamic algorithms. Moreover, we propose a variant of a dynamic algorithm by Frederickson, which was in our experience always faster than the other implementations derived from the papers. In our experiments, we considered both random and non–random inputs, with non–random inputs trying to enforce bad update patterns on the algorithms. For random inputs, a simple adaptation of a partially dynamic data structure on Kruskal's algorithm was the fastest implementation. For non–random inputs, sparsification yielded the fastest algorithm. In both cases, the performance of our variant of the algorithm of Frederickson was close to that of the fastest algorithm.

1 Introduction

A *dynamic graph algorithm* maintains a certain property – like connectivity, 2-edge connectivity, 2-vertex connectivity, or a property on minimum spanning trees – 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 operations (i.e., either insertions or deletions, but not both) is allowed. The area of dynamic graph algorithms has been a blossoming field of research in the last years, and it has produced a large body of algorithmic techniques [2, 5, 6, 7, 8, 9, 10, 11, 14]. Perhaps one of the most studied problems in this area is the fully dynamic maintenance of the minimum spanning tree of a graph [2, 5, 7, 8]. This problem is important on its own, and it finds applications to other problems as well, including many dynamic vertex and edge connectivity problems. Many elegant solutions have been proposed for the dynamic minimum spanning tree problem, such as the partitions and topology trees of Frederickson [8, 9], and sparsification [5]. In [1] we tried to make a first step toward bridging the gap between the design and theoretical analysis of dynamic graph algorithms on the one side, and their implementation, experimental tuning and practical performance evaluation on the other side. In particular, in [1] the attention was devoted to simpler fully dynamic connectivity problems: in this framework, the randomized algorithm of Henzinger and King [10], and a simple version of sparsification [5] on top of a static algorithm were implemented and tested.

In this paper, we make another important step towards this direction by performing an extensive experimental study of dynamic minimum spanning tree algorithms. Towards this end, we implemented and evaluated two important algorithmic techniques: the partitions and topology trees of Frederickson [8, 9], and sparsification on top of dynamic algorithms [5]. In particular, we implemented several dynamic minimum spanning tree algorithms based on the approach of Frederickson, and enhanced our implementation of sparsification [1] so that it applies to dynamic algorithms (such as Frederickson's algorithms) as well. This is the first implementation of the dynamic graph algorithms by Frederickson that we know of, and it required a lot of effort and engineering: we believe that this implementation is particularly valuable, as it proves that those techniques are not only theoretically interesting, but can also be effectively used in practical algorithms. Beside the implementations based on theoretically efficient al-

*Dipartimento di Informatica ed Applicazioni, Università di Salerno, Italy.

†Dipartimento di Informatica ed Applicazioni, Università di Salerno, Italy. E-mail: cattaneo@dia.unisa.it.

‡Dipartimento di Matematica Applicatica ed Informatica, Università "Ca' Foscari" di Venezia, Italy. Supported in part by the ESPRIT LTR Project no. 20244 (ALCOM-IT) and by a Research Grant from University of Venice "Ca' Foscari". E-mail: italiano@unive.it. URL: http://www.dsi.unive.it/~italiano.

gorithms, we also considered some simple-minded algorithms, which were very easy to implement in only a few lines of code. Usually, this means that their implementation constants are extremely low, and that these algorithms are likely to be very fast in practice.

With this bulk of implementations, we performed extensive tests under several variations of graph and update parameters in order to select the fastest algorithms in practice. 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.

Among six different implementations based on the papers by Frederickson [8, 9], whose running times per update are ranging from $O(m^{2/3})$ to $O(m^{1/2})$ (with m being the current number of edges in the graph), we selected the algorithm that was fastest in practice. Not surprisingly, this was not the theoretical fastest algorithm. We also propose a variant of Frederickson's algorithm which was faster in practice than all the other implementations of Frederickson's algorithms. Based on experimental tuning, we also propose an adaptation of a partially dynamic data structure on Kruskal's algorithm [12], which in our tests was the fastest implementation for random inputs. For non-random inputs, sparsification on top of our variant of Frederickson's algorithm was the fastest algorithm. All our implementations are written in `C++` and are based on **LEDA**, the library of efficient data types and algorithms developed by Mehlhorn and Näher in Saarbrücken [13], from which our work benefited greatly. Our source codes are available via anonymous ftp from `ftp.dia.unisa.it` in the directory `pub/italiano/mst`.

2 Implementing Frederickson's Algorithms

Let $G = (V, E)$ be a graph, with minimum spanning tree T. The first ingredient of the algorithms of Frederickson [8, 9] is *clustering*. Namely, 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, is maintained. Basically, this partition divides evenly G into $O(m/z)$ clusters of size z each, with $z \geq 0$ being an integer. 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 `FredI` 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 `FredII`, which yields better $O((m \log m)^{1/2})$ [1] time bounds per update. Finally, algorithm `FredIII` 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 [8] and the other in [9]), which have similar theoretical behavior. These two different partitions give rise to six different algorithms: `FredI-85`, `FredII-85`, and `FredIII-85` based on the partition defined in [8], and `FredI-91`, `FredII-91`, and `FredIII-91` based on the partition defined in [9].

Before defining the partitions and the notion of topology tree, we describe a standard graph transformation that is used throughout [8]. We convert the graph G into a graph with maximum vertex degree 3: Suppose $v \in V$ has degree $d(v) > 3$, and is adjacent to vertices $u_1, u_2, \ldots, u_d$. In the transformed graph, v is replaced by a cycle of d *loopy* edges. Namely, we substitute v by d vertices $v_1, v_2, \ldots, v_d$. For each edge (v, u) of the original graph, in position i among the list of edges adjacent to v and position j among the edges adjacent to u, we create an *actual* edge (v_i, u_j). We also create *loopy* edges (v_i, v_{i+1}) for $1 \leq i \leq d-1$, and a *loopy* edge (v_d, v_1) to close the cycle. We call these edges the *loopy edges of* v. As a result of this transformation, the graph keeps its actual edges, and has an additional $O(m)$ loopy edges. For each vertex v, the cost of its loopy edges is defined so that the minimum spanning tree T contains all of the loopy edges of v except one. Note that we can easily compute the minimum spanning tree of the original graph once we know the minimum spanning tree in the transformed graph. Furthermore, this transformation can be easily updated during insertions and deletions of edges. Without loss of generality, in this section we assume that the graph G has degree less than or equal to three. Although this transformation has no major theoretical difficulties, our experiments showed that its maintenance throughout the execution of the algorithm had a big impact in the practical performance of the algorithms.

To describe the two different partitions of [8, 9], we need a little terminology. A *vertex cluster* with respect to T is a set of vertices that induces a connected subgraph on T. The *cardinality* of a cluster is the number of vertices in it. An edge is *incident* to a cluster if exactly one of its endpoints is inside the cluster.

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

Two clusters are *adjacent* if there is a tree edge that is incident to both. A *boundary vertex* of a cluster is a vertex that is adjacent in T to some vertex not in the cluster. The *external degree* of a cluster is the number of tree edges incident to it. Let $z > 0$ be an integer, to be fixed later on.

A *balanced partition of order* z of a graph $G = (V, E)$ [8] is simply a partition of its vertex set V into vertex clusters of cardinality between z and $(3z - 2)$. The second partition [9] has more strict requirements. It is called a *restricted partition of order* z, and is defined as follows: (1) Each set in the partition yields a vertex cluster of external degree at most 3; (2) Each cluster of external degree 3 is of cardinality 1; (3) Each cluster of external degree less than 3 is of cardinality less than or equal to z; and (4) No two adjacent clusters can be combined and still satisfy the above.

Based on these definitions, a restricted partition is expected to have a larger number of clusters than a balanced partition. However, it can be shown that both a balanced and a restricted partition of order z have at most $O(m/z)$ clusters. Furthermore, both partitions can be updated in $O(z)$ time during the insertion of an edge, the deletion of an edge, or an edge swap by using greedy update rules [8, 9]. The crucial (and expensive) operations in the dynamic maintenance of these partitions are the split of a cluster into two, and the merge of two neighbor clusters into one. Once a (either balanced or restricted) partition is maintained, it is easy to maintain dynamically the minimum spanning tree of a graph in time $O(z + (m/z)^2) = O(m^{2/3})$ as described in [8]. This yields two algorithms: `FredI-85` based on balanced partitions, and `FredI-91` based on restricted partitions.

Although they both have the same asymptotical behavior, there are many practical differences between balanced and restricted partitions. Indeed, our experiments showed the following. First of all, for the same value of z a balanced partition generates substantially less clusters than a restricted partition. Second, the gap of $(2z - 2)$ between the lower and the upper bound on the cardinality of a cluster, implies that a cluster of a balanced partition has usually a longer lifetime than a restricted partition cluster, and is less affected by the updates. Finally, as a balanced partition must obey to less stringent requirements, it generates a smaller number of cluster splits and merges throughout the execution of the algorithm. We also found out experimentally that the average number of clusters affected by an update was much larger for restricted partitions than for balanced partitions. Our experiments consistently reported that `FredI-85` was faster in practice than `FredI-91`. Based on the results of these tests, we decided to do an extensive theoretical and experimental tuning with the parameters of the two partitions. In this tuning, we found out that a new partition, which we called *light partition*, was much faster in practice than either a balanced or a restricted partition. A light partition of order z is basically a relaxation of a restricted partition of order z, and is defined as follows: (1) Each cluster is of cardinality less than or equal to z; and (2) No two adjacent clusters can be combined and still satisfy the above.

A light partition of order z can be initialized so as to have $O(m/z)$ clusters by simply starting from a restricted partition of order z and by merging neighbor clusters without violating rules (1) and (2) above. The number of clusters of this light partition is thus no more than the number of clusters in the initial restricted partition (which is $O(m/z)$). Unfortunately, differently from the case of balanced and restricted partitions, it cannot be guaranteed that the number of clusters in a light partition of order z remains $O(m/z)$ throughout any sequence of updates. Hence, the worst-case asymptotical running times of algorithms based on light partitions are worse than $O(m^{2/3})$. However, our experiments showed that, when we started with an initial light partition with $O(m/z)$ clusters, the number of clusters in the light partition did not increase substantially as the number of updates increased.

Another mechanism that we found very useful in practice was to use a lazy update scheme: during insertions of edges, we did not update the light partition immediately. Rather, we recomputed in $O(\log n)$ time the minimum spanning tree of the underlying graph by means of the linking and cutting tree data structure of Sleator and Tarjan [15], and deferred the expensive task of updating the light partition when the next subsequent deletion would cause changes in the solution. This scheme was very useful in mixed sequences of insertions and deletions of edges, as it reduced substantially the time devoted to updating the partition. Its potential advantage should be clear, as in a burst of operations the same cluster could be affected by several updates during different operations: with the lazy mechanism, it would be updated only at the time of a tree edge deletion.

We implemented an algorithm based on a light partition of order $\lceil m^{2/3} \rceil$ and this lazy update mechanism. In this paper we refer to this algorithm as `FredI-Mod`. In all our experiments, `FredI-Mod` was always substantially faster than both `FredI-85` and `FredI-91`.

We now recall the definition of topology tree from [8, 9]. We apply recursively balanced or restricted partitions, yielding two different types of multi-level partitions. A *multi-level balanced partition* satisfies the following properties: (1) The clusters at level 0 contain a single vertex; (2) A cluster at level $\ell \geq 1$ is either a cluster at level $(i-1)$ of external degree 3, or the union of at most 4 clusters at level $(i-1)$ according to some allowable topologies (see [8] for the details); and (3) There is exactly one vertex cluster at the topmost level. Similarly, the notion of *multi-level restricted partition* can be formalized as follows. (1) The clusters at level 0 contain a single vertex; (2) The clusters at level $\ell \geq 1$ form a restricted partition of order 2 with respect to the tree obtained after shrinking all the clusters at level $\ell - 1$; and (3) There is exactly one vertex cluster at the topmost level. Note that the definition of a multi-level restricted partition, in particular rule (2) above, is much simpler than the definition of a multi-level balanced partition.

The *topology tree* is a hierarchical representation of G based on T. Each level of the topology tree partitions the vertices of G into connected subsets called *clusters*. More precisely, given a multi-level balanced (respectively restricted) partition for T, a *balanced* (respectively *restricted*) *topology tree* for T is a tree satisfying the following: (1) A topology tree node at level ℓ represents a vertex cluster at level ℓ in the multi-level balanced (respectively restricted) partition; and (2) The children of a node at level $\ell \geq 1$ are the vertex clusters at level $(\ell - 1)$ whose union gives the vertex cluster at level ℓ. Restricted topology trees are simpler to define and have a simpler structure than balanced topology trees: indeed nodes of restricted topology trees have degree at most two, while nodes in a balanced topology tree can have degree as large as four. It can be shown that both have height $O(\log N)$, where N is the total number of nodes in the topology tree.

As shown in [8, 9], the update of a (either balanced or restricted) topology tree because of an edge insertion, edge deletion or edge swap can be accomplished in $O(\log N)$ time. The idea is that there are a constant number of leaves affected, and they cause changes that percolate up in the topology tree, possibly causing vertex clusters in the multi-level partition to be regrouped in different ways. This is handled by rebuilding portions of the topology tree in a bottom-up fashion, and involves a constant amount of work to be done on at most $O(\log N)$ topology tree nodes.

Storing one topology tree for each cluster of the partition of order z, yields an $O(z + (m/z)\log(m/z)) = O((m \log m)^{1/2})$ time algorithm for maintaining a minimum spanning tree [8, 9]. We have two algorithms that achieve this bound: `FredII-85` that uses balanced partitions and balanced topology trees, and `FredII-91` that uses restricted partitions and restricted topology trees. In our experiments, `FredII-91` was slightly faster than `FredII-85`. This can be explained by noting that balanced topology trees are somehow more expensive to maintain than restricted topology trees, mainly due to their higher branching factor, and to the more complicated rules that define multi-level balanced partitions. Thus, the negative effect of balanced topology trees seemed to cancel the benefits of using balanced partitions rather than restricted partitions. It should be noted that in our experiments restricted topology trees had generally larger depth than balanced topology trees built on the same set of data. Thus, their better performance should be attributed to their simpler cluster formation rules.

A hybrid solution, working with a suitable combination of a balanced partition of order z with restricted topology trees, was faster than both `FredII-85` and `FredII-91`. However, even this hybrid solution was slower than `FredI-85` for most classes of graphs and update sequences. Our experiments considered graphs with as many as 149,000 edges, and thus, it would seem that even for those large graphs, the overhead introduced by topology trees would not pay off.

A *2-dimensional topology tree* for a topology tree is defined as follows. For every pair of nodes V_α and V_β at the same level in the topology tree there is a node labeled $V_\alpha \times V_\beta$ in the 2-dimensional topology tree. Let E_T be the tree edges of G (i.e., the edges in the spanning tree T): node $V_\alpha \times V_\beta$ represents all the non-tree edges of G (i.e., the edges of $E - E_T$) having one endpoint in V_α and the other in V_β. The root of the 2-dimensional topology tree is labeled $V \times V$ and represents all the non-tree edges of G. If a node is labeled $V_\alpha \times V_\beta$, and V_α has children V_{α_i}, $1 \leq i \leq p$, and V_β has children V_{β_j}, $1 \leq j \leq q$, in the topology tree, then $V_\alpha \times V_\beta$ has children $V_{\alpha_i} \times V_{\beta_j}$, $1 \leq i \leq p, 1 \leq j \leq q$, in the 2-dimensional topology tree. Using 2-dimensional topology trees yields time bounds of $O(z + m/z) = O((m)^{1/2})$ for the dynamic minimum spanning tree problem. This is the fastest theoretical bound achievable with any of Frederickson's algorithms. Not surprisingly, the benefit of 2-dimensional topology tree revealed to be mainly of theoretical interest, as the implementations that employed 2-dimensional topology trees were much slower than `FredI-85`.

3 Sparsification

Sparsification [5] 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*, i.e., it does not require the algorithm designer to know the internal details of the algorithm that he/she wants to speed up. 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* [5]. When simple sparsification is applied to $\mathcal{A}$, it produces a better bound of $O(f(n, O(n)) \log(m/n))$ for the same problem. This is achieved by means of a suitable graph decomposition into smaller subgraphs so that algorithm $\mathcal{A}$ 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 graph decomposition to eliminate the logarithmic factor, thus yielding a $O(f(n, O(n)))$ time bound. Sparsification is based on the concept of *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.

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 is represented by a sparse certificate. This tree has $O(m/n)$ leaves, and hence its height is $O(\log(m/n))$. 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 [5] for the full details of the method.

In our experiments, we used simple sparsification on top of `FredI-85`, with the certificate being the minimum spanning tree itself. 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. We called this algorithm `Spars(I-85)`. We also ran sparsification on top of `FredI-Mod`, which we referred to as `Spars(I-Mod)`.

4 Simple Algorithms

Let $G = (V, E)$ be a graph with m edges and n vertices. We first describe the algorithm that `LEDA` uses for computing the minimum spanning tree of a graph. Since this algorithm is basically an experimental tuning of Kruskal's algorithm, we describe Kruskal's algorithm first. Kruskal's algorithm consists of two phases. In the first phase, all the m edges of G are sorted in time $O(m \log m)$. We call this the *sorting phase*. In the second phase, the minimum spanning tree is computed by starting from an empty forest and by examining the edges of G in increasing order, as follows. After it is examined, an edge can be either *chosen* or *discarded*: at the end of this phase the set of chosen edges defines the minimum spanning tree. Let F be the set of edges chosen at any time (which always defines a forest), and let e be the edge to be examined next: if e induces a cycle in F, then e is discarded. Otherwise, edge e is chosen and the new forest will be $F \cup \{e\}$. Using set-union data structures [16] to test whether a new edge e will induce a cycle in F, yields a total time of $O(n + m\alpha(m, n))$ time for the second phase. We call this the *set-union phase*. Note that the algorithm has a worst-case running time of $O(m \log m)$, and its bottleneck is actually the sorting phase. This happens also in practice, and was confirmed by our experimental tests.

To speed this computation up, the minimum spanning tree algorithm of `LEDA` (called `Min_Spanning_Tree`) proceeds as follows. First, `Min_Spanning_Tree` computes in $O(m)$ time the K-th smallest edge f in G with a fast randomized algorithm for selection. Next, it computes in $O(m)$ the reduced set of edges $Y = \{e \in E \mid \text{cost}(e) \leq \text{cost}(f)\}$. Finally, `Min_Spanning_Tree` applies Kruskal's algorithm to the reduced edge set Y in time $O(K \log K)$. Denote by F be the forest obtained after this step. If F is connected or has the same number of connected components as G (which can be easily checked), then F is the minimum spanning tree (or forest) of G. Otherwise, `Min_Spanning_Tree` applies Kruskal's algorithm to G. Clearly, `Min_Spanning_Tree` has a worst-case running time of $O(m \log m)$, and thus it is not asymptotically more efficient than Kruskal's algorithm in the worst case. Actually, it is expected to be even slower than Kruskal's algorithm whenever Y fails to contain all of the minimum spanning tree edges of G. However, especially when the edge costs of G are uniformly chosen at random, this possibility is very low. In particular, with the help of the theory of random graphs [3], the following can be shown: For a suitable choice of K (namely $K = \Omega(n \log n)$), in case of random graphs the probability that `Min_Spanning_Tree` applies Kruskal's algorithm to the entire graph G is extremely low. Consequently, the choice of $K = \Theta(n \log n)$ yields an average running time of $O(m + n \log^2 n)$ for `Min_Spanning_Tree`. The `LEDA` `Min_Spanning_Tree` function actually chooses $K = 3n$, and our experimental tests gave evidence that with this choice, for random graphs LEDA's Min_Spanning_Tree almost never applied Kruskal's algorithm to the entire graph.

We now describe a simple fully dynamic algorithm, which we refer to as adhoc. This algorithm is a suitable combination of a partially dynamic data structure (based upon the linking and cutting trees of Sleator and Tarjan [15]) and Min_Spanning_Tree. Throughout the sequence of operations, adhoc maintains two data structures. First, it keeps the minimum spanning tree T of the graph G as a linking and cutting tree [15]. Second, it stores all the edges of G in a priority queue Q [4] according to their cost. When a new edge e is inserted into G, adhoc updates T and Q in time $O(\log n)$. When an edge e is deleted from G, it is first deleted from Q in $O(\log n)$ time. If e was a non-tree edge (this can be checked quickly by properly marking the tree edges), adhoc does nothing else. Otherwise, adhoc calls Min_Spanning_Tree on the edges in Q. Consequently, adhoc requires $O(\log n)$ time plus the running time of Min_Spanning_Tree in case of a tree edge deletion. If the edge to be deleted is chosen uniformly at random from G, this expensive case happens with probability n/m, yielding an average running time of $O(\log n + (n/m)(m + n\log^2 n)) = O(n + (n\log n)^2/m)$ for adhoc. One would thus expect that adhoc has very low implementation constants, and that its running time further decreases as the graph density increases. This was exactly confirmed by our experiments.

5 More Experimental Results

We now comment in more depth the results of our implementations. Most of our experiments were run on a DEC APX 4000/720 machine, with two 190 MHz CPU alpha, and 128 MBytes of RAM. This machine has a hierarchical memory architecture with a 4 MBytes cache external to the processor. We measured only the time that the algorithms spent in recomputing a new solution after each update (i.e., we did not measure the times required for loading the initial graphs and for initializing the data structures). All our times were CPU times measured in seconds with the Unix `getrusage()` command. We used the memory management of LEDA and, according to our experiments, the work done per given time was fairly independent from the system load.

In the following, we report the results of our experiments with the following algorithms:

FredI-85: The algorithm by Frederickson based on balanced partitions of order $\lceil m^{2/3}\rceil$. This algorithm has a worst-case running time of $O(m^{2/3})$. Following the analysis of [2], it can be shown that the average running time of FredI-85 is $O(n/m^{1/3})$.

FredI-Mod: Our variant of Frederickson's algorithm based on light partitions of order $\lceil m^{2/3}\rceil$ and on lazy updates.

Spars(I-85): Simple sparsification run on top of FredI-85, yielding a time bound of $O(n^{2/3}\log(m/n))$ per update in the worst case.

Spars(I-Mod): Simple sparsification run on top of FredI-Mod.

adhoc: Our combination of a partially dynamic algorithm (based on the linking and cutting trees of Sleator and Tarjan) and LEDA Min_Spanning_Tree algorithm. This algorithm has a worst-case running time of $O(m\log m)$ and an average-case running time of $O(\log n + (n\log n)^2/m)$.

Among the six algorithms of Frederickson taken from [8, 9], we choose to report our experiments with FredI-85 only, as this algorithm was consistently faster than the remaining five algorithms for almost all graph parameters and update sequences.

5.1 Random Inputs

We performed our tests on random inputs as follows. We first generated a large collection of data sets, consisting of ten samples each. There was a data set for each fixed value of the graph parameters. Each sample consisted of one initial randomly generated graph with n_0 nodes and m_0 edges, with edge costs chosen at random, plus one randomly generated sequence of σ updates for this graph. In our tests on random inputs, the updates were uniformly mixed, namely 50% were edge insertions and 50% were edge deletions, and the edge costs in the update sequence σ were chosen at random as well. We ran our implementations on each sample, and retained the average on the ten samples for each program. Figure 1 illustrates our experiments with random inputs and initial graphs of $n = 200$ vertices and different densities, and for evenly mixed random sequences of $2,000$ updates. Very similar results hold for different values of n and different lengths in the update sequence.

The fastest algorithm was adhoc, whose measured running time was clearly decreasing with the graph density. This phenomenon can be explained as follows. Recall that adhoc is very fast in all possible cases, except for deletions of tree edges, where it has to rely on Min_Spanning_Tree. In case of random inputs, however, the edge to be deleted is chosen uniformly at random among all the edges in the graph. Consequently, the probability of deleting a tree edge is at most n/m, where n is the number of vertices and m is the number of edges currently in the graph. For sparse graphs (say for graphs whose number of edges is roughly equal to the number of vertices), the probability of deleting a tree edge is thus very high, and frequent calls to Min_Spanning_Tree are expected. This was indeed the most expensive case for adhoc in our experiments. Whenever m increases, the probability of deleting a tree edge gets smaller, and the calls to Min_Spanning_Tree become less frequent. For m approaching $n^2/2$, the probability of calling Min_Spanning_Tree reduces to $O(1/n)$, and thus adhoc recomputes a minimum spanning tree quickly in $O(\log n)$ most of the times.

A similar phenomenon occurred for FredI-Mod and FredI-85, as it can be seen from Figure 1. Indeed, the most expensive operations for FredI-Mod and FredI-85 happen when the partition is forced to change. In case of FredI-85, this happens when an edge enters or leaves the current minimum spanning tree. The probability of deleting a tree edge or inserting an edge that will enter the minimum spanning tree is proportional to n/m, as shown in [2]. As the density of the graph increases, the probability of performing destructive updates in the partitions of FredI-85 becomes thus very small, and consequently the algorithm becomes faster. For FredI-Mod, in our experiments the decrease in the running times was less sharp as m increased. This seemed to be a consequence of the lazy update scheme used by FredI-Mod: delaying the updates in the partition had the effect of "smoothing" the unbalance between cheap and expensive operations.

We now turn to Spars(I-85) and Spars(I-Mod). For small graphs (for $m < 4n$), sparsification had basically the same behavior as the algorithm it was operating on. In this case indeed, the sparsification tree consisted of one single node, and sparsification basically coincided with the underlying algorithm. For $m = 4n$ there was a sharp increase in the running time of sparsification. This can be explained as follows. Whenever m/n is close to a power of 2, an evenly mixed sequence of updates would tend to produce a sparsification tree that is oscillating between two different depths. These frequent structural changes can be considered responsible for the sharp degrade of performance of sparsification in these cases.

Differently from the other algorithms we tested, Spars(I-85) and Spars(I-Mod) did not get advantage from higher densities of the underlying graphs. On the contrary, they suffered a decrease in performance. This is because operations that causes no changes in the minimum spanning tree are not necessarily the cheapest operations for sparsification. Indeed, sparsification works with a decomposition of the graph into a collection of smaller graphs: even though the minimum spanning tree of the entire graph does not change during an update, some of the smaller graphs in the decomposition may have to change their minimum spanning tree. Put in other words, sparsification "spreads" the information about the graph G into smaller sparse subgraphs and thus will never have to handle graphs of large densities. This is one of the strengths of sparsification for worst–case bounds, but it seems at the same time an inherent limit in case of random inputs.

Similar experiments were reported for graphs with different numbers of vertices. In all our experiments with random inputs (up to 700 vertices and 149,000 edges), FredI-Mod was consistently faster than FredI-85 and adhoc was the fastest algorithm. However, as the number of vertices increased, there were slightly different behaviors from the case of smaller graphs. Figure 2 illustrates our experiments with random inputs and large initial graphs of $n = 700$ vertices and different densities, and for evenly mixed random sequences of 2,000 updates. The first striking difference with Figure 1 is that in this experiment FredI-Mod was faster than adhoc in case of sparse graphs, say graphs with less than 3,000 edges. After this point, adhoc decreased sharply its running time and became superior. Another difference with Figure 1 was that in this case the running times of both FredI-85 and FredI-Mod did not decrease with m, as one would expect from theory. On the contrary, we actually measured an increase in their running times for large m. This is probably due to the fact that these algorithms required a much larger amount of memory than adhoc: thus, they were more exposed to losses in performance as the size of the graphs increased.

In all of our experiments, except in case of sparse graphs on a large number of vertices, adhoc was the fastest algorithm. This did not come unexpected as adhoc is a very simple method, with low implementation constants: furthermore, in most cases it recomputes quickly the minimum spanning tree, and as the graphs get larger, the expensive computations become very rare. We believe that all these features made adhoc one of the most natural candidates as a practical algorithm on random inputs. The surprise for us was to realize that in some cases our variant of Frederickson's algorithm was actually the fastest algorithm among our implementations.

5.2 Non–Random Inputs

We wanted to test our implementations also on structured, non–random inputs, which tried to force bad update sequences on the algorithms. We did these experiments as follows. We generated a graph G, computed its minimum spanning tree T, and then deleted the edges of T, one at the time. These are likely to be very bad update sequences, as it is well known that the hard operation in the dynamic minimum spanning tree problem is the deletion of a minimum spanning tree edge: indeed, all the other updates can be easily handled in $O(\log n)$ time in the worst case. As expected, **adhoc** was terribly hit by these update sequences, and its running time was much bigger than the other algorithms. We thus decided not to include the experimental data of **adhoc** in our figures about non-random inputs.

Figure 3 shows the results of some of our experiments with these update sequences. In this particular experiment, we choose graphs with n vertices and $60n$ edges. For $n \leq 300$, this resulted in dense graphs ($m \geq n^2/5$). For $400 \leq n \leq 700$, the graphs were of intermediate density ($6n \log n \leq m \leq 7.5n \log n$). The fastest algorithm was `Spars(I-Mod)`, followed by `FredI-Mod`, `Spars(I-85)` and finally `FredI-85`. We only comment this experiment here, as similar results were obtained for other experiments that tried to enforce bad update sequences.

Although the theoretical running time of both `FredI-85` and `Spars(I-85)` is $O(n^{2/3})$ on these particular inputs ($m = 60n$), `Spars(I-85)` was actually much faster than `FredI-85`. This shows that the speed-up introduced by sparsification can be substantial in worst-case scenarios, even for graphs of intermediate densities. The superiority of `Spars(I-85)` can be explained as follows. Each tree edge deletion forces `FredI-85` to update the balanced partition. If the edge to be deleted is inside a cluster, this is particularly expensive, as this cluster must be split and might trigger a sequence of other merges and splits. This is a highly destructive update, and therefore it is very time consuming. As sparsification spreads updates on a logarithmic number of smaller graphs (and thus on their internal balanced partitions), it seems less vulnerable to this kind of destructive updates.

What came as a surprise to us was the efficiency of `FredI-Mod`, which was consistently faster than `FredI-85` (and even faster than `Spars(FredI-85)`!) even for non-random inputs. We notice first that the lazy update scheme is not operational during a sequence consisting of tree edge deletions only. Thus, all the speed-up of `FredI-Mod` over `FredI-85` is exclusively due to the more relaxed nature of light partitions. Our experiments confirmed indeed that the sequences of cluster splits and merges triggered by destructive updates were substantially shorter on light partitions rather than on balanced partitions. The fastest algorithm was `Spars(FredI-Mod)`, except for the case $n = 200$ where `FredI-Mod` was slightly faster. Note that the speed-up of sparsification over `FredI-Mod` was not as good as the one over `FredI-85`. This is perhaps another indication of the practical value of light partitions: in our experience they were simple to implement, and seemed efficient for random inputs as well as robust to bad update patterns.

References

[1] D. Alberts, G. Cattaneo, G. F. Italiano, "An empirical study of dynamic graph algorithms", *Proc. 7th ACM-SIAM Symp. on Discrete Algorithms* (1996), 192-201.

[2] D. Alberts, M. R. Henzinger, "Average Case Analysis of Dynamic Graph Algorithms", *Proc. 6th Symp. on Discrete Algorithms* (1995), 312-321.

[3] B. Bollobás. *Random Graphs*. Academic Press, London, 1985.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. Mac-Graw Hill, NY, 1990.

[5] D. Eppstein, Z. Galil, G.F. Italiano, and A. Nissenzweig, "Sparsification—A technique for speeding up dynamic graph algorithms", *Proc. 33rd IEEE Symp. on Foundations of Computer Science* (1992), 60-69. A full version is available.

[6] 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.

[7] 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.

[8] G.N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications", *SIAM J. Comput.* 14 (1985), 781-798.

[9] 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.

[10] 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.

[11] M. R. Henzinger and V. King, Fully dynamic biconnectivity and transitive closure, *Proc. 36th IEEE Symp. Foundations of Computer Science*, pages 664-672, 1995.

[12] J. B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, *Proc. Amer. Math. Soc.* 7, (1956), 48-50.

[13] K. Mehlhorn, S. Näher, "LEDA, A platform for combinatorial and geometric computing", *Comm. ACM*, (1995), 38(1): 96-102.

[14] M. Rauch, "Improved data structures for fully dynamic biconnectivity", *Proc. 26th Symp. on Theory of Computing* (1994), 686-695.

[15] D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *J. Comp. Syst. Sci.*, 24:362-381, 1983.

[16] R. E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.*, 31:245-281, 1984.

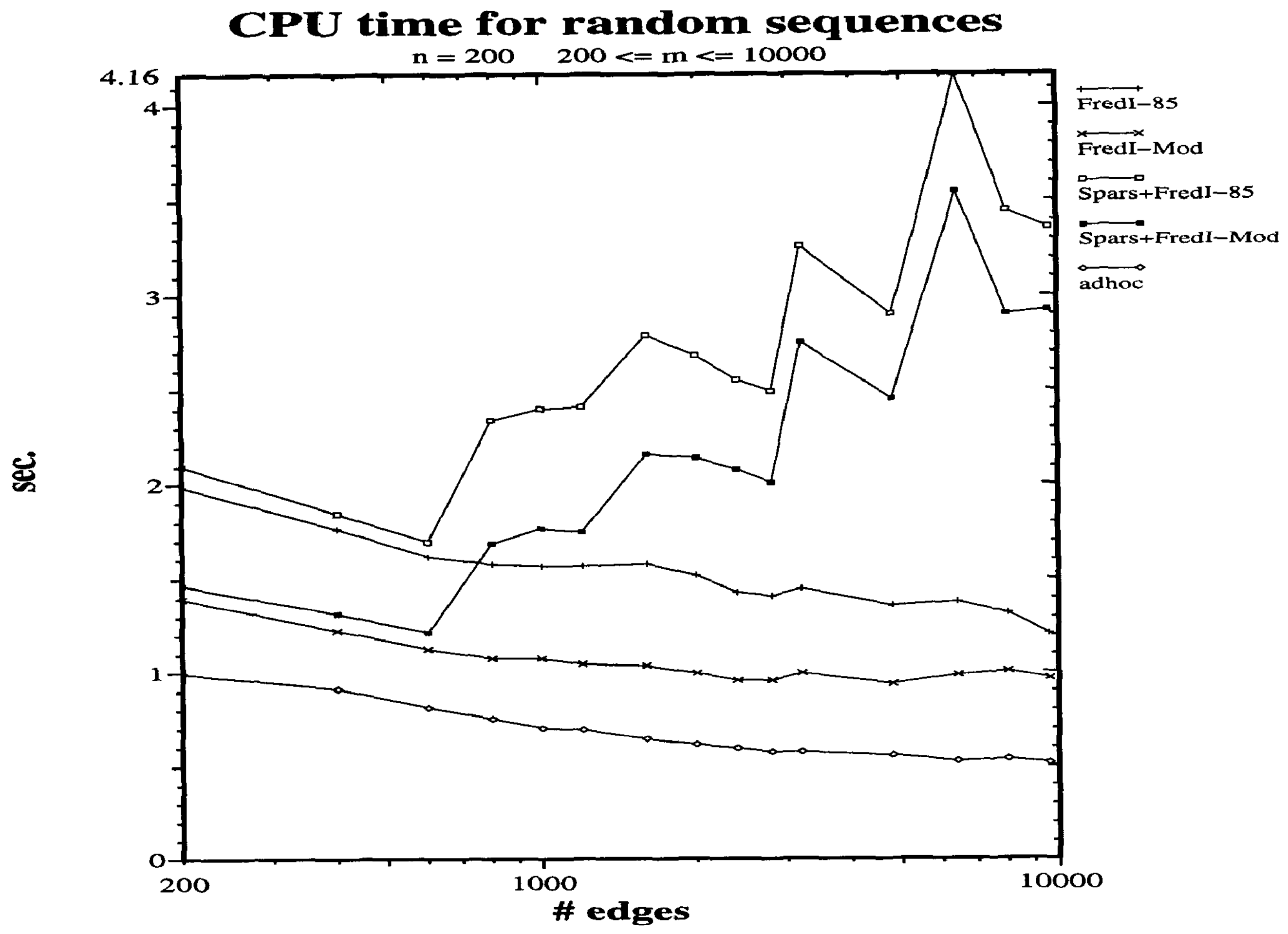

Figure 1: adhoc, FredI-Mod, FredI-85, Spars(I-Mod) and Spars(I-85) running on the same sequences of 2000 evenly mixed random updates starting from the same initial random graphs with 200 vertices. Each data set is the average of ten different samples.

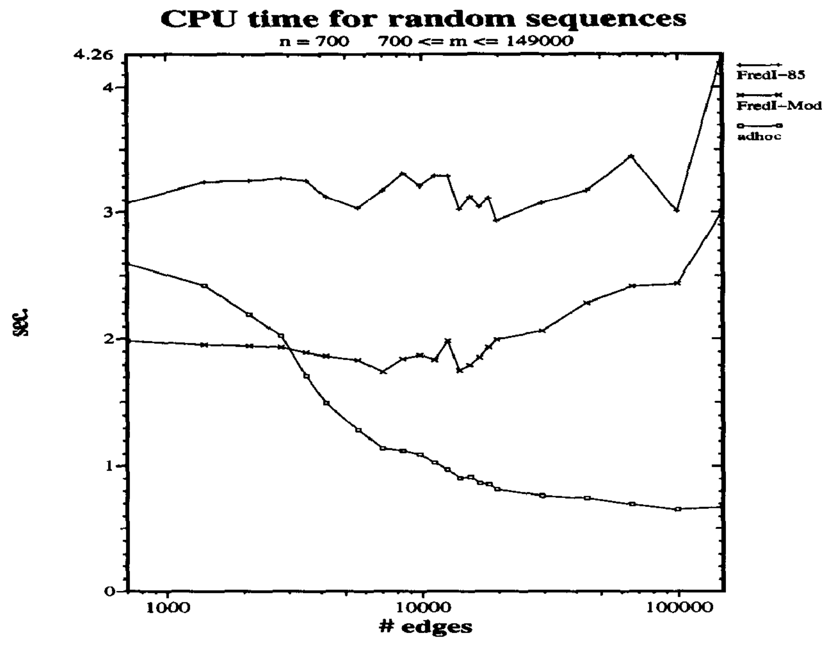

Figure 2: **adhoc**, **FredI-Mod**, **FredI-85**, **Spars(I-Mod)** and **Spars(I-85)** running on the same sequences of 2000 evenly mixed random updates starting from the same initial random graphs with 700 vertices. Each data set is the average of ten different samples.

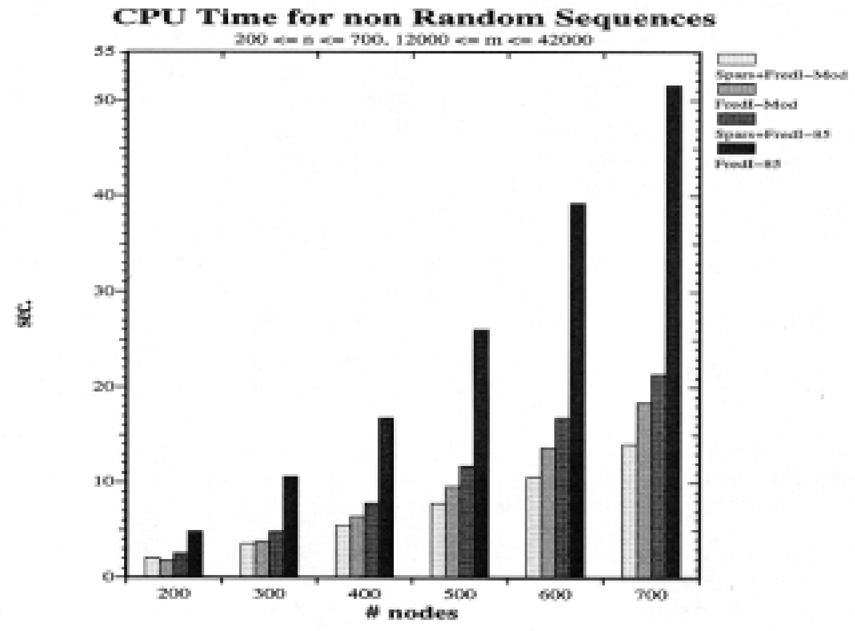

Figure 3: **FredI-Mod**, **FredI-85**, **Spars(I-Mod)** and **Spars(I-85)** running on the same sequences of bad updates starting from the same initial graphs with n vertices. Updates are deletions of tree edges.