Alcom-FT Technical Report Series
ALCOMFT-TR-02-69

1. Implementations and Experimental Studies of Dynamic Graph Algorithms

Christos D. Zaroliagis

Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece
and
Department of Computer Engineering & Informatics
University of Patras, 26500 Patras, Greece
Email: zaro@ceid.upatras.gr

Summary.
Dynamic graph algorithms have been extensively studied in the last two decades due to their wide applicability in many contexts. Recently, several implementations and experimental studies have been conducted investigating the practical merits of fundamental techniques and algorithms. In most cases, these algorithms required sophisticated engineering and fine-tuning to be turned into efficient implementations. In this paper, we survey several implementations along with their experimental studies for dynamic problems on undirected and directed graphs. The former case includes dynamic connectivity, dynamic minimum spanning trees, and the sparsification technique. The latter case includes dynamic transitive closure and dynamic shortest paths. We also discuss the design and implementation of a software library for dynamic graph algorithms.

1.1 Introduction

The traditional design of graph algorithms usually deals with the development of an algorithm that, given a static (fixed) graph G as input, solves a particular problem on G; for example, "is G connected?". A *dynamic graph*, on the contrary, is a graph which may evolve with time due to local changes that occur in G; e.g., insertion of a new edge or deletion of an edge. The challenge for an algorithm dealing with a dynamic graph is to maintain, in an environment of dynamic local changes, a desired graph property (e.g., connectivity) efficiently; that is, without recomputing everything from scratch after a dynamic change. Dynamic graphs are usually more accurate models than static graphs, since most real systems (e.g., physical networks) are not truly static.

A *dynamic algorithm* is a data structure that allows two types of operations: queries and updates. A query asks for a certain property P of the current graph G (e.g., "are vertices x and y connected in G?"), while an update operation reflects a local change in G. Typical changes include insertion of a new edge and deletion of an existing edge. An algorithm or a problem is called *fully dynamic* if both edge insertions and deletions are allowed, and it is called *partially dynamic* if either edge insertions or edge deletions are allowed (but not both). In the case of edge insertions (resp. deletions), the partially dynamic algorithm or problem is called *incremental* (resp. *decremental*).

The main goal of a dynamic algorithm is to use structural properties of the current graph G in order to handle updates efficiently, i.e., without resorting to the trivial approach of recomputing P from scratch using a static algorithm. In most cases, updates take more time than queries, and the sequence of operations (updates and queries) is provided in an on-line fashion (i.e., the operations are not known in advance).

Dynamic graph algorithms have been an active and blossoming field over the last years due to their wide applicability in a variety of contexts, and a number of important theoretical results have been obtained for both fully and partially dynamic graph problems. These results show a clear distinction between problem solving in undirected and in directed graphs: maintaining a property (e.g., connectivity) in a directed graph turns out to be a much more difficult task than maintaining the same property on an undirected graph. There is a bulk of striking results and novel techniques for undirected graphs which cannot however be transferred to directed graphs.

The challenge for dynamic algorithms to beat their (usually very efficient) static counterparts as well as the fact that their input is more complicated than the input of the corresponding static algorithms, has sometimes led to the development of rather sophisticated techniques and data structures. This, however, makes their practical assessment a non-trivial task, since the actual running times may depend on several parameters that have to do with the size and type of input, the distribution of operations, the length of the operation sequence, the update pattern in the operation sequence, and others.

Hence, it is inevitable to perform a series of experiments with several dynamic algorithms in order to be able to select the most appropriate one for a specific application. On the one hand, this experimentation often requires sophisticated engineering and fine-tuning to turn theoretically efficient algorithms into efficient implementations. On the other hand, the conducted experiments give useful insight which can be used to further improve the algorithms and the implementations.

Experimentation, however, requires proper selection of the test sets on which the implemented dynamic algorithms will be assessed, i.e, the test set should be as complete as possible. This in turn implies that both unstructured (i.e., random) and structured inputs should be considered. The former is important to either confirm the average-case analysis of an algorithm, or (if such an analysis does not exist) to understand its average-case performance. The latter is equally important as it either provides more pragmatic inputs (inputs originated from or motivated by real-world applications), or provides worst-case inputs, i.e., inputs which will enforce an algorithm to exhibit its worst-case performance. Random inputs are usually easier to generate than structured inputs, while generation of worst-case inputs is perhaps the most difficult as it depends on several factors (problem, algorithm, etc).

In this paper, we survey several implementations along with their experimental studies for dynamic problems on undirected and directed graphs. The former case includes dynamic connectivity, dynamic minimum spanning trees, and the sparsification technique. The latter case includes dynamic transitive closure and dynamic shortest paths. We also discuss the design and implementation of a software library for dynamic graph algorithms. All but one of the implementations have been done in `C++` using the LEDA platform for combinatorial and geometric computing [1.52].

To give a better picture on how the implementations stand in relation with the algorithms known from theory, the treatment of each dynamic problem starts by presenting first the known theoretical results and then discussing the available implementations, commenting on the data sets used, and concluding with lessons learned.

1.2 Dynamic Algorithms for Undirected Graphs

The implementation studies known for dynamic problems on undirected graphs concern dynamic connectivity and minimum spanning tree. For the rest of this section, $G = (V, E)$ represents an undirected graph with n vertices and m edges, unless stated otherwise.

1.2.1 Dynamic Connectivity

1.2.1.1 Theoretical background – problem and history of results. In the dynamic connectivity problem, we are interested in answering *connectivity queries* in a graph G which undergoes a sequence of updates (edge insertions and edge deletions). Given any two vertices x and y, a connectivity query asks whether there is a path in G between x and y. The dynamic connectivity problem reduces to the problem of maintaining a spanning forest in G, i.e., maintaining a spanning tree for each connected component of G. Dynamic connectivity was studied both in a fully and in a partially dynamic setting.

The first algorithm for fully dynamic connectivity was given by Harel [1.35]; it supported queries in $O(1)$ time and updates in $O(n \log n)$ time. Frederickson [1.25] reduced this update bound to $O(\sqrt{m})$. This was later improved by Eppstein et al. [1.19] to $O(\sqrt{n})$ through the use of a very simple but powerful technique called sparsification, which is a general method for producing dynamic algorithms. Further improvements came with the use of randomization. The first such algorithm (of Las-Vegas type) was presented by Henzinger & King [1.36] achieving $O(\log^3 n)$ expected amortized time for updates and $O(\log n / \log \log n)$ time for queries. The expected amortized update time was subsequently improved to $O(\log^2 n)$ by Henzinger & Thorup [1.39]. At about the same time, Nikoletseas et al. [1.55] presented a fully dynamic, probabilistic (Monte-Carlo), connectivity algorithm for random graphs and random update sequences which also achieves $O(\log^3 n)$ expected amortized time for updates, but answers queries in $O(1)$ expected amortized time. The need for randomization was removed by Holm et al. [1.40]; in that paper a deterministic algorithm for fully dynamic connectivity is presented which achieves $O(\log^2 n)$ update time and $O(\log n/\log\log n)$ query time. Very recently Thorup [1.65] presented a new randomized (Las-Vegas) fully dynamic algorithm with $O(\log n(\log\log n)^3)$ expected amortized time for updates and $O(\log n/\log\log\log n)$ time for queries. It is worth noting that the above polylogarithmic upper bounds for updates and queries are not far away from the currently best lower bound of $\Omega(\log n/\log\log n)$ [1.27, 1.53] for both operations. All the above algorithms with polylogarithmic update and query time require $O(m + n\log n)$ preprocessing time and space. Thorup also showed in [1.65] that the space bound of the algorithms in [1.40, 1.65] can be reduced to $O(m)$.

For partially dynamic connectivity, there are only two algorithms which achieve better results than those provided by the fully dynamic ones: an incremental algorithm based on Tarjan's union-find data structure [1.62] which achieves an amortized time of $O(\alpha(m,n))$ per update or query operation; a decremental randomized algorithm due to Thorup [1.64] which supports queries in $O(1)$ time and any number of edge deletions in a total $O(\min\{n^2, m\log n\} + \sqrt{nm}\log^{2.5} n)$ expected time, where m denotes the initial number of edges. This is $O(1)$ amortized expected time per operation if $m = \Omega(n^2)$.

1.2.1.2 Implementations and Experimental Studies. There are three works known regarding implementation and experimental studies of dynamic connectivity algorithms. In chronological order, these are the works by Alberts et al. [1.3], Fatourou et al. [1.22], and Iyer et al. [1.44].

The first paper investigates the practicality of sparsification-based approaches and their comparison to the randomization-based approach by Henzinger & King [1.36]. The second paper enhances this study by investigating the comparison between two randomized approaches, the one by Henzinger & King [1.36] and the other one by Nikoletseas et al. [1.55]. Finally, the third paper brings the algorithm by Holm et al. [1.40] into play and aims at investigating in practice the difference in the logarithmic improvement over the algorithm of [1.36]. Moreover, that study considerably enhances the data sets used in the experiments.

1.2.1.2.a ***The Implementation by Alberts et al.*** [1.3]

The main goal of the first experimental study for dynamic connectivity algorithms was threefold:

1. To investigate the practicality of dynamic algorithms over static ones (especially to very simple and easily implementable static algorithms).
2. To investigate the practicality of the sparsification technique and confirm the theoretical analysis regarding its average-case performance.

3. To compare dynamic algorithms based on sparsification with other dynamic algorithms and especially with the algorithm by Henzinger & King [1.36] which is based on randomization.

Sparsification is a simple and elegant technique which applies to a variety of dynamic graph problems. It can be used either on top of a static algorithm in order to produce a dynamic one, or on top of a dynamic algorithm in order to speed it up. Sparsification works as follows. The edges of G are partitioned into $\lceil m/n \rceil$ sparse subgraphs, called *groups*, each one having n edges. The remaining group, called the *small group*, contains between 1 and n edges. The information relevant for each subgraph (e.g., connectivity) is summarized in an even sparser subgraph called a *sparse certificate* (e.g., a spanning forest). In a next step certificates are merged in pairs yielding larger subgraphs which are made sparse by computing again their certificate. This step is applied recursively resulting in a balanced binary tree, called *sparsification tree*, in which each node represents a sparse certificate. Since there are $\lceil m/n \rceil$ leaves, the sparsification tree has height $\lceil \log(m/n) \rceil$. When an edge is inserted, it is placed in the small group; if there are already n edges in this group, then a new small group is created. When an edge is deleted, it is removed from the group to which it belongs and an edge from the small group is moved to the group which contained the deleted edge. If the last edge of the small group is deleted, the small group is removed. Consequently, an update operation (edge insertion/deletion) involves some changes to a $O(1)$ number of groups plus the examination of the sparse certificates (ancestors of the modified groups) in a $O(1)$ number of leaf-to-root tree paths. This in turn implies the examination of $O(\log(m/n))$ subgraphs of $O(n)$ edges each, instead of considering one large graph with m edges. This immediately speeds up an $f(n,m)$ time bound (representing either the time of a static algorithm or the update bound of a dynamic algorithm) to $O(f(n, O(n)) \log(m/n))$ and is called *simple sparsification*. The logarithmic factor of the previous bound can be eliminated with the use of more sophisticated graph decomposition and data structures resulting in the so-called *improved sparsification* (see [1.19] for the details).

Simple sparsification comes into three variants, depending on whether the certificates are recomputed by a static, fully dynamic, or partially dynamic algorithm. We shall keep the term *simple sparsification* for the first and third variants, since the second variant requires that certificates obey a so-called stability property and hence it is referred to as *stable sparsification*.

The simple sparsification was implemented in [1.3]. To achieve better running times, a few changes w.r.t. the original algorithm were introduced in the implementation:

(i) A queue keeps track of edge deletions in the groups. Namely, when an edge is deleted from a group, a pointer to that group is inserted in the queue (i.e., the queue represents "empty slots" in groups). When an edge is inserted, the first item is popped from the queue and the edge is inserted into the corresponding group. If the queue is empty, then the new edge is inserted as in the original algorithm (i.e., either in the small group, or in a new small group). As a consequence, the deletion of an edge needs no swapping of edges and involves the examination of only one leaf-to-root path. According to the experiments in [1.3], this modification yields roughly 100% speedup for edge deletions.

(ii) The above implementation may however impoverish the edge groups: an update sequence with less insertions than deletions may invalidate the group size invariant (i.e., some group may have less than n edges) and result in a sparsification tree with height larger than $\lceil \log(m/n) \rceil$. To confront this situation, the sparsification tree is rebuilt each time its actual height differs by more than one from its "correct" height of $\lceil \log(m/n) \rceil$.

(iii) During an update, not all certificates in the leaf-to-root path are recomputed. Recomputation stops at the first tree node whose certificate remains unaffected by the update, since all its ancestors will not be affected as well. This introduces significant time savings on the average, and it is also matched by a theoretical analysis [1.3] which shows that on the average the number of sparsification nodes affected by an update is bounded by a small constant.

For the dynamic connectivity problem, simple sparsification was implemented on top of an incremental algorithm (based on the `Spanning_Tree` function of LEDA) which supports edge insertions in $O(\alpha(m,n))$ time and recomputes the solution from scratch after an edge deletion in $O(n+m\alpha(m,n))$ time. This results in an update time of $O(n\alpha(n,n)\log(m/n))$. The resulting implementation is called `Sparsification`.

The second algorithm implemented in [1.3] was the fully dynamic algorithm of Henzinger & King [1.36], henceforth the HK algorithm. The algorithm maintains a spanning forest F of the current graph G. Connectivity queries are answered by checking whether two given vertices belong to the same tree of the forest. As edges are inserted or deleted, the forest is maintained so that it is kept spanning. Hence, a data structure is required which performs efficiently the operations of joining two trees by an edge, splitting a tree by deleting an edge, and checking whether two vertices belong to the same tree. In [1.36] a data structure called *Euler-tour trees* (ET-trees) is introduced for this purpose. An ET-tree is a standard balanced binary tree over the Euler tour of a tree and supports all the above operations in $O(\log n)$ time. The key idea is that when trees are cut or linked, the new Euler tours can be constructed by at most 2 splits and 2 concatenations of the original Euler tours, while rebalancing of ET-trees affects only $O(\log n)$ nodes.

Maintaining the spanning forest F of G using the ET-trees yields immediately a very efficient way to handle connectivity queries (obvious) and edge insertions: when an edge is inserted, check whether its endpoints belong to the same tree of the forest. If yes, do nothing; otherwise, insert it in the forest by simply joining the two ET-trees it connects. Both operations can be accomplished in $O(\log n)$ time. Edge deletions, however, require some additional care. If the deleted edge e is not a tree edge, then it is simply discarded. Otherwise, it causes the tree T to which it belongs to split into two other trees T_1 and T_2. In order to maintain the invariant that the forest is spanning, we have to check whether there is a non-tree edge (an edge which does not belong to any tree) that rejoins T_1 and T_2. Such an edge, if exists, is called *replacement edge*. Consequently, a dynamic connectivity algorithm must find replacement edges quickly. Henzinger & King [1.36] use two nice ideas to achieve this. The first is to use random sampling among the (possibly many) non-tree edges incident to T. However, the set of edges that rejoin T, called the candidate set, may be a small fraction of the non-tree edges adjacent to T and hence it is unlikely to find a replacement edge for e among the sampled ones. Since examining all non-tree edges adjacent to T is undesirable, another approach is required to deal with such a situation. Here comes the second idea of [1.36]: maintain a partition of the edges of G into $O(\log n)$ levels, forming $O(\log n)$ edge disjoint subgraphs $G_i = (V, E_i)$ of G, $1 \leq i \leq l = O(\log n)$. The partition is done in a way such that edges in highly-connected parts of the graph are on upper levels while edges in loosely-connected parts are at lower levels[1]. For each level i, a spanning forest F_i is maintained for the graph whose edges are in levels $j \geq i$. If a tree edge e at level i is deleted, then the non-tree edges in the smaller subtree, say T_1, are sampled. If within $O(\log^2 n)$ samples a replacement edge is found, we are done. Otherwise, the cut defined by the deletion of e is too sparse for level i (i.e., the vast majority of the non-tree edges incident on T_1 have both endpoints in T_1). In such a case, all edges crossing the cut are copied to level $i-1$ and the procedure is applied recursively on level $i-1$. Since edge insertions can cause the number of levels to increase beyond $O(\log n)$, the HK algorithm periodically rebuilds its data structure such that there are always $O(\log n)$ levels. The implementation of the above algorithm in [1.3] is referred to as `HK`.

A simplified version of the HK algorithm was also implemented in [1.3] and is referred to as `HK-var`. This version was motivated by experiments with random inputs which showed that it is very unlikely that edges move to lower levels. Hence, in the simplified version of the HK algorithm there is only one level and only $O(\log n)$ edges – instead of $O(\log^2 n)$ – are sampled. In this version, queries, edge insertions, and non-tree edge deletions still take $O(\log n)$ time, but the deletion of a tree edge may take $O(m \log n)$ worst-case time. Despite the latter, the simplified version was always faster than the original algorithm on random inputs and was more robust to input variations.

Finally, two pseudo-dynamic algorithms were implemented to provide a point of reference in the sense that they are the simplest possible methods

[1] The level notation here is the inverted version of the original algorithm in [1.36], in order to facilitate comparison with the forthcoming algorithm of [1.40].

one could resort to, using static approaches. Since these algorithms require only a few lines of code, their constants are expected to be rather low and hence likely to be fast in practice for reasonable inputs. These two algorithms are called `fast-update` and `fast-query`. The former spends only $O(1)$ time on updates (just updates two adjacency lists), but answers queries in $O(n+m)$ time using a BFS algorithm. The latter maintains a spanning forest and component labels at the vertices. Hence, a query takes $O(1)$ time (just checks equality of component labels). An update operation may force the current forest to change in which case the forest and the component labels are recomputed from scratch taking $O(n+m)$ time.

All the above implementations, `Sparsification`, `HK`, `HK-var`, `fast-update` and `fast-query` were compared experimentally in [1.3] on various types and sizes of graph inputs and operation sequences (updates intermixed with queries). Experiments were run both on random inputs (random graphs and operation sequences) as well as on non-random inputs (non-random graphs and operation sequences) representing worst-case inputs for the dynamic algorithms.

Random inputs are particularly important in the study of [1.3]. Recall that one of the main goals was to investigate the average-case performance of sparsification (and of the other algorithms), since in [1.3] the average-case running time of simple sparsification is analyzed and it is proved that the logarithmic overhead vanishes (the number of nodes affected by an update in the sparsification tree is bounded by a constant). The random inputs consisted of random graphs with different edge densities ($m \in \{n/2, n, n \ln n, n^{1.5}, n^2/4\}$). Note that the first three values constitute points where (according to random graph theory [1.9]) a radically different structural behaviour of the graph occurs: if $m \approx n \ln n$, the graph is connected (with high probability); if $n < m < n \ln n$, the graph is disconnected, has a so-called giant component of size $\Theta(n)$, and smaller components of size $O(\ln n)$ at most; if $m \approx n$, then the giant component has size $\Theta(n^{2/3})$; and if $m < n$, then the largest component has size $O(\ln n)$. The update sequences consisted of an equal number of m insertions, m deletions, and m queries, each one uniformly distributed on the candidate set. The candidate set for deletions was the set of current edges, for insertions the set of current non-edges w.r.t. the set of all possible edges, and for queries the set of all vertex pairs.

Non-random inputs aim at establishing a benchmark for inputs that could force dynamic connectivity algorithms to exhibit their worst-case performance. The non-random inputs consisted of structured graphs and operation sequences. A structured graph consists of a number k of cliques, each one containing roughly n/k vertices, and which are interconnected by $k-1$ inter-clique edges, called "bridges". The dynamic operations are only (random) insertions and deletions of bridges. As bridges are tree edges, constant insertion and deletion of them will cause the algorithms to constantly look for replacement edges. Clearly, this kind of input represents a worst-case input for dynamic connectivity algorithms.

The first issue investigated was whether the theoretical analysis for the average-case performance of simple sparsification is confirmed in practice and whether `Sparsification` is better than the static algorithm on top of which it runs. The latter turned out to be true. The former was true for unweighted graphs (i.e., for problems like dynamic connectivity), but in the case of weighted graphs (e.g., for problems like dynamic minimum spanning tree) the experimental results did not always comply with the theoretical analysis, implying that perhaps a different model of analysis is required for such a case.

Regarding the comparison among the dynamic algorithms and the simple (pseudo-dynamic) ones, the reported experiments were as follows. For random inputs, `HK-var` was the fastest (as expected from the theoretical analysis), except for very sparse graphs ($m < n$) where `fast-query` was better. For non-random inputs, `Sparsification` and `HK` were better than the other algorithms; for large sequences of updates `HK` is faster, while for shorter sequences `Sparsification` is faster (the larger the update sequence, the better becomes the amortization in the HK algorithm). The behaviour of sparsification is due to the fact that it spreads the connectivity information in a logarithmic number of small subgraphs that have to be updated even if a local change does not affect the connectivity of the graph, i.e., tree and non-tree edge deletions produce roughly the same overhead. This turns out to be advantageous in the case of non-random graphs.

Another major conclusion of the study in [1.3] was that both sparsification and the HK algorithm proved to be really practical as they compare favorably to the simple algorithms even in the case of very small graphs (e.g., graphs with 10 vertices and 5 edges).

The source code of the above implementations is available from `http://www.jea.acm.org/1997/AlbertsDynamic`.

1.2.1.2.b ***The Implementation by Fatourou et al.*** [1.22]

The main goal of that study was to compare in practice the average-case performance of the fully dynamic, probabilistic algorithm by Nikoletseas et al. [1.55], henceforth NRSY, with the HK algorithm that appears to have a similar update bound and was also (along with the `fast-query`) among the fastest implementations for random inputs in the previous study [1.3].

The NRSY algorithm is different from the HK algorithm. It alternates between two epochs, the *activation* epoch and the *retirement* epoch, while it periodically performs *total reconstructions*, i.e., it rebuilds its data structure from scratch. A total reconstruction is called successful if it achieves in finding a giant component of size $\Omega(n)$ of the input random graph. The algorithm starts with a total reconstruction and builds a spanning forest of the graph. An activation epoch starts after a successful total reconstruction and ends when an edge deletion disconnects a spanning tree of the giant component and the attempted fast reconnection fails. A retirement epoch starts either after an unsuccessful total reconstruction, or after the end of an activation epoch.

An *activation epoch* maintains a spanning forest of the graph and partitions the edges into three categories: *tree* edges, *retired* edges, and *pool* edges (which are reactivated retired edges). The activation epoch is divided into edge deletion intervals each consisting of $O(\log n)$ deletions. All edges marked retired during an edge deletion interval are re-marked as *pool* edges after the end of the next interval. An edge insertion during an activation epoch is performed as follows. If the inserted edge joins vertices in the same tree, then it is marked as *retired* and the appropriate data structures are updated; otherwise, the edge joins vertices of different trees and the component name of the smaller tree is updated. Edge deletions during an activation epoch are handled as follows. If the deleted edge is a pool or a retired edge, then it is simply deleted from all data structures it belongs. If the deleted edge is a tree edge, then we distinguish between two cases depending on whether this was an edge of the giant component or not. In the latter case, we look for a replacement edge and if the search is not successful, the tree is split and the smaller of the two resulted trees is relabeled. In the former case, a special procedure, called NeighborhoodSearch, is applied which performs two breadth-first searches (one in each tree) in tandem in an attempt to reconnect the tree of the giant component. The breadth-first searches (BFSs) stop either when the tree is reconnected or as soon as $O(\log n)$ vertices have been visited. NeighborhoodSearch proceeds in phases, where each phase starts when a vertex is visited during BFS. During a phase, an attempt is made to find a replacement edge by checking whether a randomly chosen pool edge (if such an edge exists) incident on the visited vertex reconnects the tree of the giant component and whether this reconnection is a "good" one (i.e., it does not increase the diameter of the tree). If both checks are successful, then the phase is considered successful and also in turn the NeighborhoodSearch. If all phases fail, then NeighborhoodSearch finishes unsuccessfully, the activation epoch ends, and a total reconstruction is executed. If more than one of the phases are successful, then the replacement edge which is closer to the root of the tree is selected.

A *retirement epoch* starts when the previous activation epoch ends, or when a total reconstruction fails. During the execution of a retirement epoch, the algorithm simply calls another dynamic connectivity algorithm to perform the operations. A retirement epoch lasts for (at least) $cn \log^2 n$ operations ($c > 1$), after which a total reconstruction is performed. If the total reconstruction is successful, then a new activation epoch is started; otherwise, the retirement epoch continues for another $cn \log^2 n$ operations. The process is repeated until a successful total reconstruction occurs.

In the implementation of the NRSY algorithm, henceforth `NRSY`, the `HK` implementation of [1.3] was used in the retirement epochs. In the activation epoch, each vertex maintains both a set of pool edges and a priority queue of retired edges incident to it. A counter measuring the number of operations in each epoch is maintained. When an edge becomes retired, its priority takes the value of this counter. Reactivation of retired edges occurs only before the deletion of a tree edge, i.e., before the execution of the NEIGHBORHOOD-SEARCH, which is the only procedure for which it is important that several pool edges must exist. Reactivation is performed if the difference of the priority of a retired edge and the current value of the operations counter is larger than $\log n$.

The experiments conducted in [1.22] concentrated on random inputs, since the main goal was to investigate the average-case performance of the NRSY algorithm which guarantees good performance only for these kinds of inputs. The random inputs considered were similar to those generated in [1.3] (cf. Section 1.2.1.2.a). The experiments showed that for long sequences of operations, `NRSY` is better than `HK` and `fast-query`, except for the case where the initial number of edges is small (i.e., $m < n$). For medium sequences of operations, `NRSY` and `HK` perform similarly when m is close to n, but `NRSY` is better as the graph becomes denser. Finally, for short sequences, `HK` and `fast-query` outperform `NRSY`, except for the case of non-sparse graphs. The above behaviour is due to the fact that `NRSY` spends most of its time on activation epochs as the graph becomes denser, while in sparser graphs `NRSY` alternates between retirement epochs and total reconstructions; the overhead imposed by the latter makes `NRSY` slower than `HK` in such cases. In conclusion, `NRSY` is the fastest implementation for random sequences of updates on non-sparse random graphs.

The source code of the above implementations is available from `http://www.ceid.upatras.gr/~faturu/projects.htm`.

1.2.1.2.c *The Implementation by Iyer et al.* [1.44]

The main goal of that paper was to investigate in practice the logarithmic improvement of the fully dynamic algorithm by Holm et al. [1.40], henceforth HDT, over the HK algorithm. To this end, the experimental study built upon the one by Alberts et al. [1.3] and resulted in enhancing the data sets for dynamic connectivity in several ways.

The HDT algorithm maintains a spanning forest F of the current graph G. HDT has many similarities with the HK algorithm, but it differs in the way deletions of tree edges are handled. More precisely, the difference lies in how the levels are organized and how the edges are moved between levels. Both algorithms search for replacement edges at levels no higher than that of the deleted edge. Lower levels contain more important edges (and sparser parts of the graph), while less important edges are moved to higher levels (which contain denser parts of the graph). However, the HDT algorithm starts by considering edges at the bottom level and pushes them up as they are found to be in dense components, while the HK algorithm allows edges to float up automatically but pushes them to lower levels as they are found to be in sparse components. Since the HDT algorithm also uses ET-trees to maintain the trees in the forest F, it turns out that queries, edge insertions, and non-tree edge deletions are done in a manner similar to that in the HK algorithm. Before giving the details of the deletion of a tree edge, we have to explain how the levels are organized.

Similarly to the HK algorithm, the HDT algorithm assigns to each edge e a level $l(e) \leq L = \log n$, and let F_i denote the subforest of F induced by the edges with level at least i ($F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$). Two invariants are maintained:

(i) F is a maximum w.r.t. l spanning forest, i.e., if (x, y) is a non-tree edge, then x and y are connected in $F_{l(x,y)}$.
(ii) The maximum number of nodes in a tree of F_i is $\lfloor n/2^i \rfloor$.

Initially, all edges have level 0 and as the algorithm proceeds their level is increased (but never decreased). The level of a non-tree edge is increased when it is discovered that its endpoints are close enough in F to fit in a smaller tree on a higher level. The increment of the level of a tree edge should be done with care as it may violate the second invariant. As in the HK algorithm, when a tree edge $e = (x, y)$ with level $l(e) = i$ belonging to a tree T is deleted, we have to find a replacement edge, i.e., an edge which rejoins the two subtrees T_1 and T_2 resulted from the deletion of e. Let T_1 be the smaller subtree. Since $|T| \leq \lfloor n/2^i \rfloor$, it follows that $|T_1| \leq \lfloor n/2^{i+1} \rfloor$. Hence, all edges of T_1 can increase their level to $i + 1$ preserving the invariants. All non-tree edges of T_1 with level i are visited until either a replacement edge is found (in which case we stop), or all edges have been considered. In the latter case, the level of the non-tree edge is increased to $i + 1$ (both its endpoints belong to T_1). If all non-tree edges have been visited without finding a replacement edge, then the procedure is applied recursively on level $i - 1$.

Heuristics are also considered for HDT but, contrary to [1.3], they are restricted only to those which do not invalidate the worst-case time bounds. Following the HK algorithm, the first heuristic considered is sampling; i.e., before edges are promoted to a higher level, a number of incident non-tree edges is randomly sampled and tested for a replacement edge. The second heuristic is similar to the simplified version of the HK algorithm in [1.3]: truncate levels. This is accomplished by exhaustively traversing the (small) trees at higher levels when searching for a replacement edge which gives rise to fewer levels.

The main goal in the design of the experimental setup in [1.44] was to exhibit in practice the asymptotic $O(\log n)$ improvement of the HDT algorithm over the HK one. This could be achieved by designing inputs for which the HK algorithm would indeed match its worst-case time. Consequently, the experimental test set in [1.44] considers three different types of inputs, one random and two structured ones.

The random inputs are similar to those considered in [1.3] (cf. Section 1.2.1.2.a). Since both the HK and HDT algorithms can delete non-tree edges rather easily, the conducted experiments concentrated on values of m that are close to n, i.e., $m \in \{n/2, 2n\}$, and are among the most interesting regarding the structure of random graphs [1.9].

The structured inputs are split into two groups: two-level inputs (two-level random graphs and two-level semi-random graphs), and worst-case inputs.

The *two-level inputs* are similar to the so-called "non-random inputs" in [1.3]. As mentioned above, the "non-random input" of [1.3] is roughly a path of cliques where only path (i.e., inter-clique) edges are inserted and deleted. A *two-level random graph* is a sparse random graph of cliques. More precisely, k cliques, each of c vertices, are generated ($n = kc$) and are interconnected by $2k$ randomly chosen inter-clique edges. The operation sequence consists of random insertions and deletions of inter-clique edges. This class of inputs is interesting not only as a difficult case for the algorithms (as tree edges are constantly inserted and deleted), but also as an input that exhibits the clustering behaviour which motivated the development of both algorithms. Moreover, it reflects a kind of hierarchical structure that is encountered in several physical networks (e.g., road networks consisting of highways and city streets, computer networks consisting of local area networks interconnected by wide area backbones, etc). *Semi-random graphs* are random graph instances which are strongly correlated over time. Initially a fixed number ($n/2$ or $2n$) of candidate edges is chosen and then random insertions and deletions are performed from this set only. This class is perhaps more interesting than pure random graphs when modeling network applications where links fail and recover, since usually the network is fixed and it is the fixed edges which vanish and return. By replacing each vertex of a semi-random graph with a clique, we can create a *two-level semi-random graph.*

Worst-case inputs aimed at forcing the algorithms to tighten their worst-case time bounds. While it appears difficult to construct a worst-case input for the HK algorithm, Iyer et al. [1.44] succeeded to construct one for the HDT algorithm. Such an input causes HDT to promote $O(n)$ edges through all levels for $O(\log n)$ times during a sequence of $O(n)$ operations.

The worst-case input for HDT is a 4-ary tree with leaf siblings connected to each other and an update sequence constructed as follows. Let $S(k)$ be an operation sequence at the end of which all edges below (and including) tree-level k are at a level greater or equal to k in the HDT data structure. $S(k)$ is defined recursively as follows: (i) Run $S(k-1)$. (ii) Each vertex x at tree-level $k-1$ selects two of its child edges, deletes and re-inserts them. This causes the promotion of a deleted edge (x, y) and of all edges in the subtree rooted at y. (iii) Run again $S(k-1)$ to bring back to level $k-1$ the two child edges which were deleted and re-inserted. (iv) Run step (ii) with the other two child edges.

It is not difficult to see that the sequence $S(\log_4 n)$ will cause $\Theta(n)$ tree edges at tree-level $\log_4 n$ to be promoted through $\Theta(\log n)$ levels resulting in a total time bound of $O(\log^2 n)$.

The implementations of HK and HDT algorithms in [1.44] are tuned by two parameters s and b, and hence are referred to as $\{\texttt{HK},\texttt{HDT}\}(s,b)$: s denotes the sample size and b denotes that there are only $\log n - \log b$ levels. Hence, $\texttt{HK}(16\log^2 n, 0)$ denotes the HK implementation of [1.3], $\texttt{HK}(20,n)$ denotes the `HK-var` implementation in [1.3], while $\texttt{HDT}(0,0)$ denotes the implementation of the original HDT algorithm. The implementations of HK and HDT algorithms were also compared to the `fast-update` and `fast-query` implementations developed in [1.3].

The main conclusion of the experimental study in [1.44] is that the heuristics proved to be rather beneficial and that the HDT algorithm with heuristics dominates the HK algorithm. This is due to the repeated rebuildings performed by the latter and the fact that in two-level inputs the HDT algorithm searches through a clique less often than HK. More precisely, for random inputs, where the initial graph has $m \in \{n/2, 2n\}$ edges and random sequences of insertions and deletions are performed such that the graph has always no more than m edges, $\texttt{HDT}(0,n)$ (i.e., just an ET-tree) achieves the best performance, followed closely by $\texttt{HK}(20,n)$; this is basically due to the truncation of levels. In the case of two-level inputs, $\texttt{HDT}(256,0)$ – when $k < c$ – and $\texttt{HDT}(256,64)$ – when $k \geq c$, or when a semi-random graph is considered – are the fastest. In the former case ($k < c$) this is because most of the inter-clique edges are at level 0 and hence sampling for a replacement most probably will succeed, while in the latter case this is due to the fact that the HDT algorithm searches through a clique less often than HK and due to the overhead introduced by the regular rebuildings performed by HK. Finally, in the worst-case input for HDT, the $\texttt{HDT}(256,64)$ variant achieves the best performance.

The source code of the above implementations is available from
`http://theory.lcs.mit.edu/~rajiyer/papers/IKRTcode.tar.gz`.

1.2.1.3 Lessons learned. The above experimental studies allowed us to gain a deeper insight regarding existing dynamic connectivity algorithms and their practical assessment. In particular:

- The experiments in [1.3] fully confirmed the practicality of sparsification as well as its average-case analysis, thus providing valuable knowledge for the suitability of the theoretical model used and its limitations.
- The heuristic improvements of the HK and HDT algorithms, regardless of whether they do respect the asymptotic time bounds [1.44] or not [1.3], proved to be very useful in practice.
- The study in [1.22] showed that complicated algorithms can sometimes be useful in practice, while the study in [1.44] showed that a logarithmic improvement in asymptotic performance can still have a practical impact.
- The data sets developed in [1.3] formed the cornerstone upon which the subsequent experimental studies were based. We feel that these data sets along with their considerable expansion and elaboration in [1.44] yield an important benchmark for the testing of other dynamic algorithms.

1.2.2 Dynamic Minimum Spanning Tree

1.2.2.1 Theoretical background – problem and history of results. The minimum spanning tree (MST) of a graph $G = (V, E)$, whose edges are associated with real-valued weights, is a spanning tree of minimum total weight. In the dynamic minimum spanning tree problem, we would like to maintain the MST in a graph G that undergoes a sequence of updates (edge insertions and edge deletions).

According to Frederickson [1.25], the first results for fully dynamic MST are attributed to Harel (1983) and achieve an update time of $O(n \log n)$. The first breakthroughs were given by Frederickson in [1.25, 1.26]; in those papers fully dynamic MST algorithms were presented with a running time per update ranging from $O(m^{2/3})$ to $O(m^{1/2})$. As explained in Section 1.2.1.2.a, sparsification by Eppstein et al. [1.19] reduces these running times to be in the range from $O(n^{2/3})$ to $O(n^{1/2})$. A further improvement was achieved by Henzinger & King [1.38] who gave a fully dynamic algorithm with $O(n^{1/3} \log n)$ amortized update bound. Finally, the first polylogarithmic update bound was given by Holm et al. [1.40]; they presented a fully dynamic MST algorithm with amortized update time $O(\log^4 n)$.

1.2.2.2 Implementations and Experimental Studies. There are two works known regarding implementation and experimental studies of dynamic MST algorithms. In chronological order, these are the works by Amato et al. [1.4] and by Cattaneo et al. [1.10].

The former paper is a follow up of the study in [1.3] and investigates the practical performance of both Frederickson's algorithms [1.25, 1.26] and of stable sparsification on top of dynamic algorithms [1.19]. The latter paper enhances the study in [1.4] by bringing the algorithm by Holm et al. [1.40] into play and aims at investigating its practicality in comparison to the implementations in [1.4] as well as to new simple algorithms based on static approaches.

Throughout the rest of this section, let $G = (V, E)$ be the input graph and let T be its minimum spanning tree.

1.2.2.2.a *The Implementation by Amato et al.* [1.4]

The main goal of the first experimental study for dynamic MST algorithms was:

1. To investigate the practicality of stable sparsification, i.e., spasification on top of dynamic algorithms [1.19].
2. To compare sparsification-based algorithms with Frederickson's algorithms [1.25, 1.26] for dynamic MST.
3. To investigate the practicality of the dynamic algorithms in comparison to simple-minded algorithms based on static approaches that were easy to implement and likely to be fast in practice.

The algorithms by Frederickson are based on appropriately grouping the vertices of T into *vertex clusters* (set of vertices which induce a connected subgraph of T) such that suitable partitions on V are defined either directly (yielding *balanced* or *restricted partitions*) or indirectly by recursive applications of clustering (yielding *topology trees* and *2-dimensional topology trees*). These partitions allow for an encoding of the MST which can be efficiently updated after a dynamic change in G.

Two different partitions are given in [1.25, 1.26]. The first partition [1.25] is called a *balanced partition of order* z and is simply a partition of V into vertex clusters of cardinality between z and $3z - 2$. The second partition [1.26] is called *restricted partition of order* z as it sets more requirements on how clustering is done: (i) each set in the partition yields a vertex cluster of external degree[2] at most 3; (ii) each cluster of external degree 3 has cardinality 1; (iii) each cluster of external degree less than 3 has cardinality at most z; and (iv) no two adjacent clusters can be combined and still satisfy the above. Both partitions have $O(m/z)$ clusters. The maintenance of each partition during edge insertions and deletions allows it to dynamically maintain the MST of a graph in time $O(z + (m/z)^2) = O(m^{2/3})$ [1.25]. This method yields the first two algorithms implemented in [1.4], namely `FredI-85` (based on the balanced partition) and `FredI-91` (based on the restricted partition).

Although the asymptotic behaviour of both partitions is identical, the experiments conducted in [1.4] revealed several differences between balanced and restricted partitions w.r.t. their practical behaviour that favor the former: (a) the number of clusters generated by a balanced partition is much less than those generated by a restricted partition; (b) there is a smaller number of splits and merges of clusters (which are expensive operations) in a balanced partition and hence clusters have a longer lifetime; and (c) the average number of affected clusters by an update is substantially smaller in balanced partitions than in restricted partitions. As a consequence, `FredI-85` was always faster than `FredI-91`.

The above observed differences motivated a theoretical and experimental tuning of the parameters of the two partitions resulting in a third partition called *light partition* [1.4]. This partition is a relaxed version of the restricted partition, namely: (i) each cluster is of cardinality at most z; and (ii) no two adjacent clusters can be combined and still satisfy the above. Since a light partition is a relaxation of the restricted partition, its number of clusters (at least initially) cannot be more than those of the restricted partition, i.e., $O(m/z)$. The problem, however, with light partition is that there is no guarantee that this number is preserved throughout any sequence of edge insertions and deletions. Consequently, the worst-case update bound of a dynamic MST algorithm based on light partitions are worse than $O(m^{2/3})$. On the other hand, the experiments in [1.4] showed that in practice the number of clusters in a light partition does not increase much beyond its

[2] Number of edges having their other endpoint at a different cluster.

initial number of $O(m/z)$ as the number of edge updates increases. To further increase its efficiency, the light partition was further "engineered" in [1.4] by introducing a lazy update scheme for the expensive update of the light partition: the partition is updated only in the case of tree edge deletions (i.e., in the case where there is certainly a change in the current MST). Edge insertions are handled in $O(\log n)$ time by recomputing the MST of the current graph using the dynamic tree data structure of Sleator & Tarjan [1.61]. Hence, a cluster that would otherwise be affected by several edge insertions and deletions, is now updated only at the time of a tree edge deletion. This lazy update scheme along with a light partition of order $\lceil m^{2/3} \rceil$ yields another implementation for the dynamic MST problem referred to as `FredI-Mod` [1.4]. The experiments in [1.4] showed that `FredI-Mod` was always significantly faster than both `FredI-85` and `FredI-91`.

The recursive application of balanced or restricted partitions yields different types of partitions that end up in the so-called topology tree. A *multi-level balanced partition* has the following properties: (i) the clusters at level 0 contain a single vertex; (ii) a cluster at level $i \geq 1$ is either a cluster at level $(i-1)$ of external degree 3 or the union of at most 4 clusters (according to some rules) at level $(i-1)$; (iii) there is exactly one cluster at the topmost level. A *multi-level restricted partition* obeys the same properties, except for (ii) which is stated as: the clusters at level $i \geq 1$ form a restricted partition of order 2 w.r.t. the tree obtained after shrinking all clusters at level $(i-1)$. Note that this rule makes a multi-level restricted partition to be defined in a much simpler way than a multi-level balanced partition. A *topology tree* is a tree which represents the above multi-level partitions, i.e., a *balanced* (resp. *restricted*) *topology tree* is a tree for which a node at level i represents a cluster at level i of a balanced (resp. restricted) multi-level partition, and the children of a node at level $i \geq 1$ are the clusters at level $(i-1)$ whose union gives the cluster at level i. It is easy to see that both topology trees have height $O(\log N)$, where N is the total number of nodes in the tree. In [1.25, 1.26] it is shown that updating any topology tree after an edge insertion, edge deletion, or edge swap, takes $O(\log N)$ time.

By using a topology tree to represent each cluster in a (balanced or restricted) partition of order z, we can get an $O(z + (m/z)\log(m/z)) = O(\sqrt{m}\log m)$ time algorithm for the dynamic MST problem. This yields two implementations in [1.4], namely `FredII-85` (balanced partitions and topology trees) and `FredII-91` (restricted partitions and topology trees). The experiments conducted in [1.4] showed that `FredII-91` was slightly faster than `FredII-85`, mainly due to the simpler clustering rules which the restricted topology trees and multi-level partitions obey. In [1.4], a hybrid solution was also investigated involving a suitable combination of a balanced partition of order z with restricted topology trees. However, even this hybrid solution turned out to be slower than `FredI-85` for most inputs they considered.

To efficiently maintain information about the non-tree edges, Frederickson introduces the *2-dimensional topology trees* which are defined by pairs of nodes in a topology tree. For every pair of nodes V_a and V_b at the same level of a topology tree, there is a node labeled $V_a \times V_b$ in the 2-dimensional topology tree which represents the non-tree edges of G having one endpoint in V_a and the other in V_b. If V_a (resp. V_b) has children V_{a_j}, $1 \le j \le p$ (resp. V_{b_k}, $1 \le k \le q$) in the topology tree, then $V_a \times V_b$ has children the nodes $V_{a_j} \times V_{b_k}$ ($1 \le j \le p$, $1 \le k \le q$) in the 2-dimensional topology tree. The use of a 2-dimensional topology tree yields an $O(z + m/z) = O(\sqrt{m})$ time algorithm for the dynamic MST problem. This theoretical improvement, however, does not show up in practice: all implementations that employed 2-dimensional topology trees in [1.4] were much slower than `FredI-85`.

The above implementations were enhanced by applying stable sparsification (i.e., simple sparsification where a fully dynamic algorithm is used to recompute certificates; see Section 1.2.1.2.a) on top of them. More precisely, sparsification was applied on top of `FredI-85` yielding algorithm `Spars(I-85)` with an $O(n^{2/3} \log(m/n))$ worst-case time bound, and on top of `FredI-Mod` yielding algorithm `Spars(I-Mod)`.

Finally, a simple fully dynamic algorithm was implemented in [1.4], called `adhoc`, which is a combination of a partially dynamic data structure – based on the dynamic trees of Sleator & Tarjan [1.61] – and LEDA's static MST algorithm called `Min_Spanning_Tree`, which is a fine-tuned variant of Kruskal's algorithm based on randomization with an average-case time of $O(m + n \log^2 n)$ and a worst-case time of $O(m \log m)$. The `adhoc` algorithm maintains two data structures: the MST T of G as a dynamic tree, and a priority queue Q which stores all edges of G according to their weight. When a new edge is inserted, `adhoc` updates T and Q in $O(\log n)$ time. When an edge is deleted from G, it is first deleted from Q in $O(\log n)$ time. If it was a non-tree edge, nothing else happens. Otherwise, `adhoc` calls `Min_Spanning_Tree` on the edges of Q. Consequently, `adhoc` requires $O(\log n)$ time plus the time of `Min_Spanning_Tree` in the case where a tree edge is deleted. If the edge to be deleted is chosen uniformly at random from G, then this expensive step occurs with probability n/m resulting in an average running time of $O(\log n + (n/m)(m + n \log^2 n)) = O(n + (n \log n)^2/m)$ for `adhoc`. Hence, it is natural to expect that its running time decreases as the graph density increases, a fact that was confirmed by the experiments in [1.4].

As mentioned above, in all experiments which carried out in [1.4], `FredI-85` was consistently the fastest among the six algorithms derived by the approaches in [1.25, 1.26]. Hence, this implementation was chosen to be compared experimentally with the rest of implementations, namely `FredI-Mod`, `Spars(I-85)`, `Spars(I-Mod)`, and `adhoc`.

Experiments were run on both random and non-random inputs. Random inputs aimed at investigating the average-case performance of the implemented algorithms and confirming the average-case complexity of `adhoc`.

Non-random inputs aimed at producing test sets which would make the algorithms to exhibit their worst-case performance.

Random inputs were similar to those considered in [1.3, 1.44] and consisted of random graphs (with random edge weights) and random operation sequences in which edge insertions were uniformly mixed with edge deletions. Also, the edge weights in each update sequence were chosen at random. Non-random inputs were generated by first constructing a graph, then computing its MST, and finally deleting the edges of the MST one at a time. Recall that a tree edge deletion is the most expensive operation in any of the dynamic MST algorithms considered.

In all experiments with random inputs, `adhoc` was almost always the fastest algorithm; only in the case of (initially) sparse graphs on a large number of vertices, `FredI-Mod` was faster. Among the dynamic algorithms, `FredI-Mod` was consistently faster than any of the other implementations followed by `FredI-85`. The implementations based on sparsification were the slowest. In non-random inputs, however, a "reverse" situation was reported. That is, `Spars(I-Mod)` was the fastest algorithm, followed by `FredI-Mod`, and `adhoc` was by far the worst. This was more or less expected, since random edge deletions (especially in non-sparse graphs) will most probably be non-tree edge deletions which makes `adhoc` superior for random inputs; on the contrary, tree edge deletions make `adhoc` exhibit its worst-case performance since they cause it to recompute the MST.

The different behaviour of the implementations based on sparsification can be explained as follows. Sparsification spreads the information about a graph G into smaller sparse subgraphs. Even if a random update may not change the MST of the entire G, it may happen that some of the smaller subgraphs have to change their MST and this shows up in random inputs. On the other hand, a bad update operation (tree edge deletion) will not make a big difference w.r.t. a good update, since any update spreads on a logarithmic number of smaller subgraphs. Consequently, this spreading is simultaneously an advantage (for non-random inputs) and a disadvantage (for random inputs) of sparsification. A final conclusion from the experimental study of [1.4] was the efficiency of `FredI-Mod` which was consistently faster than `FredI-85` and `Spars(I-85)` even for non-random inputs, mainly due to the more relaxed nature of light partitions.

The source code of the above implementations is available via anonymous ftp from `ftp.dia.unisa.it` in the directory `pub/italiano/mst`.

1.2.2.2.b ***The Implementation by Cattaneo et al.*** [1.10]

The main goal of that study was to investigate the practicality of the recent fully dynamic MST algorithm by Holm et al. [1.40], henceforth HDT_{mst}, and comparing it with Frederickson's dynamic algorithms, sparsification-based algorithms and new simple algorithms. To this end, the experimental study in [1.10] built upon the studies by Amato et al. [1.4] and by Iyer et al. [1.44], and resulted in considerably enhancing the quiver of dynamic MST implementations as well as our knowledge on their practicality.

The HDT$_{mst}$ algorithm is based on a decremental algorithm for MST which has many similarities with the HDT algorithm for dynamic connectivity (cf. Section 1.2.1.2.c). Then, using a rather general construction which was first introduced in [1.38], the decremental algorithm is converted into a fully dynamic one.

The decremental MST algorithm can be obtained by the HDT algorithm by doing two very simple changes. First, a minimum spanning forest (instead of any spanning forest) F is maintained. Second, there is a different way with which non-tree edges are considered for replacement edges, when a tree edge is deleted. Instead of an arbitrary order, the non-tree edges incident to the smaller subtree are considered in order of non-decreasing weight.

The partition of edges into levels is done similarly to that in the HDT algorithm (cf. Section 1.2.1.2.c). In addition to the two invariants of the HDT algorithm, the following invariant is maintained.

(iii) If e is the heaviest edge on a cycle C, then e has the lowest level on C.

It is not hard to see that the above invariant ensures that the minimum weight edge among all replacement edges is the lightest edge among all replacement edges on the highest possible level.

The approach for converting the above decremental algorithm to a fully dynamic one is as follows. A set of data structures $A = A_1, A_2, \ldots, A_s$ is maintained, where $s = \lceil \log n \rceil$ and each A_i is a subgraph of G. Let F_i be the local spanning forest maintained in A_i. Edges in F_i are referred to as local tree edges while those in F are referred to as global tree edges. All edges of G are in at least one A_i, hence $F \subseteq \bigcup_i F_i$. The algorithm maintains the invariant that for each global non-tree edge $e \in G - F$, there is exactly one i such that $e \in A_i - F_i$ and if $e \in F_j$, then $j > i$. The minimum spanning forest F is maintained as a dynamic tree data structure of Sleator & Tarjan [1.61] (ST-tree) and also as an Euler-Tour tree (ET-tree) of Henzinger & King [1.36] (in order to easily find replacement edges).

Before describing the edge insertion and deletion operations, we have to describe an auxiliary procedure, called *Update*(A, D), which updates the data structure A with a set of edges D. Let $B_j = \bigcup_{k \leq j}(A_k - F_k)$, i.e., the set of local non-tree edges in all A_k, for $k \leq j$. The procedure works as follows. It finds the smallest j such that $|(D \cup B_j) - F| \leq 2^j$ and sets $A_j = F \cup D \cup B_j$. Then, it initializes A_j as a decremental MST data structure and sets $A_k = \emptyset$ for all $k < j$.

The insertion of a new edge $e = (x, y)$ is carried out as follows. If x and y are in different trees in F, then simply add e to F. Otherwise, find the heaviest edge f in the x-y path. If e is heavier than f, then call *Update*(A, e). Otherwise, replace f with e in F and call *Update*(A, f).

The deletion of an edge e is done as follows. The edge e is deleted from all A_i which contained e and let R be the set of the collected replacement edges returned by all decremental MST data structures. If e is a global tree edge, then search in R (using the ET-tree representation of F) to find the lightest edge which reconnects F. Finally, call *Update*(A, R).

The crucial point in the update procedure is the initialization of A_j. To achieve this efficiently, Holm et al. [1.40] perform a contraction of some local tree paths (those which are not incident on any non-tree edge) using a special data structure they introduce, called *top trees*. This allows them to bound the initialization work in each update.

The implementation of the HDT$_{mst}$ algorithm in [1.10] is not based on the `HDT` implementation of Iyer et al. [1.44], because Cattaneo et al. [1.10] found the latter specifically targeted and engineered for the dynamic connectivity problem and conversion of this code for dynamic MST appeared to be a difficult task. Hence, they provided a completely new implementation that is better suited for dynamic MST. In their implementation they follow closely the above described HDT$_{mst}$ algorithm with the exception that they do not use top trees for path compression, since top trees were consuming a lot of memory. They initially used ST-trees instead, and later on observed that by completely omitting path compression a considerable gain in performance was obtained [1.23]. The resulting implementation is called `HDT`.

The simple algorithms considered in [1.10] are a kind of fast dynamization of Kruskal's algorithm. One algorithm is based on ST-trees and the other on ET-trees.

The first algorithm maintains the MST T as an ST-tree and the non-tree edges are maintained sorted in a binary search tree NT.

The insertion of a new edge (x, y) is similar to the insertion procedure of the fully dynamic case: if (x, y) should replace an existing tree edge f (which can be easily checked in $O(\log n)$ time using the ST-trees), then f is deleted from T and is inserted into NT. Otherwise, (x, y) does not belong to the MST and is inserted into NT. Clearly, insertion can be accomplished in $O(\log n)$ time.

The deletion of an edge (x, y) depends on whether it is a non-tree or a tree edge. In the former case, (x, y) is simply deleted from NT in $O(\log n)$ time. In the latter case, the deletion of (x, y) disconnects T into T_x (the subtree containing x) and T_y (the subtree containing y) and a replacement edge has to be found. To accomplish this, non-tree edges in NT are scanned in non-decreasing weight order in a manner similar to Kruskal's algorithm: if a non-tree edge (s, t) reconnects T_x and T_y (a fact which can be easily checked using the findroot(s) and findroot(t) operations of ST-trees), then the scanning is terminated; otherwise, the non-tree edge is discarded. Clearly, the total time of the deletion operation is $O(k \log n)$, where k is the number of scanned non-tree edges, which is $O(m \log n)$ in the worst case.

The implementation of the above algorithm is referred to as `ST`. Since `ST` can be implemented in a few lines of code using fast (and simple) data structures, it is expected that it performs very well in practice, especially when the update sequence contains a few tree edge deletions or when a tree edge deletion results in a small set of possible replacement edges. This fact was confirmed in most of the experiments, except for the case of sparse graphs i.e., with m close to n. In these cases, and when dealing with random graphs, a random edge deletion will most probably be a tree edge that disconnects the MST and consequently will cause the scanning of many non-tree edges until the proper replacement edge is found (if any). Actually, it turned out that most of the time in these cases was spent in executing findroot operations.

Motivated by this difficult case of `ST`, Cattaneo et al. [1.10] designed another variant, called `ET`, which in addition uses ET-trees (as they have been implemented in [1.3] and supported by the randomized search trees of [1.6]). More precisely, the information about tree edges is kept both on an ST-tree and on an ET-tree. ET-trees are used only during the deletion operation to check whether a non-tree edge reconnects the MST. It should be pointed out that ET-trees have the same asymptotic time bounds with ST-trees (cf. Section 1.2.1.2.a). Since findroot operations in randomized search trees are expected to be faster than those in ST-trees, it is consequently expected that `ET` is faster than `ST` on sparse graphs. However, it is also expected that the overhead of maintaining both ET-trees and ST-trees will show up when the findroot operations are not any more the bottleneck. Both expectations were confirmed by the experiments conducted in [1.10].

The above implementations `HDT, ST` and `ET` were experimentally compared with `Spars(I-Mod)` from [1.4] (cf. Section 1.2.2.2.a). The experimental test set was built upon the ones used in [1.4] and [1.44]. In particular, both random and structured inputs were considered.

The random inputs were identical to the random inputs used in [1.4] (cf. Section 1.2.2.2.a). The structured inputs consisted of semi-random inputs, two-level random inputs, and worst-case inputs, all generated in way very similar to that in [1.44] (cf. Section 1.2.1.2.c).

Semi-random inputs consisted of semi-random graphs with edge weights chosen at random and where update operations are chosen from a fixed set of edges in a way identical to that in [1.44].

Two-level random inputs consisted of two-level random graphs (k cliques, each one of c vertices, interconnected by $2k$ randomly chosen inter-clique edges) and edge weights chosen at random. The operation sequence was of two types. The first involved only insertions and deletions of inter-clique edges, as it happens in [1.44]. The second type involved insertions and deletions of edges inside a clique (intra-clique edges) and mixed updates between intra-clique and inter-clique edges.

Finally, a worst-case input for HDT_{mst} was constructed by suitably adapting the worst-case input of HDT in [1.44] for dynamic connectivity (cf. Section 1.2.1.2.c).

In the experiments with random inputs, `ET` was the fastest implementation in sparse graphs followed by `ST`, even in sequences with a high percentage of tree edge deletions. In non-sparse graphs, `ET` and `ST` had almost identical behaviour, with `ET` being slightly worse due to the additional overhead introduced by maintaing both ST-trees and ET-trees. `HDT` was slower than `Spars(I-Mod)` when the graph is sparse, but it takes over as the edge density increases and the overhead of the sparsification tree in `Spars(I-Mod)` shows up. These outcomes are quite interesting, since they match those of other studies [1.4, 1.44] (cf. Sections 1.2.2.2.a and 1.2.1.2.c, respectively) and show that ET-trees are a particularly valuable data structure for dynamic problems on random graphs.

Similar behaviour was reported for the case of semi-random inputs when the cardinality of the fixed set of edges (from which updates are chosen randomly) is small. However, when the cardinality of the fixed set of edges is increased, `ET` still remains the fastest followed closely by `Spars(I-Mod)`. The other two algorithms are considerably penalized in performance, with `ST` being by far the slowest – apparently due to overhead of findroot operations.

In the case of two-level inputs, both `ET` and `ST` were not competitive, because of the considerable overhead introduced by the deletion of inter-clique edges. The `HDT` implementation was faster, regardless of the clique size, in all cases where there were either no intra-clique edge deletions, or very few of them. However, as the number of intra-clique edge deletions increases, `Spars(I-Mod)` improves over `HDT`.

In the experiments with the worst-case input, `HDT` is penalized by the update sequence in executing unnecessary promotions of edges among levels and is worse than `ET` or `ST` which actually achieve their best performance of $O(\log n)$ (as non-tree edges do not exist to be considered for replacement edges). `Spars(I-Mod)` also suffered a significant performance loss by this input (it was the slowest), because a tree edge deletion carries the burden of the overhead imposed by the implementation of light partition.

1.2.2.3 Lessons learned. The above experimental studies considerably enhance our knowledge regarding the practicality of dynamic MST algorithms.

The work by Amato et al. [1.4] is an exemplary study of algorithm engineering. Thorough experimentation helped in identifying the best algorithm in practice (an otherwise extremely difficult task) from a collection of identically behaving algorithms w.r.t. the asymptotics. Experimentation also revealed the bottlenecks in performance, leading to the introduction of heuristics followed by careful fine-tuning. This considerably improved the practical performance of theoretically inferior algorithms. On the other hand, theoretically superior algorithms achieved through the use of several layers of complicated data structures turn out not to be useful in practice.

The work by Cattaneo et al. [1.10] is a follow up which on the one hand provides an implementation of the theoretically best algorithm, and on the other hand helps in identifying the cases in which that algorithm can be of real practical value.

Finally, as in the case of dynamic connectivity, the carefully designed data sets exhibited the differences in the average and worst case performances of the implemented algorithms. To this end, the data sets for dynamic connectivity developed in [1.3, 1.44] turned out to be an excellent starting point.

1.3 Dynamic Algorithms for Directed Graphs

The implementation studies known for dynamic problems in directed graphs (digraphs) concern transitive closure and shortest paths.

1.3.1 Dynamic Transitive Closure

1.3.1.1 Theoretical background – problem and history of results. Given a digraph $G = (V, E)$, the *transitive closure* (or *reachability*) problem consists in finding whether there is a directed path between any two given vertices in G. We say that a vertex v is *reachable* by vertex u iff there is a (directed) path from u to v in G. The digraph $G^* = (V, E^*)$ that has the same vertex set with G but has an edge $(u, v) \in E^*$ iff v is reachable by u in G is called the *transitive closure* of G; we shall denote $|E^*|$ by m^*. If v is reachable from u in G, then we call v a *descendant* of u, and u an *ancestor* of v. In the following we denote by $DESC[v]$ the set of descendants of v.

There are several partially dynamic algorithms for transitive closure, and some recent fully dynamic ones. All algorithms create a data structure that allows update operations (edge insertion/deletion) and query operations. A query takes as input two vertices u and v and can be either `Boolean` (returns "true" if there is a u-v path, otherwise "false") or `Path` (returns in addition the actual path if it exists). In the following, let G_0 be the initial digraph (before performing a sequence of updates) having n vertices and m_0 edges. We start with the partially dynamic algorithms.

The first partially dynamic algorithms were given by Ibaraki and Katoh [1.41]. Their incremental (resp. decremental) algorithm supported any number m of edge insertions (resp. deletions) in $O(n(m + m_0)^*)$ (resp. $O((n + m_0)m_0^*)$) time and answered `Boolean` queries in $O(1)$ time, and `Path` queries in time proportional to the number of edges of the reported path (i.e., both type of queries are answered optimally). These bounds were later improved by Italiano [1.42, 1.43], Yellin [1.66], La Poutré & van Leeuwen [1.49], and Cicerone et al. [1.12]. The decremental part of these algorithms apply only to directed acyclic graphs (DAGs). Italiano's and Yellin's data structures support both `Path` and `Boolean` queries, while the other two data structures [1.12, 1.49] support only `Boolean` queries. All these partially dynamic algorithms, except for Yellin's, support any number m of edge insertions (resp. deletions) in $O(n(m + m_0))$ (resp. $O(nm_0)$ time); i.e., for $m = \Omega(m_0)$, they achieve an $O(n)$ amortized update bound. `Boolean` and `Path` queries (where applicable) are answered optimally. The algorithm by La Poutré & van Leeuwen [1.49] can be extended to handle edge deletions in general digraphs with an amortized time of $O(m_0)$ per deletion. Also, in [1.32, 1.33] the algorithms by Italiano [1.42, 1.43] were extended so that the decremental part applies to any digraph. While the amortized time per edge deletion remains $O(m_0)$, this new algorithm is able to support `Path` queries optimally. The above data structures by Italiano, La Poutré & van Leeuwen, and Cicerone et al. can be initialized in $O(n^2 + nm_0)$ time and require $O(n^2)$ space.

Yellin's data structure has different initialization bounds depending on whether it supports `Path` queries or not. More precisely, the initialization time and space bounds are the same as those of the other algorithms, if Yellin's data structure supports only `Boolean` queries. If in addition `Path` queries are supported, then both time and space bounds become $O(n^2 + dm_0^*)$, where d is the maximum outdegree of G_0. In either variant of Yellin's algorithm: (i) Queries are supported optimally. (ii) The incremental part requires $O(d(m_0 + m)^*)$ time to process a sequence of m edge insertions starting from an initial digraph G_0 and resulting in a digraph G; d is the maximum outdegree of G. (iii) The decremental version requires $O(dm_0^*)$ time to process any sequence of m edge deletions; d is the maximum outdegree of G_0.

Henzinger & King in [1.37] gave a decremental randomized algorithm which is initialized in $O(n^2 + nm_0)$ time and space, supports any sequence of edge deletions in $O(m_0 n \log^2 n)$ expected time, and answers `Boolean` queries in $O(n/\log n)$ worst-case time. `Path` queries are supported in an additional time which is proportional to the number of edges of the reported path. Recently, two incremental algorithms were given by Abdeddaim [1.1, 1.2] who showed that if the graph is covered by a set of k vertex-disjoint paths, then any sequence of m edge insertions can be accomplished either in $O(k^2(m_0 + m) + (m_0 + m)^*)$ time [1.1], or in $O(k(m_0 + m)^*)$ time [1.2]. Both algorithms use $O(kn)$ space.

We now turn to the fully dynamic algorithms. The first two fully dynamic algorithms for transitive closure were presented in [1.37]. They are randomized (with one-side error) and are based on the decremental algorithm presented in the same paper. The first (resp. second) fully dynamic algorithm supports update operations in amortized expected time $O(\hat{m}\sqrt{n}\log^2 n + n)$ (resp. $O(n\hat{m}^{0.58}\log^2 n)$), where $\hat{m}$ is the average number of edges in G during the whole sequence of updates. Since $\hat{m}$ can be as high as $O(n^2)$, the update bounds of [1.37] can be $O(n^{2.5}\log^2 n)$ (resp. $O(n^{2.16}\log^2 n)$). Queries are answered within the same time bounds as those of the decremental algorithm. Khanna, Motwani, and Wilson [1.46] presented a fully dynamic algorithm that achieves a deterministic amortized update bound of $O(n^{2.18})$, when a lookahead of size $\Theta(n^{0.18})$ in the update sequence is allowed. The next fully dynamic algorithm was given by King and Sagert [1.48]. It is a randomized one (with one-side error) supporting updates in $O(n^{2.26})$ amortized time for general digraphs and in $O(n^2)$ worst-case time for DAGs, and supports `Boolean` queries in $O(1)$ time. These bounds were further improved by King in [1.47], who gave a deterministic algorithm with $O(n^2 \log n)$ amortized update time and $O(1)$ `Boolean` query time. Very recently, Demetrescu and Italiano [1.13] outperformed the above approaches by presenting two fully dynamic algorithms: the first is a deterministic one supporting updates in $O(n^2)$ amortized time and `Boolean` queries in $O(1)$ time; the second is a randomized algorithm (with one-side error), it applies only to DAGs, and achieves a trade-off between query and update time. The currently best worst-case time for an update is $O(n^{1.575})$, which achieves a worst-case `Boolean` query time of $O(n^{0.575})$.

1.3.1.2 Implementations and Experimental Studies. There are two works known regarding implementation and experimental studies of dynamic algorithms for transitive closure. In chronological order, these are the works by Frigioni et al. [1.32, 1.33] and by Abdeddaim [1.2].

The former paper investigates the practicality of several dynamic algorithms (most of them with identical theoretical performance) in partially and fully dynamic settings. The latter paper investigates incremental dynamic algorithms under a specific application scenario (sequence alignment).

1.3.1.2.a *The Implementation by Frigioni et al.* [1.32, 1.33]

The main goal of the first experimental study for dynamic transitive closure was threefold:

1. To investigate the practicality of dynamic algorithms over their static counterparts – especially to very simple and easily implementable ones.
2. To investigate the differences in the practical performance of several dynamic algorithms which appear to have similar asymptotic behaviour.
3. To compare partially dynamic algorithms against fully dynamic ones.

Several partially dynamic and one fully dynamic algorithm were investigated in [1.32, 1.33]. More precisely, the partially dynamic algorithms considered were those by Italiano [1.42, 1.43], Yellin [1.66], Cicerone et al. [1.12], and Henzinger & King [1.37]. The fully dynamic algorithm was one of the two presented in [1.37]. Also, as a consequence of fine-tuning of some of these algorithms, several variants were obtained (including hybridizations of partially dynamic algorithms to yield fully dynamic ones) which turned out to be quite fast in practice. All these algorithms were compared with other simple-minded approaches that were easy to implement and hence likely to be fast in practice.

The starting point was the implementation of Italiano's algorithms [1.42, 1.43]. The main idea of the data structure proposed in those papers is to associate (and maintain) with every vertex $u \in V$ a set $DESC[u]$ containing all descendants of u in G. Each such set is organized as a spanning tree rooted at u. In addition, an $n \times n$ matrix of pointers, called $INDEX$, is maintained which allows fast access to vertices in these trees. More precisely, $INDEX[i, j]$ points to vertex j in $DESC[i]$, if $j \in DESC[i]$, and it is $Null$ otherwise.

A `Boolean` query for vertices i and j is carried out in $O(1)$ time, by simply checking $INDEX[i,j]$. A `Path` query for i and j is carried out in $O(\ell)$ time, where ℓ is the number of edges of the reported path, by making a bottom-up traversal from j to i in $DESC[i]$.

The insertion of an edge (i,j) is done as follows. The data structure is updated only if there is no i-j path in G. The insertion of edge (i,j) may create new paths from any ancestor u of i to any descendant of j only if there was no previous u-j path in G. In such a case the tree $DESC[u]$ is updated using the information in $DESC[j]$ (deleting duplicate vertices) and accordingly the u-th row of $INDEX$.

The deletion of an edge (i,j) on a DAG G is done as follows. If (i,j) does not belong to any $DESC$ tree, then the data structure is not updated. Otherwise, (i,j) should be deleted from all $DESC$ trees to which it belongs. Assume that (i,j) belongs to $DESC[u]$. The deletion of (i,j) from $DESC[u]$ splits it into two subtrees, and a new tree should be reconstructed. This is accomplished as follows. Check whether there exists a u-j path that avoids (i,j); this is done by checking if there is an edge (v,j) in G such that the u-v path in $DESC[u]$ avoids (i,j). If such an edge exists, then swap (i,j) with (v,j) in $DESC[u]$, and join the two subtrees using (v,j). In such a case, (v,j) is called a *valid replacement* for (i,j), and v is called a *hook* for j. If such an edge does not exist, then there is no u-v path in G and consequently j cannot be a descendant of u anymore: delete j from $DESC[u]$ and proceed recursively by deleting the outgoing edges of j in $DESC[u]$. To quickly find valid replacement edges, an $n \times n$ matrix $HOOK$ is used in the implementation. The entry $HOOK[u,j]$ stores the pointer to the first unscanned item in j's list of incoming edges $IN[j]$, if such an item exists; otherwise, $HOOK[u,j] = Null$. It is also easy to verify that if some $x \in IN[j]$ has already been considered as a tentative hook for j, then it will never be a hook for j in any subsequent edge deletion. The implementation of the above described algorithm is referred to as `Ital`. In [1.32, 1.33], a second version of the above algorithm was implemented by removing recursion in the edge insertion and edge deletion procedures. This yielded implementation `Ital-NR`. The experiments showed that `Ital` and `Ital-NR` have almost identical performances with `Ital` quite often being slightly faster.

Italiano's algorithms were further optimized in [1.33] by providing a fine-tuned third implementation, called `Ital-Opt`, which maintains descendant trees and hook information implicitly. This implementation was consistently faster than `Ital` and `Ital-NR` in all experiments performed.

The above implementations of Italiano's algorithm were adapted to work in fully dynamic environments, although their asymptotic bounds do not hold in this case. The goal was to experimentally compare them with fully dynamic algorithms as it may happen that they work well in practice, a fact that was indeed confirmed by the experiments in [1.32, 1.33]. To handle mixed sequences of updates, Italiano's algorithm has to be modified, since now the insertion of new edges may provide new hook information for some vertices. Consequently, the $HOOK$ matrix has to be reset after each edge insertion that is followed by a sequence of edge deletions. Resetting the $HOOK$ matrix takes $O(n^2)$ time. The reset operation has been incorporated in `Ital` and `Ital-NR` when they are used in a fully dynamic environment. Since the overhead introduced by each reset may be significant (as it was also verified in practice), a *lazy approach* was adopted in [1.32, 1.33]: delay the resetting of an entry of the $HOOK$ matrix until it is required by the algorithm. This lazy approach was incorporated in `Ital-Opt`. Experiments showed a significant improvement upon Italiano's original algorithms (`Ital` and `Ital-NR`) on mixed sequences of updates.

The above ideas were extended in [1.32, 1.33] to develop a new algorithm whose decremental part applies to any digraph, and not only to DAGs. This algorithm (and its implementation) is referred to as `Ital-Gen`.

The algorithm is based on the fact that if we shrink every strongly connected component of a digraph $G = (V, E)$ to a single vertex, called *supervertex*, then the resulting graph $G' = (V', E')$ is a DAG. The idea is to use Italiano's algorithm to maintain the transitive closure in G' and additional information regarding the strongly connected components (SCCs) which is crucial for the decremental part of the algorithm.

The data structure consists of: (a) a collection of implicitly represented descendant trees (as in `Ital-Opt`); (b) an $n \times n$ Boolean matrix $INDEX$ (as in `Ital`); (c) an array SCC of length n, where $SCC[i]$ points to the SCC containing vertex i; and (d) the SCCs of G as graphs. Furthermore, with each k-vertex SCC two additional data structures are maintained: an array $HOOK$ of length n, where $HOOK[i]$ points to the incoming edge of the SCC used in the (implicitly represented) descendant tree rooted at i; and a *sparse certificate* of the SCC consisting of k vertices and $2k - 2$ edges.

The initialization involves computation of the above data structures where the computation of the SCC and their sparse representatives is performed only for the decremental part of the algorithm, i.e., before any sequence of edge deletions. (If there is no such sequence, then every vertex is taken as a SCC by itself.)

`Boolean` and `Path` queries can be answered in the same time bounds with those of `Ital`, by first looking at the $INDEX$ matrix to check whether there exists a path; if yes, then the path can be found using the $HOOK$ arrays (which provide the path in G') and the sparse certificate of each SCC (which gives the parts of the required path represented by supervertices in G').

The insertion of an edge (i, j) is done similarly to the original algorithm by Italiano. Deleting an edge (i, j) is done as follows. If (i, j) does not belong to any SCC, then use Italiano's decremental algorithm to delete (i, j) from G'. Otherwise, check if (i, j) belongs to the sparse certificate of the SCC or not. In the latter case, simply remove the edge from the SCC. In the former case, check if the deletion of (i, j) breaks the SCC. If the SCC does not break, we may need to recompute the sparse certificate. If the SCC breaks, then compute the new SCCs, update the implicit data structures and the *HOOK* arrays properly so that the information concerning descendant trees and hooks in the new G' is preserved, and finally apply Italiano's decremental algorithm to delete (i, j) from the new G'. The maintenance of the transitive closure in G' is done by a suitable modification of `Ital-Opt` which facilitates the splitting of SCCs.

To use `Ital-Gen` in a fully dynamic environment, further modifications and optimizations need to be made. Instead of recomputing SCCs, their sparse certificates and G' before any sequence of edge deletions, SCCs are merged to supervertices as soon as they are created. This way, the recomputation of the data structure before each sequence of edge deletions is avoided (thus speeding up mixed sequences of operations). This further allows one to adopt the lazy approach for resetting the data structure as applied in `Ital-Opt`. This concludes the description of Italiano's algorithm and its variants.

Yellin's data structure associates with every vertex v the doubly linked list *Adjacent*(v) of the heads of its outgoing edges, and the doubly linked list *Reaches*(v) of the tails of its incoming edges. In addition, an $n \times n$ array *INDEX* is maintained. Each entry *INDEX*$[v, w]$ has (among others) a field called *refcount* that stores the number of v-w paths in G, defined as *refcount*(v, w)= $|ref(v, w)| + 1$, if $(v, w) \in E$, and *refcount*(v, w)= $|ref(v, w)|$, otherwise, where $ref(v, w) = \{(v, z, w) : z \in V \wedge (v, z) \in E^* \wedge (z, w) \in E\}$.

The main idea for updating the data structure after the insertion of an edge (a, b) is to find, for all $x, z \in V$, the new triples (x, y, z) that should be added to $ref(x, z)$ and update *INDEX*$[v, w]$.*refcount*. The insertion algorithm first finds all vertices x such that (x, a) was an edge of G^*_{old} (the transitive closure graph before the insertion of (a, b)). In this case, (x, a, b) is a new triple in $ref(x, b)$, and *refcount*(x, b) has to be incremented. Then, the insertion algorithm considers each new edge (x, y) in G^*_{new} (the transitive closure graph after the insertion of (a, b)) and each edge (y, z) of G; (x, y) is a new transitive closure edge if its *refcount* was increased from 0 to 1. Now, (x, y, z) is a new triple for $ref(x, z)$ and *refcount*(x, z) is increased by 1. The edge deletion algorithm is the "dual" of the edge insertion algorithm described above.

To support `Path` queries, Yellin's algorithm has to be augmented with a rather "heavy" data structure called the *support graph*, which stores all transitive closure edges and all triples (x, y, z) in $ref(x, z)$. Consequently, it occupies $O(n^3)$ space, making this version of Yellin's algorithm very slow and space consuming, a fact that was indeed confirmed by the experiments in [1.32, 1.33]. It is also easy to see that the data structures in either version need no modification in order to be used in a fully dynamic environment.

The algorithm of Cicerone et al. [1.12] provides a uniform approach for maintaining several binary relationships (e.g., transitive closure, dominance, transitive reduction) incrementally on general digraphs and decrementally on DAGs. The main advantage of this technique, besides its simplicity, is the fact that its implementation does not depend on the particular problem; i.e., the same procedures can be used to deal with different problems by simply setting appropriate boundary conditions.

The approach allows a *propagation property* to be defined based on a binary relationship (e.g., transitive closure) $R \subseteq V \times V$ that describes how R "propagates" along the edges of a digraph $G = (V, E)$. More precisely, a relationship R satisfies the propagation property over G with boundary condition $R_0 \subset R$ if, for any pair $\langle x, y\rangle \in V \times V$, $\langle x, y\rangle \in R$ if and only if either $\langle x, y\rangle \in R_0$, or $x \neq y$ and there exists a vertex $z \neq y$ such that $\langle x, z\rangle \in R$ and $(z, y) \in E$. The relationship R_0 is used to define the set of elements of R that cannot be deduced using the propagation property. For example, if R is the transitive closure, then $R_0 = \{(x, x) : x \in V\}$. Actually, if R is the transitive closure, then the algorithm of [1.12] collapses to the algorithm of La Poutré and van Leeuwen [1.49].

The data structure maintains an integer matrix that contains, for each pair of vertices $\langle x, y\rangle \in V \times V$, the number $U_R[x, y]$ of edges useful to that pair. An edge $(z, y) \in E$ is *useful* to pair $\langle x, y\rangle$ if $z \neq y$ and $\langle x, z\rangle \in R$. Now, it is easy to see that, for any pair $\langle x, y\rangle \in V \times V$, $\langle x, y\rangle \in R$ if and only if either $\langle x, y\rangle \in R_0$ or $U_R[x, y] > 0$. In addition to the $n \times n$ integer matrix described above, and the binary matrix representing the boundary condition R_0, two additional data structures are maintained: (a) a set $\textsc{out}[x]$, for each vertex x, that contains all outgoing edges of x; and (b) a queue Q_k, for every vertex k, to handle edges (h, y) useful to pair $\langle k, y\rangle$. A `Boolean` query for vertices i and j takes $O(1)$ time, since it involves only checking the value $U_R[i, j]$.

After the insertion of edge (i, j) the number of edges useful to any pair $\langle k, y\rangle$ can only increase. An edge insertion is performed as follows: firstly, for each vertex k, the new edge (i, j) is inserted into the empty queue Q_k if and only if $\langle k, i\rangle \in R$, and hence it is useful to pair $\langle k, j\rangle$; then, edges (t, h) are extracted from queues Q_k, and the values $U_R[k, h]$ are increased by one, because these edges are useful to pair (k, h). Now, edges $(h, y) \in \textsc{out}[h]$ are inserted in Q_k if and only if the pair $\langle k, h\rangle$ has been added for the first time to R as a consequence of an edge insertion, i.e., if and only if $U_R[k, h] = 1$. This implies that, during a sequence of edge insertions, the edge (h, y) can be inserted in Q_k at most once.

The behaviour of an edge deletion operation is analogous. After the deletion of edge (i, j) some edges could no longer be useful to a pair $\langle k, y\rangle$, and then the corresponding value $U_R[k, y]$ has to be properly decreased. An edge deletion is handled as follows: firstly, for each vertex k, the deleted edge (i, j) is inserted into the empty queue Q_k if and only if $\langle k, i\rangle \in R$; then, edges (t, h) are extracted from queues Q_k, and the values $U_R[k, h]$ are decreased each time by one, because these edges are no longer useful to pair (k, h). Now, edges $(h, y) \in \textsc{out}[h]$ are inserted in Q_k if and only if $U_R[k, h] = 0$ and $(k, h) \notin R_0$, that is, when there is no edge useful to pair (k, h). This implies that, during a sequence of edge deletions, the edge (h, y) can be inserted in Q_k at most once. Notice that, if $U_R[k, h] > 0$, then (k, h) is still in R after deleting edge (i, j), because G is acyclic.

Two different variants of the algorithms in [1.12] were implemented in [1.32, 1.33]: the general technique, denoted as `CFNP`, and its specialization to the transitive closure problem, denoted as `CFNP-Opt`. The main difference between the two implementations is that after each edge insertion, the original algorithm (`CFNP`) performs at least a computation of $O(n)$ time in order to update the counters modified by that insertion; on the other hand, after an edge insertion `CFNP-Opt` starts its computation only when the inserted edge (i, j) introduces a new path between i and j (an idea borrowed from Italiano's approach). Thus, instead of the matrix of counters, `CFNP-Opt` simply maintains a binary matrix representing the transitive closure of the graph.

As with Yellin's algorithm, `CFNP` can be used in a fully dynamic environment without any modification on its data structure. On the other hand, `CFNP-Opt` cannot be used in such an environment.

The Henzinger-King algorithms [1.37] are based on the maintenance of BFS trees of vertices reachable from (or which reach) a specific distinguished vertex, and the fact that with very high probability every vertex in the graph reaches (or is reachable by) a distinguished vertex by a path of small distance (counted in number of edges).

Let $out(x, k)$ (resp. $in(x, k)$) denote the set of vertices reachable from (resp. which reach) vertex x by a path of distance at most k. The decremental algorithm, denoted as `HK-1`, selects at random sets of distinguished vertices S_i, for $i = 1, \ldots, \log n$, where $|S_i| = \min\{O(2^i \log n), n\}$. For every $x \in S_i$ the algorithm maintains (a) $out(x, n/2^i)$ and $in(x, n/2^i)$; and (b) $Out(x) = \bigcup_{i:x \in S_i} out(x, n/2^i)$ and $In(x) = \bigcup_{i:x \in S_i} in(x, n/2^i)$. In addition, for each $u \in V$ the sets $out(u, \log^2 n)$ and $in(u, \log^2 n)$ are maintained. The sets $out(x, k)$ and $in(x, k)$ are maintained in a decremental environment using a (modification of a) technique proposed by Even and Shiloach [1.21] for undirected graphs. Each set is called a *BFS structure*, since it implicitly maintains a spanning tree for the descendants of x as it would have been computed by a BFS algorithm. Hence, an edge deletion reduces to the maintenance of certain BFS structures.

A query for vertices u and v is carried out as follows. Check if v is in $out(u, \log^2 n)$. If not, then check for any vertex $x \in S$ whether $u \in In(x)$ and $v \in Out(x)$. If such an x exists, then there is a u-v path; otherwise, such a path does not exist with high probability. A `Boolean` query is answered in time proportional to $|S|$, i.e., in $O(n/\log n)$ time, and a `Path` query is answered in an additional $O(\ell)$ time, where ℓ is the length of the reported path.

The `HK-1` implementation in [1.32, 1.33] was further "engineered" to improve performance, by reducing the required space and the time to check whether there is a x-u (resp. u-x) path in an ($out(x,k)$ (resp. $in(x,k)$) set. The former is accomplished by keeping in $Out(x)$ and $In(x)$ only the tree with the largest depth among the potentially many trees having as root the same vertex. The latter is accomplished in $O(1)$ time by assigning to the vertices not in such a set a level greater than k.

The fully dynamic algorithm, denoted as `HK-2`, keeps the above decremental data structure to give answers if there is an "old" path between two vertices (i.e., a path that does not use any of the newly inserted edges). Updates are carried out as follows. After the insertion of an edge (i,j), compute $in(i,n)$ and $out(i,n)$. After the deletion of an edge, recompute $in(i,n)$ and $out(i,n)$ for all inserted edges (i,j), and update the decremental data structure for old paths. Rebuild the decremental data structure after $\sqrt{n}$ updates. To answer a query for u and v, check first if there is an old path between them. If not, then check if $u \in in(i,n)$ and $v \in out(i,n)$ for all i which are tails of the newly inserted edges (i,j).

Finally, three simple-minded (pseudo-dynamic) algorithms were implemented in [1.32, 1.33] and compared to the above dynamic algorithms; they were based on the following method: in the case of an edge insertion (resp. deletion), the new (resp. existing) edge is simply added to (resp. removed from) the graph and nothing else is computed. In the case of a query, a search from the source vertex s is performed until the target vertex t is reached (if an s-t path exists) or until all vertices reachable from s are exhausted. Depending on the search method used (DFS, BFS, and a combination of them), the three different implementations require no initialization time, $O(1)$ time per edge insertion or deletion, and $O(n+m)$ time per query operation, where m is the current number of edges in the graph. The implementation of the simple-minded algorithms include: `DFS`, `BFS`, and `DBFS`. The latter is a combination of `DFS` and `BFS` that works as follows. Vertices are visited in DFS order. Every time a vertex is visited, first check whether any of its adjacent vertices is the target vertex. If yes, then stop; otherwise, continue the visit in a DFS manner. The experimental results in [1.32, 1.33] showed that there were cases where `DBFS` outperformed `DFS` and `BFS`.

An extensive experimental study of this bulk of implementations was conducted in [1.32, 1.33]. The experiments were run on three different kinds of inputs: random inputs (aimed at identifying the average-case performance of the algorithms), non-random inputs (aimed at forcing the algorithms to exhibit their worst-case performance), and on a more pragmatic input motivated by a real world graph: the graph describing the connections among the autonomous systems of a fragment of the Internet network visible from *RIPE* (`www.ripe.net`), one of the main European servers. On this graph random sequences of operations were performed.

Random inputs consisted of randomly generated digraphs and DAGs with different edge densities ($m_0 \in \{0, n/2, n, n \ln n, n^{1.5}, n^2/\ln n, n^2/4\}$) and random sequences of operations. As it is mentioned in Section 1.2.1.2.a, the values $n/2, n, n \ln n$ are thresholds around which a fundamentally different structural behaviour of a random graph occurs [1.9] (the rest of the values are chosen as intermediate steps towards denser graphs). In the random sequences of operations, the queries were uniformly mixed with updates (insertions/deletions). Moreover, various lengths of such sequences were considered.

The non-random inputs were inspired from those proposed in [1.3] (cf. Section 1.2.1.2.a). They consisted of a number k of cliques (complete digraphs or complete DAGs), each one containing roughly n/k vertices, and which are interconnected by a set B of at most $k-1$ inter-clique edges, called "bridges". Depending on the type of updates, the edges in B are precisely those which are inserted or deleted from the graph during a sequence of operations. This forces the dynamic algorithms to handle dense subgraphs while the reachability information of the whole graph keeps changing. To make it even harder (and assuming an arbitrary numbering of the cliques), there is a specific order in which these edges were inserted (resp. deleted) in an incremental (resp. decremental) environment. In the incremental case, the graph G initially has no bridges. Bridges are added from B as follows: the first bridge is inserted between the first and the second clique, the second bridge between the penultimate and the ultimate clique, the third bridge between the second and the third clique, and so on. Hence, the bridge inserted last will provide new reachability information from roughly $n/2$ to the other $n/2$ vertices of G. The reverse order is followed in the case of edge deletions, where all edges from B were initially in G.

For random inputs, the reported experiments were as follows. In the incremental case as well as in the decremental case for DAGs, `Ital-Opt` and `Ital-Gen` were almost always the fastest. The simple-minded algorithms became competitive or faster in: (i) the incremental case when the initial graph was very sparse (containing less than $n/2$ edges); (ii) the decremental case when the initial graph was sparse (containing less than $n \ln n$ edges) and the sequence of operations was of medium to small size. This could be explained by the fact that as the graph becomes denser edge insertions do not add new information w.r.t. transitive closure, while edge deletions below the connectivity threshold ($n \ln n$ for random digraphs) increase considerably the work performed by the dynamic algorithms (e.g., Italiano's algorithms have greater difficulty to find hooks in this case) which cannot be amortized with the length of the operation sequence. In the decremental case for general digraphs, the simple-minded algorithms were always significantly faster than `HK-1` or `Ital-Gen`, because the former suffered by the updating of the BFS structures, while the latter suffered by the splitting of SCCs and the recomputation of their sparse certificates. Similar behaviour was observed in the fully dynamic case for general digraphs; an interesting fact was that `HK-2` was the slowest in practice, even for very small sequences of operations where the algorithm is assumed to perform well (i.e., does not perform a rebuild). In the fully dynamic case for DAGs, again `Ital-Opt` and `Ital-Gen` were the fastest when the initial graph was not sparse (having more than $n \ln n$ edges); in the sparse case, the simple-minded algorithms became competitive. The efficiency of `Ital-Opt` and `Ital-Gen` demonstrates that the lazy approach for resetting the hooks was indeed successful.

In the case of non-random inputs, the simple-minded algorithms were significantly faster than any of the dynamic algorithms. The best dynamic algorithms were `Ital-Opt` or `Ital-Gen`. An interesting observation for this type of inputs concerned the `HK-2` algorithm: for small values of k, it was the slowest. For larger values of k, it became competitive or faster than the best dynamic (`Ital-Opt` or `Ital-Gen`). This behaviour is due to the rebuilding of its data structure after a certain threshold in the length of the operation sequence. The larger the value of k, the less rebuilds are performed by `HK-2`.

Finally, the experiments with the fragment of the Internet graph gave similar conclusions to those obtained with random inputs. Additional experiments have been performed with operation sequences for which some knowledge about their update-query pattern is known in advance (e.g., the percentage of queries). Although the theoretical bounds of the dynamic algorithms may not hold in this case, these experiments might give useful suggestions on how to proceed if information about the update-query pattern is provided. The experimental results provided a quantitative idea of what is the break point, i.e., after which percentage of queries the dynamic algorithms overcome the simple-minded ones. As expected, the higher the percentage of queries, the worse the simple-minded algorithms. The break points, however, differ in each dynamic setting.

The source code of the above implementations is available from `http://www.ceid.upatras.gr/faculty/zaro/software`.

1.3.1.2.b ***The Implementation by Abdeddaim.*** [1.2]

The main goal of that paper was to investigate whether certain knowledge about the input digraph (in terms of the so-called "path cover") could help to speed up the computation time for maintaining the transitive closure in an incremental environment (edge insertions and queries). The motivation behind this interest lies on the fact that incremental transitive closure is a fundamental subroutine in algorithms for sequence alignment and that alignment graphs have the requested knowledge.

In [1.2], the incremental algorithms developed in [1.1, 1.2] were implemented and compared with implementations of Italiano's incremental algorithm [1.42] which appears to be among the fastest in the previous study (cf. Section 1.3.1.2.a).

The algorithms in [1.1, 1.2] assume that the input digraph is provided with a set of k vertex-disjoint paths that cover all vertices of the graph. Such a set is called a *path cover*. Finding a path cover of minimum cardinality is an NP-complete problem (see e.g., [1.45]). However, there are families of graphs for which a path cover (not necessarily minimum) can either be efficiently computed or is known in advance. The former case includes DAGs, where the computation of a minimum path cover reduces to a minimum flow problem [1.45]. The latter case includes alignment graphs which are used in greedy algorithms for sequence alignment [1.1]. Since computing a minimum flow takes roughly $O(nm)$ time (as it reduces to the max-flow problem), one can resort to other approaches to find a path cover in a DAG which may not be minimum. Such an algorithm is described in [1.51, pp. 11–12] and in [1.60], and takes linear time. This idea can be extended to general digraphs by shrinking each strongly connected component to a single vertex, thus yielding a DAG on which the algorithm of [1.51, 1.60] can be applied.

The main idea of the algorithms in [1.1, 1.2] is to use the initial path cover to compute, for each vertex x, a *predecessor* and a *successor frontier* which encode the number of predecessors and successors of x in each path, respectively. When a new edge (x, y) is inserted, it is first checked whether there is already an x-y path in the graph. If yes, then the frontiers are not modified. Otherwise, the algorithms consider each pair of paths P_i and P_j, and for each predecessor u of x in P_i, its successors' number in P_j is updated by the maximum of its current value and the number of successors of y in P_j. The predecessor frontiers are computed in a similar way. The difference between the two algorithms in [1.1, 1.2] lies in the order in which u and j are considered. The algorithm from [1.1] (resp. [1.2]) is referred to as `Abd97` (resp. `Abd99`).

In the experimental study of [1.2] two kind of inputs were considered: random inputs and alignment graphs.

The random inputs are different from those we have seen so far. They are constructed as follows: firstly, k paths of random lengths were generated by assigning to each vertex v a random label l_v in $[1, k]$ and considering vertices with the same label to belong to the same path. Secondly, a set of edges chosen uniformly at random were added.

The alignment graphs were taken from two benchmarks libraries, namely the BAliBASE [1.63] and the PFAM [1.8].

The `Abd97` and `Abd99` algorithms were implemented in `ANSI C` and compared to an implementation, called `Ita86` (also in `ANSI C`), of the original incremental algorithm by Italiano [1.42] that was developed in [1.2], and to the `Ital-Opt` and `CFNP-Opt` implementations from [1.32]. Note that the latter two are `C++` implementations using LEDA [1.52] and hence expected to be slower than an `ANSI C` implementation. From the theoretical analysis of `Abd97` and `Abd99` (cf. Section 1.3.1.1), it turns out that for small values of k and sufficiently large operation sequences both algorithms achieve a very good amortized time per update operation. This was exactly confirmed by the experiments in [1.2]. For both types of inputs, `Abd97` and `Abd99` were faster than the other algorithms, with `Abd97` being usually the fastest. When the value of k was getting larger, or the graph was becoming denser, `Abd99` was taking over.

It would be interesting to investigate the performance of `Abd97` and `Abd99` on the data sets developed in [1.32, 1.33].

1.3.1.3 Lessons learned. The above experimental studies allow us to obtain a better understanding of existing dynamic algorithms for transitive closure and their practical assessment, and also to detect certain differences in their practical behaviour compared with that of their undirected counterparts for dynamic connectivity. In particular:

- In partially dynamic environments on unstructured inputs, dynamic algorithms are usually faster than simpler approaches (based on static algorithms). On the other hand, the simpler approaches are considerably better than dynamic algorithms either in fully dynamic environments on unstructured inputs, or in any kind of dynamic environment on structured inputs.
- To beat the simpler approaches, thorough experimentation, fine-tuning, and sophisticated engineering of dynamic algorithms is required.
- Certain knowledge about the input digraph or the update sequence turns out to be useful in practice.
- The test suite of [1.32, 1.33], generated by a methodology analogous to that in [1.3, 1.4], along with the enhancement of more pragmatic inputs (e.g., inputs from benchmark libraries, real-world inputs, etc) can be considered as a valuable benchmark for testing other dynamic algorithms for directed graphs.

1.3.2 Dynamic Shortest Paths

1.3.2.1 Theoretical background – problem and history of results. The shortest path problem consists in finding paths of minimum total weight between specified pairs of vertices in a digraph $G = (V, E)$ whose edges are associated with real-valued weights. A path of minimum total weight between two vertices x and y is called *shortest path*; the weight of a shortest x-y path is called the *distance* from x to y. There are two main versions of the shortest path problem: the *all-pairs shortest paths* (APSP) in which we seek shortest paths between every pair of vertices in G; and the *single-source shortest path* (SSSP) in which we seek shortest paths from a specific vertex s to all other vertices in G.

The *dynamic shortest path* problem consists in building a data structure that supports query and update operations. A shortest path (resp. distance) query specifies two vertices and asks for the shortest path (resp. distance) between them. An update operation updates the data structure after an edge insertion or edge deletion or edge weight modification. There are several algorithms for both the dynamic APSP and the dynamic SSSP problems. Actually, dynamic shortest path problems have been studied since 1967 [1.50, 1.54, 1.58]. In the following, let $n = |V|$ and let m_0 denote the initial number of edges in G.

For the dynamic APSP problem and the case of arbitrary real-valued edge weights, Even & Gazit [1.20] and Rohnert [1.59] gave (independently) two fully dynamic algorithms in 1985. Both algorithms create a data structure which is initialized in $O(nm_0 + n^2 \log n)$ time, supports shortest path or distance queries optimally, and is updated either in $O(n^2)$ time after an edge insertion or edge weight decrease, or in $O(nm + n^2 \log n)$ time after an edge deletion or edge weight increase (m being the current number of edges in the graph). These were considered the best algorithms for dynamic APSP on general digraphs with arbitrary real-valued edge weights, until a very recent breakthrough achieved by Demetrescu & Italiano [1.14]: if each edge weight can assume at most S different real values, then any update operation can be accomplished deterministically in $O(Sn^{2.5} \log^3 n)$ amortized time and a distance query in $O(1)$ time. In the same paper, an incremental randomized algorithm (with one-sided error) is given which supports an update in $O(Sn \log^3 n)$ amortized time.

In the case where the edge weights are nonnegative integers, a number of results were known. Let C denote the largest (integer) value of an edge weight. In [1.7], Ausiello et al. gave an incremental algorithm that supports queries optimally, and updates its data structure in $O(Cn^3 \log(nC))$ time after a sequence of at most $O(n^2)$ edge insertions or at most $O(Cn^2)$ edge weight decreases. More recently, a fully dynamic algorithm was given by King [1.47] which supports queries optimally, and updates (edge insertions or deletions) in $O(n^{2.5}\sqrt{C \log n})$ amortized time (amortized over a sequence of operations of length $\Omega(m_0/n)$). Also, in the same paper a decremental algorithm is given which supports any number of edge deletions in $O(m_0 n^2 C)$ time (i.e., in $O(n^2 C)$ amortized time per deletion if there are $\Omega(m_0)$ deletions). We note that very efficient dynamic algorithms are known for special classes of digraphs (planar, outerplanar, digraphs of small treewidth, and digraphs of small genus) with arbitrary edge weights; see [1.11, 1.17].

The efficient solution of the dynamic SSSP problem is a more difficult task, since almost optimal algorithms are known for the static version of the problem. Nothing better than recomputing from scratch is known about the dynamic SSSP problem, with the exception of a decremental algorithm presented in [1.24]. That algorithm assumes integral edge weights in $[1..C]$ and supports any sequence of edge deletions in $O(m_0 nC)$ time (i.e., $O(nC)$ amortized time per deletion, if there are $\Omega(m_0)$ deletions).

For the above and other reasons most of the research for the dynamic SSSP problem has been concentrated on different computational models. One such model is the *output complexity cost model* introduced by Ramalingam & Reps [1.56, 1.57] and extended by Frigioni et al. in [1.29, 1.31]. In this model, the time-cost of a dynamic algorithm is measured as a function of the number of updates to the output information of the problem caused by input updates.

Let δ denote an input update (edge insertion, edge deletion, or edge weight modification) to be performed on the given digraph G, and let V_δ be the set of *affected vertices*, i.e., the vertices that change their output value as a consequence of δ (e.g., for the SSSP problem their distance from the source s). In [1.56, 1.57], the cost of a dynamic algorithm is measured in terms of the *extended size* $\|\delta\|$ of the change in input and output. Parameter $\|\delta\|$ equals the sum of $|V_\delta|$ and the number of edges that have at least one affected endpoint. Note that $\|\delta\|$ can be $O(m)$ in the worst-case, and that both $\|\delta\|$ and $|V_\delta|$ depend only on the problem instance. In [1.56, 1.57] a fully dynamic algorithm is provided that updates its data structure after a change δ in the input in time $O(\|\delta\| + |V_\delta| \log |V_\delta|)$. Queries are answered optimally.

In [1.29], the cost of a dynamic algorithm is measured in terms of the number of changes on the *output information* of the problem. In the case of the SSSP problem, the output information is the distances of the vertices from s and the shortest path tree. The output complexity cost is measured in this case as a function of the *number of output updates* $|U_\delta|$, where U_δ (the set of output updates) consists of those vertices which, as a consequence of δ, either change their distance from s, or must change their parent in the shortest path tree (even if they maintain the same distance). Differently from the model of [1.56, 1.57], U_δ depends on the current shortest path tree (i.e., on the algorithm used to produce it.) In [1.29] an incremental algorithm for the dynamic SSSP is given which supports queries optimally and updates its data structure in time $O(k|U_\delta| \log n)$ where k is a parameter bounded by structural properties of the graph. For general graphs $k = O(\sqrt{m})$, but for special classes of graphs it can be smaller (e.g., in planar graphs $k \leq 3$). A fully dynamic algorithm with the same update and query bounds is achieved in [1.31].

All the above algorithms [1.29, 1.31, 1.56, 1.57] for the dynamic SSSP problem require that the edge weights are nonnegative and that there are no zero-weighted cycles in the graph either before or after an input update δ. These restrictions have been waived in [1.30] where a fully dynamic algorithm is presented that answers queries optimally and updates its data structure in $O(\min\{m, kn_a\} \log n)$ time after an edge weight decrease (or edge insertion), and in $O(\min\{m \log n, k(n_a + n_b) \log n + n_c\})$ time after an edge weight increase (or edge deletion). Here, n_a is the number of affected vertices, n_b is the number of vertices that preserve their distance from s but change their parent in the shortest path tree, and n_c is the number of vertices that preserve both their distance from s and their parent in the shortest path tree.

1.3.2.2 Implementations and Experimental Studies. There are two works known regarding implementation and experimental studies of dynamic algorithms for shortest paths. In chronological order, these are the works by Frigioni et al. [1.28] and by Demetrescu et al. [1.15].

Both studies deal with the dynamic SSSP problem and aim at identifying the practicality of dynamic algorithms over static approaches. The former paper investigates the case of non-negative edge weights, while the latter investigates the case of arbitrary edge weights.

1.3.2.2.a ***The Implementation by Frigioni et al.*** [1.28]

The main goal of the first experimental study on dynamic shortest paths was to investigate the practicality of fully dynamic algorithms over static ones, and to experimentally validate the usefulness of the output complexity model.

In particular, the fully dynamic algorithms by Ramalingam & Reps [1.56] and by Frigioni et al. [1.31] have been implemented and compared with a simple-minded, pseudo-dynamic algorithm based on Dijkstra's algorithm. Both fully dynamic algorithms are also based on suitable modifications of Dijkstra's algorithm [1.16]. Their difference lies in how the outgoing edges of a vertex are processed when its distance from s changes. In the following, let $d(v)$ denote the current distance of a vertex v from s and let $c(u,v)$ denote the weight of edge (u,v).

The algorithm of Ramalingam & Reps [1.56] maintains a DAG $SP(G)$ containing all vertices of the input graph G and exactly those edges that belong to at least one shortest path from s to all other vertices of G.

In the case of an edge insertion, the algorithm proceeds in a Dijkstra-like manner on the vertices affected by the insertion. Let (v,w) be the inserted edge. The algorithm stores the vertices in a priority queue Q with priority equal to their distance from w. When a vertex x of minimum priority is deleted from Q, then all of its outgoing edges (x,y) are traversed. A vertex y is inserted in Q, or its priority is updated, if $d(x)+c(x,y)<d(y)$. In such a case, (x,y) is added to $SP(G)$ and all incoming edges of y are deleted from $SP(G)$. If $d(x)+c(x,y)=d(y)$, then (x,y) is simply added to $SP(G)$.

In the case of an edge deletion, the algorithm proceeds in two phases. Let A (resp. B) denote the set of affected (resp. unaffected) vertices. In the first phase the set A of affected vertices is determined by performing a kind of topological sorting on $SP(G)$. Let (v,w) be the deleted edge. Vertex w is put into A, if its indegree in $SP(G)$ is zero after the deletion of (v,w). If $w \in A$, then all of its outgoing edges are deleted from $SP(G)$. If this yields new vertices of zero indegree, then they added to A, and their outgoing edges are deleted from $SP(G)$. The process is repeated until all vertices are exhausted or there are no more vertices of zero indegree in $SP(G)$. In the second phase, the new distances of the vertices in A are determined. This is done by first shrinking the subgraph induced by B on a new super-source s', and then by adding, for each edge (x,y) with $x \in B$ and $y \in A$, the edge (s',y) with weight equal to $d(x)+c(x,y)$. Finally, run Dijkstra's algorithm with source s' on the resulting graph and update $SP(G)$ as in the case of edge insertion. The implementation of the above algorithm is referred to as `RR` in [1.28].

The algorithm by Frigioni et al. [1.31] is based on a similar idea, but an additional data structure is maintained on each vertex v in order to "guess" which neighbors of v have to be updated when the distance from v to the source changes. This data structure is based on the notions of the *level* and the *owner* of an edge. The *backward level* of an edge (y,z), associated with vertex

z, is defined as $BL_y(z) = d(z) - c(y, z)$; the *forward level* of an edge (x, y), associated with vertex x, is defined as $FL_y(x) = d(x) + c(x, y)$. Intuitively, these levels provide information about the shortest available path from s to z that passes through y. The *owner* of an edge is one of its two endpoints, but not both. The incoming and outgoing edges of a vertex y is partitioned into those owned by y and into those not owned by y. The incoming and outgoing edges not owned by y are stored in two priority queues F_y (for the incoming) and B_y (for the outgoing) with priorities determined by their forward and backward levels, respectively. Any time a vertex y changes its distance from s, the algorithm traverses all edges owned by y and an appropriately chosen subset of edges not owned by y.

When the insertion of an edge (v, w) decreases $d(w)$, then a priority queue Q' is used, as in Dijkstra's algorithm, to find new distances from s. However, differently from Dijkstra's and the `RR` algorithm, when a vertex y is deleted from Q' and its new distance decreases, only those not-owned outgoing edges (y, z) are scanned whose priority in B_y is greater than the new $d(y)$, since only in this case (i.e., $BL_y(z) = d(z) - c(y, z) > d(y)$) $d(z)$ is decreased by a shortest s-z path that passes through y.

In the case of an edge deletion, the algorithm proceeds in two phases (like `RR`). In the first phase, the affected vertices are determined. When a vertex y increases its distance due to the edge deletion, then in order to find the best possible alternative shortest s-y path, only those not-owned incoming edges (x, y) are scanned whose priority in F_y is smaller than $d(y)$. The vertex x which minimizes $FL_y(x) - d(y)$ is the new parent of y in the shortest path tree. The increase in $d(y)$ is propagated to the outgoing edges owned by y. In the second phase, the distances of the affected vertices are computed by performing a Dijkstra-like computation on the subgraph induced by those vertices and by considering only edges between affected vertices. The implementation of the above algorithm is referred to as `FMN` in [1.28].

Finally, a pseudo-dynamic algorithm, called `Dij`, was implemented based on LEDA's implementation of Dijkstra's algorithm. It simply recomputes from scratch the shortest path information, only when an input update affects this information. Since Dijkstra's implementation in LEDA uses Fibonacci heaps, all priority queues implemented in `RR` and `FMN` are also Fibonacci heaps.

The implementations `RR`, `FMN`, and `Dij` were experimentally compared in [1.28] using three kinds of inputs: random inputs, structured inputs, and on the graph describing the connections among the autonomous systems of a fragment of the Internet network visible from *RIPE* (`www.ripe.net`), one of the main European servers.

For random inputs, two types of operation sequences were performed on random graphs: randomly generated sequences of updates, and modifying sequences of updates. In the latter type, an operation is selected uniformly at random among those which actually modify some shortest path from the source. Edge weights are chosen randomly.

The structured input consisted of a special graph and a specific update sequence on that graph. The graph consists of a source s, a sink t, and a set X of k other vertices $x_1, \ldots, x_k$. The edge set consists of edges $(s, x_i), (x_i, s), (t, x_i)$ and (x_i, t), for $1 \leq i \leq k$. The sequence of updates consists of alternated insertions and deletions of the single edge (s, t) with a proper edge weight. The motivation for this input was to exhibit experimentally the difference between the complexity parameters $\|\delta\|$ used by `RR` and $|U_\delta|$ used by `FMN`, since the theoretical models proposed by [1.56] and [1.29] are different and do not allow for a direct comparison of these parameters. Clearly, with this input after each dynamic operation only the distance of t changes. Hence, it is expected that as the size of the neighborhood of the affected vertices increases, `FMN` should dominate over `RR`: after the insertion (resp. deletion) of (s, t), `RR` visits all k edges outgoing from (resp. incoming to) t, while `FMN` visits only those edges "owned" by t.

The input based on the fragment of the Internet graph consists of unary weights and random update sequences.

In all experiments performed with any kind of input, both `RR` and `FMN` were substantially faster than `Dij` (although in the worst-case the bounds of all algorithms are identical). In the case of random inputs, `RR` was faster than `FMN` regardless of the type of the operation sequence. In the cases of structured input and of the input with the fragment of the Internet graph, `FMN` was better. An interesting observation was that on any kind of input, the edges scanned by `FMN` were much less than those scanned by `RR` (as expected). However, `FMN` uses more complex data structures which, in the case of random inputs, eliminate this advantage.

The source code of the above implementations is available from `http://www.jea.acm.org/1998/FrigioniDynamic`.

1.3.2.2.b ***The Implementation by Demetrescu et al.*** [1.15]

The main goal of that paper was to investigate the practical performance of fully dynamic algorithms for the SSSP problem in the case of digraphs with arbitrary edge weights.

In particular, the algorithms considered were the recent fully dynamic algorithm by Frigioni et al. [1.30], referred to as `FMN-gen`; a simplified version of it, called `DFMN`; a variant of the `RR` algorithm, referred to as `RR-gen` [1.57] which works with arbitrary edge weights; and a new simple dynamic algorithm, referred to as `DF`.

The common idea behind all these algorithms is to use the Edmonds-Karp technique [1.18] to transform an SSSP problem with arbitrary edge weights to another one with nonnegative edge weights without changing the shortest paths. This is done by replacing each edge weight $c(x, y)$ by its reduced version $r(x, y) = d(x) - d(y) + c(x, y)$ (the distances $d(\cdot)$ are provided by the input shortest path tree), running Dijkstra's algorithm to the graph with the reduced edge weights (which are nonnegative), and then trivially obtain the actual distances from those based on the reduced weights.

In the case of an edge insertion or weight decrease operation, `FMN-gen` and `DFMN` behave similarly to `FMN` (cf. Section 1.3.2.2.a), while `DF` and `RR-gen` behave similarly to `RR` (cf. Section 1.3.2.2.a). However, `DF` has not been designed to be efficient according to the output complexity model as `RR` had, and its worst-case complexity is $O(m + n \log n)$.

In the case of an edge deletion or weight increase operation, there are differences in the algorithms. Algorithm `FMN-gen` proceeds similarly to `FMN` (cf. Section 1.3.2.2.a), but the traversal of the not-owned incoming edges becomes more complicated as zero-weighted cycles should be handled. The `DFMN` algorithm is basically the `FMN-gen` algorithm without the partition of the incoming and outgoing edges into owned and not-owned. This allows for simpler and, as experiments showed, faster code. Finally, the `DF` algorithm, as in the case of edge insertion or weight decrease operation, uses the classical complexity model and not the output complexity one, and its worst-case complexity is $O(m+n \log n)$. Let (x, y) be the edge whose weight is increased by a positive amount Δ. The algorithm consists of two phases, called *initializing* and *updating*. In the initializing phase, all vertices in the subtree $T(y)$ of the shortest path tree rooted at y are marked. Each marked vertex v finds its "best" unmarked neighbor u in its list of incoming edges. This yields a (not necessarily shortest) s-v path whose weight, however, is used as the initial priority of v (i.e., of an affected vertex) in a priority queue H used in the updating phase. If u is not `nil` and $d(u) + c(u, v) - d(v) < \Delta$, then the priority of v equals $d(u)+c(u, v)-d(v)$; otherwise, it equals Δ. In either case, the initial priority is an upper bound on the actual distance. The updating phase simply runs Dijkstra's algorithms on the marked vertices inserted in H with the above initial priorities.

The experiments in [1.15] were conducted only on random inputs. In particular, they were performed on randomly generated digraphs and various update sequences, which enhance in several ways the random inputs considered in [1.28] (cf. Section 1.3.2.2.a).

Random digraphs were generated such that all vertices are reachable from the source and edge weights are randomly selected from a predetermined interval. The random digraphs come in two variants: those forming no negative or zero weight cycles, and those in which all cycles have weight zero.

The update sequences were random update sequences (uniformly mixed sequences of edge increase and decrease operations that do not introduce negative or zero weight cycles), modifying update sequences (an operation is selected uniformly at random among those which actually modify some shortest path from the source), and alternate update sequences (updates alternate between edge weight increase and decrease operations and each consecutive pair of increase-decrease operation is performed on the same edge).

In all experiments, `FMN-gen` was substantially slower than `DFMN`, since it uses more complex data structures. In the experiments with arbitrary edge weights, but no zero-weighted cycles, `DF` was the fastest algorithm followed by `RR-gen`; `DFMN` is penalized by its additional effort to identify affected vertices in a graph that may have zero-weighted cycles. It is interesting to observe that `RR-gen` is slightly faster than `DF` when the range of values of the edge weights is small. In the case of inputs which included zero-weighted cycles, either in the initial graph or because of a specific update sequence which tried to force cycles in the graph to have weight zero, `DFMN` outperformed `DF`. Note that in this case `RR-gen` is not applicable.

The source code of the above implementations is available from `ftp://www.dis.uniroma1.it/pub/demetres/experim/dsplib-1.1`.

1.3.2.3 Lessons learned. The experimental studies in [1.28] and [1.15] enhance our knowledge on the practicality of several algorithms for the dynamic SSSP problem. In particular:

- The output cost model is not only theoretically interesting, but appears to be quite useful in practice.
- Fully dynamic algorithms for the SSSP problem compare favorably in practice to almost optimal static approaches.
- The random test suite developed initially in [1.28] and considerably expanded and elaborated in [1.15] provides an important benchmark of random inputs for future experimental studies with dynamic shortest path algorithms.

1.4 A Software Library for Dynamic Graph Algorithms

A systematic effort to build a software repository of implementations of dynamic graph algorithms has been recently initiated in [1.5].

A library of dynamic algorithms has been developed, written in `C++` using LEDA, and is provided as the *LEDA Extension Package on Dynamic Graph Algorithms* (LEP-DGA). The library includes several implementations of simple and sophisticated dynamic algorithms for connectivity, minimum spanning trees, single-source and all-pairs shortest paths, and transitive closure. Actually, the afore mentioned implementations of dynamic connectivity in [1.3] (cf. Section 1.2.1.2.a), dynamic minimum spanning tree in [1.4] (cf. Section 1.2.2.2.a), dynamic transitive closure in [1.32, 1.33] (cf. Section 1.3.1.2.a), and dynamic single-source shortest paths in [1.28] (cf. Section 1.3.2.2.a), are part of the LEP-DGA.

All implementations in the library are accompanied by several demo programs, experimentation platforms, as well as correctness checkers. The library is easily adaptable and extensible, and is available for non-commercial use from `http://www.mpi-sb.mpg.de/LEDA/friends/dyngraph.html`.

All dynamic data structures in the LEP-DGA are implemented as `C++` classes derived from a common base class `dga_base`. This base class defines a common interface for all dynamic algorithms. Except for the usual goals of efficiency, ease of use, extensibility, etc, special attention has been drawn on some domain specific design issues. Two main problems arose in the implementation of the library.

- *Missing Update Operations:* Dynamic algorithms usually support only a subset of all possible update operations, e.g., most dynamic graph algorithms cannot handle single vertex deletions and insertions.
- *Maintaining Consistency:* In an application, a dynamic graph algorithm D may run in the background while the graph changes due to a procedure P which is not aware of D. Consequently, there has to be a means of keeping D consistent with the current graph, because P will not use a possible interface for changing the graph provided by D, but will use the graph directly. Whether D exists or not should have no impact on P.

It was decided to support all update operations for convenience. Those updates which are not supported by the theoretical background are implemented by reinitializing the data structure for the new graph. This may not be very efficient, but it is better than exiting the whole application. The documentation tells the users which updates are supported efficiently or not. The fact that the user calls an update which theoretically is not supported results only in a (perhaps very small) performance penalty. This enhances the robustness of the applications using the library or alternatively reduces the complexity of handling exceptional situations.

An obvious approach to maintain consistency between a graph and a dynamic data structure D working on that graph is to derive D from the graph class. However, this may not be very flexible. In the case where there are more than one dynamic graph data structures working on the same graph, things could get quite complicated with this approach. Instead, the following approach was used, motivated by the observer design pattern of Gamma et al. [1.34]. A new graph type `msg_graph` has been created which sends messages to interested third parties whenever an update occurs. The base class `dga_base` of all dynamic graph algorithms is one such third party; it receives these messages and calls the appropriate update operations which are virtual methods appropriately redefined by the specific implementations of dynamic graph algorithms.

1.5 Conclusions

We have surveyed several experimental studies which investigate the practicality of dynamic algorithms for fundamental problems in graphs. These studies try to exhibit advantages and limitations of important techniques and algorithms, and to identify the best algorithm for a given input.

In all studies considered, it was evident that sophisticated engineering and fine-tuning of dynamic algorithms is often required to make them competitive or better than simpler, pseudo-dynamic approaches based on static algorithms. Moreover, there were cases where the simpler approaches cannot be beaten by any dynamic algorithm.

In an attempt to draw some rough conclusions on the practicality of dynamic algorithms, we could say that for problems in non-sparse unstructured (random) inputs involving either undirected or directed graphs and operation sequences that are not very small, the dynamic algorithms are usually better than simpler, pseudo-dynamic approaches. In the case of more structured (non-random) inputs, there is a distinction in the behaviour depending on whether the input graph is directed or not. In the latter case, the dynamic algorithms dominate the simpler approaches, while in the former we witness a reverse situation (the simpler algorithms outperform the dynamic ones).

The experimental methodology followed in most papers allows us to sketch some rough guidelines that could be useful in future studies:

- The data sets should be carefully designed to include both unstructured (random) inputs and more structured inputs that include semi-random inputs, pragmatic inputs, and worst-case inputs.
- In any given data set, several values of the input parameters (e.g., number of vertices and edges, length of the operation sequence) should be considered. It was clear from the surveyed experimental studies that several algorithms do not exhibit a stable behaviour and their performance depends on the input parameters. For example, most update bounds are amortized; consequently, the length of the operation sequence turns out to be an important parameter as it clearly determines how well the update bound is amortized in the conducted experiment. In all cases, the measured quantities (usually the CPU time) should be averaged over several samples in order to reduce variance.
- It is important to carefully select the hardware platform upon which the experiments will be carried out. This does not only involve memory issues that eventually appear when dealing with large inputs, but also allows investigation of the practical performance of dynamic algorithms on small inputs. For example, in the latter case it is often necessary to resort to slower machines in order to be able to exhibit the difference among the algorithms.

The experimental methodology followed and the way the test suites developed and evolved in the various studies (usually building upon and enhancing previous test sets) constitute an important guide for future implementors and experimentors of dynamic graph algorithms.

Acknowledgements. The author would like to thank the anonymous referees for several helpful suggestions and comments that improved the paper.

The author is also indebted to Umberto Ferraro and Pino Italiano for various clarifications regarding their work.

References

1.1 S. Abdeddaim. On Incremental Computation of Transitive Closure and Greedy Alignment. In *Proc. 8th Symp. on Combinatorial Pattern Matching – CPM'97*, *Lecture Notes in Computer Science*, Vol. 1264 (Springer-Verlag, 1997), pp. 167-179.
1.2 S. Abdeddaim. Algorithms and Experiments on Transitive Closure, Path Cover and Multiple Sequence Alignment. In *Proc. 2nd Workshop on Algorithm Engineering and Experiments* – ALENEX 2000, pp. 157–169, 2000.
1.3 D. Alberts, G. Cattaneo, and G. F. Italiano. An Empirical Study of Dynamic Graph Algorithms. *ACM Journal of Experimental Algorithmics*, 2:5, 1997. Preliminary version in Proc. SODA'96.
1.4 G. Amato, G. Cattaneo, and G. F. Italiano. Experimental Analysis of Dynamic Minimum Spanning Tree Algorithms. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms* – SODA'97, pp. 314–323, 1997.
1.5 D. Alberts, G. Cattaneo, G.F. Italiano, U. Nanni, and C. Zaroliagis. A Software Library of Dynamic Graph Algorithms. In *Proc. Workshop on Algorithms and Experiments* – ALEX'98, pp. 129–136, 1998.
1.6 C. Aragon and R. Seidel. Randomized Search Trees. In *Proc. 30th Symp. on Foundations of Computer Science* – FOCS'89, pp. 540-545, 1989.
1.7 G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental Algorithms for Minimal Length Paths. *Journal of Algorithms*, 12:615–638, 1991.
1.8 A. Bateman, E. Birney, R. Durbin, S. Eddy, K. Howe, and E. Sonnhammer. The PFAM Protein Families Database. *Nucleid Acids Research*, 28:263–266, 2000.
1.9 B. Bollobás. *Random Graphs.* Academic Press, New York, 1985.
1.10 G. Cattaneo, P. Faruolo, U. Ferraro-Petrillo, and G.F. Italiano. Maintaining Dynamic Minimum Spanning Trees: An Experimental Study. In *Proc. 4th Workshop on Algorithm Engineering and Experiments* – ALENEX 2002, *Lecture Notes in Computer Science* (Springer-Verlag 2002), to appear.
1.11 S. Chaudhuri and C. Zaroliagis. Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica*, 27:212–226, 2000.
1.12 S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A Uniform Approach to Semi Dynamic Problems in Digraphs. *Theoretical Computer Science*, 203 (1):69–90, 1998.
1.13 C. Demetrescu and G. F. Italiano. Fully Dynamic Transitive Closure: Breaking through the $O(n^2)$ Barrier. In *Proc. 41st IEEE Symp. on Foundations of Computer Science* – FOCS 2000, pp. 381–389, 2000.
1.14 C. Demetrescu and G. F. Italiano. Fully Dynamic All Pairs Shortest Paths with Real Edge Weights. In *Proc. 42nd IEEE Symp. on Foundations of Computer Science* – FOCS 2001, pp. 260–267, 2001.
1.15 C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U.Nanni. Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study. In *Proc. 4th Workshop on Algorithm Engineering* – WAE 2000, *Lecture Notes in Computer Science* (Springer-Verlag 2000), to appear.
1.16 E. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

1.17 H. Djidjev, G. Pantziou, and C. Zaroliagis. Improved Algorithms for Dynamic Shortest Paths. *Algorithmica*, 28:367–389, 2000.
1.18 J. Edmonds and R. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19:248–264, 1972.
1.19 D. Eppstein, Z. Galil, G.F. Italiano, A. Nissenzweig. Sparsification – A Technique for Speeding Up Dynamic Graph Algorithms. *Journal of the ACM*, 44:669–696, 1997. Preliminary version in Proc. FOCS'92.
1.20 S. Even and H. Gazit. Updating Distances in Dynamic Graphs. *Methods of Operations Research*, 49:371–387, 1985.
1.21 S. Even and Y. Shiloach. An On-Line Edge Deletion Problem. *Journal of the ACM*, 28:1–4, 1981.
1.22 P. Fatourou, P. Spirakis, P. Zarafidis, and A. Zoura. Implementation and Experimental Evaluation of Graph Connectivity Algorithms using LEDA. In *Proc. 3rd Workshop on Algorithm Engineering* – WAE'99, *Lecture Notes in Computer Science*, vol. 1668 (Springer-Verlag, 1999), pp. 124–138.
1.23 U. Ferraro-Petrillo. Personal Communication, February 2002.
1.24 P. Franciosa, D. Frigioni, and R. Giaccio. Semi-dynamic Shortest Paths and Breadth-First Search on Digraphs. In *Proc. Symp.on Theorecial Aspects of Computer Science* – STACS'97, *Lecture Notes in Computer Science*, vol. 1200 (Springer-Verlag, 1997), pp. 33–46.
1.25 G.N. Frederickson. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *SIAM Journal on Computing*, 14:781–798, 1985.
1.26 G.N. Frederickson. Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and k Smallest Spanning Trees. In *Proc. 32nd IEEE Symp. on Foundations of Computing* – FOCS'91, pp.632–641, 1991.
1.27 M.Fredman and M.R. Henzinger. Lower Bounds for Fully Dynamic Connectivity Problems in Graphs. *Algorithmica*, 22(3):351–362, 1998.
1.28 D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Experimental Analysis of Dynamic Algorithms for the Single Source Shortest Paths Problem. *ACM Journal of Experimental Algorithmics*, 3:5, 1998.
1.29 D. Frigioni, A. Marchetti-Spaccamela, and U.Nanni. Semi-Dynamic Algorithms for Maintaining Single-Source Shortest Path Trees. *Algorithmica*, 22(3):250–274, 1998.
1.30 D. Frigioni, A. Marchetti-Spaccamela, and U.Nanni. Fully Dynamic Shortest Paths and Negative Cycles Detection on Digraphs with Arbitrary Arc Weights. In *Proc. 6th European Symposium on Algorithms* – ESA'98, *Lecture Notes in Computer Science*, vol. 1461 (Springer-Verlag, 1998), pp. 320-331.
1.31 D. Frigioni, A. Marchetti-Spaccamela, and U.Nanni. Fully Dynamic Algorithms for Maintaining Shortest Paths Trees. *Journal of Algorithms*, 34(2):351–381, 2000. Preliminary version in Proc. SODA'96.
1.32 D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Schäfer, and C. Zaroliagis. An Experimental Study of Dynamic Algorithms for Directed Graphs. In *Proc. 6th European Symposium on Algorithms* – ESA'98, *Lecture Notes in Computer Science*, vol. 1461 (Springer-Verlag, 1998), pp. 368-380.
1.33 D. Frigioni, T. Miller, U. Nanni, and C. Zaroliagis. An Experimental Study of Dynamic Algorithms for Transitive Closure. *ACM Journal of Experimental Algorithmics*, 6, 2001.
1.34 E. Gamma, R. Helm, R. Johnson, J.M. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison-Wesley, 1995.
1.35 D. Harel. On-Line Maintenance of the Connected Components of Dynamic Graphs. Manuscript, 1982.

1.36 M.R. Henzinger and V. King. Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation. In *Proc. 27th ACM Symposium on Theory of Computing* – STOC'95, pp. 519–527, 1995.
1.37 M.R. Henzinger and V. King. Fully Dynamic Biconnectivity and Transitive Closure. In *Proc. 36th IEEE Symposium on Foundations of Computer Science* – FOCS'95, pp. 664–672, 1995.
1.38 M.R. Henzinger and V. King. Maintaining Minimum Spanning Trees in Dynamic Graphs. In *Proc. 24th Int'l Colloquium on Automata, Languages, and Programming* – ICALP'97, *Lecture Notes in Computer Science*, vol. 1256 (Springer-Verlag, 1997), pp. 594-604.
1.39 M.R. Henzinger and M. Thorup. Sampling to Provide or to Bound: with Applications to Fully Dynamic Graph Algorithms. *Random Structures and Algorithms*, 11:369–379, 1997. Preliminary version in Proc. ICALP'96.
1.40 J. Holm, K. de Lichtenberg, and M. Thorup. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. In *Proc. 30th ACM Symp. on Theory of Computing* – STOC'98, pp. 79–89, 1998.
1.41 T. Ibaraki and N. Katoh. On-Line Computation of Transitive Closure of Graphs. *Information Processing Letters*, 16:95–97, 1983.
1.42 G. F. Italiano. Amortized Efficiency of a Path Retrieval Data Structure. *Theoretical Computer Science*, 48:273–281, 1986.
1.43 G. F. Italiano. Finding Paths and Deleting Edges in Directed Acyclic Graphs. *Information Processing Letters*, 28:5–11, 1988.
1.44 R. Iyer, D. Karger, H. Rahul, and M. Thorup. An Experimental Study of Poly-Logarithmic Fully-Dynamic Connectivity Algorithms. In *Proc. 2nd Workshop on Algorithm Engineering and Experiments* – ALENEX 2000, pp. 59–78, 2000.
1.45 H. Jagadish. A Compression Technique to Materialize Transitive Closure. *ACM Trans. on Database Systems*, 15(4):558–598, 1990.
1.46 S. Khanna, R. Motwani, and R. Wilson. On Certificates and Lookahead in Dynamic Graph Problems. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms* – SODA'96, pp. 222–231, 1996.
1.47 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, pp. 81–91, 1999.
1.48 V. King, and G. Sagert. A Fully Dynamic Algorithm for Maintaining the Transitive Closure. In *Proc. 31st ACM Symposium on Theory of Computing* – STOC'99, pp. 492–498, 1999.
1.49 J. A. La Poutré, and J. van Leeuwen. Maintenance of Transitive Closure and Transitive Reduction of Graphs. In *Proc. 14th Workshop on Graph-Theoretic Concepts in Computer Science* – WG'88, *Lecture Notes in Computer Science*, vol. 314 (Springer-Verlag 1988), pp. 106–120.
1.50 P. Loubal. A Network Evaluation Procedure. *Highway Research Record*, 205 (1967):96-109.
1.51 K. Mehlhorn. *Data Structures and Algorithms. Vol.2: Graph algorithms and NP-completeness.* Springer-Verlag, 1984.
1.52 K. Mehlhorn and S. Naher. *LEDA - A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.
1.53 P.B. Miltersen, S. Subramanian, J. Vitter, and R. Tamassia. Complexity Models for Incremental Computation. *Theoretical Computer Science*, 130(1):203–236, 1994.
1.54 J. Murchland. The Effect of Increasing or Decreasing the Length of a Single Arc on All Shortest Distances in a Graph. Technical Report, LBS-TNT-26, London Business School, Transport Network Theory Unit, London, UK, 1967.

1.55 S. Nikoletseas, J. Reif, P. Spirakis, and M. Yung. Stochastic Graphs have Short Memory: Fully Dynamic Connectivity in Poly-Log Expected Time. In *Proc. 22nd Int'l Colloquium on Automata, Languages, and Programming* – ICALP'95, *Lecture Notes in Computer Science*, vol. 944 (Springer-Verlag, 1995), pp. 159-170.
1.56 G. Ramalingam and T. Reps. On the Computational Complexity of Dynamic Graph Problems. *Theoretical Computer Science*, 158:233–277, 1996.
1.57 G. Ramalingam and T. Reps. An Incremental Algorithm for a Generalization of the Shortest-Paths Problem. *Journal of Algorithms*, 21:267–305, 1996.
1.58 The Parametric Problem of Shortest Distances. *USSR Computational Math. and Math. Phys.*, 8(5):336-343, 1968.
1.59 H. Rohnert. A Dynamization of the All Pairs Least Cost Path Problem. In *Proc. Symp. Theoretical Aspects of Computer Science* – STACS'85, *Lecture Notes in Computer Science*, vol. 182 (Springer-Verlag, 1985), pp. 279–286.
1.60 K. Simon. An Improved Algorithm for Transitive Closure on Acyclic Graphs. *Theoretical Computer Science*, 58:325–346, 1988.
1.61 D. Sleator and R. Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences*, 24:362–381, 1983.
1.62 R. Tarjan. Efficiency of a Good but not Linear Set Union Algorithm. *Journal of ACM*, 22:215–225, 1975.
1.63 J. Thompson, F. Plewniak, and O. Poch. BAliBASE: A Benchmark Alignments Database for Evaluation of Multiple Sequence Alignment Programs. *Bioinformatics*, 15:87–88, 1999.
1.64 M. Thorup. Decremental Dynamic Connectivity. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms* – SODA'97, pp. 305–313, 1997.
1.65 M. Thorup. Near-Optimal Fully-Dynamic Graph Connectivity. In *Proc. 32nd ACM Symposium on Theory of Computing* – STOC 2000, pp. 343–350, 2000.
1.66 D. M. Yellin. Speeding Up Dynamic Transitive Closure for Bounded Degree Graphs. *Acta Informatica*, 30:369–384, 1993.