SIAM J. COMPUT.
Vol. 37, No. 5, pp. 1455–1471

IMPROVED DYNAMIC REACHABILITY ALGORITHMS FOR DIRECTED GRAPHS*

LIAM RODITTY† AND URI ZWICK†

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 [*J. ACM*, 28 (1981), pp. 1–4], Italiano [*Inform. Process. Lett.*, 28 (1988), pp. 5–11], Henzinger and King [*Proceedings of the* 36*th Annual Symposium on Foundations of Computer Science*, Milwaukee, WI, 1995, pp. 664–672], and Frigioni et al. [*ACM J. Exp. Algorithmics*, 6 (2001), (electronic)].

Key words. dynamic algorithms, transitive closure, strongly connected components

AMS subject classifications. 68W40, 68W20, 68W05, 68Q25

DOI. 10.1137/060650271

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 [5], improving an algorithm of King [15], recently obtained 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

*Received by the editors January 18, 2006; accepted for publication (in revised form) March 15, 2007; published electronically January 30, 2008. A preliminary version of this paper appeared in Proceedings of the 43rd Annual Syposium on Foundations of Computer Science, Vancouver, BC, Canada, 2002. This work was supported by grant I-792-136.6/2003 from the German-Israeli Foundation for Scientific Research and Development.

http://www.siam.org/journals/sicomp/37-5/65027.html

†School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel (liamr@tau.ac.il, zwick@tau.ac.il).

of a nonconstant 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 [5, 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 handle only *acyclic* graphs, and can insert or delete only one edge at a time. Furthermore, it relies on fast rectangular matrix multiplication and thus may not 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 $t \leq \sqrt{n}$, 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 $t \leq (m\log n)^{1/\omega}$, 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 [3]). Note that this is essentially the same tradeoff as that of the first algorithm. But, when $m \geq n^{\omega/2}\log n$, 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 1.1.

In the time bounds given above for decremental algorithms, m stands for the *initial* number of edges in the graph. In time bounds given above 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. (Phases will be defined later.)

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 [19], Frigioni et al. [8], Demetrescu and Italiano [5], and by Baswana, Hariharan, and Sen [1]. A summary of previous decremental algorithms for maintaining the transitive closure, and for answering reachability queries is given in Table 1.2. (All the algo-

TABLE 1.1
Fully dynamic reachability algorithms.

Graphs	Algorithm	Query	Amortized update time	Reference
DAGs	Monte Carlo	$O(1)$	$O(n^2)$	[16]
DAGs	Monte Carlo	$O(n^{0.58})$	$O(n^{1.58})$	[5]
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	$O(1)$	$O(n^2)$	[5]
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**

TABLE 1.2
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	$O(1)$	$O(m^2)$	[19, 8]
General	Deterministic	$O(1)$	$O(n^3)$	[5]
General	Monte Carlo	$O(1)$	$\tilde{O}(mn^{4/3})$	[1]
General	**Las Vegas**	$\boldsymbol{O(1)}$	$\boldsymbol{O(mn)}$	**This paper**

rithms 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 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 [2] 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.

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.

Recent developments. Since the appearance of the preliminary version of this paper, some additional dynamic algorithms for maintaining the transitive closure and for answering reachability queries were obtained. None of them, however, supersedes the results presented in this paper. Roditty [20] obtained another fully dynamic algorithm, with an amortized update time of $O(n^2)$ and a worst-case query time of $O(1)$, for maintaining the transitive closure matrix of a general graph. Sankowski [24] obtained a randomized algorithm with a worst-case update time of $O(n^2)$ and a worst-case query time of $O(1)$ for maintaining, with high probability, the transitive closure matrix. We [22] obtained yet another algorithm for answering reachability queries that has an amortized update time of $O(m + n \log n)$ and a query time of $O(n)$. Finally, Krommidas and Zaroliagis [18] have implemented some of the algorithms presented in this paper and they report that they work fairly well in practice.

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 a seemingly 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 2.1. 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 [26], Sharir [25], Gabow [9], or Chapter 22 of Cormen et al. [4]). 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, then 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 an SCC, i.e., if $A(u) \neq A(v)$, then nothing needs to be updated. If e is contained in an SCC C with representative vertex w, i.e., $A(u) = A(v) = w$, and e is not contained in the trees $In(w)$ and $Out(w)$, then again, the SCCs of the graph do not change and we need only 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 an SCC, and the partition of the graph into SCCs did not change.

***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 $Out(w_j)$ of $G[C_j]$ 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 $findSCC(C, w)$.

FIG. 2.1. *A randomized decremental algorithm for maintaining strongly connected components.*

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 the following theorem.

THEOREM 2.1. *The algorithm of Figure* 2.1 *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 affect only the running time, not the answers given.) It remains, therefore, to show that the expected time spent in processing *all* edge deletions is only $O(mn)$.

Let $f(m, n)$ be an upper bound on 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.) In the upper bound $f(m,n)$ we charge $m'n'$ units of time for the decremental maintenance of the in-tree and out-tree of an SCC containing, initially, n' vertices and m' edges, even if the actual cost of maintaining these trees is smaller.

We claim that

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

for some $k \ge 2$ and $m_1, m_2, \ldots, m_k \ge 0$, $n_1, n_2, \ldots, n_k \ge 1$ such that $\sum_{i=1}^{k} m_i \le m$ 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 ith 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 $f(m,n) \le mn + \sum_{i=1}^{k} f(m_i,n_i)$.

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 the following lemma.

LEMMA 2.2. $f(m,n) \le 2mn$.

Proof. The proof is 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

$$mn + 2\sum_{i=1}^{k} m_i n_i - \sum_{i=1}^{k} \frac{m_i n_i^2}{n} \;\le\; 2mn.$$

Letting $x_i = m_i/m$ and $y_i = n_i/n$, so that $x_i, y_i \ge 0$, $\sum_{i=1}^{k} x_i \le 1$, and $\sum_{i=1}^{k} y_i = 1$, we get after a simple manipulation that we have to verify that

$$2\sum_{i=1}^{k} x_i y_i - \sum_{i=1}^{k} x_i y_i^2 \;\le\; 1.$$

We show that in fact

$$2\sum_{i=1}^{k} x_i y_i - \sum_{i=1}^{k} x_i y_i^2 \;\le\; \sum_{i=1}^{k} x_i \;\le\; 1.$$

This follows as we have

$$\sum_{i=1}^{k} x_i - 2\sum_{i=1}^{k} x_i y_i + \sum_{i=1}^{k} x_i y_i^2 \;=\; \sum_{i=1}^{k} x_i (1-y_i)^2 \;\ge\; 0.$$

This completes the proof of the lemma. □

This completes the proof of the theorem. □

3. Decremental maintenance of the transitive closure. Our goal in this section is to prove the following two theorems.

THEOREM 3.1. *There is a randomized algorithm for maintaining the transitive closure matrix of a graph that undergoes a sequence of edge deletions whose total expected running time is* $O(mn)$.

THEOREM 3.2. *There is a deterministic algorithm for maintaining the transitive closure matrix of a graph that undergoes a sequence of edge deletions whose total running time is* $O(mn+del\cdot m)$, *where del is the number of delete operations performed on the graph. Each delete operation may remove an arbitrary set of edges from the graph.*

The first result is obtained by combining the algorithm for the decremental maintenance of the strongly connected components of a graph, described in the previous section, with an algorithm of Frigioni et al. [8] for the decremental maintenance of the transitive closure matrix. The second result is obtained by a small modification of the algorithm of Frigioni et al. For completeness, we sketch the operation of the algorithm of Frigioni et al. and describe the modifications needed to obtain our results.

Italiano [14] describes a deterministic algorithm, with a total running time of $O(mn)$, for the decremental maintenance of the transitive closure matrix of an *acyclic* directed graph. Frigioni et al. [8] extend Italiano's algorithm 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 $O(m^2)$.

We begin with a sketch of the operation of Italiano's algorithm. The algorithm maintains, in addition to the transitive closure matrix M, a collection of reachability trees, one rooted at every vertex of the graph. The tree of a vertex v, denoted by $T(v)$, contains all the vertices in the current version of the graph that are reachable from v. Note that $M(v,u)=1$ if and only if $u \in T(v)$. Every vertex u has two linked lists $E_{in}(u)$ and $E_{out}(u)$ of its incoming and outgoing edges that were not yet deleted. If $u \in T(v)$, then we let $p(v,u)$ be a pointer to the edge (u',u) in $E_{in}(u)$ such that u' is the parent of u in $T(v)$. If $u \notin T(v)$, then $p(v,u)=null$.

When an edge (u,w) is deleted from the graph, the algorithm performs the following operations. For every vertex v, it checks whether (u,w) is an edge of the tree $T(v)$. (This is done by checking whether $p(v,w)$ points to (u,w).) If (u,w) is not an edge of $T(v)$, then nothing needs to be done. Otherwise, if (u,w) is a tree edge, the algorithm tentatively sets $p(v,w)$ to point to the next edge (u',w) in $E_{in}(w)$, or to *null*, if (u,w) is the last edge of $E_{in}(w)$. Note that if $u' \in T(v)$, then the new edge (u',w) reconnects w to $T(v)$. As this condition still needs to be checked, the algorithm sets $R(v) \leftarrow \{w\}$. If (u,w) is not an edge of $T(v)$, or if (u,w) is the last edge in $E_{in}(w)$, it lets $R(v) \leftarrow \phi$. The set $R(v)$ is thus the set of vertices with tentative parent pointers that might or might not connect them to the remaining part of $T(v)$. After these operations, the edge (u,w) is removed from the graph, i.e., from the lists $E_{out}(u)$ and $E_{in}(w)$.

For every vertex v, the algorithm now needs to check the tentative pointers of the vertices in $R(v)$. While there is a vertex $w \in R(v)$, the algorithm scans the edges of $E_{in}(w)$, starting from the edge pointed to by $p(v,w)$, until an edge (u',w) for which $p(v,u') \neq null$ is found, or until the list $E_{in}(w)$ is exhausted. If such an edge is found, then w is removed from $R(v)$. If the list $E_{in}(w)$ is exhausted before finding such an edge, the algorithm sets $p(v,w)$ to *null* and $M(v,w)$ to 0. It then scans all

the outgoing edges of w. If (w, w') is a tree edge, then it adds w' to $R(v)$.

As shown by Italiano [14], the algorithm sketched previously correctly maintains the transitive closure of an *acyclic* graph that undergoes a sequence of edge deletions, and its total running time is $O(mn)$. To see that the total running time of the algorithm is indeed $O(mn)$, note that the lists $E_{in}(u)$ and $E_{out}(u)$ are examined only once per reachability tree.

The algorithm of Frigioni et al. [8] maintains the strongly connected components of the graph, and the *skeleton* of the graph, i.e., the acyclic graph induced on the strongly connected components. The skeleton is maintained using Italiano's algorithm [14].

For each SCC, the algorithm of Frigioni et al. [8] 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 to check whether the SCC decomposed. As this may happen every time an edge is deleted, the worst case total running time of the algorithm may be $\Omega(m^2)$.

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. This proves Theorem 3.1.

To provide a proof of Theorem 3.2 we need to sketch the operation of the algorithm of Frigioni et al. [8] in more detail. The algorithm of Frigioni et al. [8] is similar to the algorithm of Italiano [14], with SCCs playing the role played by vertices in the algorithm of Italiano. Some of the details, however, are slightly more involved. A sketch of (a variant of) the algorithm of Frigioni et al. [8] follows.

Every vertex u has a pointer $C(u)$ to the component containing it. For every component C, the algorithm maintains three linked lists $E_{in}(C)$, $E_{out}(C)$, and $E_{int}(C)$ of the incoming, outgoing, and the internal edges of the component C. An edge (u, w) belongs to $E_{in}(C)$ if $u \notin C$ while $w \in C$, to $E_{out}(C)$ if $u \in C$ while $w \notin C$, and to $E_{int}(C)$ if $u, w \in C$. For every component C the algorithm maintains a tree $T(C)$ of all the components reachable from C. If $C_2 \in T(C_1)$, then we let $p(C_1, C_2)$ be a pointer to the edge (u', u) in $E_{in}(C_2)$ such that $C(u')$ is the parent of $C(u) = C_2$ in $T(C_1)$. If $C_2 \notin T(C_1)$, then $p(C_1, C_2) = null$.

The algorithm handles the deletion of a set of edges E' in the following way. First it lets $E' = E'_{int} \cup E'_{ext}$, where E'_{int} are edges that connect two vertices that are in the same component, while E'_{ext} are edges that connect vertices in two different components. The algorithm first removes all the edges of E'_{int} and then all the edges of E'_{ext}.

The first step is the computation of the new SCCs. This can be done *deterministically* in $O(m + n)$ time by simply recomputing the SCCs from scratch. (To get an algorithm that satisfies the conditions of Theorem 3.1 we replace this step with the randomized algorithm of the previous section.) Suppose that a component C breaks into k new components $C_1, C_2, \ldots, C_k$. The first step is to split the lists $E_{in}(C)$, $E_{out}(C)$, and $E_{int}(C)$ into new lists $E_{in}(C_i)$, $E_{out}(C_i)$, and $E_{int}(C_i)$, for $1 \leq i \leq k$, and to replace the pointers $p(D, C)$, for every component D by new pointers $p(D, C_i)$. This step also constructs for each component D a set $R(D)$ of the components that need to be reconnected, if possible, to $T(D)$. We also initialize new reachability trees for the new components $C_1, C_2, \ldots, C_k$.

The incoming edges of C are now split between the new components $C_1, C_2, \ldots, C_k$. The splitting of $E_{in}(C)$ is done as follows. We scan the edges $E_{in}(C)$ and move each

edge (u, w), where $w \in C_i$, to the list $E_{in}(C_i)$. If $p(D, C) = (u, w)$, then we let $p(D, C_i) = (u, w)$, while $p(D, C_j)$, for each $j \neq i$, is set to the next edge to be added to $E_{in}(C_j)$. In the latter case, we also add C_j to $R(D)$, as C_j lost its link to $T(D)$. (Note that $p(D, C_j)$ now is not the edge connecting C_j to $T(D)$ but rather the first edge that should be checked.) The list $E_{out}(C)$ is split in a similar manner. Finally, each edge $(u, w) \in E_{int}(C)$ with $u \in C_i$ and $w \in C_j$ is moved into $E_{int}(C_i)$, if $i = j$, and to $E_{out}(C_i)$ and $E_{in}(C_j)$, otherwise. It is important to note that the edges of $E_{int}(C)$ are placed at the end of the lists $E_{out}(C_i)$ and $E_{in}(C_j)$. We can now remove the edges of E'_{int} from the graph.

We now deal with the deletion of the edges of E'_{ext}. Suppose that $(u, w) \in E'_{ext}$. If $p(D, C) = (u, w)$, we move $p(D, C)$ to point to the next edge in $E_{in}(C)$, or to null if there is no such edge, and add C to $R(D)$.

Finally, we try to repair the trees. For every component D, while $R(D)$ is not empty, we choose $C \in R(D)$ and scan the edges of $E_{in}(C)$, starting from $p(D, C)$, until an edge (u, w) for which $p(D, C(u)) \neq null$ is found. If such an edge is found, we let $p(D, C)$ point to this edge and remove C from $R(D)$. Otherwise, we find all the components C' for which $p(D, C') = (w, v)$ with $w \in C$ and add them to $R(D)$.

This completes the sketch of (a variant of the) algorithm of Frigioni et al. [8]. Frigioni et al. [8] show that the algorithm correctly maintains the transitive closure matrix of the graph. It is easy to check that the worst case total running time of the algorithm is indeed $O(mn + del \cdot m)$, where *del* is the number of delete operations performed. This completes the proof of Theorem 3.2.

4. Fully dynamic reachability algorithms.

4.1. The first fully dynamic algorithm. Our first fully dynamic reachability algorithm is given in Figure 4.1. 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] described in the previous section.

The algorithm works in *phases*. In the beginning of each phase, a decremental reachability data structure is initialized. We let S be the set of vertices that were centers of insertions during the phase. Initially $S = \phi$. When a set of edges E_v, all 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 insertion centers, 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 $v \in Out(w)$. If such a vertex w exists, then the answer is "yes."

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, then 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

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. Let $E \leftarrow E - E'$.
2. Delete E' from the decremental data structure.
3. For every $w \in S$, rebuild the trees $In(w)$ and $Out(w)$.

***insert*(E_v)**:
1. Let $E \leftarrow E \cup E_v$.
2. Let $S \leftarrow S \cup \{v\}$.
3. If $|S| > t$, then call *init*.
4. Otherwise, construct the trees $In(v)$ and $Out(v)$.

FIG. 4.1. *The first fully dynamic reachability algorithm for general graphs.*

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 the following theorem.

THEOREM 4.1. *For any $t \leq \sqrt{n}$, the algorithm of Figure* 4.1 *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 be constructed.

The total expected amortized cost per insert or delete operation 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). We can get a deterministic version of the algorithm, with the same time bounds, by using the variant of the decremental algorithm of Frigioni et al. [8] described in section 3. □

4.2. The second fully dynamic algorithm. Our second fully dynamic reachability algorithm is given in Figure 4.2. 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].

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. $E \leftarrow E - E'$.
2. Delete E' from the decremental data structure.
3. 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.
4. Call $build(S)$.

***insert*(E_v)**:

1. $E \cup E \cup E_v$.
2. Let $S \leftarrow S \cup \{v\}$.
3. If $|S| > 2t$, then call *init*.
4. 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.
5. Call $build(S)$.

FIG. 4.2. *The second fully dynamic reachability algorithm for general graphs.*

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 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 [27].

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 dynamic 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. We note in passing that similar ideas are also used by Zwick [28] and King [15].

THEOREM 4.4. *For any $t \le (m \log n)^{1/\omega}$, the algorithm of Figure 4.1 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. 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 $|S| \le 2t$, 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

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.

***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 $v \in Out(w_2)$.

***delete*(E')**:

1. $E \cup E - E'$.
2. Delete E' from the decremental data structure.
3. 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.
4. Update the transitive closure B^* of the graph H after the removal of edges from F.

***insert*(E_v)**:

1. $E \cup E \cup E_v$.
2. Let $S \leftarrow S \cup \{v\}$.
3. If $|S| > 2t$, then call *init*.
4. 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.
5. Update the transitive closure B^* of the graph H after the addition of v to it.

FIG. 4.3. *A third fully dynamic reachability algorithm for general graphs.*

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 4.3. 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 [5], to maintain the matrix B^*. We now claim the following theorem.

THEOREM 4.5. *For any $1 \leq t \leq \sqrt{m}$, the algorithm of Figure 4.3 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 5.1. 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.

***init*(*V*)**: • For every $v \in V$ construct reachability trees $In(v)$ and $Out(v)$ and initialize appropriate decremental data structures for them.
***query*(*u*, *v*)**: • For every $w \in V$, check whether $u \in In(w)$ and $v \in Out(w)$.
***delete*(*E′*)**: • Delete E' from all reachability trees and update each one of them using the decremental single-source reachability algorithm for DAGs.
***insert*(E_v)**: • Call $init(\{v\})$.

FIG. 5.1. *A very simple dynamic reachability algorithm for acyclic graphs.*

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 of these trees can be *individually* maintained in $O(m)$ total time. Our algorithm exploits this fact.

THEOREM 5.1. *The algorithm of Figure 5.1 handles each insert operation, which 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 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 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. □

6. Dynamic estimation of the size of reachability sets. Cohen [2] 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 [2] 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 [2] for exact details.

Here we make 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 semidynamic setting, to obtain more efficient algorithms as there is only a logarithmic number of trees to which we save reachability information. The cost of maintaining a reachability tree while edges are added to the graph is $O(m)$. Thus, this leads to an incremental algorithm whose total running time is $O(m \log n + q)$. The cost of maintaining a reachability tree while edges are removed from a directed acyclic graph is $O(m)$; thus, this leads to a decremental algorithm whose total running time is $O(m \log n + q)$ in directed acyclic graphs.

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, de Lichtenberg, and Thorup [12]).

Many open problems still remain. Among them are the following.

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 reachability tree, from a *single* source in a general directed graph whose total running time is $O(mn)$? (Note that the decremental maintenance of a single-source shortest paths tree seems to be a harder task than just maintaining a reachability tree. See [23].)
3. Is there a *deterministic* decremental algorithm for maintaining the transitive closure of a general directed graph whose total running time is $O(mn)$?

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 Symposium on Theory of Computing, Montréal, QC, Canada, 2002, pp. 117–123.

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

[3] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed., MIT Press, Cambridge, MA, 2001.

[5] C. Demetrescu and G. F. Italiano, *Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier*, in Proceedings of the 41st Annual Symposium on Foundations of Computer Science, Redondo Beach, CA, 2000, pp. 381–389.

[6] C. Demetrescu and G. F. Italiano, *Trade-offs for fully dynamic transitive closure on DAGs: Breaking through the $O(n^2)$ barrier*, J. ACM, 52 (2005), pp. 147–156.

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

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

[9] H. N. Gabow, *Path-based depth-first search for strong and biconnected components*, Inform. Process. Lett., 74 (2000), pp. 107–114.

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

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

[12] J. Holm, K. de Lichtenberg, and M. Thorup, *Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity*, J. ACM, 48 (2001), pp. 723–760.

[13] X. Huang and V. Y. Pan, *Fast rectangular matrix multiplications and applications*, J. Complexity, 14 (1998), pp. 257–299.

[14] G. F. Italiano, *Finding paths and deleting edges in directed acyclic graphs*, Inform. Process. Lett., 28 (1988), pp. 5–11.

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

[16] V. King and G. Sagert, *A fully dynamic algorithm for maintaining the transitive closure*, J. Comput. System Sci., 65 (2002), pp. 150–167.

[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 Conference, COCOON, Guilin, China, 2001, pp. 269–277.

[18] I. Krommidas and C. D. Zaroliagis, *An experimental study of algorithms for fully dynamic transitive closure*, in Lecture Notes in Comput. Sci. 3669, Springer, Berlin, 2005, pp. 544–555.

[19] J. A. La Poutré and J. van Leeuwen, *Maintenance of transitive closure and transitive reduction of graphs*, in Proc. of the 13th WG, 1987.

[20] L. Roditty, *A faster and simpler fully dynamic transitive closure*, in Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, 2003, pp. 404–412.

[21] L. Roditty and U. Zwick, *Improved dynamic reachability algorithms for directed graphs*, in Proceedings of the 43rd Annual Symposium on Foundations of Computer Science, Vancouver, BC, Canada, 2002, pp. 679–688.

[22] L. Roditty and U. Zwick, *A fully dynamic reachability algorithm for directed graphs with an*

almost linear update time, in Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, 2004, pp. 184–191.

[23] L. Roditty and U. Zwick, *On dynamic shortest paths problems*, in Proceedings of the 12th Annual European Symposium on Algorithms, Bergen, Norway, 2004, pp. 580–591.

[24] P. Sankowski, *Dynamic transitive closure via dynamic matrix inverse (extended abstract)*, in Proceedings of the 45th Annual Symposium on Foundations of Computer Science, Rome, Italy, 2004, pp. 509–517.

[25] M. Sharir, *A strong-connectivity algorithm and its application in data flow analysis*, Comput. Math. Appl., 7 (1981), pp. 67–72.

[26] R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

[27] J. D. Ullman and M. Yannakakis, *High-probability parallel transitive-closure algorithms*, SIAM J. Comput., 20 (1991), pp. 100–125.

[28] U. Zwick, *All pairs shortest paths using bridging sets and rectangular matrix multiplication*, J. ACM, 49 (2002), pp. 289–317.