A faster and simpler fully dynamic transitive closure

Liam Roditty *

Abstract

We obtain a new fully dynamic algorithm for maintaining the transitive closure of a directed graph. Our algorithm maintains the transitive closure matrix in a total running time of $O(mn + (ins + del) \cdot n^2)$, where *ins* (*del*) is the number of insert (delete) operations performed. Here n is the number of vertices in the graph and m is the *initial* number of edges in the graph. Obviously, reachability queries can be answered in constant time. The space required by the algorithm is $O(n^2)$. Our algorithm can also support path queries. If v is reachable from u, the algorithm can produce a path from u to v in time proportional to the length of the path. The best previously known algorithm for the problem is due to Demetrescu and Italiano [3]. Their algorithm has total running time of $O(n^3 + (ins + del) \cdot n^2)$. The query time is also constant. We also present an algorithm for directed acyclic graphs (DAGs) with a total running time of $O(mn + ins \cdot n^2 + del)$. Our algorithms are obtained by combining some new ideas with techniques of Italiano [7], King [8], King and Thorup [10] and Frigioni *et al.* [4]. We also note that our algorithms are extremely simple and can be easily implemented.

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 [3], improving an algorithm of King [8], obtained recently an algorithm for dynamically maintaining the transitive closure under a sequence of edge insertions and deletions with a total running time of $O(n^3 + (ins + del) \cdot n^2)$, where n is the number of vertices in the graph and *ins* (*del*) is the number of insert (delete) operations performed. King and Thorup [10] 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.

We present an algorithm that maintains the transitive closure matrix with a total running time of $O(mn + (ins + del) \cdot n^2)$. Here n is the number of vertices in the graph and m is the *initial* number of edges in the graph. 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. The time needed for computing the transitive closure of a graph using the best available algorithm that does not resort to fast matrix multiplication, is $\Omega(mn)$. Thus, if an explicit transitive closure matrix is to be maintained, our algorithm, with a total running time of $O(mn + (ins + del) \cdot n^2)$, is essentially optimal. For directed acyclic graphs (DAGs), we obtain an even better result. We present a very simple algorithm whose total running time is $O(mn + ins \cdot n^2 + del)$. In this algorithm, the amortized cost of each delete operation is covered by preceding insert operations. Our algorithms are also essentially optimal in terms of space usage. In particular, both algorithms use only $O(n^2)$ space, even when path queries are supported.

A fully dynamic transitive closure algorithm supports the following operations:

- Insert(E_u, u): Insert a set of edges all incident to the vertex u. We refer to u as the *insertion center.*
- Delete(E'): Delete an arbitrary set of edges from the graph.
- Query(u, v): Is the vertex v reachable from u?

A decremental (incremental) algorithm is an algorithm that can handle only deletions (insertions). Many *partially* dynamic algorithms were developed. Roditty and Zwick [12], improving an algorithm of Baswana *et al.* [1], obtained recently a decremental algorithm for *general* directed graphs that processes any sequence of edge deletions in $O(mn)$ *total* expected time, essentially the time needed for computing the transitive closure of the initial graph. Italiano [6] and independently La Poutré and van Leeuwen [11] obtained an incremental algorithm for *general* directed graphs with an amortized time of $O(n)$ per edge insertion.

A comparison of our fully dynamic transitive closure algorithms and the previously available ones is given in Table 1. We denote by *ins* (*del*) the number of insert (delete) operations performed. Note that we only consider here algorithms that explicitly maintain

*School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: `liamr@post.tau.ac.il`.

Graph	Type of algorithm	Path query	Total running time	Reference
DAGs	Monte Carlo	No	$O(n^3 + (ins + del) \cdot n^2)$	[9]
DAGs	**Deterministic**	**Yes**	$O(mn + ins \cdot n^2 + del)$	**This paper**
General	Monte Carlo	No	$O(n^{3.26} + (ins + del) \cdot n^{2.26})$	[9]
General	Deterministic	Yes	$O(n^3 \log n + ins \cdot n^2 \log n + del)$	[8]
General	Deterministic	No	$O(n^3 + (ins + del) \cdot n^2)$	[3]
General	**Deterministic**	**Yes**	$O(mn + (ins + del) \cdot n^2)$	**This paper**

Table 1: Fully dynamic transitive closure algorithms.

the transitive closure matrix, and thus have a constant time query. We refer the reader to [12] for details on reachability algorithms with a non-constant query time.

King and Sagert [9] gave the first algorithm whose update time is faster than computing the transitive closure from scratch. Their algorithm counts the number of different paths between each two vertices. Obviously, this counting technique can only work for DAGs. In this case the total running time is $O(n^3 + (ins + del) \cdot n^2)$. They extended this technique to general graphs by decomposing the graph into its strongly connected components (SCCs). In the general case, the total running time of their algorithm is $O(n^{3.26} + (ins + del) \cdot n^{2.26})$. Their algorithm is a Monte-Carlo algorithm. The number of paths between two vertices can be exponential in n. To reduce the word size their algorithm is randomized with one-sided error. The general algorithm uses rectangular matrix multiplication.

Next, King [8] presented a *deterministic* algorithm that substantially improved the randomized algorithm of King and Sagert [9]. She presented a general framework for the fully dynamic transitive closure and also for the fully dynamic all pairs shortest paths problem. Her framework is composed from two ingredients. A decremental algorithm that maintains the 'old' paths of the initial graph and an algorithm that maintains a forest of in-trees and out-trees around each vertex that served as an insertion center. These trees are updated after each deletion. The total running time of the algorithm is $O(n^3 \log n + ins \cdot n^2 \log n + del)$.

Demetrescu and Italiano [3], improving the algorithm of King [8], obtained recently an algorithm with a total running time of $O(n^3 + (ins + del) \cdot n^2)$. Their algorithm uses a general framework of dynamic evaluation of polynomials over matrices.

We introduce a new concept named *dynamic blocks.* The dynamic blocks are a relaxed version of the strongly connected component of dynamic graphs. Using this new concept we improve the total running time of all previously known algorithms. Our algorithm has a total running time of $O(mn + (ins + del) \cdot n^2)$. As mentioned above, for explicitly maintaining the transitive closure matrix, this is essentially the best algorithm possible. Our algorithm uses very simple techniques. We use the framework of King [8] with more efficient ingredients. For the maintenance of the initial graph through a sequence of edge deletions we use a variant of the algorithm of Frigioni *et al.* [4] whose total running time is $O(mn + del \cdot m)$. For the maintenance of a forest of in-trees and out-trees we introduce a new algorithm with an initial cost of $O(n^2)$ for each new tree and an amortized cost of $O(n^2)$ for the deletion of an arbitrary set of edges from the whole forest. The total running time of our algorithm on a forest of $O(n)$ trees is $O((ins + del) \cdot n^2)$.

The rest of this extended abstract is organized as follows. In the next section we present a new decremental algorithm for maintaining a tree of the SCCs reachable from a given vertex. A modification of this algorithm is used in Section 3 to obtain an algorithm for the maintenance of a forest of in-trees and out-trees whose total running time is $O((ins + del) \cdot n^2)$. In Section 4 we obtain our new fully dynamic transitive closure algorithm for general graphs using King's framework with our forest algorithm and the variant of the algorithm of Frigioni *et al.* [4]. In Section 5 we show how to support path queries without increasing the space requirements. We end in Section 6 with some concluding remarks and open problems.

2 Reachability tree of strongly connected components

In this section we consider the dynamic maintenance of a reachability tree from an arbitrary vertex in a general directed graph under a sequence of edge deletions. The algorithm that we present is slower then building the reachability tree each time from scratch using BFS or DFS. This algorithm is presented here in an attempt to describe our ideas in the simplest possible setting. In Section 3 we use this algorithm as one of the ingredients of our fully dynamic algorithm.

We start with the following simple lemma:

```
InitTree(u):
1. Allocate a Tree structure and its arrays
2. using Tree.
3.     elem ← GetSCC(G), reach[1 : |elem|] ←'*' and initialize at using elem
4.     Call BuildMatrix(Tree) and then ReconnectTree(Tree, u, 1)

BuildMatrix(Tree):
1. using Tree.
2.     M[1:n, 1:n] ← A, m ← n
3.     for i ← 1 to 2n
4.         if elem[i] ≠ NULL then
5.             m ← m + 1 and col[i] ← m
6.             for v ← 1 to n do M[v, m] ← $\min_{x \in elem[i]} M[v, x]$

ReconnectTree(Tree, u, t):
1. using Tree.
2.     for i ← 1 to 2n do if reach[i] = 1 then reach[i] ← '*'
3.     reach[at[u]] ← 1
4.     for i ← 1 to 2n do if reach[i] ='*' then Reconnect(Tree, i, t)

Reconnect(Tree, i, t):
1. using Tree.
2.     while row[i] ≤ n do
3.         if M[row[i], col[i]] ≤ t and row[i] ∉ elem[i] then
4.             if reach[at[row[i]]] ='*' then Reconnect(Tree, at[row[i]], t)
5.             if reach[at[row[i]]] = 1 then reach[i] ← 1; return
6.         row[i] ← row[i] + 1
7.     reach[i] ← 0

Delete(E'):
1. Update A with E' and elem' ← GetSCC(G)
2. using Tree.
3.     for i ← 1 to 2n
4.         if elem[i] ≠ NULL and elem'[i] = NULL then reach[i] ← 0
5.         if elem[i] = NULL and elem'[i] ≠ NULL then reach[i] ←'*'
6.     elem ← elem', call BuildMatrix(Tree) and then ReconnectTree(Tree, u, 1)
```

Figure 1: A decremental algorithm for maintaining a tree of SCCs reachable from u.

LEMMA 2.1. *Let $G = (V, E)$ be a directed graph that undergoes a sequence of edge deletions. The number of different SCCs during all the deletion process is at most $2n - 1$.*

Proof. We prove it by building a forest whose vertices are the SCCs of the graph G during all the deletion process. Each decomposed SCC is connected to the SCCs that were created from it by the deletion. In this forest, each non-leaf vertex has at least two children. As there are at most n leafs, the total number of vertices is at most $2n - 1$. □

The new algorithm is given in Figure 1. The algorithm maintains the SCCs of the graph that are reachable form the vertex u. It handles any sequence of edge deletions in $O(n^2 + del \cdot n^2)$ total running time, where del is the number of delete operations. Each query is answered correctly in $O(1)$ worst-case time. This algorithm uses a similar approach to the one used by King and Thorup [10] to save space.

We denote by A the adjacency matrix of the graph, where $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = \infty$ otherwise. We denote by M an adjacency matrix of size $n \times (n + |S|)$, where S is the set of the SCCs of the current graph. The rows of the matrix are the vertices of the graph. The first n columns are the vertices of the graph and the rest of the columns correspond to the current SCCs of the graph. We set $M[1{:}n, 1{:}n]$ to A. Each column j, where $j > n$ is associated with a SCC.

The entry $M[u, j]$ is set to 1 if there is an edge from u to a vertex in the SCC corresponding to the j-th column, otherwise it is set to ∞.

In order to maintain the tree information we define a structure named *Tree* that contains the following arrays. An array named *elem* that holds for each SCC a pointer to a list of its vertices. An array named *at* that holds for each vertex the index of the SCC that contains it. An array named *col* is used to associate each SCC to the right column in the matrix M. An array named *row* is used to hold the tail of the edge that connects each SCC to the reachability tree. An array named *reach* is used to hold for each SCC if it is reachable from u. By Lemma 2.1, the total number of different *SCCs* in any sequence of edge deletions is at most $2n - 1$, thus all the arrays in use are of size $2n$.

Note that for each SCC i we can check whether it still exists, i.e., was not decomposed, using the array *elem*. If it still exists, then *elem*[i] points to a list containing the vertices of the SCC, otherwise *elem*[i] is *NULL*. If *reach*[i] = 1 then SCC i is reachable from u. After a delete operation we can access the matrix entry that connects i to u using *row*[i] and *col*[i] to check whether the connecting edge is still present in the graph.

As mentioned above the *Tree* structure holds these arrays. In our algorithm description we use the line 'using Tree.' when we like to access the arrays directly.

The algorithm starts by calling *InitTree*(u) to initialize all the arrays mentioned above. The SCCs of the graph are computed using any linear time algorithm (see Tarjan [14], Sharir [13], Gabow [5], or Chapter 22 of Cormen *et al.* [2]). We assume that *GetSCC*(G) builds the array *elem* as defined above. We also assume that in the first call to *GetSCC*(G) the SCCs are assigned consecutive numbers starting from 1. The SCCs are marked in the array *reach* with '$*$' to point that the algorithm needs to check whether they are reachable from u. Next, it builds the matrix M using the current set of SCCs of the graph, by calling *BuildMatrix*(*Tree*). Finally, in order to create an implicit tree of SCCs rooted at *at*[u], *ReconnectTree* is called. We note that the last parameter of *ReconnectTree* will be used in Section 3.

Edge deletions are handled in a very similar way. First the set of edges E' is removed from the graph G. Then, the matrix M is rebuilt and the tree is reconnected. We now claim:

THEOREM 2.1. *The algorithm handles the initialization process and a sequence of del deletions in a total worst-case time of* $O(n^2 + del \cdot n^2)$.

Proof. We analyze the cost of the procedure *BuildMatrix* and *ReconnectTree*, which are the most time consuming steps. We show that the *total* cost of a sequence of one initialization and *del* deletions is at most $O(n^2 + del \cdot n^2)$. We first analyze the cost of the procedure *BuildMatrix*. By Lemma 2.1 the number of columns is $O(n)$. Each column participates in the creation of at most one other column, thus the total cost of building the matrix is $O(del \cdot n^2)$. Next, we show that the total time spent in *ReconnectTree* is $O(n^2 + del \cdot n)$. It is clear that the total time spent in reconnecting the tree is equal to the number of access operations performed to the matrix M. For the SCC i the entry [v, *col*[i]] is first accessed when *row*[i] is advanced from $v - 1$ to v. If this entry connects i to the SCCs reachability tree of u then after each deletion this entry is checked to see whether it is still a valid connection to the reachability tree of u. The first and the last accesses are charged to the creation of the tree. After advancing *row*[i] from v to $v + 1$ the entry [v, *col*[i]] is never accessed again. Thus, the total number of first and last accesses to all the cells is $O(n^2)$. Each access between the first and the last access is done only once in each deletion. Thus, the total time spent in the tree in checking whether an entry is still a valid connection to the reachability tree of u is $O(del \cdot n)$. In each deletion we calculate the SCCs of the graph. Thus the total cost of a sequence of one initialization operation and *del* deletions is at most $O(n^2 + del \cdot n^2)$. □

The vertex set used by the algorithm is the set of the SCCs of the graph, thus the set of edges considered by the algorithm does not contain any cycle. If an entry does not connect a SCC to u, then this entry is never examined again. This method was first suggested by La Poutré and van Leeuven [11] and independently by Italiano [7] for the decremental maintenance of the transitive closure of DAGs. It was extended by La Poutré and van Leeuven [11] and later by Frigioni *et al.* [4] to general graphs by using the SCCs graph.

Next, we show that the algorithm maintains correctly the reachability tree of u. We show that the procedure *ReconnectTree* finds correctly all the SCCs in S that are reachable from u.

THEOREM 2.2. *Let i be a SCC. When ReconnectTree terminates, reach*[i] *is set to* 1 *if and only if i is reachable from u.*

Proof. First note that *ReconnectTree* terminates as the graph of SCCs is acyclic. We prove that *ReconnectTree* finds a path from u to i if and only if there is one. The proof is by induction on the number of SCCs on a path from u to i. The basis of the induction is easily established by setting *reach*[*at*[*row*[i]]] to 1 in line 3. If there is a path from u to i, then there exist an entry [v, *col*[i]], where *at*[v] = i' and i' is reachable from u. Unless the SCC i is connected to the tree, at some stage, by a different path, *row*[i] is advanced and eventually *row*[i] = v. The path from u to i' is shorter than the path from u to i, thus by the induction hypothesis the recursive call to *Reconnect*$(Tree, i', t)$ connects i' to the tree in line 3 of *Reconnect* and in line 4 *reach*$[i]$ is set to 1. □

3 A forest of block trees

In this section we extend the previous algorithm to a fully dynamic transitive closure algorithm where the initial graph is empty. We allow the insertion of a set of edges, all touching the same vertex. We refer to such a vertex as the *center* of the insertion. An in-tree (all the vertices that reach the center) and an out-tree (all the vertices that are reachable from the center) are formed around the center and a cost of $O(n^2)$ is charged to the insertion for all future deletions from these two trees. The total time for maintaining such a forest using the algorithm of the previous section is $O((ins + del) \cdot n^2)$, where ins (del) is the number of the insertions (deletions). In the next Section we use the framework of King [8] with our forest as one of its ingredients to obtain a fully dynamic transitive closure algorithm for any graph (not necessarily empty) with a total running time of $O(mn + (ins + del) \cdot n^2)$.

The main idea of the algorithm of this section is that the same adjacency matrix can serve as the adjacency matrix for all the trees in the forest. We show that if the vertex sets of the trees of the forest are maintained in a certain way, then the size of the adjacency matrix is at most $O(n^2)$ and the construction of this matrix can be done in $O(n^2)$ time.

We now define the insertion index of each edge and insertion center.

DEFINITION 3.1. (INSERTION INDEX) *Consider an insertion of a set E_u of edges, centered at u. Let k be the number of vertices, other than u, that served as insertion centers before this insert operation, but not after it. The insertion index of the set E_u and the center u is set to $k + 1$. If the initial graph is not empty, the index of its edges is set to* 0.

Note that an insert operation may change the insertion index of previously inserted edges and previous insertion centers. The insertion indices are maintained using an array of size n named *Index* and a counter k initialized to 0. If an insertion is performed around the vertex u for the first time, then $k \leftarrow k+1$. If u is already assigned an insertion index i, then the insertion index of each edge and center with index between $i + 1$ and k is decremented by 1 and k remains unchanged. In both cases the insertion index of the edges of E_u is set to k and *Index*$[k]$ is set to u.

Next, we extend the definition of the adjacency matrix of the graph using the insertion index of each edge.

DEFINITION 3.2. (INDEXED ADJACENCY MATRIX) *Let $G = (V, E)$ be a directed graph. Its indexed adjacency matrix A is defined as follows: If $(u, v) \in E$ then $A[u, v]$ is the insertion index of the edge (u, v). If $(u, v) \notin E$ then $A[u, v] = \infty$.*

Let $\mathcal{S}$ be the set of SCCs of the graph G after the last update (deletion or insertion). We define $\mathcal{S}_0$ to be a set whose elements are $\{v\}$, for each $v \in V$. Using these sets we define *dynamic blocks* for each insertion index as follows.

DEFINITION 3.3. (DYNAMIC BLOCKS) *Two vertices v_1, v_2 belong to the same block, with respect to u, if and only if v_1 and v_2 were in the same SCC of the graph after the last insertion operation centered at u, and after every subsequent delete operation. Let $\mathcal{B}_i$ be the set of blocks with respect to the vertex u with insertion index i, where $1 \leq i \leq k$ and let $\mathcal{B}_0 = \mathcal{S}_0$.*

Note that blocks are dynamic in nature, as they may change after each update operation. (An insert operation may only change the blocks with respect to the last insertion center). Next, we prove that the dynamic blocks satisfy the following *containment* property.

LEMMA 3.1. (CONTAINMENT PROPERTY) *Each block from the block set $\mathcal{B}_i$ contains one or more blocks from the block set $\mathcal{B}_{i-1}$.*

Proof. For each $B_i \in \mathcal{B}_i$ and $v \in B_i$ there exists $B_{i-1} \in \mathcal{B}_{i-1}$ such that $v \in B_{i-1}$. We prove that $B_{i-1} \subseteq B_i$. Assume on the contrary that there is a vertex v' such that $v' \in B_{i-1}$ but $v' \notin B_i$. By the definition of dynamic blocks if v and v' are not in the same block of $\mathcal{B}_i$, then there was a delete operation after the insertion corresponding to index $i - 1$ that placed v and v' in two different SCCs. By the same definition if v and v' were placed in different SCCs after the insertion corresponding to index $i - 1$, then v and v' can not be in the same block of $\mathcal{B}_{i-1}$. □

Using Lemma 3.1 we prove the following theorem.

THEOREM 3.1. *If $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_k$ are the dynamic blocks of a directed graph with n vertices that undergoes a sequence of edge insertions and deletions then $|\cup \mathcal{B}_i| \leq 2n - 1$.*

Proof. We build a forest and we show that its size is at most $2n - 1$. Let $|\cup \mathcal{B}_i|$ be the set of vertices of the forest. (Note that if $B \in \mathcal{B}_i$ and $B \in \mathcal{B}_{i-1}$, B appears as a vertex only once). If we connect each block in level j to the two or more blocks in level $i < j$ that it contains, we get a forest in which each non-leaf vertex has at least two children. As there are at most n leafs we get by Lemma 3.1 that $|\cup \mathcal{B}_i| \leq 2n - 1$. □

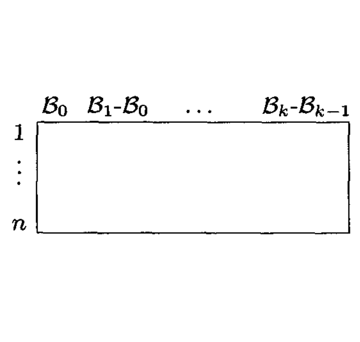
```
InitForest:
 1. $k \leftarrow 0$, allocate an array $Index$ of size $n$

BuildMatrix(Forest):
 1. $M[1{:}n, 1{:}n] \leftarrow A$, $m \leftarrow n$
 2. for $i \leftarrow 1$ to $k$
 3.    using $Forest[i]$.
 4.      for $j \leftarrow 1$ to $2n$
 5.         if $|subblock[j]| = 1$ then $col[j] \leftarrow Forest[i-1].col[subblock[j][1]]$
 6.         if $|subblock[j]| > 1$ then
 7.            $m \leftarrow m + 1$, $col[j] \leftarrow m$
 8.            for $v \leftarrow 1$ to $n$
 9.               $M[v, m] \leftarrow \min_{b \in subblock[j]} M[v, Forest[i-1].col[b]]$

InsertF(E_u, u):
 1. if $\exists i$ such that $Index[i] = u$ then
 2.    for $j \leftarrow i$ to $k - 1$
 3.       $Index[j] \leftarrow Index[j+1]$, $Forest[j] \leftarrow Forest[j+1]$
 4. else
 5.    $k \leftarrow k + 1$ and add $Forest[k]$
 6. update arrays $sublock$ and $elem$ of $Forest[k]$ using $GetSCC(G)$, $Index[k] \leftarrow u$
 7. $BuildMatrix(Forest)$
 8. $ReconnectTree(Forest[k], Index[k], k)$

DeleteF(E'):
 1. Update $A$ with $E'$ and $\mathcal{S} \leftarrow GetSCC(G)$
 2. for $i \leftarrow 1$ to $k$
 3.    $subblock' \leftarrow UpdateBlocks(Forest[i].subblock, \mathcal{S})$
 4. for $i \leftarrow 1$ to $k$
 5.    for $j \leftarrow 1$ to $2n$
 6.       if $Forest[i].subblock[j] \neq NULL$ and $subblock'[j] = NULL$ then $Forest[i].reach[j] \leftarrow 0$
 7.       if $Forest[i].subblock[j] = NULL$ and $subblock'[j] \neq NULL$ then $Forest[i].reach[j] \leftarrow$'*'
 8.    $Forest[i].subblock \leftarrow subblock'$
 9. $BuildMatrix(Forest)$.
10. for $i \leftarrow 1$ to $k$ do $ReconnectTree(Forest[i], Index[i], i)$.
```

Figure 2: Maintaining a forest.

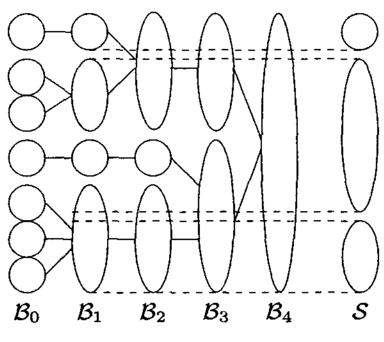
Using these dynamic blocks we maintain a forest of in-trees and out-trees. The algorithm appears in Figure 2. This algorithm is a generalization of the algorithm from Section 2. Instead of working with just one tree and allowing only deletions we allow also insertion of a set of edges around a certain vertex. The algorithm maintains an in-tree and an out-tree around each such insertion center. The vertex set of each tree is the set of dynamic blocks as defined above. The use of dynamic blocks maintains two important properties of the trees. The total number of different blocks in each tree is at most $2n - 1$ and the set of edges with insertion index less than or equal to i does not contain a cycle on the blocks of the set $\mathcal{B}_i$.

The insertions are handled as follow. After each insertion we check whether this is a new insertion center and update the insertion indices accordingly. We then build the matrix M from scratch and a new in-tree and a new out-tree are initialized with $\mathcal{B}_k$ as their vertex set. Note that the insertion index is passed to the reconnection procedure. This allows the connection procedure to consider only edges that were present in the graph when the tree was formed and by that the set of edges considered by the reconnection process does not contain any cycle on the set of the tree blocks. To simplify the presentation we only consider the construction of the out-trees. The in-trees are constructed similarly.

The deletions are handled as follow. After each deletion the set of new SCCs of the graph is recalculated



$\mathcal{B}_0$ $\mathcal{B}_1$-$\mathcal{B}_0$ ... $\mathcal{B}_k$-$\mathcal{B}_{k-1}$
1
n
(a)



$\mathcal{B}_0$ $\mathcal{B}_1$ $\mathcal{B}_2$ $\mathcal{B}_3$ $\mathcal{B}_4$ $\mathcal{S}$
(b)

Figure 3: (a) The forest adjacency matrix M; (b) Maintaining dynamic blocks, where $n = 7$ and $k = 4$.

and the dynamic blocks are updated accordingly. The matrix M is rebuilt with the new blocks and then each tree is reconnected by calling to the reconnection procedure with the tree insertion index.

The adjacency matrix M used by the reconnection process in each tree is composed of columns that are the elements of the sets $\mathcal{B}_0, \mathcal{B}_1 - \mathcal{B}_0, \ldots, \mathcal{B}_k - \mathcal{B}_{k-1}$ and rows which are the vertices of the graph as appears in Figure 3(a). By Theorem 3.1 $|\cup \mathcal{B}_i| \leq 2n - 1$ thus the size of the matrix M is $O(n^2)$.

This algorithm uses an array of size n named *Forest*. Each element of this array is the *Tree* structure from the previous section with some minor changes. The structure contains the same arrays as in Section 2 with an additional array *subblock*. This array holds for each block the blocks from the previous insertion index that compose it as appears in Figure 3(b). The array *elem* holds for each block the vertices that compose it. This change is necessary for the implementation of *BuildMatrix*.

THEOREM 3.2. *The algorithm of Figure 2 maintains the forest with a total running time of* $O((ins + del) \cdot n^2)$.

Proof. The procedure *DeleteF* updates at most n dynamic block sets using *UpdateBlocks*. The update process of the blocks can be done in $O(n)$ time for each block set, thus the total time for updating the dynamic blocks is $O(n^2)$. Next the matrix M is rebuilt. By Theorem 3.1 the number of columns in M is at most $O(n)$. By Lemma 3.1 (the containment property) each column participate in the creation of exactly one other column, thus the building time of the matrix M is at most $O(n^2)$. The reconnection cost is partitioned into two different costs. In each tree the cost of checking whether the last entry that connected a block to the tree is still a valid connection is charged to the deletion and gives a total of $O(n)$ for a tree and $O(n^2)$ for all the trees. The cost of inquiring new entries of the matrix M is charged to the insertion that created each of the trees. Thus, the total time for all the deletions is $O(del \cdot n^2)$ amortized time. The procedure *InsertF* creates an in-tree and an out-tree for the insertion center. Each tree creation is charged with $O(n^2)$ time for all the later inquiries of the matrix entries like in Theorem 2.1. The use of dynamic blocks maintains the property that the total number of new blocks that will be created in a tree during any sequence of deletions is $O(n)$. The total time for all the insertions is $O(ins \cdot n^2)$ time. Thus , the total running time of the algorithm is $O((ins + del) \cdot n^2)$. □

4 Fully dynamic transitive closure

In this section we obtain our new fully dynamic transitive closure algorithm by using the framework suggested by King [8]. King's framework is composed from two ingredients. A decremental algorithm that maintains the 'old' paths of the initial graph and an algorithm that maintains a forest of in-trees and out-trees around each insertion center. To maintain explicitly the transitive closure matrix a matrix denoted by *count* counts all the insertion centers that lie on a path between each pair of vertices. To enable path queries these centers are saved in a list for each pair.

We replace the ingredients of King's framework. For the maintenance of the initial graph through a sequence of edge deletions we use a variant of the algorithm of Frigioni *et al.* [4]. Their algorithm works in a total time of $O(m^2)$, but it is possible to modify it to work in time of $O(mn + del \cdot m)$. (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). Clearly, we use the algorithm from Section 3 for the maintenance of the forest of in-trees and out-trees.

Figure 4 sketches the fully dynamic transitive closure algorithm.

THEOREM 4.1. *The algorithm of Figure 4 handles each insert operation in* $O(n^2)$ *worst-case time and each*

```
Init:
  1. Initialize a decremental reachability data structure and the matrix count.
  2. Call InitForest.
Insert($E_u$, $u$):
  1. Call InsertF($E_u$, $u$) and update the count matrix.
Delete($E'$):
  1. Call DeleteF($E'$) and update the count matrix.
  2. Delete from the 'old' data structure the edges of $E'$ with insertion index 0.
Query($u$, $v$):
  1. If there exists an 'old' path form $u$ to $v$ or $count(u, v) > 0$ return true.
  2. Return false.
```

Figure 4: Fully dynamic transitive closure algorithm.

delete operation in $O(n^2)$ amortized-time. The algorithm answers each query correctly in $O(1)$ worst-case time. The algorithm has a total running time of $O(mn + (ins + del) \cdot n^2)$. The space used by the algorithm is $O(n^2)$.

Proof. By Theorem 3.2 the total time for maintaining the forest is $O(ins \cdot n^2 + del \cdot n^2)$. The total cost of the deletion only data structure is at most $O(mn + del \cdot m)$. Thus, the total time of our algorithm is $O(mn + (ins + del) \cdot n^2)$. □

In the case of DAGs we can obtain a better result. Italiano [7] presented a decremental algorithm for DAGs with a total running time of $O(mn)$ for any sequence of deletions. This algorithm maintains a forest of out-trees from each vertex of the graph. On each such a tree the total time spent by the algorithm is at most $O(m)$. By using King's framework with a forest of in-trees and out-trees maintained by the algorithm of Italiano, we obtain a fully dynamic algorithm for the transitive closure on DAGs whose total running time is $O(mn+ins \cdot n^2+del)$.

5 Supporting path queries

In this section we extend the algorithm to support path queries without increasing the space in use. The technique for that is based on the one used by King and Thorup [10]. King maintains in her framework a list for each pair of vertices. If an insertion center w is on a path from u to v, then w is added to $list(u, v)$. The size of each list may be $O(n)$. Thus, the total space in use may be $O(n^3)$. Using the method King and Thorup [10] presented in the context of fully dynamic APSP it is possible to reduce the total space in use to $O(n^2)$. The main idea is to keep only one list of insertion centers. In this list the insertion centers are ordered by the insertion index. The latest insertion center is placed at the end of the list. Each pair of vertices u and v points to the oldest insertion center w, which is a witness to a path from u to v. If w is no longer a witness to a path from u to v or if there was a new insertion around w and it may no longer be the oldest witness, the pair u and v scans the list for a new witness from the successor of w. This method still keeps the time bound unchanged. When a pair of vertices scans the list each center, which the pair scans, was charged for their scanning in its insertion. Thus, the scanning process is simply paid by the $O(n^2)$ cost of each insertion. This is the way the witnesses are maintained and obtained for each pair of vertices. Clearly, this method uses $O(n^2)$ space. In our implementation the array *Index* maintains the list of the insertion centers. If the pair u and v points to the *i-th* insertion center w which is a witness to a path of length ℓ, then using the in-tree and the out-tree of w this path can be obtained in $O(\ell)$ time. The vertex set of the in-tree and the out-tree of w is the block set $\mathcal{B}_i$. First, a list of blocks indices $b_1, \ldots, b_t$ is obtained such that $u \in \mathcal{B}_i[b_1]$ and $v \in \mathcal{B}_i[b_t]$ and all the blocks are different (unless $b_1 = b_t$). Such a list can be obtained by scanning simultaneity from b_t backwards to w and from b_1 forward to w. For each new block on the path from u to w the algorithm checks whether that block was scanned from v backwards to w. The same is done to each new block on the path from v backwards to w. When a common block is found the algorithm terminates. By this method a list of different blocks from u to v of size t is obtained in $O(t)$. The list of the blocks is composed from two parts. The first part is $b_1, \ldots, b_r$, which was obtained from the in-tree, and the second part is $b_r, \ldots, b_t$, which was obtained from the out-tree. Note that the block b_r is the common block, which stopped the algorithm. By the in-tree and the out-tree of w we can produce the list $x_1, x_2, \ldots, x_r, y_r, \ldots, y_{t-1}, y_t$ of $t + 1$ vertices. The list satisfies that $x_p \in \mathcal{B}_i[b_p]$ and $y_q \in \mathcal{B}_i[b_q]$, where $1 \leq p \leq r$ and $r \leq q \leq t$. Note that $u = x_1$ and $v = y_t$. Each x_p is the head of the edge that connects b_{p-1} to b_p. In a similar way each y_q is the tail of the edge that connects b_q to b_{q+1}. If x_{p-1} and x_{p+1} are in the same SCC, then x_p is also in this SCC. For each SCC of the graph we maintain an in-tree and out-tree from an arbitrary vertex. Each vertex which is the tail (head) of an incoming (outgoing) edge to (from) the SCC is also in the in-tree (out-tree). We build these trees in $O(m)$ after each update. A path from x_p to the first $x_{p'}$ which is not in the SCC of x_p is produced by the trees of the SCC of x_p. This process is repeated till reaching v.

THEOREM 5.1. *The algorithm can produce a path of length ℓ in $O(\ell)$ worst-case time. The required space of the algorithm remains $O(n^2)$.*

6 Concluding remarks and open problems

We presented a fully dynamic algorithm for the maintenance of the transitive closure matrix of a general graph whose total running time is essentially optimal. However, many problems still remain open, among them:

1. Is it possible to extend our simple algorithm for DAGs to general directed graphs while keeping the time bound unchanged and by that to obtain an algorithm that charges *all* the cost of a delete operation to preceding insertions?

2. Is it possible not to use amortization? Or more precisely, is it possible that both the insertion and the deletion time will be $O(n^2)$ *worst-case*.

3. If a non-constant time query is allowed, i.e., the transitive closure matrix is maintained *implicitly*, 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?

Acknowledgment I like to thank Uri Zwick, Vera Asudin and Eyal Even-Dar for their helpful remarks.

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, Québec*, 2002.

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

[3] C. Demetrescu and G.F. 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.

[4] 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.

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

[6] G.F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(3):273–281, 1986.

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

[8] 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.

[9] 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.

[10] 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.

[11] J.A. 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.

[12] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proceedings of the 43th Annual IEEE Symposium on Foundations of Computer Science, Vancouver, Canada*, 2002.

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

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