Decremental maintenance of strongly connected components

Liam Roditty

Abstract

We consider the problem of maintaining the strongly connected components (SCCs) of an \( n \)-nodes and \( m \)-edges directed graph that undergoes a sequence of edge deletions. Recently, in SODA 2011, Łącki presented a deterministic algorithm that preprocess the graph in \( O(mn) \) time and creates a data structure that maintains the SCCs of a graph under edge deletions with a total update time of \( O(mn) \). The data structure answers strong connectivity queries in \( O(1) \) time. The worst case update time after a single edge deletion might be as large as \( O(mn) \). In this paper we reduce the preprocessing time and the worst case update time of Łącki’s data structure from \( O(mn) \) to \( O(m \log n) \). The query time and the total update time remain unchanged.

1 Introduction

We consider the problem of maintaining strongly connected components (SCCs) of a directed graph that undergoes a sequence of edge deletions (decremental updates). This problem is a fundamental problem in the area of dynamic graph algorithms. It is a key ingredient in several dynamic algorithms for maintaining reachability between vertices ("does \( u \) reaches \( v \) with a directed path?") of a directed graph [7, 4, 6, 2].

The problem of decremental maintenance of strongly connected components was first addressed explicitly by Frigioni et al. [2]. They presented an algorithm with a total update time of \( O(m^2) \), which is as bad as computing strongly connected components after each deletion from scratch using a static algorithm. They showed, however, that if all the deleted edges are chosen at random the expected total update time is \( O(mn) \).

Roditty and Zwick [7] presented a Las-Vegas algorithm with an expected total update time of \( O(mn) \). The adversary in the dynamic model that they considered is assumed to be oblivious to the random choices of the algorithm.

In a recent breakthrough Łącki [4] presented a deterministic algorithm with a total update time of \( O(mn) \), and thus solved the open problem posed by Roditty and Zwick in [7]. The advantage of Łącki’s result over the result of Roditty and Zwick in [7] is obvious, as it is a deterministic algorithm that works in a more general dynamic model with the same total update time. However, it suffers from two serious drawbacks. First, its preprocessing time is \( O(mn) \). Second, its worst case update time after the deletion of a single edge might be \( O(mn) \) as well. It is not to hard to see that the result of Roditty and Zwick in [7] does not suffer from these two drawbacks, as it is based on decremental maintenance of breath-first-search trees, which can be created in \( O(mn) \) time and updated after a single edge deletion in \( O(m) \) worst case time [3]. In this paper we close this remaining gap (up to a logarithmic factor) between Łącki’s result and the result of Roditty and Zwick by proving the following Theorem:

THEOREM 1.1. There is a deterministic algorithm that preprocesses a directed graph \( G = (V, E) \) in \( O(m \log n) \) time into a data structure of size \( O(m + n) \), that maintains the strongly connected components of \( G \) under edge deletions with a total update time of \( O(mn) \), a worst case update time of \( O(m \log n) \) and \( O(1) \) query time.

The preprocessing time and the worst case update time are important both from the practical and the theoretical perspectives. From the practical perspective, when we are given an input graph \( G \), we have no clue how many edges will be deleted from \( G \). In such a case investing \( O(mn) \) time in preprocessing the graph and creating a dynamic data structure might be very wasteful, if after that only a small number of edges are deleted from the graph. More specifically, if the preprocessing time takes \( O(mn) \) then as long as less than \( \sqrt{n} \) edges are deleted from the graph, it is more efficient to compute using a static algorithm the SCCs each time from scratch. It might be, however, that if we had the dynamic data structure then the actual cost of deleting these edges, excluding the expensive preprocessing time, was much smaller than computing SCCs from scratch after each edge deletion. Using our efficient preprocessing algorithm we avoid such a scenario and enjoy both from fast preprocessing time and from the efficiency of the dynamic data structure.

This problem is very interesting from the theoretical perspective. It is closely related to the problem of maintaining the transitive closure of a directed graph. The total update time for decremental maintenance of transitive closure is \( O(mn) \) [7, 4]. This is the best that one can hope
for, as long as the cost of computing the transitive closure of a graph is $O(mn)$. However, for the decremental SCCs problem there is no obvious reason why the total update time should be $O(mn)$, as the computation of SCCs takes $O(m)$ time. A prerequisite for obtaining a deterministic algorithm with a total update time of $O(mn^{1-\epsilon})$, for some $\epsilon > 0$, is to obtain a deterministic algorithm with $O(mn^{1-\epsilon})$ preprocessing time, as we present in this paper.

The question of efficient preprocessing time and worst case update time was addressed for other closely related dynamic problems. We mention here several examples. Roditty [5] presented a fully dynamic (edges are both added and deleted from the graph) algorithm for maintaining the transitive closure of a graph with a preprocessing time of $O(mn)$ and an amortized update time of $O(n^2)$. This improved upon the algorithm of Demetrescu and Italiano [1] whose preprocessing time is $O(n^3)$. Sankowski [8] obtained a fully dynamic algorithm for the same problem with a worst case update time of $O(n^2 \log^3 n)$. Thorup [9] obtained a fully dynamic algorithm with a total update time of $O(mn^{1-\epsilon})$.

The rest of this paper is organized as follows. In the next Section we provide some preliminary notations. In Section 3 we present the approach of Łącki [4] for decremental maintenance of SCCs. In Section 4 we present our new algorithms that obtain almost linear preprocessing and worst case update time.

2 Preliminaries

Let $G = (V, E)$ be a directed graph. Let $(V^{\text{sc}}, E^{\text{sc}}) = \text{SCC}(G)$ be the strongly connected components (SCCs) graph of $G$, that is, its vertices $V^{\text{sc}}$ represent SCCs of $G$ and its edges $E^{\text{sc}}$ are the projection of the endpoints of edges of $G$ that connect different SCCs of $G$ to the SCCs. With a slight abuse of notations we will treat vertices of $V^{\text{sc}}$ also as vertex sets of $G$, that is, for a vertex $C \in V^{\text{sc}}$ and a vertex $u \in V$ in the SCC of $G$ that $C$ represents we may write $u \in C$. We also refer to a vertex $v$ as a set of a single vertex $\{v\}$. When $C$ itself is composed of a nesting of SCC, that is, $C = C_1, \ldots, C_k$ and each $C_i$ can be also an SCC, we denote with $\text{VL}(C)$ the vertices of of the original graph $G$ that belong to $C$. In such a case $C_i \in C$ means that $C_i$ is in the first level nested in $C$. For a vertex $v \in V$, we use $G \setminus v$ to denote the graph that is resulted from deleting $v$ and all its edges from $G$. For an edge $e \in E$, we use $G \setminus e$ to denote the graph that is resulted from deleting $e$ from $G$.

3 Łącki’s approach to decremental strong connectivity maintenance

In this Section we describe Łącki’s approach to decremental strong connectivity maintenance. The description is biased towards our needs in this paper. The reader is referred to [4] for more information. Let $G = (V, E)$ be a strongly connected directed graph. We now describe a split process that creates a directed acyclic graph (DAG) from $G$ that can be used as a strong connectivity certificate for $G$.

Definition 1. (A Split Graph) Let $v \in V$ be an arbitrary vertex of $G$. Let $(V^{\text{sc}}, E^{\text{sc}}) = \text{SCC}(G \setminus v)$. The split graph $G_v = (V_v, E_v)$ is defined as follows. The vertex set $V_v$ equals to $V^{\text{sc}} \cup \{v_{in}, v_{out}\}$. The edge set $E_v$ equals to $E^{\text{sc}} \cup H$, where

$$H = \{(C, v_{in}) \mid C \in V^{\text{sc}} \land \exists(u, v) \in E \land u \in C\} \cup \{(v_{out}, C) \mid C \in V^{\text{sc}} \land \exists(v, u) \in E \land u \in C\}.$$

An edge $(a, b) \in E$ is mapped to an edge in $E_v$ if $a = v$ or $b = v$ or $a \in C$ and $b \in C'$, where $C, C' \in V^{\text{sc}}$ and $C \neq C'$.

It is straightforward to see that $G_v$ is a DAG. We refer to $G_v$ as the split graph for $G$ around vertex $v$. The next simple observation shows that $G_v$ can serve as a certificate for the strong connectivity of $G$.

Observation 1. Let $G = (V, E)$ be a strongly connected graph and let $v \in V$ be an arbitrary vertex of it. Let $G_v = (V_v, E_v)$ be the split graph around $v$ of $G$. The graph $G$ is strongly connected if and only if $v_{out}$ can reach every vertex in $G_v$ and every vertex of $G_v$ can reach $v_{in}$.

Using the definition of a split graph we can obtain an hierarchical decomposition for a strongly connected graph $G = (V, E)$. We pick an arbitrary vertex $v \in V$ for which we construct a split graph $G_v$. The root of the hierarchy contains the following information:

- The component name $V$
- The split vertex $v$
- The split graph $(V_v, E_v)$
- The edges of $E$ that are mapped to edges of $E_v$

Let $(V^{\text{sc}}, E^{\text{sc}}) = \text{SCC}(G \setminus v)$, where $V^{\text{sc}} = \{C_1, C_2, \ldots, C_k\}$. Next, we create a node in the hierarchy for every $C \in V^{\text{sc}}$. This is done by picking an arbitrary vertex $s \in C$ and computing the split graph $G_s$. In the node of the hierarchy that is formed for $C$ we maintain the same information as for the root. We assume that
each node contains a pointer to its parent and pointers to all its children. The process proceeds in a recursive manner at every node of the hierarchy whose split graph contains a vertex that corresponds to a set of two or more vertices from $V$. The leaves of the hierarchy are nodes that every vertex in their split graph vertex set contains a single vertex of $G$. For a given SCC of size $\ell$ in the hierarchy it is possible to list its vertices in $O(\ell)$ time. A pseudo-code that summarizes the algorithm is given in Algorithm 3.1.

ALGORITHM 3.1. PreProcess($G^{\text{sp}}=(V^{\text{sp}}, E^{\text{sp}})$)

Let $v$ be an arbitrary vertex from $V^{\text{sp}}$;
$(V^{\text{sc}}, E^{\text{sc}}, E^{\text{Intra}}) \leftarrow \text{SCC}(G^{\text{sp}} \setminus v)$;
for all $C \in V^{\text{sc}}$
  $p(C) \leftarrow V^{\text{sp}}$;
end for
$s(V^{\text{sp}}) = v$;
$V^{\text{sp}} \leftarrow V^{\text{sc}} \cup \{v_{in}, v_{out}\}$;
$E^{\text{sp}} \leftarrow E^{\text{sc}} \cup \{(v_{in}, C) | (v, x) \in E^{\text{sp}} \land x \in C\} \cup \{(C, v_{out}) | (x, v) \in E^{\text{sp}} \land x \in C\}$;
for all $C \in V^{\text{sc}}$
  PreProcess($C, E^{\text{Intra}}(C)$, where $E^{\text{Intra}}(C) \subseteq E^{\text{sp}}$
  is the set of edges with both endpoints in $C$);
end for

Next, we analyze the time that it takes to construct the hierarchy.

LEMMA 3.1. The time that it takes for Algorithm 3.1 to construct the hierarchy is $O(mn)$ in the worst case.

Proof. Let $h$ be the depth of the hierarchy and let $C$ be an SCC at depth $i$ of the hierarchy, where $1 \leq i \leq h$. The dominant cost in the creation of the node of $C$ in the hierarchy is the cost of computing its split graph $G_s$ around an arbitrary vertex $s \in C$. This cost is linear in the number of edges of $E$ with both endpoints are in $C$. Moreover, an edge cannot be in more than one component of level $i$. As an edge is scanned exactly once in every level and there are $h$ levels, the total cost is $O(mh)$. In the worst case $h = n$, thus the worst case running time is $O(mn)$.

In [4] Łącki shows that such an hierarchy can be maintained under edge deletions in $O(mn)$ total update time. The worst case update time of a single edge deletion is $O(mn)$ as well.

4 An almost linear time algorithm for constructing the split graphs hierarchy

4.1 The algorithm We now present a new algorithm for constructing the hierarchy of split graphs. Let $G = (V, E)$ be a strongly connected directed graph and let $V = \{v_1, v_2, \ldots, v_n\}$. Assume that $id(v_i) = i$ for every $1 \leq i \leq n$. Our algorithm works as follows. It receives as an input a set of vertices $V^{\text{sp}}$ that made up an SCC and the set of edges $E^{\text{sp}}$ that belong to this SCC. The id of every vertex which is also a vertex of the graph $G$ is either its original id or $\infty$. The id of a vertex that is in itself an SCC (either of $G$ or of other SCCs) is $\infty$. Let $V^{\text{VRTX}} \subseteq V^{\text{sp}}$ be the set of vertices with $id$ different from $\infty$ and let $V^{\text{VRTX}} \subseteq V^{\text{VRTX}}$ be the $\lceil |V^{\text{VRTX}}|/2 \rceil$ smallest ids vertices of $V^{\text{VRTX}}$. The set $V^{\text{HIGH}}$ contains the $\lfloor |V^{\text{VRTX}}|/2 \rfloor$ highest ids vertices of $V^{\text{VRTX}}$ and the rest of the vertices of $V^{\text{sp}}$, that is, those vertices with id $\infty$. Let $E^{\text{HIGH}} \subseteq E^{\text{sp}}$ be the set of edges with both endpoints in $V^{\text{HIGH}}$. The algorithm computes strongly connected components for the graph $(V^{\text{HIGH}}, E^{\text{HIGH}})$. We assume that the algorithm that computes strongly connected components returns the set of components $V^{\text{sc}}$, the set of inter component edges $E^{\text{sc}}$, and the set of intra component edges $E^{\text{Intra}} \subseteq E^{\text{HIGH}}$. Next, we create a new graph $(V^{\text{NEW}}, E^{\text{NEW}})$. Its set of vertices $V^{\text{NEW}}$ contains the vertices of $V^{\text{sc}}$ and the vertices of $V^{\text{LOW}}$. Its set of edges $E^{\text{NEW}}$ contains the set of inter component edges $E^{\text{sc}}$, the set $E^{\text{LOW}}$, and edges of $E^{\text{sp}}$ with one endpoint from $V^{\text{LOW}}$ and another from $V^{\text{HIGH}}$ which are represented in $E^{\text{NEW}}$ as edges between the endpoint from $V^{\text{LOW}}$ and the component of $V^{\text{sp}}$ that contains the endpoint from $V^{\text{HIGH}}$. The algorithm now checks what is the size of $V^{\text{LOW}}$ and proceeds according to it.

Case 1: $|V^{\text{LOW}}| = 1$.
In this case the node in the hierarchy that corresponds to the input SCC $V^{\text{sp}}$ can be obtained from a simple modification of the graph $(V^{\text{NEW}}, E^{\text{NEW}})$ as follows. Let $v \in V^{\text{LOW}}$. The split vertex $s(V^{\text{sp}})$ is set to $v$. The split graph $(V^{\text{sp}}, E^{\text{sp}})$ is built as follows. Its set of vertices $V^{\text{sp}}$ contains the set of vertices $V^{\text{sc}}$ and two additional vertices $v_{in}$ and $v_{out}$. Its set of edges contains the edges $E^{\text{sc}}$ and the edges of $E^{\text{NEW}}$, where each incoming edge of $v$ is replaced with an incoming edge to $v_{in}$ and each outgoing edge of $v$ is replaced with an outgoing edge from $v_{out}$. The parent in the hierarchy of each vertex of $V^{\text{sc}}$ is set to $V^{\text{sp}}$.

Case 2: $|V^{\text{LOW}}| > 1$.
In this case the algorithm performs a recursive call with the graph $(V^{\text{NEW}}, E^{\text{NEW}})$.

In both cases in its final step the algorithm iterates through the vertices of $V^{\text{sc}}$ and for each $C \in V^{\text{sc}}$ that is formed first at this call, that is, $C \notin V^{\text{sp}}$, it performs a recursive call with the graph $(C, E^{\text{Intra}}(C))$, where $C$ stands for the set of vertices of $V^{\text{HIGH}}$ that are contained in $C$ and $E^{\text{Intra}}(C)$ is the set of edges with both endpoints in $C$. A pseudo-code that summarizes the algorithm is given.
in Algorithm 4.1.

**ALGORITHM 4.1.** *PreProcess((V^{inp}, E^{inp}))*

\[
\begin{align*}
V^{\text{Vtx}} & \leftarrow \{v \mid v \in V^{\text{inp}} \land id(v) \neq \infty\} \\
V^{\text{Low}} & \leftarrow \{v \mid v \in V^{\text{Vtx}} \text{ and } v\text{'s id is among the } \\
\lfloor |V^{\text{Vtx}}|/2 \rfloor \text{ smallest ids}\} \\
V^{\text{High}} & \leftarrow V^{\text{Vtx}} \setminus V^{\text{Low}} \\
E^{\text{Low}} & \leftarrow \{(x, y) \mid (x, y) \in E^{\text{inp}} \land x \in V^{\text{Low}} \land y \in V^{\text{Low}}\} \\
E^{\text{High}} & \leftarrow \{(x, y) \mid (x, y) \in E^{\text{inp}} \land x \in V^{\text{High}} \land y \in V^{\text{High}}\} \\
\{V^{\text{Scc}}, E^{\text{Scc}}, E^{\text{intra}}\} & \leftarrow \text{Scc}(V^{\text{High}}, E^{\text{intra}}) \\
\text{for all } C \in V^{\text{Scc}} \text{ do } & \\
\quad \quad id(C) & \leftarrow \infty; \\
\text{end for } & \\
V^{\text{New}} & \leftarrow V^{\text{Low}} \cup V^{\text{Scc}} \\
E^{\text{New}} & \leftarrow \{(x, C) \mid x \in V^{\text{Low}} \land C \in V^{\text{Scc}} \land \exists y \in C \text{ s.t. } (x, y) \in E^{\text{inp}}\} \\
E^{\text{New}} & \leftarrow E^{\text{New}} \cup \{(C, x) \mid x \in V^{\text{Low}} \land C \in V^{\text{Scc}} \land \exists y \in C \text{ s.t. } (y, x) \in E^{\text{inp}}\} \\
E^{\text{New}} & \leftarrow E^{\text{New}} \cup E^{\text{intra}} \cup E^{\text{Low}} \\
\text{if } |V^{\text{Low}}| = 1 \text{ then } & \\
\quad \quad \text{for all } C \in V^{\text{Scc}} \text{ do } & \\
\quad\quad\quad p(C) & \leftarrow V^{\text{inp}} \\
\quad\quad \text{end for } & \\
\text{end if } & \\
\end{align*}
\]

The hierarchy is created by calling PreProcess with the input graph \(G = (V, E)\). In order to analyze the algorithm we define its execution tree.

**DEFINITION 2.** *(Execution Tree)* Each node in the execution tree corresponds to a call to PreProcess. We associate with each node \(N\) all the sets that Pre-Process gets as input as well as the sets that it computes during its execution. We add \(N\) as a subscript to denote this relation. The root of the tree corresponds to the initial call PreProcess\((V, E)\). The children of a node \(N\) of the tree are the nodes that correspond to calls to PreProcess that were initiated by the call of PreProcess that corresponds to \(N\).

In the next Lemma we prove an important property of the algorithm.

**LEMMA 4.1.** Let \(T\) be the execution tree of PreProcess\((V, E)\). Let \(C \subseteq V\) be an SCC of \(G\) and assume that there is a node \(N \in T\) such that \(VL(V^{\text{N}}) = C\). The algorithm computes a split graph for \(C\) using the minimal id vertex of \(C\).

**Proof.** Consider the path \(P_1, \ldots, P_t\) from \(N = P_1\) to a leaf \(F = P_t\), where \(P_i\) is the node that corresponds to the call PreProcess\((V_i^{\text{inp}}, E_i^{\text{inp}})\), for every \(2 \leq i \leq t\). Notice that for every \(P \in \{P_1, \ldots, P_t\}\) it holds that \(VL(V^{\text{P}}) = VL(V^{\text{P'}})\). For every \(P \in \{P_1, \ldots, P_{t-1}\}\) it holds that \(|V^{\text{Low}}_P| \leq |V^{\text{Low}}_P/2|\) and as we do not change the id of the smallest half of vertices (among those with id different from \(\infty\)) it follows that when we reach to \(F\) the set \(V^{\text{Low}}_F\) contains only the smallest id vertex of \(C\), and a split graph is constructed for \(C\) using this vertex.

**4.2 Running time** We will use the execution tree to analyze the running time of Algorithm 4.1.

**LEMMA 4.2.** Let \(T\) be the execution tree of PreProcess\((V, E)\). The running time of PreProcess\((V, E)\) is:

\[
O(\Sigma_{N \in T}|E^{\text{inp}}_N|).
\]

**Proof.** The cost of every call to PreProcess\((V^{\text{inp}}, E^{\text{inp}})\), excluding the recursive calls performed during the call, is \(O(E^{\text{inp}})\), as we do not have isolated vertices. Consider a leaf \(F \in T\). The cost of the call PreProcess\((V^{\text{inp}}, E^{\text{inp}}_F)\) is \(O(E^{\text{inp}}_F)\) as there are no recursive calls from PreProcess\((V^{\text{inp}}, E^{\text{inp}}_F)\). Thus, if we sum the costs over the execution tree from its leafs to the root we get that the running time of PreProcess\((V, E)\) is \(O(\Sigma_{N \in T}|E^{\text{inp}}_N|)\).

It stems from the above Lemma that in order to bound the running time of PreProcess\((V, E)\) it is enough to bound \(O(\Sigma_{N \in T}|E^{\text{inp}}_N|)\). In the next Lemma we establish a relation between the sets of edges \(E\) and the set of edges \(\bigcup_{N \in T} E^{\text{inp}}_N\).

**LEMMA 4.3.** Let \(N \in T\) and let \(C, C' \subseteq V^{\text{inp}}\). There is an edge \((C, C')\) in \(E^{\text{N}}\) if and only if there is an edge \((x, y)\) in \(E\) such that \(x \in VL(C)\) and \(y \in VL(C')\).

**Proof.** The proof is by an induction on the execution tree. The basis of the induction is proved for the root of the tree. As the input in this case is \(E\) the claim trivially holds. Assume now that the claim holds for some node \(N\). We show that the claim holds for its children as well. Consider a child that corresponds to a recursive call PreProcess\((C, E^{\text{intra}}(C))\) executed for an SCC \(C\) of \((V^{\text{intra}}, E^{\text{intra}})\). All vertices of \(C\) are also vertices of \(V^{\text{N}}\). Consider a pair \(A, B \in C\). By the induction hypothesis
there is an edge \((A, B)\) in \(E^N_N\) if an only if there is an edge \((x, y)\) in \(E\) such that \(x \in VL(A)\) and \(y \in VL(B)\). Such edges are the only edges placed by the algorithm in \(E^N_N(C)\).

Consider now the child that corresponds to the recursive call \(\text{PreProcess}(V^N_N, E^N_N)\). Let \(A, B \in V^N_N = V^N_{\text{SCC}} \cup V^N_{\text{LOW}}\). Assume first that both \(A\) and \(B\) are in \(V^N_N\). By the induction hypothesis there is an edge \((A, B)\) in \(E^N_N\) if and only if there is an edge \((x, y)\) in \(E\) such that \(x \in VL(A)\) and \(y \in VL(B)\). These edges are in \(E^N_N\) and are added to \(E^N_N\). No other edges between vertices of \(V^N_{\text{LOW}}\) are added to \(E^N_N\). Consider now the case that both \(A\) and \(B\) are in \(V^N_{\text{SCC}}\). There is an edge between \(A\) and \(B\) in \(E^N_N\) if and only if there is an edge \((A', B')\) in \(E^N_N\), where \(A', B' \in V^N_{\text{INP}}\), \(VL(A') \subseteq VL(A)\) and \(VL(B') \subseteq VL(B)\). By the induction hypothesis there is an edge \((A', B')\) in \(E^N_N\) if and only if there is an edge \((x, y)\) in \(E\) such that \(x \in VL(A')\) and \(y \in VL(B')\). Finally, consider the case that \(A \in V^N_{\text{INP}}\) and \(B \in V^N_{\text{SCC}}\). There is an edge between \(A\) and \(B\) in \(E^N_N\) if and only if there is an edge \((A', B')\) in \(E^N_N\), where \(B' \in V^N_{\text{INP}}\), and \(VL(B') \subseteq VL(B)\). By the induction hypothesis there is an edge \((A, B')\) in \(E^N_N\) if and only if there is an edge \((x, y)\) in \(E\) such that \(x \in VL(A)\) and \(y \in VL(B')\).

For every \(e \in E\), let \(M(e)\) be the total number of nodes in \(T\) that \(e\) is mapped to an edge in the input of their corresponding call. In the next Lemma we formulate the running time of the algorithm using the edges of \(G\).

**Lemma 4.4.** \(\Sigma_{N \in T} |E^N_N| \leq \Sigma_{e \in E} M(e)\)

**Proof.** From Lemma 4.3 it follows that for every \(N \in T\) and every \(e' \in E^N_N\) there is at least one edge \(e \in E\) that is mapped to \(e'\).

Our target now is to obtain a bound on \(M(e)\). The next Lemma and the Corollary that follows it are needed in order to obtain such a bound.

**Lemma 4.5.** Let \(N \in T\) and let \(N_1, \ldots, N_t \in T\) be the children of \(N\) in \(T\). If \(e \in E\) is mapped to an edge in \(E^N_N\) then there is at most one node \(N_i \in \{N_1, \ldots, N_t\}\) such that \(e\) is mapped to an edge in \(E^N_N\). Moreover, if \(e\) is not mapped to an edge in \(E^N_N\) then it is not mapped to an edge in the input of any child of \(N\).

**Proof.** Let \(e' \in E^N_N\) be the edge that \(e\) is mapped to. If \(e' \in E^N_N\) then \(e'\) is either belong to an SCC of \((V^N_{\text{INP}}, E^N_N)\) or between two SCCs. In the former case it will be passed as an input to the recursive call of the SCC that contains it, if this SCC is new. In the later case and in the case that \(e' \in E^N_N \setminus E^N_N\) the algorithm adds an edge to \(E^N_N\) that \(e'\) is mapped to and thus also \(e\). If \(|V^N_{\text{INP}}| > 1\) it implies that there is an edge that \(e\) is mapped to in the input of the recursive call for the graph \((V^N_{\text{INP}}, E^N_N)\). If \(|V^N_{\text{INP}}| = 1\) then the split graph is constructed and \(e'\) does not take part in any recursive call. We conclude that there is at most one recursive call whose input contains an edge that \(e\) is mapped to.

If \(e = (x, y)\) is not mapped to an edge in \(E^N_N\) then it cannot be that there is a pair of vertices \(C, C' \in V^N_N\) such that \(x \in VL(C)\) and \(y \in VL(C')\), since if this is the case it follows from Lemma 4.3 that there is an edge \((C, C') \in E^N_N\) which \(e\) is mapped to. As there is no pair of vertices \(C, C' \in V^N_N\) such that \(x \in VL(C)\) and \(y \in VL(C')\), every recursive call \(\text{PreProcess}(V^N_N, E^N_N)\) that is initiated from \(\text{PreProcess}(V^N_N, E^N_N)\), will not have a pair of vertices \(C, C' \in V^N_N\) such that \(x \in VL(C)\) and \(y \in VL(C')\). Thus, \(e\) is not mapped to an edge in \(E^N_N\), as required.

The next Corollary stems from the above Lemma:

**Corollary 4.1.** Let \(e \in E\) and let \(P_1 \in T\) be the deepest node that \(e\) is mapped to an edge in \(E^P_{\text{INP}}\). Let \(P_1, \ldots, P_t\) be a path from \(P_1\) to the root \(P_i\). Let \(Q \in T\). The edge \(e\) is mapped to an edge in \(E^Q_{\text{INP}}\), if and only if \(Q \in \{P_1, \ldots, P_t\}\).

**Proof.** Consider an arbitrary edge \(e \in E\). Let \(P_1\) be the deepest node in \(T\) that \(e\) is mapped to an edge in its input \(E^P_{\text{INP}}\). Let \(P_1, \ldots, P_t\) be a path in \(T\) between \(P_i\) and the root \(P_i\). By applying Lemma 4.5 to every \(P \in \{P_1, \ldots, P_{t-1}\}\), starting from \(P_i\), we get that \(e\) is mapped to an edge in \(E^P_{\text{INP}}\), for every \(P \in \{P_1, \ldots, P_{t-1}\}\). To prove that these are the only nodes that \(e\) is mapped to an edge in their input, assume on the contrary, that \(e\) is mapped to an edge in \(E^Q_{\text{INP}}\), for some \(Q \notin \{P_1, \ldots, P_t\}\). As before, from Lemma 4.5 it follows that \(e\) is mapped to an edge in \(E^Q_{\text{INP}}\), for every \(Q'\) on the path from \(Q\) to the root. This implies that there is a node in \(T\) that is common to both paths and \(e\) is mapped to the input of two of its children, a contradiction to Lemma 4.5.

We now turn to analyze the running time of Algorithm 4.1.

**Lemma 4.6.** For every \(e \in E\), \(M(e) \leq O(\log n)\).

**Proof.** Consider an arbitrary edge \(e \in E\). Let \(P_1 \in T\) be the deepest node that \(e\) is mapped to an edge in \(E^P_{\text{INP}}\). Let \(P_1, \ldots, P_t\) be the path between \(P_i\) and the root. From Corollary 4.1 it follows that \(e\) is mapped to an edge in \(E^P_{\text{INP}}\), if and only if \(P_i \in \{P_1, \ldots, P_t\}\). Let \(P_i \in \{P_1, \ldots, P_t\}\), where \(1 \leq i < t\). Consider the input \((V^P_{\text{INP}}, E^P_{\text{INP}})\).

There are two possible cases:
Case 1: \((V_{P_i}^{\text{NS}}, E_{P_i}^{\text{NS}}) = (C, E_{P_i}^{\text{INT}}(C))\), where \(C\) is an SCC of \(V_{P_i}^{\text{NS}}\).

In this case \(C\) contains vertices with id different from \(\infty\) that comes only from the set \(V_{P_i}^{\text{NS}} \setminus V_{P_i}^{\text{LOW}}\). The size of this set is \(\lfloor |V_{P_i}^{\text{NS}}|/2\rfloor\). Thus, \(|V_{P_{i+1}}^{\text{NS}}| \leq \lfloor |V_{P_i}^{\text{NS}}|/2\rfloor\). Recall that \(|V_{P_i}^{\text{LOW}}| = \lfloor |V_{P_i}^{\text{NS}}|/2 \rfloor\) and that \(|V_{P_i}^{\text{LOW}}| = \lfloor |V_{P_i}^{\text{NS}}|/2 \rfloor\). The later inequality implies that \(|V_{P_{i+1}}^{\text{NS}}| \leq |V_{P_i}^{\text{NS}}|/2\). In case that \(|V_{P_i}^{\text{NS}}|\) is even we get that:

\[|V_{P_i}^{\text{NS}}| \leq |V_{P_i}^{\text{NS}}|/2 \leq (\lfloor |V_{P_i}^{\text{NS}}|/2 \rfloor)/2 = |V_{P_i}^{\text{LOW}}|/2.\]

In case that \(|V_{P_i}^{\text{NS}}|\) is odd we get that:

\[|V_{P_i}^{\text{NS}}| \leq |V_{P_i}^{\text{NS}}|/2 \leq (\lfloor |V_{P_i}^{\text{NS}}|/2 \rfloor)/2 \leq (|V_{P_i}^{\text{LOW}}| + 1)/2 \leq |V_{P_i}^{\text{LOW}}|/2 + 1.\]

Case 2: \((V_{P_i}^{\text{NS}}, E_{P_i}^{\text{NS}}) = (V_{P_i}^{\text{NS}}, E_{P_i}^{\text{INT}}(C))\).

In this case \(V_{P_i}^{\text{NS}} = V_{P_i}^{\text{LOW}} \cup V_{P_i}^{\text{NEW}}\) and only \(V_{P_i}^{\text{NEW}}\) contains vertices with id different from \(\infty\). Thus, \(|V_{P_{i+1}}^{\text{NS}}| \leq |V_{P_i}^{\text{LOW}}| + 1\).

From the above two cases it follows that \(|V_{P_i}^{\text{LOW}}| \leq (|V_{P_i}^{\text{LOW}}| + 1)/2 + 1 \) for every \(P_i \in \{P_1, \ldots, P_{\ell-1}\}\). Thus, \(\ell \leq O(\log n)\) and \(M(e) \leq O(\log n)\).

We summarize the Section with the following Lemma:

**Lemma 4.7.** The running time of \(\text{PreProcess}(V, E)\) is \(O(m \log n)\).

**Proof.** From Lemma 4.2 it follows that the running time of \(\text{PreProcess}(V, E)\) is \(O(\sum_{e \in E} E_{P_i}^{\text{NS}})\). From Lemma 4.4 it follows that \(\sum_{e \in E} E_{P_i}^{\text{NS}} \leq E_{\text{INT}}(e).\) From Lemma 4.6 it follows that for every \(e \in E, M(e) \leq O(\log n)\). Thus, running time of \(\text{PreProcess}(V, E)\) is \(O(m \log n)\).

### 4.3 Correctness

The hierarchy of Láckí is constructed using Algorithm 3.1 in \(O(mn)\) time. The vertices are chosen in an arbitrary order. In order to define a specific hierarchy we change the way Algorithm 3.1 chooses vertices. Instead of picking an arbitrary vertex of an SCC and computing a split graph with it, the algorithm picks a vertex whose id is minimal among the vertices of the SCC. We denote with \(H = H(G)\) such an hierarchy that is constructed for a strongly connected input graph \(G\) using Algorithm 3.1. We will now show that Algorithm 4.1 computes the hierarchy \(H\) for the input \(G\). We refer to SCCs of \(H\) with their names, that is, \(C \in H\) but we also use \(C\) to denote the set of vertices from \(V\) that are in \(C\).

When we write \(\{v_1, \ldots, v_t\} \in H\) it means that there is an SCC \(C \in H\) such that \(C = \{v_1, \ldots, v_t\}\).

The next Lemma is a key ingredient in our later arguments.

**Lemma 4.8.** Let \(C \subseteq V\) be an SCC in \(H\). Let \(V_C = \{C_1, \ldots, C_t\}\) be a partition of the vertices of \(C\) such that each \(C_i \in V_C\) is an SCC of \(G\). Let \(E_C\) be the projection of \(E\) on \(V_C\). If \(A\) is an SCC computed by the execution of \(\text{PreProcess}(V_C, E_C)\) (Algorithm 4.1) then there is an SCC \(D \subseteq H\) such that \(D = \text{VL}(A)\).

**Proof.** Let \(D \subseteq \text{VL}(V_C^{\text{max}})\) be a maximal SCC in \(H\) of vertices from \(\text{VL}(V_C^{\text{max}})\), that is, every \(D' \in H\) such that \(D \subseteq D'\) satisfies \(D' \cap V_C^{\text{low}} \neq \emptyset\). Every \(v \in D\) is in some \(C_i \in V_C\), where \(1 \leq i \leq \ell\), and \(C_i \in V_C^{\text{max}}\). Thus, \(V_C^{\text{max}}\) contains a vertex \(A\) such that \(D \subseteq \text{VL}(A)\).

Let \(B = \text{VL}(A)\ \cup \ D\). For every \(B' \subseteq B \subseteq \text{VL}(V_C^{\text{max}})\) it cannot be that \(H\) contains an SCC \(D \cup B'\) as it contradicts the maximality of \(D\). As \(\text{VL}(V_C^{\text{max}})\) is an SCC in \(H\) there is an SCC \(H\) that contains \(D \cup B\). Let \(K \subseteq \text{VL}(V_C^{\text{max}})\) be the deepest SCC in \(H\) that contains \(D \cup B\). Let \(x \in K\) be the split vertex of \(K\) in \(H\). The split vertex is minimal among all vertices of \(K\), thus, it must be that \(x \notin \text{VL}(V_C^{\text{max}})\), as otherwise, \(K \subseteq \text{VL}(V_C^{\text{max}})\) and contradicts the maximality of \(D\). Consider \(K \setminus x\), this graph contains all vertices of \(\text{VL}(V_C^{\text{max}})\) and might contain even more, however, there is no SCC in this graph that contains \(D \cup B\). This contradicts the fact that \(A\), where \(\text{VL}(A) = D \cup B\) is an SCC of \(V_C^{\text{max}}, E^{\text{max}}\), since for every SCC \(S\) of \((V_C^{\text{max}}, E^{\text{max}})\), \(\text{VL}(S) \subseteq K \setminus x\) and there must be an SCC \(S' \subseteq K \setminus x\) such that \(\text{VL}(S) \subseteq S'\). Thus, \(\text{VL}(A) = D\).

Next, we use the above Lemma to prove that every SCC computed by Algorithm 4.1 is in \(H\).

**Lemma 4.9.** Let \(T\) be the execution tree of Algorithm 4.1 and let \(N \in T\). If \(C\) is an SCC in the graph \((V_N^{\text{max}}, E_N^{\text{max}})\) (or equivalently a vertex in \((V_N^{\text{max}}, E_N^{\text{max}})\)) then \(V_L(C) \subseteq H\).

**Proof.** The proof is by induction on the execution tree \(T\). Let \(R \in T\) be the root of \(T\). Algorithm 4.1 is applied to a graph \(G = (V, E)\) that is a strongly connected graph. The hierarchy \(H\) has \(V\) in its root as well. Thus, we can apply Lemma 4.8 and get that every \(C \in V_R^{\text{NS}}\) is in \(H\). Consider now a node \(N \in T\) and assume that the claim holds for every node on the path between \(R\) and \(N\). We will show that the claim also holds for every child of \(N\). There are two possible types of children for \(N\), nodes that correspond to a call on an SCC of \((V_N^{\text{max}}, E_N^{\text{max}})\) and a node that corresponds to the call on \((V_N^{\text{NS}}, E_N^{\text{NS}})\). We consider first the former type, that is, \(C \in V_N^{\text{NS}}\).
and $F \in T$ is a child of $N$ that corresponds to the call PreProcess$(C, E_N^{\text{intra}}(C))$. By the induction hypothesis it follows that $C \subseteq H$ and since it is an SCC we can apply Lemma 4.8 and get for every $D \in V_P^{\text{new}}$ that $\text{VL}(D) \in H$. Consider now the call PreProcess$(V_P^{\text{new}}, E_N^{\text{new}})$ and let the node $F$ be a child of $N$ that corresponds to it. Let $N'$ be the last node on the path from $N$ to the root that satisfies $\text{VL}(V_N^P) = \text{VL}(V_N^P)$. Let $N''$ be the parent of $N'$. It holds that $V_N^P$ is an SCC in $V_N^{\text{intra}}$. By applying the induction hypothesis to $N''$ we get that $\text{VL}(V_N^P) \in H$. This also implies that $\text{VL}(V_N^P) = \text{VL}(V_N^P) \in H$. As $V_F^P = V_N^{\text{new}}$ we get that $\text{VL}(V_F^P)$ is an SCC of $H$ and we can use Lemma 4.8 and get for every $D \in V_P^{\text{new}}$ that $\text{VL}(D) \in H$.

We now show that every SCC of $H$ is computed by Algorithm 4.1.

**Lemma 4.10.** Let $C \subseteq V$ be an SCC in $H$. There is a node $N \in V$ such that $D$ is an SCC of $(V_N^{\text{new}}, E_N^{\text{new}})$ and $\text{VL}(D) = C$.

**Proof.** Let $N$ be a the closest node to the root of $T$ such that $C \subseteq \text{VL}(V_N^P)$ and $|V_N^{\text{new}}|$ is the smallest possible. If $C = \text{VL}(V_N^P)$ and $N'$ is the parent of $N$ then $V_N^P$ is an SCC of $(V_N^{\text{new}}, E_N^{\text{new}})$ and the claim holds for $N'$, so assume that $C \subseteq \text{VL}(V_N^P)$.

Consider the path $P_1, \ldots, P_{\ell}$ from $N = P_1$ to a leaf $F = P_{\ell}$, where $P_i$ is the node that corresponds to the call PreProcess$(V_{P_i}^{\text{new}}, E_{P_{i-1}}^{\text{new}})$, for every $2 \leq i \leq \ell$. Notice that for every $P_i \in \{P_1, \ldots, P_{\ell}\}$ it holds that $\text{VL}(V_P^P) = \text{VL}(V_P^P)$. As $N$ is the node with the smallest $|V_N^{\text{new}}|$ that contains $C$, for every $P \in \{P_1, \ldots, P_{\ell}\}$ it must be that $V_P^{\text{new}} \cap C \neq \emptyset$ as otherwise there is an SCC $C'$ in $V_P^{\text{new}}$ such that $C \subseteq \text{VL}(C')$. The call with this $C'$ has an input of smaller size than $|V_N^{\text{new}}|$ and it contains $C$. Consider now the leaf $F$. In the leaf $|V_F^{\text{new}}| = 1$ and this vertex is in $C$ as well. Denote this vertex with $s$. This vertex has the minimal id in $\text{VL}(V_N^P)$. By Lemma 4.9 it follows that there is an SCC $C'$ in $H$ such that $C' = \text{VL}(V_N^P)$. The split vertex of $C'$ is $s$ as it has the minimal id. However, $s$ is also the minimal id vertex in $C$ (as $C \subseteq C'$) and has to be its split vertex as well. Thus, we reach to a contradiction as a vertex cannot be a split vertex of two different components in $H$.

We summarize this Section with the following Lemma:

**Lemma 4.11.** Running Algorithm 4.1 with the input $(V, E)$ computes the hierarchy $H$.

**Proof.** From Lemma 4.9 it follows that for every SCC $C$ that is computed during the whole execution of PreProcess$(V, E)$ it holds that $\text{VL}(C)$ is an SCC in $H$. From Lemma 4.10 it follows that for every $D \subseteq V$ which is an SCC of $H$ there exists a node $N \in T$ such that $D = \text{VL}(V_N^P)$. Thus, there is a node $N \in T$ with input $(V_N^{\text{new}}, E_N^{\text{new}})$ if and only if there is an SCC $D \subseteq H$ such that $D = \text{VL}(V_N^{\text{new}}) \subseteq V$. The root of $H$ is $V$ and the root of the hierarchy constructed by Algorithm 4.1 is also $V$. The parent-child relations are also the same. Consider $N \in T$ with input $(V_N^{\text{new}}, E_N^{\text{new}})$ that corresponds to $D \subseteq H$. From Lemma 4.1 the split vertex of $V_N^P$ is its minimal id vertex, which is also the split vertex of $D$.

### 4.4 Worst case update time

In this Section we prove a Lemma that allows us to use the faster preprocessing algorithm to obtain a worst case update time of $O(m \log n)$, while keeping the total update time $O(mn)$. We compute the hierarchy in $O(m \log n)$ time and use it to support edge deletions using the approach of Łącki. The main idea is that in order to avoid expensive updates, the update procedure of Łącki is monitored such that if it already run for $\Theta(m \log n)$ time, without completing the update, it is stopped and the new hierarchy is computed from scratch in $O(m \log n)$.

Let $G_1 = (V, E_1)$ and let $G_2 = G_1 \setminus e$, where $e \in E_1$ is an edge that is being deleted. In order to apply the above idea without increasing the total update time we have to show that the update procedure of Łącki when applied to $H(G_1)$ after the deletion of $e$ produces the hierarchy $H(G_2)$ which can be computed in $O(m \log n)$ time.

**Lemma 4.12.** Let $G_1 = (V, E_1)$ and let $G_2 = G_1 \setminus e$, where $e \in E_1$. If the update procedure of Łącki is operated on $H_1 = H(G_1)$ after the deletion of $e$ the resulted hierarchy is $H_2 = H(G_2)$.

**Proof.** Assume first that the root of $H_1$ was not decomposed. Let $H'$ be the hierarchy produced by the update procedure of Łącki. The root of $H'$ and $H_2$ is the same. The split vertex of the root is the same in $H'$ and in $H_2$, thus the split graph is the same, which implies that the children of the root of $H'$ are the same as the children of the root of $H_2$. Consider an arbitrary SCC $F$ of $H'$ and assume that for every SCC $F'$ on the path from the root to $F$ it holds that the children of $F'$ in $H'$ are the same as the children of $F'$ in $H_2$. We show that this is the case for $F$ as well. As the induction hypothesis holds for $F'$'s parent it follows that $F$ is also in $H_2$. Its split vertex in $H'$ is the same as its split vertex in $H_2$, thus the split graph is the same, and the children of $F$ in $H'$ are the same as the children of $F$ in $H_2$.

In case that the root is decomposed to $C_1, \ldots, C_t$ we have the hierarchies $H^{C_1}, \ldots, H^{C_t}$ and $H^{C_1}, \ldots, H^{C_t}$. Using similar arguments we can show that $H^{C_i} = H^{C_i}$, for every $1 \leq i \leq t$.  

Copyright © SIAM. Unauthorized reproduction of this article is prohibited.
4.5 Proof of Theorem 1.1 Given a directed graph $G = (V, E)$ we first compute its SCCs. For each SCC we apply Algorithm 4.1 to obtain an hierarchical decomposition of it. From Lemma 4.7 it follows that the running time of this stage is at most $O(m \log n)$. From Lemma 4.11 it follows that the hierarchy is correct and satisfies the invariant that the split vertex of every SCC in the hierarchy is its minimal id vertex.

We maintain the hierarchy using Łącki’s update procedure, while monitoring it, such that, if it already run for $\Theta(m \log n)$ time after a single edge deletion, without completing the update, it is stopped and the new hierarchy is computed from scratch in $O(m \log n)$. Since we have started from hierarchy that satisfies the minimal id invariant it follows from Lemma 4.12 that the hierarchy obtained from scratch is the same as the one that the update procedure of Łącki would have computed if it was run until its end. The main advantage of this approach is that we avoided an expensive operation without effecting the total update time. Thus, the total update time is $O(mn)$ and the worst case update time is $O(m \log n)$. The query time remains $O(1)$.

References