Research Academic
COMPUTER TECHNOLOGY INSTITUTE

TECHNICAL REPORT No. TR 2005/07/01

AN EXPERIMENTAL STUDY OF ALGORITHMS FOR FULLY DYNAMIC TRANSITIVE CLOSURE

Ioannis Krommidas Christos D. Zaroliagis

July 2005; revised August 2006

An Experimental Study of Algorithms for Fully Dynamic Transitive Closure[*]

Ioannis Krommidas[†] Christos Zaroliagis[†]

August 2, 2006

Abstract

We have conducted an extensive experimental study on algorithms for fully dynamic transitive closure. We have implemented the recent fully dynamic algorithms by King [18], Roditty [24], Roditty & Zwick [25, 26], and Demetrescu & Italiano [10] along with several variants of them, and compared them to pseudo fully dynamic and simple-minded algorithms developed in a previous study [13]. We tested and compared these implementations on random inputs, synthetic (worst-case) inputs, and on inputs motivated by real-world graphs. Our experiments reveal that some of the dynamic algorithms can really be of practical value in many situations.

1 Introduction

The *transitive closure* (or *reachability*) problem in a digraph G consists in finding whether there is a directed path between any two vertices in G. In this paper, we are concerned with the dynamic version of the problem, namely with the maintenance of transitive closure when G undergoes a sequence of edge insertions and deletions. This is a fundamental and extensively studied problem. An algorithm is called *fully dynamic* if it supports both edge insertions and deletions, and *partially dynamic* if either insertions or deletions (but not both) are supported; in the former case the partially dynamic algorithm is called *incremental*, while in the latter it is called *decremental*.

Recently, we have witnessed a number of important theoretical breakthroughs regarding fully dynamic transitive closure [10, 19, 18, 20, 24, 25, 26], which is our main concern in this work. These fully dynamic algorithms can be roughly divided into two categories: those using combinatorial techniques [18, 20, 24, 25, 26] and those which do not exclusively use such techniques [10, 19, 25]. Moreover, some of these algorithms apply to Directed Acyclic Graphs (DAGs), while some others to general digraphs.

In this paper, we concentrate on *fully dynamic algorithms* for maintaining the transitive closure of a *general digraph* $G = (V, E)$, and hence we mention the results related to this case only. Starting from algorithms which do not exclusively use combinatorial techniques, King & Sagert [19] presented a randomized (Monte Carlo) algorithm achieving $O(n^{2.26})$ amortized update time and $O(1)$ query time (if not stated otherwise, queries are considered Boolean and their associate time worst-case), where $n = |V|$. These results were improved by Demetrescu & Italiano in [10], where a deterministic algorithm is presented achieving $O(1)$ query time and $O(n^2)$ amortized update time.

[*]This work was partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contracts no. IST-2002-001907 (integrated project DELIS) and no. FP6-021235-2 (project ARRIVAL). A preliminary version of this work appeared in [21].

[†]Computer Technology Institute, N. Kazantzaki Str, Patras University Campus, 26500 Patras, Greece, and Dept of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece. Emails: {krommudi,zaro}@ceid.upatras.gr

Finally, Roditty & Zwick [25], building on the above as well as on a new decremental algorithm (see below), presented a randomized (Monte Carlo) algorithm, which has an $O(m^{0.43})$ query time and an $O(m^{0.58}n)$ amortized update time, where m denotes the current number of edges in G plus the number of edges to be inserted or deleted. Turning now to the fully dynamic algorithms that are mostly based on combinatorial techniques, King in [18] gave a deterministic algorithm that achieves an $O(n^2 \log n)$ amortized update time and $O(1)$ query time. This algorithm (as well as the aforementioned ones in [10, 19]) uses $O(n^3)$ space. A space-saving technique introduced by King & Thorup [20] reduces the space requirements to $O(n^2 \log n)$. Roditty & Zwick [25] presented a deterministic algorithm with an $O(\sqrt{n})$ query time and an $O(m\sqrt{n})$ amortized update time. This algorithm (as well as the aforementioned one in [25]) is based on a new decremental randomized (Las Vegas) algorithm that achieves $O(n)$ amortized update time and $O(1)$ query time. Further improvements have been made by Roditty in [24], where a deterministic algorithm is presented that achieves $O(n^2)$ amortized update time and $O(1)$ query time. Finally, Roditty & Zwick [26] have very recently presented a new deterministic algorithm that supports each update in $O(m + n \log n)$ time and each query in $O(n)$ time.

Despite the above theoretical progress, we are not aware of any practical assessment of any of the aforementioned algorithms. Our prime goal in this paper is to advance our knowledge on the practical aspects of this recent and important theoretical work regarding fully dynamic maintenance of transitive closure. In particular, we follow up the advances in [18, 20, 24, 25, 26] on fully dynamic algorithms based on combinatorial techniques, as well as these in [10], with an extensive comparative experimental study and investigate their practical merits. Previous experimental studies regarding maintenance of transitive closure [1, 13] have mostly focussed on the assessment of partially dynamic algorithms. The only experimental comparison regarding fully dynamic transitive closure was made by Frigioni et al [13], where a fully dynamic algorithm of Henzinger & King [14] was compared to a new algorithm, called `Ital-Gen`, developed in that paper. `Ital-Gen` is based on a hybridization and extension of Italiano's partially dynamic algorithms [15, 16], works in a fully dynamic setting, but update times can be analyzed and bounded only for partially dynamic operation sequences. We shall refer to such algorithms as *pseudo fully dynamic*. The experiments conducted in [13] showed that `Ital-Gen` was considerably more efficient in practice than the fully dynamic algorithm in [14].

In this work, we have implemented and experimentally compared all the aforementioned combinatorially based fully dynamic algorithms [18, 20, 24, 25, 26], as well as the algorithm of Demetrescu & Italiano [10], for maintaining transitive closure in general digraphs along with some new variants of them. In particular, from the former set we have implemented the space-saving version of King's algorithm [18, 20] along with two new variants, the algorithm of Roditty & Zwick [25], the algorithm of Roditty [24] along with a new variant, and the very recent algorithm of Roditty & Zwick [26]. In addition, we have implemented the decremental algorithm of Roditty & Zwick [25], which we modified and fine-tuned so that it can work in a fully dynamic environment. We call this pseudo fully dynamic algorithm `RZ-Opt`. We compared the above implementations to the `Ital-Gen` algorithm developed in [13] and also to the simple-minded algorithms (easily implementable and with very small constants) presented in the same study. Our experiments were conducted on three types of inputs: random inputs, synthetic inputs that are worst-case inputs for the dynamic algorithms, and real-world inputs.

Our experiments showed that, regardless of the type and size of input, the algorithm of Demetrescu & Italiano [10], the space-saving version of King's algorithm [18, 20] and its new variants were by far the slowest, followed by the algorithm of Roditty [24]. The performance of the latter is actually surprising, since at least for some cases it is theoretically better than the algorithm in [25].

For random inputs, the pseudo fully dynamic algorithms `Ital-Gen` and `RZ-Opt` were dramatically faster than any of the fully dynamic ones or their variants, with `RZ-Opt` being usually the fastest. Regarding fully dynamic algorithms, the first interesting outcome is that the theoretically inferior – with respect to [26] – algorithm of Roditty & Zwick in [25] was the fastest. The second interesting outcome is that Demetrescu & Italiano's [10] algorithm exhibits an excellent locality of reference and achieves the smallest ratio of cache misses w.r.t. *any* algorithm in our study. However, its performance degrades, since it requires a vast number of main memory accesses to maintain its data structures.

For synthetic inputs, the fastest algorithm in all cases were the simple-minded ones. Regarding the dynamic algorithms, we observed that the situation is similar to the random inputs as long as the graph consists of strongly connected components (SCCs) of small size. When, however, the size of SCCs increases, the fully dynamic algorithms of Roditty & Zwick [25, 26] perform dramatically better than the pseudo fully dynamic ones. This implies that the fully dynamic algorithms demonstrate their theoretical superiority by learning quickly the specific structure of these graphs and benefiting substantially from it.

The experimental results with the real-world inputs were similar to those of random inputs.

Related Work. There are several papers complementing the wealth of theoretical work on dynamic graph algorithms with extensive experimental studies. In particular, there is a bulk of studies regarding the dynamic shortest path problem [8, 9, 12], as well as the dynamic minimum spanning tree and connectivity problems in undirected graphs [2, 3, 7, 17]. See [28] for a recent survey.

2 Algorithms and their Implementation

Let $G = (V, E)$ be a digraph with n vertices and an initial number of m_0 edges. If there is a directed path from a vertex u to a vertex v, then u is called an *ancestor* of v, v is called a *descendant* of u, and v is said to be *reachable* from u. 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.

In the rest of this section we give a short description of the algorithms considered in our experimental study. For simplicity, we consider only Boolean queries, and note that all algorithms can report the actual path in time proportional to its number of edges. Moreover, some algorithms support an extended set of insert and delete operations. In the following, we shall denote by m' the number of edges to be inserted and/or deleted, and consequently $m = m_0 + m'$.

2.1 The Algorithms of Italiano and their Extensions

We start with the partially dynamic algorithms of Italiano [15, 16]. The incremental algorithm applies to any digraph, while the decremental applies to DAGs. We describe the modified and fine-tuned implementation of these algorithms, presented in [13], and which is referred to as `Ital-Opt`.

Italiano's algorithm maintains for each vertex $u \in V$ a tree $Desc[u]$, which contains all descendants of u. The $Desc$ trees are maintained implicitly using a $n \times n$ matrix $Parent$. If a vertex v belongs to $Desc[u]$, then $Parent[u, v]$ points to the edge which connects v to its parent in $Desc[u]$, otherwise $Parent[u, v] = Null$.

During insertion of an edge (v, w), the data structure is updated only if (v, w) creates new paths from any ancestor u of v to any descendant of w; otherwise, it is simply added to the graph. In the former case, $Desc[u]$ is expanded using the information in $Desc[w]$. The deletion of an edge $e = (i, j)$ is done as follows. The edge e must be deleted from every tree $Desc[u]$ to which it belongs. Due to the deletion of e, the tree $Desc[u]$ breaks into two subtrees and it is updated as follows. If there exists an edge (x, j) such that the u-x path in $Desc[u]$ does not contain e, then reconstruct $Desc[u]$ by joining the two subtrees using the edge (x, j). In this case, x is called the *hook* of j. Otherwise, j is not a descendant of u, therefore j is deleted from $Desc[u]$ and the same process is applied recursively by deleting the outgoing edges of j in $Desc[u]$. A Boolean query for vertices

i and j is carried out in $O(1)$ time, by checking $Parent[i, j]$. The incremental part of `Ital-Opt` requires $O(n(m_0 + m'))$ time to process a sequence of m' edge insertions, while the decremental part requires $O(nm_0)$ time to process any number of edge deletions. The latter bound depends heavily on the fact that once a vertex x is considered as a hook for some vertex j, then it will never be a hook for j in any subsequent edge deletions.

Based on `Ital-Opt`, Frigioni et al. [13] developed a new algorithm called `Ital-Gen` that can handle edge insertions and deletions in general digraphs. In particular, it handles m' edge insertions in $O(n(m_0 + m'))$ time, while and it handles any sequence of edge deletions in $O(m_0^2)$ worst-case time.

The main idea of `Ital-Gen` is that if every strongly connected component (SCC) is replaced by a single vertex (called supervertex), then the resulting graph G' is a DAG, whose transitive closure can be maintained using `Ital-Opt`. For each SCC (supervertex) C, the algorithm maintains: (1) a graph representing C; (2) an array $Parent$ of length n, where $Parent[w]$ points to the edge that connects C to its parent in $Desc[w]$ (if such an edge does not exist, then $Parent[w] = Null$); (3) a *sparse certificate* S of C, which is a sparse subgraph of C such that if there exists a x-y path in C, then there also exists a x-y path in S (and vice versa). In addition, the algorithm maintains an $n \times n$ matrix $Index$ such that $Index(i, j)$ is true iff there is an i-j path.

The insertion of an edge is done similarly to `Ital-Opt`. A query can be answered in $O(1)$ time by checking the $Index$ matrix. The deletion of an edge $e = (u, w)$ is done as follows. If e belongs to a SCC C, then check whether e belongs to the sparse certificate S of C. If it does not, then nothing is done. Otherwise, check whether C has broken and in such a case the new SCCs are computed and the data structures are updated accordingly. If e does not belong to a SCC, then `Ital-Opt` is used to remove it.

In [13], it is also described how both `Ital-Opt` and `Ital-Gen` can be modified so that they can be used in a fully dynamic environment (to handle mixed sequences of edge insertions and deletions). The modification for `Ital-Opt` is based on a lazy updating of the hook values during edge insertions. The modification for `Ital-Gen` is based on the modified `Ital-Opt` and on the fact that instead of recomputing SCCs, their sparse certificates, and G' before any sequence of edge deletions, SCCs are merged to supervertices as soon as they are created.

2.2 The Algorithm of King and its Variants

King's algorithm [18] uses forests of BFS trees. An *Out* (resp. *In*) *BFS* tree of depth d rooted at vertex r is a data structure which maintains vertices reachable from (resp. reaching) r, and whose distance from r is less than or equal to d. Maintenance of this data structure for any sequence of edge deletions can be done in $O(m_0 d)$ time. The algorithm maintains $k = \lceil \log_2 n \rceil$ forests F^1, F^2, $\ldots$, F^k, where each F^i contains a pair of BFS trees In_u^i and Out_u^i of depth $d = 2$ rooted at every vertex $u \in V$. In addition, a number $k + 1$ of $n \times n$ matrices $count^i$, $i = 0, 1, \ldots, k$, and a number k of $n \times n$ matrices $list^i$, $i = 1, \ldots, k$, are maintained. Matrix $list^i(u, w)$ contains all vertices z such that $u \in In_z^i$ and $w \in Out_z^i$, and $count^i(u, w)$ is the number of these vertices. Matrix $count^0$ is defined as follows. If $(x, y) \in E$, then $count^0(x, y) = 1$, otherwise $count^0(x, y) = 0$.

The BFS trees of the forest F^1 are constructed for the graph G. The BFS trees of the forest F^i, $i > 1$, are constructed for the graph $G^i = (V, E^i)$, where E^i is defined as follows: $E^i = \{(x, y) : \text{count}^{i-1}(x, y) > 0\}$. The main idea of this algorithm is that if there exists a path L between two vertices u, w and the length of L is less than or equal to 2^j, then $count^j(u, w) > 0$.

The deletion of an edge e (or of a set of edges) is done as follows. The edge e is removed from any BFS tree of forest F^1 it belongs, and the matrices $count^1$ and $list^1$ are updated accordingly. Subsequently, the pairs of vertices x, y such that $count^1(x, y)$ became 0 are removed from the BFS trees of forest F^2. This process is repeated for all $2 \le i \le k$, until the matrices $count^k$ and $list^k$

are updated. The insertion of an edge (or a set of edges) incident to a vertex u is done as follows. The trees In_u^1, Out_u^1 are built from scratch and $count^1$, $list^1$ are updated accordingly. Then, BFS trees In_u^i and Out_u^i, $i = 2, \dots, k$ are built from scratch, and matrices $count^i$ and $list^i$ are updated. Each update operation is handled in $O(n^2kd) = O(n^2 \log n)$ amortized time. Boolean queries are answered in $O(1)$ time by checking $count^k$.

King and Thorup in [20] proposed a space saving version of this algorithm. Graphs $G^i = (V, E^i)$ are maintained using incidence matrices and BFS trees are built using these matrices. Specifically, if $(u, w) \in E^i$, then $M(u, w) = 1$, otherwise $M(u, w) = 0$ (M is the incidence matrix representing G^i). However, the maintenance of a BFS tree for any sequence of edge deletions costs now $O(n^2d)$ time, which is amortized across the edge deletions (and across all trees) and does not affect the amortized update bound. We shall refer to the implementation of this algorithm as `King-1`.

In addition, we have implemented a variant of this algorithm, called `King-2`. The idea is to maintain BFS trees of depth $d > 2$. In this case the number of necessary forests is reduced from $\lceil \log_2 n \rceil$ to $\lceil \log_d n \rceil$, and we wanted to investigate whether the reduction of forests affects performance. In `King-2` we considered $d = 8$, which reduces the number of forests by $2/3$. The asymptotic complexity of the update operations is the same with those of `King-1`. Furthermore, we have implemented another variant, called `King-3`, that maintains BFS trees of depth D, where D is the diameter of the graph, and therefore it requires only one forest of BFS trees.

2.3 The Algorithms of Roditty and Zwick

2.3.1 The Algorithms in [25] and their Extensions

Roditty and Zwick proposed in [25] a randomized (Las Vegas) decremental algorithm for maintaining the transitive closure of a graph. The algorithm is a combination of the decremental part of `Ital-Gen` in [13] with a new decremental algorithm for maintaining the SCCs of a digraph that also presented in [25]. The crucial observation is that the decremental part of `Ital-Gen` requires $O(nm_0)$ time to handle any sequence of edge deletions, if it does not perform any computations to determine whether a SCC has broken. This part in the `Ital-Gen` algorithm is now handled by the new algorithm in [25] for maintaining the SCCs in a decremental environment. Since this algorithm also requires $O(nm_0)$ time, the total running time of the decremental algorithm for maintaining the transitive closure for any sequence of edge deletions is $O(nm_0)$.

The algorithm for maintaining the SCCs works as follows. During initialization, the SCCs of the graph are computed and in each SCC C_j an In-BFS tree $In(w_j)$ and an Out-BFS tree $Out(w_j)$ rooted at w_j is initialized, where w_j is a random vertex belonging to C_j and called the *random representative* of C_j. In addition, an array A of length n is initialized such that $A(u) = w$ for every vertex u, where w is the random representative of the SCC containing u. Deletion of an edge (x, y) is done as follows. If x and y belong to different SCCs, then nothing is done. Otherwise, let C be the SCC containing x and y. The BFS trees of C are updated accordingly and in order to determine whether C has broken, it suffices to check whether $x \in In(w)$ and $y \in Out(w)$, where $w = A(x) = A(y)$. If C has broken, the new SCCs to which C breaks are computed and new BFS trees are constructed, except for the new SCC C' which contains w and inherits the BFS trees rooted at w.

The decremental transitive closure algorithm of Roditty and Zwick can handle only edge deletions. We have modified this algorithm so it can handle both edge insertions and deletions, without affecting the performance of the algorithm when it handles edge deletions. Specifically, edge deletions are processed as in the "original" algorithm and an edge insertion is handled as follows. If the new edge connects two different SCCs, then the incremental part of `Ital-Gen` is used to update the data structures used. If a new SCC C is created (due to the merge of two or more SCCs), then an In-BFS tree and an Out-BFS tree for C are initialized, apart from the other tasks performed by the algorithm. On the other hand, if the new edge belongs to a SCC, then the BFS trees of that SCC are updated as we describe below. We shall refer to the above pseudo fully dynamic algorithm that can work in a fully dynamic environment as `RZ-Opt`. Note that `RZ-Opt` handles any sequence of edge deletions in $O(nm_0)$ time and handles m' edge insertions in $O(m'(n + m_0 + m'))$ time. Consequently, we expect that `RZ-Opt` would be more efficient than `Ital-Gen` in handling edge deletions and `Ital-Gen` would be more efficient in handling edge insertions.

The update of an Out-BFS tree due to the insertion of an edge (x, y) is done as follows. Let $depth(x)$ be the depth of vertex x in the BFS tree. If $depth(y) > depth(x) + 1$, then, due to the insertion of (x, y), x becomes parent of y and we set $depth(y) = depth(x) + 1$. Then, we proceed recursively by examining all outgoing edges of y to check if other vertices are affected. This process requires $O(n + m_0 + m')$ worst-case time. The update of an In-BFS tree is done in a similar way. Consequently, in a sequence of m' edge insertions, up to $2m'$ trees may have to be updated, yielding an $O(m'(n+m_0+m'))$ worst-case cost. A boolean query is answered by checking the $Index$ matrix.

A final remark regarding the algorithm for maintaining the SCCs. When a SCC C breaks, then the new SCC C', which contains the random representative w of C, inherits the BFS trees rooted at w. The vertices of the trees, which do not belong to C', are chopped from the trees, so the trees do not have to be built from scratch. In our implementation of `RZ-Opt` these trees are built from scratch, therefore `RZ-Opt` may spend more than $O(nm_0)$ time to handle a sequence of m' edge deletions. Despite this fact, however, `RZ-Opt` proved to be competitive to the fastest algorithms implemented by Frigioni et al. [13] and for random inputs `RZ-Opt` was the fastest algorithm.

We now turn to the combinatorial fully dynamic algorithm in [25]. In its initialization phase, a decremental data structure for maintaining the transitive closure is initialized. This data structure can be maintained using `RZ-Opt` (or `Ital-Gen`). The insertion of an edge (or a set of edges) incident to a vertex u is done as follows. Vertex u is added to a set S of vertices and an ancestor (resp. descendant) tree $In(u)$ (resp. $Out(u)$) rooted at u is built. If the size of S becomes equal to a predetermined parameter t ($t = \sqrt{n}$ in [25] and in our implementation), then all data structures are re-initialized. The deletion of a set E' of edges is done as follows. First, every $e \in E'$ is removed from the decremental data structure. Then, for every $w \in S$, the trees $In(w)$ and $Out(w)$ are rebuilt. A query for an u-w path is computed as follows. First the decremental data structure is queried and if the answer is yes, then there exists a u-w path in G. If the answer is no and if a vertex z exists such that $u \in In(z)$ and $w \in Out(z)$, then again a u-w path exists in G. Otherwise, there is no u-w path. We shall refer to the implementation of this algorithm as `RZ-1`.

2.3.2 The Algorithm of Roditty [24]

The recent fully dynamic algorithm proposed by Roditty [24] is inspired by the algorithm of King [18]. It uses a decremental data structure for maintaining paths composed of "old" edges (edges belonging to the initial graph) and an algorithm for maintaining a forest of in-trees (ancestor trees) and out-trees (descendant trees) around each *insertion center* (i.e., the vertex incident to the current set of edge insertions). Boolean queries are answered in $O(1)$ time using an $n \times n$ matrix $count$ such that each entry $count(x, y)$ equals the number of insertion centers that lie on a path from x to y.

An out-tree (in-tree) around a vertex u maintains the so-called *blocks* with respect to u that are reachable from (reach) u. Two vertices x, y belong to the same *block with respect to* u, if x and y belong to the same SCC after the last edge insertion centered at u and after every subsequent delete operation. Generally, blocks change over time, because the insertion of a set of edges incident to a vertex u, may change the blocks with respect to u, while an edge deletion may change every block that exists so far. For example, if there exists a path between two vertices u and v, then there exists a vertex x such that the block, which contains u with respect to x, belongs to the in-tree of x, and the block, which contains v with respect to x, belongs to the out-tree of x. This occurs when there exists a path that contains at least one edge, which did not belong to the initial graph.

The main idea of this algorithm is that all in-trees and out-trees can be maintained implicitly using a single adjacency matrix M of size $O(n^2)$. This matrix is constructed in $O(n^2)$ time and it is updated after each edge deletion or insertion in $O(n^2)$ time. The algorithm has total running time $O(nm_0 + n^2m')$, where m' is the number of edge insertions and edge deletions performed.

In our implementation, we have used `RZ-Opt` as the decremental structure for the above algorithm, because it has been the fastest algorithm in handling edge deletions in general digraphs. We shall refer to the implementation of this algorithm as `Rod`.

Our experiments revealed that this algorithm spends a significant amount of time in building the adjacency matrix M (M is built from scratch at every update operation). The algorithm must maintain entries $M(v,x) = \min_w M(v,w)$, where v is a vertex, x is a block, and w is a vertex (or a block) belonging to x. The value of each entry $M(v,x)$ in `Rod` is computed using a for loop across all entries. Since these values are non-negative, we can exit the loop as soon as a zero entry is found. We have generated a variant of the algorithm, called `Rod-Opt`, based on this fact to see whether it affects performance.

2.3.3 The Algorithm of Roditty and Zwick in [26]

The very recent algorithm of Roditty and Zwick [26] is a combination of a new persistent dynamic algorithm for maintaining the SCCs of a graph with a new decremental algorithm for maintaining reachability trees presented in [26].

The persistent algorithm for strong connectivity works as follows. During the insertion of an edge (or of a set of edges incident to a vertex), a new version of the graph is created. The algorithm maintains all versions of the graph and each one of them, once created, is not affected by any edge insertion. On the other hand, each edge deletion applies to all versions of the graph. Each SCC of version i of the graph (created by the i-th insert operation) is either a SCC of version $i-1$ or a union of SCCs of version $i-1$. As a result, the SCCs of all versions of the graph can be maintained as a forest, where each SCC is represented by a node and the parent of each SCC is the smallest SCC that contains it. The edge set of the graph is partitioned into $t+1$ edge sets H_i ($i = 1, \ldots, t+1$). If an edge e connects two different SCCs in the current version of the graph, then $e \in H_{t+1}$. Otherwise, $e \in H_j$ where j is the version of the graph at which e became an internal edge of some SCC. When an edge insertion occurs, it suffices to use the edge set H_{t+1} to check whether new SCCs are formed (and to compute them). In order to achieve this, a Union-Find algorithm is used, which can efficiently merge SCCs by representing them as sets of vertices and return the SCC to which a vertex belongs. Using the Union-Find algorithm, an edge set H' from H_{t+1} can be constructed in $O(m\alpha(m,n))$ worst case time ($m = m_0 + m'$), where each endpoint of an edge e corresponds to a SCC, and then compute the SCCs of the graph with edge set H'. Roditty & Zwick [26] use this algorithm in a very clever way in order to maintain a reachability tree in a decremental environment (see [26] for the details) at a total cost of $(m + n \log n)$.

The fully dynamic algorithm for maintaining the transitive closure maintains a pair of reachability trees In_u, Out_u for each vertex $u \in V$. The reachability tree Out_u (resp. In_u) maintains SCCs reachable from (resp. reaching) u. When a set of edges incident to a vertex u is inserted into the graph G, a new version G^u of the graph is created, and the trees In_u, Out_u are built from scratch. Each pair of trees In_u, Out_u is maintained with respect to G^u. Each version G^u undergoes only edge deletions and is replaced by a new version when another edge insertion around vertex u occurs. When an edge deletion occurs, the forest of SCCs is updated using the persistent algorithm for strong connectivity. If a SCC C contained in a reachability tree breaks, then C is replaced by the SCCs to which it breaks, and the algorithm checks whether these SCCs can be connected to the tree. The algorithm handles each update operation in $O(m + n \log n)$ amortized time. A boolean query $(u,\, v)$ is answered in $O(n)$ time by checking for each vertex w whether $u \in In_w$ and $v \in Out_w$. We refer to this algorithm as `RZ-P`.

2.4 The algorithm of Demetrescu and Italiano

The main idea of the algorithm of Demetrescu and Italiano [10] (see also [11]) is to reduce the transitive closure problem to the problem of maintaining polynomials over matrices subject to updates of their variables. The algorithm takes advantage of the following equivalence: If G is a directed graph and X_G is its adjacency matrix, then computing the kleene closure X_G^* of X_G is equivalent to computing the transitive closure of G.

Let X_b^a denote a Boolean matrix. The basic data structure (we shall refer to it as Struct1) used by the algorithm maintains polynomials P over such matrices of degree 2; i.e., P is of the form $P = \sum_{i=1}^{h} X_1^i \cdot X_2^i$. This structure (after an initialization phase which takes $O(hn^\omega + hn^2)$ time, ω is the exponent of matrix multiplication) is able to maintain P efficiently when a Boolean matrix X_b^a is changed. This is done by maintaining integer matrices $Prod_a, a = 1, \ldots, h$, where each $Prod_a$ maintains a "lazy" count of the number of witnesses of the product $X_1^a \cdot X_2^a$. More specifically, if $X_1^a[x,y] = X_2^a[y,z] = 1$, then y is a witness of pair (x,z), i.e., $Prod_a[x,z] = |y : \{X_1^a[x,y] = X_2^a[y,z] = 1\}|$. The operations supported are *SetRow*, *SetCol*, *LazySet*, *Reset*. Each of these operations updates P after a matrix X_b^a has changed. *SetRow* / *SetCol* updates P when some entries of a specific row / column of X_b^a flip to 1. *Reset* updates P when any entries of X_b^a have flipped to 0. *LazySet* updates P lazily when any entries of X_b^a have flipped to 1. This is done, by updating X_b^a but not P, which could be updated by subsequent *SetRow* / *SetCol* operations. P is maintained correctly under a sequence of these operations, if no *LazySet* operations are performed. If a *LazySet* is performed, then some entries of P which should be set to 1, could remain 0. The operations *SetRow*, *SetCol*, *LazySet* require $O(n^2)$ time (worst case) and *Reset* requires $O(n^2)$ amortized time. If only *Reset* operations are allowed, then the amortized cost of *Reset* is $O(n)$. Struct1 uses $O(hn^2)$ space.

Polynomials P_k of degree $k > 2$ can be maintained by using Struct1, because each P_k can be represented by a sum of $O(k^2)$ polynomials of degree 2. The structure which maintains P_k, supports the operations *SetRow*, *SetCol*, *LazySet*, *Reset*. These operations are similar to those supported by the basic data structure.

If G is a directed graph, X its adjacency matrix, X^* the kleene closure of X, then X^* can be defined as follows (n is the number of nodes of the graph, and also the size of X, X^*). Let A, B, C, D be submatrices (of size $\frac{n}{2} \times \frac{n}{2}$) of X and E, F, G, H be submatrices (of size $\frac{n}{2} \times \frac{n}{2}$) of X^*. Then X^* can be computed recursively by using the following equations [10, 11]:

$$\begin{array}{lll}
P = D^* & E_1 = Q^* & H_2 = R^* \\
Q = A + BP^2C & E_2 = E_1BH_2^2CE_1 & E = E_1 + E_2 \\
F_1 = E_1^2BP & F_2 = E_1BH_2^2 & F = F_1 + F_2 \\
G_1 = PCE_1^2 & G_2 = H_2^2CE_1 & G = G_1 + G_2 \\
H_1 = PCE_1^2BP & R = D + CE_1^2B & H = H_1 + H_2
\end{array}$$

Thus, it suffices to maintain the polynomials $Q, E_2, F_1, F_2, G_1, G_2, H_1, R, E, F, G, H$ and the closure matrices P, E_1, H_2 of size $\frac{n}{2} \times \frac{n}{2}$. Each such closure matrix of size $\frac{n}{2} \times \frac{n}{2}$ is maintained recursively by 12 polynomials and 3 closures of size $\frac{n}{4} \times \frac{n}{4}$, and so on. When an edge insertion or deletion occurs, the transitive closure information is updated by properly updating the 12 polynomials and the 3 matrix closures of size $\frac{n}{2} \times \frac{n}{2}$ (each matrix closure is updated recursively). In this way the algorithm can handle insertion of a set of edges around a vertex u (such an insertion is called a $u-$centered insertion) and deletion of an arbitrary set of edges. Each update operation requires $O(n^2)$ amortized time. However, if only edge deletions are performed, then each operation requires $O(n)$ amortized time. Boolean queries can be answered in $O(1)$ time.

In our implementation, which we refer to as `DI`, we have not used matrix multiplication, however this affects only the initialization time of the algorithm, because in update operations matrix multiplication is not used.

2.5 Simple-minded Algorithms

Frigioni et al [13] developed three simple-minded algorithms for maintaining the transitive closure which are based on graph-searching algorithms. These simple algorithms maintain no information about the transitive closure. When an edge insertion or edge deletion occurs, then the particular edge is simply added or removed from G, resulting in a $O(1)$ time update operation. Queries are answered in $O(n+m)$ worst-case time by applying some graph-searching algorithm starting from the source vertex and terminating the algorithm as soon as the the target vertex is found or the graph is exhausted. The graph-searching algorithms used were `BFS`, `DFS`, and `DBFS` (vertices are visited in DFS order, but every time a vertex is visited we check whether the target vertex is any of its adjacent ones).

2.6 Summary

The theoretical time and space bounds of all algorithms and their variants considered in our study are summarized in Fig. 1. Recall that $m = m_0 + m'$.

Algorithm	**Reference**	**Amortized Update Time**		**Query Time**	**Space**
`Ital-Gen` (†)	[13]	$O(n)$ per insertion	$O(m)$ per deletion	$O(1)$	$O(n^2)$
`RZ-Opt` (†)	This paper	$O(m)$ per insertion	$O(n)$ per deletion	$O(1)$	$O(n^2)$
`RZ-1`	[25]	$O(m\sqrt{n})$		$O(\sqrt{n})$	$O(n^2)$
`Rod`	[24]	$O(n^2)$		$O(1)$	$O(n^2)$
`Rod-Opt`	This paper	$O(n^2)$		$O(1)$	$O(n^2)$
`RZ-P`	[26]	$O(m + n\log n)$		$O(n)$	$O(nm)$
`King-1`	[18]	$O(n^2 \log n)$		$O(1)$	$O(n^2 \log n)$
`King-2`	This paper	$O(n^2 \log n)$		$O(1)$	$O(n^2 \log n)$
`King-3`	This paper	$O(n^2 D)$		$O(1)$	$O(n^2)$
`DI`	[10]	$O(n^2)$		$O(1)$	$O(n^2)$
`Simple`	[13]	$O(1)$		$O(n+m)$	$O(n+m)$

Figure 1: Algorithms considered. Amortized bounds for $m' = \Theta(m)$ edge insertions/deletions. D denotes the diameter of the graph. (†) Pseudo fully dynamic algorithms.

3 Experimental Results

For our experimental study we used the experimental platform developed by Frigioni et al [13]. We implemented each algorithm as a `C++` class using LEDA [22]. Each class inherits from a common base class for dynamic graph algorithms developed by Alberts et al [4]. We used the correctness checking program developed in [13] and verified the correctness of our implementations. The source code is available from `http://www.ceid.upatras.gr/faculty/zaro/software/`. The experiments were run on three different computing environments; namely, (i) a Sun UltraSparc II (USparc-II) with 4 processors at 300 MHz, Solaris 7 operating system, 1.2GB of main memory, and 2MB L2 cache per processor; (ii) an Intel Pentium 4 (P4) at 1.6 GHz, with linux SUSE 7.3 operating system, 512MB of main memory, and 512KB L2 cache; and (iii) an AMD Athlon at 1.9 GHz, with linux Mandrake 10 operating system, 512MB of main memory, and 256KB L2 cache. We used this variety of computing environments to investigate whether it affects the relative performance of algorithms, especially regarding memory accesses and cache effects since all algorithms require $\Omega(n^2)$ space.

In all experiments conducted we did not observed any substantial difference in the relative performance of the implementations (the experiments on USparc-II were run on a single processor). The same applies for the simulation of cache misses with Valgrind [27]. For that reason we will mostly report experiments run on P4. We only mention that `RZ-P` (as expected) was by far the most memory demanding algorithm, and after a certain point its performance is dominated by the swaps executed between main and secondary memory. Due to this fact, we were practically unable to run large input instances (e.g., graphs with more than 800 vertices) on P4 and Athlon.

We performed experiments on three classes of inputs: (a) random inputs, involving random sequences of update and query operations performed on random digraphs; (b) synthetic inputs that are worst-case inputs for the fully dynamic algorithms involving specific sequences of bad update patterns; and (c) inputs motivated by real-world graphs.

In the following and for the case of simple-minded algorithms, we report results only with the fastest of them in the particular class of inputs.

3.1 Random Inputs

We performed our tests on random digraphs with $n \in [100, 700]$ vertices and several values on the initial number of edges m_0. For these values of n and m_0, we considered various lengths of operation sequences $|\sigma| \in [500, 50000]$. We generated a large collection of data sets, each consisting of 5 to 10 samples, and corresponded to a fixed value of graph parameters and $|\sigma|$. The reported values are average CPU times over the samples. The random sequence of operations consisted of update operations (insertions/deletions) and queries (Boolean). As it is customary with similar studies (e.g., [8, 13]), we considered an on-line environment with no prediction of the future and where queries and updates are equally likely. In particular, we considered two types of patterns: uniformly mixed queries and updates (each occurring with probability 1/2, where an update can equally likely be an insertion or deletion), and uniformly mixed insertions, deletions, and queries (each such operation occurs with probability 1/3). Since we are dealing with random graphs, it is important to recall some of their structural properties which depend on the edge density [6]. If a random (di)graph has more than $n \ln n$ edges, then it is with high probability (w.h.p) (strongly) connected. If its number of edges is below $n \ln n$ and above n, then the graph has a giant component of size $\Theta(n)$ and several small components the largest of which has size $O(\ln n)$. When the number of edges is about n, then the giant component has size $\Theta(n^{2/3})$, while when the number of edges drops below n, then the largest component has size $O(\ln n)$. Moreover, the diameter of a random (di)graph ranges w.h.p. from constant (dense graphs) to $O(\log n)$ (sparse graphs) [6, 23].

Our experiments revealed that Demetrescu and Italiano's algorithm, King's algorithm and its variants were by far the slowest, followed by `Rod` and `Rod-Opt`, even for small input instances and moderate operation sequences with 50% of queries (which is in favour of these algorithms, as a query is a $O(1)$ time operation). Fig. 2 demonstrates precisely this behaviour. The other algorithms are put there only to give a flavor of comparison. Their precise performance will be discussed later with the help of Figures 3 and 4.

The bad behaviour of `King-1`, `King-2`, `King-3`, `Rod` and `Rod-Opt` can be explained by the fact that they maintain incidence matrices. This slows down the construction and the update of the trees maintained, since they require quadratic time regardless of the edge density. The cost of traversing the outgoing edges of a vertex v requires now $O(n)$, instead of $O(\text{out-degree}(v))$, time and as a result the construction of a depth 2 tree, with $O(n)$ nodes at depth 1, requires $O(n^2)$ time, because one must traverse the outgoing edges of $O(n)$ nodes. In addition, these algorithms maintain a matrix *count*, such that $count(x, y)$ equals the number of insertion centers that lie on a

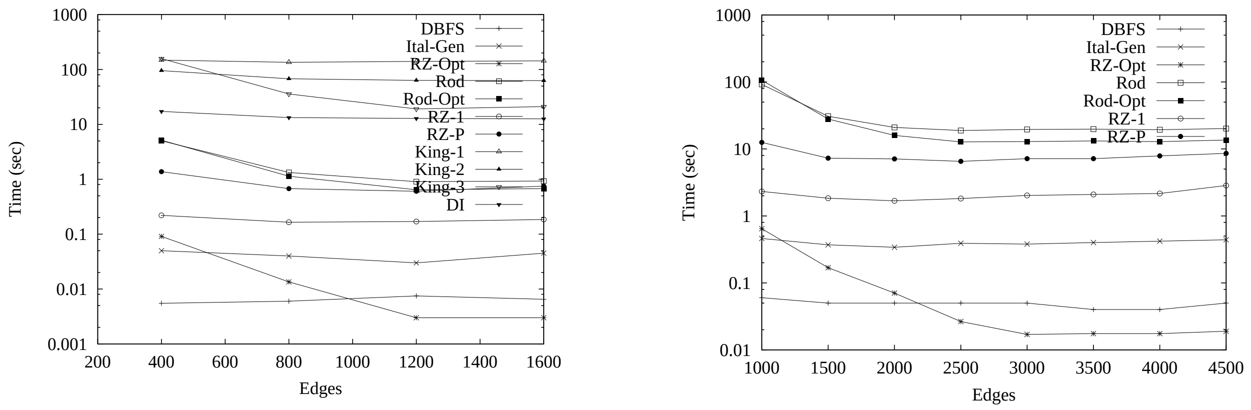


Figure 2: Random digraphs. Experiments on P4. Left: $n = 150$, $|\sigma| = 1000$ (50% queries), all algorithms. Right: $n = 300$, $|\sigma| = 5000$ (50% queries), DI and King's algorithms are excluded. Time is shown in logarithmic scale.

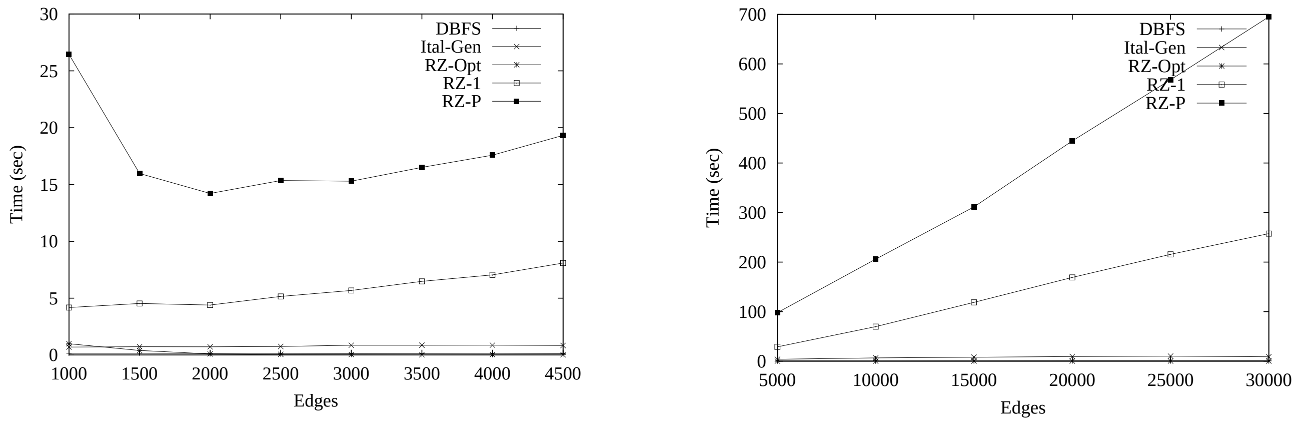


Figure 3: Random digraphs with $n = 300$. Experiments run on P4. Left: $|\sigma| = 5000$ (33% queries). Right: $|\sigma| = 30000$ (33% queries).

path from x to y (`King-1` and `King-2` maintain more than one such matrices). The maintenance of these matrices turns out to be costly. Among the variants of King's algorithm, we observe that `King-3` is almost always the fastest. This could be explained by the fact that the number of trees it maintains is smaller than those of `King-1` and `King-2`. When the graph becomes very sparse, however, `King-3` is slower than the other two due to the overhead of maintaining a BFS tree for the large component. The right graphic of Fig. 2, in which `DI` and King's algorithms are excluded, shows the difference between `Rod` and `Rod-Opt`. The latter is from 1.5 (50% queries) to 2 (33% queries) times faster than the original due to the heuristic of aborting the loop as soon as a zero $M(v, w)$ entry has been found.

We now turn to `DI`. One possible explanation for the bad performance of `DI` is that it maintains a large number of polynomials of degree $k > 2$. Each such polynomial is maintained by $O(k^2)$ polynomials of degree 2 and the update of such a degree 2 polynomial requires $O(n^2)$ time. Moreover, the algorithm maintains 3 closure matrices of size $\frac{n}{2} \times \frac{n}{2}$, each of which is maintained recursively with 12 polynomials and 3 closure matrices of size $\frac{n}{4} \times \frac{n}{4}$, and so on. However, this recursive maintenance turns out to be inefficient, because `DI` becomes slower as the recursion depth increases. Even in the case where no recursion has been used, as in Fig. 2, `DI` was also slow. On the other hand, `DI` (as expected) is faster than King's algorithm and its variants, because it manages to exhibit a better locality of reference. Indeed, simulation of cache misses with Valgrind revealed that the cache behaviour of `DI` is dramatically (about 50–100 times) better than King's algorithm and its variants. Actually, `DI` has the *smallest* ratio of cache misses w.r.t. *any* algorithm in our

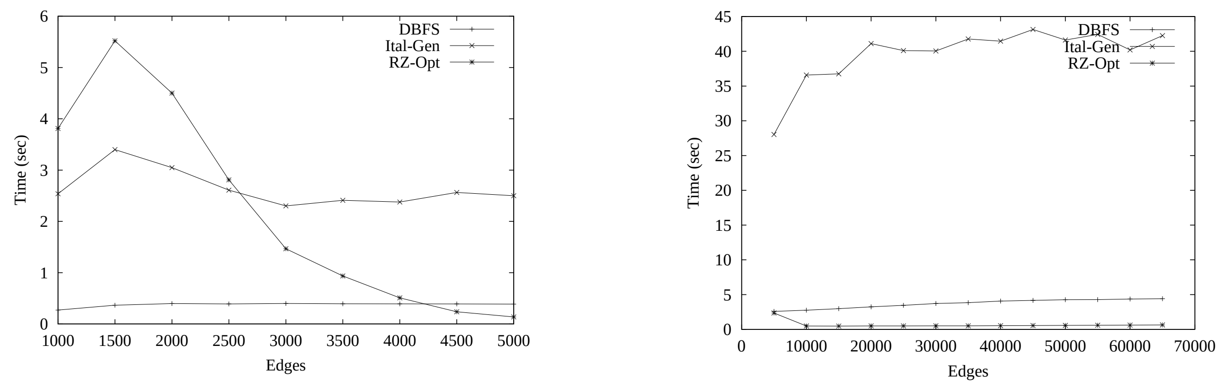


Figure 4: Random digraphs with $n = 700$. Experiments run on P4. Left: $|\sigma| = 5000$ (33% queries). Right: $|\sigma| = 50000$ (33% queries).

study (even w.r.t. the simple-minded ones). However, its performance degrades, since it requires a vast number of main memory accesses to maintain the matrices.

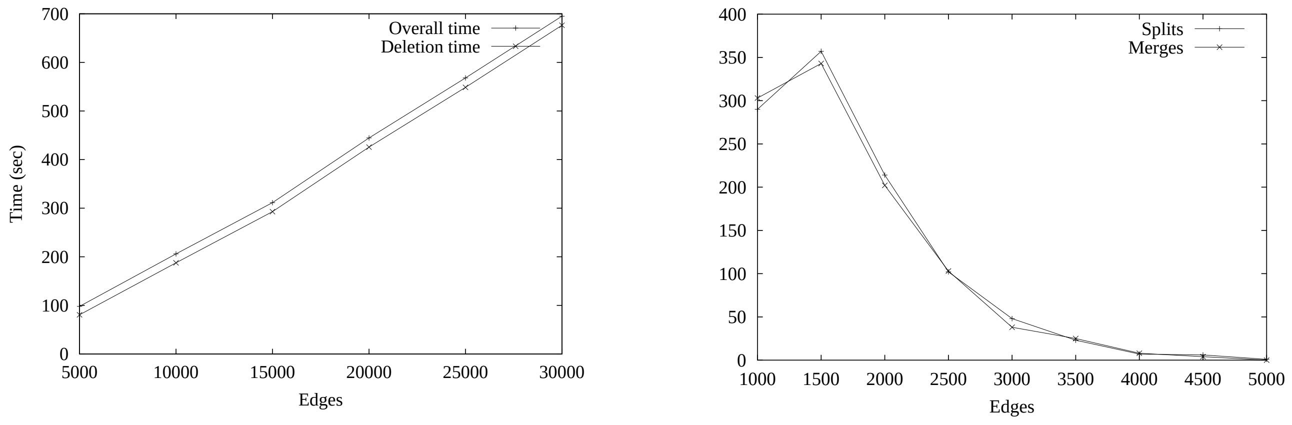
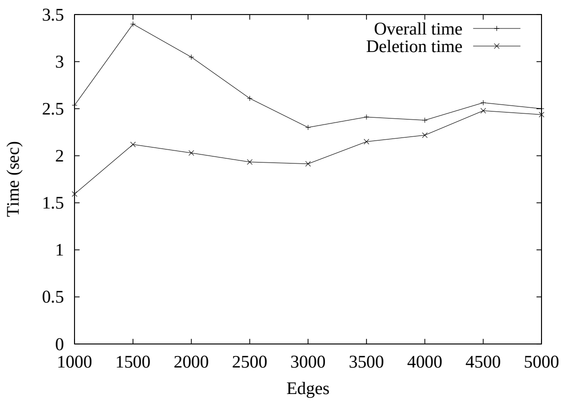


Figure 5: Experiments with random digraphs on P4. Left: deletion time vs overall time of `RZ-P` for $n = 300$ and $|\sigma| = 30000$ (33% queries). Right: splits and merges of SCCs in `RZ-Opt` and `Ital-Gen` for $n = 700$ and $|\sigma| = 5000$ (33% queries).

The comparison of the rest of the algorithms is shown better in the experiments reported in Fig. 3. Algorithms `DBFS`, `Ital-Gen`, and `RZ-Opt` clearly outperform `RZ-1` and `RZ-P`. These two latter algorithms, although faster than those of King and Roditty, have execution times that are significantly larger than those of the simple-minded or pseudo fully dynamic algorithms. `RZ-P` is penalized by its larger query time, by its large memory demand after a certain point (see the right graphic of Fig. 3), and most importantly by the cost of maintaining the forest of SCCs across all versions of the graph, which is rather expensive during deletions. Fig. 5(left) demonstrates precisely this latter fact, as `RZ-P` spends almost all of its time in handling edge deletions. `RZ-1` is faster than `RZ-P` due to its smaller query time and the fact that it uses `RZ-Opt` to handle edge deletions. Its main drawback, however w.r.t. `DBFS`, `Ital-Gen`, and `RZ-Opt`, seems to be the fact that its decremental data structure (i.e., `RZ-Opt`) must be rebuilt from scratch following each sequence of a relatively small ($\sqrt{n}$) number of operations.

We now turn to the three faster implementations `DBFS`, `Ital-Gen`, and `RZ-Opt`. Fig. 4 illustrates their performance. Similar results were reported for other sizes of the operation sequence and different percentage of queries. We observe that when the graph is relatively sparse (less than $n \ln n$ edges), `DBFS` is the fastest algorithm. For denser graphs with more than $n \ln n$ edges, `RZ-Opt` is considerably faster, since in this case the digraph is almost surely strongly connected. The



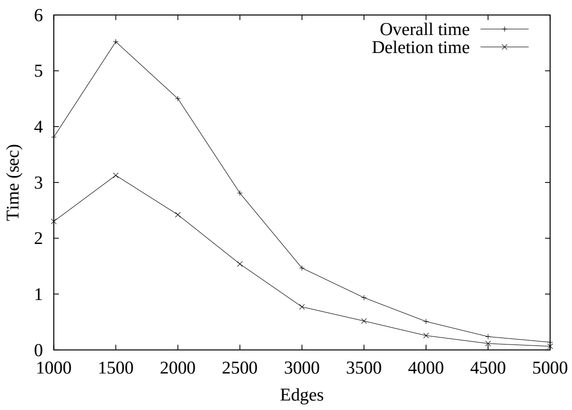
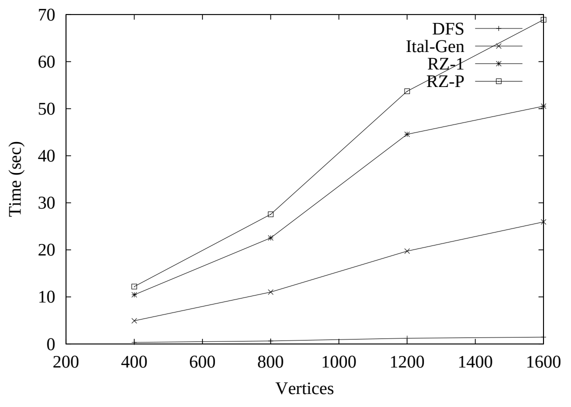


Figure 6: Random digraphs with $n = 700$ and $|\sigma| = 5000$ (33% queries). Experiments on P4. Deletion time vs overall time (their difference is practically the insertion time, since queries take negligible time). Left: `Ital-Gen`. Right: `RZ-Opt`.

differences between the performance of `Ital-Gen` and `RZ-Opt` in this case can be explained by how they handle edge insertions and deletions in SCCs. `Ital-Gen` is not efficient in handling edge deletions because if an edge is removed from a SCC, then it may spend $O(n + m)$ time to determine whether that SCC has broken. Moreover, even if the SCC does not break, it may still need to rebuild the sparse certificate of the SCC, and this is independent of the edge density of the graph, a fact that consequently applies to the total edge deletion time, as Fig. 6(left) illustrates. This claim is also confirmed with the support of Fig. 5(right): although the number of SCCs that split decreases, the total deletion time is practically unaffected. On the other hand, `RZ-Opt` is not efficient in handling edge insertions because if a new edge is created in a SCC, then the algorithm may spend $O(n + m)$ time to update the BFS trees it maintains. However, this slows down the performance of the algorithm only in the case where the graph is sparse. When the edge density increases, `RZ-Opt` performs better, since the BFS trees have small depth due to the fact that the diameter of the graph decreases. This is precisely reported in Fig. 6(right), where initially (sparse edge density) the time for edge insertion is a fair portion of the overall time; however, as the edge density increases, insertion time gradually decreases, and the same applies to the overall time. The above suggest that `RZ-Opt` is highly dominated by the merges and splits of SCCs, a fact that can be easily confirmed by an inspection of the curves of Fig. 5(right) with the overall time curve of Fig. 6(right). Consequently, in sparse graphs, where many splits and merges of SCCs occur, both algorithms have more-or-less the same performance, since they have to frequently re-initialize the data structures maintaining the SCCs (either for insertion or deletion purposes). As soon as the strong connectivity threshold ($n \ln n$) is approached and/or surpassed `RZ-Opt` outperforms `Ital-Gen`, as it performs much less work mainly for deletions.

3.2 Synthetic Inputs

We have considered and slightly modified the specific structured inputs introduced by Frigioni et al [13], which enforce the dynamic algorithms to exhibit their worst-case behaviour. These graphs consist of a sequence of $s = \lceil n/k \rceil$ cliques $C_1, \ldots, C_s$, each of size k, interconnected with a set of "bridges". A bridge is a pair of directed edges connecting a node of C_i with a node of C_{i+1}, and vice versa. Insertions and deletions are only performed on bridges and in a specific order. In the case of edge insertions, first the bridge pair between C_1 and C_2 is inserted (one edge of the pair at a time), the second bridge between C_{s-1} and C_s, the third between C_2 and C_3, and so on. Hence, the bridge inserted last will provide new reachability and SCC information from roughly $n/2$ to the other $n/2$ vertices of the graph. The reverse order is followed in the case of edge deletions. The



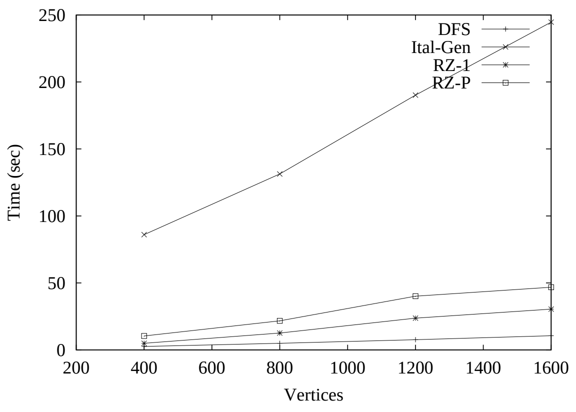


Figure 7: Synthetic digraphs with $|\sigma| = 1920$ (33% queries). Experiments run on USparc-II. Left: clique size 10. Right: clique size 80.

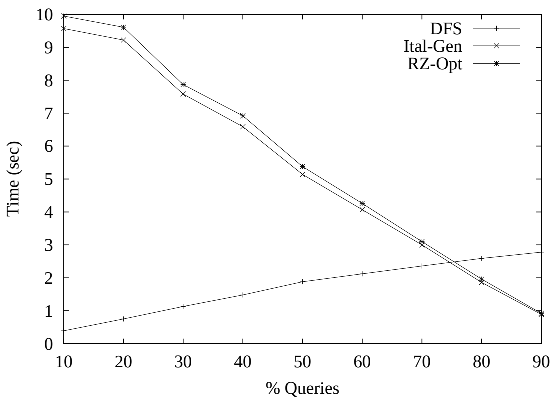
fully dynamic sequence consists of alternating subsequences of $2s-2$ insertions and $2s-2$ deletions intermixed evenly with queries.

As with random inputs, `DI`, King's algorithm and its variants as well as `Rod` and `Rod-Opt` were the worst, and hence we do not report results for these algorithms. From the pseudo fully dynamic, `Ital-Gen` was always faster than `RZ-Opt` (due to the inefficient insertion procedure of the latter; see below), and hence we report results only with the former. Fig. 7 illustrates the performance of the rest of the algorithms considered. Similar results hold for smaller or larger operation sequences and different computing environments; we report results on USparc-II (the machine with the largest memory) to include large values of n. We observed that `DFS` was always the fastest algorithm.

The performance of `Ital-Gen` deteriorates as the clique size k increases, since for large k we get large SCCs whose maintenance becomes very costly due to their splits and merges. Note that a split (resp. merge) of a SCC occurs every two edge deletions (resp. insertions), and the algorithm must build the data structures in those SCCs from scratch. On the other hand, `RZ-1` and `RZ-P` perform better than `Ital-Gen` as the value of k increases, since they can handle better the splits and merges of large SCCs. As a side remark, the good performance of `RZ-1` indicates that `RZ-Opt` (which is used by `RZ-1` for deleting edges) is worse than `Ital-Gen` mainly due to the inefficient handling of edge insertions. For `RZ-P` a large value of k implies a small number of insertion centers (tails of edges inserted), and consequently a small number of versions of the graph that the algorithm must maintain. In addition, the maintenance of the reachability trees has a very low cost, since the algorithm has to check only the external to a SCC edges, i.e., the bridges. However, `RZ-P` is slower than `RZ-1` probably due to the overhead caused by deletions, in order to maintain the forest of SCCs across all versions of the graph. In conclusion, the fully dynamic algorithms demonstrate their theoretical superiority by learning quickly the specific structure of the synthetic graphs and benefiting substantially from it.

3.3 Real-world Inputs

Apart from random and synthetic inputs, we have run the algorithms on inputs motivated by real-world graphs. The first graph we have used describes the connections and policy strategies among the autonomous systems of a fragment of the Internet visible from RIPE (`www.ripe.net`) [5], one of the main European servers. The graph has 1259 vertices and 5101 edges, and has been also used in [13], where (for the purpose of their study) it has been converted to a DAG by changing the direction of a few edges. The second graph describes a US road network (`ftp://edcftp.cr.usgs.gov`). Such graphs have been used in [8]. This specific graph has 576 vertices and 1762 edges. On these graphs, we run random sequences of operations, similar to those used for random digraphs.



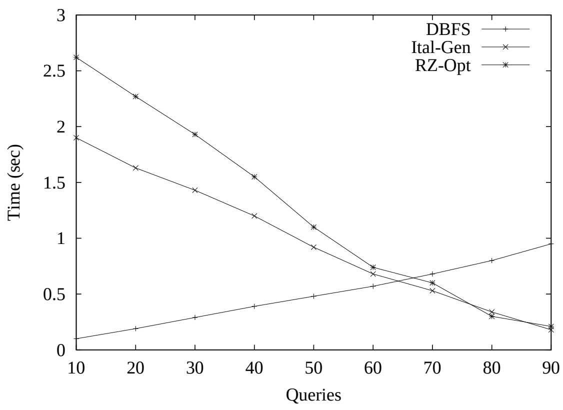


Figure 8: Experiments on P4. Left: RIPE fragment of Internet, $|\sigma| = 15000$. Right: US road network, $|\sigma| = 5000$.

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. In addition, we performed experiments with different percentage of queries in the operation sequence (from 10% to 90%). These experiments may give useful suggestions on how to proceed if one knows in advance the update-query pattern. Fig. 8 illustrates the performance of the fastest algorithms for fully dynamic sequences of 15000 (left) and 5000 (right) operations. Since these graphs are relatively sparse, it needs more than 65% of queries in order to beat the simple algorithms. The performance of `RZ-Opt` and `Ital-Gen` are almost identical. This is due to the fact that: the first graph is a DAG and both implementations perform almost identical tasks, and the second graph is sparse therefore the two implementations have almost the same performance (as explained in section 3.1).

3.4 Partially Dynamic Inputs

Although the focus of the paper is on fully-dynamic algorithms, and thus on fully dynamic sequences of operations, we have also conducted experiments on partially dynamic sequences. For random digraphs and in the case of decremental sequences, `RZ-Opt` is the fastest algorithm except for the case where the initial graph is sparse (in this case `DBFS` is faster). Fig. 9 illustrates this fact. In the case of incremental sequences `Ital-Gen` is the fastest, followed closely by `RZ-Opt`. Fig. 10 illustrates this fact. The fully dynamic algorithms in both decremental and incremental sequences are slower than any of `DBFS`, `Ital-Gen`, and `RZ-Opt`.

In the case of synthetic digraphs, `DFS` is the fastest algorithm. Among the rest, `Ital-Gen` and `RZ-Opt` are the best. Fig. 11 illustrates this fact. Finally, in the case of real-world digraphs, the results are similar to those of random digraphs. This very good performance of pseudo dynamic algorithms is a result of the fact that they are designed to handle efficiently only partial dynamic sequences.

4 Conclusions

We have implemented some recent fully dynamic algorithms along with several variants of them for maintaining the transitive closure in a digraph, and compared them experimentally with pseudo fully dynamic and simple-minded algorithms. Our experimental study shows that fully dynamic algorithms perform very well on structured inputs, although they cannot beat the simple ones. In unstructured inputs, the pseudo fully dynamic algorithms are much better.

References

[1] S. Abdeddaim. Algorithms and Experiments on Transitive Closure, Path Cover and Multiple Sequence Alignment. In *Proc. 2nd Workshop on Algorithm Engineering and Experiments* – ALENEX 2000, pp. 157–169, 2000.

[2] D. Alberts, G. Cattaneo, and G. F. Italiano. An Empirical Study of Dynamic Graph Algorithms. *ACM Journal of Experimental Algorithmics*, 2(5), 1997. Preliminary version in Proc. SODA'96.

[3] G. Amato, G. Cattaneo, and G. F. Italiano. Experimental Analysis of Dynamic Minimum Spanning Tree Algorithms. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms* – SODA'97, pp. 314–323, 1997.

[4] D. Alberts, G. Cattaneo, G.F. Italiano, U. Nanni, and C. Zaroliagis. A Software Library of Dynamic Graph Algorithms. In *Proc. Workshop on Algorithms and Experiments* – ALEX'98, pp. 129–136, 1998.

[5] 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.

[6] B. Bollobas. Random Graphs. Academic Press, New york, 1985.

[7] G. Cattaneo, P. Faruolo, U. Ferraro-Petrillo, and G.F. Italiano. Maintaining Dynamic Minimum Spanning Trees: An Experimental Study. In *Proc. 4th Workshop on Algorithm Engineering and Experiments* – ALENEX 2002.

[8] C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms* – SODA 2004, pp.362-371.

[9] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U.Nanni. Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study. In *Proc. 4th Workshop on Algorithm Engineering* – WAE 2000, *Lecture Notes in Computer Science*, Vol. 1982 (Springer-Verlag 2000), pp.218-229.

[10] C. Demetrescu and G. F. Italiano. Fully Dynamic Transitive Closure: Breaking through the $O(n^2)$ Barrier. In *Proc. 41st IEEE Symp. on Foundations of Computer Science* – FOCS 2000, pp. 381–389, 2000.

[11] C. Demetrescu Fully Dynamic Algorithms for Path Problems on Directed Graphs. PhD thesis, Department of Computer and Systems Science, University of Rome "La Sapienza", February 2001.

[12] 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.

[13] D. Frigioni, T. Miller, U. Nanni and C. Zaroliagis. An Experimental Study of Dynamic Algorithms for Transitive Closure. *ACM Journal of Experimental Algorithmics*, 6(9), 2001.

[14] M.R. Henzinger and V. King. Fully Dynamic Biconnectivity and Transitive Closure. In *Proc. 36th IEEE Symposium on Foundations of Computer Science* – FOCS'95, pp. 664–672, 1995.

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

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

[17] R. Iyer, D. Karger, H. Rahul, and M. Thorup. An Experimental Study of Poly-Logarithmic Fully-Dynamic Connectivity Algorithms. In *Proc. 2nd Workshop on Algorithm Engineering and Experiments* – ALENEX 2000, pp. 59–78, 2000.

[18] V. King. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *Proc. 40th IEEE Symposium on Foundations of Computer Science* – FOCS'99, pp.81-91, 1999.

[19] V. King, and G. Sagert. A Fully Dynamic Algorithm for Maintaining the Transitive Closure. In *Proc. 31st ACM Symposium on Theory of Computing* – STOC'99, pp. 492–498, 1999.

[20] V. King and M. Thorup. A Space Saving Trick for Directed Dynamic Transitive Closure and Shortest Path Algorithms. In *Proc. 7th Comp. and Combinatorics Conference* – COCOON 2001, pp.268-277, 2001.

[21] I. Krommudas and C. Zaroliagis. An Experimental Study of Algorithms for Fully Dynamic Transitive Closure". In *Algorithms* - ESA 2005, Lecture Notes in Computer Science Vol. 3669 (Springer-Verlag, 2005), pp. 544-555.

[22] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[23] J. Reif and P. Spirakis. Expected Parallel Time and Sequential Space Complexity of Graph and Digraph Problems. *Algorithmica* 7:597-630, 1992.

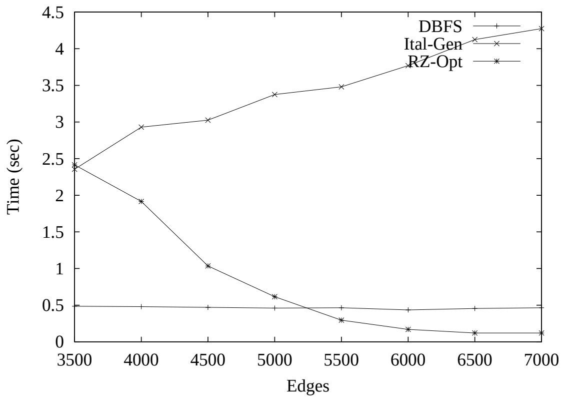
[24] L. Roditty. A faster and simpler fully dynamic transitive closure. *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms* – SODA 2003, pp. 404-412.

[25] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proc. 43rd IEEE Symposium on Foundations of Computer Science* – FOCS 2002, pp.679-690, 2002.

[26] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proc. 36th ACM Symp. on Theory of Computing* – STOC 2004.

[27] Valgrind. `http://valgrind.kde.org/`.

[28] C. Zaroliagis. Implementations and Experimental Studies of Dynamic Graph Algorithms. Chapter 11, in *Experimental Algorithmics* (Eds. R. Fleischer, B. Moret, and E. Meineche-Schmid), Springer-Verlag, 2002, pp. 229-278.



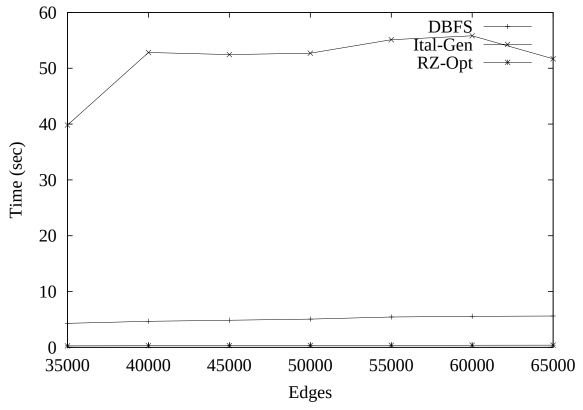
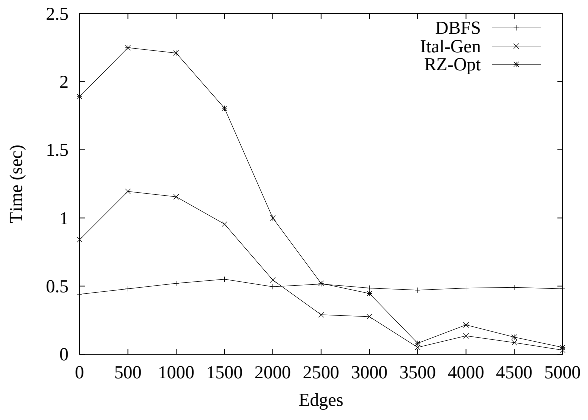


Figure 9: Deletions on random digraphs with $n = 700$. Experiments run on P4. Left: $|\sigma| = 5000$ (50% queries). Right: $|\sigma| = 50000$ (50% queries).



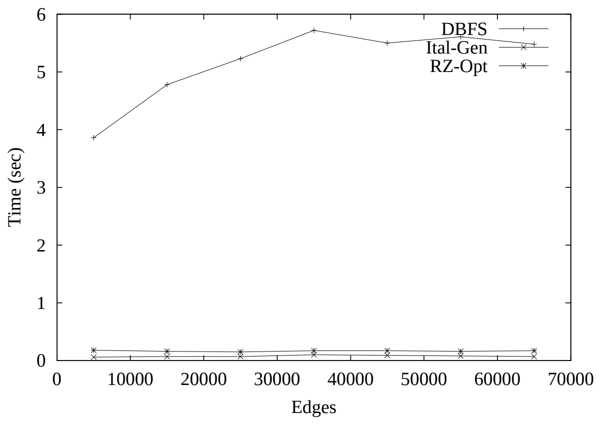
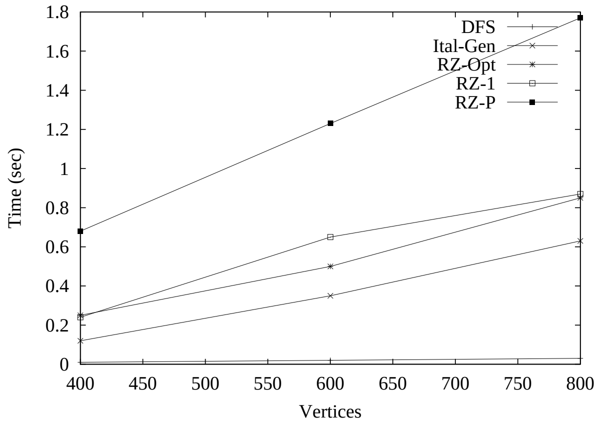


Figure 10: Insertions on random digraphs with $n = 700$. Experiments run on P4. Left: $|\sigma| = 5000$ (50% queries). Right: $|\sigma| = 50000$ (50% queries).



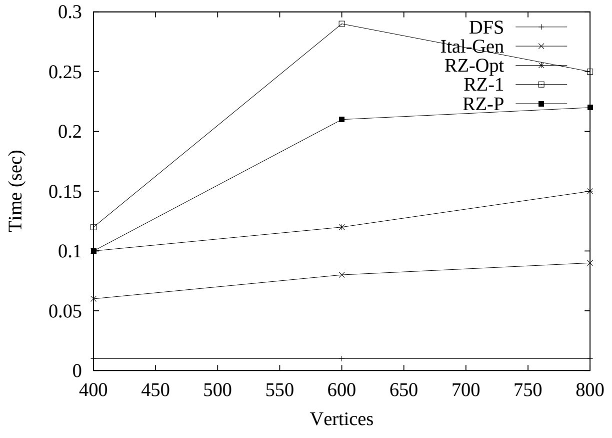


Figure 11: Synthetic digraphs with $|\sigma| = 160$ (50% queries), clique size 10. Experiments run on P4. Left: Deletions. Right: Insertions.