Faster Dynamic 2-Edge Connectivity in Directed Graphs

Loukas Georgiadis
University of Ioannina, Greece

Konstantinos Giannis
Gran Sasso Science Institute, L'Aquila, Italy

Giuseppe F. Italiano
LUISS University, Rome, Italy

Abstract

Let G be a directed graph with n vertices and m edges. We present a deterministic algorithm that maintains the 2-edge-connected components of G under a sequence of m edge insertions, with a total running time of $O(n^2 \log n)$. This significantly improves upon the previous best bound of $O(mn)$ for graphs that are not very sparse. After each insertion, our algorithm supports the following queries with asymptotically optimal efficiency:

- Test in constant time whether two query vertices v and w are 2-edge-connected in G.
- Report in $O(n)$ time all the 2-edge-connected components of G.

Our approach builds on the recent framework of Georgiadis, Italiano, and Kosinas [FOCS 2024] for computing the 3-edge-connected components of a directed graph in linear time, which leverages the minset-poset technique of Gabow [TALG 2016].

Additionally, we provide a deterministic decremental algorithm for maintaining 2-edge-connectivity in strongly connected directed graphs. Given a sequence of m edge deletions, our algorithm maintains the 2-edge-connected components in total time $n^{2+o(1)}$, while supporting the same queries as the incremental algorithm. This result assumes that the edges of a fixed spanning tree of G and of its reverse graph G^R are not deleted. Previously, the best known bound for the decremental problem was $O(mn \log n)$, obtained by a randomized algorithm without restrictions on the deletions.

In contrast to prior dynamic algorithms for 2-edge-connectivity in directed graphs, our method avoids the incremental computation of dominator trees, thereby circumventing the known conditional lower bound of $\Omega(mn)$.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Connectivity, dynamic algorithms, directed graphs

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.26

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

The design of dynamic graph algorithms is a classical area of research in theoretical computer science, where the input graph evolves through a sequence of updates, typically edge insertions and deletions. The goal of a dynamic algorithm is to update the solution to a problem more efficiently than recomputing it from scratch after each change. A problem is said to be *fully dynamic* if both insertions and deletions are allowed, and *partially dynamic* if only one type of update is permitted. The latter includes the *incremental* setting (only insertions) and the *decremental* setting (only deletions).

A fundamental problem in this domain is the computation and maintenance of edge-connected components in both undirected and directed graphs, driven by various practical and theoretical applications (see, e.g., [30]). Let $G = (V, E)$ be a strongly connected directed graph (digraph) with n vertices and m edges. An edge $e \in E$ is called a *strong bridge* if its removal disconnects the graph, i.e., $G \setminus e$ is no longer strongly connected. More generally, a subset $C \subseteq E$ is a *cut* if its removal disconnects the graph. If $|C| = k$, we refer to C as a k-sized cut of G. A directed graph is said to be k-edge-connected if it contains no $(k-1)$-cuts. Two vertices v and w are k-edge-connected, denoted $v \leftrightarrow_k w$, if there exist k edge-disjoint directed paths from v to w and k edge-disjoint directed paths from w to v. (Note that paths from v to w and from w to v need not be edge-disjoint with each other.) By Menger's Theorem [27], this is equivalent to requiring that every set of fewer than k edge deletions preserves strong connectivity between v and w. A *k-edge-connected component* of G is a maximal subset of vertices $U \subseteq V$ such that $u \leftrightarrow_k v$ for all $u, v \in U$. These components form a partition of V, since $\leftrightarrow_k$ is an equivalence relation [14].

Connectivity problems are significantly more challenging in directed graphs than in undirected ones (see, e.g., [8, 18, 23]). Until recently, it was known how to compute the k-edge-connected components of undirected graphs in linear time only for $k \leq 5$ [7, 10, 12, 20, 25, 28, 29, 33, 38]. In a very recent breakthrough, Korhonen [24] presented an $k^{O(k^2)}m$ time algorithm for computing the k-edge connected components of an undirected graph, which yields linear-time algorithms for any fixed k. In contrast, for directed graphs, linear-time algorithms are only known for $k \leq 3$ [13, 16, 33].

Despite significant progress in fully dynamic algorithms for several fundamental connectivity problems in undirected graphs (see, e.g., [19, 21, 22, 31, 37]), their directed counterparts remain substantially harder [4]. This difficulty is further underscored by conditional lower bounds [1, 17]. In particular, Abboud and Vassilevska [1] showed that any fully dynamic algorithm for maintaining whether a directed graph has more than two strongly connected components (SCCs) must incur $\Omega(m^{1-\epsilon})$ update or query time (for any constant $\epsilon > 0$) unless the Strong Exponential Time Hypothesis (SETH) fails. Due to such hardness results, much of the research has focused on partially dynamic scenarios. For the problem of dynamically maintaining the SCCs of a digraph, years of effort culminated in the following breakthrough results. For the decremental setting, Bernstein, Probst, and Wulff-Nilsen [4] developed a randomized algorithm (against an oblivious adaptive adversary) with $O(m \log^4 n)$ total expected time, while very recently van den Brand et al. [39] gave a deterministic algorithm with $m^{1+o(1)}$ total update time. For the incremental problem, also very recently, Chen et al. [6] gave a deterministic algorithm with $m^{1+o(1)}$ total update time. We note that the algorithms of [6, 39] explicitly maintain the SCCs, while [4] only supports queries of whether two vertices are strongly connected.

In this paper, we revisit the dynamic maintenance of 2-edge-connected components in directed graphs, a problem first explored by Georgiadis, Italiano, and Parotsidis [15], who presented an incremental algorithm with total time $O(mn)$ and space $O(m+n)$. After each insertion, their algorithm supports the following queries in asymptotically optimal time:

- *query*(v, w): Test in $O(1)$ time whether two vertices v and w are 2-edge-connected.
- *report*$()$: Report all 2-edge-connected components in $O(n)$ time.

Moreover, when the answer to a *query*(v, w) is negative, their algorithm returns in constant time a "witness", i.e., a strong bridge that appears in all paths from v to w or in all paths from w to v.

The decremental version was studied in [11], where a randomized algorithm with total running time $O(mn \log n)$ and space $O(n^2 \log n)$ was presented.

Our results

We present new deterministic, incremental and decremental algorithms for maintaining the 2-edge-connected components of a directed graph that significantly improve upon the prior time bounds for graphs that are not very sparse.

▶ **Theorem 1.** *We can maintain the* 2*-edge-connected components of a digraph with* n *vertices through a sequence of edge insertions in* $O(n^2 \log n)$ *total time. After each insertion, we can test in* $O(1)$ *time whether two vertices are* 2*-edge-connected and report the components in* $O(n)$ *time.*

We also achieve nearly the same asymptotic bound in the decremental setting, under certain assumptions on the edges that may be deleted.

▶ **Theorem 2.** *We can maintain the* 2*-edge-connected components of a strongly connected digraph with* n *vertices through a sequence of edge deletions in* $n^{2+o(1)}$ *total time, provided that the edges of a fixed spanning tree of* G *and a fixed spanning tree of the reverse graph* G^R *are not deleted. After each deletion, we can test in* $O(1)$ *time whether two vertices are* 2*-edge-connected and report the components in* $O(n)$ *time.*

The bound stated in Theorem 2 assumes that we maintain SCCs decrementally using the algorithm of van den Brand et al. [39]. If instead we use the algorithm of Bernstein, Probst, and Wulff-Nilsen [4], we obtain a randomized $O(n^2 \log^4 n)$-time algorithm that supports constant-time *query*(v, w) queries, but does not support *report*().

Both our incremental and our decremental algorithms require $O(n^2)$ space. Our results build on the recent framework of Georgiadis, Italiano, and Kosinas [13], which computes the 3-edge-connected components of a digraph in linear time using Gabow's *minset-poset* technique [8] to represent all minimum edge-cuts of a digraph. From these results, it follows that the 2-edge-connected components of a digraph G can be identified as the strongly connected components of two specially constructed *labeling graphs*. Although these labeling graphs can have size $O(mn)$, we show how to maintain a compact representation of size $O(n^2)$ that can be efficiently updated in the dynamic setting.

Our techniques are fundamentally different from those of [11, 15]. The algorithm of [15] relies on maintaining dominator trees incrementally, while [11] maintains the SCCs of $G \setminus v$ for every vertex v, using n instances of a decremental SCCs algorithm [26]. This also supports decremental dominator tree maintenance in $O(mn \log n)$ time and $O(n^2 \log n)$ space. Importantly, [11] showed that maintaining dominator trees incrementally or decrementally in total time $O((mn)^{1-\epsilon})$ (for some constant $\epsilon > 0$) is not possible unless the OMv Conjecture [17] fails. This bound applies even to algorithms that do not explicitly maintain the dominator tree but can answer parent queries. Consequently, both [11] and [15] are subject to this hardness barrier, whereas our new algorithms are not. However, unlike [15], our algorithms do not provide a witness edge when the answer to a *query*(v, w) is negative.

2 Preliminaries and notation

We assume that the reader is familiar with standard graph terminology. All graphs in this paper are directed, i.e., an edge $e = (u, v)$ in digraph G is directed from u, the *tail* of e, to v, the *head* of e. Also, we allow G to have multiple edges between the same pair of vertices. To simplify our bounds, we will assume that G may have no more than two copies of each edge, so that the number of edges is $m = O(n^2)$. (Notice that adding more than two copies of the same edge (u, v) does not affect 2-edge-connectivity.) If C is a set of edges, then C^R denotes the set of the reversed edges from C. We also let $G^R = (V, E^R)$ denote the reverse graph of $G = (V, E)$, i.e., the digraph that results from G after reversing the orientation of all edges. For a graph or a set of edges F, we use $V(F)$ to denote the set of the endpoints of the edges in F. If F consists of a single edge e, we may simply write $V(e)$ (which denotes the set of the endpoints of e). Also, for a graph F, we use $E(F)$ to denote the set of edges in F.

Let $G = (V, E)$ be a digraph. For any $S, T \subseteq V$, we denote the set of edges whose tail is in S and their head in T by $E(S, T) = \{(u, v) \mid (u, v) \in E, u \in S, v \in T\}$. We denote the set of outgoing edges from a set $S \subseteq V$ to the rest of the graph by $\delta(S) = E(S, V \setminus S)$, and the number of such edges by $out(S) = |\delta(S)|$. Similarly, we denote the set of incoming edges from the rest of the graph to S by $\rho(S) = E(V \setminus S, S)$ and the number of such edges by $in(S) = |\rho(S)|$. When S contains only a single vertex u, we slightly abuse notation and write $out(u), in(u)$ instead of $out(\{u\}), in(\{u\})$.

For a set of edges $E' \subseteq E$ and a set of vertices $S \subseteq V$, $\rho_{E'}(S)$ denotes the set of edges in E' that enter S from $V \setminus S$.

2.1 Flow graphs, dominators, and bridges

A *flow graph* is a directed graph with a distinguished *start vertex* s such that every vertex is reachable from s. Let $G = (V, E)$ be a strongly connected graph. Since G is strongly connected, all vertices are reachable from s and reach s, so we can view both G and G^R as flow graphs with start vertex s. To avoid ambiguities, throughout the paper, we will denote those flow graphs respectively by G_s and G_s^R. Let G_s be a flow graph with start vertex s. An edge (u, v) is a *bridge* of G_s if all paths from s to v include (u, v).[1] A vertex u is a *dominator* of a vertex v (*u dominates v*) if every path from s to v in G_s contains u. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree, the *dominator tree* D: u dominates v if and only if u is an ancestor of v in D. The dominator tree and the bridges of a flow graph can be computed in linear time [2, 5].

2.2 Cuts

A cut is a partition of the vertices of a graph into two disjoint subsets S, T. A cut determines a cut-set $E(S, T)$, since the removal of these edges makes all vertices in T unreachable from vertices in S. We may use the term "cut" interchangeably to denote either the partition (S, T) or the set of edges $E(S, T)$. (The meaning will always be clear from the context.) We say that a cut (S, T) is a k-sized cut if $out(S) = in(T) = k$; then we say S is a k-out set and T is a k-in set. A cut (S, T) is trivial if $|S| = 1$ or $|T| = 1$. A cut C *separates vertex s from vertex t* if all paths from s to t contain an edge in C. We refer to such a cut C as an *s-t cut.* Any partition of the vertices into two sets S and $T = V \setminus S$, such that $s \in S$ and $t \in T$ naturally defines an s-t cut. We also refer to the partition (S, T) as an s-t cut of G. The size of this cut is equal to $|E(S, T)|$, i.e., the number of edges directed from S to T. An *s-t mincut* is a cut (S, T) of minimum size such that $s \in S$ and $t \in T$. Consider a flow graph G_s and a k-in set T such that $s \notin T$. We call T a k-sized s-cut of G. Then, all paths from s to T contain an edge in $\rho(T)$. Clearly, any cut of G is an s-cut of G_s or an s-cut of G_s^R, which allows us to focus only on the s-cuts of G.

[1] Throughout the paper, to avoid confusion, we use consistently the term *bridge* to refer to a bridge of a flow graph and the term *strong bridge* to refer to a strong bridge in the original graph.

2.3 Trees and fundamental cycles

Let T be a rooted tree. Throughout, we assume that the edges of T are directed away from the root. For each directed edge (u, v) in T, we say that u is a parent of v (and we denote it by $t(v)$) and that v is a child of u. Every vertex except the root has a unique parent. If there is a (directed) path from vertex v to vertex w in T, we say that v is an ancestor of w and that w is a descendant of v, and we denote this path by $T[v, w]$. If $v \neq w$, we say that v is a proper ancestor of w and that w is a proper descendant of v, and denote by $T(v, w]$ the path in T to w from the child of v that is an ancestor of w.

A spanning tree T of a flow graph G_s with start vertex s is rooted at s and contains a unique path from s to any other vertex. For an edge $e \notin T$, we let $T(e)$ denote the fundamental cycle of e in T, i.e., the cycle that is formed in T when we add edge e, ignoring edge directions. Let $e = (u, v) \notin T$, and let z be the nearest common ancestor of u and v in T. Then $T(e) \setminus e$ consists of two directed paths, one from z to u, and another from z to v. (One of these paths may consist of a single vertex.) For a set $S \subseteq V(S)$, we let $LCA(S)$ denote the lowest common ancestor in T of all vertices in S.

Let G_s be a flow graph and let T be a spanning tree of G_s rooted at s. We let $\mathcal{N}$ denote the set of *non-tree edges* of G_s, i.e., $\mathcal{N} = E(G_s) \setminus T$. For any vertex $v \in G_s$, we let $\rho_{\mathcal{N}}(v)$ denote the set of non-tree edges that are incoming to v.

2.4 Minimum 1-in sets

Let G_s be a strongly connected digraph with a fixed start vertex s. A set of vertices S is called a 1-in set if $s \notin S$ and $in(S) = |E(V \setminus S, S)| = 1$. For every vertex $v \neq s$ that is contained in a 1-in set, we let $M(v)$ denote the (inclusion-wise) minimum 1-in set that contains v. We call this the M-set of v. Note that $M(v)$ is well-defined due to the submodularity of cuts. Specifically, we have the following.

▶ **Lemma 3.** *Let S and S' be two 1-in sets that contain a vertex $v \neq s$. Then, $S \cap S'$ is also a 1-in set.*

Proof. We have that $S \cap S'$ and $S \cup S'$ contain v but not s. By the submodularity of cuts, we have:

$$in(S \cap S') + in(S \cup S') \leq in(S) + in(S') \quad (1)$$

Since S and S' are 1-in sets, the right hand side of (1) equals 2. Since $S \cap S'$ and $S \cup S'$ are cuts that separate v from s, we have $in(S \cap S') \geq 1$ and $in(S \cup S') \geq 1$. Now (1) implies that $in(S \cap S') = 1$. This shows that $S \cap S'$ is a 1-in set. ◀

If v is not contained in a 1-in set, then we let $M(v) = V(G)$. We use $M_R(v)$ to denote the M-set of v in G_s^R. The importance of considering the M-sets (in both G_s and G_s^R) is demonstrated in the following proposition.

▶ **Proposition 4** ([13]). *Let G be a strongly connected digraph with a fixed start vertex s. Then, for any two vertices u and v, we have $u \leftrightarrow_2 v$ if and only if $M(u) = M(v)$ and $M_R(u) = M_R(v)$.*

According to Proposition 4, in order to compute the 2-edge-connected components of G, it is sufficient to compute the M-sets in G_s and G_s^R. As in [13], our approach is to compute the partition of $V(G)$ into the sets of vertices that have same M-set in G_s, and the sets of vertices that have the same M-set in G_s^R. It follows by Gabow [8], that this partition (w.r.t.

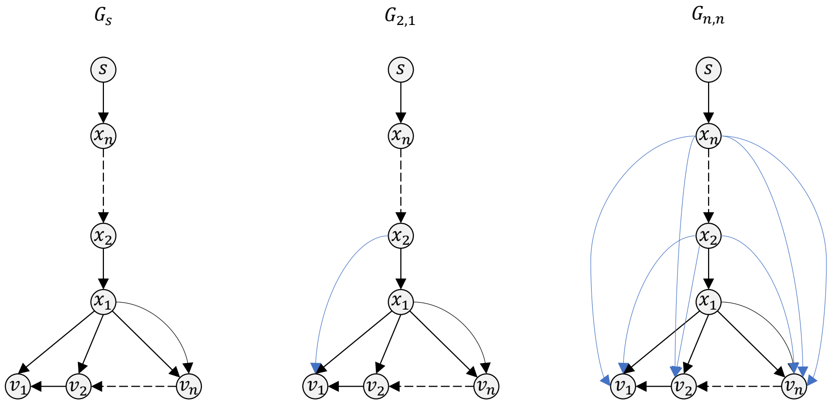

Figure 1 Example of flow graph and an insertion sequence that elicits $\Theta(n^2)$ changes of the $M(v_i)$ sets. Each graph $G_{k,l}$ is formed by adding the edges (x_i, v_j) for $1 \leq l \leq j$ and $2 \leq i \leq k$. Hence, $G_{k,l+1}$ results from $G_{k,l}$ after inserting the edge (x_k, v_{l+1}), and $G_{k+1,1}$ results from $G_{k,n}$ after inserting the edge (x_{k+1}, v_1). In G_s, we have $M(v_i) = \{x_1, v_i, v_{i+1}, \ldots, v_n\}$, for $1 \leq i \leq n$. In $G_{k,l}$, $M(v_i) = \{x_k, x_{k-1}, \ldots, x_1, v_i, v_{i+1}, \ldots, v_n\}$, for $1 \leq i \leq l$.

either G_s or G_s^R) corresponds to the strongly connected components of a *labeling graph LG*. So, our goal here is to show how to maintain this information efficiently in the incremental or decremental setting. Note that there are insertion sequences (or deletion sequences) that can cause $O(n^2)$ changes of the M-sets. See Figure 1.

First, we need to extend the definition of M-sets to edges of G_s as follows. Let e be an edge such that there is a 1-in set that contains both endpoints of e. Then $M(e)$ denotes the edge-set of the induced subgraph of the minimum 1-in set that contains both endpoints of e. To obtain the minimum 1-in set of each vertex v, [8] uses an augmented version of G_s, defined as follows. For every vertex $v \neq s$ of G_s, we introduce a new vertex v', two parallel edges of the form (v, v'), and two parallel edges of the form (v', v). Let us call $G_s^+ = (V, E)$ the resulting graph. Then, it follows that for every vertex $v \neq s$ of G_s, we have that $M(v)$ corresponds in a natural way to $M((v, v'))$ (in G_s^+).

Since, by Lemma 3, the $M(e)$ sets are closed under intersection, they admit a poset representation. For any edge $e \in E$, let $[e] = \{f : M(f) = M(e)\}$. We define the following relation on the edges of G_s: for $e, f \in E$, let $[e] \succ [f]$ if and only if $M(f) \subset M(e)$. Hence, $(\{[e]\}, \succ)$ forms a poset that represents all the minimum M-sets.

3 Computing minimum 1-in sets via Gabow's minset poset

Let $G = (V, E)$ be a strongly connected digraph, and let s be an arbitrary vertex of G. We view G as a flow graph with start vertex s. Recall that a 1-in set X is a set of vertices such that $X \subseteq V \setminus s$ and $in(X) = |E(V \setminus X, X)| = 1$. Consider the flow graph G_s, and let T be a spanning tree rooted at s. For simplicity, sometimes we slightly abuse notation and refer to T as the set of tree edges. For any set of vertices X, we denote by $T[X]$ the subgraph of T induced by X. We say that X is *cut* by T if $\rho(X) \subseteq T$ (that is, the only edges that enter X are tree edges of T), and X is *cospanned* by T if $T[X]$ is a tree. The next characterization follows from [8].

▶ **Lemma 5** ([8]). *Let G_s be a flow graph with start vertex s, and let T be a spanning tree of G_s. Any set of vertices $X \subseteq V \setminus s$ is a 1-in set of G_s if and only if it is cut and cospanned by T.*

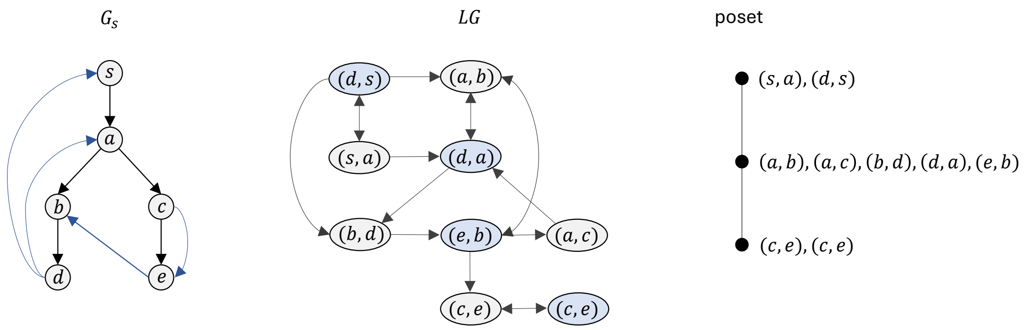

Figure 2 A strongly connected flow graph G_s, its corresponding labeling graph LG, and the minimum 1-in set poset. In G_s the tree edges are black and the non-tree edges are blue. In LG, the vertices that correspond to tree edges are gray and the vertices that correspond to non-tree edges are blue. We have $M((a,b)) = \{(a,b),(a,c),(b,d),(d,a),(e,b),(c,e),(c,e)\}$.

According to Lemma 5, if X is a 1-in set of G_s, then $T[X]$ is a tree with root r and the only edge entering X is the edge $(t(r), r) \in T$.

3.1 Labeling function and labeling graph LG

Gabow [8] defines a *labeling graph* LG with the property that the strongly connected components of LG correspond to the edges of G_s that have the same M-set. For any vertex $v \neq s$, there is at most one incoming edge to v that can be the incoming edge to $M(v)$, and the rest have the property that both of their endpoints also belong to $M(v)$. Thus, the idea is to assign a label to every edge e, which is essentially a set of pointers to edges that participate in the same M-set as e. Thus, the M-sets are given as the reachability sets of various edges with respect to this labeling. However, we cannot afford to explicitly compute the M-sets of all vertices, as their total size can be quadratic to $|V(G_s)|$.

In [8], the vertex set $V(LG)$ of LG consists of the edges of G, and the edges $(f,g) \in E(LG)$ are defined by a labeling function $\mathcal{L}_c : E \mapsto 2^E$. In our case, this labeling function becomes

$$\mathcal{L}_c(e) = \begin{cases} \rho_{\mathcal{N}}(u) \cup \rho_{\mathcal{N}}(v) & e = (u,v) \in T \\ T(e) & e \in \mathcal{N} \end{cases} \tag{2}$$

This function implies a labeling graph LG that has vertex set E (i.e., one vertex for each edge of G), and edges (e,f) where $e \in E$ and $f \in L(e)$. For $e, f \in E$, we say that f is a successor of e if there is a path from e to f in LG. The important property of the labeling graph LG is that the minimum set $M(e)$ of each edge e is equal to the set of all successors of e in LG [8]. This implies the following key proposition. (See Figure 2.)

▶ **Proposition 6** ([8, 13]). *Let G_s be a strongly connected flow graph with start vertex s, and let e and f be any two edges.*
(a) *If e is contained in a 1-in set, then $M(e) = M(f)$ if and only if e and f are strongly connected in LG.*
(b) *If e is not contained in a 1-in set, then e and f are strongly connected in LG if and only if f is also not contained in a 1-in set.*

By Proposition 6, we have that any two vertices v and w are 2-edge-connected if and only if (v,v') and (w,w') are strongly connected both in the labeling graph LG^+ of G_s^+ and in the labeling graph LG_R^+ of $(G_s^+)^R$.

Note that each tree edge $e = (v, w)$ has out-degree equal to $|\rho_{\mathcal{N}}(v)| + |\rho_{\mathcal{N}}(w)|$ in LG, while each non-tree edge f has out-degree equal to the number of tree edges in $T(f)$. Hence, LG has m vertices and $O(mn)$ edges. Nevertheless, Gabow [8] showed that the nodes $[e]$ of the corresponding poset $(\{[e]\}, \succ)$, which form a compact representation of these M-sets sufficient for our purposes, can be computed in $O(m)$ time by a clever implementation of an algorithm for computing the SCCs [7] of LG. (We remark that the node-finding algorithm of [8] does not compute the complete poset, which has $O(n)$ vertices and $O(n^2)$ edges.) The key idea is to use appropriate data structures, based on set merging [9], to avoid generating all edges of LG. This algorithm critically depends on a specific ordering of operations within the SCC algorithm, which makes it unsuitable for use in incremental or decremental settings.

4 Incremental algorithm

In this section, we present our incremental algorithm for maintaining the 2-edge-connected components of a digraph G. We consider, first, the case where G is strongly connected, and let T be a spanning tree of the corresponding flowgraph G_s, for an arbitrarily chosen start vertex s. Our algorithm operates on a modified labeling graph, denoted by $\widehat{LG}$, with the following properties: (i) $\widehat{LG}$ contains $O(n)$ vertices and $O(n^2)$ edges, (ii) the SCCs of $\widehat{LG}$ correspond to the poset nodes $[x]$ of the minimum 1-in sets of G_s, where $x \in V(G) \cup T$, and (iii) we can efficiently maintain $\widehat{LG}$ through a sequence of edge insertions.

4.1 Modified labeling graph

The modified labeling graph $\widehat{LG}$ is defined as follows. The vertex set of $\widehat{LG}$ consists of the tree edges of T and the vertices of G, i.e., $V(\widehat{LG}) = T \cup V(G)$. The edge set is defined by the following modified labeling function $\widehat{\mathcal{L}}_c : T \cup V(G) \mapsto 2^{T \cup V(G)}$:

$$\widehat{\mathcal{L}}_c(x) = \begin{cases} \{u, v\} & x = (u, v) \in T \\ \{e \in T : \exists f \in \rho_{\mathcal{N}}(v) \text{ such that } e \in T(f)\} & x = v \in V(G) \end{cases} \tag{3}$$

Hence, $(x, y) \in E(\widehat{LG})$ if and only if $y \in \widehat{\mathcal{L}}_c(x)$. Note that both LG and $\widehat{LG}$ are bipartite graphs, since any edge connects a tree edge $e \in T$ with a non-tree edge $f \in \mathcal{N}$ in the former, and a tree edge $e \in T$ with a vertex $v \in V(G)$ in the latter. While LG has m vertices and $O(mn)$ edges, $\widehat{LG}$ has $2n - 1$ vertices and $O(n^2)$ edges.

▶ **Lemma 7.** *For any two edges $f, g \in T$, g is reachable from f in LG if and only if g is reachable from f in $\widehat{LG}$.*

Proof. Suppose g is reachable from f in LG. Let P be a path from f to g in LG. Since LG is bipartite, the length of P is even. Consider two consecutive edges (x, y) and (y, z) on P, such that $x, z \in T$ and $y \in \mathcal{N}$. Also, let $x = (u, v)$. Then, by the definition of $\mathcal{L}_c$, $y \in \rho_{\mathcal{N}}(u) \cup \rho_{\mathcal{N}}(v)$, and $z \in T(y)$. Then, by the definition of $\widehat{\mathcal{L}}_c$, $\widehat{LG}$ contains (x, u) and (x, v), and either (u, z) or (v, z). In both cases, x reaches z in $\widehat{LG}$. Hence, it follows by induction on the length of P that if g is reachable from f in LG then g is also reachable from f in $\widehat{LG}$.

We show the contrapositive by similar arguments. Suppose g is reachable from f in $\widehat{LG}$. Let P be a path from f to g in $\widehat{LG}$. Since $\widehat{LG}$ is bipartite, the length of P is even. Consider two consecutive edges (x, y) and (y, z) on P, such that $x, z \in T$ and $y \in V(G)$. Also, let $x = (u, v)$. Then, by the definition of $\widehat{\mathcal{L}}_c$, $y \in \{u, v\}$, $z \in T(e)$ where $e \in \rho_{\mathcal{N}}(y)$. Then, by the definition of $\mathcal{L}_c$, LG contains the edges (x, e) and (e, z), and so x reaches z in LG. Hence, it follows by induction on the length of P that if g is reachable from f in $\widehat{LG}$ then g is also reachable from f in LG. ◀

An alternative but equivalent definition of $\widehat{LG}$ can be obtained by applying a sequence of vertex contractions to LG. Specifically, for each vertex $v \in V(G)$, we contract all non-tree edges in $\rho_{\mathcal{N}}(v)$ and subsequently eliminate any duplicate edges. While this may appear more intuitive, we adopt the original formulation as it enables the construction of $\widehat{LG}$ in $O(n^2)$ total time.

▶ **Corollary 8.** *For any two edges $f, g \in T$, $[f] = [g]$ if and only if f and g are strongly connected in $\widehat{LG}$.*

By Proposition 4 and Corollary 8, to determine whether two vertices u and v are 2-edge-connected in G, it suffices to check whether the edges (u, u') and (v, v') in the augmented graph G_s^+ are strongly connected in both the modified labeling graph of G_s^+ and that of its reverse $(G_s^+)^R$. The following lemma shows that, in fact, it is not necessary to work with the augmented graph explicitly.

▶ **Lemma 9.** *For any two vertices $u, v \in V(G)$, (v, v') is reachable from (u, u') in the modified labeling graph $\widehat{LG}^+$ of G_s^+ if and only if v is reachable from u in $\widehat{LG}$.*

Proof. Suppose $\widehat{LG}^+$ contains a path P from (u, u') to (v, v'). Since $\widehat{LG}^+$ is bipartite, and because the only non-tree edges entering (u, u') in G_s^+, except its copy (u, u'), are in $\rho_{\mathcal{N}}(u)$, the next vertex on P after (u, u') must be u. Consider now the penultimate vertex w on P. Then $w \in V(G)$, and from the definition of $\widehat{\mathcal{L}}_c$ (equation (3)), there is a non-tree edge $e \in \rho_{\mathcal{N}}(w)$ such that $(v, v') \in T(e)$. But this is possible only for $w = v$ and $e = (v', v)$. Hence, P contains a path from u to v, and so, $\widehat{LG}$ also contains a path from u to v.

Now suppose that $\widehat{LG}$ contains a path P from u to v. From the definition of $\widehat{\mathcal{L}}_c$ (equation (3)), $\widehat{LG}^+$ contains an edge from (u, u') to u. Also, since $(v', v) \in \rho_{\mathcal{N}}(v)$ and $(v, v') \in T((v', v))$, $\widehat{LG}^+$ also contains an edge from v to (v, v'). Hence, $\widehat{LG}^+$ contains a path from (u, u') to (v, v'). ◀

▶ **Corollary 10.** *Let G be a strongly connected digraph with a fixed start vertex s. Then, for any two vertices u and v, we have $u \leftrightarrow_2 v$ if and only if u and v are strongly connected in the modified labeling graph $\widehat{LG}$ of G_s and in the modified labeling graph $\widehat{LG}_R$ of G_s^R.*

4.2 Incremental construction of $\widehat{LG}$

Here we describe how to construct $\widehat{LG}$ incrementally as edges are added to a strongly connected graph G with a designated start vertex s. The labeling graph is defined with respect to a fixed spanning tree T of the flow graph G_s, rooted at s. (Similarly, we have a fixed spanning tree T_R of G_s^R, rooted at s, that defines the labeling graph of G_s^R.)

We initialize $\widehat{LG}$ by inserting a vertex for each $v \in V(G)$ and for each tree edge $e \in T$. Also, for each tree edge $e = (u, v)$, we add to $\widehat{LG}$ the edges (e, u) and (e, v).

Let v be a vertex of G, and let e be a tree edge of T. We say that e is *covered* by v if there is an edge $f \in \rho_{\mathcal{N}}(v)$ such that $e \in T(f)$, i.e., if $e \in \widehat{\mathcal{L}}_c(v)$. For each vertex $v \in V(G)$, we maintain the set of tree edges that are covered by v, using the following simple fact.

▶ **Lemma 11.** *Let v be a vertex such that $|\rho_{\mathcal{N}}(v)| \geq 1$, and let T_v be the set of tree edges that are covered by v. Then, T_v is a tree, rooted at the lowest common ancestor in T of all the vertices in $V(\rho_{\mathcal{N}}(v))$.*

Proof. Consider an edge $f \in \rho_{\mathcal{N}}(v)$. Then, v is contained in the fundamental cycle $T(f)$, so T_v is connected. Since $T_v \subseteq T$, T_v is a tree. Let $root_v$ be the root of T_v. If $\rho_{\mathcal{N}}(v)$ contains a single edge $e = (u, v)$, then T_v is rooted at $LCA(V(e))$. Now suppose $|\rho_{\mathcal{N}}(v)| > 1$. Let

$e = (u, v)$ and $e' = (w, v)$ be two edges in $\rho_{\mathcal{N}}(v)$. Let $z = LCA(u, v)$ and $z' = LCA(w, v)$. Since both z and z' are ancestors of v in T, we have $LCA(z, z') \in \{z, z'\}$. We conclude that $root_v = LCA(V(\rho_{\mathcal{N}}(v)))$. ◀

Lemma 11 allows us to use simple data structures to maintain T_v for each vertex v. Specifically, we do not maintain T_v explicitly, but keep a bit vector b_v, indexed by the vertices of $V(G)$ that indicates the vertices participating in T_v. Also, we store the current root of T_v. During the construction we maintain the following invariant (I): For any vertex u, we have $b_v[u] = 1$ if and only if $(t(u), u)$ is covered by v. Thus, $b_v[u] = 1$ if and only if $(t(u), u) \in \widehat{\mathcal{L}}_c(v)$, which means that $\widehat{LG}$ contains the edge $(v, (t(u), u))$. So, intuitively, we can view the bit vectors b_v as forming an adjacency matrix of the vertices $v \in V(G)$ in $\widehat{LG}$.

Initially, we set $b_v[u] = 0$, for all vertices $u \in V(G)$, and set $root_v = v$. (The root of T_v is the only vertex in T_v that is not marked in b_v.) Furthermore, we need to be able to test the ancestor-descendant relation in T. There are several simple $O(1)$-time tests of this relation [34]. The most convenient one for us is to number the vertices of T from 1 to n in preorder and compute the number of descendants of each vertex v. We denote these numbers by $pre(v)$ and $size(v)$, respectively. Then v is a descendant of u if and only if $pre(u) \leq pre(v) < pre(u) + size(u)$.

Suppose now that an edge $e = (u, v)$ is added into G. Then, since T is a spanning tree of G_s, e is a new non-tree edge in $\rho_{\mathcal{N}}(v)$. So, we need to find the tree edges $f \in T(e)$ such that $\widehat{LG}$ does not contain the edge (v, f). Equivalently, we need to find the vertices in $T(e)$ that are not already marked in b_v. Let x be the lowest common ancestor of u and v in T. (Note that we only use x for reference and do not need to find it explicitly.) To find the relevant unmarked vertices, we traverse the part of the cycle $T(e)$ from u to x as follows. Let $y = u$ be the current vertex. While y is not an ancestor of v, we check if $b_v[y] = 1$. If not, then we set $b_v[y] = 1$, add the edge from v to $(t(y), y)$ in $\widehat{LG}$, and set $y = t(y)$. Otherwise, we let $y = root_v$. This procedure stops as soon as y is an ancestor of v. At this point, if $b_v[y] = 0$, then we set $root_v = y$. We repeat the same procedure for $y = v$, where we stop when y becomes an ancestor of u.

▶ **Lemma 12.** *The above procedure correctly updates $\widehat{LG}$ in $O(n^2)$ total time for all edge insertions.*

Proof. It is easy to verify that the procedure maintains invariant (I) correctly, because of Lemma 11. Also, the algorithm inserts an edge (v, e) into $\widehat{LG}$ if and only if e is covered by v, which is in accordance to the labeling function $\widehat{\mathcal{L}}_c$. Now we bound the total running time for the construction of $\widehat{LG}$ after all insertions. The total running time is dominated by the time we need to locate tree edges that are just covered by each insertion. Let $e = (u, v)$ be a newly added edge to G. The above procedure visits at most two vertices that are already marked in b_v. For all other visited vertices u, we have $b_v[u] = 0$ before the visit and $b_v[u] = 1$ after the visit, excluding $root_v$, which also is visited at most twice per added edge. Hence, the total time throughout the whole sequence of insertions is bounded by $O(n^2)$. ◀

4.3 Incremental computation of the 2-edge-connected components

Let G be a strongly connected graph that undergoes edge insertions. We chose an arbitrary start vertex s, and compute a spanning trees T of G and a spanning T_R of G^R, rooted at s. We maintain two instances of the modified labeling graph of Section 4.1, $\widehat{LG}$ that represents the minimum 1-in sets of G, and $\widehat{LG}_R$ that represents the minimum 1-in sets of G^R, using the two fixed spanning trees T and T_R. When an edge (u, v) is inserted into G, we execute the update operation of Section 4.2 for $\widehat{LG}$ and for $\widehat{LG}_R$. Note that for $\widehat{LG}_R$, we search for tree edges of T_R that are covered by u.

We maintain the SCCs of $\widehat{LG}$ and $\widehat{LG}_R$ incrementally, by running on each labeling graph the incremental SCCs algorithm of Bender et al. [3] for dense graphs. For a digraph with n vertices, the algorithm of [3] runs in $O(n^2 \log n)$ total time. The modified labeling graphs $\widehat{LG}$ and $\widehat{LG}_R$ have $O(n)$ vertices and $O(n^2)$ edges, since during their construction we never add duplicate edges. By Lemma 12, we can construct them incrementally in $O(n^2)$ time, and the total time for maintaining their SCCs is also $O(n^2 \log n)$.

We turn now to the query operations *query*(v, w) and *report*$()$. Each SCC of $\widehat{LG}$ (and similarly of $\widehat{LG}_R$) is represented by a canonical vertex, and the partition of the vertices into SCCs is maintained through a disjoint set union data (DSU) structure [36, 35]. The DSU data structure supports the operation *unite*(p, q), which, given canonical vertices p and q, merges the SCCs containing p and q into one new SCC and makes p the canonical vertex of the new SCC. It also supports the query *find*(v), which returns the canonical vertex of the SCC containing v. Since we aim at constant time queries, we use such a data structure that can support each *find* operation in worst-case $O(1)$ time and any sequence of *unite* operations in total time $O(n \log n)$ [36]. This way, we can identify the canonical vertex of the auxiliary component containing a query vertex in constant time. Hence, by Corollary 10, we can answer *query*(v, w) in G also in constant time, by testing if u and v are strongly connected in both $\widehat{LG}$ and $\widehat{LG}_R$.

To answer a *report*$()$ query, we create, for each vertex $v \in V(G)$, a label $\mathit{label}(v) = \langle c_v, c_v^R, v \rangle$, where c_v and c_v^R are the canonical vertices in the SCCs of $\widehat{LG}$ and $\widehat{LG}_R$, respectively, that contain v. As above, each of these canonical vertices is available in $O(1)$ time. We form a list L consisting of $\mathit{label}(v)$ for all $v \in V(G)$, and sort them lexicographically in $O(n)$ time using bucket sorting. Then, in the sorted list L, the vertices of the same 2-edge-connected component appear consecutively, since they have the same canonical vertices in their labels. Therefore, we can report the 2-edge-connected components of G in $O(n)$ time.

4.4 Extension to general digraphs

Now we extend our incremental algorithm to general (not strongly connected) digraphs. We note that Proposition 6 requires us to use labeling graphs that correspond to strongly connected flow graphs. To that end, we construct a two-level data structure, that uses various instances of the incremental SCCs algorithm of Bender et al. [3], as mentioned in Section 4.3.

Let G be the input digraph subject to edge insertions. The top level of our data structure, that we refer to as ISC(G), maintains the strongly connected components of G with the use of the incremental SCCs algorithm of [3]. More precisely, ISC(G) maintains the SCCs of G, represented with a DSU data structure, and the condensation of G, denoted by $\bar{G}$, which is the directed acyclic graph that results from G after we contract each SCC into its canonical vertex. We note that [3] also maintains a topological ordering of $\bar{G}$, and when a new SCC is formed, it removes loops and duplicate edges. The bottom level of our data structure maintains the information about the 1-in sets and 1-out sets within each SCC C of G, in a structure I2EC(C). Specifically, for each strongly connected component C of G, with a designated start vertex s, we store a spanning tree T of $G[C]$ and a spanning tree T_R of $G^R[C]$, rooted at s. Also, we maintain the data structures of Sections 4.2 and 4.3.

Now we describe how to handle an edge insertion. Suppose a new edge (x, y) is inserted into G. If x and y are located the same SCC C of G, then we execute the insertion procedure for I2EC(C), described in Sections 4.2 and 4.3. Otherwise, we execute the insertion in the ISC(G) data structure. Note that this operation inserts the edge $(\mathit{find}(x), \mathit{find}(y))$ into the condensation $\bar{G}$ of G. If this insertion does not create a cycle in $\bar{G}$ then we are done. Otherwise, ISC(G) finds a new SCC of $\bar{G}$, corresponding to a new SCC of G, that is contracted into some canonical vertex. As a result, we need to update the bottom-level structure for the involved components of G.

Let C be the new SCC of G. The data structure ISC(G) identifies all components $C_1, C_2, \ldots, C_k$ of G that are merged into C after the insertion of (x, y), along with the edges $E(C_i, C_j)$ that connect distinct components. (Each such edge (u, v) satisfies $u \in C_i$, $v \in C_j$, where C_i precedes C_j in a topological ordering of the components.) Without loss of generality, assume that C_1 is the largest of these components. We choose the canonical (start) vertex s of C to be the start vertex of C_1. We refer to this component C_1 as the *principal component of* C. Also, we refer to the vertices of C_1 as the *principal vertices of* C. The remaining vertices in $C_2, \ldots, C_k$ are the *secondary vertices of* C.

Now we describe how to construct the data structure I2EC(C) for the new component C. We describe only the construction of the structures for $G[C]$. The structures for the reverse graph $G^R[C]$ are updated similarly. First, we extend the spanning tree T_1 of $G[C_1]$, which is rooted at s, to a spanning tree T of $G[C]$ rooted at s, so that $T_1 \subseteq T$. To achieve this, it suffices to traverse the edges $E(C_i, C_j)$ and the edges of the two spanning trees that we maintain for each component $C_2, \ldots, C_k$. This is enough to construct T, since the two spanning trees of each C_i form a sparse strongly connected subgraph of $G[C_i]$. (Note that we cannot afford to traverse all edges of $G[C_i]$.) Once T is constructed, we traverse it to recompute $\mathit{pre}(v)$ and $\mathit{size}(v)$ for all $v \in T$. This enables testing the ancestor-descendant relation in T in $O(1)$ time.

Next, we need to update the structures that keep track of the covered edges of T for each vertex $v \in C$. To initialize these structures for the new component C, we maintain the structures b_v and root_v, for all principal vertices v as they are. Then, we insert the nontree edges $e = (u, v)$ such that u is a secondary vertex in C. For each secondary vertex u, we compute b_u and root_u from scratch. Hence, in effect we construct I2EC(C) by inserting the secondary vertices and their adjacent edges in the data structure I2EC(C_1) of the principal component.

▶ **Lemma 13.** *The above procedure correctly updates* I2EC(C)*, for all SCCs C of G in $O(n^2 \log n)$ total time over all edge insertions.*

Proof. The correctness of the algorithm follows from Lemma 12, and the fact that the top structure ISC(G) maintains the SCCs of G. Next, we bound the total running time for any sequence of edge insertions.

First, we bound the total running time required to update the spanning tree T of $G[C]$, for each SCC formed in G. Let C be a new SCC of G that is formed by merging the components $C_1, \ldots, C_k$, where C_1 is the principal component. Let n_i be the number of vertices in each component C_i, and let $m_{ij} = |E(C_i, C_j)|$ be the number of edges connecting C_i and C_j. Then, the construction of T takes time proportional to $\sum_{i=1}^{k} n_i + \sum_{1 \le i,j \le k} m_{ij} = n_C + \sum_{1 \le i,j \le k} m_{ij}$, where $n_C = \sum_{i=1}^{k} n_i$ is the number of vertices in the new component C. Note that the second term, i.e., the sum $\sum_{1 \le i,j \le k} m_{ij}$ is charged only once during the construction of all spanning trees. Hence, the overall construction time for all spanning trees is $O(n^2 + m) = O(n^2)$.

Next, we consider the time required to maintain the data structures for the tree edges covered by each vertex $v \in V(G)$. Note that for as long as v is a principal vertex in a component C, the total time spent to process the edges in $\rho_{\mathcal{N}}(v)$ and update b_v and root_v is $O(n_C)$. Next, we consider the contribution of each vertex as a secondary vertex. Every time we merge a sequence of secondary components $C_2, \ldots, C_k$ with a principal component C_1, we charge $O(n_C)$ time to each secondary vertex v, since we recompute b_v and $root_v$ from scratch. Let $C_v^1, \ldots, C_v^\xi$ be the sequence of SCCs that contain v as a secondary vertex. Let n_v^i be the number of vertices in C_v^i just before it gets merged into a larger component. Then, $n_v^i \leq n_v^{i+1}/2$, for $1 \leq i < \xi$, and $n_v^\xi \leq n$. The time required to maintain the data structures for the tree edges covered by v in C_v^i is proportional to n_v^i. Hence, the total time for the whole sequence of components $C_v^1, \ldots, C_v^\xi$ is bounded by $\sum_{i=1}^{\xi} n_v^i \leq \sum_{i=0}^{\log n} n/2^i < 2n$. This gives an $O(n^2)$ total bound for all vertices.

Finally, we consider the total time required to maintain the incremental SCCs data structures. Such a structure for the subgraph induced by a component C with n_C vertices requires $O(n_C^2 \log n_C)$ total time. We distribute this cost to the n_C vertices of the component, so each vertex is charged a cost of $O(n_C \log n_C)$. Hence, for as long as a vertex v is a principal vertex in its component C, it is charged a cost of $O(n_C \log n_C)$, where n_C is the number of vertices in C just before it is merged as a secondary component. If this never happens, then n_C is the final number of vertices in C. It remains to bound the contribution of v as a secondary vertex, which we can do as above. Let $C_v^1, \ldots, C_v^\xi$ be the sequence of SCCs that contain v as a secondary vertex, where each C_v^i has n_v^i vertices just before it gets merged into a larger component. As before, $n_v^i \leq n_v^{i+1}/2$, for $1 \leq i < \xi$, and $n_v^\xi \leq n$. Then, the cost charged to v for the whole sequence of components $C_v^1, \ldots, C_v^\xi$ is bounded by $\sum_{i=1}^{\xi} n_v^i \log n_v^i \leq \log n \sum_{i=0}^{\log n} n/2^i < 2n \log n$. This gives an $O(n^2 \log n)$ total bound for all vertices. ◀

5 Decremental algorithm

In this section, we present our decremental algorithm for maintaining the 2-edge-connected components of a digraph G. We can assume that G is strongly connected, as otherwise we can process each SCC separately. Let s be an arbitrarily chosen start vertex of G. Let T be a fixed spanning tree of flow graph G_s, and let T_R be a fixed spanning tree of flow graph G_s^R. (Both T and T_R are rooted at s.) The algorithm operates on the assumption that the edges of T and T_R are never deleted throughout the deletion sequence.

We describe how to efficiently update the edges of the modified labeling graph $\widehat{LG}$ of Section 4.1, as we delete edges in G. To achieve this, we need to maintain some additional information about the tree edges that are covered by each vertex $v \in V(G)$.

For a tree edge $e \in T$, we define $cover_v(e)$ to be the number of edges $f \in \rho_{\mathcal{N}}(v)$ that cover e, i.e., such that $e \in T(f)$. Then $(v, e) \in E(\widehat{LG})$ if and only if $cover_v(e) > 0$. Our approach is to maintain the $cover_v(e)$ values using a dynamic tree data structure [32]. This way, we can update $\widehat{LG}$ efficiently, and maintain its SCCs decrementally using the algorithm of [39].

A *dynamic tree* $\mathcal{T}$ is a data structure that efficiently maintains a collection of rooted trees, whose edges have real-valued costs, under dynamic operations such as linking and cutting edges, while supporting cost updates and queries on paths and subtrees. For our purposes, we will assume that each such data structure maintains a single tree that corresponds to the spanning tree T of G_s with root s. Recall that for any vertex $v \neq s$, $t(v)$ denotes the parent of v in T. Here, we also let $T[v, w]$ denote the tree path between two vertices v and w, ignoring edge directions.

We will use the following dynamic tree operations, which are supported in $O(\log n)$ time.

- *cost*(v): If $v \neq s$, then return the cost of the edge $(t(v), v)$.
- *update*(v, w, x): Add x to the cost of all edges on the tree path $T[v, w]$.
- *mincost*(v, w): Return a vertex $u \in T[v, w]$ such that the edge $(t(u), u)$ has minimum cost among the edges on the tree path $T[v, w]$.

We note that the original description of Sleator and Tarjan [32] has the operation $update(v, x)$, which adds the value x to the cost of the edges on the path from v to the root of $\mathcal{T}$, and the operation $mincost(v)$, which returns an edge of minimum cost on the path from v to the root of $\mathcal{T}$. We can implement our versions of *update* and *mincost*, by using the operation $evert(v)$ of [32], which makes v the root of $\mathcal{T}$, as follows. To implement $update(v, w, x)$, we do $evert(w)$, $update(v, x)$, and $evert(s)$. Similarly, to implement $mincost(v, w)$ we do $evert(w)$, $mincost(v)$, and $evert(s)$.

To initialize $\widehat{LG}$, we compute a spanning tree T of G rooted at s, and insert the edges of $\widehat{LG}$ as in Section 4.2. We also compute the $pre(v)$ and $size(v)$ values for all $v \in T$, so that we can test the ancestor-descendant relation in T in $O(1)$ time. Then, for each vertex $v \in V(G)$, we maintain a dynamic tree data structure $\mathcal{T}_v$, which implements the operations *update* and *mincost* on T, where each tree edge $e \in T$ has cost $cover_v(e)$. Initially, all edge costs in $\mathcal{T}_v$ are zero. Then, we process each edge $(u, v) \in \rho_{\mathcal{N}}(v)$, and execute $update(v, u, +1)$.

During the execution of the deletion sequence, we do the following. Let $e = (u, v)$ be the next edge of G to be deleted. By our assumption, $e \in \rho_{\mathcal{N}}(v)$, and we need to find the tree edges $f \in T(e)$ that are covered only by e. For each such tree edge f, we delete the corresponding edge (v, f) from $\widehat{LG}$.

To find these edges $f \in T(e)$, first we execute $update(v, u, -1)$. Then, we recursively search a tree path $T[x, y]$ for uncovered edges (that is, tree edges $f \in T[x, y]$ with $cover_v(f) = 0$), where initially $x = u$ and $y = v$, as follows. We compute $w = mincost(x, y)$, and check if $cost(w) \neq 0$. If this is the case, then all the edges on $T[x, y]$ are still covered by v, and we are done. Otherwise, we delete the edge $(v, (t(w), w))$ from $\widehat{LG}$, and repeat recursively this step for the two paths of $T[x, y] \setminus (t(w), w)$.

In more detail, if w is an ancestor of x, then we recursively search for uncovered edges on $T[x, w]$ and on $T[y, t(w)]$. Otherwise, if w is an ancestor of y, then we recursively search for uncovered edges on $T[y, w]$ and on $T[x, t(w)]$. Hence, we obtain the following bound.

▶ **Lemma 14.** *We can maintain the modified label graph $\widehat{LG}$ under a sequence of edge deletions of G in $O(n^2 \log n)$ total time.*

Proof. Consider the deletion of an edge $e \in \rho_{\mathcal{N}}(v)$. The above procedure finds a tree edge $f \in T(e)$ that becomes uncovered by v using a constant number of dynamic tree operations. For each edge f with $cover_v(f) = 0$, we delete the corresponding edge (v, f) in $\widehat{LG}$. Since $\widehat{LG}$ has $O(n^2)$ edges, and each dynamic tree operation takes $O(\log n)$ time, the bound follows. ◀

From Lemma 14 and the fact that we use the decremental algorithm of [39] to maintain the SCCs of $\widehat{LG}$, we obtain Theorem 2.

References

1. Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443, 2014. doi:10.1109/FOCS.2014.53.
2. Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999. doi:10.1137/S0097539797317263.
3. Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2), December 2015. doi:10.1145/2756553.
4. Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, pages 365–376, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3313276.3316335.

5 Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert E. Tarjan, and Jeffery R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008. doi:10.1137/070693217.

6 Li Chen, Rasmus Kyng, Yang P. Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, s-t shortest path, and minimum-cost flow. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, pages 1165–1173, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649745.

7 Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107–114, 2000. doi:10.1016/S0020-0190(00)00051-X.

8 Harold N. Gabow. The minset-poset approach to representations of graph connectivity. *ACM Transactions on Algorithms*, 12(2):24:1–24:73, February 2016. doi:10.1145/2764909.

9 Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985. doi:10.1016/0022-0000(85)90014-5.

10 Zvi Galil and Giuseppe F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, March 1991. doi:10.1145/122413.122416.

11 Loukas Georgiadis, Thomas Dueholm Hansen, Giuseppe F. Italiano, Sebastian Krinninger, and Nikos Parotsidis. Decremental data structures for connectivity and dominators in directed graphs. In *ICALP*, pages 42:1–42:15, 2017. doi:10.4230/LIPIcs.ICALP.2017.42.

12 Loukas Georgiadis, Giuseppe F. Italiano, and Evangelos Kosinas. Computing the 4-edge-connected components of a graph in linear time. In *Proc. 29th European Symposium on Algorithms*, pages 47:1–47:17, 2021. doi:10.4230/LIPIcs.ESA.2021.47.

13 Loukas Georgiadis, Giuseppe F. Italiano, and Evangelos Kosinas. Computing the 3-edge-connected components of directed graphs in linear time. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024*, pages 62–85, 2024. doi:10.1109/FOCS61266.2024.00015.

14 Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-edge connectivity in directed graphs. *ACM Transactions on Algorithms*, 13(1):9:1–9:24, 2016. doi:10.1145/2968448.

15 Loukas Georgiadis, Giuseppe F. Italiano, and Nikos Parotsidis. Incremental 2-edge-connectivity in directed graphs. In *ICALP*, pages 49:1–49:15, 2016. doi:10.4230/LIPIcs.ICALP.2016.49.

16 Loukas Georgiadis, Giuseppe F. Italiano, and Nikos Parotsidis. Strong connectivity in directed graphs under failures, with applications. *SIAM J. Comput.*, 49(5):865–926, 2020. doi:10.1137/19M1258530.

17 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

18 Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020. doi:10.1137/18M1180335.

19 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001. doi:10.1145/502090.502095.

20 John E. Hopcroft and Robert E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973. doi:10.1137/0202012.

21 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 510–520, USA, 2017. Society for Industrial and Applied Mathematics. doi:10.5555/3039686.3039718.

22 Wenyu Jin and Xiaorui Sun. Fully dynamic s-t edge connectivity in subpolynomial time (extended abstract). In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 861–872, 2022. doi:10.1109/FOCS52979.2021.00088.

23 Ken-Ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *Journal of the ACM*, 66(1), December 2018. doi:10.1145/3274663.

24 Tuukka Korhonen. Linear-time algorithms for k-edge-connected components, k-lean tree decompositions, and more. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, STOC '25, pages 111–119, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3717823.3718123.

25 Evangelos Kosinas. Computing the 5-edge-connected components in linear time. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1887–2119, 2024. doi:10.1137/1.9781611977912.76.

26 Jakub Łącki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Transactions on Algorithms*, 9(3):27, 2013. doi:10.1145/2483699.2483707.

27 Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

28 Wojciech Nadara, Mateusz Radecki, Marcin Smulewicz, and Marek Sokołowski. Determining 4-edge-connected components in linear time. In *Proc. 29th European Symposium on Algorithms*, pages 71:1–71:15, 2021. doi:10.4230/LIPIcs.ESA.2021.71.

29 Hiroshi Nagamochi and Toshihide Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan J. Indust. Appl. Math*, 9(163), 1992. doi:10.1007/BF03167564.

30 Hiroshi Nagamochi and Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity.* Cambridge University Press, 2008. 1st edition. doi:10.1017/CBO9780511721649.

31 Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time . In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 950–961, Los Alamitos, CA, USA, October 2017. IEEE Computer Society. doi:10.1109/FOCS.2017.92.

32 Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

33 Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

34 Robert E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974. doi:10.1137/0203006.

35 Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

36 Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984. doi:10.1145/62.2160.

37 Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, February 2007. doi:10.1007/s00493-007-0045-2.

38 Yung H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130–146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP). doi:10.1016/j.jda.2008.04.003.

39 Jan van den Brand, Li Chen, Rasmus Kyng, Yang P. Liu, Simon Meierhans, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-Linear Time Algorithms for Decremental Graphs: Min-Cost Flow and More via Duality . In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2010–2032, Los Alamitos, CA, USA, October 2024. IEEE Computer Society. doi:10.1109/FOCS61266.2024.00120.