Alcom-FT Technical Report Series
ALCOMFT-TR-01-102

COMPUTER TECHNOLOGY INSTITUTE

TECHNICAL REPORT No. TR 2000/08/03

AN EXPERIMENTAL STUDY OF DYNAMIC ALGORITHMS FOR TRANSITIVE CLOSURE

Daniele Frigioni Tobias Miller
Umberto Nanni Christos D. Zaroliagis

31 August 2000

An Experimental Study of Dynamic Algorithms for Transitive Closure*

Daniele Frigioni[†‡] Tobias Miller[§] Umberto Nanni[†] Christos Zaroliagis[¶]

31 August 2000

Abstract

We perform an extensive experimental study of several dynamic algorithms for transitive closure. In particular, we implemented algorithms given by Italiano, Yellin, Cicerone et al., and two recent randomized algorithms by Henzinger and King. We propose a fine-tuned version of Italiano's algorithms as well as a new variant of them, both of which were always faster than any of the other implementations of the dynamic algorithms. We also considered simple-minded algorithms that were easy to implement and likely to be fast in practice. We tested and compared the above implementations on random inputs, on non-random inputs that are worst-case inputs for the dynamic algorithms, and on an input motivated by a real-world graph.

*This work was partially supported by the IST Programme of EU under contract no. IST-1999-14186 (ALCOM-FT) and the Human Potential Programme of EU under contract no. HPRN-CT-1999-00104 (AMORE). The work of the first author was partially supported by the Fellowships Program n. 201.15.13 of the Italian National Research Council (CNR).

†Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza". Via Salaria 113 - 00198 - Roma, Italy. E-mails: {frigioni,nanni}@dis.uniroma1.it.

‡Dipartimento di Ingegneria Elettrica, Università di L'Aquila, Monteluco di Roio, I-67040 L'Aquila, Italy.

§Transport-, Informatik- und Logistik-Consulting GmbH, Frankfurt, Germany.

¶Computer Technology Institute and Department of Computer Engineering & Informatics, University of Patras, 26500 Patras, Greece. E-mail: zaro@ceid.upatras.gr.

1 Introduction

Dynamic graph algorithms maintain a certain property (e.g., connectivity) of a graph that changes dynamically over time. Typical changes include insertion of a new edge and deletion of an existing edge. The challenge for a dynamic algorithm is to maintain, in an environment of dynamic changes, the desired graph property efficiently, i.e., without recomputing everything from scratch after a dynamic change. 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 (but not both) are allowed. In the case of edge insertions (resp. deletions), the partially dynamic algorithm is called *incremental* (resp. *decremental*).

Dynamic graph algorithms have been an active and blossoming research field over the last years, due to their applications in a variety of contexts including operating systems, information systems, network management, and graphical applications. A number of important theoretical results have been obtained for both fully and partially dynamic maintenance of several properties on undirected graphs (see e.g., [12, 13, 14, 15, 22, 31]). Recently, an equally important effort has started on implementing these techniques and showing their practical merits [1, 2]. These were the first implementations concerning fully dynamic maintenance of certain properties (connectivity, minimum spanning tree) in undirected graphs, as well as the first implementation of sparsification, a technique for speeding up dynamic graph algorithms [13].

On the other hand, the development of fully dynamic algorithms for maintenance of various properties on directed graphs (digraphs) turned out to be a much harder problem and much of the research so far was concentrated on the design of partially dynamic algorithms (see e.g., [4, 7, 8, 11, 26, 27, 28, 32]). Only recently, fully dynamic algorithms have started to appear for maintenance of shortest path trees [18, 19, 30] and transitive closure [9, 23, 24, 25]. However, despite the number of interesting theoretical results achieved, very little has been done so far with respect to implementations even for the most fundamental problems (the only implementation effort known to us is concerned with the maintenance of shortest path trees [10, 17]).

In this paper, we are making a step forward in bridging the gap between theoretical results for transitive closure and their implementation by studying the practical properties of several dynamic algorithms for this problem. The main goal of the paper is to investigate in practice the theoretical evidence that dynamic algorithms for transitive closure are more efficient than their static counterparts, and characterize their behavior in a given range of parameters.

The experiments reported in this paper concern incremental algorithms on digraphs, decremental algorithms on directed acyclic graphs (DAGs) and fully dynamic algorithms on digraphs and DAGs. We have implemented the dynamic algorithms for transitive closure proposed in [8, 23, 26, 27, 32], as well as several variants of them. In particular, motivated by the very good experimental behavior of Italiano's algorithms [26, 27], we developed and implemented a new algorithm which is a variant of those algorithms, and whose decremental part applies to any digraph (i.e., not only to DAGs). We also considered some simple-minded algorithms that were easy to implement (in just a few lines of code) and hence likely to be fast in practice, since the involved implementation constants are usually very low.

All the implementations have been written in `C++` using the `LEDA` library for combinatorial and geometric computing [29] and constitute part of the LEDA Extension Package on Dynamic Graph Algorithms [3]. The source codes of our implementations, a demo program, and a graphic experimentation platform are available from `http://www.ceid.upatras.gr/faculty/zaro/software`.

Our experiments have been performed on random inputs (randomly generated graphs and operation sequences), non-random inputs that are worst-case inputs for the dynamic algorithms, and on an input motivated by a real world graph: the graph describing the connections among the autonomous systems of a fragment of the Internet network [6] visible from *RIPE* (`www.ripe.net`), one of the main European servers.

In the case of random inputs, we considered graphs with various values for the number of vertices n, initial number of edges m_0, and length of operation sequence. In the incremental case as well as in the decremental case for DAGs a fine-tuned version of Italiano's algorithms [26, 27] and our new variant of these algorithms 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 ($m_0 \leq n/2$); (ii) the decremental case when the initial graph was sparse ($m_0 \leq n \log n$) and the sequence of operations was of medium to small size. In the decremental case for general digraphs, the simple-minded algorithms were always significantly faster than the decremental algorithm of [23] or our new variant of Italiano's algorithms. A similar behavior was observed in the fully dynamic case for general digraphs; an interesting fact was that the theoretically fastest algorithm [23] was the slowest in practice, even for very small sequences of operations where the algorithm is assumed to perform well. In the fully-dynamic case for DAGs, again the fine-tuned version or our new variant of Italiano's algorithms were the fastest when the initial graph was not sparse ($m_0 > n \log n$); in the sparse case, the simple-minded algorithms became competitive. In the case of non-random inputs, the simple-minded algorithms were significantly faster than any of the dynamic algorithms. Our experiments with the fragment of the Internet graph gave similar conclusions to those obtained from random inputs. On this graph, we also performed experiments with operation sequences for which some knowledge about their update-query pattern is known in advance. Although the theoretical bounds of the dynamic algorithms may not hold in this case, we believe that these experiments give useful suggestions on how to proceed if information about the update-query pattern is provided.

The paper is organized as follows. In Section 2, we give the description of the various algorithms that we have considered along with several implementation details. In Section 3, we provide the results of our comparative experimental study on these algorithms. Finally, in Section 4 we give some concluding remarks and discuss future work. Preliminary portions of this work appeared in [20].

2 Description of the implemented algorithms

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.

We have considered both partially and fully dynamic algorithms for transitive closure. Our starting point was the implementation of three partially dynamic algorithms as well as of several variants of them. These algorithms were Italiano's algorithm [26, 27], Yellin's algorithm [32], and the algorithm of Cicerone et al. [8]. The incremental version of these algorithms applies to any digraph, while the decremental one applies only to DAGs. 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). Italiano's and Yellin's data structures support both `Path` and `Boolean` queries, while the data structure of Cicerone et.al. [8] supports only `Boolean` queries.

Concerning fully dynamic algorithms, we have implemented one of the two randomized algorithms proposed by Henzinger and King [23]. This algorithm is based on a new partially dynamic (decremental) randomized algorithm given in the same paper [23] which we also implemented. The algorithms support both `Boolean` and `Path` queries.

In the remainder of this section, we give a succinct description of the dynamic algorithms considered along with several implementation details.

2.1 Italiano's algorithm and its variants

2.1.1 The main approach

The main idea of the data structure proposed in [26, 27] 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. If G_0 is the initial digraph (before performing a sequence of updates) having n vertices and m_0 edges, the data structure requires $O(n^2)$ space, and is initialized in $O(n^2 + nm_0)$ time.

A `Boolean` query for vertices i and j is carried out in $O(1)$ time, by simply checking the value of $INDEX[i,j]$. A `Path` query for vertices i and j is carried out in $O(\ell)$ time, where ℓ is the number of edges of the reported path, as follows: if $INDEX[i,j] = Null$, then there is no i-j path in G. Otherwise, $INDEX[i,j]$ points to vertex j in $DESC[i]$. By maintaining in each $DESC$ tree parent pointers, a bottom-up traversal from j to i in $DESC[i]$ gives the required path.

The incremental part requires $O(n(m_0 + m))$ time to process a sequence of m edge insertions, while the decremental one requires $O(nm_0)$ time to process any number (m) of edge deletions; hence, if $m = \Omega(m_0)$, then an update operation takes $O(n)$ amortized time.

The insertion of an edge (i,j) is done as follows. First note that the data structure needs to be updated only if there is no i-j path in G. Then, notice that 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 $DESC[u]$ must be updated using the information in $DESC[j]$. This is done as follows: make a copy $DESC1[j]$ of $DESC[j]$; prune $DESC1[j]$ by deleting those vertices that are already in $DESC[u]$; link $DESC1[j]$ to vertex i in $DESC[i]$; and update the u-th row of $INDEX$.

To implement the edge deletion operation, the data structure needs to be augmented since the deletion of an edge creates additional problems. Recall that in this case G is assumed to be acyclic. Let (i,j) be the edge to be deleted. If (i,j) does not belong to any $DESC$ tree, then the data structure does not need to be updated. Otherwise, it 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 edge (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 the edge (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 find quickly valid replacement edges, the data structure is augmented such that with every vertex y, the set of the tails of its incoming edges $IN[y]$ is maintained. Provided that each such set is correctly updated after an edge deletion, checking whether there is a hook for j in $DESC[u]$, after the deletion of (i,j), is accomplished by checking if $IN[j] \cap DESC[u] \neq \emptyset$. It is easy to verify that in such a case, there exists a u-j path in $DESC[u]$ which avoids (i,j). To perform the test $IN[j] \cap DESC[u] \neq \emptyset$ efficiently, i.e., without scanning the whole set $IN[j]$ every time a hook for j is required, a pointer is associated with vertex j in $DESC[u]$ that points to the first unscanned item in $IN[j]$. To implement this, we have introduced an $n \times n$ matrix $HOOK$ whose entry $HOOK[u,j]$ stores the pointer to the first unscanned item in $IN[j]$, 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. We shall refer to the implementation of the above described algorithm as `Ital`.

We have implemented a second version of Italiano's algorithm by removing the recursion in the edge insertion and deletion procedures to see whether (and how) recursion might affect performance. We shall refer to this implementation as `Ital-NR`. Our experiments showed that `Ital` and `Ital-NR` have almost identical performances with `Ital` being quite often slightly faster. Both implementations `Ital` and `Ital-NR` follow closely the description of the algorithms in [26, 27].

We have tried to further optimize Italiano's algorithm by providing a third implementation, named `Ital-Opt`, in which we maintain descendant trees implicitly. More precisely, the $INDEX$ matrix and all $DESC$ trees have been replaced by a single $n \times n$ matrix $PARENT$. The entry $PARENT[i,j]$ stores a pointer to the edge that connects j to its parent in $DESC[i]$, if $j \in DESC[i]$; otherwise, $PARENT[i,j] = Null$. Clearly, `Boolean` and `Path` queries can be easily answered using the $PARENT$ matrix and be carried out within the same resource bounds. We have further incorporated the hook information in the $PARENT$ matrix and thus eliminated the need for the $HOOK$ matrix. The idea is, throughout a sequence of edge deletions, to store in $PARENT[u,j]$ the first unscanned item in $IN[j]$ (the hook for j) with respect to $DESC[u]$.

Although Italiano's algorithm is not designed for fully dynamic problems, we were interested to study its performance in such a case and compare it with fully dynamic algorithms. Of course, the above mentioned bounds for edge insertions and deletions do not hold in a fully dynamic environment on DAGs. The algorithm has to be modified to handle mixed sequences of edge insertions and deletions. The reason is that due to the insertion of an edge e_1 some other edge e_2 which was not previously a hook for its tail vertex, may now become a hook. Hence, 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), we decided to alternatively adopt a lazy approach: delay the resetting of an entry of the $HOOK$ matrix until it is required by the algorithm. We implemented this lazy approach in `Ital-Opt`. Experiments showed a significant improvement upon Italiano's original algorithms (`Ital` and `Ital-NR`) on mixed sequences of updates.

2.1.2 Generalization

We have extended the above ideas and developed a new algorithm whose decremental part applies to any digraph, and not only to DAGs.

This algorithm is based on the fact that if we shrink every strongly connected component of the input digraph $G = (V, E)$ to a single vertex (which we will call *supervertex*), then the resulting graph $G' = (V', E')$ is a DAG. The idea is to use Italiano's algorithm to maintain the transitive closure of G' and additional information regarding the strongly connected components (SCCs) which is crucial for the decremental part of the algorithm. Note that a similar idea was used in [28]; however the data structures and techniques used in that paper can answer only `Boolean` queries. To maintain path information in G', we do not use the original data structures of `Ital-Opt`, since whenever a SCC breaks (due to edge deletions) V' has to be updated and consequently its corresponding supervertex has to be replaced in all descendant trees. To simplify the splitting of SCCs and to avoid having to maintain up to n copies of each supervertex, we had to modify the data structure of `Ital-Opt` for maintaining transitive closure in G'.

We store G' implicitly by maintaining for each supervertex C a list of incoming, $C.In$, and outgoing, $C.Out$, edges. The rest of the data structures used in `Ital-Gen` are:

1. A collection of implicitly represented descendant trees using the $PARENT$ arrays (see below), one for each vertex in V (not in V').

2. A Boolean matrix *INDEX* whose entry $INDEX[i, j]$ is true if there is an i-j path and false otherwise.

3. An array *Scc* of length n, where $Scc[v]$ points to the SCC containing vertex $v \in V$.

4. A set S of SCCs each one represented as a graph. The *Scc* array is used to access the elements of S. For each k-vertex SCC C we maintain:

   (a) A $PARENT$ array of length n such that $C.PARENT[v]$, $v \in V$, contains a pointer to an incoming to the SCC edge that connects it to its parent in $DESC[v]$, if such an edge exists; otherwise the entry is $Null$. (This set of arrays can be considered as a splitting of the $PARENT$ matrix used in `Ital-Opt`.)

   (b) A $HOOK$ array of length n such that $C.HOOK[v]$, $v \in V$, contains a pointer to the first edge from $C.In$ (the incoming to the SCC edges) that has not yet been considered as a hook with respect to v.

   (c) A *sparse certificate* of the SCC consisting of k vertices and $2k - 2$ edges. The sparse certificate is a subgraph of the SCC, has the same set of vertices, and has the property that if there is a path between any two vertices in the SCC, then there is also a path between the same vertices in the sparse certificate.

A sparse certificate of a SCC is computed as follows. Take any vertex r of the SCC and perform a DFS rooted at r. Then, reverse the directions of all edges in the SCC and perform a second DFS rooted at r. Restore the original direction of the edges in the second DFS tree. It can be easily verified that the union of the two resulted trees is the required sparse certificate.

The initialization of `Ital-Gen` requires $O(n^2 + nm_0)$ time and $O(n^2)$ space, and involves computation of the above data structures where the computation of the SCCs and their sparse certificates 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 as 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 $PARENT$ 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 `Ital` and hence has the same time bound: if there is already an i-j path, then nothing is done. Otherwise, the insertion of (i, j) may create new paths from an ancestor u of i to any descendant of j, if of course there was no previous u-j path in G. In such a case, j is inserted as a child of i in the (implicitly represented) descendant tree rooted at u. This is done by setting $HOOK[u] = (i, j)$ in the $HOOK$ array associated with the SCC containing j. This process is performed recursively for all outgoing edges of j.

Deleting an edge (i, j) is done as follows. If (i, j) does not belong to any SCC, then we use Italiano's decremental algorithm to delete (i, j) from G'. Otherwise, we check if (i, j) belongs to the sparse certificate of a SCC or not. In the latter case, we simply remove the edge from the SCC. In the former case, we 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 we compute the new SCCs, update properly the $PARENT$ and $HOOK$ arrays so that the information concerning descendant trees and hooks in the new G' is preserved, and finally we apply Italiano's decremental algorithm to delete (i, j) from the new G'.

Let us discuss in more detail the splitting of a SCC C that breaks into t new SCCs C_i, $1 \leq i \leq t$, as a result of deleting an edge e. We consider e as temporarily removed, since updating of the data structures has to be done before the actual removal of e (performed by Italiano's decremental algorithm). Updating of the data structure consists of four steps: (i) Compute the C_i's and update S as well as the array Scc. (ii) For each C_i, compute its graph representation $C_i.G$, its list $C_i.Out$ of outgoing edges, and its sparse certificate. (iii) Update all descendant trees that contain C. (iv) For each C_i, compute its list $C_i.In$ of incoming edges and the array $HOOK$.

Steps (i) and (ii) can be easily done: call a DFS-based strongly connected components algorithm to compute the C_i's and then update S and Scc. Then, traverse all vertices and edges of G and construct $C_i.G$ and $C_i.Out$. The sparse certificate of C_i can be constructed from $C_i.G$ as explained above. Step (iii) involves the replacement of C in each $DESC[v]$ that contains it with the new supervertices C_i in a way that $DESC[v]$ keeps its tree structure. This is done by firstly identifying the particular SCC C_j that either contains v or contains the tail of the edge $C.PARENT[v]$ that connects C to its parent in $DESC[v]$. The entry $PARENT[v]$ in C_j simply inherits its value from the corresponding entry $C.PARENT[v]$. Then, compute a DFS spanning tree on G' with root C_j and remove all but the nodes (supervertices) representing the (new SCCs) C_i's. The resulting graph is still a tree (since C was strongly connected) and identifies the "connections" of the new SCCs in $DESC[v]$. This tree will replace C in $DESC[v]$. Note that the $PARENT$ pointers regarding the rest of the nodes in $DESC[v]$ do not need to change, since they are references to edges of G. Hence, the tree structure of the new $DESC[v]$ is properly and correctly maintained. Step (iv) has to set $C_i.In$ and $C_i.HOOK$ so that throughout a sequence of edge deletions no edge has to be considered as a hook with respect to a vertex $v \in V$ more than once. An edge is inserted to $C_i.In$ if its tail belongs to C_i and its head does not. These edges were previously either in $C.In$ or in $C.G$. We scan the list $C.In$, from start to end, to find those edges and insert them in $C_i.In$ in the same order. Then, we traverse $C.G$ and insert the relevant edges by appending them to $C_i.In$ (i.e., they are inserted after those from $C.In$). Clearly, these latter edges were never considered as hooks before. The $C_i.HOOK$ array can be updated by remembering which was the last edge in $C.In$ considered as a hook with respect to each vertex $v \in V$. This information can be obtained by using the $C.HOOK[v]$ entries.

Lemma 1 *Any sequence of edge deletions in* `Ital-Gen` *requires* $O(n^2 + m_0^2)$ *time.*

Proof. The application of Italiano's decremental algorithm to handle edge deletions in the DAG G' results in a total of $O(nm_0)$ time for all edge deletions. Let us now count the overhead induced by the computation and splitting of SCCs.

At most $n-1$ SCCs have to be split throughout any sequence of edge deletions. Moreover, there can be at most n SCCs in the final graph. Hence, at most $2n - 1$ SCCs are created by splitting. Assume that a SCC splits into t new SCCs C_i, $1 \leq i \leq t$. Step (i) of splitting can be accomplished within $O(n + m_0)$ time. Steps (ii), (iii), and (iv) can be accomplished in $O(n + m_0)$ time per C_i, i.e., in a total of $O(t(n+m_0))$ time. Therefore, the overall time required for all splittings of SCCs throughout any sequence of edge deletions is $O(n(n+m_0))$.

Finally, we have to count the time required for determining whether a SCC breaks and to recompute its sparse certificate if necessary. This can be done in $O(n+m_0)$ time and can happen at most m_0 times. Thus, checking whether a SCC breaks and recomputing its sparse certificate throughout any sequence of edge deletions requires $O(nm_0+m_0^2)$ time.

Hence, any sequence of edge deletions requires a total of $O(n^2+m_0^2)$ time. ∎

If all edges to be deleted are selected uniformly at random from E, we can show a better bound (provided $n \leq m_0$) based on the fact that it is very unlikely that an edge, chosen to be deleted, belongs to a sparse certificate.

Lemma 2 *Any sequence of edge deletions in* `Ital-Gen` *requires $O(n^2+nm_0)$ expected time, if all edges to be deleted are selected uniformly at random from E.*

Proof. It is clear from the proof of the preceding lemma that the bottleneck in time is checking whether a SCC breaks and recomputing its sparse certificate if necessary. If the edge to be deleted does not belong to the sparse certificate of the SCC, then clearly neither checking nor recomputation is necessary.

Assume that a sequence of m edge deletions is performed and that the edges to be deleted are chosen uniformly at random from E. There are at most $2n-2$ edges in all sparse certificates. Hence, the probability P_i that, after i edge deletions $0 \leq i \leq m-1$, the next edge to be deleted belongs to a sparse certificate is

$$P_i = \begin{cases} \frac{2n-2}{m_0-i}, & \text{if } i < m_0-2n+2 \\ 1, & \text{if } i \geq m_0-2n+2 \end{cases}$$

Note that after m_0-2n+2 edge deletions, the graph contains at most $2n-2$ edges which we may assume that all belong to the sparse certificates. Hence, the expected time for checking whether the SCCs break and recomputing sparse certificates is

$$\begin{aligned}\sum_{i=0}^{m-1} P_i \cdot O(n+m_0-i) &= \sum_{i=0}^{m_0-2n+1} \frac{2n-2}{m_0-i} \cdot O(n+m_0-i) + \sum_{i=m_0-2n+2}^{m-1} 1 \cdot O(n+m_0-i) \\ &= \sum_{i=0}^{m_0-2n+1} \frac{2n-2}{m_0-i} \cdot O(m_0-i) + \sum_{i=m_0-2n+2}^{m-1} O(n) \\ &= \sum_{i=0}^{m_0-2n+1} O(n) + O(n^2) = O(nm_0+n^2)\end{aligned}$$

∎

The preceding discussion along with Lemmata 1 and 2 has established the following.

Theorem 1 *Let $G=(V,E)$ be a digraph with n vertices and m_0 edges. After a sequence of m edge insertions, the incremental part of algorithm* `Ital-Gen` *can maintain the transitive closure of G in $O(n(m_0+m))$ time. After any sequence of edge deletions, the decremental part of algorithm* `Ital-Gen` *can maintain the transitive closure of G in $O(n^2+m_0^2)$ worst-case time, or in $O(n^2+nm_0)$ expected time provided that all edges to be deleted are selected uniformly at random from E. In either case an initialization phase is required which takes $O(n^2+nm_0)$ time and $O(n^2)$ space.*

To use Ital-Gen in a fully dynamic environment, we have made some further modifications and optimizations. Instead of recomputing SCCs, their sparse certificates and G' before any sequence of edge deletions, we merge SCCs to supervertices as soon as they are created. This way, we avoid recomputing the data structure before each sequence of edge deletions (thus speeding up mixed sequences of operations). This further allows us to adapt the lazy approach for resetting the data structure as described in Section 2.1.1.

2.2 Yellin's algorithm

We first describe the Boolean version of Yellin's algorithm (Yellin), and then we show how to extend this data structure in order to handle Path queries. The bounds for update operations given below hold regardless of the type of the query used.

Yellin's data structure associates with every vertex $v \in V$ 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 three fields: *edgeTarget* (pointing to vertex w in *Adjacent*(v), if it exists), *closureSource* (pointing to vertex v in *Reaches*(w), if there is a v-w path in G), and *refcount*. The latter field stores the number of v-w paths in G. More precisely, let $ref(v, w) = \{(v, z, w) : z \in V \wedge (v, z) \in E^* \wedge (z, w) \in E\}$. Then, *refcount*$(v, w) = |ref(v, w)| + 1$, if $(v, w) \in E$, and *refcount*$(v, w) = |ref(v, w)|$, otherwise. The above data structure can be initialized in $O(n^2 + nm_0)$ time and $O(n^2)$ space.

The incremental version of Yellin requires $O(d(m_0+m)^*)$ time to process a sequence of m edge insertions starting from an initial n-vertex, m_0-edge digraph G_0 and resulting in a digraph G; d is the maximum outdegree of G and $(m_0+m)^*$ is the number of edges in G^*. The decremental version requires $O(dm_0^*)$ time to process any sequence of m edge deletions; d is the maximum outdegree of G_0. A Boolean query for vertices i and j takes $O(1)$ time, since it involves only checking the value *INDEX*$[i, j]$.*refcount*.

The main idea for updating the data structure after the insertion of the 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 finds first 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* increased from 0 to 1. Now, (x, y, z) is a new triple for $ref(x, z)$ and *refcount*(x, z) is incremented.

The edge deletion algorithm is the "dual" of the edge insertion algorithm described above. After the deletion of an edge (a, b), we have to find, for all $x, z \in V$, the triples (x, y, z) that should not belong anymore to $ref(x, z)$ and to use them to update *INDEX*$[v, w]$.*refcount*. The deletion algorithm finds first all vertices x such that (x, a) was an edge of G^*_{old} (the transitive closure graph before the deletion of (a, b)). In this case, (x, a, b) should be removed from $ref(x, b)$, and *refcount*(x, b) has to be decremented. If its *refcount* is now 0, then (a, b) is no longer a transitive closure edge. Then, the deletion algorithm considers each edge (x, y) that is not in G^*_{new} (the transitive closure graph after the deletion of (a, b)) and each edge (y, z) of G. Now the triple (x, y, z) is no longer a triple for $ref(x, z)$ and *refcount*(x, z) is decremented. If it becomes 0, then (x, z) is no longer a transitive closure edge.

The data structure described so far cannot return an arbitrary x-y path in G, if such a path exists. To support the Path query, the data structure is augmented with the so called *support graph*. The support graph is a digraph consisting of two kinds of vertices: closure vertices, that have a label (x, z) corresponding to an edge in G^*, and join vertices, that have a label (x, y, z) corresponding to the triple (x, y, z) in $ref(x, z)$. The array *INDEX* is also augmented with a fourth field, $INDEX[x, z].SGnode$, which points to the closure vertex labeled (x, z). It is not hard to see that having the support graph `Path` queries can be easily answered. The support graph can be updated after an edge insertion or edge deletion within the same resource bounds. It is worth mentioning that the actual description in [32] of updating the support graph after an edge deletion is erroneous. However, we have managed to correct the error and provided a correct implementation. The `Path` version of `Yellin`, denoted as `Yellin-SG`, requires $O(n^2 + dm_0^*)$ time and space to be initialized, where d is the maximum outdegree of G_0.

Our implementations `Yellin` and `Yellin-SG` follow closely the description of the algorithms in [32] which do not give much space for optimizations. It is also easy to see that the data structures in both versions need no modification in order to use them in a fully dynamic environment.

2.3 The algorithm of Cicerone et al.

The algorithm in [8] 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.

Let $R \subseteq V \times V$ be a binary relationship (e.g., transitive closure) on the vertices of a digraph $G = (V, E)$. The approach in [8] allows to define a *propagation property* based on R that describes how R "propagates" along the edges of G. More formally, 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 relation 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 $R_0 = \{(x, x) : x \in V\}$.

The main idea is to maintain 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 further 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 $\langle k, y\rangle$. If G_0 is the initial digraph having n vertices and m_0 edges, the data structure requires $O(n^2)$ space, and is initialized in $O(n^2 + nm_0)$ time.

The incremental part of the algorithm in [8] requires $O(n(m_0 + m))$ to process a sequence of m edge insertions, while the decremental one requires $O(nm_0)$ time to process any number (m) of edge deletions; hence, if $m = \Omega(m_0)$, then an update operation takes $O(n)$ amortized time. 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: first of all, 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 incremented 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 behavior 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 decremented. An edge deletion is handled as follows: first of all, 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 decremented 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, there is no edge useful to pair (k,h) left. 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 $C_R[k,h]>0$, then (k,h) is still in R after deleting edge (i,j) only because G is acyclic. In fact, let us suppose $h=j$, $(j,x_1,x_2,\ldots,x_p,j)$ is a cycle with $i \neq x_l$ for each $l=1,2,\ldots,p$, and $C_R[k,j]=2$ because of edges (i,j) and (x_p,j). In this situation, after the deletion of (i,j), the edge (x_p,j) is no longer useful to pair $\langle k,j\rangle$, whereas it is still considered by counter $C_R[k,j]$.

In the case where the binary relationship R is the transitive closure, the algorithms proposed in [8] collapse to the solution provided by La Poutré and van Leeuwen in [28].

We implemented two different variants of the algorithms in [8]. The general technique described above, denoted as `CFNP`, and its specialization to the transitive closure problem, denoted as `CFNP-Opt`. The main difference between the two implementations is the following: 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 path between i and j and no such path existed before (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.

Our implementations follow closely the description of the algorithms given above. We tried several optimizations (e.g., regarding the implementation of the queues), but we observed almost no difference in performance.

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.

2.4 The Henzinger-King algorithms

The algorithms in [23] 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).

2.4.1 The decremental algorithm

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 $i=1,\ldots,\log(n\log^2 n)$, where $|S_i|=\min\{O(2^i \log n, n)\}$. Let $S=\bigcup_i S_i$. Note that $|S|=O(n/\log n)$. Algorithm `HK-1` uses the following data structures.

1. For every $x \in S_i$ the algorithm maintains:

   (a) $out(x,n/2^i)$ and $in(x,n/2^i)$.

(b) $Out(x) = \bigcup_{i:x \in S_i} out(x, n/2^i)$ and $In(x) = \bigcup_{i:x \in S_i} in(x, n/2^i)$.

2. For each $u \in V$ the sets $out(u, \log^2 n)$ and $in(u, \log^2 n)$ are maintained.

The $out(x, k)$ and $in(x, k)$ sets are maintained in a decremental environment using a (modification of a) technique proposed by Even and Shiloach [16] 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. The idea is to maintain for each vertex j that belongs to such a structure T a set $Up(j)$ that stores the incoming edges of j whose tails belong to the level above j in T, i.e., to level $level(j) - 1$, where $level(j)$ is the length of the BFS path from x to j. If an edge (i, j) is deleted from T, the algorithm removes it from $Up(j)$. If $Up(j)$ is not empty, then another edge is selected which replaces (i, j) and hence T is correctly updated. If $Up(j)$ is empty, then there is no x-j path of length $level(j)$. However, there may be an x-j path of length greater than $level(j)$. Consequently, the algorithm increments $level(j)$ by one and computes the new $Up(j)$. If $Up(j)$ is still empty, continue as above. Since $level(j)$ has been increased, the level of some descendants of j may also have to be increased and as a consequence their $Up(\ )$ sets have to be updated. This is done in a recursive manner starting with the outgoing edges of j. During the course of the algorithm each edge is processed as many times as its tail drops a level. Since this can happen at most k times, a BFS structure of depth k can be maintained in $O(km_0)$ time after any sequence of edge deletions.

The above idea is used to update the data structure of `HK-1` after an edge deletion. Hence, any sequence of edge deletions requires $O(m_0 n \log^2 n)$ time.

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 $x \in S$ if $u \in In(x)$ 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. A `Path` query is answered in an additional $O(\ell)$ time, where ℓ is the length of the reported path.

In our implementation we have done the following optimizations. Firstly, we tried to reduce the amount of information required to be stored. Observe that if in $Out(x)$ (or in $In(x)$) there are more than one trees having as root the same vertex, we need only to keep the tree with the largest depth. Hence, it is sufficient to maintain with each distinguished vertex $x \in S$, two sets $out(x, d)$ and $in(x, d)$, where $d = \max_{i:x \in S_i}\{n/2^i\}$. Secondly, we wanted in every $out(x, k)$ (resp. $in(x, k)$) set to check whether there is a x-u (resp. u-x) path in $O(1)$ time. We implemented this by assigning to the vertices not in such a set a level greater than k.

Two BFS structures are stored with each distinguished vertex and hence at most four of them can be stored with each vertex of the graph. Since a BFS structure can be initialized in $O(n + m_0)$ time and space, the data structures of `HK-1` are initialized in $O(n^2 + nm_0)$ time and space.

2.4.2 The fully dynamic algorithm

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. Let m_0 be either the initial number of edges or the number of edges in G at the time of the last rebuild. The total time for a sequence of no more that $\sqrt{n}$ insertions and $\sqrt{n}$

deletions is $O(m_0 n \log^2 n + n\sqrt{n})$. Let $\hat{m}$ be the average number of edges in G during the sequence of updates. Since $m_0 \leq \hat{m} + \sqrt{n}$, we have an $O(\hat{m}\sqrt{n}\log^2 n + n)$ amortized bound per update. The algorithm is initialized in $O(n^2 + nm_0)$ time and space.

To answer a query for vertices 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). Clearly, the query bounds are the same as those of `HK-1`.

2.5 The simple-minded algorithms

Except for the above mentioned dynamic algorithms, we also considered some simple-minded algorithms which were easy to implement in a few lines of code. This usually implies that the implementation constants are very low and hence these algorithms are likely to be fast in practice. The simple-minded algorithms that we implemented are 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), we have made three different implementations that 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. Our implementation of the simple-minded algorithms include: `DFS`, `BFS`, and `DBFS` which is a combination of `DFS` and `BFS` that works as follows. Vertices are visited in DFS order. Every time a vertex is visited we first check whether any of its adjacent vertices is the target vertex. If yes, then we stop; otherwise, we continue the visit in a DFS manner. As we shall see, there were cases where `DBFS` outperformed `DFS` and `BFS`.

2.6 Summary

For ease of reference, we give in Figure 1 the theoretical time and space bounds of all algorithms described in this section and used in our experimental study. The `Simple` algorithms are `DFS`, `BFS`, and `DBFS`. The initialization, space and `Boolean` query bounds are worst case. Concerning the `Simple` algorithms, the time bounds of insertions and deletions are worst case over a single operation. The time bounds of insertions and deletions for the remaining algorithms (except `HK-2`) are worst case over sequences of $\Omega(m)$ updates. `Path` queries can be answered in $O(\ell)$ additional time with respect to `Boolean` queries, where ℓ is the number of edges of the reported path (recall that `CFNP`, `CFNP-Opt` and `Yellin` do not support `Path` queries).

3 Experimental results

All of our implementations were written as `C++` classes using several advanced data types of `LEDA` [29] and constitute part of the LEDA Extension Package (LEP) on Dynamic Graph Algorithms [3]. Each class is derived from a new, especially designed, base class defining a common interface for all classes. Every class is installed in an interface program that allows to read graphs and sequences from files, to perform operations, to store the results again on files and to show them on graphics. As it is customary with all algorithms in the LEP on Dynamic Graph Algorithms, we also use in our implementations a new class of dynamic graphs, called *message graph* [3], which is derived from LEDA's data type `graph`. Its main feature is that different dynamic algorithms, maintaining different properties, can operate on the same graph.

Algorithm	Space	Initialization	`Boolean`	Insertions (digraphs)	Deletions (DAGs)	Deletions (digraphs)
`Simple`	$O(n+m_0)$	$O(1)$	$O(n+m_0 \pm m)$	$O(1)$	$O(1)$	$O(1)$
`Ital`	$O(n^2)$	$O(n^2+nm_0)$	$O(1)$	$O(n(m_0+m))$	$O(nm_0)$	–
`Ital-NR`	$O(n^2)$	$O(n^2+nm_0)$	$O(1)$	$O(n(m_0+m))$	$O(nm_0)$	–
`Ital-Gen`	$O(n^2)$	$O(n^2+nm_0)$	$O(1)$	$O(n(m_0+m))$	$O(nm_0)$	$O(n^2+m_0^2)$ $\tilde{O}(n^2+nm_0)$
`Ital-Opt`	$O(n^2)$	$O(n^2+nm_0)$	$O(1)$	$O(n(m_0+m))$	$O(nm_0)$	–
`Yellin`	$O(n^2+dm_0^*)$	$O(n^2+dm_0^*)$	$O(1)$	$O(d(m_0+m)^*)$	$O(dm_0^*)$	–
`Yellin-SG`	$O(n^2+dm_0^*)$	$O(n^2+dm_0^*)$	$O(1)$	$O(d(m_0+m)^*)$	$O(dm_0^*)$	–
`CFNP`	$O(n^2)$	$O(n^2+nm_0)$	$O(1)$	$O(n(m_0+m))$	$O(nm_0)$	–
`CFNP-Opt`	$O(n^2)$	$O(n^2+nm_0)$	$O(1)$	$O(n(m_0+m))$	$O(nm_0)$	–
`HK-1`	$O(n^2+nm_0)$	$O(n^2+nm_0)$	$O(\frac{n}{\log n})$	–	$\tilde{O}(nm_0 \log^2 n)$	
`HK-2`	$O(n^2+nm_0)$	$O(n^2+nm_0)$	$O(\frac{n}{\log n})$	$\tilde{O}(m_0 n \log^2 n + n\sqrt{n})$ total time of $\sqrt{n}$ updates		

Notation

m_0	initial number of edges (i.e., before a sequence of operations)
m	number of updates in a sequence of operations
a^*	number of edges in the transitive closure of a graph with a edges
d	maximum outdegree
$\tilde{O}$	expected time

Figure 1: Theoretical bounds of the implemented algorithms.

We have augmented our implementations with additional procedures that helped us to easily verify correctness. For example, a `Path`(u, v) operation has been implemented either to return the u-v path (if it exists), or to exhibit a cut in the graph separating the vertices reachable by u from the rest of the graph. In the former case, the correctness check is trivial. In the latter case, the heads of all edges in the cut should belong to the part containing u.

Except for the above "visual" check, we have also developed a very simple correctness checking program (in the spirit of [29]). The program takes the graph at the end of an operation sequence and does the following: for each vertex pair of the graph it issues a reachability query using a dynamic algorithm and checks whether the answer obtained is identical to the answer given by issuing the same query using a correct LEDA program (e.g., the static DFS algorithm). Correct implementation implies that no mismatches should be found (a fact that was indeed verified by our experiments).

All of our experiments were run on two machines with sufficiently large main memory since all algorithms require at least quadratic space. The first machine was a SUN Ultra-Sparc with two 200 MHz processors and 1GB of main memory and the other machine was a SUN Sparc-20 with two 64 MHz processors and 256 MB of main memory. Even with the former machine, we had problems to deal with dense instances of big graphs. We used the GNU `g++` compiler (version 2.8.1) with `-O` optimization and disabled LEDA's low-level consistency checks by specifying `-DLEDA_CHECKING_OFF`. The given time bounds are average CPU times in seconds over the results of 5 to 10 different tests. The CPU times were measured using the `LEDA` function `used_time()` (we also measured times with the `Unix` function `getrusage`, but observed no substantial difference with the times reported by `used_time()`).

We performed a comparative experimental study with all our implementations using different types and sizes of inputs. Since there are no former experimental studies for dynamic transitive closure, there are no libraries or benchmark inputs. Moreover, in the dynamic setting it is less obvious what constitutes a "typical" or "interesting" input and, perhaps due to the lack of available implementations, there are not yet many applications of dynamic transitive closure that could provide "real-world" inputs.

We performed our experiments on three kinds of inputs:

(i) Random inputs, i.e., random sequences of operations (edge insertions intermixed with queries, or edge deletions intermixed with queries, or fully-dynamic sequences) performed on random digraphs and random DAGs. This kind of experiments identifies the average-case performance of the algorithms.

(ii) Non-random inputs that are worst-case inputs for the dynamic algorithms. These inputs try to enforce bad update patterns on the algorithms.

(iii) An input motivated by a real world graph: the graph describing the connections and policy strategies among the autonomous systems of a fragment of the Internet network [6] visible from *RIPE* (`www.ripe.net`), one of the main European servers. On this graph, we performed random sequences of operations, since no "real" sequences were provided.

We initially compared our implementations of dynamic algorithms with a static algorithm for transitive closure, since we were firstly interested to verify that using the dynamic algorithms is better than recomputing transitive closure from scratch. To this end, we used the static `TRANSITIVE_CLOSURE` algorithm provided by LEDA whose implementation is based on the algorithm given in [21] and requires $O(nm)$ time. This algorithm was called only in the case where a `Boolean` query was issued that is preceded by a dynamic change (edge insertion/deletion) on G; i.e., no recomputation was performed after an edge insertion/deletion, or between consecutive query operations. Despite this fact, it was incredibly slower than any of the dynamic algorithms or the simple-minded approaches described in Section 2 and for this reason we do not report on experimental results regarding comparison with `TRANSITIVE_CLOSURE`.

Another interesting observation was that the performances of `Ital` and `Ital-NR` were almost always identical in all experiments we conducted (on any kind of input), with `Ital` being quite often slightly faster, and hence we report experiments only on `Ital`. The performance of `Yellin-SG` was so slow and space consuming in our experiments, due to the maintenance of the support graph (occupying $O(n^3)$ space), that we do not report results regarding comparison with this algorithm. Among the implementations of the simple-minded algorithms (`BFS`, `DFS` and `DBFS`), we usually select the fastest and report comparisons only with it. Finally, we did not notice any substantial difference in the behavior of the algorithms when executed on sequences of updates with `Path` queries or sequences of updates with `Boolean` queries. For this reason we report experiments only on sequences of updates with `Boolean` queries.

3.1 Random inputs

For this class of experiments we performed our tests on random digraphs with different densities and on random sequences of operations. We generated a large collection of data sets, each consisting of 5 samples. A data set corresponded to a fixed value of graph parameters. Each sample consisted of a random digraph initialized to contain n vertices and m_0 edges, and a random sequence σ of update operations (insertions/deletions) intermixed with `Boolean` queries. Each query and update were uniformly mixed, i.e., occurred with probability 1/2 (we considered – as it is also assumed in [8, 23, 26, 27, 32] – an on-line environment with no prediction of the future, and where queries and updates are equally likely). We ran our algorithms on each sample and retained the average CPU time in seconds over the 5 samples for each program.

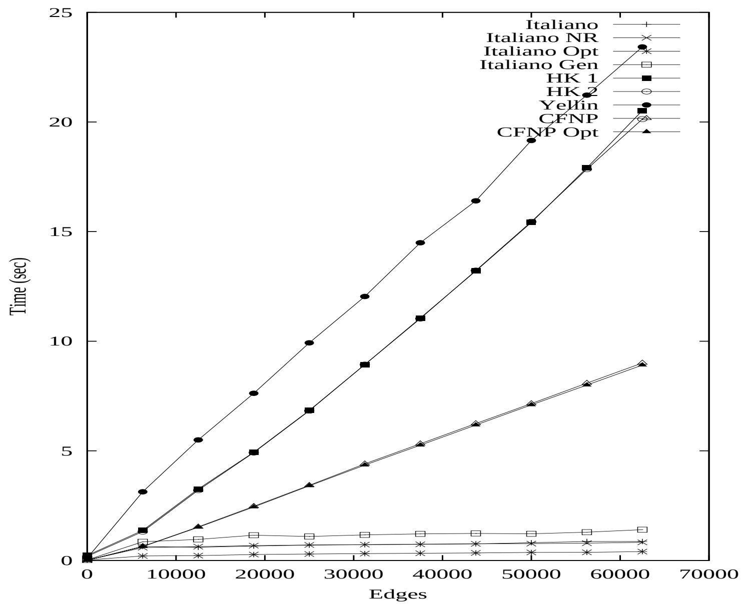

Figure 2: Initialization times for all dynamic algorithms on digraphs with 500 vertices.

We performed a series of tests on graphs with $n = 100, 300, 500, 700$ vertices and m_0 close to values in the set $\{0, n/2, n, n \ln n, n^{1.5}, n^2/\ln n, n^2/4\}$ (not all values apply to each case; e.g., it does not make sense to consider a decremental environment with $m_0 = 0$). For these values of n and m_0, we considered various lengths $|\sigma|$ of operation sequences: very small ($|\sigma| = 10, 20$), small ($|\sigma| = 50$), medium ($|\sigma| = 500$), large ($|\sigma| = 5000$), and very large ($|\sigma| = 50000$). We conducted also experiments with other values of n and $|\sigma|$ and reported very similar results. The choice for the above values of m_0 was motivated by the fact that we are dealing with random graphs and random graph theory provides many interesting results regarding their structural properties [5]. If $m \approx n \ln n$, then with high probability a random (di)graph is (strongly) connected. With m below $n \ln n$, the graph is disconnected (with high probability) and exhibits an interesting behavior as m approaches n. While m stays even slightly more than n, the random graph has a giant component of size $\Theta(n)$ and several small components the largest of which has size $O(\ln n)$. If $m \approx n$, then the giant component has size $\Theta(n^{2/3})$. Finally, if m drops slightly below n, then the largest component has size $O(\ln n)$. Hence, the values $n/2, n$ and $n \ln n$ exhibit points where a fundamentally different structural behavior of the graph occurs. The larger values of m_0 considered ($n^{1.5}, n^2/\ln n$) are chosen mostly as intermediate steps towards denser graphs ($m_0 = n^2/4$).

Figure 2 illustrates the time required for initializing the data structures in the case $n = 500$ (experiments were run on the Sparc-20). This gives an indication of the particular overhead introduced by each data structure (hidden by the asymptotic notation). It is interesting to observe that Italiano's data structure and its variants have the fastest initialization times which seem to be independent of m_0. The initialization times of the rest of the data structures grow almost linearly with m_0. This behavior of Italiano's data structures can be explained by the fact that they are initialized by repeated calls of the edge insertion procedure[1]. Hence, the majority of the work is performed until the graph reaches its connectivity threshold after which very little happens.

[1]We have actually investigated alternative approaches for initialization in an effort to avoid the repeated calls of

3.1.1 Edge insertions

We start with the case of general digraphs and later report on the experiments performed on DAGs. In all the experiments with edge insertions on digraphs the behavior of the algorithms was more or less the expected one. In particular, the dynamic algorithms performed the majority of their work until the graph reaches its connectivity threshold above which all algorithms have a stable behavior. This behavior is exhibited regardless of the initial graph density and the length of the operation sequence. This can be explained by the fact that as the graph becomes denser, an edge insertion does not add many information w.r.t. transitive closure. Figure 3 (left column) illustrates the relative performances of the algorithms for large and small sequences of operations in digraphs with 500 vertices.

Almost always the fastest algorithms were `Ital-Opt` and `CFNP-Opt`, with no significant difference in their performances. The other versions of Italiano's algorithm, `Ital` and `Ital-Gen`, were quite close to them, with `Ital-Gen` being a bit slower for large and medium sequences of operations. This is probably due to the merging of SCCs in `Ital-Gen` as new edges are added to the graph. The slowest algorithms were always `Yellin` and `CFNP` (they were slower than any of the simple-minded approaches). A somewhat interesting observation was that quite earlier than the connectivity threshold (i.e., as soon as $m_0 > n/2$), the simple-minded algorithms became slower than Italiano's algorithms and `CFNP-Opt` (they were from 2 to 9 times slower). This could be explained by the fact that, for each query, `CFNP-Opt` and the Italiano's algorithms perform only a table lookup, while the simple-minded algorithms need $\Omega(n)$ time. For $m_0 \leq n/2$, `DBFS` (the fastest simple-minded algorithm) becomes competitive with `Ital-Opt`.

Even if all the above implemented algorithms work for any digraph, we also performed experiments on random DAGs to check whether the behavior of the algorithms was indeed similar. The dynamic algorithms exhibited an almost identical behavior with that observed for the case of general digraphs. The behavior of the simple-minded algorithms, however, was different. The running times of `DFS`, `DBFS` and `BFS` grow almost linearly with the edge density of the initial graph. This might be explained by the fact that the probability to answer "true" to a reachability query in a digraph is larger than that of issuing the same query to a DAG: if in a DAG there exists a path from a vertex x to a vertex y, then there is no path from y to x. Since queries are generated uniformly at random from all vertex pairs, the simple-minded algorithms, when applied to DAGs, will more frequently fail to find the requested path and hence will more often exhaustively search the graph. This causes the simple-minded algorithms to become even slower than `CFNP` and `Yellin` after a certain point (usually after the connectivity threshold). Figure 3 (right column) illustrates the relative performances of the algorithms for large and small sequences of operations in DAGs with 500 vertices. A more detailed set of experimental results for medium size sequences is given in Figures 10 and 11 in the appendix.

We also performed experiments with `HK-2` both on general digraphs and on DAGs. Its performance was extremely slow, e.g., it was at least 10 times slower than `DBFS` even for very small operation sequences, and hence we don't report comparisons with this algorithm. We believe that this behavior is due to the (apparently) costly rebuilding of the data structure after $\sqrt{n}$ updates as it is evident from Figure 2 (see also Sections 3.1.3 and 3.2).

---

the edge insertion procedure and hopefully speedup the initialization time of Italiano's algorithms. For example, we ran a DFS algorithm from each vertex of the graph to create the descendant trees and consequently build the *INDEX* or *PARENT* matrices. Experiments showed, however, that the alternative approaches were always slower than the original one.

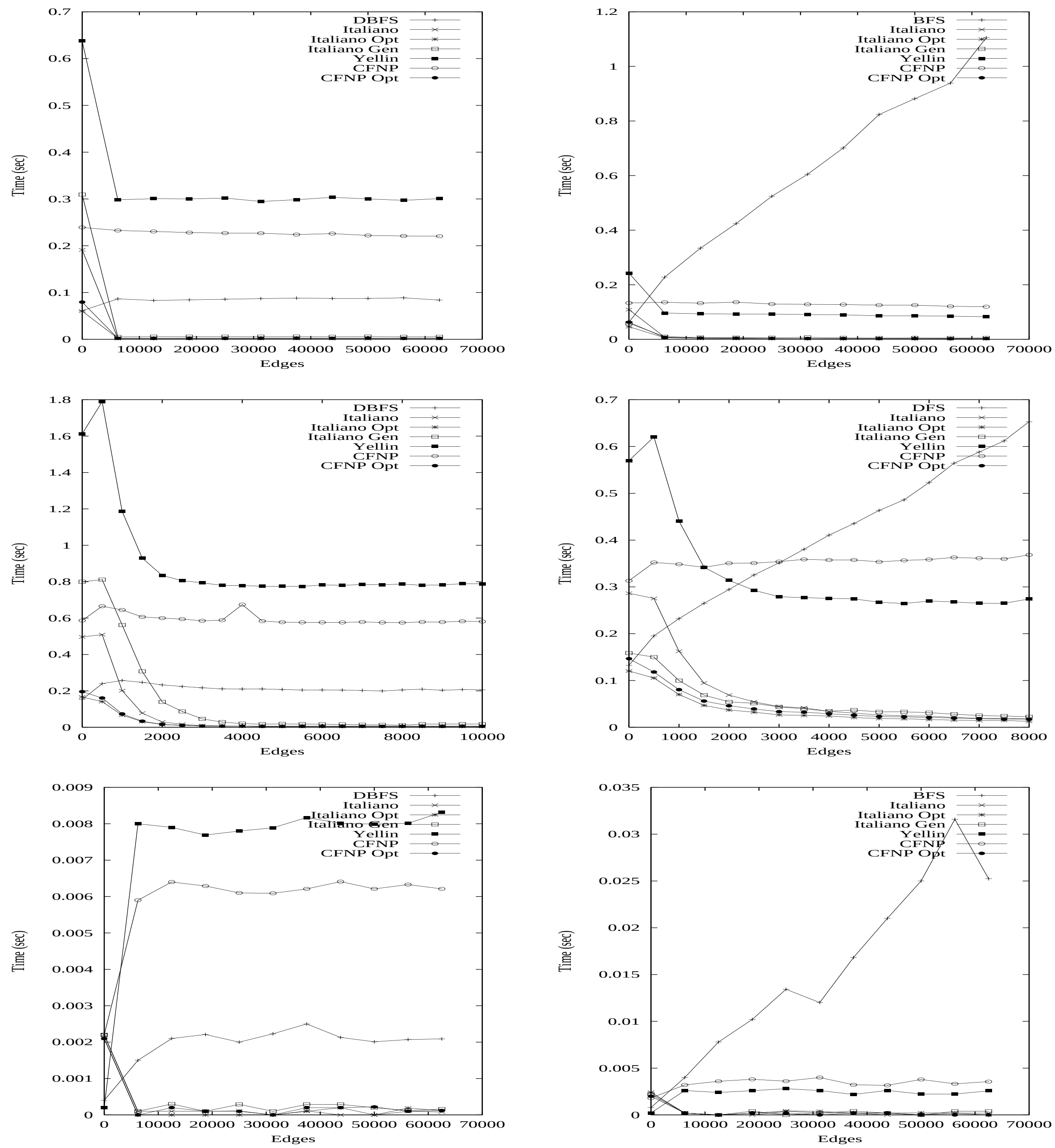

Figure 3: Edge insertions and queries on digraphs (left column) and DAGs (right column) with 500 vertices. The top and middle rows show results for large sequences (5000 operations), while the bottom row shows results for small sequences (50 operations). The experiments on the top row were run on the Ultra-Sparc, while the rest on the Sparc-20.

3.1.2 Edge deletions

We start with the case of DAGs, since most decremental algorithms were designed for this case. Later we will consider the case of general digraphs.

As expected from the theory, the dynamic algorithms should perform well for sufficiently long operation sequences as in this case they achieve the best amortized bound per edge deletion. In our experiments with very large and large sequences, almost all dynamic algorithms were faster than any of the simple-minded ones regardless of the initial graph density; the fastest algorithm was `Ital-Gen`. There were two exceptions: (a) `HK-1` which becomes slower than the simple-minded algorithms when $m_0 \leq n^{1.5}$; (b) in the case of large sequences and when $m_0 \leq n \ln n$, the simple-minded algorithms become competitive with the fastest four dynamic algorithms `Ital-Gen`, `Ital-Opt`, `CFNP`, and `Yellin`. It is worth noting that our optimized versions of Italiano's algorithms (`Ital-Gen` and `Ital-Opt`) were from 2.5 to 5 times faster than the original algorithm (`Ital`).

In the case of medium, small and very small sequences, the behavior of the algorithms is the same with that of large and very large sequences when $m_0 > n \ln n$. If $m_0 \leq n \ln n$, then the simple-minded algorithms become faster than any of the dynamic ones. The best simple-minded (`DFS`) was at most 1.5 times faster than the best dynamic algorithm (`Ital-Gen`). This behavior can be explained from the fact that the dynamic algorithms perform much more work below the connectivity threshold (e.g., Italiano's algorithms have greater difficulty in this case to find the hook) which cannot be amortized with the length of the sequence. The slowest dynamic algorithm was almost always `HK-1` (at least 3 times slower than `DFS`).

The behavior of `HK-1` is not perhaps surprising, since it has the worst theoretical bound for processing a sequence of edge deletions and also answers each query in $O(n/\log n)$ time. The surprise for us was that `Ital-Gen` was faster than `Ital-Opt`. While we cannot explain this theoretically, we suspect that this is due to a bigger cache miss rate. In `Ital-Gen` descendant and hook information is checked on local arrays; in `Ital-Opt` this is checked on a global matrix. Figure 4 (right column) illustrates the relative performances of the algorithms for large and small sequences of operations in DAGs with 500 vertices. A more detailed set of experimental results for medium size sequences is given in Figure 12 in the appendix.

We now turn to the case of sequences of edge deletions and queries on general digraphs. We have conducted experiments with the simple-minded algorithms, `HK-1` and `Ital-Gen`. Our experiments have shown that the simple-minded algorithms were always extremely faster than `HK-1` and `Ital-Gen`. The best simple-minded (`DBFS`) was about 6 (resp. 12) times faster than `HK-1` (resp. `Ital-Gen`) when $m_0 > n \ln n$, i.e., in the range where the dynamic algorithms are assumed to have a good performance. `Ital-Gen` is quite slow in this range since it has the worst theoretical bound caused by the splitting of SCCs and the recomputation of their sparse certificates. However, `Ital-Gen` becomes faster than `HK-1` when $m_0 \leq n \ln n$. These results hold regardless of the value of n and the length of the sequence. An interesting observation about `Ital-Gen` is that its performance remains stable regardless of the value of m_0 and the length of the operation sequence. In contrast, `HK-1` is stable only above the connectivity threshold. Figure 4 (left column) illustrates the performances of the algorithms for large and small sequences of operations in general digraphs with 500 vertices. A more detailed set of experimental results for medium size sequences is given in Figure 13 in the appendix.

Finally, we can perhaps draw another conclusion about `HK-1`. As mentioned above, the query time introduces a certain overhead in processing any sequence of operations. The rest of the overhead comes from the updating of the BFS structures. By observing the behavior of the curve of `HK-1` on both general digraphs and DAGs, it seems that the algorithm performs quite a lot of

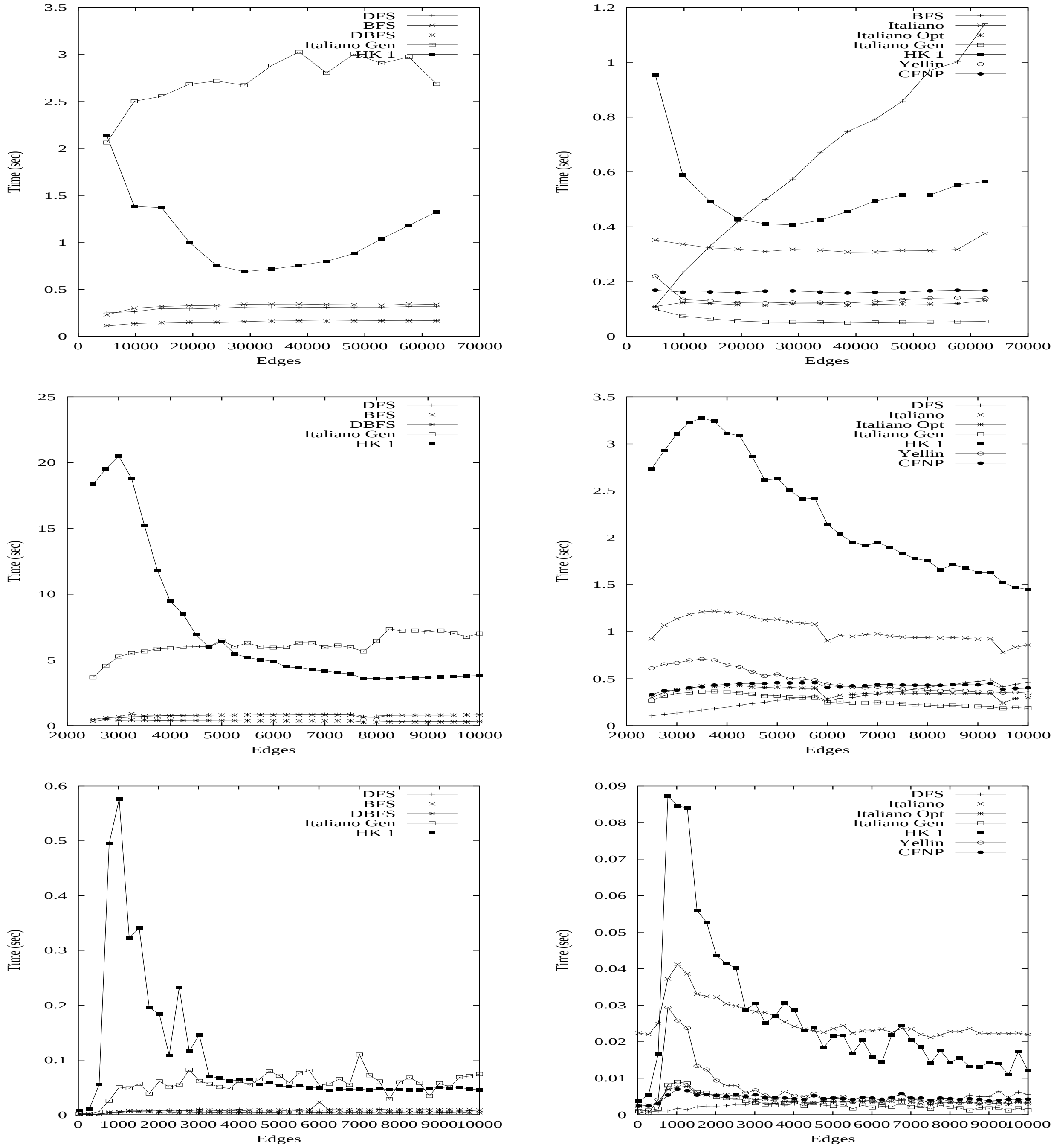

Figure 4: Edge deletions and queries on digraphs (left column) and DAGs (right column) with 500 vertices. The top and middle rows show results for large sequences (5000 operations), while the bottom row shows results for small sequences (50 operations). The experiments on the top row were run on the Ultra-Sparc, while the rest on the Sparc-20.

work below the connectivity threshold for updating the BFS structures. This is both due to their size ($O(nm_0)$) and due to the fact that as the graph becomes sparser it is difficult to update quickly the $Up(\ )$ sets.

3.1.3 Edge insertions and deletions

In the case of a fully dynamic environment, we decided to make experiments not only with the fully dynamic algorithm we had implemented (`HK-2`) and the simple-minded ones, but also with hybridizations of the partially dynamic algorithms. We were interested to study their practical behavior in such an environment, although we knew that the theoretical bounds do not hold in this case. As mentioned in Section 2, the use of `CFNP` and `Yellin` in a fully dynamic setting on DAGs was straightforward. Using Italiano's algorithm, however, required some more work (cf. Section 2.1) regarding the resetting of hook entries before any sequence of edge insertions that is preceded by a sequence of edge deletions. This affects considerably the amortized bounds per operation for `Ital` and `Ital-NR`, since resetting all hooks takes $O(n^2)$ time. In `Ital-Opt` and `Ital-Gen` we followed a lazy approach, i.e., we reset a hook entry only when it was required by the algorithm.

We start with the experiments performed on general digraphs. We experimented with the simple-minded algorithms, `HK-2` and `Ital-Gen`. The rebuilding of data structure in `HK-2` after $\sqrt{n}$ updates seems to be a big bottleneck in performance and a source of memory consumption. In the graphs considered in our study, we were able to run `HK-2` up to termination only for small and very small sequences of operations. For medium sequences we were able to perform experiments only for sparse instances of our graphs. Large and very large sequences were almost impossible to run. From the theoretical analysis, we expected to see a good behavior of `HK-2` for small sequences of updates, but this fact was not confirmed by our experiments (we only observed that for very small sequences of updates the gap between `HK-2` and `Ital-Gen` decreases). In general, the performances of `HK-2` and `Ital-Gen` were extremely slower than that of simple-minded algorithms regardless of the length of the sequences and of the size of the initial graph: `HK-2` was at least 10 times slower than `Ital-Gen` which in turn was about 6 times slower than `DBFS` (the fastest simple-minded). Figure 5 (left column) illustrates the performances of the algorithms for large and small sequences of operations in general digraphs with 500 vertices. A more detailed set of experimental results for medium size sequences is given in Figure 14 in the appendix.

The experiments on DAGs were more promising regarding the hybridized dynamic algorithms (experiments with `HK-2` were similar to the case of general digraphs). As in the case of edge insertions and deletions on DAGs, the running time of the simple-minded algorithms increases with the density of the initial graph. The behavior of all dynamic algorithms was stable after a certain point (usually the connectivity threshold) with `Ital-Gen` being the fastest. All dynamic algorithms, except `Ital`, were faster than the simple-minded ones when $m_0 > n \ln n$; `Ital-Gen` was from 2 to 10 times faster than `BFS` (best simple-minded), which in turn was about 6 times faster than `Ital`. This also demonstrates that the lazy approach in resetting the hooks was indeed successful. When $m_0 \leq n \ln n$, the simple-minded algorithms became competitive with `Ital-Gen`. Finally, we would like to notice that `Ital-Gen` and `Ital-Opt` were always faster than `Yellin` and `CNFP`. Figure 5 (right column) illustrates the performances of the algorithms for large and small sequences of operations in DAGs with 500 vertices. A more detailed set of experimental results for medium size sequences is given in Figure 15 in the appendix.

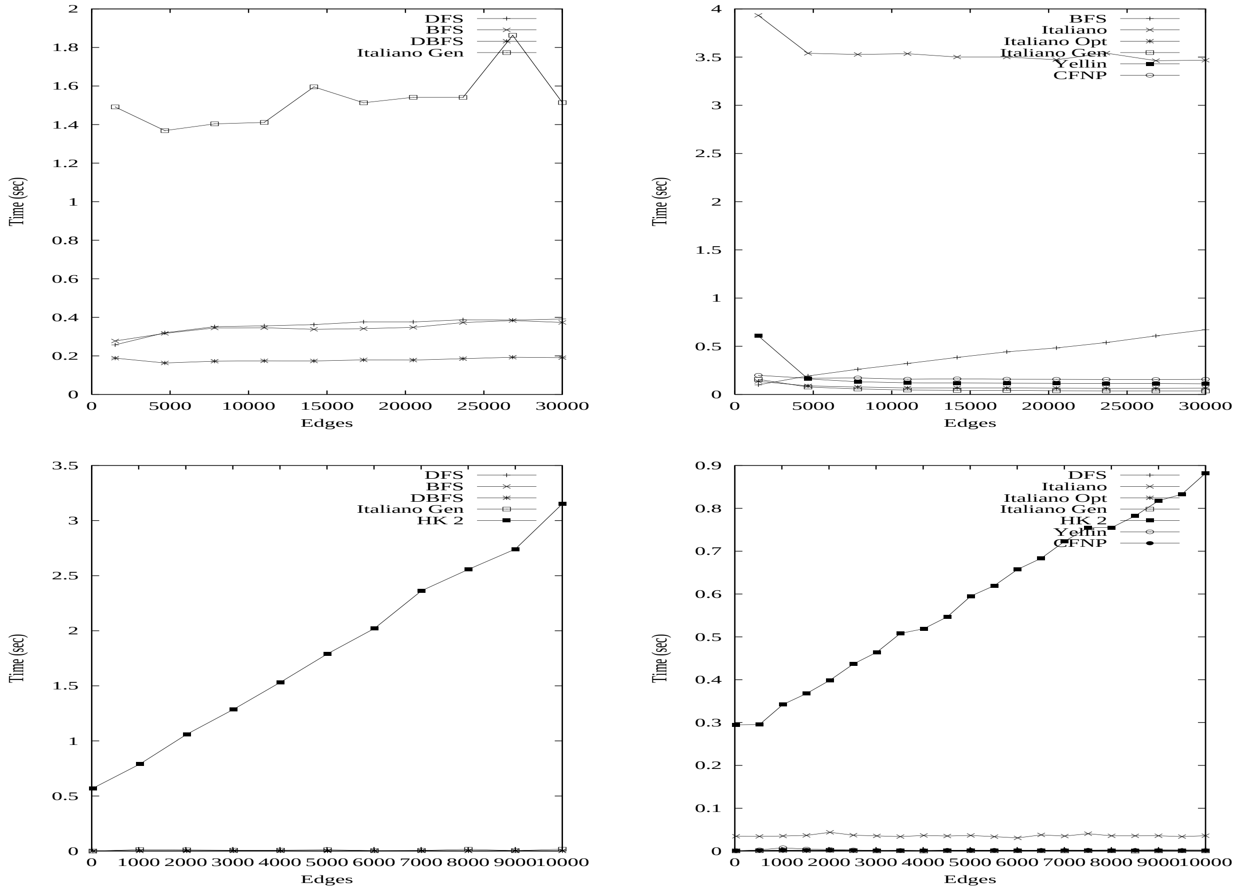

Figure 5: Edge insertions, deletions and queries on digraphs (left column) and DAGs (right column) with 500 vertices. The top row shows results for large sequences (5000 operations), while the bottom row shows results for small sequences (50 operations). All experiments were run on the Ultra-Sparc.

3.2 Non-random inputs

In addition to random inputs, we also wanted to test our implementations on structured, non-random inputs that try to enforce bad update patterns for the dynamic algorithms. The construction of our non-random inputs were inspired from those proposed in [1]. We considered both non-random digraphs and non-random DAGs.

Non-random digraphs and their sequences were defined as follows. Each such graph $G(n,k)$ is characterized by two parameters: n being the number of vertices and an integer k, $k > 0$. $G(n,k)$ has exactly n vertices grouped into $s = \lceil n/k \rceil$ different groups V_i, $1 \leq i \leq s$. The first $s-1$ groups have exactly k vertices each, while the last group has the remaining vertices. Each V_i induces a subgraph G_i of $G(n,k)$, which is the complete graph $K_{|V_i|}$ on V_i. We shall occasionally refer to k as the clique size of $G(n,k)$. Furthermore, there is a set B of up to $s-1$ edges in $G(n,k)$: edge e_i, $1 \leq i \leq s-1$, is a bridge from a vertex in V_i to a vertex in V_{i+1}. Depending on the type of updates we want to generate, the edges in B are precisely those which are inserted or deleted from the graph during a sequence of operations. This will force the dynamic algorithms to handle dense subgraphs while the reachability information of the whole graph keeps changing. In an incremental (resp. decremental) environment the sequence of edge insertions (resp. deletions)

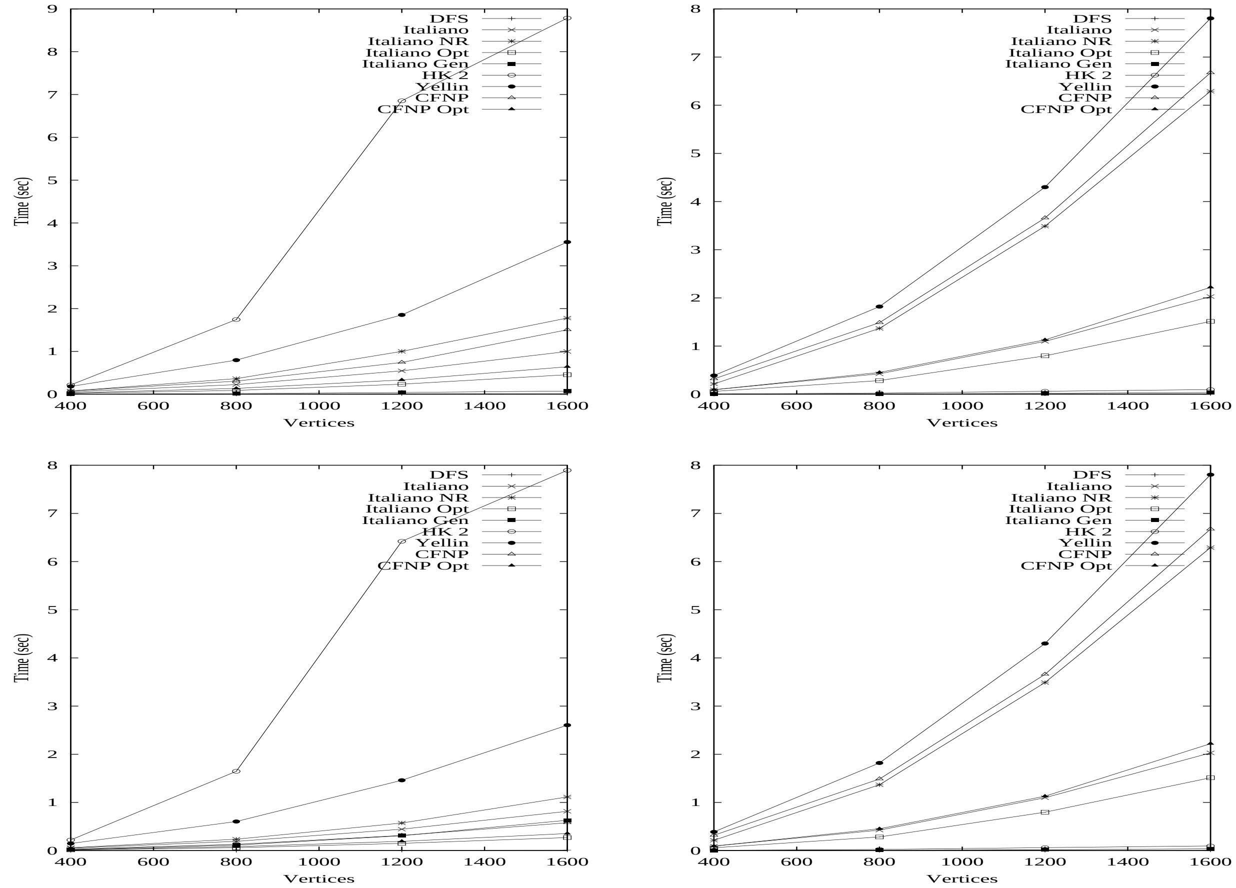

Figure 6: Edge insertions and queries on non-random digraphs (top row) and DAGs (bottom row) for various numbers of vertices. The left column shows results for graphs with clique size 10 and sequences of up to 320 operations, while the right column shows results for graphs with clique size 40 and sequences of up to 80 operations.

is intermixed evenly with queries. There is a specific order of edge insertions (resp. deletions) in an incremental (resp. decremental) environment. In the incremental case, $G(n,k)$ has initially no bridges. Bridges are added from B as follows: the first is inserted between G_1 and G_2, the second between G_{s-1} and G_s, the third between G_2 and G_3, and so on. Hence, the bridge inserted last will provide new reachability information from roughly $n/2$ to the other $n/2$ vertices of $G(n,k)$. The reverse order is followed in the case of edge deletions, where all edges from B were initially in $G(n,k)$. Finally, in the case of fully dynamic environments, the sequence of operations consists of alternating subsequences of $s-1$ edge insertions and $s-1$ edge deletions intermixed evenly with queries.

Non-random DAGs were defined analogously: each G_i is simply the complete DAG on V_i. The operation sequences were generated similarly to those for non-random digraphs.

We considered non-random digraphs and DAGs with $n = 400, 800, 1200, 1600$ vertices and $k = 5, 10, 20, 40$. (We conducted experiments with other values of n and k and reported very similar results.) Note that the larger the k, the more structure $G(n,k)$ has. When k is small, $G(n,k)$ is basically a sparse graph. We ran many experiments on these graphs in incremental, decremental, and fully dynamic environments. Due to the size of the graphs, we had to use the

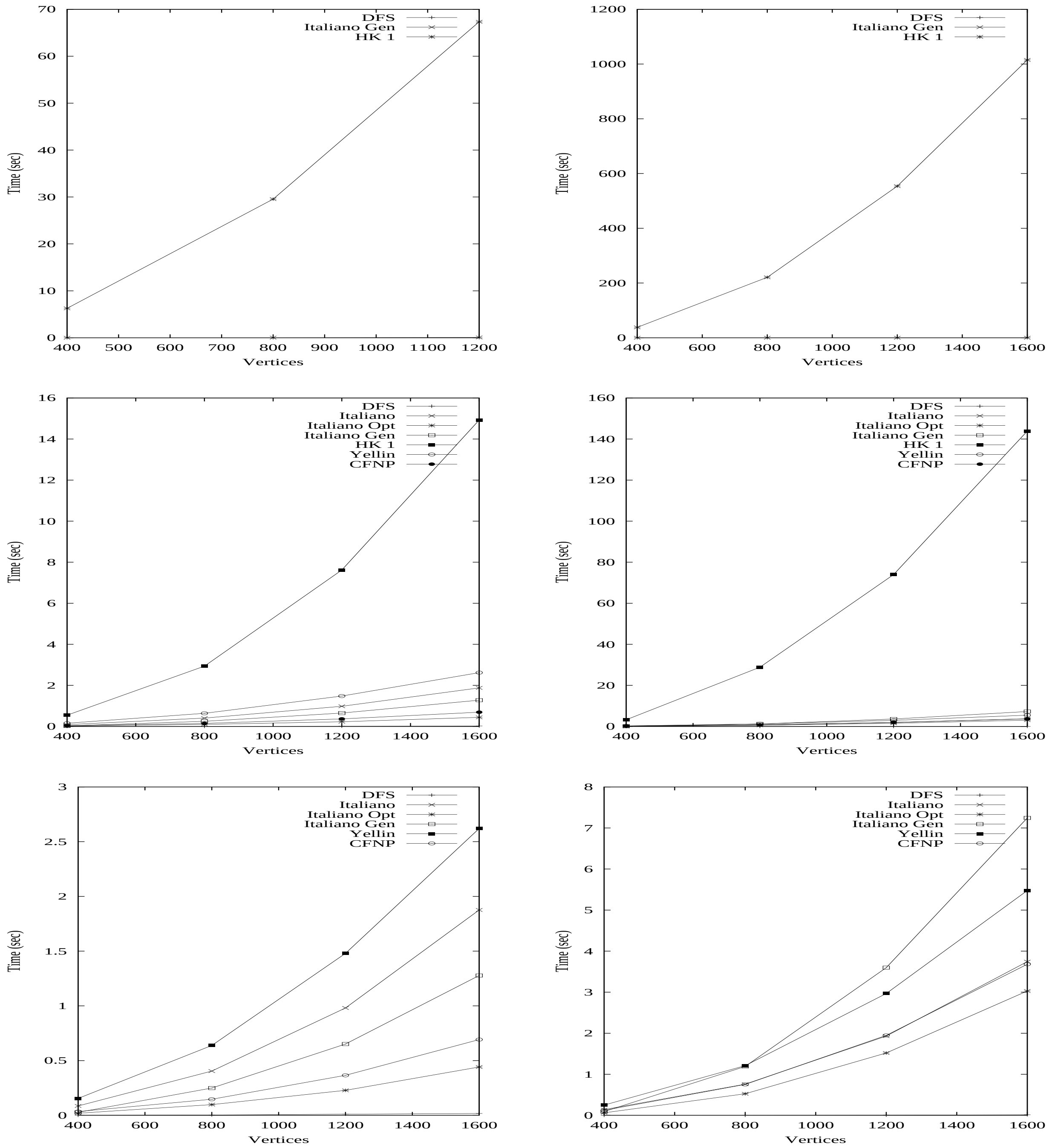

Figure 7: Edge deletions and queries on non-random digraphs (top row) and DAGs (middle and bottom rows) for various numbers of vertices. The left column shows results for graphs with clique size 10 and sequences of up to 320 operations, while the right column shows results for graphs with clique size 40 and sequences of up to 80 operations. The graphics of the middle and bottom rows concern the same experiment; they only differ in the inclusion or not of `HK-1`.

machine with the larger memory (Ultra-Sparc). We generated a large collection of data sets, each consisting of 10 samples. We ran our algorithms on each sample and retained the average CPU time in seconds over the 10 samples for each program. In all cases the simple-minded algorithms were significantly faster (the fastest being `DFS`) than any of the dynamic algorithms.

In the case of edge insertions, `DFS` was at least 4 times faster than the best dynamic (`Ital-Gen`). The `HK-2` algorithm exhibited an interesting behavior demonstrating the cost of rebuilding its data structure in practice. For small values of k (e.g., 5,10), `HK-2` was the slowest. For larger values of k (e.g., 40) it was competitive with `Ital-Gen` and in the case of non-random DAGs it became faster than `Ital-Gen`; see Figure 6. This could be explained by the fact that in all experiments we considered with $k = 40$, the number of updates either never exceeded $\sqrt{n}$ or exceeded it only once. Hence at most one rebuilding of the data structure took place.

We now turn to the case of edge deletions. For non-random DAGs, `DFS` was at least 4 times faster than the best dynamic (`Ital-Opt`). `HK-1` was always by far the slowest (regardless of the values of n and k); see Figure 7. For non-random digraphs, `DFS` was from 2 to 3 times faster than `Ital-Gen`. `HK-1` was again extremely slow, especially on graphs with strong structural properties ($k = 40$); cf. Figure 7.

Finally, we consider the case of a fully dynamic environment. Again `DFS` was at least 4 times faster than the best dynamic, `Ital-Gen` for non-random digraphs and `Ital-Opt` for non-random DAGs. The `HK-2` algorithm exhibited a behavior analogous to that in the incremental case. For small values of k (e.g., 5,10), `HK-2` was the slowest. For larger values of k (e.g., 40), the gap in its performance w.r.t. `Ital-Gen` was reduced drastically in non-random digraphs (but `HK-2` was never faster), while in the case of non-random DAGs it became faster than `Ital-Opt`; see Figure 8. The difference in behavior of `HK-2` between the incremental (cf. right column graphics of Figure 6) and the fully dynamic case (cf. top right and middle right graphics of Figure 8) is due to the fact that the algorithm performs much more work (involving calls to `HK-1`) when edge deletions occur. The impact of rebuilding the data structure is illustrated in the bottom row of Figure 8, where the performances of the algorithms are shown w.r.t. the length of the operation sequence in non-random DAGs with $n = 1600$ and $k = 10$ (bottom left), and $k = 40$ (bottom right). In the $k = 10$ case, there are "jumps" in the performance of `HK-2` about every 80 operations (i.e., after about $40 = \sqrt{n}$ updates). In the case $k = 40$, no rebuild occurs during the sequence of operations.

3.3 "Real-world" inputs

In this section we describe results obtained on an input motivated by a real-world graph. Namely, we considered a graph I describing the connections and policy strategies among the autonomous systems of a fragment of the Internet network visible from *RIPE* (`www.ripe.net`), one of the main European servers. The complete interpretation of I is provided in [6]. Each vertex in I represents an autonomous system, i.e., a group of IP networks having a single clearly defined routing policy which is run by one or more network operators. Edges represent *directed logical links.* A *logical link* between two autonomous systems denotes that either there is a direct connection between the two systems, or that the two systems share a gateway. A *direction* in a logical link represents a routing policy strategy between the two autonomous systems or the direction in which messages are accepted. The graph I has 1259 vertices and 5101 edges.

Since most of our dynamic algorithms (`Ital`, `Ital-Opt`, `CFNP`, `Yellin`) can be used in a decremental or fully dynamic setting only on DAGs and since running `HK-2` on this big graph was problematic, we converted I to a DAG by changing the direction of some edges. On the resulted graph, we performed random sequences of operations, since no "real" sequences of operations were

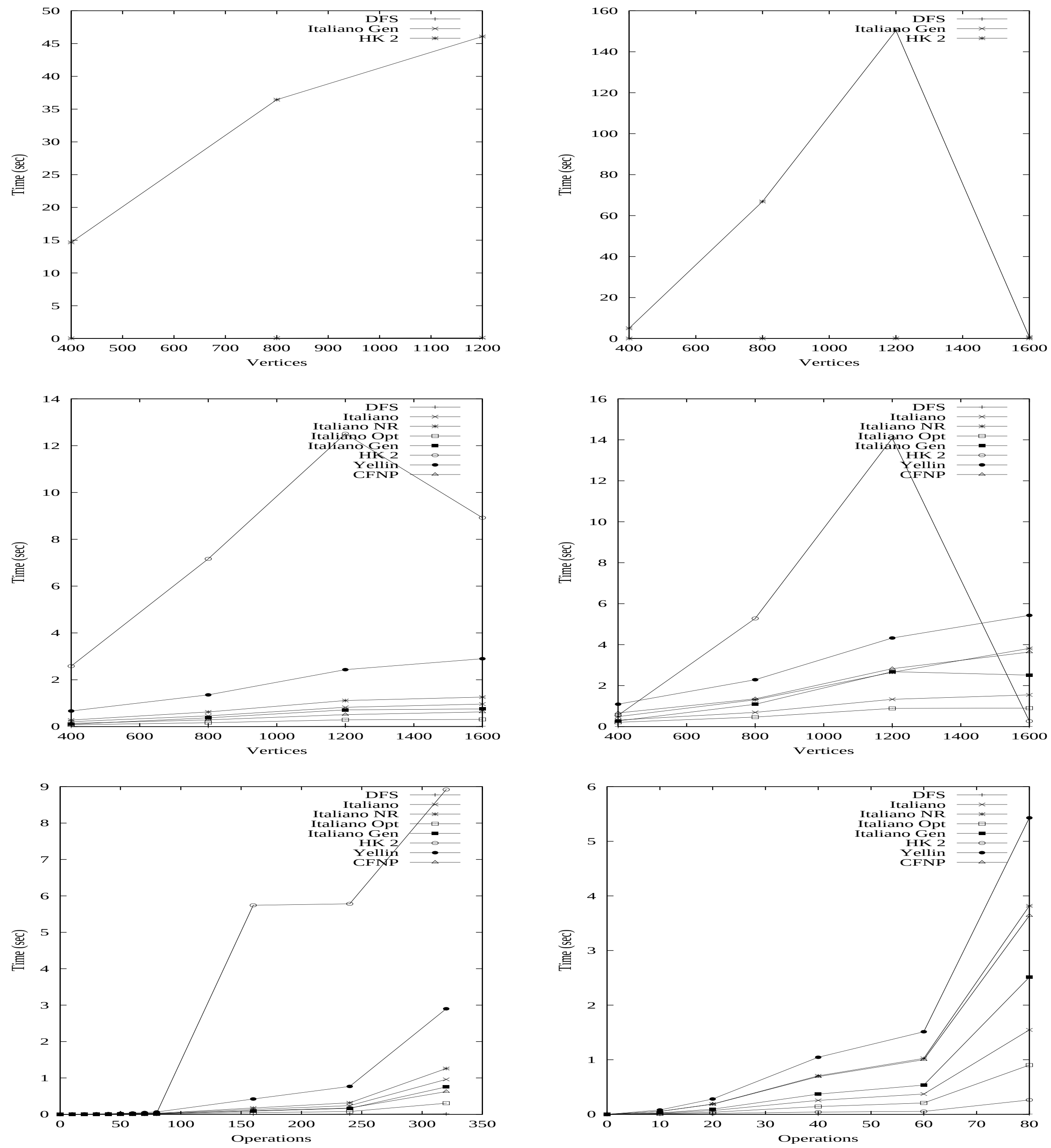

Figure 8: Edge insertions, deletions and queries on non-random digraphs (top row) and DAGs (middle and bottom rows) for various numbers of vertices. The left column shows results for graphs with clique size 10 and sequences of up to 320 operations, while the right column shows results for graphs with clique size 40 and sequences of up to 80 operations. The graphics of the bottom row is a detailed view (w.r.t. the number of operations) of the part of the experiment in the middle row that concerns the graph with 1600 vertices.

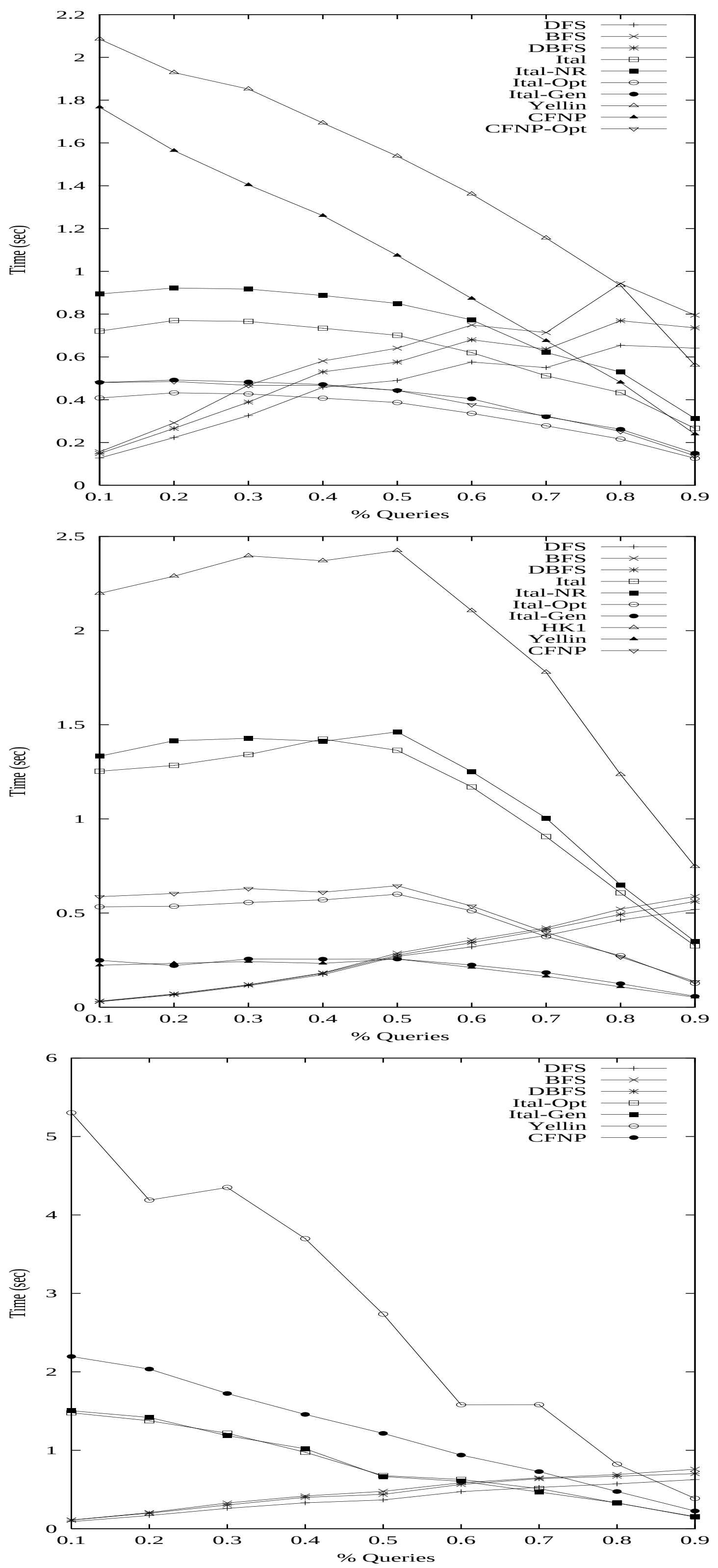

Figure 9: Experimental results with the fragment of the internet graph. From top to bottom: incremental, decremental, and fully dynamic sequences.

provided. We ran several experiments with various lengths of operation sequences and observed no substantial differences in the behavior of the algorithms compared with the experiments on random inputs. For this reason, we decided to perform experiments with different percentage of queries in the sequence of operations (from 10% to 90%). These experiments may give useful suggestions on how to proceed if one knows in advance the update-query pattern. The results for large sequences of operations are illustrated in Figure 9 (the experiments were run on the Ultra-Sparc). Similar results hold for smaller or larger sequences. The results show the expected behavior, but also provide 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 worst the simple-minded algorithms. The break points, however, differ in each dynamic setting.

4 Conclusions

In this paper we have performed an extensive experimental study of several dynamic algorithms for transitive closure. We have reported experiments on random inputs, on more structured (non-random) inputs and on an input motivated by a real-world graph. We have shown with experimental data that there are several cases where some of the dynamic algorithms can be quite fast in practice.

We plan to continue this experimental work by implementing the recent fully dynamic algorithms in [9, 24, 25]. The efficient implementation of these algorithms, however, may be very time consuming. Nevertheless, we believe that once all these implementations are available, an extensive experimental study of all algorithms may shed new light in the development of better dynamic algorithms for transitive closure.

Acknowledgments

We are grateful to Giulio Pasqualone and Guido Schaefer for their contribution to the development of the software used in the reported experiments and to Pino Italiano for many helpful discussions. We are also indebted to the Max-Planck-Institut für Informatik for the generous availability of the required computing resources.

References

[1] D. Alberts, G. Cattaneo, and G. F. Italiano. An empirical study of dynamic graph algorithms. *ACM Journal of Experimental Algorithmics*, 2:# 5, 1997.

[2] 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*, pp. 314–323, 1997.

[3] D. Alberts, G. Cattaneo, G.F. Italiano, U. Nanni, and C. Zaroliagis. A software library of dynamic graph algorithms. In *Proc. 1st Workshop on Algorithms and Experiments*, pp. 129–136, 1998.

[4] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12:615–638, 1991.

[5] B. Bollobás. *Random Graphs*, Academic Press, New York, 1985.

[6] T. Bates, E. Gerich, L. Joncheray, J-M. Jouanigot, D. Karrenberg, M. Terpstra, and J. Yu. Representation of IP routing policies in a routing registry. Technical report, RIPE-181, October 1994.

[7] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. *Algorithmica*, 27 (3):212–226, 2000, Special Issue on Treewidth, Graph Minors, and Algorithms.

[8] 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.

[9] 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*, to appear, 2000.

[10] 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. Lecture Notes in Computer Science*, to appear, 2000.

[11] H. Djidjev, G. Pantziou, and C. Zaroliagis. Improved algorithms for dynamic shortest paths. *Algorithmica*, in print.

[12] D. Eppstein, Z. Galil, G. F. Italiano. Dynamic graph algorithms. CRC Handbook of Algorithms and Theory of Computation, Chapter 22, CRC Press, 1999.

[13] 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.

[14] D. Eppstein, Z. Galil, G. F. Italiano, T. H. Spencer. Separator based sparsification I: Planarity testing and minimum spanning trees. *Journal of Computer and System Science*, 52 (1):3–27, 1996, Special issue of STOC'93.

[15] D. Eppstein, Z. Galil, G. F. Italiano, T. H. Spencer. Separator based sparsification II: Edge and vertex connectivity. *SIAM Journal on Computing*, 28:341–381, 1999.

[16] S. Even and Y. Shiloach. An on-Line edge deletion problem. *Journal of the ACM*, 28:1–4, 1981.

[17] 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.

[18] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34 (2):251–281, 2000.

[19] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni. Fully dynamic shortest paths and negative cycles detection in digraphs with arbitrary arc weights. In *Proc. 6th European Symposium on Algorithms*, pp. 320–331. *Lecture Notes in Computer Science*, vol. 1461, 1998.

[20] 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*, pp. 368-380. *Lecture Notes in Computer Science*, vol. 1461, 1998.

[21] A. Goralcikova, and V. Konbeck. A reduct and closure algorithm for graphs. In *Proc. 4th Mathematical Foundations of Computer Science*, pp. 301–307. *Lecture Notes in Computer Science*, vol. 74, 1979.

[22] M. R. Henzinger, and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th ACM Symposium on Theory of Computing*, pp. 519–527, 1995.

[23] M. R. Henzinger, and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. 36th IEEE Symposium on Foundations of Computer Science*, pp. 664–672, 1995.

[24] 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*, pp. 81–91, 1999.

[25] V. King, and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proc. 31st ACM Symposium on Theory of Computing*, pp. 492–498, 1999.

[26] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[27] G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28:5–11, 1988.

[28] 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*, pp. 106–120. *Lecture Notes in Computer Science*, vol. 314, 1988.

[29] K. Mehlhorn and S. Naher. *LEDA - A platform for combinatorial and geometric computing.* Cambridge University Press, 1999.

[30] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

[31] M. Rauch. Improved data structures for fully dynamic biconnectivity. In *Proc. 26th ACM Symposium on Theory of Computing*, pp. 686-695, 1994.

[32] D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993.

A Appendix

We present here a detailed set of experimental results regarding the random digraphs and DAGs we considered for sequences σ of operations of medium size (i.e., $|\sigma| = 500$). All experiments were run on the Ultra-Sparc.

n	m_0	DFS	BFS	DBFS	Ital	Ital-Opt	Ital-Gen	Yellin	CFNP	CFNP-Opt
100	50	0.001	0.002	0.001	0.006	0.001	0.007	0.016	0.004	0.002
100	150	0.002	0.003	0.002	0.005	0.001	0.009	0.015	0.005	0.001
100	450	0.003	0.003	0.002	0.001	0.000	0.001	0.004	0.004	0.000
100	1010	0.003	0.003	0.001	0.000	0.001	0.000	0.004	0.004	0.000
100	2410	0.003	0.003	0.001	0.000	0.000	0.001	0.004	0.004	0.000
300	150	0.003	0.004	0.004	0.016	0.006	0.011	0.028	0.009	0.008
300	350	0.006	0.007	0.007	0.038	0.009	0.036	0.099	0.020	0.012
300	1710	0.012	0.012	0.007	0.000	0.000	0.002	0.020	0.015	0.000
300	5200	0.009	0.010	0.005	0.000	0.000	0.001	0.017	0.014	0.000
300	22500	0.009	0.010	0.005	0.000	0.000	0.001	0.018	0.014	0.000
500	250	0.004	0.003	0.003	0.017	0.009	0.014	0.010	0.013	0.013
500	450	0.004	0.005	0.005	0.055	0.016	0.033	0.111	0.025	0.020
500	3110	0.017	0.020	0.010	0.001	0.001	0.004	0.038	0.028	0.001
500	11200	0.015	0.018	0.009	0.000	0.000	0.000	0.031	0.025	0.000
500	67000	0.012	0.016	0.008	0.000	0.000	0.000	0.032	0.023	0.000
700	350	0.006	0.006	0.006	0.021	0.015	0.020	0.006	0.019	0.019
700	1010	0.014	0.019	0.017	0.105	0.030	0.098	0.258	0.058	0.032
700	4510	0.027	0.031	0.016	0.001	0.000	0.004	0.052	0.043	0.000
700	18500	0.023	0.024	0.011	0.000	0.000	0.001	0.048	0.038	0.000
700	76000	0.024	0.029	0.014	0.000	0.000	0.000	0.050	0.037	0.000

Figure 10: Insertions and queries on Digraphs

n	m_0	DFS	BFS	DBFS	Ital	Ital-NR	Ital-Opt	Ital-Gen	Yellin	CFNP	CFNP-Opt
100	50	0.001	0.001	0.001	0.003	0.003	0.001	0.001	0.006	0.002	0.001
100	150	0.001	0.002	0.002	0.002	0.002	0.001	0.001	0.006	0.002	0.001
100	450	0.002	0.002	0.002	0.001	0.001	0.000	0.001	0.002	0.002	0.001
100	1010	0.003	0.003	0.002	0.000	0.000	0.000	0.001	0.002	0.002	0.000
100	2410	0.004	0.005	0.004	0.000	0.000	0.000	0.001	0.002	0.002	0.000
300	150	0.003	0.003	0.004	0.010	0.011	0.005	0.007	0.013	0.007	0.006
300	350	0.004	0.005	0.005	0.015	0.016	0.005	0.008	0.029	0.009	0.006
300	1710	0.009	0.010	0.010	0.001	0.002	0.001	0.001	0.008	0.009	0.001
300	5200	0.013	0.014	0.014	0.001	0.000	0.000	0.001	0.006	0.008	0.000
300	22500	0.047	0.037	0.043	0.000	0.000	0.000	0.000	0.005	0.007	0.000
500	250	0.003	0.004	0.003	0.013	0.014	0.008	0.012	0.007	0.012	0.011
500	450	0.004	0.004	0.004	0.027	0.028	0.010	0.016	0.041	0.015	0.013
500	3110	0.014	0.018	0.018	0.003	0.004	0.002	0.003	0.014	0.017	0.002
500	11200	0.034	0.033	0.039	0.000	0.001	0.000	0.001	0.010	0.016	0.001
500	67000	0.163	0.111	0.145	0.000	0.001	0.000	0.001	0.008	0.013	0.000
700	350	0.006	0.006	0.006	0.018	0.017	0.014	0.017	0.005	0.016	0.017
700	1010	0.009	0.011	0.011	0.045	0.051	0.017	0.025	0.089	0.029	0.019
700	4510	0.023	0.028	0.029	0.004	0.006	0.003	0.004	0.020	0.026	0.003
700	18500	0.056	0.049	0.063	0.001	0.000	0.001	0.001	0.015	0.023	0.001
700	76000	0.228	0.151	0.223	0.000	0.000	0.000	0.000	0.015	0.023	0.000

Figure 11: Insertions and queries on DAGs

n	m_0	DFS	BFS	DBFS	Ital	Ital-NR	Ital-Opt	Ital-Gen	HK-1	Yellin	CFNP
100	450	0.002	0.002	0.002	0.008	0.008	0.002	0.002	0.016	0.007	0.003
100	1010	0.002	0.002	0.002	0.004	0.004	0.001	0.001	0.009	0.003	0.002
100	2410	0.003	0.004	0.003	0.004	0.004	0.002	0.001	0.007	0.002	0.003
300	300	0.003	0.003	0.003	0.016	0.017	0.007	0.004	0.022	0.005	0.006
300	1710	0.007	0.009	0.009	0.032	0.033	0.011	0.010	0.078	0.018	0.012
300	5200	0.011	0.014	0.013	0.019	0.019	0.006	0.004	0.034	0.007	0.009
300	22500	0.044	0.037	0.043	0.018	0.017	0.006	0.003	0.026	0.007	0.009
500	250	0.003	0.003	0.003	0.030	0.031	0.011	0.005	0.022	0.002	0.011
500	500	0.004	0.003	0.003	0.035	0.035	0.013	0.007	0.037	0.008	0.012
500	3110	0.013	0.017	0.017	0.059	0.059	0.020	0.016	0.124	0.025	0.022
500	11200	0.036	0.033	0.041	0.042	0.042	0.014	0.007	0.057	0.013	0.018
500	67000	0.164	0.112	0.148	0.036	0.035	0.011	0.005	0.051	0.012	0.016
700	350	0.006	0.006	0.006	0.056	0.056	0.018	0.008	0.041	0.001	0.018
700	700	0.006	0.006	0.006	0.063	0.064	0.021	0.010	0.081	0.014	0.020
700	4510	0.022	0.027	0.028	0.104	0.105	0.036	0.033	0.282	0.046	0.036
700	18500	0.062	0.054	0.073	0.070	0.072	0.021	0.012	0.124	0.019	0.028
700	76000	0.231	0.154	0.225	0.066	0.065	0.020	0.009	0.109	0.018	0.026

Figure 12: Deletions and queries on DAGs

n	m_0	DFS	BFS	DBFS	Ital-Gen	HK-1
100	450	0.004	0.005	0.004	0.032	0.082
100	1010	0.005	0.005	0.002	0.028	0.022
100	2410	0.004	0.005	0.002	0.035	0.016
300	300	0.006	0.007	0.006	0.005	0.044
300	1710	0.016	0.019	0.011	0.101	0.168
300	5200	0.016	0.019	0.009	0.118	0.085
300	22500	0.018	0.021	0.010	0.164	0.058
500	250	0.006	0.006	0.006	0.008	0.043
500	500	0.007	0.007	0.006	0.012	0.089
500	3110	0.030	0.036	0.019	0.215	0.291
500	11200	0.028	0.032	0.015	0.230	0.150
500	67000	0.030	0.034	0.016	0.247	0.124
700	350	0.012	0.013	0.012	0.010	0.084
700	700	0.012	0.012	0.013	0.020	0.178
700	4510	0.049	0.054	0.032	0.332	0.508
700	18500	0.048	0.045	0.024	0.399	0.284
700	76000	0.047	0.053	0.027	0.403	0.227

Figure 13: Deletions and queries on Digraphs

n	m_0	DFS	BFS	DBFS	Ital-Gen	HK-2
100	125	0.003	0.003	0.002	0.012	0.860
100	438	0.006	0.006	0.004	0.019	1.304
100	1010	0.005	0.005	0.002	0.018	
100	2410	0.006	0.007	0.002	0.025	
300	125	0.006	0.006	0.006	0.010	5.547
300	281	0.008	0.007	0.010	0.020	5.610
300	1710	0.022	0.023	0.014	0.070	
300	5200	0.021	0.021	0.011	0.073	
300	22500	0.023	0.023	0.012	0.134	
500	229	0.006	0.006	0.005	0.019	6.360
500	490	0.007	0.007	0.007	0.038	6.856
500	3110	0.036	0.040	0.024	0.136	
500	11200	0.034	0.039	0.019	0.143	
500	67000	0.035	0.036	0.019	0.188	
700	334	0.013	0.013	0.013	0.027	15.859
700	700	0.012	0.013	0.013	0.065	16.703
700	4510	0.060	0.062	0.038	0.200	
700	18500	0.059	0.060	0.030	0.234	
700	76000	0.059	0.059	0.034	0.175	

Figure 14: Insertions, Deletions and queries on Digraphs. An empty entry denotes that the particular experiment couldn't finish due to memory limitations.

n	m_0	DFS	BFS	DBFS	Ital	Ital-NR	Ital-Opt	Ital-Gen	HK-2	Yellin	CFNP
100	125	0.002	0.001	0.001	0.014	0.013	0.002	0.002	0.410	0.010	0.003
100	438	0.002	0.002	0.002	0.015	0.016	0.002	0.002	0.489	0.005	0.004
100	1010	0.002	0.003	0.003	0.013	0.013	0.001	0.000		0.002	0.002
100	2410	0.004	0.005	0.004	0.012	0.012	0.001	0.000		0.002	0.003
300	125	0.003	0.003	0.003	0.154	0.156	0.007	0.005	2.780	0.002	0.008
300	281	0.003	0.003	0.004	0.155	0.155	0.010	0.007	2.770	0.010	0.009
300	1710	0.009	0.012	0.011	0.155	0.159	0.007	0.006		0.013	0.012
300	5200	0.013	0.015	0.015	0.105	0.106	0.004	0.002		0.007	0.009
300	22500	0.049	0.038	0.044	0.106	0.106	0.004	0.002		0.007	0.010
500	229	0.003	0.003	0.003	0.606	0.610	0.016	0.011	3.127	0.002	0.015
500	490	0.003	0.004	0.004	0.641	0.639	0.019	0.015	3.202	0.021	0.018
500	3110	0.015	0.018	0.019	0.630	0.632	0.012	0.011		0.022	0.022
500	11200	0.035	0.036	0.042	0.530	0.530	0.008	0.006		0.013	0.019
500	67000	0.194	0.125	0.161	0.339	0.338	0.006	0.004		0.010	0.015
700	334	0.006	0.006	0.006	1.338	1.341	0.026	0.017	7.846	0.002	0.025
700	700	0.006	0.007	0.006	0.693	0.693	0.014	0.012	7.485	0.029	0.016
700	4510	0.027	0.033	0.032	1.421	1.418	0.019	0.019		0.034	0.033
700	18500	0.076	0.066	0.088	1.290	1.291	0.011	0.008		0.020	0.029
700	76000	0.276	0.190	0.279	1.023	1.045	0.011	0.005		0.017	0.026

Figure 15: Insertions, Deletions and queries on DAGs. An empty entry denotes that the particular experiment couldn't finish due to memory limitations.