An Experimental Study of Algorithms for Fully Dynamic Transitive Closure*

Ioannis Krommias and Christos Zaroliagis

Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece; and Dept of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece.
{krommias,zero}@ceid.upatras.gr

Abstract. We have conducted an extensive experimental study on some recent, theoretically outstanding, algorithms for fully dynamic transitive closure along with several variants of them, and compared them to pseudo fully dynamic and simple-minded algorithms developed in a previous study. 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 fully 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 this (fundamental and extensively studied) problem, namely with the maintenance of transitive closure when $G$ undergoes a sequence of edge insertions and deletions. 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 algorithm is called incremental, while in the latter decremental.

Recently, we have witnessed a number of important theoretical breakthroughs regarding fully dynamic transitive closure [4, 11, 10, 12, 15–17]. These fully dynamic algorithms can be roughly divided into two categories: those using combinatorial techniques [10, 12, 15–17] and those which do not exclusively use such techniques [4, 11, 16]. Moreover, some of these algorithms apply to DAGs, while some others to general digraphs. In this paper, we concentrate on fully dynamic algorithms for maintaining the transitive closure of general digraphs only.

Despite the above theoretical progress, we are not aware of any practical assessment of any of the aforementioned algorithms. Our prime goal is to advance our knowledge on the practical aspects of this recent and important theoretical work. Previous experimental studies regarding maintenance of transitive closure [1, 6] have mostly focussed on the assessment of partially dynamic algorithms. The only experimental comparison regarding fully dynamic transitive closure was

* This work was partially supported by the IST Programme (6th FP) of EC under contract No. IST-2002-001907 (integrated project DELIS).
made by Frigioni et al. [6], where a fully dynamic algorithm in [7] was compared to a new algorithm, called Ital-Gen, developed in [6]. Ital-Gen is based on a hybridization and extension of Italiano’s partially dynamic algorithms [8,9], 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 in [6] showed that Ital-Gen was considerably faster than the fully dynamic algorithm in [7].

In this work, we have implemented and experimentally compared all the aforementioned combinatorially based fully dynamic algorithms [10,12,15–17], as well as the algorithm of Demetrescu & Italiano [4], along with some new variants of them. In particular, from the former set we have implemented the space-saving version of King’s algorithm [10,12] along with two new variants, the algorithm of Roditty & Zwick [16], the algorithm of Roditty [15] along with a new variant, and the very recent algorithm of Roditty & Zwick [17]. In addition, we have implemented the decremental algorithm of Roditty & Zwick [16], 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. A summary of the theoretical bounds of all these algorithms can be found in Fig. 1. We compared the above implementations to Ital-Gen and also to the simple-minded algorithms presented in [6]. Our experiments were conducted on three types of inputs: random inputs, synthetic (worst-case) inputs, and real-world inputs.

Our experiments showed that, regardless of the type and size of input, the algorithm of Demetrescu & Italiano [4], the space-saving version of King’s algorithm [10,12] and its new variants were by far the slowest, followed by the algorithm of Roditty [15]. The performance of the latter is actually surprising, since at least for some cases it is theoretically better than the algorithm in [16]. 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 [17] – algorithm of Roditty & Zwick in [16] was the fastest. The second interesting outcome is that Demetrescu & Italiano’s [4] 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). 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 [16,17] 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. Due to space limitations, some parts are omitted from this version. Full details can be found in [13].
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 $v$, and $v$ 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 following, we 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 [8, 9]. The incremental algorithm applies to any digraph, while the decremental applies to DAGs. In [6], a modified and fine-tuned implementation of these algorithms was presented, called Ital-0pt. Ital-0pt maintains for each $u \in V$ a tree $Desc[u]$, which contains all descendants of $u$. The $Desc$ trees are maintained implicitly using a $n \times n$ matrix $Parent$ and a Boolean query for vertices $i$ and $j$ is carried out in $O(1)$ time, by checking $Parent[i, j]$. The main idea of the algorithm is that the maintenance of a tree during a sequence of edge deletions can be done efficiently, if the graph is a DAG. More details in [6, 8, 9].

Based on Ital-0pt, Frigioni et al. [6] developed a new algorithm called Ital-Gen that can handle edge insertions and deletions in general digraphs. The main idea of Ital-Gen is that if every strongly connected component (SCC) is replaced by a single vertex, then the resulting graph $G'$ is a DAG, whose transitive closure can be maintained using Ital-0pt. For each SCC $C$, the algorithm maintains (among other) a sparse certificate $S$ (sparse subgraph of $C$. If an edge $e$ is removed from $C$ and if $e$ belongs to $S$, then Ital-Gen checks whether $C$ has broken. 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. A query can be answered in $O(1)$ time by checking this matrix. Both Ital-0pt and Ital-Gen can be modified so that they can work in a fully dynamic environment [6].

2.2 The Algorithm of King and its Variants

King's algorithm [10] uses forests of BFS trees and a set of matrices to facilitate query answering in $O(1)$ time. 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 $In_u$ and $Out_u$ of depth $d = 2^i$ rooted at every vertex $u \in V$. The BFS trees of the forest $F_1$ are constructed for $G$. 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: if there is a path between two vertices $u, w$ whose length is less than or equal to $2^i$, then $(u, w) \in E^i$. 
During the deletion of an edge $e$ (or of a set of edges) the edge $e$ is removed from any BFS tree of forest $F^i$, $1 \leq i \leq k$, it belongs. In the case of an insertion of an edge (or a set of edges) incident to a vertex $u$ the trees $In_u^i$, $Out_u^i$, $i = 1, \ldots, k$ are built from scratch.

King and Thorup in [12] proposed a space saving version of this algorithm. Graph $G^i = (V, E^i)$ is maintained using an incidence matrix $M$: if $(u, w) \in E^i$, then $M(u, w) = 1$, otherwise $M(u, w) = 0$. The maintenance of a BFS tree using these matrices costs now $O(n^2d)$ time, however this 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 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

The Algorithms in [16] and their Extensions. Roditty and Zwick proposed in [16] a randomized (Las Vegas) decremental algorithm, which is a combination of the decremental part of Itai-Gen [6] with a new decremental algorithm [16] for maintaining the SCCs of a digraph. The crucial observation is that Itai-Gen requires $O(nm_0)$ time to handle any sequence of edge deletions, if the check whether a SCC has broken is handled by the new decremental algorithm [16] for maintaining the SCCs. The main idea of the decremental algorithm which maintains the SCCs is the following. In each SCC $C$ of the graph it maintains an In-BFS tree $In(w)$ and an Out-BFS tree $Out(w)$ rooted at a random vertex $w \in C$. If an edge $(x, y)$ belonging to $C$ is deleted, then the BFS trees are updated accordingly and, to determine whether $C$ breaks it suffices to check whether $x \in In(w)$ and $y \in Out(w)$. 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 $w$.

We have modified the decremental transitive closure algorithm of Roditty and Zwick so that it can handle edge insertions, without affecting the performance 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 Itai-Gen is used. On the other hand, if the new edge belongs to a SCC, then its BFS trees are updated in a recursive manner by lifting up nodes towards the root of the tree. We call this pseudo fully dynamic algorithm RZ-Opt. This algorithm handles any sequence of edge deletions in $O(nm)$ time and $m'$ edge insertions in $O(m'(n + m_0 + m'))$ time. Consequently, we expect RZ-Opt to be faster when handling edge deletions, and Itai-Gen to be faster when handling edge insertions.

A final remark concerns the algorithm for maintaining the SCCs. When a SCC $C$ breaks, then the BFS trees rooted at $w \in C$ are inherited by the new
SCC that contains \( w \). 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 \( m' \) edge deletions. Despite this fact, however, RZ-Opt proved to be competitive to the fastest algorithms implemented by Frigioni et al. [6].

We now turn to the combinatorial fully dynamic algorithm in [16]. Initially, a decremental data structure DD for maintaining the transitive closure is built (DD could be RZ-Opt or Itai-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 \(|S|\) becomes equal to a predetermined parameter \( t \) \((t = \sqrt{n} in [16] and in our implementation)\), 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 DD. 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 DD 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. Otherwise, there is no \( u-w \) path. We shall refer to the implementation of this algorithm as RZ-1.

The Algorithm of Roditty [15]. The recent fully dynamic algorithm proposed by Roditty [15] is inspired by the algorithm of King [16]. It uses a decremental data structure for maintaining paths composed of 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. The insertion of a set of edges incident to \( u \), may change the blocks with respect to \( u \), while an edge deletion may change every block that exists so far. 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 \( O(n^2) \). Each update of the matrix requires \( O(n^2) \) time. In our implementation, called Rod, we have used RZ-Opt as the decremental structure, because it has been the fastest algorithm in handling edge deletions in general digraphs.

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.
The Algorithm of Roditty and Zwick in [17]. The very recent algorithm of Roditty and Zwick [17] 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 [17].

The persistent algorithm for strong connectivity works as follows. During the insertion of an edge (or a set of edges incident to a vertex), a new version of the graph is created and the algorithm maintains all versions. Each version of the graph, once created, is not affected by any edge insertion, while, each edge deletion applies to all versions. Each SCC of version \( i \) (created by the \( i \)-th insert operation) is either a SCC 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. 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 at which \( e \) became an internal edge of some SCC. When an edge insertion occurs, \( H_{t+1} \) is used to compute new SCCs of the graph. 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. Roditty & Zwick [17] use this algorithm in a very clever way in order to maintain a reachability tree in a decremental environment (see [17] for the details) at a total cost of \( O(m + n \log n) \).

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 \( u \) is inserted into the graph \( G \), a new version \( G^u \) of \( G \) is created, and the trees \( In_u, Out_u \) are built from scratch. Trees \( In_u, Out_u \) are 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 \( u \) 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. 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 [4] (see also [5]) 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 \( X^+_i \) 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, where \( \omega \) is the exponent
of matrix multiplication) is able to maintain \( P \) in \( O(n^2) \) time when a Boolean matrix \( X_k^* \) is changed. Struct1 uses \( O(m^2) \) 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.

Let \( G \) be a directed graph, \( X \) its adjacency matrix, \( X^* \) the Kleene closure of \( X \), and \( n \) the number of nodes of the graph. Then \( X^* \) can be computed recursively by computing 12 polynomials and 3 closure matrices of size \( \frac{n}{2} \times \frac{n}{2} \) \([4, 5]\). Each such closure matrix of size \( \frac{n}{2} \times \frac{n}{2} \) is maintained recursively by 12 polynomials and 3 closures of size \( \frac{n}{2} \times \frac{n}{2} \), 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 \( u \) and deletion of an arbitrary set of edges. 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 and Summary

Frigioni et al \([6]\) developed three simple-minded algorithms based on the following idea. 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 among 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). The theoretical time and space bounds of all algorithms and their variants considered in our study are summarized in Fig. 1.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Reference</th>
<th>Amortized Update Time</th>
<th>Query Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itai-Gem ([1])</td>
<td>([6])</td>
<td>(O(n)) per ins (O(m)) per del</td>
<td>(O(1)) (O(n^2))</td>
<td></td>
</tr>
<tr>
<td>RZ-Opt ([1])</td>
<td>This paper</td>
<td>(O(m)) per ins (O(n)) per del</td>
<td>(O(1)) (O(n^2))</td>
<td></td>
</tr>
<tr>
<td>RZ-1 ([16])</td>
<td>(O(m\sqrt{n}))</td>
<td>(O(n\sqrt{n}))</td>
<td>(O(n)) (O(n^2))</td>
<td></td>
</tr>
<tr>
<td>Rod ([13])</td>
<td>(O(n^2))</td>
<td>(O(1)) (O(n^2))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rod-Opt</td>
<td>This paper</td>
<td>(O(n^2))</td>
<td>(O(1)) (O(n^2))</td>
<td></td>
</tr>
<tr>
<td>RZ-P ([17])</td>
<td>(O(m + n \log n))</td>
<td>(O(n)) (O(nm))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>King-1 ([10, 12])</td>
<td>(O(n^2 \log n))</td>
<td>(O(1)) (O(n^2 \log n))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>King-2</td>
<td>This paper</td>
<td>(O(n^2 \log n))</td>
<td>(O(1)) (O(n^2 \log n))</td>
<td></td>
</tr>
<tr>
<td>King-3</td>
<td>This paper</td>
<td>(O(n^2 D))</td>
<td>(O(1)) (O(n^2))</td>
<td></td>
</tr>
<tr>
<td>DI ([4])</td>
<td>(O(n^2))</td>
<td>(O(1)) (O(n^2))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple ([6])</td>
<td>(O(1))</td>
<td>(O(n + m)) (O(n + m))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Amortized bounds for \( m' = \Theta(m) \) edge insertions/deletions. \( D \) denotes the diameter of the graph and \( m = m_0 + m' \). \((\dag)\) Pseudo fully dynamic algorithms.
3 Experimental Results

For our experimental study we used the experimental platform developed by Frigioni et al [6]. We implemented each algorithm as a C++ class using LEDA [14] and the library for dynamic graph algorithms [2]. We used the correctness checking program in [6] to verify the correctness of our implementations. The source code is available from http://www.csid.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 \( O(n^2) \) space.

In all experiments conducted we did not observe any substantial difference in the relative performance of the implementations. The same applies for the simulation of cache misses with Valgrind (valgrind.kde.org). For that reason we will mostly report experiments run on P4. R2-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 unable to run large input instances (e.g., graphs with more than 500 vertices) on P4 and Athlon. For the case of simple-minded algorithms, we report results only with the fastest of them in the particular class of inputs.

We performed experiments on three classes of inputs.

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). Following similar studies (e.g., [3, 6]), 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).

Our experiments revealed that DI, King-1, King-2 and King-3 were by far the slowest, followed by Rod and Rod-Opt, even for small input instances and moderate operation sequences with 50% of queries. The bad behaviour of King-1, King-2, King-3, Rod and Rod-Opt is due to the maintenance of incidence matrices. This slows down the construction and the update of the trees maintained, since they require quadratic time regardless of the edge density. Among the variants of King’s algorithm, we observe that King-3 is almost always the fastest due to the fact that it maintains less trees than King-1 and King-2. When the graph becomes very sparse, however, King-3 is slower due to the overhead of
maintaining a BFS tree for the large (giant) component. Finally, Rod-Opt is from 1.5 (50% queries) to 2 (33% queries) times faster than Rod 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 (the update of each one costs $O(n^2)$ time). Moreover, the recursive maintenance of the closure matrices turns out to be inefficient, because DI becomes slower as the recursion depth increases. 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. Actually, DI has the smallest ratio of cache misses w.r.t. any algorithm in our study.

The comparison of the rest of the algorithms is shown in Fig. 2. Simple-minded and pseudo fully dynamic algorithms clearly outperform RZ-1 and RZ-P. RZ-P is penalized by its larger query time, by its large memory demand after a certain point (see Fig. 2(right)), and most importantly by the maintenance of SCCs across all versions of the graph (quite expensive during deletions). Fig. 5(left) shows clearly that 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, seems to be the fact that its decremental data structure (i.e., RZ-Opt) must be rebuilt every $O(\sqrt{n})$ operations.

We now turn to the three faster implementations DBFS, Ita1-Gen, and RZ-Opt; see Fig. 3. When the graph is relatively sparse (less than $n \ln n$ edges), DBFS is the fastest algorithm, while RZ-Opt is considerably faster in denser graphs. The differences between the performance of Ita1-Gen and RZ-Opt can be explained by how they handle SCCs. Ita1-Gen is not efficient in handling edge deletions in SCCs because even if a SCC does not break, it may spend $O(n + m)$ time to determine whether the SCC has broken and to rebuild its sparse certificate. This claim is confirmed with the support of Fig. 5(right) and Fig. 3(left): although the number of SCCs that split decreases, the running 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 it may spend $O(n + m)$ time to update the BFS trees. However, as the edge density increases, RZ-Opt performs better, since the BFS trees have small depth. 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. 3(left). Consequently, in sparse graphs, where many splits and merges of SCCs occur, both algorithms have more-or-less the same performance. As soon as the strong connectivity threshold $(n \ln n)$ is approached and/or surpassed RZ-Opt outperforms Ita1-Gen, as it performs much less work mainly for deletions.

**Synthetic Inputs.** We have considered and slightly modified the synthetic inputs introduced by Frigon et al [6], which enforce the algorithms to exhibit their worst-case behaviour. These graphs consist of a sequence of $s = \lfloor n/k \rfloor$ 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, such that the bridge inserted/deleted last would provide new reachability and SCC information from roughly $n/2$ to the other $n/2$ vertices of the graph.

As with random inputs, D1, King's algorithm and its variants as well as Rod and Rod-Opt were the worst, and hence we do not report results for them. Ita1-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. 4 illustrates the performance of the rest of the algorithms. 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 Ita1-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 data structures in those SCCs must be built from scratch. On the other hand, RZ-1 and RZ-P perform better than Ita1-Gen as $k$ 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 Ita1-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 therefore it maintains less versions of the graph. 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.

Real-world Inputs. We have also used two inputs motivated by real-world graphs. The first graph has 1259 vertices and 5101 edges and represents a fragment of the Internet visible from RIPE (www.ripe.net) that has been used in [6]. The second graph describes a US road network (ftp://edftp.cr.usgs.gov) having 576 vertices and 1762 edges, and has been used in [3]. On these graphs, we run random sequences of operations, and we observed similar results to the experiments on random inputs.

References

Fig. 2. Random digraphs with $n = 300$. Experiments run on P4. Left: $|\sigma| = 5000$ (33% queries). Right: $|\sigma| = 30000$ (33% queries).

Fig. 3. Random digraphs with $n = 700$. Experiments run on P4. Left: $|\sigma| = 5000$ (33% queries). Right: $|\sigma| = 50000$ (33% queries).

Fig. 4. Synthetic digraphs with \(|\sigma| = 1920\) \((33\%\) queries\). Experiments run on U.Sparc-II. Left: clique size 10. Right: clique size 80.

Fig. 5. Experiments 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-Gpt and Ital-Gen, for \(n = 700\) and \(|\sigma| = 5000\) \((33\%\) queries\).