A fully dynamic reachability algorithm for directed graphs with an almost linear update time *

Liam Roditty † *Uri Zwick* †

February 26, 2004

Abstract

We obtain a new fully dynamic algorithm for the reachability problem in directed graphs. Our algorithm has an amortized update time of $O(m + n \log n)$ and a worst-case query time of $O(n)$, where m is the (current) number of edges in the graph, and n is the number of vertices in the graph. Each update operation either inserts a set of edges that touch the same vertex, or deletes an arbitrary set of edges. The algorithm is deterministic and uses fairly simple data structures. This is the first algorithm that breaks the $O(n^2)$ update barrier for all graphs with $o(n^2)$ edges.

One of the ingredients used by this new algorithm may be interesting in its own right. It is a new dynamic algorithm for *strong* connectivity in directed graphs with an interesting persistency property. Each insert operation creates a new version of the graph. A delete operation deletes edges from *all* versions. Strong connectivity queries can be made on each version of the graph. The algorithm handles each update in $O(m\alpha(m, n))$ amortized time, and each query in $O(1)$ time. Note that the update time of $O(m\alpha(m, n))$, in case of a delete operation, is the time needed for updating *all* versions of the graph.

1 Introduction

The dynamic reachability problem, i.e., the problem of maintaining a directed graph that undergoes a sequence of edge insertions and deletions in a way that facilitates the fast answering of reachability queries, is a well studied and well motivated problem.

Roditty [Rod03] and Demetrescu and Italiano [DI00], improving an algorithm of King [Kin99], obtained recently algorithms for dynamically maintaining the transitive closure of a graph with an amortized insert/delete time of $O(n^2)$, where n is the number of vertices in the graph. These algorithms support *extended* insert and delete operations in which a 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, if the transitive closure of the graph is to be explicitly maintained. When the number of queries after each update operation is relatively small, it is desirable to have a dynamic algorithm with a smaller update time, at the price of a larger non-constant query time. Roditty and Zwick [RZ02], improving results of Henzinger and King [HK95], obtained an algorithm with an $O(m\sqrt{n})$ amortized update time and $O(\sqrt{n})$ query time. Demetrescu and Italiano [DI00] obtained an algorithm with an $O(n^{1.58})$ amortized update time and an $O(n^{0.58})$ query time. Their algorithm works, however, only for directed acyclic graphs (DAGs). It also relies on fast matrix multiplication.

*This is a preliminary version of a paper to appear in STOC'04,

†School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: {liamr,zwick}@post.tau.ac.il.

Type of graphs	Type of algorithm	Amortized update time	Query time	Reference
General	Deterministic	$O(n^2)$	$O(1)$	[DI00, Rod03]
General	Monte Carlo	$O(m\sqrt{n}\log^2 n)$	$O(n/\log n)$	[HK95]
General	Deterministic	$O(m\sqrt{n})$	$O(\sqrt{n})$	[RZ02]
General	Monte Carlo	$O(m^{0.58}n)$	$O(m^{0.43})$	[RZ02]
DAGs	Monte Carlo	$O(n^{1.58})$	$O(n^{0.58})$	[DI00]
General	**Deterministic**	$\boldsymbol{O(m+n\log n)}$	$\boldsymbol{O(n)}$	**This paper**

Table 1: Fully dynamic reachability algorithms.

We present here a deterministic algorithm that exhibits a new update/query trade-off for the dynamic reachability problem. Our new algorithm handles each (extended) update operation in $O(m + n\log n)$ amortized time, and answers each query in $O(n)$ worst-case time. Here m is the number of edges currently in the graph, plus the number of edges to be inserted or deleted, and n is the number of vertices in the graph. This is the first algorithm that breaks the $O(n^2)$ update barrier for all graphs with $o(n^2)$ edges. The new algorithm and some of the previously existing algorithms for the dynamic reachability problem are compared in Table 1. Our algorithm has the fastest update time, among all algorithms that work on all graphs, but alas the slowest query time. It should be noted, however, that a query time of $O(n)$ is still much faster, in most cases, than the trivial query time of $O(m)$.

Our algorithm uses only simple data structures and does not rely on fast matrix multiplication. As the number of edges deleted in a single update operation may be $\Omega(m)$, a delete time of $O(m + n\log n)$ is almost optimal for extended delete operations. Also note that a dynamic reachability algorithm with an amortized time of $O(n^{2-\epsilon})$ for an extended insert operation and a query time of $O(n^{1-\epsilon})$, for some $\epsilon > 0$, yields immediately an $O(n^{3-\epsilon})$ time algorithm for the static transitive closure problem. (Each graph can be constructed using n extended insert operations, and the transitive closure matrix can then be built using n^2 queries.) Obtaining such an algorithm without the use of algebraic matrix multiplication algorithms would be a major breakthrough.

One of the ingredients used by this new algorithm may be interesting in its own right. It is a new dynamic algorithm for maintaining the *strongly* connected components of a directed graphs. This algorithm has an interesting persistency property. Each insert operation creates a new version of the graph. A delete operation deletes edges from *all* versions. Strong connectivity queries, i.e., queries as to whether two vertices u and v are in the same strongly connected component, can be made on each version of the graph. The algorithm handles each update in $O(m\alpha(m,n))$ amortized time, where $\alpha(m,n)$ is a functional inverse of the Ackermann function (see [Tar75]), and answers each query in $O(1)$ time. The update time of $O(m\alpha(m,n))$ is the time needed for updating *all* versions of the graph. The algorithm uses only $O(m+n)$ space, no matter how many versions of the graph are held by the data structure. Furthermore, given a vertex v and a version number, the data structure can list the set of vertices that are in the strongly connected component of the vertex v, in the given version of the graph, in time that is proportional to the size of the component.

The rest of this extended abstract is organized as follows. In the next section we describe the new dynamic algorithm for maintaining strongly connected components. In Section 3 we describe a new algorithm for the decremental maintenance of a reachability tree composed of strongly connected components. In Section 4 we combine the results of the preceding sections to obtain the promised dynamic reachability algorithm. We end in Section 5 with some concluding remarks.

2 Fully dynamic strong connectivity with persistency

In this section we describe a fully dynamic strong connectivity algorithm for directed graphs with a certain persistency property. This algorithm plays a pivotal role in the fully dynamic reachability algorithm of Section 4. The strong connectivity algorithm supports the following operations:

$Insert(E')$	–	Create a new version of the graph, initially identical to the latest version of the graph, and add the set of edges E' to it.
$Delete(E')$	–	Delete the set of edges E' from *all* versions of the graph.
$Query(u, v, i)$	–	Check whether u and v are in the same strongly connected component of the i-th version of the graph.

The set E' of edges that are inserted or deleted in each update operation is completely arbitrary. Our algorithm supports each update operation in $O(m\alpha(m, n))$ amortized time, where m is the number of edges in the last version of the graph, plus the number of edges to be inserted or deleted. Note that this is the time needed to update *all* versions of the graph in case of a delete operation. Each query is answered in $O(1)$ worst-case time.

To obtain the almost linear update time we combine a novel dynamic edge partitioning technique with two classical algorithms: the *Union-Find* algorithm (see Tarjan [Tar75]) and an algorithm for answering *LCA* (Least Common Ancestor) queries (see Harel and Tarjan [HT84], Bender and Farach-Colton [BF00], and Alstrup *et al.* [AGKR02]). The space used by the algorithm is *linear*, i.e., $O(m + n)$, no matter how many versions of the graph are now in existence. From now and on we refer to a strongly connected component simply as a *component*.

More formally, the algorithm maintains the (strongly connected) components of a sequence of graphs $G_1, G_2, \ldots, G_t$, where t is the number of insert operations performed so far. Here $G_i = (V, E_i)$ is the graph created by the i-th insert operation. We assume, for simplicity, that $G_0 = (V, E_0)$, the initial graph, is a graph with no edges, i.e., $E_0 = \phi$. See Figure 1(a) for example. The update and query operations are formally defined as follows:

> $\boldsymbol{Insert(E')}$: $t \leftarrow t + 1$, $E_t \leftarrow E_{t-1} \cup E'$
> $\boldsymbol{Delete(E')}$: $E_i \leftarrow E_i - E'$, for $1 \leq i \leq t$
> $\boldsymbol{Query(u, v, i)}$: Are u and v in the same component of G_i?

Note that the edge sets of the graph sequence $G_0, G_1, \ldots, G_t$ satisfy $E_0 \subseteq E_1 \subseteq \cdots \subseteq E_t$. Hence each component of G_i is either a component of G_{i-1} or a union of components of G_{i-1}. The components of all the graphs in the sequence $G_0, G_1, \ldots, G_t$ are arranged, therefore, in an hierarchy. This hierarchy can be naturally represented as a forest. The nodes of the forest are the components of the graphs $G_0, G_1, \ldots, G_t$, with no duplications. The parent of a component w in the forest is the smallest component that strictly contains w. To each component w we assign a *version* number which is the index i of the first graph G_i in the sequence in which w is a component. It is easy to see that the total number of nodes in the component forest is at most $2n - 1$. A simple component forest is depicted in Figure 1.

The algorithm maintains the component forest of the sequence $G_0, G_1, \ldots, G_t$. Strong connectivity queries can then be easily reduced to LCA queries on this forest. To efficiently maintain the component forest, we define the following partition of the edge sets $E_1 \subseteq E_2 \subseteq \cdots \subseteq E_t$:

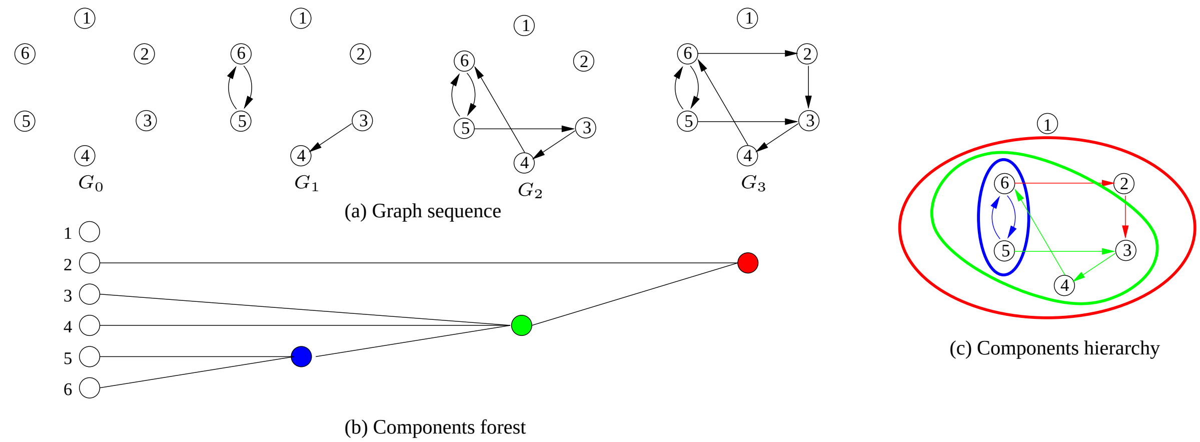

Figure 1: (a) The graph sequence $G_0, G_1, \ldots, G_3$. (b) The components forest for the above sequence. (c) The components hierarchy for the above sequence.

Definition 2.1 (Dynamic edge partitioning) *Let H_i be the dynamic edge set of G_i.*

$$\begin{aligned} H_i &= \{(u,v) \in E_i \mid Query(u,v,i) \wedge (\neg Query(u,v,i-1) \vee (u,v) \notin E_{i-1})\} \\ H_{t+1} &= E_t \setminus \cup_{i=1}^{t} H_i \end{aligned}$$

It follows from this definition that $E_t = \cup_{i=1}^{t+1} H_i$ and $H_i \cap H_j = \phi$ for each $1 \leq i, j \leq t+1$. Note that H_i is composed from all the edges that were inter-component edges in G_{i-1}, or were not present in G_{i-1}, and become intra-component edges in G_i. The set H_{t+1} contains all the edges that are inter-component edges in G_t, the last graph of the sequence. This edge partitioning is dynamic in the sense that any change to the component forest may move edges from H_i to H_j, where $i < j$. We say that H_i is the dynamic edge set of version i.

The new algorithm is given in in Figure 2. The algorithm uses dynamic edge partitioning to create a component forest for a graph sequence. The forest is maintained by two arrays. An array named *parent* saves for each node in the forest a pointer to its parent. An array named *version* saves for each node its creation version (i.e. the creation version of its component). After each update operation the forest is preprocessed in $O(n)$ time by a Least Common Ancestor (LCA) algorithm to answer LCA queries in constant time. The vertices u and v are in the same component in version i if and only if the version of their LCA node is less than or equal to i. Thus, any strong connectivity query on a graph from the sequence is answered in constant time. The initialization routine simply initializes the data structure with an empty graph (i.e. the graph G_0). In this case the components forest is composed from n trees of one node and each such node component is a vertex in G_0.

An insert operation is handled as follows: First, the version counter is incremented and the new edges are added to H_t. (Note that H_t, prior to this operation, contains the inter-component edges of the latest graph). Next, a temporary set of edges H' is created by contracting the endpoints of H_t edges with respect to the components of G_{t-1}. Algorithm *SCC* computes the components of the graph whose edge set is H'. This is easily done in $O(|H'|)$ time (see Tarjan [Tar82], Sharir [Sha81], Gabow [Gab00], or Chapter 22 of Cormen *et al.* [CLRS01]). Finally, using the components of G_t new nodes are added, if necessary, to the component forest and all inter-component edges from H_t are moved to H_{t+1}. A delete operation is handled in a similar manner. The same process in done to each successive pair of graphs in the sequence starting with the pair G_1 and G_2. The component forest is built from scratch and a *Union-Find* algorithm is used to unite the components that formed a new component. Using the *Find* operation when contracting the dynamic edge set H_i of version i to H' the components which contain an edge endpoints are found in $O(\alpha(m,n))$ time. Next, we analyze the running time of the algorithm.

Init:

$t \leftarrow 0$
$H_1 \leftarrow \phi$
for each $v \in V$ do
    parent$[v] \leftarrow$ *null*
    version$[v] \leftarrow 0$

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

$t \leftarrow t + 1$
$H_t \leftarrow H_t \cup E'$
FindScc(H_t, t)
Shift(H_t, H_{t+1})
Pre-LCA(*parent*)

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

for each $v \in V$ do
    parent$[v] \leftarrow$ *null*
for $i \leftarrow 1$ to t
    $H_i \leftarrow H_i \setminus E'$
    FindScc(H_i, i)
    Shift(H_i, H_{i+1})
$H_{t+1} \leftarrow H_{t+1} \setminus E'$
Pre-LCA(*parent*)

Query$(\boldsymbol{u, v, i})$:

return (*version*$[LCA(u, v)] \leq i$)

Figure 2: A fully dynamic strong connectivity algorithm with persistency

FindScc$(\boldsymbol{H, i})$:

$H' \leftarrow \{(Find(u), Find(v)) \mid (u, v) \in H\}$
$\mathcal{C} \leftarrow SCC(H')$
for each $C = \{w_1, w_2, \ldots, w_{|C|}\} \in \mathcal{C}$ do
    if $|C| > 1$ then
        $c \leftarrow NewNode$
        version$[c] \leftarrow i$
        for $j \leftarrow 1$ to $|C|$
            if $j > 1$ then *Union*(w_1, w_j)
            parent$[w_j] \leftarrow c$

Shift$(\boldsymbol{H_1, H_2})$:

for each $(u, v) \in H_1$
    if *Find*$(u) \neq$ *Find*(v) then
        $H_1 \leftarrow H_1 \setminus \{(u, v)\}$
        $H_2 \leftarrow H_2 \cup \{(u, v)\}$

Figure 3: Procedures used by the fully dynamic strong connectivity algorithm

Theorem 2.2 *Each insertion is processed in $O(m\alpha(m,n))$ worst-case time and each deletion is processed in $O(m\alpha(m,n) + t)$ amortized time. The space complexity of the algorithm is $O(m + n)$.*

Proof: After an insertion the algorithm contracts the set of edges H_t to H' and computes the components of the graph $G(V', H')$. Both operations are done in linear time. The insertion is also charged with $O(m\alpha(m,n))$ for future edges that will be shifted to its dynamic edge set. The deletion computes the components for the whole graphs sequence. The dynamic edges partitioning induce a hierarchy of components. If there is not any shift of edges then the components of G_i are computed using the dynamic edge set H_i. The dynamic edge sets are disjoint thus the total cost of this computation is $O(m)$. However, the contraction process that creates H' uses the *Find* operation of the *Union-Find* data structure in order to find for each vertex its components in the forest. Thus, the total deletion time is $O(m\alpha(m,n))$. When an edge is shifted from H_i to H_{i+1} then this edge will be scanned more than once during the deletion but this cost was already charged from the insertion that created version $i + 1$. □

The delete time can be easily reduced from $O(m\alpha(m,n) + t)$ to $O(m\alpha(m,n))$. We omit the details.

The algorithm, as described above, only supports strong connectivity queries. It is not difficult to extend it, however, to identify and report decompositions of strongly connected components. To accomplish that, we consider two consecutive states of the component forest, one just before and one just after a delete

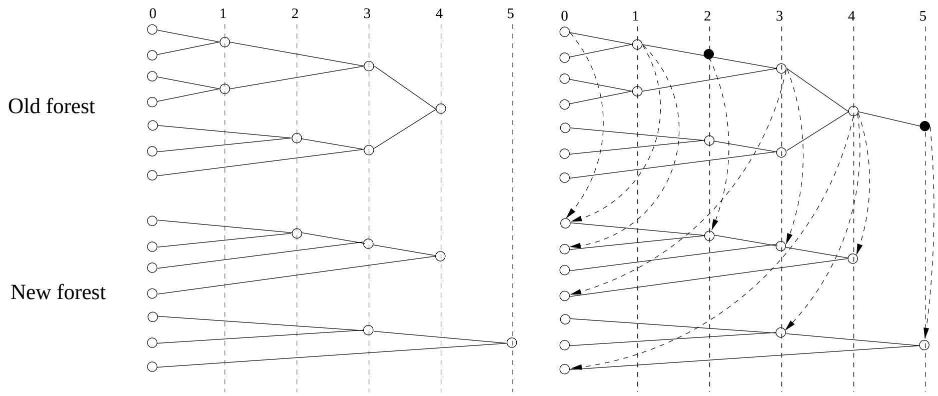

Figure 4: Comparing two consecutive component forests.

operation. By comparing these two forests it is not difficult to determine which components in the old forest decomposed, and into which components of the new forest. We can then add pointers from each component of the old forest into the maximal components of the new forest that are contained in it. This is shown pictorially in Figure 4. We can also add to the component forest, if necessary, intermediate nodes that represent components of the old forest at times in which they did not merge into other components. (See for example the black nodes in Figure 4.) We omit the details from this extended abstract.

3 Decremental maintenance of reachability trees

In this section we describe a new decremental algorithm for maintaining reachability trees. A reachability tree is used to maintain the set of vertices that are reachable from a certain vertex r of a graph $G = (V, E)$ that undergoes a sequence of edge deletions. The nodes of the tree, however, are not individual vertices of G but rather strongly connected components of G. (Clearly, if u and v are in the same component, they are either both reachable, or not reachable, from r.) The root of the tree is the component containing r. Figure 5 shows, for example, a reachability tree that contains five components. Vertices that are contained in other components of G are not reachable from r.

We focus in this section on the decremental maintenance of one reachability tree. The fully dynamic reachability algorithm of the next section will maintain a collection of such trees, each one having a different root and defined with respect to a different version of the graph. More precisely, if $G_0, G_1, \ldots, G_t$ is a sequence of graphs resulting from a sequence of edge insertions and deletions, as defined in Section 2, the fully dynamic algorithm will maintain up to n reachability trees, each one in a different graph G_i. Note that even though insertion operations are performed, each graph G_i, once defined, is only loosing edges.

The new algorithm relies heavily on the ability of the algorithm of the preceding section to efficiently identify decompositions of strongly connected components. It also uses a new way of dealing with such decompositions. These new techniques are used in conjection with ideas borrowed from Italiano [Ita88], Frigioni *et al.* [FMNZ01] and Roditty [Rod03].

3.1 The data structure

The algorithm uses the following simple data structures. For every component w of the graph, the algorithm maintains a doubly linked list *active*$[w]$ that contains all the *active* vertices of the component. A vertex v is active if it has *uninspected inter-component* incoming edges. An edge $(u, v) \in E$ is said to be an inter-component edge if and only if u and v are not in the same strongly connected component of G. Initially, all edges are uninspected. When an inter-component edge is inspected by the algorithm it is either found

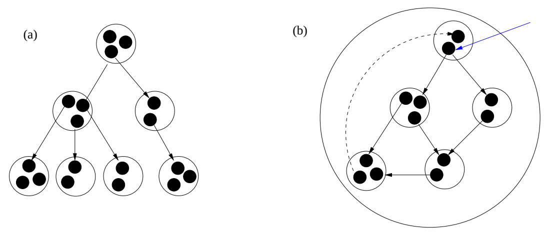

Figure 5: A simple reachability tree

not to be useful, and is permanently removed from the graph, or it becomes a tree edge. Due to a technical reason, tree edges are also considered to be uninspected. (It may seem a bit unnatural to consider tree edges as being uninspected, but this simplifies the description of the algorithm.) For every active vertex v, the algorithm maintains a doubly linked list $in[v]$ containing all the uninspected inter-component edges that enter v. (If v is active, the list $in[v]$ cannot be empty.) For every vertex v, active or inactive, the algorithm maintains a doubly linked list $out[v]$ containing all the edges emanating from v. See Figure 6(a).

The reason for keeping only inter-component edges in the data structure described is fairly obvious. Intra-component edges are certainly important, as they keep components strongly connected, but they can never be serve as tree edges in a reachability tree. (Intra-component edges are self-loops in the component graph.) It should be noted, however, that as edges are deleted from the graph G, components split and edges that were intra-component edges become inter-component edges. Such edges are then added to the data structure, as we describe below.

The algorithm maintains the following invariant:

> **Tree Invariant:** If w is a component of G, v is the first vertex in $active[w]$, and u is the first vertex in $in[v]$, then (u, v) is the tree edge connecting component w to the tree. In particular, if $active[w]$ is empty, then w is not connected to the tree and the vertices of w cannot be reached from r.

The algorithm also maintains an array c indexed by the vertices of the graph. For every vertex $v \in V$, $c[v]$ is the name of the component currently containing v.

3.2 Splitting components and updating the inter-component edges

Suppose that the data structures described above correctly represent a reachability tree from a given vertex r in a graph $G = (V, E)$. A subset $E' \subseteq E$ of edges is now deleted. How do we efficiently update the data structure so that it will represent a reachability tree from r in the new graph $G' = (V, E - E')$?

The required updates are done in two steps. In the first step, described here, we take into account decompositions of components, add new inter-component edges to the data structure and remove inter-component edges that were deleted. These changes may violate the tree invariant and hence destroy the tree. In the next step, to be described in the next subsection, we reconnect the tree and reestablish the tree invariant.

We assume that the data structures of Sections 2 give us a list of the strongly connected components of G that split as the result of the deletion of the edge set E'. Furthermore, for every component w that split, we get the list $w_1, w_2, \ldots, w_k$ of the new components in G' into which w split. For each component w_j we get a pointer to location of w_j in the component forest. (Recall that $G = G_i$, for some i.) We can thus obtain the size $|w_j|$ of a component w_j in constant time, and list the elements of w_j in $O(|w_j|)$ time. We also get the list of edges that become inter-component edges as a result of the deletion of E'. (If $G = G_i$, then these are just the edges that were moved out of H_i as a result of this delete operation.)

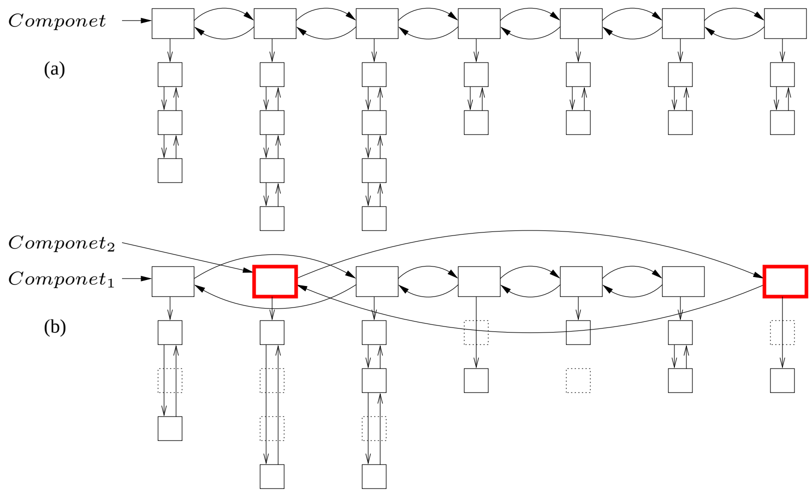

Figure 6: Data structures used by the decremental algorithm for maintaining reachability trees.

Suppose that component w split into $w_1, w_2, \ldots, w_k$. We need to generate new linked lists *active*$[w_j]$, for $j = 1, \ldots, k$, that contain the active vertices of the new components. A naive way of doing it would be to scan the list *active*$[w]$ and move each vertex in it into one of the new lists. This would require, however, $O(|w|) = O(\sum_{j=1}^{k} |w_j|)$ time. Using this approach, we may end up spending a total of $\Omega(n^2)$ time just on constructing the active lists. (This may happen, for example, if ℓ-th delete operation caused a component of size $n - \ell$ to breaks into two components of sizes $n - \ell - 1$ and 1.)

Luckily, there is a slightly less naive way of constructing the lists *active*$[w_j]$, for $j = 1, \ldots, k$. Let us assume that $|w_1| \geq |w_2|, \ldots, |w_k|$. Instead of building the list *active*$[w_1]$ from scratch, we will simply delete from *active*$[w]$ all the vertices that do not belong to w_1. The deleted vertices will be placed in one of the lists *active*$[w_j]$, for $j = 2, \ldots, k$. (As an example, see Figure 6(b).) We can do this by scanning the lists of vertices contained in the components $w_2, \ldots, w_k$. (We assume that for each vertex v we have a pointer to its appearance, if any, in an active list.) The total time required for constructing the lists *active*$[w_j]$, for $j = 1, \ldots, k$, is now only $O(\sum_{j=2}^{k} |w_j|)$. (Note that the sum starts now from 2, and not from 1.) This makes a huge difference! If a vertex is moved from one active list to another, the size of the component containing it must have decreased by a factor of at least 2. Each vertex is then moved at most $\log_2 n$ times and the total amount of time spent on constructing these lists is at most $O(n \log n)$. (Note that the decision as to who will 'inherit' the list *active*$[w]$ is made according to the *size* of the newly formed components, and not according to the number of active vertices they contain. Thus, the fact that active vertices may become inactive and then active again, due to the appearance of new inter-component edges, does not invalidate the argument. The cost of adding a vertex that becomes active to an active list can be charged to an incoming edge that became an inter-component edge.)

New inter-component edges can be added to the data structure at a constant cost per edge. Similarly, edges can be deleted from the data structure at a constant cost per edge. (We assume the existence of the necessary pointers.) As each edge becomes an inter-component edge only once, and each edge is only deleted once, the total cost of adding and removing edges from the data structure, over the whole sequence of edge deletions, is only $O(m)$, where m is the number of edges present in the initial graph G.

3.3 Reconnecting the tree after edge deletions

The operations carried out above may have disconnected the reachability tree. Tree edges may have been deleted, and components may have split. Let W be a set containing all the new components with no incoming tree edge, and all the old components that lost their incoming tree edges. (If the root r

is contained in a new component, this new component is *not* added to W.) The set W can be easily constructed in $O(|W|)$ time. As a result of the deletion of tree edges and the splitting of components, the old reachability tree may become a forest. The root components of this forest are precisely the components in $W \cup \{c[r]\}$. The set W is therefore the set of components that we should try to reconnect to the tree.

The algorithm tries to reconnect each component $w \in W$ to the reachability tree in the following way. It scans the incoming edges of the vertices of $active[w]$ for an edge $(u, v) \in E$ entering the component w (i.e., $v \in active[w]$ and $u \in in[v]$) such that $active[c[u]] \neq \phi$ or $c[u] = c[r]$. During the scan, edges that do not satisfy the required condition become inspected and are therefore removed from the corresponding $in[v]$ list. A vertex v whose list $in[v]$ becomes empty ceases to be active and is removed from $active[w]$.

If an edge $(u, v) \in E$ satisfying the required condition is found, it becomes a tentative tree edge. This edge hangs w on a component $c[u]$ that is either connected to the tree, or is part of a tree whose root is currently in W. We remove w from W.

If no edge satisfying the required condition is found, we scan all inter-component edges emanating from w. (The vertices contained in w are found by accessing the component forest of Section 2. The edges emanating from each vertex v are stored in $out[v]$.) For each such edge (v, u) we check whether it is a (tentative) tree edge. If it is, we add $c[u]$ to W. We again remove w from W. The component w was disconnected from the tree and will never be connected again.

The correctness of the reconnection algorithm follows from arguments similar to the ones used by Roditty [Rod03], which in turn are based on arguments of Italiano [Ita88] and Frigioni *et al.* [FMNZ01]. We omit the formal proof due to lack of space. As the algorithm inspects every edge only once in each direction, and as the total cost of handling the decompositions of components is $O(n \log n)$, the total time needed to maintain a reachability tree is only $O(m + n \log n)$.

4 An almost linear fully dynamic reachability algorithm

In this section we describe our fully dynamic reachability algorithm. The algorithm supports the following operations:

$Insert(E^v)$	– Insert a set of edges E^v, all touching a given vertex v, into the graph.
$Delete(E')$	– Delete all the edges of E' from the graph. The set E' is an arbitrary set of edges.
$Query(u, v)$	– Check whether there is a directed path from u to v in the current version of the graph.

The algorithm uses a framework first suggested by King [Kin99]. The concrete implementation presented here uses the decremental algorithm for maintaining reachability trees of Section 3, which in turn uses the fully dynamic strong connectivity algorithm of Section 2.

An insert operation $Insert(E^v)$, in which each inserted edge either enters or exits v, is said to be *centered* around v. For every vertex v that served as a center of an insert operation the algorithm maintains, decrementally, a pair $T_{in}[v]$ and $T_{out}[v]$ of reachability trees. The tree $T_{out}[v]$ contains all the (strongly connected) components that can be reached from v. The tree $T_{in}[v]$ contains all the components from which v can be reached. These trees are maintained with respect to a dynamic graph G^v that is defined as follows. The graph G^v is initialized by taking a snapshot of the graph G just after the insert operation that triggered the creation of these trees. Subsequent insert operations do not alter the graph G^v. Edges that are subsequently deleted from the graph *are* deleted, however, from G^v. As can be seen, the graph G^v is only subjected to delete operations, and hence a decremental maintenance of $T_{in}[v]$ and $T_{out}[v]$ is sufficient. When the vertex v serves again as an insertion center, the reachability trees $T_{in}[v]$ and $T_{out}[v]$ are rebuilt

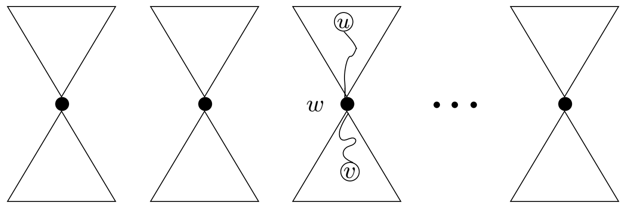

Figure 7: The in and out trees used by the fully dynamic reachability algorithm.

from scratch, and the graph G^v is reinitialized. In general, the algorithm maintains a collection of up to n in and out trees, as can be seen in Figure 7.

To be more precise, we can define a dynamic sequence of graphs, $G_0, G_1, \ldots, G_t$, as in Section 2. Here t is the number of insert operations issued so far. Let $G_i = (V, E_i)$, for $i = 1, \ldots, t$. The i-th insert operation, operation $Insert(E^{v_i})$ creates a new graph $G_i = (V, E_i)$, where $E_i = E_{i-1} \cup E^{v_i}$. A delete operation $Delete(E')$, occurring after the t-th insert operation, deletes the edges of E' from all versions of the graph, i.e., $E_i \leftarrow E_i - E'$, for $i = 1, \ldots, t$. Note that each graph G_i in the sequence is a dynamic graph. It undergoes, however, only edge deletions. The reachability trees $T_{in}[v_i]$ and $T_{out}[v_i]$, created as the result of the i-th insert operation, are maintained with respect to G_i. (In other words, $G^{v_i} = G_i$.)

To check whether there is a directed path from u to v in the current version of the graph, i.e., in $G = G_t$, we check whether there is a vertex w such that $u \in T_{in}[w]$ and $v \in T_{out}[w]$. This can easily be done in $O(n)$ time. If there is such a vertex w, then there is clearly a path from u to v in G_t, as there is already such a path in G_i, where i is the index of the last insert operation centered on w. Conversely, if there is a path from u to v in $G = G_t$, let w be the last vertex on the path to serve as an insertion center. All the edges of the past were present, therefore, when the trees $T_{in}[w]$ and $T_{out}[w]$ were built, and therefore $u \in T_{in}[w]$ and $v \in T_{out}[w]$, as required.

All that remains, therefore, is to analyze the complexity of the new fully dynamic algorithm. The amortized cost of updating the data structures that maintain the strongly connected components of *all* the graphs in the sequence $G_0, G_1, \ldots, G_t$ is only $O(m\alpha(m, n))$. After an insert operation $Insert(E^v)$ we need to initialize data structures for the decremental maintenance of the two reachability trees $T_{in}[v]$ and $T_{out}[v]$. Using the the data structures of Section 3, relying on the data structures of Section 2 to detect and report decompositions of strongly connected components, the *total* cost of building and maintaining these trees is $O(m + n \log n)$. Note that this covers all future maintenance operations associated with these trees. The total amortized cost of an insert operation is, therefore, $O(m\alpha(m, n) + n \log n) = O(m + n \log n)$. (If $m \geq n \log n$, then $\alpha(m, n) = O(1)$.) After a delete operations we need to update all reachability trees. But, the cost of all these updates is already paid for! Thus, the amortized cost of a delete operation is only $O(m\alpha(m, n))$. We thus obtain:

Theorem 4.1 *The new fully dynamic reachability algorithm handles each update operation in $O(m + n \log n)$ amortized time, and answers each reachability query in $O(n)$ worst-case time.*

5 Concluding remarks and open problems

We obtained a new fully dynamic reachability algorithm with an amortized update time of $O(m + n \log n)$ and a query time of $O(n)$. Reducing the update time to $O(m)$ is an interesting open problem. The major open problem, however, is to reduce the query time without significantly increasing the update time. In particular, is it possible to reduce the query time of the algorithm to $O(m/n)$?

References

[AGKR02] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proceedings of SPAA'02*, pages 258–264, 2002.

[BF00] M.A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of LATIN'00*, pages 88–94, 2000.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, second edition, 2001.

[DI00] C. Demetrescu and G.F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proceedings of FOCS'00*, pages 381–389, 2000.

[FMNZ01] 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.

[Gab00] H.N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, 2000.

[HK95] M. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the FOCS'95*, pages 664–672, 1995.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13, 1984.

[Ita88] G.F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988.

[Kin99] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of FOCS'99*, pages 81–91, 1999.

[Rod03] L. Roditty. A faster and simpler fully dynamic transitive closure. In *Proceedings of SODA'03*, 2003.

[RZ02] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proceedings of FOCS'02*, pages 679–689, 2002.

[Sha81] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Tar82] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 11:146–159, 1982.