Improved Deterministic Algorithms for Decremental Reachability and Strongly Connected Components

JAKUB ŁĄCKI, University of Warsaw

This article presents a new deterministic algorithm for decremental maintenance of the transitive closure in a directed graph. The algorithm processes any sequence of edge deletions in $O(mn)$ time and answers queries in constant time. Previously, such time bound has only been achieved by a randomized Las Vegas algorithm. In addition to that, a few decremental algorithms for maintaining strongly connected components are shown, whose time complexity is $O(n^{1.5})$ for planar graphs, $O(n \log n)$ for graphs with bounded treewidth and $O(mn)$ for general digraphs.

Categories and Subject Descriptors: E.1 [**Data Structures**]: *Graphs and networks*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Algorithms, dynamic algorithms, decremental algorithms, graph algorithms, graphs, strongly connected components, transitive closure

ACM Reference Format:
Łącki, J. 2013. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms* 9, 3, Article 27 (June 2013), 15 pages.
DOI:http://dx.doi.org/10.1145/2483699.2483707

1. INTRODUCTION

In this article, we consider decremental maintenance of the transitive closure in a directed graph. We present a data structure, which supports two operations:

— *query*(u, v) — check if there is a directed path from u to v,
— *delete*(u, v) — delete an edge from u to v.

The data structure is based on a novel representation of a strongly connected graph, which might be of independent interest.

The problem of dynamic maintenance of the transitive closure has been given considerable attention in recent years (e.g., Baswana et al. [2007], Demetrescu and Italiano [2005], Frigioni et al. [2001], Henzinger and King [1995], Italiano [1988], La Poutré and van Leeuwen [1987], and Roditty and Zwick [2008]). In this article, we focus on algorithms with constant query time. One of the most natural measures of efficiency of such algorithms is the total time needed to process an arbitrary sequence of updates. In particular, for decremental graph problems one usually bounds the total time needed to update the data structure over deletions of all edges (in arbitrary order). This is caused by the fact that a single edge deletion can cause the maintained

A preliminary version of this article appears in the *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2011)*, 1438–1445.
Author's address: J. Łącki, Institute of Informatics, University of Warsaw, ul.Banacha 2, 02-097 Warsaw, Poland; email: j.lacki@mimuw.edu.pl.

property to change significantly, for example, it might affect as much as $\Theta(n^2)$ cells in the transitive closure matrix.

1.1. Previous Results

Two $O(m^2)$ deterministic decremental algorithms for transitive closure were obtained independently, one by La Poutré and van Leeuwen [1987] and the other by Frigioni et al. [2001]. Demetrescu and Italiano [2005] gave an $O(n^3)$ deterministic algorithm, which is faster for dense graphs. Since all those three algorithms maintain the transitive closure explicitly, each query can be answered in constant time.

These results have been improved with the use of randomized algorithms. Henzinger and King [1995] gave a Monte Carlo algorithm, which is capable of handling arbitrary digraphs with total update time of $O(mn \log^2 n)$. However, it does not maintain the transitive closure explicitly and needs as much as $O(\frac{n}{\log n})$ time for each query.

Baswana et al. [2007] showed an $O(mn^{4/3} \sqrt[3]{\log n})$ Monte Carlo algorithm, whereas Roditty and Zwick [2008] obtained an $O(mn)$ Las Vegas algorithm. Both these algorithms handle queries in constant time.

The problem of decremental maintenance of strongly connected components has been addressed less often. An algorithm suggested by Frigioni et al. [2001] requires $O(m^2)$ worst-case time, which is as slow as recomputing the strongly connected components after each update from the beginning. However, if all deleted edges are selected at random, the expected time is $O(mn)$. Roditty and Zwick [2008] showed a Las Vegas algorithm that maintains strongly connected components in $O(mn)$ total expected time for any sequence of edge removals.

1.2. Our Results

We present an algorithm that maintains the structure of strongly connected components over a sequence of edge deletions in $O(mn)$ total time. It is based on a simple representation of a strongly connected component, which reduces the problem of decremental strong connectivity to maintaining connectivity in a set of directed acyclic graphs. Using our algorithm, we obtain a deterministic decremental $O(mn)$ algorithm for transitive closure with constant query time, thus solving an open problem posed in Roditty and Zwick [2008]. This is a significant advancement over the best currently known deterministic algorithms, which run in $O(m^2)$ or $O(n^3)$ total time.

Our decremental algorithm for maintaining strongly connected components has been recently improved by Roditty [2013]. He obtains a $O(m \log n)$ worst-case bound on both the initialization time and the update time. The total time of executing any sequence of operations remains unchanged and amounts to $O(mn)$.

Moreover, we give two algorithms for decremental maintenance of strongly connected components in special classes of graphs. First, we obtain $O(n^{1.5})$ time complexity for planar graphs or $O(\sqrt{n})$ amortized time per one deletion, if all edges are eventually deleted. To the best of our knowledge, the only comparable results are fully dynamic algorithms for the transitive closure in planar graphs. Although they are more powerful, they do not process queries in constant time.

The oldest one was shown by Subramanian [1993]. It handles updates and queries in $O(n^{2/3} \log n)$ time. Diks and Sankowski [2007] presented an algorithm that can process updates and queries in $O(\sqrt{n} \log^3 n)$ and $O(\sqrt{n} \log^2 n)$ time respectively. Moreover, Sankowski and Mucha [2010] presented an algorithm processing operations in $O(n^{(\omega-1)/2})$ amortized time, where ω is the exponent of $n \times n$ matrix multiplication. However, it is an algorithm with lookahead and it needs to know $O(\sqrt{n})$ operations in advance.

We also show an algorithm for decremental maintenance of strongly connected components in graphs with treewidth bounded by a constant or, more generally, graphs with separators of constant size. It runs in $O(n \log n)$ time. We have not found any comparable algorithms solving this problem.

The rest of this article is organized as follows. In the next section, we describe a simple decremental reachability algorithm for acyclic graphs, which we later use. Section 3 introduces an *SCC-tree*, which is a tree-like data structure for representing strongly connected graphs, capable of efficient updates after edge deletions. Using this structure, we show how to decrementally maintain the transitive closure. The last part of the article presents how to build SCC-trees of smaller size for planar graphs and graphs with bounded treewidth. This leads to more efficient algorithms for these classes of graphs.

1.3. Preliminaries

Let $G = (V, E)$ be a directed graph. By $V(G)$ and $E(G)$, we denote its vertex and edge set, respectively. For vertices $u, v \in V$, $u \xrightarrow{G} v$ is used to denote that there is a directed path in G from u to v. In particular $v \xrightarrow{G} v$, as there is an empty path connecting v with v.

A nonempty set $S \subseteq V$ is *strongly connected* if for any two vertices $s_1, s_2 \in S$, $s_1 \xrightarrow{G} s_2$. This is equivalent to stating that there exists a vertex $s_1 \in S$, such that for every $s_2 \in S$ both $s_1 \xrightarrow{G} s_2$ and $s_2 \xrightarrow{G} s_1$. An inclusion-maximal strongly connected set is a *strongly connected component* or, to shorten notation, an SCC. Every vertex belongs to exactly one strongly connected component.

Let $v \in V$. An *in-edge* of v is an arbitrary element of $\{xv | xv \in E\}$, whereas an *out-edge* of v is an element of $\{vx | vx \in E\}$. If G has no directed cycles, we call it a DAG (directed acyclic graph). A vertex v in a DAG is called a *source* if it has no in-edges, whereas a *sink* is a vertex with no out-edges. Lastly, we assume that G has no isolated vertices, which assures that the number of vertices is asymptotically not greater than the number of edges.

2. DECREMENTAL SINGLE-SOURCE REACHABILITY IN DAGS

In this section, we describe a simple algorithm for single source decremental reachability in acyclic graphs. The algorithm is similar to the one presented by Italiano [1988]. We are given a DAG and our aim is to maintain the set of vertices reachable from the source under a sequence of edge deletions.

Observe that in a single-source DAG every vertex is reachable from the source. This means that as we delete edges, a vertex v might become disconnected from the source only if v or some predecessor of v loses its last in-edge.

We show an auxiliary function FINDUNREACHABLEDOWN(D, S, w), which is based on this observation. Given a DAG $D = (V, E)$ with a distinguished source w and a set S that contains all its other sources (and possibly also some other vertices), FINDUNREACHABLEDOWN(D, S, w) returns a pair (U, I), where U is the set of vertices that are not reachable from w and I is the set of all edges incident to U. Note that this is more than what is necessary for single-source reachability in DAGs, but we use the additional information later on. The pseudocode of FINDUNREACHABLEDOWN is given as Algorithm 1.

LEMMA 2.1. *Algorithm 1 is correct. It runs in $O(|S| + |U| + |I|)$ time.*

PROOF. Fix a topological ordering $v_1, v_2, v_3, \ldots$ of the DAG. Without loss of generality, we can assume $v_1 = w$. The proof proceeds by induction, we will show that for each

Algorithm 1 FindUnreachableDown($(V, E), S, w$)

```
Require: $(V, E)$ is a DAG, $S \subseteq V$, $w$ has no in-edges
 1: $U := \emptyset$ {unreachable vertices}
 2: $I := \emptyset$ {their incident edges}
 3: $Q :=$ EmptyQueue
 4: for all $x \in S \setminus \{w\}$ do
 5:     if $x$ has no in-edges then
 6:         Enqueue($Q$, $x$)
 7: while $Q \neq \emptyset$ do
 8:     $v :=$ Dequeue($Q$)
 9:     $U := U \cup \{v\}$
10:     $I := I \cup$ IncidentEdges($v$)
11:     for all $vx \in E$ do
12:         $E := E \setminus \{vx\}$
13:         if $x$ has no in-edges then
14:             Enqueue($Q$, $x$)
    return $(U, I)$
```

positive integer k, the procedure correctly identifies the unreachable vertices among $v_1, \ldots, v_k$.

Observe that U consists of the vertices that have been inserted to Q at some point. Hence, the basis of the induction ($k = 1$) is trivial – w is inserted to Q neither in 6th line, nor in 14th (for this to happen, w would have to have an in-edge).

Now, assume $k > 1$. There are two ways for a vertex v_k to become unreachable from w. Either it has no in-edges or all its direct predecessors are not reachable from w. In the first case, v_k has to belong to S, so it is added to queue Q and consequently to U. If all direct predecessors of v_k are not reachable, then, by the induction hypothesis, they have been correctly identified by the algorithm and all their outgoing edges have been deleted in the 12th line of the pseudocode. Consequently, v_k is also inserted to Q.

To show that I is computed correctly, it suffices to note that for every vertex added to U, we add all its incident edges to I. Observe that some edges are deleted in the 12th line, but it is easy to see that all these edges already belong to I. This proves the correctness of the function.

To bound the running time, we first observe the following. The fifth line requires $O(|S|)$ time. The number of iterations of the while loop can easily be bounded by $O(|U|)$. Finally, each edge that we examine in the for loop in the 11th line is added to I. Since a vertex can be added to a queue only at the moment its last in-edge is erased, we do not process any edge from I more than once. □

We also define an analogous function FindUnreachableUp(G, S, w), where w is a distinguished sink and S is a set containing all other sinks. It returns a pair (U, I), where U is the set of vertices from which there is no path to w and I is the set of edges incident to U.

In the following, these two functions are called together. Let us define FindUnreachable(G, S) as the function computing the union of their results. Note that we do not specify the source and sink, as later on we apply the function only to DAGs with a clearly distinguished source and sink.

Using FindUnreachableDown, we obtain a single-source decremental reachability algorithm for DAGs.

Lemma 2.2. *Let $G = (V, E)$ be a directed acyclic graph, with $|E| = m$ and $|V| = n$. There exists a $O(m)$ total time deterministic algorithm for decremental single-source reachability in G.*

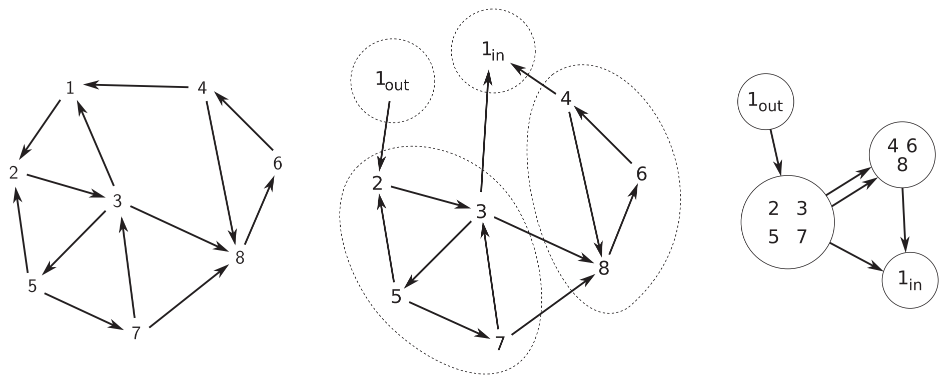

Fig. 1. Graph G, SPLIT$(G,1)$ and CONDENSE(SPLIT$(G,1)$).

PROOF. We maintain the set of vertices that are reachable from a fixed vertex s. First, we delete all vertices that are not reachable from s in $O(m)$ time, thus obtaining a DAG with a single source. When an edge $uv \in E$ is deleted, we call FINDUNREACHABLEDOWN$(G,\{v\},s)$. The vertices that become unreachable are found and deleted from the graph, together with their incident edges. Thus, the running time of FINDUNREACHABLEDOWN$(G,\{v\},s)$ is linear in the number of deleted edges. As there are initially m edges in the graph, it follows that the total time of all *delete* operations is $O(m)$. The set of reachable vertices is maintained explicitly, so we can answer each query in constant time. □

By running one copy of the above algorithm for every vertex of the graph, we obtain an algorithm for the transitive closure.

COROLLARY 2.3. *There exists a deterministic algorithm for decremental maintenance of the transitive closure of a DAG, which runs in $O(mn)$ total time.*

3. DECREMENTAL MAINTENANCE OF STRONGLY CONNECTED COMPONENTS

In this section, we present an algorithm for maintaining strongly connected components under a sequence of edge deletions. It takes a directed graph with n vertices and m edges as input.

The algorithm handles the following operations:

— *query*(u,v) — check if u and v are in the same SCC,
— *delete*(u,v) — delete an edge from u to v.

After an initialization in $O(mn)$ time, it takes $O(1)$ time to answer each query and the total running time of all *delete* operations amounts to $O(mn)$. Hence, if all edges are eventually removed, the amortized running time of a single delete operation is $O(n)$.

Since the edges can only be deleted, the SCCs only decompose. In the beginning, we partition the graph into SCCs, using a standard linear-time algorithm (see, e.g., Cormen et al. [2001]) and after that we work with each SCC separately. Hence, from now on, we assume that the input graph is initially strongly connected.

We start by introducing some definitions illustrated in Figure 1. Let $G = (V,E)$ be a directed graph.

Definition 3.1. A *condensation* of G is a directed graph G' which is obtained from G by contracting all its strongly connected components. Each vertex of G' is a *set of vertices* of the corresponding SCC. Possible multiple edges in G' are preserved, but self loops are removed.

We denote the condensation of G by CONDENSE(G).

It is well known that a condensation of any graph contains no cycles. Every edge belonging to CONDENSE(G) corresponds to some edge from G in a natural way. In the following, we sometimes identify such pairs of edges.

Definition 3.2. Let $d \in V(G)$. By SPLIT(G, d) we denote a graph obtained from G by splitting d into two vertices: d_{in} and d_{out}. The edges incident to d are distributed between those new vertices. The first is given all in-edges and the second all out-edges. Moreover, we define SPLITANDCONDENSE(G, d) := CONDENSE(SPLIT(G, d)).

Observe that for any digraph G SPLIT(G, d) is not strongly connected and consequently SPLITANDCONDENSE(G, d) contains multiple vertices. In particular, it has a single source $\{d_{out}\}$ and a single sink $\{d_{in}\}$. We treat those two vertices as the *distinguished* source and sink in the graph, that is, they are the source and the sink used by FINDUNREACHABLE. We also define a reverse operation.

Definition 3.3. Let H be a directed graph or a set of vertices. A *merge* operation, denoted by MERGE(H, d_1, d_2, d), replaces all occurrences of d_1 and d_2 within H with d.

We remark that d_1 and d_2 do not necessarily need to be vertices or elements of H. For example SPLITANDCONDENSE(G, d) contains vertices $\{d_{out}\}$ and $\{d_{in}\}$ and MERGE(SPLITANDCONDENSE(G, d), d_{out}, d_{in}, d) merges them into a vertex $\{d\}$.[1] In the following we usually give only the first argument, as by default we merge the vertices that are created by SPLIT.

The basis for our algorithm is the observation that by tracing connectivity of $G_d :=$ SPLITANDCONDENSE(G, d), we can retrieve information about *strong* connectivity of G. More precisely, let G be a strongly connected graph, and assume that some edges are deleted from G one by one. Our goal is to detect the moment when G ceases to be strongly connected. For the time being, assume that each delete operation removes an edge that belongs to G_d.

First, we compute G_d. Since this is an acyclic graph, by Lemma 2.2, we can efficiently maintain the set of vertices reachable from $\{d_{out}\}$ and the set of vertices from which there is a path to $\{d_{in}\}$, under a sequence of edge deletions. We claim that the moment one of this sets becomes different from $V(G_d)$, G stops being strongly connected. In order to handle deletions of edges that are not present in G_d, the same idea is then recursively applied to the SCCs of G_d. We now give a formal description of these properties.

LEMMA 3.4. *Let G be a directed graph and G_d =* SPLITANDCONDENSE(G, d), *where $d \in V(G)$. Moreover, let (U, I) be the result of* FINDUNREACHABLE(G_d, S), *where S contains all nondistinguished (i.e., different from $\{d_{out}\}$ and $\{d_{in}\}$) sources and sinks of G_d. Then:*

(1) *All elements of* MERGE(U) *form separate SCCs of G.*
(2) *If $U \neq V(G_d)$, then the SCC of G that contains d is given by* MERGE($\bigcup(V(G_d) \setminus U)$). *Otherwise, d belongs to a single-vertex SCC.*
(3) *The strongly connected components of G are given by* MERGE($\bigcup(V(G_d) \setminus U) \cup U$),

See Figure 2 for illustration.

PROOF. The proof is divided into two cases.

Case 1. $\{\{d_{in}\}, \{d_{out}\}\} \cap U = \emptyset$.

[1]Note the difference between x and $\{x\}$.

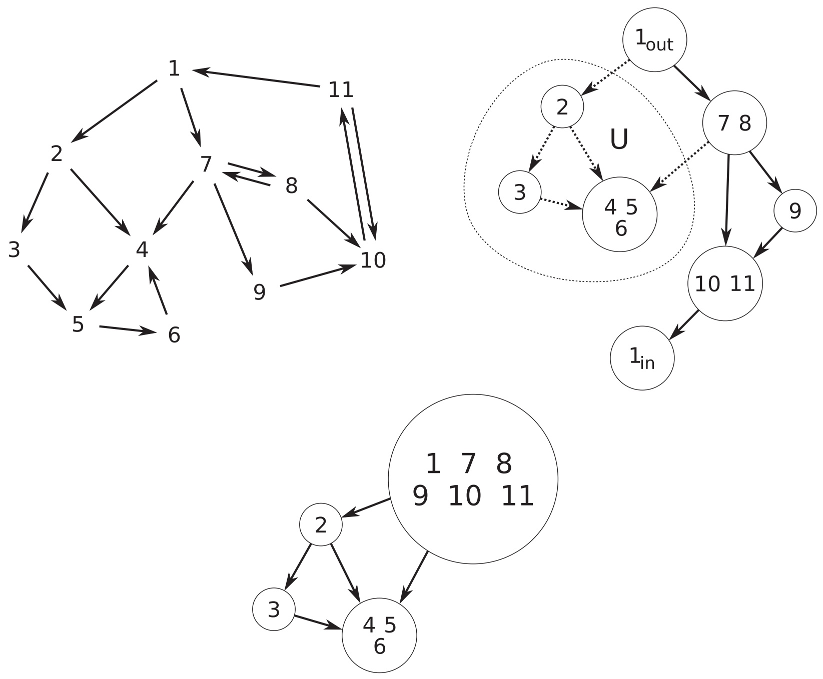

Fig. 2. Graph G, SPLITANDCONDENSE$(G, 1)$ and CONDENSE(G). The dotted arrows in the middle figure represent edges from the set I, where (U, I) is the result computed by FINDUNREACHABLE(SPLITANDCONDENSE$(G, 1), \{\{4, 5, 6\}\}$).

Proof of Claim (1). This assumption implies that MERGE$(U) = U$. Fix $u \in U$. It is a vertex of G_d, so it has to be a strongly connected set in G. We now show that it is also a maximal strongly connected set. Assume that there is a vertex $v_1 \in u$ and $v_2 \notin u$, such that $v_1 \xrightarrow{G} v_2$ and $v_2 \xrightarrow{G} v_1$. At least one of these two paths cannot be mapped to a path between different vertices in G_d, as G_d is acyclic. This means that either v_1-to-v_2 or v_2-to-v_1 path in G has to pass through d. Consequently, $v_1 \xrightarrow{G} d \xrightarrow{G} v_1$, so there is a path from d_{out} to u and from u to d_{in}, a contradiction.

Proof of Claim (2). The assumption that $\{\{d_{in}\}, \{d_{out}\}\} \cap U = \emptyset$ implies that $U \neq V(G_d)$. From Claim (1), we know that the SCC containing d has to be a subset of $V(G) \setminus \bigcup U = \text{MERGE}(\bigcup(V(G_d) \setminus U))$. It turns out that MERGE$(\bigcup(V(G_d) \setminus U))$ is strongly connected. Indeed, by the definition of FINDUNREACHABLE, for every $v \in V(G_d) \setminus U$, both $\{d_{out}\} \xrightarrow{G_d} v$ and $v \xrightarrow{G_d} \{d_{in}\}$, so we easily infer that MERGE$(\bigcup(V(G_d) \setminus U))$ is an SCC.

Proof of Claim (3). From Claims (1) and (2), we have that the SCCs are given by MERGE$(U) \cup$ MERGE$(\bigcup(V(G) \setminus U)) =$ MERGE$(\bigcup(V(G_d) \setminus U) \cup U)$.

Case 2. $\{\{d_{in}\}, \{d_{out}\}\} \cap U \neq \emptyset$.

Proof of Claims (1) and (2). If either $\{d_{out}\}$ or $\{d_{in}\}$ belongs to U then, obviously, they both do and there is no $\{d_{out}\}$-to-$\{d_{in}\}$ path in G_d. This means that $U = V(G_d)$, since if there was a vertex $v \in V(G_d)$ such that $v \notin U$, we would have $\{d_{out}\} \xrightarrow{G_d} v \xrightarrow{G_d} \{d_{in}\}$.

Since there is no d_{out}-to-d_{in} path in G_d, there is also no simple cycle containing d in G. This implies that d forms a single-vertex SCC. Splitting d into two vertices does not break strong connectivity of any set in G, so SCCs in G_d are the same as SCCs in G, with the exception that we have to replace d_{out} and d_{in} with d. Hence, the SCCs are given by $\textsc{Merge}(V(G_d)) = \textsc{Merge}(U)$.

Proof of Claim (3). From the proof of Claims (1) and (2), we know that $U = V(G_d)$, so $\bigcup(V(G_d) \setminus U)$ is an empty set. We have already shown that the set of SCCs is equal to $\textsc{Merge}(U)$, which is in turn equal to $\textsc{Merge}(\bigcup(V(G_d) \setminus U) \cup U)$, as claimed.

This completes the proof. □

Lemma 3.4 is the basis for handling edge deletions in our algorithm. Let G be a strongly connected graph and $G_d = \textsc{SplitAndCondense}(G, d)$. After an edge $uv \in E(G_d)$ is deleted, we can delete it from G_d and then use the lemma to check how the SCCS of G change. We call $\textsc{FindUnreachable}(G_d, \{u, v\})$. Denote the computed result by (U, I). By the lemma, the SCC containing d shrinks to $\textsc{Merge}(\bigcup(V(G_d) \setminus U)) \cup \{d\}$. Moreover, new SCCs, given by $\textsc{Merge}(U) \setminus \{\{d\}\}$, emerge in G.

In addition to that, if we maintain $\textsc{SplitAndCondense}(G, d)$, we can easily compute $\textsc{Condense}(G)$ after each edge deletion.

LEMMA 3.5. *Let G be a directed graph and $G_d = \textsc{SplitAndCondense}(G, d)$. Denote by (U, I) the result computed by $\textsc{FindUnreachable}(G_d, S)$, where S contains all nondistinguished sources and sinks of G_d. Then, $\textsc{Condense}(G) = (V_C, E_C)$ can be computed in $O(|S| + |V_C| + |E_C|)$ time.*

PROOF. From Lemma 3.4, it follows that the vertex set of $\textsc{Condense}(G)$ can be computed with a single call to $\textsc{FindUnreachable}$. Moreover, we can easily obtain E_C from I. Let v_d be the vertex of $\textsc{Condense}(G)$ representing the SCC containing d. Fix an edge $uv \in I$. If one of its endpoints does not belong to U, this endpoint has to be set to v_d in $\textsc{Condense}(G)$. For illustration see the dotted edges entering set U in Figure 2. By Lemma 3.4, $\textsc{FindUnreachable}$ runs in $O(|S| + |V_C| + |E_C|)$ time, which dominates the time needed to compute $\textsc{Condense}(G)$. □

3.1. Graph Representation

Lemma 3.4 gives us a way of tracing strong connectivity of G as long as we only delete edges belonging to $\textsc{SplitAndCondense}(G, d)$. We also have to maintain SCCs of $\textsc{Split}(G, d)$ and to do this, we build a graph representation which recursively uses the same idea.

Definition 3.6. Let G be a strongly connected graph. An *SCC-tree* is constructed as follows:

— An SCC-tree for a single-vertex graph $G = (\{v\}, \emptyset)$ is a node containing G.
— If G has more than one vertex, the root R of the tree contains $\textsc{SplitAndCondense}(G, d)$ (d is chosen arbitrarily) and for each SCC induced by $v \in \textsc{SplitAndCondense}(G, d)$, we add its SCC-tree as a subtree of R, with the exception that we only add one child for d instead of two separate children for d_{in} and d_{out}.

An example SCC-tree is depicted in Figure 3. To prevent ambiguity, in the following, *node* refers to a vertex of an SCC-tree, so that it would not be confused with vertices of the original graph. We use capital letters to denote nodes and lowercase letters to denote vertices. An *inner node* is any nonleaf node. We denote the parent node of N by $p(N)$, the graph (which is always a DAG) in node N by $D(N)$ and the induced subgraph of G represented by the subtree rooted at N by $G(N)$.

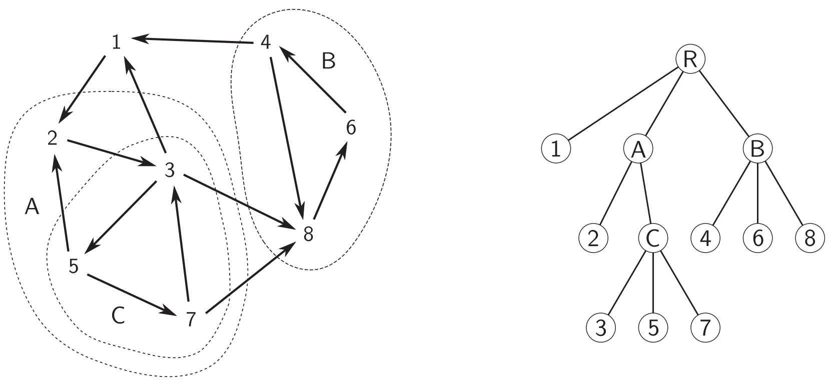

Fig. 3. An example SCC tree of the graph from Figure 1. The root node R corresponds to the entire graph and for every other inner node the subgraph represented by it is marked in the left figure.

The tree has exactly n leaves, since there is exactly one leaf for every vertex. This implies that its height δ and the total number of nodes is linear in n, as each inner node of the tree has at least two children.

Let N be an arbitrary inner node. There exists a natural bijection between vertices of MERGE($D(N)$) and child nodes of N. Hence, there are $O(n)$ vertices in all those DAGs in total. Recall that each vertex $v \in V(D(N))$ is a subset of vertices of the original graph. Although these sets can be big, in the implementation we assume that each vertex can be represented in constant space. The underlying set can be retrieved in linear time by iterating through the whole subtree corresponding to v.

Each edge of G is stored in only one node, namely, an edge uv is kept in the lowest common ancestor of the leaves corresponding to u and v. Hence, the resulting structure uses $O(n + m)$ space.

The tree can be built in $O(m\delta)$ time, using a linear-time algorithm for finding SCCs, because it takes $O(m)$ total time to build its every level. This leads to the following lemma.

LEMMA 3.7. *An SCC-tree can be constructed in $O(m\delta)$ time, where δ is the height of the resulting tree. It requires $O(n + m)$ space.*

3.2. Updating an SCC-Tree After Edge Deletion

In this section, we describe the process of handling an edge deletion. The algorithm maintains one SCC-tree for each SCC of the graph. When an edge belonging to an SCC is deleted, we update the corresponding SCC-tree, so that it is an SCC-tree for the modified graph. If an SCC decomposes, we compute an SCC-tree for each newly created SCC.

In order to answer queries, we maintain an array $SCC[v]$, such that $SCC[v] = SCC[w]$ iff v and w are in the same strongly connected component, which allows us to answer queries in constant time. Initially, since we have assumed that the initial graph is strongly connected, for each v, $SCC[v]$ points to the root of the only SCC-tree.

3.2.1. Deleting an Edge from an SCC-Tree Root. We first consider the case when the deleted edge uv belongs to the root node of an SCC-tree. Denote the root node by R and assume that the first vertex chosen to be split during the construction of the tree is d, that is, $D(R) = \text{SPLITANDCONDENSE}(G, d)$ and $uv \in E(D(R))$. Moreover, denote by G' the graph obtained from G by deleting uv. We delete uv from $D(R)$. Observe that $D(R)$ *after* the deletion is equal to $\text{SPLITANDCONDENSE}(G', d)$.

Following Lemma 3.4, we invoke $\textsc{FindUnreachable}(D(R), \{u, v\})$. Note that $\{u, v\}$ contains all nondistinguished sources and sinks of $D(R)$. Assume that it returns a pair (U, I).

By Lemma 3.4, all elements of $\textsc{Merge}(U)$ form new SCCs. Observe that for each $v \in \textsc{Merge}(U), u \neq \{d\}$, there exists a child subtree of R which is an SCC-tree representing v. We disconnect those subtrees from R and make separate trees out of them.

Then, we update the representation of the SCC containing d. If $U \neq V(D(R))$, the elements of U are no longer in the SCC containing d, so we remove all vertices of U (and their incident edges) from $D(R)$. Otherwise, from now on, d belongs to a single-vertex SCC, so we create leaf node representing it. As a result, we can handle this case in $O(|U| + |I|)$ time.

It remains to update the *SCC* array. For each newly born SCC, we iterate through all its vertices and update the corresponding entries in the *SCC* array, so that they point to roots of the appropriate SCC-trees.

3.2.2. Deleting an Edge from an Inner Node. The case when we remove an edge uv from a graph $D(N)$ in an inner node N of an SCC-tree is only slightly more complex. We update the SCC-tree starting from N and then going up the tree. Observe that all other nodes of the SCC-tree are not affected by the deletion, as the induced subgraph of G represented by each of these nodes remains strongly connected.

Recall that $G(N)$ denotes the induced subgraph of G represented by N, that is, $\bigcup V(D(N)) = V(G(N))$. As in the previous case, after we remove uv from $D(N)$ we have $D(N) = \textsc{SplitAndCondense}(G, d)$ (here G is the entire graph *after* deleting uv). Hence, we can use Lemma 3.5 to compute $\textsc{Condense}(G(N))$ in linear time. This can achieved by calling $\textsc{FindUnreachable}(D(N), \{u, v\})$. Denote the result of $\textsc{FindUnreachable}$ by (U, I). If $\textsc{Condense}(G(N))$ is a single-vertex graph, no further work needs to be done. Otherwise, we have to update node N and its parent $p(N)$.

Updating N can be done in the same way as in the previous case. Some vertices and edges are removed from $D(N)$, as well as the subtrees corresponding to the removed vertices. In the previous cases, for each element of $\textsc{Merge}(U) \setminus \{\{d\}\}$, we created a new SCC-tree. Now, the root of each such SCC-tree will become a sibling of N. The details follow.

Let v_N be the vertex in $D(p(N))$ that corresponds to N. Before the deletion, it corresponded to a strongly connected induced subgraph of G, but now this subgraph is no longer strongly connected. In order to update $D(p(N))$, we plant $\textsc{Condense}(G(N))$ in place of v_N in $D(p(N))$. However, in the algorithm, we do not simply delete v_N and then insert $\textsc{Condense}(G(N))$ instead of it. We achieve the same effect, but we keep the vertex v_N in $D(p(N))$ (it will still represent the SCC containing d, which has just shrunk) and add all other vertices of $\textsc{Condense}(G(N))$ to $D(p(N))$. This small change is important for the running time analysis.

More precisely, the operation consists of three steps. Let $U' = \textsc{Merge}(U) \setminus \{\{d\}\}$.

(1) Vertices from U' and edges of I are moved from $D(N)$ to $D(p(N))$.
(2) Each child subtree of N that corresponds to elements of U' is moved one level up, that is, it becomes a child of $p(N)$.
(3) Since v_N now represents a smaller SCC, some edges that are incident to v_N have to be corrected, as they should now point to the newly inserted elements of U in $D(p(N))$

We call this operation a *lift up* of a graph.

In order to perform the third step, we have to update the edges that belong to $D(p(N))$, were incident to v_N, but should now be incident to a vertex of U'. We iterate through all vertices of $\bigcup U'$ and all their incident edges in the entire graph. If we find an edge where only one endpoint belongs to $\bigcup U'$, we update the corresponding edge in $D(p(N))$ to point to an appropriate element of U'.

After a graph is lifted up, $D(p(N))$ may become invalid, namely it might no longer be a single-source and single-sink DAG. Observe that replacing a vertex with an acyclic graph cannot result in a cycle emerging in $D(p(N))$.

Hence, the only problem that can arise after the lift up is that some vertices become disconnected from the source or lose connection to the sink. This causes the SCC represented by $p(N)$ to decompose into smaller components. Thus, we use Lemma 3.4 again and call FINDUNREACHABLE$(D(p(N)), U' \cup \{v_N\})$. Note that we use the fact that all nondistinguished sources in $D(p(N))$ are among $U' \cup \{v_N\}$.

If FINDUNREACHABLE$(D(p(N)), U \cup \{v_N\})$ returns a nonempty result, $G(p(N))$ decomposes into smaller SCCs and, by Lemma 3.5, we compute CONDENSE$(G(p(N)))$ in linear time. We lift it up to $D(p(p(N)))$ and continue the process going up the tree as long as necessary. At the end, it might turn out that the whole SCC decomposes into smaller components and the *SCC* array has to be updated.

The pseudocode for updating an SCC-tree is given as Algorithms 2 and 3.

```
Algorithm 2 DELETEEDGE($T, e$)
Input: an SCC-tree $T$ and an edge $e = uv \in E(D(N))$
1: $N :=$ the node of $T$ containing $e$
2: remove $uv$ from $D(N)$
3: UPDATESCC-TREE($N, \{u, v\}$)
```

LEMMA 3.8. *Given an SCC-tree of height δ, the algorithm processes any sequence of edge deletions in $O(m\delta)$ total time and answers each query in $O(1)$ time, using $O(n+m)$ space.*

PROOF. By Lemma 3.7, the initialization takes $O(m\delta)$ time and the tree uses $O(n+m)$ space. Line 1 in Algorithm 2 requires $O(1)$ time, as for each edge we can maintain a pointer to the node containing it. In the remaining part of the proof, we focus on bounding the running time of Algorithm 3.

The key observation is that throughout all *delete* operations every vertex and every edge moves up the SCC-tree it belongs to and each operation we perform can be charged to one of these events.

Formally speaking, consider an edge uv of the graph G. It belongs to some DAG $D(N)$, where N is a node in an SCC-tree T. Define the *level* of uv as the depth of N in T. Similarly, we define the *level* of a vertex v of G as the depth of the leaf containing v in the SCC-tree that v belongs to. During the course of the algorithm, the level of each edge and vertex may only decrease. To prove this, we analyze the parts of the algorithm that affect the levels. Edges and vertices are moved between nodes in lines 5 and 12. In this case, they are moved from N to $p(N)$, so their level decreases. Levels also change in 16th line. The child subtrees of N that belong to S become siblings of N. Again, this can only decrease the levels of edges and vertices. A similar argument can be used for line 31. No other parts of the algorithm modify the levels, so we infer that the levels can only decrease. The remaining part of the proof shows that all the operations performed by the algorithm can be charged to decreases in levels of edges or vertices.

First, we show that all calls to FINDUNREACHABLE take $O(m\delta)$ time. By Lemma 2.1, the running time of a single FINDUNREACHABLE operation is linear with respect to the size of its input and output. Every edge it returns is immediately either lifted up (the level of the edge decreases) or deleted from the SCC-tree and its second parameter

Algorithm 3 UpdateSCC-tree(N, A)

Input: an SCC-tree node N and a set of altered vertices $A \subseteq V(D(N))$

```
1: $(U, I) :=$ FindUnreachable$(D(N), A)$
2: if $U = \emptyset$ then return
3: $d :=$ the split vertex for $N$
4: $U' :=$ Merge$(U) \setminus \{\{d\}\}$
5: remove $(U', I)$ from $D(N)$
6: if $D(N) = (\{\{d_{in}\}, \{d_{out}\}\}, \emptyset)$ then
7:     $D(N) :=$ Merge$(D(N))$
8: $\mathcal{S} :=$ subtrees of $N$ corresponding to elements of $U'$
9: if $N$ is not the root node then
10:
11:     {Step 1}
12:     add $U'$ and $I$ to $D(p(N))$
13:
14:     {Step 2}
15:     for all $S \in \mathcal{S}$ do
16:         set the parent of $S$ to $p(N)$
17:
18:     {Step 3}
19:     for all $u \in U'$ do
20:         for all $x \in u$ do
21:             $cv[x] :=$ vertex in $D(p(N))$ corresponding to $u$
22:     for all $v \in \bigcup U'$ do
23:         for all $uv$ incident to $v$ in $E(G)$ do
24:             if for some endpoint $x$ of $uv$, $x \notin U'$ then
25:                 update edge $uv$ in $D(p(N))$ replacing endpoint $x$ with $cv[x]$
26:
27:     $v_N :=$ the vertex in $D(p(N))$ corresponding to $N$
28:     UpdateSCC-tree$(p(N), U' \cup \{v_N\})$
29: else
30:     for all $S \in \mathcal{S}$ do
31:         disconnect $S$ from $N$ and make a separate tree out of it
32:         $V_S :=$ vertices of the subgraph represented by $S$
33:         for all $v \in V_S$ do
34:             $SCC[v] := S$
```

is either of constant size or is a set of vertices that have just been lifted up. It follows that the total running time of FindUnreachable is linear in the total number of single edge or vertex level decreases, hence it can be bounded by $O(m\delta)$.

Similarly, we bound the running time of the loops from lines 19–25. They run in the time that is proportional to the number of edges incident to vertices from $\bigcup U'$ (in the entire graph). This can, however, be charged to the decrease in the level of elements of $\bigcup U'$.

It remains to show that updating the $SCC[v]$ array is efficient, but again every time we update $SCC[v]$, the level of v decreases (see line 31). The lemma follows. □

From this lemma and the fact that the height of an SCC-tree can be bounded by $O(n)$, we infer the desired result.

Theorem 3.9. *There exists a deterministic algorithm for decremental maintenance of strongly connected components, which runs in $O(mn)$ total time and answers queries in $O(1)$ time, using $O(n + m)$ space.*

This result can be used to give a faster dynamic algorithm for decremental transitive closure. We follow the approach by Roditty and Zwick [2008]. They use an algorithm by Frigioni et al. [2001] and replace the part responsible for detecting SCCs decompositions with their decremental randomized algorithm. Then, they observe that Frigioni et al. [2001] algorithm runs in $O(mn)$ total time, if the time needed to maintain the structure of SCCs is excluded. This way, they obtain an $O(mn)$ total time randomized algorithm for decremental transitive closure. Using our approach for maintaining SCCs, we obtain the following results.

THEOREM 3.10. *There exists a deterministic, decremental algorithm for maintaining the transitive closure of a directed graph that processes any sequence of delete operations in $O(mn)$ time.*

Note that the preliminary version of this article also contained an alternative description of obtaining a decremental transitive closure algorithm, which did not refer to other papers, but it was incorrect, as the promised time bounds did not hold.

4. APPLICATIONS TO SPECIAL CLASSES OF GRAPHS

In this section, we show that the decremental algorithm from Section 3 can be improved for some graphs with additional properties. Until now, while building an SCC-tree for a given graph, in each step we selected an arbitrary vertex to be split. We show that in some cases, we may obtain an SCC-tree of lower height if this vertex is chosen in a more careful way. To do that, we utilize separators.

Definition 4.1. A *vertex separator* of an undirected graph $G = (V, E)$ is a subset of vertices, whose removal decomposes the graph into components of size at most $\alpha|V|$, for some constant $0 < \alpha < 1$. A family of graphs $\mathcal{F}$ is called $f(n)$-separable if

— for every $F \in \mathcal{F}$, and every subgraph $H \subseteq F$, $H \in \mathcal{F}$,
— for every graph $F \in \mathcal{F}$, such that $|V(F)| = n$, F has a vertex separator of size $f(n)$.

The input for an algorithm which maintains SCCs is a directed graph. Throughout this section, the theorems for undirected graphs are often applied to digraphs. In such cases, we treat all edges of the graph as undirected.

In the following, we consider two important separable families of graphs.

THEOREM 4.2. [LIPTON AND TARJAN 1979]. *Planar graphs are $\sqrt{8n}$-separable. The separators can be found in linear time.*

THEOREM 4.3. [BODLAENDER 1996; REED 1992]. *Graphs of treewidth at most k are k-separable. Assuming that k is a constant, the separators can be found in linear time.*

The following lemma shows how to take advantage of small vertex separators.

LEMMA 4.4. *Let $G = (V, E)$ be a directed strongly connected graph, such that $G \in \mathcal{F}$, where $\mathcal{F}$ is Cn^s-separable ($s \geq 0$). Moreover, assume that the separators for every graph in $\mathcal{F}$ can be found in linear time. Then, we can build an SCC-tree for G of height $O(h(n))$ in $O(|E|h(n))$ time, where $h(n) = O(n^s)$ for $s > 0$ and $h(n) = O(\log n)$ for $s = 0$.*

PROOF. First, compute the separator of G in linear time. Then, we build an SCC-tree for G, and the consecutive elements from the separator are selected as the vertices to be split. After using all the vertices from the separator, the graph breaks into components of size at most $\alpha|V|$ $(0 < \alpha < 1)$.[2] Hence, if we denote the height of an SCC-tree for a graph with n vertices by $h(n)$, we get

$$\begin{aligned} h(n) &= O(1) \text{ for } n = O(1) \\ h(n) &\le Cn^s + h(\alpha n) && (1) \\ &\le \textstyle\sum_{i=0}^{O(\log n)} C(\alpha^i n)^s \\ &= n^s \textstyle\sum_{i=0}^{O(\log n)} C\alpha^{is}. && (2) \end{aligned}$$

Now, if $s > 0$, the sum converges to a constant, and we obtain $h(n) = O(n^s)$. Otherwise, if $s = 0$, the sum consists of a logarithmic number of constant terms, and we have $h(n) = O(\log n)$. By Lemma 3.7, the whole tree can be built in $O(|E|h(n))$ time, so the lemma follows. □

THEOREM 4.5. *Let $G = (V, E)$ be a directed planar graph and $|V| = n$. There exists a decremental algorithm for maintaining its strongly connected components, which runs in $O(n^{1.5})$ total time.*

PROOF. First, we find strongly connected components of G. We can work with each SCC separately, so we assume that the graph is strongly connected. By Theorem 4.2 and Lemma 4.4, we can build an SCC-tree of height $O(\sqrt{n})$ in $O(n^{1.5})$ time. Using Lemma 3.8, we obtain the algorithm. □

THEOREM 4.6. *Let $G = (V, E)$ be a directed graph, whose treewidth is bounded by a constant k. There exists a decremental deterministic algorithm that maintains its strongly connected components in $O(n \log n)$ total time.*

PROOF. First, we find strongly connected components of the graph and treat each of them separately. Without loss of generality, we assume that G is strongly connected. Using Theorem 4.3, for every subgraph of G we can find a vertex separator in linear time. The size of this separator is kn^0. By Lemma 4.4, we can build SCC-tree for G of height $O(\log n)$ in $O(n \log n)$ time. Here, we use the well-known fact that in graphs with constant treewidth the number of edges is bounded by $O(n)$ (see, e.g., Reed [1992]). Using Lemma 3.8, we obtain the algorithm. □

5. CONCLUSION

We have presented an $O(mn)$ decremental deterministic algorithm for the transitive closure. It matches the time bound for currently best known decremental deterministic algorithms for all-pairs reachability in DAGs, as well as single-source reachability in general graphs. It remains an open problem, if any of those potentially simpler cases can be solved more efficiently. An interesting question is also whether the structure of SCC-tree introduced in this paper can be used for constructing any other dynamic graph algorithms. The main problem regarding decremental maintenance of strongly connected components, which remains open, is whether SCCs can be maintained in $o(mn)$ time. In this article, we have shown such results for some specific classes of graphs.

[2]Note that it would suffice that the graph breaks into SCCs of size at most $\alpha|V|$, but we do not see how to take advantage of this property.

ACKNOWLEDGMENTS

I would like to thank Łukasz Bieniasz-Krzywiec, Marek Cygan, Krzysztof Diks, Łukasz Kowalik, Dariusz Leniowski and Piotr Sankowski for their help, support and useful remarks.

REFERENCES

Surender Baswana, Ramesh Hariharan, and Sandeep Sen. 2007. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *J. Algor. 62,* 2, 74–92.

Hans L. Bodlaender. 1996. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput. 25,* 6, 1305–1317.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* 2nd Ed. MIT Press and McGraw-Hill Book Company.

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

Krzysztof Diks and Piotr Sankowski. 2007. Dynamic plane transitive closure. In *Proceedings of ESA*. Lars Arge, Michael Hoffmann, and Emo Welzl Eds., Lecture Notes in Computer Science, vol. 4698, Springer, 594–604.

Daniele Frigioni, Tobias Miller, Umberto Nanni, and Christos D. Zaroliagis. 2001. An experimental study of dynamic algorithms for transitive closure. *ACM J. Exper. Algorithmics 6*, 9.

Monika Rauch Henzinger and Valerie King. 1995. Fully dynamic biconnectivity and transitive closure. In *Proceedings of FOCS*. 664–672.

Giuseppe F. Italiano. 1988. Finding paths and deleting edges in directed acyclic graphs. *Inf. Process. Lett. 28,* 1, 5–11.

Richard J. Lipton and Robert E. Tarjan. 1979. A separator theorem for planar graphs. *SIAM J. Appl. Math. 36,* 2, 177–189. http://www.jstor.org/stable/2100927.

Johannes A. La Poutré and Jan van Leeuwen. 1987. Maintenance of transitive closures and transitive reductions of graphs. In *Proceedings of WG*. Herbert Göttler and Hans Jürgen Schneider Eds., Lecture Notes in Computer Science, vol. 314, Springer, 106–120.

Bruce A. Reed. 1992. Finding approximate separators and computing tree width quickly. In *Proceedings of STOC*. ACM, 221–228.

Liam Roditty. 2013. Decremental maintenance of strongly connected components. In *Proceedings of SODA*. 1143–1150.

Liam Roditty and Uri Zwick. 2008. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput. 37,* 5, 1455–1471.

Piotr Sankowski and Marcin Mucha. 2010. Fast dynamic transitive closure with lookahead. *Algorithmica 56,* 2, 180–197.

Sairam Subramanian. 1993. A fully dynamic data structure for reachability in planar digraphs. In *Proceedings of ESA*. Thomas Lengauer Ed., Lecture Notes in Computer Science, vol. 726, Springer, 372–383.

Received June 2011; revised February 2013; accepted February 2013