Improved dynamic reachability algorithms for directed graphs

Liam Roditty and Uri Zwick *
School of Computer Science, Tel Aviv University
Tel Aviv 69978, Israel
E-mail: {liamr,zwick}@post.tau.ac.il

Abstract

We obtain several new dynamic algorithms for maintaining the transitive closure of a directed graph, and several other algorithms for answering reachability queries without explicitly maintaining a transitive closure matrix. Among our algorithms are:

(i) A decremental algorithm for maintaining the transitive closure of a directed graph, through an arbitrary sequence of edge deletions, in $O(mn)$ total *expected time, essentially the time needed for computing the transitive closure of the initial graph. Such a result was previously known only for* acyclic *graphs.*

(ii) Two fully dynamic algorithms for answering reachability queries. The first is deterministic and has an amortized insert/delete time of $O(m\sqrt{n})$, and worst-case query time of $O(\sqrt{n})$. The second is randomized and has an amortized insert/delete time of $O(m^{0.58}n)$ and worst-case query time of $O(m^{0.43})$. This significantly improves the query times of algorithms with similar update times.

(iii) A fully dynamic algorithm for maintaining the transitive closure of an acyclic graph. The algorithm is deterministic and has a worst-case insert time of $O(m)$, constant *amortized delete time of $O(1)$, and a worst-case query time of $O(n/\log n)$.*

Our algorithms are obtained by combining several new ideas, one of which is a simple sampling *idea used for detecting decompositions of strongly connected components, with techniques of Even and Shiloach [7], Italiano [14], Henzinger and King [10], and Frigioni et al. [8]. We also adapt results of Cohen [3] on estimating the* size *of the transitive closure to the dynamic setting.*

*Work supported in part by THE ISRAEL SCIENCE FOUNDATION founded by The Israel Academy of Sciences and Humanities.

1 Introduction

The problem of maintaining the transitive closure of a dynamic directed graph, i.e., a directed graph that undergoes a sequence of edge insertions and deletions, is a well studied and well motivated problem. Demetrescu and Italiano [6], improving an algorithm of King [15], obtained recently an algorithm for dynamically maintaining the transitive closure under a sequence of edge insertions and deletions with an amortized insert/delete time of $O(n^2)$, where n is the number of vertices in the graph. King and Thorup [17] reduced the space requirements of these algorithms. All these algorithms support *extended* insert and delete operations in which an arbitrary set of edges, all touching the same vertex, may be inserted, and a completely arbitrary set of edges may be deleted, all in one update operation.

When the transitive closure of a graph is explicitly maintained, it is of course possible to answer every reachability query, after each update, in $O(1)$ time. As the insertion or deletion of a single edge may change $\Omega(n^2)$ entries in the transitive closure matrix, an amortized update time of $O(n^2)$, in the worst-case, is essentially optimal. When the number of queries after each update operation is relatively small, it is desirable to have a dynamic algorithm with a smaller update time, at the price of a non-constant query time. Such algorithms can escape the $\Omega(n^2)$ lower bound on the amortized update time by *implicitly* maintaining the transitive closure matrix.

Several dynamic algorithms for answering reachability queries, without explicitly maintaining the transitive closure, were developed. Most recently, Demetrescu and Italiano [6] gave such a Monte Carlo algorithm with an amortized update time of $O(n^{1.58})$ and worst-case query time of $O(n^{0.58})$. They exhibit, in fact, a tradeoff between the update and query times. Smaller query times may be obtained at the cost of higher update times. However, their algorithm can only handle *acyclic* graphs, and can only insert or delete one edge at a time. Furthermore, it relies on fast rectangular matrix multiplication, and thus may not

Table 1. Fully dynamic reachability algorithms.

Graphs	Algorithm	Query	Amortized update time	Reference
DAGs	Monte Carlo	$O(1)$	$O(n^2)$	[16]
DAGs	Monte Carlo	$O(n^{0.58})$	$O(n^{1.58})$	[6]
DAGs	**Deterministic**	$\boldsymbol{O(\frac{n}{\log n})}$	$\boldsymbol{O(m)}$	**This paper**
General	Monte Carlo	$O(\frac{n}{\log n})$	$O(m\sqrt{n}\log^2 n)$	[10]
General	Monte Carlo	$O(\frac{n}{\log n})$	$O(m^{0.58}n)$	[10]
General	Monte Carlo	$O(1)$	$O(n^{2.26})$	[16]
General	Deterministic	$O(1)$	$O(n^2 \log n)$	[15]
General	Deterministic	$O(1)$	$O(n^2)$	[6]
General	**Deterministic**	$\boldsymbol{O(\sqrt{n})}$	$\boldsymbol{O(m\sqrt{n})}$	**This paper**
General	**Monte Carlo**	$\boldsymbol{O(m^{0.43})}$	$\boldsymbol{O(m^{0.58}n)}$	**This paper**

be very efficient in practice. Earlier, Henzinger and King [10] gave two Monte Carlo algorithms for answering reachability queries. The first algorithm has an amortized update time of $O(m\sqrt{n}\log^2 n)$ and a worst-case query time of $O(n/\log n)$, where m is the number of edges in the graph. The second one has an amortized update time of $O(m^{0.58}n)$ and a query time of $O(n/\log n)$.

We present two new fully dynamic reachability algorithms for general graphs that improve upon the results of Henzinger and King [10]. The first is a *deterministic* algorithm that has an amortized update time of $O(m\sqrt{n})$ and a worst-case query time of $O(\sqrt{n})$. The update time of this algorithm is faster by a polylogarithmic factor than the update time of the first algorithm of Henzinger and King [10] while the query time is reduced from $O(n/\log n)$ to $O(\sqrt{n})$. Furthermore, we can obtain a tradeoff between the update and query times. For every $t \le \sqrt{n}$, we can get an update time of $O(mn/t)$ and query time $O(t)$. This algorithm is purely combinatorial and does not use fast matrix multiplication algorithms.

Our second algorithm is a randomized algorithm with an amortized update time of $O(m^{0.58}n)$ and worst-case query time of $O(m^{0.43})$. This improves the query time of the second algorithm of Henzinger and King [10] from $O(n/\log n)$ to $O(m^{0.43})$. This algorithm does use fast matrix multiplication. We again get a tradeoff. For every $t \le (m\log n)^{1/\omega}$, we can get an update time of $O(mn\log n/t)$ and a query time of $O(t)$, where $\omega < 2.376$ is the matrix multiplication exponent (see Coppersmith and Winograd [4]). Note that this is essentially the same tradeoff as that of the first algorithm. But, when $m \ge n^{\omega/2}\log n$, larger values of t may be chosen, giving lower update times.

We also obtain a fully dynamic reachability algorithm for acyclic graphs. This algorithm is deterministic and has a *linear* amortized update time of $O(m)$ and a worst-case query time of $O(n/\log n)$. A comparison between our dynamic reachability algorithms and the previously available ones is given in Table 1.

In time bounds above given for decremental algorithms, m stands for the *initial* number of edges in the graph. In time bounds given for fully dynamic algorithms, m stands for the *maximum* number of edges in the graph during the *phase* in which the update operation is performed.

One of the ingredients used in obtaining the improved fully dynamic reachability algorithms is an improved *decremental* algorithm for maintaining the transitive closure. A decremental algorithm is an algorithm that can handle deletions but not insertions. Italiano [14] obtained a decremental algorithm for *acyclic* graphs that processes any sequence of deletions in $O(mn)$ time. Slower algorithms for general, i.e., not necessarily acyclic, graphs were obtained by La Poutré and van Leeuven [18], Frigioni *et al.* [8], Demetrescu and Italiano [6], and by Baswana *et al.* [1]. A summary of previous decremental algorithms for maintaining the transitive closure, and for answering reachability queries is given in Table 2. (All the algorithms there, except that of Henzinger and King [10], explicitly maintain the transitive closure matrix.)

We obtain a new randomized decremental algorithm for maintaining the transitive closure of arbitrary, not necessarily acyclic, graphs. It processes *any* sequence of edge deletions in a total expected time of $O(mn)$. The algorithm is a Las Vegas algorithm, i.e., its answers are always correct. This matches the time bound of Italiano [14] for acyclic graphs, and answers an open problem raised there. As mentioned in the abstract, a time bound of $O(mn)$ is essentially

Table 2. Decremental reachability algorithms.

Graphs	Algorithm	Query	Total update time	Reference
DAGs	Deterministic	$O(1)$	$O(mn)$	[14]
General	Monte Carlo	$O(\frac{n}{\log n})$	$O(mn \log^2 n)$	[10]
General	Deterministic	$O(1)$	$O(m^2)$	[18, 8]
General	Deterministic	$O(1)$	$O(n^3)$	[6]
General	Monte Carlo	$O(1)$	$\tilde{O}(mn^{4/3})$	[1]
General	**Las Vegas**	$\boldsymbol{O(1)}$	$\boldsymbol{O(mn)}$	**This paper**

optimal for the problem, as $\Omega(mn)$ time is needed just for computing the transitive closure of the initial graph using the currently best matrix multiplication-free algorithm. The new decremental algorithm is based on a very simple sampling idea.

Next, we adapt the results of Cohen [3] on estimating the size of the transitive closure to the dynamic setting. In particular, we obtain an incremental algorithm that can process any sequence of edge insertions and requests to estimate the number of vertices reachable from a certain vertex in $O(m \log n + q)$ time, where m is the total number of edges inserted and q is the number of queries. We also obtain such a decremental algorithm for acyclic graphs. In the fully dynamic setting, we can provide such estimates at the cost of $O(\log n)$ reachability queries.

We can also extend the results of Cicerone *et al.* [2] so that they will apply to general, not necessarily acyclic, graphs. Details would again be given in the full version of the paper.

The rest of this extended abstract is organized as follows. In the next section we present a new decremental algorithm for maintaining the strongly connected components of a directed graph. This algorithm is used in Section 3 to obtain the $O(mn)$ decremental algorithm for maintaining the transitive closure of general directed graphs. In Section 4 we then describe *three* new fully dynamic reachability algorithms for general graphs. (Only two of them were mentioned above.) In Section 5 we describe a new fully dynamic reachability algorithm for acyclic graphs. In Section 6 we sketch our dynamic size estimation results. We end in Section 7 with some concluding remarks and open problems.

2 Decremental maintenance of strongly connected components

In this section we consider the dynamic maintenance of the strongly connected components (SCCs) of a directed graph under a sequence of edge deletions. This is an easier problem than the maintenance of the transitive closure of a graph. In Section 3, however, we use the results of this section to obtain an improved decremental algorithm for the maintenance of the transitive closure, and in Section 4 we use this decremental algorithm as a building block in our new fully dynamic reachability algorithms.

The new algorithm is given in Figure 1. It handles any sequence of edge deletions and queries in $O(mn + q)$ total *expected* time, where q is the number of queries. Each query is answered correctly in $O(1)$ worst-case time. The expected amortized time per edge deletion, if all edges are eventually deleted, is $O(n)$.

The algorithm starts by computing the SCCs of the graph using any linear time algorithm (see Tarjan [20], Sharir [19], Gabow [9], or Chapter 22 of Cormen *et al.* [5]). In each SCC C of the graph it then constructs and maintains a shortest-paths in-tree $In(w)$ and a shortest-paths out-tree $Out(w)$ rooted at a *random* representative w of this SCC. These shortest-paths trees are maintained using the decremental algorithm of Even and Shiloach [7], as adapted to directed graphs by Henzinger and King [10]. If C is composed of n' vertices and m' edges, the total cost of maintaining these two shortest-paths trees, over any sequence of edge deletions, is $O(m'n')$.

The algorithm also maintains an array A of length n that holds for every vertex v the representative vertex of the SCC containing v. Using this array it is easy to answer any strong connectivity query in $O(1)$ time.

Edge deletions are handled as follows. If the edge $e = (u, v)$ is not contained in a SCC, i.e., if $A(u) \neq A(v)$, then nothing needs to be updated. If e is contained in a SCC C with representative vertex w, i.e., $A(u) = A(v) = w$, and e is not contained the trees $In(w)$ and $Out(w)$, then again, the SCCs of the graph do not change and we only need to record that the edge e was deleted.

The difficult case, of course, is when e is contained in one of the trees $In(w)$ or $Out(w)$. In this case, we use the decremental algorithm to update the shortest-paths trees $In(w)$ and $Out(w)$. If after this update we have $u \in In(w)$

and $v \in Out(w)$, then there is still a directed path from u to v in the graph. Thus C is still a SCC, and the partition of the graph into SCCs did not change.

If $u \notin In(w)$ or $v \notin Out(w)$, then clearly C is no longer a SCC. We construct, in $O(m' + n')$ time, the new SCCs $C_1, C_2, \ldots, C_k$ to which C decomposed. Let C_i be the new SCC containing w. We let $w_i = w$ be the representative of C_i. In every other SCC C_j, for $j \neq i$, we choose a *random* representative $w_j \in C_j$.

By removing from $In(w)$ and $Out(w)$ the vertices that do *not* belong to C_i, we obtain shortest-paths trees that span C_i. It is crucial, for the analysis of the algorithm, to note that the decremental data structures maintaining these two shortest-paths trees do not have to be reinitialized.

From each random representative w_j, for $j \neq i$, we build from scratch shortest-paths trees $In(w_j)$ and $Out(w_j)$ that span C_j, and initialize the data structure of Even and Shiloach [7] for maintaining them. Finally, we update the array A accordingly. We now claim:

Theorem 2.1 *The algorithm of Figure 1 correctly handles any sequence of deletions and strong connectivity queries. Each query is answered in $O(1)$ time. The expected running time of the algorithm, for any sequence of deletions and queries is $O(mn + q)$, where q is the number of queries.*

Proof: The correctness of the algorithm follows easily from the above discussion. (Note that the random choices of the representatives only affect the running time, not the answers given.) It remains, therefore, to show that expected time spent in processing *all* edge deletions is only $O(mn)$.

Let $f(m, n)$ be the expected running time of the algorithm on the worst possible strongly connected graph with m edges and n vertices, and for the worst sequence of edge deletions. (If the initial graph is not strongly connected, we repeat the analysis in each strongly connected component.) We claim that

$$f(m,n) \;\leq\; mn + \sum_{i=1}^{k} \left(f(m_i, n_i) - \frac{m_i n_i^2}{n} \right) \;,$$

for some $k \geq 2$ and $m_1, m_2, \ldots, m_k \geq 0$, $n_1, n_2, \ldots, n_k \geq 1$ such that $\sum_{i=1}^{k} m_i \leq m$ and $\sum_{i=1}^{k} n_i = n$. Here, k is the number of SCCs to which the graph breaks when it is no longer strongly connected, and m_i and n_i, respectively, are the number of edges and vertices in the i-th SCC. (Note that k, the n_i's and the m_i's do not depend on the random choices made by the algorithm.)

The term mn covers the initialization cost of the algorithm and the cost of *all future maintenance operations* performed on the shortest-paths trees $In(w)$ and $Out(w)$. When the graph breaks into the k SCCs, the algorithm continues independently on each one of them. So we clearly have $f(m, n) \leq mn + \sum_{i=1}^{k} f(m_i, n_i)$.

This naive estimate fails, however, to take advantage of the following fact. The new component that contains w, the representative of the original component, *inherits* the shortest-paths trees $In(w)$ and $Out(w)$, and does not have to pay for their construction and maintenance. Furthermore, as w was randomly chosen, the larger a new component is, the more likely it is to receive this 'gift'. The probability that a new component of n_i vertices will contain w is n_i/n. Thus, with a probability of n_i/n, the term $m_i n_i$, incorporated into $f(m_i, n_i)$, can be dispensed with, giving the desired relation. We now claim:

Lemma 2.2 *$f(m, n) \leq 2mn$.*

Proof: The proof is, of course, by induction. The basis of the induction is easily established. Suppose now that the claim holds for any (m', n') with $m' < m$ and $n' < n$. We show that it also holds for (m, n). We have to verify that

$$mn + 2\sum m_i n_i - \sum \frac{m_i n_i^2}{n} \;\leq\; 2mn \;.$$

Letting $x_i = m_i/m$ and $y_i = n_i/n$, so that $x_i, y_i \geq 0$, $\sum x_i \leq 1$ and $\sum y_i = 1$, we get after a simple manipulation that we have to verify that

$$2\sum x_i y_i - \sum x_i y_i^2 \;\leq\; 1 \;.$$

We show that in fact $2\sum x_i y_i - \sum x_i y_i^2 \leq \sum x_i \leq 1$. This follows as we have $\sum x_i - 2\sum x_i y_i + \sum x_i y_i^2 = \sum x_i (1 - y_i)^2 \geq 0$. This completes the proof. □

This completes the proof of the theorem. □

3 Decremental maintenance of the transitive closure

Frigioni *et al.* [8] extend the decremental algorithm of Italiano [14] so that it could handle general, not necessarily acyclic, graphs. Frigioni *et al.* [8] report that their algorithm works well in practice, though its worst-case time complexity is $O(m^2)$.

The algorithm of Frigioni *et al.* [8] maintains the strongly connected components of the graph, and the *skeleton* of the graph, i.e., the DAG induced on the strongly connected components. The skeleton is maintained using Italiano's algorithm [14]. For each SCC, the algorithm maintains a *sparse certificate* composed of an in-tree and an out-tree rooted at an arbitrary vertex. When an edge from this certificate is deleted, their algorithm may have to spend $O(m + n)$ time in checking whether the SCC decomposed. As this may happen every time an edge is deleted, the total running time of the algorithm may be $\Omega(m^2)$.

init(V):

1. Allocate an array A of size n.
2. Choose a random vertex $w \in V$.
3. Call $findSCC(V, w)$.

findSCC(C, w):

1. Find the SCCs $C_1, C_2, \ldots, C_k$ of the graph $G[C]$.
2. In each SCC C_j, where $1 \leq j \leq k$, do:
   (a) If $w \in C_j$, then let $w_j \leftarrow w$. Otherwise, choose a *random* representative $w_j \in C_j$.
   (b) For every $v \in C_j$, let $A(v) \leftarrow w_j$.
   (c) Initialize decremental data structures for maintaining a shortest-paths in-tree $In(w_j)$ and a shortest-paths out-tree $Out(w_j)$ rooted at w_j.

query(u, v):

1. If $A(u) = A(v)$ then 'yes', otherwise 'no'.

delete(u, v):

1. If $A(u) \neq A(v)$, i.e., u and v are not in the same SCC, do nothing.
2. Otherwise, let $w \leftarrow A(u)$, and let C be the vertices of the SCC containing w.
3. Delete the edge (u, v), if necessary, from the trees $In(w)$ and $Out(w)$ using the appropriate decremental data structures.
4. If $u \notin In(w)$ or $v \notin Out(w)$, i.e., C decomposed, then call $findSCC(C, w)$.

Figure 1. A randomized decremental algorithm for maintaining strongly connected components.

However, the total running time of the algorithm of Frigioni *et al.* [8], *excluding* the time needed to detect decompositions of SCCs is only $O(mn)$. Thus, combining their algorithm with our algorithm for maintaining the SCCs yields a decremental algorithm for maintaining the transitive closure of general graphs with a total expected time of $O(mn)$, matching the time bound of Italiano [14] for acyclic graphs.

4 Fully dynamic reachability algorithms

4.1 The first fully dynamic algorithm

Our first fully dynamic reachability algorithm is given in Figure 2. It is essentially a combination of an algorithm of Henzinger and King [10] with our improved decremental reachability algorithm, or with the somewhat slower, but deterministic algorithm of Frigioni *et al.* [8].

The algorithm works in *phases*. In the beginning of each phase, a decremental reachability data structure is initialized. When a set of edges E_v touching v is inserted, we add v to S and construct reachability trees $In(v)$ and $Out(v)$ rooted at v. When the size of S, the set of the vertices that were centers of insertions in the current phase, reaches t, a parameter fixed in advance, the phase is declared over, and all the data structures are reinitialized.

The deletion of an arbitrary set E' of edges is handled as follows. First, the edges of E' are removed from the decremental data structure. Next, for every $w \in S$, the shortest-paths trees $In(w)$ and $Out(w)$ are rebuilt from scratch.

A query $query(u, v)$ is answered as follows. First the decremental data structure is queried to see whether there is a directed path from u to v composed solely of edges that were present in the graph at the start of the current phase. If not, it is checked whether there exists $w \in S$ such that $u \in In(w)$ and $v \in Out(w)$.

It is easy to check that the answer given for each query is always correct. Clearly, if $query(u, v)$ returns 'yes', then there is indeed a path from u to v in the graph. Suppose now that there is a path p from u to v in the graph. If this path uses only 'old' edges, i.e., edges that were not inserted in the current phase, the decremental data structure would signal that out. Otherwise, let w be the *last* vertex on a path from u to v that was the center of an insert operation during the current phase. This insert operation added w to S and constructed the trees $In(w)$ and $Out(w)$. At the time of this insertion all the edges of the path p were already present in the graph, so $u \in In(w)$ and $v \in Out(w)$. Some edges from these trees may be subsequently deleted, but as the path p remains in the graph, the vertex u would stay in $In(w)$, and similarly v would stay in $Out(w)$. This completes the proof of correctness. We claim:

init: 1. Initialize a decremental reachability data structure. 2. Let $S \leftarrow \phi$.
query(u, v): 1. Query the decremental reachability data structure. 2. For each $w \in S$ check if $u \in In(w)$ and $v \in Out(w)$.
delete(E'): 1. Delete E' from the decremental data structure. 2. For every $w \in S$, rebuild the trees $In(w)$ and $Out(w)$.
insert(E_v): 1. Let $S \leftarrow S \cup \{v\}$. 2. If $\|S\| > t$, then call $init$. 3. Otherwise, construct the trees $In(v)$ and $Out(v)$.

Figure 2. The first fully dynamic reachability algorithm for general graphs.

Theorem 4.1 *For any $t \leq \sqrt{n}$, the algorithm of Figure 2 handles each insert or delete operation in $O(mn/t)$ amortized time, and answers each query correctly in $O(t)$ worst-case time. In particular, when $t = \sqrt{n}$, the amortized update time is $O(m\sqrt{n})$, and the worst-case query time is $O(\sqrt{n})$.*

Proof: Assume, at first, that our decremental reachability algorithm is used. The expected complexity of setting up the decremental data structure in the beginning of each phase, and of handling all subsequent delete operations on it is only $O(mn)$. As each phase, except possibly the last phase, is composed of at least t update operations, we can cover this cost by charging $O(mn/t)$ of these operations to each update.

Each delete operation involves the recomputation of up to $2t$ trees. This is easily done in $O(mt)$ time. An insert operation is even cheaper as only two trees need to constructed.

The total expected amortized cost per an insert or delete operations is therefore $O(mn/t + mt)$. When $t \leq \sqrt{n}$, the first term dominates the second and the expected cost per operation is $O(mn/t)$, assuming that at least t update operations are performed. The query time is clearly $O(t)$.

As presented, the algorithm is randomized (Las Vegas). It turns out, however, that the same time bounds may be obtained using the decremental algorithm of Frigioni *et al.* [8], as the total running time of their algorithm is $O(mn + del \cdot m)$, where del is the number of delete operations performed. (Obtaining such a result when each delete operation deletes only one edge from the graph is easy. Handling the more general case in which each delete operation may delete an arbitrary set of edges from the graph requires more care. The full details will appear in the full version of this paper). □

4.2 The second fully dynamic algorithm

Our second fully dynamic reachability algorithm is given in Figure 3. It is essentially a combination of a second algorithm of Henzinger and King [10] with our decremental reachability algorithm, or with the algorithm of Frigioni *et al.* [8].

The algorithm again works in *phases*. In the beginning of each phase, a decremental reachability data structure is again initialized. The algorithm again maintains a set S of *special* vertices. For each vertex $w \in S$, the algorithm maintains an in-tree $In(w)$ and an out-tree $Out(w)$. These trees are *shortest-paths* trees that contain all vertices that are at distance at most $(cn \ln n)/t$ from w, where c is some fixed constant. (For concreteness, we choose $c = 10$.) These trees are maintained using the algorithm of Even and Shiloach [7]. In the beginning of each phase, t *random* vertices are placed in S.

The algorithm also maintains two Boolean matrices, A_1^* of size $n \times |S|$, and A_2 of size $|S| \times n$. The columns of A_1^* and the rows of A_2 are indexed by the elements of S. For every $u \in V$ and $w \in S$, we have $A_1^*(u, w) = 1$ if and only if there is a path (of arbitrary length) in the graph from u to w, and $A_2(w, u) = 1$ if and only if there is a path of length at most $(cn \ln n)/t$ from w to u.

When a set of edges E_v touching v is inserted, we add v to the set S of special vertices and construct shortest-paths trees $In(v)$ and $Out(v)$ of depth at most $(cn \ln n)/t$ rooted at v. When the size of the set S reaches $2t$, a parameter

init:

1. Initialize a decremental reachability data structure.
2. Let S be a *random* set of t vertices.
3. For every $w \in S$, construct shortest-paths trees $In(w)$ and $Out(w)$ of depth at most $(cn \ln n)/t$, and initialize decremental data structure for them.
4. Call $build(S)$.

build(S):

1. Construct Boolean matrices A_1, A_2 and B of sizes $n \times |S|$, $|S| \times n$, and $|S| \times |S|$:
   (a) $A_1(u, w) = 1$ iff $u \in In(w)$, for every $u \in V$ and $w \in S$.
   (b) $A_2(w, v) = 1$ iff $v \in Out(w)$, for every $w \in S$ and $v \in V$.
   (c) $B(w_1, w_2) = 1$ iff $w_1 \in In(w_2)$, for every $w_1, w_2 \in S$.
2. Compute B^*, the transitive closure of B, and $A_1^* = A_1 B^*$.

query(u, v):

1. Query the decremental data-structure.
2. Check whether there exists $w \in S$ such that $A_1^*(u, w) = A_2(w, v) = 1$.

delete(E'):

1. Delete E' from the decremental data structure.
2. For every $w \in S$, update the shortest-paths trees $In(w)$ and $Out(w)$ of depth at most $(cn \ln n)/t$ using the decremental algorithm for maintaining shortest-paths trees.
3. Call $build(S)$.

insert(E_v):

1. Let $S \leftarrow S \cup \{v\}$.
2. If $|S| > 2t$, then call $init$.
3. Otherwise, construct shortest-paths trees $In(v)$, $Out(v)$ of depth at most $(cn \ln n)/t$ and initialize a data structure for maintaining them under a sequence of edge deletions.
4. Call $build(S)$.

Figure 3. The second fully dynamic reachability algorithm for general graphs.

fixed in advance, the phase is over, and all data structures are reinitialized.

The deletion of an arbitrary set E' of edges is handled as follows. First, the edges of E' are removed from the decremental data structure. Next, for every $w \in S$, the shortest-paths trees $In(w)$ and $Out(w)$ are updated using the algorithm of Even and Shiloach [7].

A query $query(u, v)$ is answered as follows. First the decremental data structure is queried to see whether there is a directed path from u to v composed solely of 'old' edges. If not, it is checked whether there exists $w \in S$ such that $A_1^*(u, w) = A_2(w, v) = 1$. The correctness of the algorithm relies on the following observation of Ullman and Yannakakis [21]:

Lemma 4.2 *Let $G = (V, E)$ be a directed graph on n vertices. Let S be a set of $(cn \ln n)/t$ random vertices. Then, with a probability of at least $1 - n^{-(c-3)}$, for every two vertices $u, v \in V$, if there is a path from u to v in G, then there is also such a path that among any t consecutive vertices on it there is a vertex from S.*

As stated, the lemma applies to a fixed graph. However, it is easy to adapt it to our dynamical setting:

Corollary 4.3 *Let $G_1, G_2, \ldots, G_\ell$ be directed graphs on the same set of n vertices. Let S be a set of $(cn \ln n)/t$ random vertices. Then, with a probability of at least $1 - \ell n^{-(c-3)}$, for every $1 \le i \le \ell$ and every $u, v \in V$, if there is a path from u to v in G_i, then there is also such a path in G_i that among any t consecutive vertices on it there is a vertex from S.*

The random set S may be chosen, of course, without knowing the sequence of graphs. Due to lack of space, we omit the straightforward proof of the lemma. We note in passing that similar ideas are also used by Zwick [22] and King [15].

Theorem 4.4 *For any $t \leq (m \log n)^{1/\omega}$, the algorithm of Figure 2 handles each insert or delete operation in $O(mn \log n/t)$ amortized time, and answers each query correctly, with very high probability, in $O(t)$ worst-case time. In particular, when $t = (m \log n)^{1/\omega}$, the expected amortized update time is $O((m \log n)^{1-1/\omega} n)$, and the worst-case query time is $O((m \log n)^{1/\omega})$.*

Proof: The correctness of the algorithm follows easily from Corollary 4.3. We omit the details, due to lack of space. As with the previous algorithm, the $O(mn)$ complexity of setting up and maintaining the decremental data structure is split among the at least t updates operations of a phase.

In the beginning of each phase, the algorithm also sets up $2t$ shortest-paths trees of depth at most $(cn \ln n)/t$. The cost of setting up and maintaining these trees throughout the phase, using the algorithm of Even and Shiloach [7], is $O(2t \cdot m \cdot \frac{cn \ln n}{t}) = O(mn \log n)$. This cost is again split among the update operations of the phase.

Each delete operation updates the decremental data structure and the shortest-paths trees. These operations are already accounted for. It also involves the call $build(S)$. As $|S| \leq 2t$, the complexity of this procedure is $O(\frac{n}{t} \cdot t^{\omega}) = O(nt^{\omega-1})$, where $\omega < 2.376$ is the exponent of matrix multiplication.

Each insert operation constructs two new shortest-paths trees of depth at most $(cn \ln n)/t$. The total cost of maintaining these trees throughout the phase, using the algorithm of Even and Shiloach [7] is $O(mn \log n/t)$. The cost of calling $build(S)$ is again $O(nt^{\omega-1})$.

Thus, the total amortized cost per each update operation is $O(\frac{mn \log n}{t} + nt^{\omega-1})$. When $t \leq (m \log n)^{1/\omega}$, the first term is the dominant term, and the expected amortized time per update is $O((mn \log n)/t)$. Each query is clearly answered in $O(t)$ worst-case time. □

We note in passing that the tradeoff of an amortized update time of $O((mn \log n)/t)$ and query time of $O(t)$ can be extended to values of t that are slightly larger than $(m \log n)^{1/\omega}$ using the fast rectangular matrix multiplication algorithms of Huang and Pan [13].

4.3 A third fully dynamic reachability algorithm

Our third fully dynamic reachability algorithm for general graphs is given in Figure 4. It is somewhat similar to our second algorithm. However, it does not maintain the matrices A_1^* and A_2, and it uses a fully dynamic algorithm, e.g., the algorithm of Demetrescu and Italiano [6], to maintain the matrix B^*. We claim:

init:

1. Initialize a decremental reachability data structure.
2. Let S be a *random* set of t vertices.
3. For every $w \in S$, construct shortest-paths trees $In(w)$ and $Out(w)$ of depth at most $(cn \ln n)/t$, and initialize decremental data structures for maintaining them.
4. Construct a directed graph $H = (S, F)$, where $F = \{(w_1, w_2) \in S^2 \mid w_1 \in In(w_2)\}$.
5. Initialize a *fully* dynamic algorithm for maintaining the transitive closure B^* of H.

$\boldsymbol{query(u,v)}$:

1. Query the decremental data-structure.
2. Check whether there exist $w_1, w_2 \in S$ such that $u \in In(w_1)$, $B^*(w_1, w_2) = 1$ and $v \in Out(w_2)$.

$\boldsymbol{delete(E')}$:

1. Delete E' from the decremental data structure.
2. For every $w \in S$, update the shortest-paths trees $In(w)$ and $Out(w)$ of depth at most $(cn \ln n)/t$ using the appropriate decremental algorithms.
3. Update the transitive closure B^* of the graph H after the removal of edges from F.

$\boldsymbol{insert(E_v)}$:

1. Let $S \leftarrow S \cup \{v\}$.
2. If $|S| > 2t$, then call $init$.
3. Otherwise, construct shortest-paths trees $In(v)$ and $Out(v)$ of depth at most $(cn \ln n)/t$ and initialize decremental data structures for maintaining them.
4. Update the transitive closure B^* of the graph H after the addition of v to it.

Figure 4. A third fully dynamic reachability algorithm for general graphs.

Theorem 4.5 *For any $1 \leq t \leq \sqrt{m}$, the algorithm of Figure 4 handles each insert or delete operation in $O(mn \log n/t)$ amortized time, and answers each query correctly, with very high probability, in $O(t^2)$ worst-case time. In particular, when $t = m^{(1-\epsilon)/2}$, the amortized update time is $O(m^{(1+\epsilon)/2} n \log n)$ and the worst-case query time is $O(m^{1-\epsilon})$.*

Proof: The correctness proof of the algorithm is identical to the correctness proof of the second fully dynamic algorithm. The cost of initializing a phase is $O(mn \log n + t^3)$. The cost of an insert operation is $O(mn \log n/t + t^2)$. (The first term is the cost of constructing and maintaining the trees $In(v)$ and $Out(v)$. The second term is the cost of updating of the matrix B^* using the fully dynamic algorithm for maintaining the transitive closure.) The added cost of a delete operation is only $O(t^2)$, the cost of updating B^*. Thus, the amortized cost of each update operation is $O(mn \log n/t + t^2)$. As $t \leq \sqrt{m}$, the first term is always dominant. The query time is clearly $O(t^2)$. □

5 A very simple fully dynamic reachability algorithm for acyclic graphs

A very simple fully dynamic reachability algorithm for acyclic graphs is presented in Figure 5. The algorithm is based on the main idea of King [15]. The acyclicity assumption allows us to greatly simplify the algorithm, and to obtain the first fully dynamic reachability algorithm, for acyclic graphs, with a *linear*, i.e., $O(m)$, amortized update time. The query time of the algorithm, $O(n/\log n)$, is quite large. However, it is still much faster than the $\Omega(m)$ time that may be needed to answer such a query without a dynamic data structure.

Italiano [14] showed that, in acyclic graphs, a forest of reachability trees, one rooted at each vertex, can be decrementaly maintained in $O(mn)$ total time. His result is, in fact, stronger. Each one of these trees can be *individually* maintained in $O(m)$ total time. Our algorithm exploits this fact.

Theorem 5.1 *The algorithm of Figure 5 handles each insert operation, that keeps the graph acyclic, in $O(m)$ worst-case time, each delete operation in $O(1)$ amortized time, and answers every reachability query correctly in $O(n/\log n)$ worst-case time.*

Proof: The algorithm starts by constructing a forest of in-trees and a forest of out-trees. Each one of these trees is individually maintained using the data structure of Italiano [14]. When a set E' of edges is deleted, we simply update each one of these trees individually. To insert a set E_v of edges, we simply rebuild the trees $In(v)$ and $Out(v)$. The cost of building these two trees, and of maintaining them through all future delete operations is only $O(m)$. Thus, the cost of all delete operations is covered by either the initialization cost, of $O(mn)$, or by preceding insert operations.

A query $query(u, v)$ is answered by checking whether there is a $w \in V$ such that $u \in In(w)$ and $v \in Out(w)$. If there is a path p from u to v, then this condition holds when w is the last vertex on the path that was the center of an insert operation, or by u and v themselves, if no such insertions took place. As described, each query would require $O(n)$ time.

However, it is easy to reduce the query time to $O(n/\log n)$. The algorithm essentially maintains two $n \times n$ Boolean matrices A and B such that $A(u, w) = 1$ if and only if $u \in In(w)$, and $B(w, v) = 1$ if and only if $v \in Out(w)$. We can *pack* each row of A and B into $n/\log n$ machine words, and each query would then require only $O(n/\log n)$ time. □

$\boldsymbol{init(V)}$:

- ◇ For every $v \in V$ construct reachability trees $In(v)$ and $Out(v)$ and initialize appropriate decremental data structures for them.

$\boldsymbol{query(u, v)}$:

- ◇ For every $w \in V$, check whether $u \in In(w)$ and $v \in Out(w)$.

$\boldsymbol{delete(E')}$:

- ◇ Delete E' from all reachability trees and update each one of them using the decremental single-source reachability algorithm for DAGs.

$\boldsymbol{insert(E_v)}$:

- ◇ Call $init(\{v\})$.

Figure 5. A very simple dynamic reachability algorithm for acyclic graphs.

6 Dynamic estimation of the size of reachability sets

Cohen [3] presents an $O(m)$ time randomized algorithm that estimates, for every vertex of a given directed graph, the number of vertices that are reachable from that vertex. We discuss here adaptations of her ideas to the dynamic setting.

One of the variants of the algorithm of Cohen [3] works roughly as follows. It chooses a random permutation on the vertices of the graph and labels the vertices according to it. For every vertex v, it then finds the smallest label $s(v)$ assigned to a vertex reachable from v. In the static setting, this can be easily done in $O(m)$ time. Then, $n/s(v)$ is a reasonable estimate to the number of vertices reachable from v. To obtain higher accuracy and higher confidence, this experiment is repeated several times and the results are combined in several possible ways. See Cohen [3] for exact details.

We make here the simple observation that a request to estimate the size of a reachability set can be reduced to $O(\log n)$ reachability queries. This is done as follows. Let $\epsilon > 0$. Add $k = \log_{1+\epsilon} n$ new vertices $u_1, u_2, \ldots u_k$ to the graph. For every $1 \leq i \leq k$, add an edge (v, u_i) for every vertex $v \in V$ whose label is in $[(1+\epsilon)^{i-1}, (1+\epsilon)^i]$. Now, for every $v \in V$, the queries $query(v, u_i)$, for $1 \leq i \leq k$, allow us to estimate $s(v)$ with a relative error of ϵ, which is good enough for our purposes. Furthermore, these queries involve only $k = O(\log n)$ destinations. This can be exploited, especially in the semi-dynamic setting, to obtain more efficient algorithms. As mentioned in the introduction, we can obtain an incremental algorithm whose total running time is $O(m \log n + q)$, and a decremental algorithm with the same time bound for acyclic graphs. Details will be given in the full version of the paper.

7 Concluding remarks and open problems

We presented an essentially optimal decremental algorithm for maintaining the transitive closure of a general graph. We also presented several improved fully dynamic algorithms for the reachability problem. There is still a huge gap, however, between the results obtained here, and elsewhere, for *directed* graphs, and the polylogarithmic results available for *undirected* graphs (see Henzinger and King [11] and Holm *et al.* [12]).

Many open problems still remain. Among them are:

1. Is there a decremental algorithm for maintaining the strongly connected components of a directed graph whose total running time is $o(mn)$?
2. Is there a decremental algorithm for maintaining a shortest-paths tree, or even just a reachability tree, from a *single* source in a general directed graph whose total running time is $o(mn)$?
3. Is there a *deterministic* decremental algorithm for maintaining the transitive closure of a general directed graph whose total running time is $O(mn)$?
4. Is there a fully dynamic reachability algorithm with an amortized update time of $o(n^2)$, and worst-case query time of $o(m)$ for *general* directed graphs? (In other words, can the '$O(n^2)$ barrier' be broken for general, not necessarily acyclic, graphs?)

References

[1] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for transitive closure and all-pairs shortest paths. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing, Montréal, Canada*, pages 117–123, 2002.

[2] S. Cicerone, D. Frgioni, U. Nanni, and F. Pugliese. A uniform approach to semi-dynamic problems on digraphs. *Theoretical Computer Science*, 203:69–90, 1998.

[3] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.

[4] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, second edition, 2001.

[6] C. Demetrescu and G. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proceedings of the 41th Annual IEEE Symposium on Foundations of Computer Science, Redondo Beach, California*, pages 381–389, 2000.

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

[8] D. Frigioni, T. Miller, U. Nanni, and C. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM Journal of Experimental Algorithmics*, 6, 2001.

[9] H. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, 2000.

[10] M. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science, Milwaukee, Wisconsin*, pages 664–672, 1995.

[11] M. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[12] J. Holm, K. de Lichtenberg, and M. Thorup. Polylogarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

[13] X. Huang and V. Pan. Fast rectangular matrix multiplications and applications. *Journal of Complexity*, 14:257–299, 1998.

[14] G. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988.

[15] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, New York, New York*, pages 81–91, 1999.

[16] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the 31th Annual ACM Symposium on Theory of Computing, Atlanta, Georgia*, pages 492–498, 1999.

[17] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Computing and Combinatorics Conference, Guilin, China*, pages 269–277, 2001.

[18] J. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proceedings of the 13th International Workshop on Graph-Theoretic Concepts in Computer Science, Amsterdam, The Netherlands*, volume 314, 1987.

[19] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[20] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 11:146–159, 1982.

[21] J. Ullman and M. Yannakakis. High probability parallel transitive closure algorithms. *SIAM Journal on Computing*, 20, 1991.

[22] U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49:289–317, 2002.