2016 IEEE 57th Annual Symposium on Foundations of Computer Science

Decremental Single-Source Reachability and Strongly Connected Components in $\widetilde{O}(m\sqrt{n})$ Total Update Time

Shiri Chechik*, Thomas Dueholm Hansen†, Giuseppe F. Italiano‡, Jakub Łącki§, Nikos Parotsidis¶
*Tel Aviv University, Tel Aviv, Israel. Email: schechik@cs.tau.ac.il
†Aarhus University, Aarhus, Denmark. Email: tdh@cs.au.dk
‡University of Rome Tor Vergata, Rome, Italy. Email: giuseppe.italiano@uniroma2.it
§Sapienza University of Rome, Rome, Italy. Email: j.lacki@mimuw.edu.pl
¶University of Rome Tor Vergata, Rome, Italy. Email: nikos.parotsidis@uniroma2.it

***Abstract*—We present randomized algorithms with a total update time of $\tilde{O}(m\sqrt{n})$ for the problems of decremental single-source reachability and decremental strongly connected components on directed graphs. This improves recent breakthrough results of Henzinger, Krinninger and Nanongkai [STOC 14, ICALP 15]. In addition, our algorithms are arguably simpler.**

***Keywords*-dynamic algorithm; single-source reachability; strongly connected components**

I. Introduction

Dynamic graph algorithms are designed to answer queries on graphs subject to updates, such as adding or removing a vertex or an edge. Typically, one is interested in a very small query time (either constant or poly-log), while minimizing the update time as much as possible. A dynamic algorithm is said to be *incremental* if it handles only insertions, *decremental* if it handles only deletions, and *fully dynamic* if it handles both insertions and deletions. Dynamic graph algorithms have been extensively studied in the last four decades, and efficient dynamic algorithms with poly-log update times are known for many basic problems on undirected graphs, including dynamic connectivity, dynamic 2-edge connectivity, dynamic 2-vertex connectivity, and dynamic minimum spanning tree (see, e.g., [1]–[6]).

Dealing with directed graphs seems much more challenging. In fact, up until very recently, even for very basic reachability problems (e.g., decrementally maintaining whether a fixed node s can reach a fixed node t), no sublinear (in the number of vertices) update time algorithms were known for general graphs. Very recently, in an important breakthrough, Henzinger, Krinninger and Nanongkai [7], managed to circumvent this barrier, by presenting a randomized decremental single-source reachability algorithm with total update time $O(mn^{0.984+o(1)})$ (that is with $n^{1-\epsilon}$ amortized update time for some fixed ϵ). They later improved the running time to $O(mn^{0.9+o(1)})$ [8]. Both these algorithms are quite involved and solve the more general problem of single-source decremental reachability. In this problem we are given a directed graph G and a source node s and the goal is to maintain the set of nodes that are reachable from s, subject to edge deletions.

In this paper we present an improved randomized decremental single-source reachability algorithm with an improved running time of $O(m\sqrt{n}\log n)$. In addition, our algorithm is arguably simpler.

Previous Results: A naive approach to decremental single-source reachability (SSR) is to simply recompute from scratch all the vertices that are reachable from the source after every deletion. This can be done in $O(m+n)$ time per update, and gives a total update time of $O(m^2)$ over all deletions. Even and Shiloach [9] were the first to beat this naive approach, with a decremental algorithm that runs in $O(mn)$ total update time. (A similar scheme was independently found by Dinitz [10].) Although the algorithm of Even and Shiloach solves a more general problem than decremental single-source reachability, it was still the best known for the problem and the $O(mn)$ barrier stood for more than three decades. Only in the special case of directed acyclic graphs, decremental SSR could be solved faster, i.e, in $O(m)$ total update time [11].

A closely related problem is the decremental maintenance of strongly connected components (SCC), where we are required answer queries of the form: "Given two vertices u and v, do u and v belong to the same SCC?". This problem is almost equivalent to the decremental SSR problem: A solution for decremental SCC trivially implies a decremental SSR algorithm with the same running time, while a decremental SSR algorithm with running time $O(mn^{\beta})$, with $\beta=\Omega(1)$, implies a decremental SCC algorithm with essentially the same *expected* total update time (see [7], [12]). This justifies why for many years the best known upper bound for a decremental SCC algorithm was also $O(mn)$ [12]–[14]. The lack of improvement made researchers in the field wonder whether $O(mn^{1-\epsilon})$ was possible [12], [13], [15]. In a recent breakthrough, Henzinger, Krinninger and Nanongkai [7] showed that this is indeed possible and presented algorithms for decremental SCC and SSR with $O(mn^{0.984+o(1)})$ total update time. They later [8] improved this to $O(mn^{0.9+o(1)})$. Both these results are Las Vegas randomized.

0272-5428/16 $31.00 © 2016 IEEE
DOI 10.1109/FOCS.2016.42

Our Results: We present improved randomized decremental SSR and SCC algorithms with total update time of $O(m\sqrt{n}\log n)$. This matches the best known result for planar graphs [13], up to logarithmic factors. In addition to the substantially improved bounds, our algorithms are arguably simpler than the algorithms in [7], [8]. Our single-source reachability algorithm can be generalized to an algorithm for maintaining reachability from k fixed vertices, at the cost of increasing the running time by an additive $O(km\log n)$ term. (Due to lack of space, the details are deferred to the full version of the paper.) We focus our attention on giving a faster decremental SCC algorithm, as a decremental SSR algorithm follows from it quite easily. Our decremental SCC algorithm uses two existing decremental SCC algorithms. One is the $O(mn)$ total expected time algorithm by Roditty and Zwick [12] and the other is the $O(mn)$ total worst-case time algorithm by Łącki [13]. What is crucial for our result, is that in some special cases the running time bounds of both these algorithms can be improved. The algorithm by Roditty and Zwick uses Even-Shiloach trees (in short ES-trees) to maintain BFS trees under edge deletions. This data structure requires $O(mn)$ total time in the general case, but only $O(m\delta)$ time, if the diameter of the graph does not exceed δ. On the other hand, the performance of Łącki's algorithm can be improved if the graph contains a small separator, i.e. a small set of vertices, whose removal considerably reduces the sizes of the SCCs.

We show that each graph has one of the two aforementioned properties. Namely, we prove that in a graph of diameter $\Omega(q\log n)$ there exists a set of k vertices (a separator), whose removal results in the largest SCC having size at most $n-kq$. This allows us to combine the algorithm of Roditty and Zwick with the algorithm by Łącki. We use the former as long as the graph has small diameter (as it uses ES-trees internally it can also measure the diameter up to a constant factor), and once the diameter exceeds a predefined threshold $\delta=\Theta(\sqrt{n})$, we switch to the algorithm by Łącki. At this point the graph has a separator of small size, which we can use to make the algorithm by Łącki run efficiently. We believe that our structural result regarding separators may be of independent interest.

II. Preliminaries

Let $G=(V,E)$ be a *directed* graph with vertex set $V=V(G)$ and edge set $E=E(G)$. We denote the number of vertices and edges by n and m, respectively. G is *strongly connected* if every vertex is reachable from every other vertex. The *strongly connected components* (in short SCCs) of G are its maximal strongly connected subgraphs. The SCCs of a graph can be computed in $O(m+n)$ time [16]. We let $\overleftarrow{G}$ be the graph obtained by reversing the direction of all edges in G, and refer to $\overleftarrow{G}$ as the *reversed graph of* G. Note that G and $\overleftarrow{G}$ have the same SCCs. We denote by $G\setminus S$ (resp., $G\setminus uv$) the graph obtained after deleting a set S of vertices (resp., an edge uv) from G. Additionally, we let $G[S]$ be the subgraph of G induced by the set of vertices S. Let H be a strongly connected graph. We say that deleting an edge uv *breaks* H, if $H\setminus uv$ is not strongly connected. Let u, v be two vertices of G. We denote by $dist(u,v)$ the distance from u to v. If there is no path from u to v, $dist(u,v)=\infty$. The *diameter* of G is the largest distance in G. If G is not strongly connected, its diameter is equal to ∞.

For a given initial graph G, the *decremental SCC* problem asks us to maintain a data structure that allows edge deletions and that can answer whether (arbitrary) pairs of vertices are in the same SCC. The goal is to update the data structure as quickly as possible while still answering queries in constant time. The *decremental single-source reachability* (SSR) problem similarly asks us to maintain a data structure that allows edge deletions and that can answer whether (arbitrary) vertices are reachable from a fixed given source $v\in V(G)$. Our algorithm for maintaining SCCs under edge deletions is obtained by combining two previous data structures: Even-Shiloach trees [9] and Łącki's SCC-decomposition [13]. We next briefly describe these two data structures.

A. ES-tree

Even and Shiloach [9] introduced a data structure, commonly referred to as an Even-Shiloach tree (ES-tree), to maintain a breadth-first search (BFS) tree from a given source vertex under edge deletions. Let G be a graph and $r\in V(G)$. Clearly, G is strongly connected iff the BFS trees from r in G and $\overleftarrow{G}$ both contain all vertices of G. Checking whether G remains strongly connected when edges are deleted can therefore be done by maintaining two ES-trees F and $\overleftarrow{F}$ for G and $\overleftarrow{G}$, respectively. Our algorithm uses such a pair of ES-trees from a shared source r to detect when an edge deletion breaks a strongly connected subgraph. An ES-tree can also be modified such that it maintains a set of vertices, whose distance from the root is at most δ, where δ is a parameter specified when the ES-tree is built. The data structure may report all vertices whose distance from the root is more than δ (which includes vertices not reachable from the root). Maintaining such an ES-tree takes $O(m\delta)$ total time. Let G be a graph, $r\in V(G)$, and $\delta>0$. Our algorithm uses a data structure that maintains two independent δ-depth-bounded ES-trees. Both trees are rooted in r and have depth bound δ. One of them is built in G and the other one in $\overleftarrow{G}$. We call such a data structure an *in-out ES-tree*.

Definition 1 (In-out ES-tree)**.** *Let G be a graph, $r\in V(G)$, and $\delta>0$. A δ-depth-bounded* in-out ES-tree *of G is a data structure $\mathcal{E}=(G,r,\delta,D)$ that maintains the set D of all vertices $v\in V(G)$ with either $\mathrm{dist}(r,v)>\delta$ or $\mathrm{dist}(v,r)>\delta$, and that supports the following four operations:*

- *Delete an edge uv from G, and update D accordingly.*
- *List all vertices of D.*

- *List all vertices $C \subseteq D$ that are unreachable from or cannot reach r in G.*
- *Delete a vertex $v \in D$ from both D and G (together with its incident edges).*

Note that D includes all vertices that are unreachable from or cannot reach r. In our algorithm we only access the in-out ES-trees by running the two query operations given in Definition 1. We let BUILD-ES-TREE(G, r, δ) be a function that builds an in-out ES-tree $\mathcal{E} = (G, r, \delta, D)$ for G. We say that the depth invariant of $\mathcal{E}$ is *violated* if the set D is nonempty. Observe that if the depth invariant is *not* violated, then the graph G represented by $\mathcal{E}$ is strongly connected. The following lemma follows straightforwardly from the implementation of an in-out ES-tree.

Lemma 2. *Let G be a directed graph, $n = |V(G)|$, $m = |E(G)|$, $r \in V(G)$, and $\delta > 0$.*

- *The total time spent building an in-out ES-tree $\mathcal{E} = (G, r, \delta, D)$ and handling all delete operations is $O(m\delta)$.*
- *The running time of each operation that lists vertices of D is linear in the size of D.*
- *The running time of each operation that lists vertices of $C \subseteq D$ that are unreachable from or cannot reach r in G is linear in the size of D and the total degree of vertices of D (in G).*

B. SCC-decomposition

Łącki [13] introduced a decomposition of a strongly connected graph G that can be efficiently updated under edge deletions. To solve the decremental SCC problem, such a decomposition can be maintained for each SCC. His idea was that, in terms of strong connectivity, a strongly connected subgraph may essentially be viewed as a single vertex from the point of view of the rest of the graph. We can thus contract strongly connected subgraphs into single vertices and separately maintain connectivity with respect to internal and external edges. We refer to Łącki's decomposition as an *SCC-decomposition*. An SCC-decomposition recursively partitions the graph G into smaller strongly connected subgraphs. This generates a rooted tree T, whose root r represents the entire graph, and where the subtree rooted at each node ϕ represents some strongly connected subgraph G_ϕ (we refer to vertices of T as nodes to distinguish T from G). Every internal (non-leaf) node ϕ is a vertex of G_ϕ, and the children of ϕ correspond to SCCs of $G_\phi \setminus \phi$. The algorithm by Łącki recursively partitions the graphs until the leaves represent single vertices. We cut the recursion short and instead operate with *partial* SCC-decompositions, in which the leaves represent strongly connected subgraphs rather than single vertices.

Definition 3 (SCC-decomposition)**.** *Let $G = (V, E)$ be a strongly connected graph. An* SCC-decomposition *of G is a rooted tree T, whose nodes form a partition of V. For a node ϕ of T we define G_ϕ to be the subgraph of G induced by the union of all descendants of ϕ (including ϕ). Then, the following hold:*

- *Each internal node ϕ of T is a single-element set.*[1]
- *Let ϕ be any internal node of T, and let $H_1, \ldots, H_t$ be the SCCs of $G_\phi \setminus \phi$. Then the node ϕ has t children $\phi_1, \ldots, \phi_t$, where $G_{\phi_i} = G[\phi_i] = H_i$ for all $i \in \{1, \ldots, t\}$.*

An SCC-decomposition of a graph G that is not strongly connected is a collection of SCC-decompositions of the SCCs of G. We say that T is a partial *SCC-decomposition when the leaves of T are not required to be singletons.*

Observe that for each node ϕ, the graph G_ϕ is strongly connected. Moreover, the subtree of T rooted at ϕ is an SCC-decomposition of G_ϕ. To build an SCC-decomposition T of a strongly connected graph G we pick an arbitrary vertex v, put it in the root of T, then recursively build SCC-decompositions of SCCs of $G \setminus \{v\}$ and make them the children of v in T. Since the choice of v is arbitrary, there are many ways to build an SCC-decomposition of the same graph. As shown in [13], the total initialization and update time of an SCC-decomposition is $O(m\gamma)$, where γ is the depth of the decomposition. Thus, it is desirable to build decompositions of low depth.

SCC-decomposition and separators: For any set $S \subseteq V(G)$ we can build a partial SCC-decomposition of G by picking vertices of S (in arbitrary order). This results in a tree T where the vertices of S are at the top, and where the leaves represent SCCs of $G \setminus S$. In the following we use BUILD-SCC-DEC(G, S) to refer to this procedure.

Lemma 4. *Let G be a directed graph and $S \subseteq V(G)$.* BUILD-SCC-DEC(G, S) *builds a partial SCC-decomposition T with $|S|$ internal nodes. Each leaf of T is exactly the vertex set of one SCC of $G \setminus S$. The procedure runs in $O(|E(G)| \cdot |S|)$ time.*

Let T be a partial SCC-decomposition of G and ϕ be a leaf node of T. Note that if we modify T by replacing ϕ by a partial SCC-decomposition of $G[\phi]$, we obtain a new partial SCC-decomposition of G (in this case we say that we *expand* ϕ). The node set of this new partial SCC-decomposition is a finer partition of vertices of G. If S is a small set of vertices, such that the SCCs of $G \setminus S$ are small, then the SCC-decomposition computed by BUILD-SCC-DEC(G, S) has low depth. Łącki [13] used this observation and the planar separator theorem to obtain an $O(n^{1.5})$ total update time algorithm for the decremental SCC problem in *planar* graphs. The key insight of our result is that we may find a suitable set S in graphs of high diameter. Thus, we expand each leaf ϕ of the SCC-decomposition when the diameter of G_ϕ grows large. Up to this point we maintain strong

[1] In this case, we sometimes abuse notation and assume that ϕ is the vertex itself.

Procedure Build-SCC-Dec(G,S)

Input: A strongly connected graph G and $S \subseteq V(G)$.
Output: A partial SCC-decomposition T of G whose internal nodes are exactly the vertices of S.

if $S = \emptyset$ **then**
  return the tree T consisting of a single node ϕ with associated graph $G_\phi = G$.
Pick an arbitrary vertex $v \in S$.
Make v the root of T, and let $G_v = G$.
Compute the SCCs $H_1, \ldots, H_t$ of $G \setminus \{v\}$.
foreach $i \in \{1, \ldots, t\}$ **do**
  Recursively compute $T_i = \text{BUILD-SCC-DEC}(H_i, S \cap V(H_i))$.
  Make the subtree T_i a child of v in T.
return T.

connectivity for the subgraph of the leaf with an in-out ES-tree, which is efficient due to the low diameter.

Handling deletions: We next very briefly sketch Łącki's procedure for maintaining an SCC-decomposition when an edge uv is deleted (see [13] for more details). Łącki does not consider partial SCC-decompositions, but his technique easily extends to this case since all updates propagate toward the root. In particular, a partial SCC-decomposition T of a graph G is isomorphic to an SCC-decomposition of the graph that is obtained from G by contracting the strongly connected subgraphs represented by the leaves of T.

The procedure for updating an SCC-decomposition T is a recursive function that takes a node ϕ of T as a parameter, and possibly makes a single recursive call with the parent of ϕ. Thus, the update process is bottom-up and affects some number of immediate ancestor of ϕ. When node ϕ is processed, some child node ϕ' (and its subtree) may be moved up in the tree T, i.e., it becomes a sibling of ϕ. Łącki shows that the running time of this step is proportional to the total degree of vertices in the subtree rooted at ϕ'. Moreover, the running time of such steps dominates the running time of the update procedure. Since nodes only move up in T, it follows that maintaining the SCC-decomposition T under edge deletions can be done in total time $O(\gamma m)$, where $\gamma \leq n$ is the initial depth of T.

In this paper we modify the SCC-decomposition by replacing subtrees of the SCC-decomposition with in-out ES-trees. We will call such an SCC-decomposition an *augmented SCC-decomposition*. Let T be an augmented SCC-decomposition. Whenever an edge uv is deleted, where u and v belong to distinct nodes of T, the in-out ES-trees are not affected at all. On the other hand, when u and v belong to the same leaf ϕ of T, we update the in-out ES-tree $\mathcal{E}_\phi$ for ϕ as follows. Let $H_1, \ldots, H_\ell$ be the SCCs of $G_\phi \setminus uv$. We first update $\mathcal{E}_\phi$ to represent the SCC H_i that contains the root of $\mathcal{E}_\phi$, and then call FIX-SCC-DEC, which is a function for propagating the update in an SCC-decomposition. This procedure is given SCC-decompositions of all $H_1, \ldots, H_\ell$, except H_i, as arguments. In this extended abstract, we do not describe the procedures for updating the internal nodes of an SCC-decomposition (e.g., FIX-SCC-DEC), as they are essentially the same as in [13].

III. OUR ALGORITHM

Our goal is to maintain the SCCs of a given directed graph G under edge deletions, and to answer in constant time whether (arbitrary) pairs of vertices are in the same SCC. To answer such queries we assign a unique identifier to each SCC and store it with all its vertices. To check whether two vertices are in the same SCC we simply check if their identifiers match. We use essentially the same algorithm to solve the decremental single-source reachability (SSR) problem: to reduce decremental SSR to decremental SCC, simply add an edge from every vertex back to the source. A vertex is now reachable from the source if and only if it is in the same SCC as the source in the modified graph.

Consider the algorithm that maintains an in-out ES-tree for each SCC of G. It uses the in-out ES-trees to detect when an SCC breaks, and then computes new in-out ES-trees for the resulting new SCCs. Roditty and Zwick [12] showed that when the root of every in-out ES-tree is chosen uniformly at random from its SCC, the total expected update time for a graph with n vertices and m edges is $O(mn)$. We use the same idea, but further reduce the update time by bounding the depth of the in-out ES-trees. (The use of random roots is also the only randomized element in our algorithm.) In particular, the depth of all in-out ES-trees will be upper bounded by $O(\sqrt{n})$. When the depth invariant of an in-out ES-tree is violated, we then need to maintain connectivity for the vertices that are too far from its root.

Consider the case when the deletion of an edge uv causes the depth invariant of an in-out ES-tree $\mathcal{E} = (H, r, \delta, D)$ to be violated. Similar to the simple algorithm, our algorithm computes the set of vertices $C \subseteq D$ that lost their strong connectivity to the root r of $\mathcal{E}$. It then removes C from $\mathcal{E}$ and recomputes in-out ES-trees for the new SCCs of $H[C]$ (using uniformly random sources). It is, however, possible that the depth invariant remains violated for $\mathcal{E}$ or the new in-out ES-trees after C is removed. Note, however, that $H \setminus C$ is strongly connected at this point, and so are the new SCCs by definition. The challenge is therefore to restore the depth invariant for an in-out ES-tree of a subgraph that we know is strongly connected. This is where we use Łącki's SCC-decomposition [13].

Let G^0 be the original graph for which our data structure is initialized, and let $n_0 = |V(G^0)|$. Throughout, we use n_0 to distinguish G^0 from other graphs and subgraphs that we operate with. We also introduce two parameters: $\delta_0 = 8\lceil\sqrt{n_0}\rceil$ and $q_0 = \lfloor\sqrt{n_0}/\log n_0\rfloor$. We will use depth threshold δ_0 for all our in-out ES-trees, and we refer to q_0 as a quality parameter. We will later explain the significance of δ_0 and q_0.

Procedure Fix-Depth($\mathcal{E}$)

Input: An in-out ES-tree $\mathcal{E} = (G, r, \delta_0, D)$ for a strongly connected graph G.
Output: An augmented SCC-decomposition T of G, consisting of a single tree.

if $D = \emptyset$ **then**
  Let T be the tree consisting of a single node ϕ with associated graph $G_\phi = G$, and let $\mathcal{E}_\phi = \mathcal{E}$.
  return T
if *More than half the vertices of G have distance at most $\delta_0/2$ to r, and more than half the vertices of G have distance at most $\delta_0/2$ from r* **then**
  Compute $S = \text{FIND-SEPARATOR}(G, r, \delta_0)$.
  Compute $T = \text{BUILD-SCC-DEC}(G, S)$.
  Let $\phi_1, \ldots, \phi_\ell \subseteq V(G)$ be the leaves of T.
  Compute $\{T_1, \ldots, T_\ell\} = \text{AUGMENTED-SCCS}(\mathcal{E}, \{G[\phi_1], \ldots, G[\phi_\ell]\})$
  foreach $i \in \{1, \ldots, \ell\}$ **do**
    Replace the leaf ϕ_i of T by the subtree T_i.
  return T.
else
  Destroy $\mathcal{E}$.
  return $\text{BUILD-BALANCED-SCC-DEC}(G)$.

Let G be a strongly connected graph, and let $\mathcal{E} = (G, r, \delta_0, D)$ be an in-out ES-tree for G. Recall that $T = \text{BUILD-SCC-DEC}(G, S)$, where $S \subseteq V(G)$, is a partial SCC-decomposition of G whose internal nodes are exactly the vertices of S. When the depth invariant of $\mathcal{E}$ is violated, our algorithm picks an appropriate set S, and computes the corresponding partial SCC-decomposition $T = \text{BUILD-SCC-DEC}(G, S)$. The leaves of T are the SCCs of $G \setminus S$, and as before our algorithm recomputes new in-out ES-trees for these SCCs, with the exception of the SCC H that contains r, if a majority of the vertices in H have distance greater than $\delta_0/2$ to or from r, respectively. (We explain later why this distinction is important.) The in-out ES-tree for H is instead obtained by removing all vertices not in H from $\mathcal{E}$. If the depth invariant is still violated for some of the resulting in-out ES-trees, the procedure is repeated recursively, extending the SCC-decomposition. In the end we are left with an augmented SCC-decomposition: A partial SCC-decomposition T for G, where each leaf ϕ of T has an in-out ES-tree $\mathcal{E}_\phi$ for $G[\phi]$. The procedure is described formally in $\text{FIX-DEPTH}(\mathcal{E})$. It calls the function $\text{AUGMENTED-SCCS}(\mathcal{E}, \{G[\phi_1], \ldots, G[\phi_\ell]\})$, which constructs augmented SCC-decompositions for the SCCs $G[\phi_1], \ldots, G[\phi_\ell]$, possibly by reusing the original in-out ES-tree $\mathcal{E}$ (which will be detailed later). Note that we did not yet describe how the set S is chosen, nor why we use $\delta_0 = 8\lceil\sqrt{n_0}\rceil$ as depth threshold.

Our data structure for maintaining SCCs under edge deletions is thus a partial SCC-decomposition where the leaves are represented by depth-bounded in-out ES-trees (an augmented SCC-decomposition). When the depth threshold is sufficiently high, the partial SCC-decomposition will have no internal nodes, and we get the simple algorithm by Roditty and Zwick [12]. On the other hand, if the depth threshold only allows in-out ES-trees consisting of single vertices, we get Łącki's algorithm [13]. Both algorithms use $O(mn)$ total update time, but for different reasons. To improve the running time we balance the depth of the partial SCC-decomposition and the depth of the in-out ES-trees against each other. This is achieved by carefully choosing the depth threshold for the ES-trees, and by carefully selecting the internal nodes for the partial SCC-decomposition. Note however that any choice of depth threshold and internal nodes gives a correct algorithm; these choices only affect the running time.

Before defining the choice of S in $\text{FIX-DEPTH}(\mathcal{E})$, we first describe how an augmented SCC-decomposition T is updated when an edge uv is deleted. We refer to $\text{DELETE-EDGE}(T, uv)$ for a formal description of the procedure. As in Łącki's algorithm [13], we first find the lowest common ancestor ϕ of u and v, and if ϕ is an internal node we proceed as in Łącki's algorithm (we refer to this procedure as $\text{DELETE-EDGE-FROM-SCC-DEC}(T, uv)$). If ϕ is a leaf, then we remove uv from the corresponding in-out ES-tree $\mathcal{E}_\phi = (G[\phi], r_\phi, \delta_0, D_\phi)$, and if this does not break the depth invariant then we are done. If the depth invariant of $\mathcal{E}_\phi$ is violated, then we find the set C_ϕ of vertices that are not strongly connected to the root r_ϕ of $\mathcal{E}_\phi$ in $G[\phi]$. We compute the SCCs of $G[C_\phi]$, and compute an in-out ES-tree from a random root in every SCC. We also remove C_ϕ and all its incident edges from $\mathcal{E}_\phi$. (We ignore for now the case where $\mathcal{E}_\phi$ is destroyed and reconstructed.) We now have an in-out ES-tree for every SCC of $G[\phi] \setminus uv$, but since the depth invariant may be violated, we make a call to $\text{FIX-DEPTH}(\mathcal{E})$ for each such in-out ES-tree $\mathcal{E}$. This gives us an augmented SCC-decomposition of every SCC of $G[\phi] \setminus uv$. In fact this situation is virtually identical to the case in Łącki's algorithm [13] when an edge deletion breaks the strongly connected subgraph of some internal node, and we proceed correspondingly: The augmented SCC-decomposition $T_{\ell+1}$ that corresponds to the original in-out ES-tree $\mathcal{E}_\phi$ replaces ϕ in T, and the remaining augmented SCC-decompositions $T_1, \ldots, T_\ell$ are attached higher in the tree T by making a call to $\text{FIX-SCC-DEC}(T, uv, \phi', \{T_1, \ldots, T_\ell\})$, where ϕ' is the parent of ϕ in T. This is again done as in Łącki's algorithm. Note that if $G[\phi] \setminus uv$ is strongly connected (i.e., $C_\phi = \emptyset$) then this step is not needed. If ϕ has no parent and $G[\phi] \setminus uv$ has more than one SCC, then the new SCCs represented by $T_1, \ldots, T_\ell$ are also new SCCs of the original graph G, and the identifiers of the vertices in these new SCCs are updated.

It remains to explain our use of the depth threshold, our choice of internal nodes for the augmented SCC-decomposition, and when to destroy and recompute in-out ES-trees. In order to make efficient use of Łącki's SCC-decomposition, we must make sure that the (combined) initial depth is low. Recall from the discussion at the end

Procedure Delete-Edge(T,uv)

Input: An augmented SCC-decomposition T for a graph G, and an edge uv to be deleted from G. Every leaf ϕ of T is associated with an in-out ES-tree $\mathcal{E}_\phi = (G[\phi], r_\phi, \delta_0, D_\phi)$.

Output: An augmented SCC-decomposition T' for $G' = G \setminus uv$.

1 Let $\phi = LCA(u, v)$ be the lowest common ancestor of u and v in T.
2 **if** ϕ *is an internal node of* T **then**
3 **return** DELETE-EDGE-FROM-SCC-DEC(T,uv).
4 Remove uv from the in-out ES-tree $\mathcal{E}_\phi$.
5 **if** $D_\phi = \emptyset$ **then**
6 **return** T
7 Using Lemma 2, compute the set $C_\phi \subseteq \phi$ of vertices that are not strongly connected to r_ϕ in $G[\phi]$.
8 Compute the SCCs $H_1, \ldots, H_\ell$ of the graph $G[C_\phi]$ induced by C_ϕ.
9 Compute $H_{\ell+1} = G[\phi] \setminus C_\phi$.
10 Compute $\{T_1, \ldots, T_{\ell+1}\} =$ AUGMENTED-SCCS($\mathcal{E}_\phi, \{H_1, \ldots, H_{\ell+1}\}$).
11 Let ϕ' be the parent of ϕ in T, or let $\phi' = \top$ if ϕ has no parent.
12 Replace ϕ by $T_{\ell+1}$ in T.
13 **if** $\phi' = \top$ *or* $C_\phi = \emptyset$ **then**
14 **return** $T' = \{T, T_1, \ldots, T_\ell\}$.
15 **else**
16 **return** FIX-SCC-DEC(T,uv,ϕ',$\{T_1, \ldots, T_\ell\}$).

of Section II-B that this can be achieved by using good *separators*. We therefore introduce the following definition.

Definition 5 (q-separator)**.** *Let $G = (V, E)$ be a graph with n vertices, and let $q \geq 1$ be an integer. A* q-separator *for G is a non-empty set of vertices $S \subseteq V$, such that each SCC of $G \setminus S$ contains at most $n - q \cdot |S|$ vertices. We refer to q as the* quality *of S.*

For our application it is desirable for separators to have as high quality as possible. However, since no SCC contains fewer than one vertex, every q-separator S must have $q \cdot |S| < n$. We next describe a procedure FIND-SEPARATOR(G, r, δ) for computing a high-quality separator for a graph with a high diameter. The procedure starts by computing a BFS tree T rooted at r for either G or the reversed graph $\overleftarrow{G}$, such that T has depth at least δ. If no such tree exists the procedure returns the empty set. Define the ith layer $L_i \subseteq V(G)$ of T to be the set of vertices at distance i from the root v in T. We now consider two cases: Either the layers from 0 up to $\delta/2$ contain at least half the vertices of G, or they contain less than half the vertices of G. In the first case, we let j be the lowest index such that $q \leq j \leq \delta/2$ and $|L_j| \leq 2^{j/q-1}$. We show that such an index exists, and that L_j is a q-separator. We thus let FIND-SEPARATOR(G, r, δ) return L_j. In the second case, we similarly return the layer L_j, where j is the highest index such that $\delta/2 \leq j \leq \delta - q$ and $|L_j| \leq 2^{(\delta-j)/q-1}$.

Lemma 6. *Let G be a strongly connected graph with n vertices and m edges, let $r \in V(G)$, and let δ be an integer. Then the procedure* FIND-SEPARATOR(G, r, δ) *computes a q-separator for G with quality $q = \lfloor \delta/(2 \log n) \rfloor$ in $O(m)$ time if there exists a vertex $v \in V(G)$ with either $dist(r, v) \geq \delta$ or $dist(v, r) \geq \delta$. If no such vertex v exists then the procedure returns the empty set.*

Proof: Let T be the BFS tree that is produced by FIND-SEPARATOR(G, r, δ), i.e., T is rooted at r and has depth at least δ. If no such BFS tree exists, then the procedure returns the empty set as it should. We will assume for simplicity that δ is even.

Let $L_i \subseteq V(G)$ be the ith layer of T. Observe that since T is a BFS tree, each layer L_j separates the lower layers from the higher layers, that is, there are no edges from L_i to L_k for $i < j < k$. It follows that the largest SCC of $G \setminus L_j$ has size at most $\max\{\sum_{i<j} |L_i|, \sum_{k>j} |L_k|\}$. Assume that the first $\delta/2+1$ layers $L_0, \ldots, L_{\delta/2}$ contain at most half the vertices of G. (Otherwise we may reverse the order of the layers and use an analogous argument.) This means that for $j \leq \delta/2$, we have $\sum_{i<j} |L_i| < n/2 \leq \sum_{k>j} |L_k|$, and thus the largest SCC of $G \setminus L_j$ has size at most $\sum_{k>j} |L_k| = n - \sum_{i \leq j} |L_i|$. Let $q = \lfloor \delta/(2 \log n) \rfloor$, and note that $\lfloor q \log n \rfloor \leq \delta/2$. Let j be the lowest index such that $q \leq j \leq \lfloor q \log n \rfloor$ and $|L_j| \leq 2^{j/q-1}$. We show that such an index exists, and that L_j is a q-separator. The lemma then follows from the fact that L_j is computed in $O(m)$ time.

Assume that $|L_i| > 2^{i/q-1}$ for all $q \leq i < j$. Since $|L_i| \geq 1$ for all $i < q$, we have that $\sum_{i=0}^{j-1} |L_i| > q + \sum_{i=q}^{j-1} 2^{i/q-1} \geq q + \sum_{i=0}^{j-q-1} 2^{(i+q)/q-1} = q + \sum_{i=0}^{j-q-1} 2^{i/q} = q + \frac{1-2^{(j-q)/q}}{1-2^{1/q}} = q + \frac{2^{j/q-1}-1}{2^{1/q}-1}$. Note that $(1+1/x)^x \geq 2$ for $x \geq 1$, which implies that $2^{1/x} - 1 \leq 1/x$. Hence, $\sum_{i=0}^{j-1} |L_i| > q + (2^{j/q-1} - 1) \cdot q = q \cdot 2^{j/q-1}$.

Assume by contradiction that there is no index j such that $q \leq j \leq \lfloor q \log n \rfloor$ and $|L_j| \leq 2^{j/q-1}$. Then $|L_i| > 2^{i/q-1}$ for all $q \leq i \leq \lfloor q \log n \rfloor$, which means that $\sum_{i=0}^{\lfloor q \log n \rfloor} |L_i| > q \cdot 2^{(\lfloor q \log n \rfloor + 1)/q-1} \geq \frac{1}{2} \cdot 2^{(q \log n)/q} = n/2$. Since $\lfloor q \log n \rfloor \leq \delta/2$, this contradicts the assumption that the layers $L_0, \ldots, L_{\delta/2}$ contain at most half the vertices of G, and therefore there must exist a lowest index j such that $q \leq j \leq \lfloor q \log n \rfloor$ and $|L_j| \leq 2^{j/q-1}$.

Observe that $|L_i| > 2^{i/q-1}$ for all $q \leq i < j$. When combined with $|L_j| \leq 2^{j/q-1}$, this implies that $\sum_{i=0}^{j-1} |L_i| > q \cdot 2^{j/q-1} \geq q \cdot |L_j|$. Since $\sum_{i=0}^{j-1} |L_i| \leq n/2$ this also implies that $|L_j| < n/(2q)$. It follows that the largest SCC of $G \setminus L_j$ has size at most $\sum_{k>j} |L_k| = n - \sum_{i \leq j} |L_i| \leq n - \sum_{i=0}^{j-1} |L_i| \leq n - q \cdot |L_j|$. Since $L_j \neq \emptyset$, this proves that L_j is a q-separator, and the lemma follows. ∎

Note that when we call FIND-SEPARATOR(G, r, δ) on line 5 in FIX-DEPTH($\mathcal{E}$) we use depth $\delta_0 = 8\lceil\sqrt{n_0}\rceil$, whereas we only use depth $\delta_0/4 = 2\lceil\sqrt{n_0}\rceil$ on line 2 in BUILD-BALANCED-SCC-DEC(G). It then follows from

Procedure Build-Balanced-SCC-Dec(G)

Input: A strongly connected graph G.
Output: An augmented SCC-decomposition T of G.

Pick an arbitrary vertex $v \in V(G)$.
Compute $S = \text{FIND-SEPARATOR}(G, v, \delta_0/4)$.
if $S = \emptyset$ **then**
  Pick $r \in V(G)$ uniformly at random.
  return $\text{FIX-DEPTH}(\text{BUILD-ES-TREE}(G, r, \delta_0))$.
else
  Compute $T = \text{BUILD-SCC-DEC}(G, S)$.
  Let $\phi_1, \ldots, \phi_\ell \subseteq V(G)$ be the leaves of T.
  Let $i_{\max} = \arg\max_{i \le \ell} |\phi_i|$.
  foreach $i \in \{1, \ldots, \ell\} \setminus \{i_{\max}\}$ **do**
    Pick $v_i \in \phi_i$ uniformly at random.
    Compute $\mathcal{E}_i = \text{BUILD-ES-TREE}(G[\phi_i], v_i, \delta_0)$
    Compute $T_i = \text{FIX-DEPTH}(\mathcal{E}_i)$.
    Replace the leaf ϕ_i of T by the subtree T_i.
  Compute $T_{i_{\max}} = \text{BUILD-BALANCED-SCC-DEC}(G[\phi_{i_{\max}}])$.
  Replace the leaf $\phi_{i_{\max}}$ of T by the subtree $T_{i_{\max}}$.
  return T.

Procedure Augmented-SCCs($\mathcal{E}, \{H_1, \ldots, H_\ell\}$)

Input: An in-out ES-tree $\mathcal{E} = (G, r, \delta_0, D)$, and a collection $\{H_1, \ldots, H_\ell\}$ of disjoint, strongly connected subgraphs of G.
Output: A collection of augmented SCC-decompositions $\{T_1, \ldots, T_\ell\}$, such that T_i is an augmented SCC-decomposition for H_i.

Let $i_{\max} = \arg\max_{i \le t} |H_i|$.
if $r \in \phi_{i_{\max}}$ **then**
  Remove $V(H_{i_{\max}})$ and its adjacent edges from $\mathcal{E}$.
  Compute $T_{i_{\max}} = \text{FIX-DEPTH}(\mathcal{E})$.
else
  Destroy $\mathcal{E}$, and compute $T_{i_{\max}} = \text{BUILD-BALANCED-SCC-DEC}(H_{i_{\max}})$.
foreach $i \in \{1, \ldots, \ell\} \setminus \{i_{\max}\}$ **do**
  Pick $v_i \in \phi_i$ uniformly at random.
  Compute $\mathcal{E}_i = \text{BUILD-ES-TREE}(H_i, v_i, \delta_0)$.
  Compute $T_i = \text{FIX-DEPTH}(\mathcal{E}_i)$.
return $\{T_1, \ldots, T_\ell\}$.

Lemma 6 that the produced separators always have quality at least $q = \lfloor \lceil \sqrt{n_0} \rceil / \log n \rfloor \ge \lfloor \sqrt{n_0} / \log n_0 \rfloor = q_0$. We use this to show that the combined increase in depth for the partial SCC-decomposition is at most $O(n_0/q_0) = O(\sqrt{n_0} \log n_0)$, which means that the total time spent on maintaining the SCC-decomposition is $O(m_0 \sqrt{n_0} \log n_0)$, where m_0 is the number of edges in the initial graph.

To bound the time spent on in-out ES-trees, one could try to argue that every time a new in-out ES-tree is created, it will contain at most half of the vertices from the graph that it came from. This would imply that every vertex only appears in a logarithmic number of in-out ES-trees. Since the contribution of one vertex to the work performed by an ES-tree is proportional to its degree multiplied by the maximum depth of the ES-tree, it would follow that the total time spent on maintaining ES-trees is $O(m_0 \delta_0 \log n_0) = O(m_0 \sqrt{n_0} \log n_0)$. Unfortunately, it is not true that every new in-out ES-trees has half the size of the graph that it came from. To fix the argument we exploit that the root of the in-out ES-tree was picked uniformly at random.

Consider the point at which half the vertices from the original graph G of an in-out ES-tree $\text{FIX-DEPTH}(\mathcal{E})$ have distance greater than $\delta_0/2$ to or from the root r of $\text{FIX-DEPTH}(\mathcal{E})$, respectively. (Note that this includes the case where half the vertices are unreachable, i.e., have infinite distance to or from r.) Since $r \in V(G)$ was chosen uniformly at random, with probability $1/2$ the situation would have been similar for half of the choices of roots. This restricts the structure of the graph, and we prove that in this case we can repeatedly remove q_0-separators from the largest SCC until it is reduced to half the size of the original graph. To do so we use the function $\text{BUILD-BALANCED-SCC-DEC}(G)$. We then get that each vertex only appears in a logarithmic number of in-out ES-trees in expectation. In order to repeat the argument, we destroy and recompute $\text{FIX-DEPTH}(\mathcal{E})$ before calling BUILD-BALANCED-SCC-DEC.

To initialize our data structure for some given graph G^0, we compute the SCCs $H_1, \ldots, H_\ell$ of G^0, and then for each $i \le \ell$, we compute an in-out ES-tree $\mathcal{E}_i$ with depth threshold $\delta_0 = 8\lceil \sqrt{n_0} \rceil$ from a random source r_i in H_i. For each $i \le \ell$ we then call $\text{FIX-DEPTH}(\mathcal{E}_i)$ to get an augmented SCC-decomposition.

IV. Decremental maintenance of SCCs

Let $G = (V, E)$ be the initial graph, and let G^j be the graph after j edges have been deleted from G. In particular $G^0 = G$. Also let T^j be the partial SCC-decomposition that our algorithm constructs for G^j. In this section we let $m_0 = |E(G^0)|$ and $n_0 = |V(G^0)|$. To bound the running time of the operations that modify our partial SCC-decomposition, we use the following function that assigns potential to a partial SCC-decomposition T of a graph G. We will separately bound the time spent on building and updating the ES-trees. Let $v \in V(G)$. We denote by $\text{level}_T(v)$ the depth in T of the node that contains v. Then, the potential function is given by $\Phi(T) := \sum_{v \in V(G)} \deg_G(v) \cdot \text{level}_T(v)$. The analysis by Łącki [13] can be interpreted as showing that the total time spent updating internal nodes of an SCC-decomposition T is $O(\Phi(T))$. This is done by charging work performed by the algorithm to a decrease of the potential. Let us first briefly analyze how each function affects the levels of vertices.

1) As noted in Section II-B the procedure for updating the internal nodes of a partial SCC-decomposition after deleting an edge, may only decrease the levels.
2) Each call to FIX-DEPTH may result in replacing a leaf node of an SCC-decomposition with a multi-level subtree computed with BUILD-SCC-DEC. Thus, these two operations may only increase the levels of vertices.

In [13], the SCC-decomposition is first built, which accumulates some potential, and then the potential may only drop. Thus, in order to bound the running time it suffices to bound the initial potential. In our case, we bound the total potential increase caused by calls to FIX-DEPTH and BUILD-SCC-DEC. The level of a vertex v only increases if v belongs to a leaf. When v is an internal node of the SCC-decomposition, its level only decreases.

There are two atomic operations that modify SCC-decompositions. The first one moves a node one level up (in the procedure for updating partial SCC-decomposition). The second consists of expanding a leaf node of an SCC-decomposition to a multilevel subtree computed with BUILD-SCC-DEC. The leaves of this subtree may then be recursively expanded with other subtrees. For the purpose of our analysis, we track the levels of nodes (and the potential) after each such atomic operation. Note that whenever we build an in-out ES-tree for a graph on $n \leq n_0$ vertices, we use the depth threshold δ_0. This allows us to find q_0-separators, where $q_0 = \lfloor \sqrt{n_0}/\log n_0 \rfloor$.

We now analyze BUILD-SCC-DEC(G, S). This function is called by FIX-DEPTH and BUILD-BALANCED-SCC-DEC, and it calls itself recursively. We refer to the first two cases as *initial* calls to BUILD-SCC-DEC. The following lemma follows from Lemma 6 and the fact that S is constructed by FIND-SEPARATOR(G, r, δ), where $\delta \geq \delta_0/4 = 2\lceil \sqrt{n_0} \rceil$, i.e., the procedure returns a q_0-separator, where $q_0 = \lfloor \sqrt{n_0}/\log n_0 \rfloor$.

Lemma 7. *In each initial call to* BUILD-SCC-DEC(G, S), *S is a q_0-separator of G, where $q_0 = \lfloor \sqrt{n_0}/\log n_0 \rfloor$.*

Lemma 8. *Let $v \in G^0$. The total increase in the level of v in the course of handling all edge deletions is $O(\sqrt{n_0}\log n_0)$.*

Proof: Observe that the level of v only increases when v belongs to a leaf of the partial SCC-decomposition, and this leaf is expanded to a partial SCC-decomposition built with a call BUILD-SCC-DEC(H, S). We claim that in this case the size of the node containing v decreases significantly. By Lemma 4, BUILD-SCC-DEC(H, S) builds an SCC-decomposition T' of depth at most $|S|$, where each leaf corresponds to an SCC of $H \setminus S$. By Lemma 7, S is a $\lfloor \sqrt{n_0}/\log n_0 \rfloor$-separator of H. Hence, each SCC of $H \setminus S$ has size at most $|V(H)| - \lfloor \sqrt{n_0}/\log n_0 \rfloor |S|$. Thus, when the leaf containing v is replaced with T', the level of v increases by at most $|S|$. Since the size of the set containing v is reduced by at least $\lfloor \sqrt{n_0}/\log n_0 \rfloor |S|$, the level of v can increase at most $\sqrt{n_0}\log n_0$ times. ∎

Corollary 9. *The total potential increase in the course of handling all edge deletions is $O(m_0\sqrt{n_0}\log n_0)$.*

The following lemma follows from Corollary 9, and the fact that FIND-SEPARATOR is guaranteed to find a q_0-separator when it is called from BUILD-SCC-DEC.

Lemma 10. *The total time spent in* FIND-SEPARATOR, BUILD-BALANCED-SCC-DEC *and* BUILD-SCC-DEC, *excluding the case when* FIND-SEPARATOR *returns the empty set in* BUILD-BALANCED-SCC-DEC, *is $O(m_0\sqrt{n_0}\log n_0)$.*

Lemma 10 accounts only for a part of the time spent by calls to FIX-DEPTH. We attribute the remainder of the work of FIX-DEPTH to handling in-out ES-trees, which we describe in Section IV-A. We also still need to bound the running time of FIX-SCC-DEC and DELETE-EDGE-FROM-SCC-DEC, but as it depends on the analysis of in-out ES-trees, we postpone it to Lemma 15.

A. ES-trees

We first show that the costs of all DELETE-EDGE operations on an ES-tree are dominated by the cost of the BUILD-SCC-DEC operation and the cost of all FIX-DEPTH operations. The proofs of the following two lemmas are deferred to the full paper.

Lemma 11. *Let $m_0 = |E(G_0)|$ and $n_0 = |V(G_0)|$. The total running time of all* FIX-DEPTH *operations (excluding the ones that exit immediately and take $O(1)$ time) is $O(m_0\sqrt{n_0}\log n_0)$.*

Lemma 12. *The total time spent inside* DELETE-EDGE *(that is, excluding the calls to* DELETE-EDGE-FROM-SCC-DEC, FIX-DEPTH, FIX-SCC-DEC, *and* BUILD-BALANCED-SCC-DEC*) is dominated by the time spent on building the ES-tree and calls to* FIX-DEPTH.

For every leaf ϕ of each partial SCC-decomposition we maintain an in-out ES-tree representing G_ϕ. As the in-out ES-trees in the leaf nodes are subdivided, new in-out ES-trees have to be built.

Lemma 13. *Let ϕ be a leaf of an augmented SCC-decomposition, let $n = |\phi|$, let $G[\phi]$ be the subgraph corresponding to ϕ at the time that the leaf was created, and let $\mathcal{E}_\phi = (G[\phi], r_\phi, \delta_0, D_\phi)$ be the in-out ES-tree for ϕ. In particular r_ϕ is a uniformly random vertex from ϕ. Then until $\mathcal{E}_\phi$ is destroyed, every new in-out ES-tree created for a subgraph of $G[\phi]$ contains at most $n/2$ vertices. Let $H_{\max}$ be the largest SCC of the graph for $\mathcal{E}_\phi$ right before $\mathcal{E}_\phi$ is destroyed, and let $T_{\max} =$ BUILD-BALANCED-SCC-DEC$(H_{\max})$. Then with probability $1/2$ the largest leaf of $T_{\max}$ contains at most $n/2$ vertices, and the SCCs other than $H_{\max}$ also contain at most $n/2$ vertices.*

Proof: We only sketch the proof. Additional details can be found in the full version of the paper.

Observe that we destroy $\mathcal{E}_\phi$ as soon as we cut away more than half of its vertices. Therefore, every new in-out ES-tree that is created from $G[\phi]$ before $\mathcal{E}_\phi$ is destroyed can at most contain $n/2$ vertices. Also, when $\mathcal{E}_\phi$ is destroyed, only the largest SCC $H_{\max}$ of its subgraph at that time can contain more than $n/2$ vertices. To prove the lemma it therefore

suffices to show that the largest leaf of $T_{\max}$ contains at most $n/2$ vertices with probability $1/2$.

Once the root r_ϕ of $\mathcal{E}_\phi$ is chosen, the remainder of our algorithm for handling $\mathcal{E}_\phi$ is deterministic, given a fixed sequence of edge deletions $e_1, e_2, \ldots, e_s$. For every choice of r_ϕ, we may therefore ask how many edges from the sequence are deleted before the algorithm destroys $\mathcal{E}_\phi$. In particular, we may sort the vertices of $G[\phi]$ in non-decreasing order according to the time it takes for the resulting in-out ES-tree to be destroyed. Let $v_1, v_2, \ldots, v_n$ be this sequence. For some choice of a root v_i, let $\mathcal{E}_i$ be the corresponding in-out ES-tree for $G[\phi]$, let $S_1^{(i)}, S_2^{(i)}, \ldots, S_{\ell_i}^{(i)} \subseteq \phi$ be the sequence of separators that are removed from $G[\phi]$ up to the point where $\mathcal{E}_i$ is destroyed, and let $t(i)$ be the number of edges $e_1, \ldots, e_{t(i)}$ that are deleted before this happens. (Note that $t(i) \leq t(j)$ for $i \leq j$.) Finally, let $G^{(i)}$ be what remains of $G[\phi]$ right before $\mathcal{E}_i$ is destroyed, i.e., $G^{(i)} = (G[\phi] \setminus (S_1^{(i)} \cup \ldots \cup S_{\ell_i}^{(i)})) \setminus \{e_1, \ldots, e_{t(i)}\}$.

Let $v_r = r_\phi$ be the chosen root. Observe that with probability $1/2$ at least $n/2$ vertices appear before v_r in the sequence $v_1, v_2, \ldots, v_n$. If this is not the case, then we ignore the outcome, so for the remainder of the proof we will assume that r appears in the second half of the sequence.

Recall that BUILD-BALANCED-SCC-DEC($H_{\max}$) repeatedly removes a separator from $H_{\max}$. We must therefore show that we can keep doing this until the largest SCC contains at most $n/2$ vertices. To do so we prove the following claim. Let S be any (possibly empty) subset of ϕ, and suppose that $G^{(r)} \setminus S$ has an SCC H with more than $n/2$ vertices. Then we show that for every $v_i \in V(H)$ with $i \leq n/2$, there exists some vertex $v \in V(H)$ such that either $dist(v_i, v) \geq \delta_0/2$ or $dist(v, v_i) \geq \delta_0/2$ in H. Since $|V(H)| > n/2$ there is at least one $v_i \in V(H)$, which implies that the diameter of H is at least $\delta_0/2$. For every $u \in V(H_{\max})$, there therefore exists some vertex $v \in V(H_{\max})$ such that either $dist(v, u) \geq \delta_0/4$ or $dist(u, v) \geq \delta_0/4$ in $H_{\max}$. Hence FIND-SEPARATOR($H_{\max}, u, \delta_0/4$) finds a q_0-separator for any $u \in V(H_{\max})$. To repeat the argument we let S be the union of the separators produced so far.

Suppose the largest SCC H of $G^{(r)} \setminus S$ has more than $n/2$ vertices. We then prove by contradiction that for every $v_i \in V(H)$ with $i \leq n/2$, there exists some vertex $v \in V(H)$ such that either $dist(v_i, v) \geq \delta_0/2$ or $dist(v, v_i) \geq \delta_0/2$ in H. Hence we assume that $dist(v_i, v) < \delta_0/2$ and $dist(v, v_i) < \delta_0/2$ for some $v_i \in V(H)$ and all $v \in V(H)$.

Consider the separators $S_1^{(i)}, S_2^{(i)}, \ldots, S_{\ell_i}^{(i)} \subseteq \phi$ that would have been generated if we had chosen v_i as the root of $\mathcal{E}_i$, and consider the SCC $H^{(i)}$ that contains v_i in $G^{(i)}$. We prove by induction that $S_j^{(i)} \cap V(H) = \emptyset$ for all $j \in \{1, \ldots, \ell_i\}$. This implies that H is a subgraph of $H^{(i)}$. Indeed, then H is also an SCC of $G' = (G^{(r)} \setminus S) \setminus (S_1^{(i)} \cup \ldots \cup S_{\ell_i}^{(i)})$, and since G' is a subgraph of $G^{(i)}$ it follows that v_i can only be part of a smaller SCC in G'. Since removing edges and vertices from a graph only increases the distance between pairs of vertices, it follows that distances to and from v_i in H are at least as large as in $H^{(i)}$. The key observation is then that $\mathcal{E}_i$ was destroyed because at most $n/2$ vertices had distance at most $\delta_0/2$ to or from v_i in $H^{(i)}$, respectively. Since the same is true for H, it follows from $dist(v_i, v) < \delta_0/2$ and $dist(v, v_i) < \delta_0/2$ for all $v \in V(H)$ that $|V(H)| \leq n/2$, which contradicts our assumption that $|V(H)| > n/2$.

We use a similar argument to prove that $S_j^{(i)} \cap V(H) = \emptyset$ for all $j \in \{1, \ldots, \ell_i\}$. The key property we use from our algorithm is that when we compute a separator on line 5 of FIX-DEPTH, more than half of the vertices have distance at most $\delta_0/2$ to and from the root of the in-out ES-tree, respectively. This means that in FIND-SEPARATOR the constructed separator only contains vertices with distance at least $\delta_0/2$ either to or from this root. ■

Lemma 14. *Let G be a graph, with $n = |V(G)|$ and $m = |E(G)|$. Assume that edges are deleted from G. For each SCC we maintain an in-out ES-tree up to depth $O(\sqrt{n})$ (n is the current size of the SCC) rooted in a vertex chosen uniformly at random. After edges are deleted and an SCC H is partitioned into $H_1, \ldots, H_k$ we do the following. Let W_i be the SCC containing the root of the in-out ES-tree $\mathcal{E}$ of H. If W_i contains at least $n/2$ vertices then we reuse the in-out ES-tree $\mathcal{E}$ of H in the SCC W_i and for $j \neq i$ we build new in-out ES-trees for H_j. If W_i contains less than $n/2$ vertices then we invoke the procedure* BUILD-BALANCED-SCC-DEC. *Then, maintaining all the in-out ES-trees and all calls to* BUILD-BALANCED-SCC-DEC *requires $O(m\sqrt{n}\log n)$ expected time.*

Proof: Maintaining an in-out ES-tree with depth threshold $\Theta(\sqrt{n})$ of a graph that initially has m edges and n vertices requires $O(m\sqrt{n})$ time. For simplicity, we may charge all this time to the initialization of the in-out ES-tree and ignore the constant factor. Thus, in the following we assume that this time is exactly $m\sqrt{n}$.

We bound separately the time spend for the calls of BUILD-BALANCED-SCC-DEC and the time for the ES-trees. Recall that we call BUILD-BALANCED-SCC-DEC in the case where W_i contains less than half of the vertices. Every recursive call of BUILD-BALANCED-SCC-DEC spends $O(m)$ time to find a $\lfloor\sqrt{n}/\log n\rfloor$-separator S. Since every time the largest SCC is at least $\lfloor\sqrt{n}/\log n\rfloor$ vertices smaller that the SCC it belonged previously, this can happen at most $\sqrt{n}\log n$ times. By Lemma 13 the probability that a call to FIND-SEPARATOR returns $S = \emptyset$ before BUILD-BALANCED-SCC-DEC halts is at most 1/2. Thus, in expectation the number of call to FIND-SEPARATOR is $O(\sqrt{n}\log n)$. This results in total $O(m\sqrt{n}\log n)$ expected time in the calls to FIND-SEPARATOR from BUILD-BALANCED-SCC-DEC. Additionally, all calls to FIX-DEPTH and BUILD-ES-TREE are bounded below. By Lemma 10, the time spent on calls of BUILD-SCC-DEC is bounded by $O(m\sqrt{n}\log n)$. Thus, the

total time spent on calls of BUILD-BALANCED-SCC-DEC, excluding the time for building and maintaining the in-out ES-trees, is $O(m\sqrt{n}\log n)$ in expectation.

We next analyze the time for all ES-trees. Note that we initiate new ES-trees only for SCCs that are of size at most half their previous SCC. Therefore, every node belongs to at most $O(\log n)$ ES-trees. Thus for all trees the total update time is $O(m\sqrt{n}\log n)$. In the second case, by Lemma 13 with constant probability all the SCCs are at most a constant fraction of the size of the original SCC. As mentioned above, w.h.p. this happens to every node at most $O(\log n)$ times. Thus this part also costs $O(m\sqrt{n}\log n)$ total update time. ■

It remains to bound the running time of FIX-SCC-DEC and DELETE-EDGE-FROM-SCC-DEC (the proof is deferred to the full version).

Lemma 15. *The total expected running time of* FIX-SCC-DEC *and* DELETE-EDGE-FROM-SCC-DEC *is* $O(m\sqrt{n}\log n)$.

B. Main result

Lemma 16. *Let G be a strongly connected graph subject to edge deletions, $n = |V(G)|$ and $m = |E(G)|$. Then, we can maintain SCC-decompositions of all SCCs of G in $O(m\sqrt{n}\log n)$ expected time.*

Proof: Consider the operations one by one. The total expected time spent on BUILD-ES-TREE and on updating all the ES-trees in the leaves of the SCC-decomposition is $O(m\sqrt{n}\log n)$ by Lemma 14. On the other hand, finding separators and all calls to BUILD-SCC-DEC take $O(m\sqrt{n}\log n)$ time, by Lemma 10. The running time of FIX-DEPTH is split between finding the separators, calling BUILD-ES-TREE and performing operations that are dominated by the running time of BUILD-ES-TREE. We have already accounted for each of these cases. The cost of DELETE-EDGE operations on in-out ES-trees can be charged to other operations, by Lemma 12. Finally, the total time spent in all the DELETE-EDGE operations executed on internal nodes of SCC-decompositions is bounded by $O(m\sqrt{n}\log n)$ by Lemma 15. Therefore, the total running time is $O(m\sqrt{n}\log n)$. ■

Theorem 17. *Let G be a directed graph, $n = |V(G)|$ and $m = |E(G)|$. There exists an algorithm for maintaining strongly connected components of G subject to edge deletions, that processes edge deletions in $O(m\sqrt{n}\log n)$ total expected time and at any point is able to answer queries that ask whether two vertices are contained in the same strongly connected component in $O(1)$ time. It uses $O(m+n)$ space.*

ACKNOWLEDGMENT

S. Chechik is supported by the Israel Science Foundation (grant no. 1528/15), T. D. Hansen by the Carlsberg Foundation, grant no. CF14-0617, G. F. Italiano by MIUR under Project AMANDA (Algorithmics for MAssive and Networked DAta), and J. Łącki by the EU FET project MULTIPLEX no. 317532 and the Google Focused Award on "Algorithms for Large-scale Data Analysis".

REFERENCES

[1] M. R. Henzinger and M. Thorup, "Sampling to provide or to bound: With applications to fully dynamic graph algorithms," *Random Struct. Algorithms*, vol. 11, no. 4, pp. 369–379, 1997.

[2] J. Holm, K. de Lichtenberg, and M. Thorup, "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity," *J. ACM*, vol. 48, no. 4, pp. 723–760, 2001.

[3] J. Holm, E. Rotenberg, and C. Wulff-Nilsen, "Faster fully-dynamic minimum spanning forest," in *Proc. of 23rd ESA*, 2015, pp. 742–753.

[4] M. Thorup, "Near-optimal fully-dynamic graph connectivity," in *Proc. of 32nd STOC*. ACM, 2000, pp. 343–350.

[5] C. Wulff-Nilsen, "Faster deterministic fully-dynamic graph connectivity," in *Proc. of 24th SODA*, 2013, pp. 1757–1769.

[6] B. M. Kapron, V. King, and B. Mountjoy, "Dynamic graph connectivity in polylogarithmic worst case time," in *Proc. of 24th SODA*, 2013, pp. 1131–1142.

[7] M. Henzinger, S. Krinninger, and D. Nanongkai, "Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs," in *Proc. of 46th STOC*, 2014, pp. 674–683.

[8] ——, "Improved algorithms for decremental single-source reachability on directed graphs," in *Proc. of 42nd ICALP*, 2015, pp. 725–736.

[9] S. Even and Y. Shiloach, "An on-line edge-deletion problem," *J. ACM*, vol. 28, no. 1, pp. 1–4, 1981.

[10] Y. Dinitz, "Dinitz' algorithm: The original version and Even's version," in *Theoretical Computer Science, Essays in Memory of Shimon Even*, 2006, pp. 218–240.

[11] G. F. Italiano, "Finding paths and deleting edges in directed acyclic graphs," *Inf. Process. Lett.*, vol. 28, no. 1, pp. 5–11, 1988.

[12] L. Roditty and U. Zwick, "Improved dynamic reachability algorithms for directed graphs," *SIAM J. Comput.*, vol. 37, no. 5, pp. 1455–1471, 2008.

[13] J. Lacki, "Improved deterministic algorithms for decremental reachability and strongly connected components," *ACM Transactions on Algorithms*, vol. 9, no. 3, p. 27, 2013.

[14] L. Roditty, "Decremental maintenance of strongly connected components," in *Proc. of 24th SODA*, 2013, pp. 1143–1150.

[15] V. King, "Fully dynamic transitive closure," in *Encyclopedia of Algorithms*. Springer, 2008.

[16] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.