HAL
open science

Incremental Strong Connectivity and 2-Connectivity in Directed Graphs

Loukas Georgiadis, Giuseppe F Italiano, Nikos Parotsidis

To cite this version:

Loukas Georgiadis, Giuseppe F Italiano, Nikos Parotsidis. Incremental Strong Connectivity and 2-Connectivity in Directed Graphs. Latin American Symposium on Theoretical Informatics, 2018, Buenos Aires, Argentina. hal-01925953

HAL Id: hal-01925953

https://inria.hal.science/hal-01925953

Submitted on 18 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

arXiv:1802.10189v1 [cs.DS] 27 Feb 2018

Incremental Strong Connectivity and 2-Connectivity in Directed Graphs

Loukas Georgiadis[1], Giuseppe F. Italiano[2], and Nikos Parotsidis[2]

[1]University of Ioannina, Greece.
[2]University of Rome Tor Vergata, Italy.

Abstract

In this paper, we present new incremental algorithms for maintaining data structures that represent all connectivity cuts of size one in directed graphs (digraphs), and the strongly connected components that result by the removal of each of those cuts. We give a conditional lower bound that provides evidence that our algorithms may be tight up to a sub-polynomial factors. As an additional result, with our approach we can also maintain dynamically the 2-vertex-connected components of a digraph during any sequence of edge insertions in a total of $O(mn)$ time. This matches the bounds for the incremental maintenance of the 2-edge-connected components of a digraph.

1 Introduction

A dynamic graph algorithm aims at updating efficiently the solution of a problem after each update, faster than recomputing it from scratch. A dynamic graph problem is said to be *fully dynamic* if the update operations include both insertions and deletions of edges, and it is said to be *incremental* (resp., *decremental*) if only insertions (resp., deletions) are allowed. In this paper, we present new incremental algorithms for some basic connectivity problems on directed graphs (digraphs), which were recently considered in the literature [20]. Before defining the problems and stating our bounds, we need some definitions.

Let $G = (V, E)$ be a digraph. G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* (in short *SCCs*) of G are its maximal strongly connected subgraphs. Two vertices $u, v \in V$ are *strongly connected* if they belong to the same SCC of G. An edge (resp., a vertex) of G is a *strong bridge* (resp., a *strong articulation point*) if its removal increases the number of SCCs in the remaining graph. See Figure 1. Given two vertices u and v, we say that an edge (resp., a vertex) of G is a *separating edge* (resp., a *separating vertex*) for u and v if its removal leaves u and v in different SCCs. Let G be strongly connected: G is 2-*edge-connected* (resp., 2-*vertex-connected*) if it has no strong bridges (resp., no strong articulation points). Two vertices $u, v \in V$ are said to be 2-*edge-connected* (resp., 2-*vertex-connected*), denoted by $u \leftrightarrow_{2\mathrm{e}} v$ (resp., $u \leftrightarrow_{2\mathrm{v}} v$), if there are two edge-disjoint (resp., internally vertex-disjoint) directed paths from u to v and

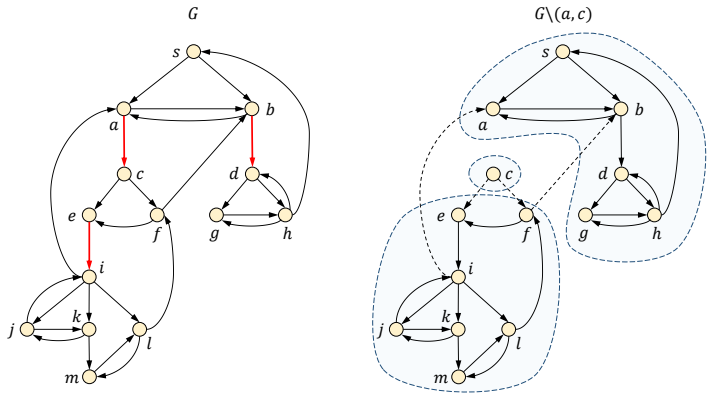

Figure 1: A strongly connected digraph G with strong bridges shown in red (better viewed in color), and the SCCs of $G \setminus e$ after the deletion of the strong bridge $e = (a, c)$.

two edge-disjoint (resp., internally vertex-disjoint) directed paths from v to u. (Note that a path from u to v and a path from v to u need not be edge-disjoint or internally vertex-disjoint). A 2-*edge-connected component* (resp., a 2-*vertex-connected component*) of a digraph $G = (V, E)$ is defined as a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2e} v$ (resp., $u \leftrightarrow_{2v} v$) for all $u, v \in B$. Given a digraph G, we denote by $G \setminus e$ (resp., $G \setminus v$) be the digraph obtained after deleting edge e (resp., vertex v) from G.

Let $G = (V, E)$ be a strongly connected graph. In very recent work [20], we presented an $O(n)$-space data structure that, after a linear-time preprocessing, is able to answer in asymptotically optimal (worst-case) time all the following queries on a static digraph:

- Report in $O(1)$ time the total number of SCCs in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v) in G.
- Report in $O(1)$ time the size of the largest and of the smallest SCCs in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v) in G.
- Report in $O(n)$ time all the SCCs of $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v).
- Test in $O(1)$ time whether two query vertices u and v are strongly connected in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v).
- For any two query vertices u and v that are strongly connected in G, report all edges e (resp., vertices v) such that u and v are not strongly connected in $G \setminus e$ (resp., $G \setminus v$) in time $O(k + 1)$, where k is the number of separating edges (resp., separating vertices).

As pointed out in [20, 37], this data structure is motivated by applications in many areas, including computational biology [21, 34] social network analysis [30, 44], network resilience [40] and network immunization [4, 7, 32].

A dynamic version of the aforementioned data structure can be used to monitor the critical components (i.e., edges and vertices) whose removal disrupts the

underlying graphs, in graphs that change over time. An ideal scenario is to design efficient algorithms in the fully dynamic setting. However, we show that no data structure that can answer any of the queries that we consider in sublinear time in the number of edges, can be maintained faster than recomputing the data structure from scratch unless a widely believed conjecture is proved wrong. There are real-word dynamic networks where edge deletion occur rarely, in which case the incremental setting finds applications. Such networks include, for instance, communication networks, road networks, the power grid.

Our Results. We show a conditional lower bound for the fully dynamic version of this problem. More specifically, let $G = (V, E)$ be a digraph with n vertices that undergoes m edge updates from an initially empty graph. We prove that any fully dynamic algorithm that can answer any of the queries considered here requires either $\Omega(m^{1-o(1)})$ amortized update time, or $\Omega(m^{1-o(1)})$ query time, unless the Strong Exponential Time Hypothesis [27, 28] is false.

Motivated by this hardness result, we focus on the incremental version of this problem. We present an incremental version of the data structure introduced in [20], which can be maintained throughout a sequence of edge insertions. In particular, we show how to maintain a digraph G undergoing edge insertions in a total of $O(mn)$ time, where n is the number of vertices and m the number of edges after all insertions, so that all the queries we consider can be answered in asymptotically optimal (worst-case) time after each insertion. As an additional result, with our approach we can also maintain the 2-vertex-connected components of a digraph during any sequence of edge insertions in a total of $O(mn)$ time. After every insertion we can test whether two query vertices are 2-vertex-connected and, whenever the answer is negative, produce a separating vertex (or an edge) for the two query vertices. This matches the bounds for the incremental maintenance of the 2-edge-connected components of a digraph [19].

Before our work, no algorithm for all those problems was faster than recomputing the solution from scratch after each edge insertion, which yields a total of $O(m^2)$. Our algorithms improve substantially over those bounds. In addition, we show a conditional lower bound for the total update time of an incremental data structure that can answer queries of the form "are u and v strongly connected in $G \setminus e$", where $u, v \in V$, $e \in E$. In particular, we prove that the existence of a data structure that supports the aforementioned queries with total update time $O((mn)^{1-\epsilon})$ (for some constant $\epsilon > 0$). Therefore, a polynomial improvement of our bound leads to a breakthrough.

Related Work. Many efficient algorithms for several dynamic graph problems have been proposed in the literature, including dynamic connectivity [24, 26, 35, 36, 42], minimum spanning trees [12, 14, 25, 26, 35], edge/vertex connectivity [12, 26] on undirected graphs, and transitive closure [10, 23, 31, 39] and shortest paths [2, 9, 31, 43] on digraphs. Dynamic problems on digraphs are known to be harder than on undirected graphs and most of the dynamic algorithms on undirected graphs have polylog update bounds, while dynamic algorithms on digraphs have higher polynomial update bounds. The hardness of dynamic algorithms on digraphs has been recently supported also by conditional lower bounds [1].

In [15], the decremental version of the data structure considered in this paper is presented. The total time and space required to maintain decrementally the data structure is $O(mn \log n)$ and $O(n^2 \log n)$, respectively: here m is the num-

ber of edges in the initial graph. We remark that our incremental algorithms are substantially different from decremental algorithms of [15], and indeed the techniques that we use here are substantially different from [15]. More specifically, the main approach of [15] is to maintain the SCCs in $G \setminus v$ for each $v \in V$, by carefully combining n appropriate instances of the decremental SCCs algorithm from [33]. This allows to maintain decrementally the dominator tree in $O(mn \log n)$ total time and $O(n^2 \log n)$ space. On the contrary, in the incremental setting it is already known how to maintain dominator trees, and the main challenge is to maintain efficiently information about nesting loops throughout edge insertions. This allows us to achieve better bounds than in the decremental setting [15]: namely, $O(mn)$ total time and $O(m + n)$ space.

In [19] we presented an incremental algorithm that maintains the 2-edge-connected components of a directed graph with n vertices through any sequence of edge insertions in a total of $O(mn)$ time, where m is the number of edges after all insertions. After each insertion, we can test in constant time if two query vertices v and w are 2-edge-connected, and if not we can produce in constant time a "witness" of this property, by exhibiting an edge that is contained in all paths from v to w or in all paths from w to v.

Our Technical Contributions. Our fist contribution is to dynamize the recent data structure in [20], which hinges on two main building blocks: dominator trees and loop nesting trees (which are reviewed in Section 2). While it is known how to maintain efficiently dominator trees in the incremental setting [18], the incremental maintenance of loop nesting trees is a challenging task. Indeed, loop nesting trees are heavily based on depth-first search, and maintaining efficiently a dfs tree of a digraph under edge insertions has been an elusive goal: no efficient solutions are known up to date, and incremental algorithms are available only in the restricted case of DAGs [13]. To overcome these inherent difficulties, we manage to define a new notion of strongly connected subgraphs of a digraph, which is still relevant for our problem and is independent of depth first search. This new notion is based on some specific nesting loops, which define a laminar family. One of the technical contributions of this paper is to show how to maintain efficiently this family of nesting loops during edge insertions. We believe that this result might be of independent interest, and perhaps it might shed further light to the incremental dfs problem on general digraphs.

Our second contribution, the incremental maintenance of the 2-vertex-connected components of a digraph, completes the picture on incremental 2-connectivity on digraphs by complementing the recent 2-edge connectivity results of [19]. We remark that 2-vertex connectivity in digraphs is much more difficult than 2-edge connectivity, since it is plagued with several degenerate special cases, which are not only more tedious but also more cumbersome to deal with. For instance, 2-edge-connected components partition the vertices of a digraph, while 2-vertex-connected components do not. Furthermore, two vertices v and w are 2-edge-connected if and only if the removal of any edge leaves v and w in the same SCC. Unfortunately, this property no longer holds for 2-vertex connectivity, as for instance two mutually adjacent vertices are always left in the same strongly connected component by the removal of any other vertex, but they are not necessarily 2-vertex-connected.

2 Dominator trees, loop nesting trees and auxiliary components

In this section we review the two main ingredients used by the recent framework in [20]: dominator trees and loop nesting trees. As already mentioned in the introduction, one of the main technical difficulties behind our approach is that the incremental maintenance of loop nesting trees seems an elusive goal. We then review the notion of auxiliary components, which is used in Section 3 to overcome this difficulty. We remark that both dominator trees and auxiliary components can be maintained efficiently during edge insertions [18, 19].

Throughout, we assume that the reader is familiar with standard graph terminology, as contained for instance in [8]. Given a rooted tree, we denote by $T(v)$ the set of descendants of v in T. Given a digraph $G = (V, E)$, and a set of vertices $S \subseteq V$, we denote by $G[S]$ the subgraph induced by S. Moreover, we use $V(S)$ and $E(S)$ to refer to the vertices of S and to the edges adjacent to S, respectively. The *reverse digraph* of G, denoted by $G^R = (V, E^R)$, is obtained by reversing the direction of all edges. A *flow graph* F is a directed graph (digraph) with a distinguished start vertex $s \in V(F)$, where all vertices in $V(F)$ are reachable from s in F. We denote by G_s the subgraphs of G induced by the vertices that are reachable from s; that is, G_s is a flow graph with start vertex s. Respectively, we denote by G_s^R the subgraphs of G^R induced by the vertices that are reachable from s. If 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.

Dominator trees. A vertex v is a *dominator* of a vertex w (v *dominates* w) if every path from s to w contains v. The dominator relation in G can be represented by a tree rooted at s, the *dominator tree* D, such that v dominates w if and only if v is an ancestor of w in D. See Figure 2. We denote by $dom(w)$ the set of vertices that dominate w. Also, we let $d(w)$ denote the parent of a vertex w in D. Similarly, we can define the dominator relation in the flow graph G_s^R, and let D^R denote the dominator tree of G_s^R, and $d^R(v)$ the parent of v in D^R. The dominator tree of a flow graph can be computed in linear time, see, e.g., [3, 6]. An edge (u, v) is a *bridge* of a flow graph G_s if all paths from s to v include (u, v).[1] Let s be an arbitrary start vertex of G. As shown in [29], an edge $e = (u, v)$ is strong bridge of G if and only if it is either a bridge of G_s or a bridge of G_s^R. As a consequence, all the strong bridges of G can be obtained from the bridges of the flow graphs G_s and G_s^R, and thus there can be at most $2(n-1)$ strong bridges overall. After deleting from the dominator trees D and D^R respectively the bridges of G_s and G_s^R, we obtain the *bridge decomposition* of D and D^R into forests $\mathcal{D}$ and $\mathcal{D}^R$. Throughout the paper, we denote by D_u (resp., D_u^R) the tree in $\mathcal{D}$ (resp., $\mathcal{D}^R$) containing vertex u, and by r_u (resp., r_u^R) the root of D_u (resp., D_u^R). The following lemma from [17] holds for a flow graph G_s of a strongly connected digraph G (and hence also for the flow graph G_s^R of G^R).

Lemma 2.1. ([17]) *Let G be a strongly connected digraph and let (u, v) be a strong bridge of G. Also, let D be the dominator tree of the flow graph G_s, for*

[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.

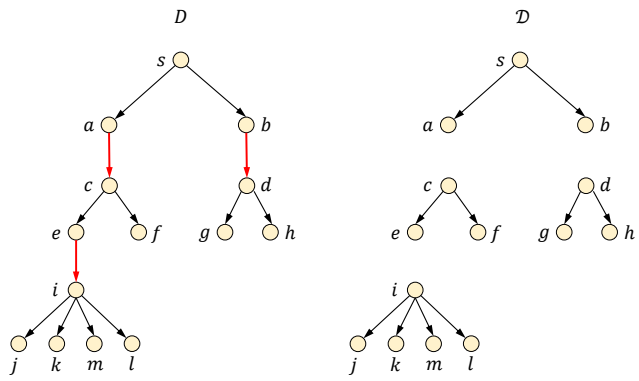

Figure 2: The dominator tree D (on the left) of the digraph of Figure 1 with start vertex s, and its bridge decomposition $\mathcal{D}$ (on the right).

an arbitrary start vertex s. Suppose $u = d(v)$. Let w be any vertex that is not a descendant of v in D. Then there is path from w to v in G that does not contain any proper descendant of v in D. Moreover, all simple paths in G from w to any descendant of v in D must contain the edge $(d(v), v)$.

Loop nesting forests. Let G be a digraph, and G_s the flow graph with an arbitrary start vertex s. A *loop nesting forest* represents a hierarchy of strongly connected subgraphs of G_s [41], defined with respect to a dfs tree T of G_s, rooted at s, as follows. For any vertex u, $loop(u)$ is the set of all descendants x of u in T such that there is a path from x to u in G containing only descendants of u in T. Any two vertices in $loop(u)$ reach each other. Therefore, $loop(u)$ induces a strongly connected subgraph of G; it is the unique maximal set of descendants of u in T that does so. The $loop(u)$ sets form a laminar family of subsets of V: for any two vertices u and v, $loop(u)$ and $loop(v)$ are either disjoint or nested. The *loop nesting forest* H of G_s, with respect to T, is the forest in which the parent of any vertex v, denoted by $h(v)$, is the nearest proper ancestor u of v in T such that $v \in loop(u)$ if there is such a vertex u, and null otherwise. Then $loop(u)$ is the set of all descendants of vertex u in H, which we also denote as $H(u)$ (the subtree of H rooted at vertex u). A loop nesting forest can be computed in linear time [6, 41]. When G is strongly connected, each vertex is contained in a loop, and H is a tree, rooted at s. Therefore, we refer to H as the *loop nesting tree* of G_s (see Figure 3).

Auxiliary components. Let G_s be a flow graph and D and $\mathcal{D}$ be the dominator tree and the bridge decomposition of G_s, respectively. Let $e = (u, v)$ be a bridge of the flow graph G_s. We say that an SCC C in $G[D(v)]$ is an *e-dominated component* of G. We also say that $C \subseteq V$ is a *bridge-dominated component* if it is an e-dominated component for some bridge e: bridge-dominated components form a laminar family [19]. An *auxiliary component* of G_s is a maximal subset of vertices $C \cap D_v$ such that C is a subset of a $(d(r_v), r_v)$-dominated component. Each auxiliary component C is represented by an arbitrarily chosen vertex $u \in C$, which we call the *canonical vertex* of C. For each vertex $v \in C$, we refer to the canonical vertex of C by c_v. That is, if u is the canonical vertex of an auxiliary component then $c_u = u$. Following the bridge decomposition $\mathcal{D}$ of the dominator tree D of G_s, the auxiliary components are defined with respect to

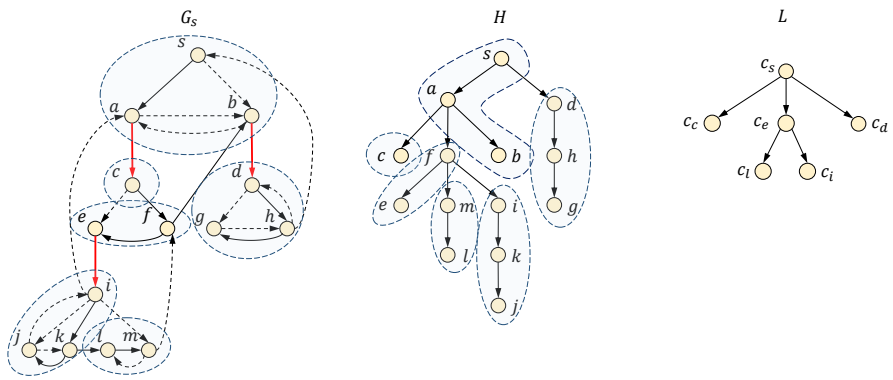

Figure 3: The flow graph G_s of the graph of Figure 1 with solid dfs edges (left); the loop nesting tree H of G_s generated by the dfs traversal on the left (middle); the hyperloop nesting tree L of G_s (right). The bridges of G_s are shown red. The grouped vertices in both G_s and L represent the auxiliary components of G_s.

the start vertex s.

3 Hyperloop nesting forest

In this section, we introduce the new notion of *hyperloop nesting forest*, which, differently from loop nesting forest, can be maintained efficiently during edge insertions, as we show in Section 5. Given a canonical vertex $v \neq c_s$, we define the *hyperloop of* v, and denote it by $hloop(v)$, as the set of canonical vertices that are in the same $(d(r_v), r_v)$-dominated component as v. As a special case, all canonical vertices that are strongly connected to s are in the hyperloop $hloop(c_s)$. It can be shown that hyperloops form a laminar family of subsets of V, with respect to the start vertex s: for any two canonical vertices u and v, $hloop(u)$ and $hloop(v)$ are either disjoint or nested (i.e., one contains the other). This property allows us to define the *hyperloop nesting forest* L of G_s as follows. The parent $\ell(v)$ of a canonical vertex v in L is the (unique) canonical vertex u, $u \notin D(r_v)$, with the largest depth in D, such that $v \in hloop(u)$. If there is no vertex $u \notin D(r_v)$, such that $v \in hloop(u)$, then $\ell(v) = \emptyset$; notice that in this case v is not strongly connected to s as well. See Figure 3. Then, $hloop(u)$ is the set of all descendants of a canonical vertex u in L, which we also denote as $L(u)$ (the subtree of L rooted at vertex u). Similarly to the loop nesting forest, the hyperloop nesting forest of a strongly connected digraph is a tree.

We begin the study of the hyperloop nesting forest by showing that it is unique, and thus, depends solely on the structure of the graph. (We consider a fixed choice of the canonical vertices of the auxiliary components.)

Lemma 3.1. *The hyperloop nesting forest of a flowgraph G_s is unique.*

Proof. Let v be a canonical vertex of G_s. By the definition of the hyperloop nesting forest L, the parent $\ell(v)$ of v in L is the canonical vertex $u \notin D(r_v)$ with maximum depth in D, such that u and v are in the same $(d(r_u), r_u)$-dominated component. Then, the fact that both u and v are descendants of r_u implies that u is unique and so the lemma follows. □

□

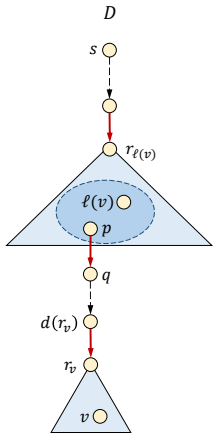

Figure 4: A representation of the relation between a canonical vertex v and its parent in L with respect to the bridge decomposition of D.

Given a vertex u in a flow graph G_s, we define its level, denoted by $level(u)$, to be the number of bridges (v, w) of G_s such that w is an ancestor of u in D. In other words, the level of u equals the number of strong bridges that appear in all paths from s to u in G_s. As a result, all vertices in the same tree of the bridge decomposition have the same level. In the next lemma we show that each canonical vertex has at most one ancestor in L at each level.

Lemma 3.2. *Let G_s a flow graph and let u be a canonical vertex of G_s. All ancestors of u in the hyperloop nesting forest have unique level.*

Proof. Let w and v be two distinct ancestors of u in L such that $level(w) = level(v)$. By the definition of the hyperloop nesting forest both r_w and r_v are ancestors of u in D. Then, $r_w = r_v$. Assume by contradiction that u is strongly connected with both w and v in $G[D(r_w)]$. Then w and v are strongly connected in $G[D(r_w)]$. By the definition of the auxiliary components, w and v are in the same auxiliary component, and thus $c_w = c_v$. A contradiction to the fact that both w and v are canonical vertices. The lemma follows. □

□

The following lemma characterizes the relationship between the loop nesting forest H and the hyperloop nesting forest L of a flow graph G_s. More specifically, it shows that L can be obtained from H by contracting all the vertices of each auxiliary component into their canonical vertex. This yields immediately a linear-time algorithm to compute the hyperloop nesting forest of a flow graph G_s: we first compute a loop nesting forest H of G_s [6, 41] and then contract each vertex v to c_v in H. In the following we denote by h_v the unique ancestor of v in H, such that $h_v \in D_v$ and $h(h_v) \notin D_v$. If $v \in D_s$, it follows that $h_v = s$. We can compute h_v, for all $v \in V$ in $O(n)$ time [20].

Lemma 3.3. *For every vertex v, the following hold:*

- *The canonical vertices of v and h_v are the same.*

- *The canonical vertex of the parent of c_v in L is the canonical vertex of the parent of h_v in H, including the case where $h(h_v) = \emptyset$.*

Proof. Let T be the dfs traversal that generated H. In the case where $v \in D_s$, it trivially follows that $c_{h_v} = c_v = c_s$. Now assume that $v \notin D_s$. Note that r_v is an ancestor of h_v in T since all paths from s to h_v go through r_v by Lemma 2.1. Therefore, also all the descendants of h_v in H are descendant of r_v in T. The fact that h_v is an ancestor of v in H it means that v is a descendant of h_v in T and v has a path P to h_v using only descendants of h_v in T. The path P cannot contain vertices that are not in $D(r_v)$ since otherwise by Lemma 2.1 they contain r_v, which contradicts the fact that all vertices on P contain descendants of h_v in T (recall that r_v is an ancestor of h_v in T). It follows that h_v has a path to v in $G[D(r_v)]$ and v has a path to h_v in $G[D(r_v)]$. Thus, $c_v = c_{h_v}$ by the definition of the auxiliary components. Exactly the same argument can be applied on the reverse graph to show that $c^R_{h^R_v} = c^R_v$.

Consider now $h(h_v)$. In the case where $v \in D_s$, it trivially follows that $h(c_{h_v}) = h(c_v) = \emptyset$. Therefore in the following we assume that $v \notin D_s$. First, we deal with the case where $c_{h(h_v)} = \emptyset$. Since $v \notin D_s$, the only case that this can happen is when $h(h_v) = \emptyset$. That means, there is no ancestor w of h_v in T, such that h_v has a path to w using only vertices in $T(w)$. We show that this means, there is also no vertex $z \notin D(r_v)$, such that v and z are strongly connected in G. Assume, for the sake of contradiction that there is such vertex z, and let C be the SCC containing both h_v, and therefore also v, and z. Moreover, let vertex $z' \in C, z' \neq h_v$ be the vertex that is visited first by the dfs that generated T. Then all vertices in C are descendants of z' in T since they are all reachable from z' and they were not visited by the dfs before z'. Hence, h_v is a descendant of z' in T. A contradiction to the fact that $h(h_v) = \emptyset$. Therefore, there is no vertex $z \notin D(r_v)$, such that v and z are strongly connected in $G \supset G[D(r_z)]$. Thus, $\ell(c_v) = \emptyset$. Now we consider the case where $c_{h_v} = c_v \neq \emptyset$. The fact that $r_{h(h_v)}$ is an ancestor of $h(h_v)$ in D implies that $r_{h(h_v)}$ is an ancestor of $h(h_v)$ also in T. By the definition of H it follows that h_v is a descendant of $h(h_v)$ in T, and h_v has a path P to $h(h_v)$ using only descendants of $h(h_v)$ in T. The path P cannot contain vertices that are not in $D(r_{h(h_v)})$ since otherwise, by Lemma 2.1, they contain $r_{h(h_v)}$, which contradicts the fact that all vertices on P contain descendants of $h(h_v)$ in T (recall that $r_{h(h_v)}$ is an ancestor of $h(h_v)$ in T). Therefore $h(h_v)$ and h_v are strongly connected in $G[r_{h(h_v)}]$, and by definition so do $c_{h(h_v)}$ and c_{h_v}. Hence, $c_{h(h_v)}$ is an ancestor of c_{h_v} in L. Now we show that there is no other canonical vertex $w \neq h(h_v)$ such that $level(c_w) > level(h(h_v))$ and c_w is an ancestor of c_{h_v} in L. This implies that $\ell(c_v) = c_{h(h_v)}$. Assume, for the sake of contradiction, that there is such a vertex w. Then h_v and c_w are in the same SCC C in $G[D(r_{c_w})]$. Let $z \in C$ be the first among the vertices in C visited by the dfs that generated T. Then all vertices in C are descendants of z in T since they are all reachable from z and they were not visited by the dfs before z. Thus, h_v is a descendant of z in H. Now we show that this implies that z is also a descendant of $h(h_v)$ in H. By the definition of the hyperloop nesting forest r_z and $r_{h(h_v)}$ are ancestors of h_v in D. Furthermore, since $level(z) > level(h(h_v))$ it holds that $r_{h(h_v)}$ is an ancestor of r_z in D. By the fact that $h(h_v)$ and z are ancestors of h_v in T it holds that $h(h_v)$ and z have ancestor-descendant relation, and hence, z is a descendant of $h(h_v)$ in T (as r_z and $r_{h(h_v)}$ are ancestors of h_v in D and $level(z) > level(h(h_v))$).

Since h_v has a path to $h(h_v)$ using only descendants of $h(h_v)$ in T, it follows that z also has a path to $h(h_v)$ using only descendants of $h(h_v)$ in T. Thus, z is a descendant of $h(h_v)$ in H. In summary, z if an ancestor of h_v and a descendant of $h(h_v)$ in H; a contradiction to the definition of $h(h_v) \neq z$. This concludes the lemma. □

□

4 Updating the dominator tree after an edge insertion

In this section, we briefly review the algorithm from [18] that updates the dominator tree of a flow graph G_s after an edge insertion. Let G_s be a flow graph with start vertex s. Let (x, y) be the edge to be inserted. Let D be the dominator tree of G_s before the insertion; we let D' be the the dominator tree of G'_s. In general, for any function f on V, we let f' be the function after the update. We say that vertex v is *D-affected* by the update if $d(v)$ (its parent in D) changes, i.e., $d'(v) \neq d(v)$. We let $nca_D(x, y)$ denote the nearest common ancestor of x and y in the dominator tree D.

Lemma 4.1. ([38]) *If v is D-affected, then it becomes a child of $nca_D(x, y)$ in D', i.e., $d'(v) = nca_D(x, y)$.*

We say that a vertex is *D-scanned* if it is a descendant of a D-affected vertex after an edge insertion. Note that every D-affected vertex is also D-scanned since each vertex is a descendant of itself in D. There are two key ideas behind the incremental dominators algorithm. First, the algorithm updates D' in time proportional to number of the edges incident to D-scanned vertices. Second, after an edge insertion, all D-scanned vertices decrease their depth in D' by at least one. Since throughout a sequence of edge insertions the depth of a vertex can only decrease, each vertex can be D-scanned at most $n - 1$ times, and thus the algorithm examines at most $(n - 1)$ times the adjacency list of each vertex. This implies that the algorithm has $O(mn)$ total update time, where m is the number of edges in the graph after all insertions.

We now prove the following lemma, which shows that after the insertion of an edge (x, y), the ancestors and descendants of the vertices $v \notin D'(nca_D(x, y))$ do not change. We use this later later on in our incremental algorithm for maintaining the hyperloop nesting forest of a flow graph.

Lemma 4.2. *Let u be a vertex that is not a descendant of $nca_D(x, y)$ in D'. Then $dom(u) = dom'(u)$ and $D'(u) = D(u)$. Moreover, if $(d(u), u)$ was a bridge in G_s, then it remains a bridge in G'_s.*

Proof. The claim that $D'(u) = D(u)$ follows immediately from the fact that all D-scanned vertices (which were the only vertices v for which $dom(v) \neq dom'(v)$) were in $D(nca_D(x, y))$ and therefore remain in $D'(nca_D(x, y)) = D(nca_D(x, y))$. This also implies that $dom(u) = dom'(u)$, as for each $w \in dom(u)$ it holds that $w \notin D(nca_D(x, y))$. Now let $(d(u), u)$ be a bridge in G_s, and assume by contradiction that $(d(u), u)$ is no longer a bridge in G'_s. This implies there is a path from s to u avoiding $(d(u), u)$ in G'_s. Since $(d(u), u)$ was the only incoming edge to $D(u) = D'(u)$, the path in G' must contain and edge (w, z) such that

Algorithm 1: Initialize(G, s)

Set s to be the designated start vertex of G.
Compute the dominator tree D and the set of bridges Br of the corresponding flow graph G_s.
Compute the bridge decomposition $\mathcal{D}$ of D, and the auxiliary components of G_s.
Compute the loop nesting forest H of the G_s.
Construct the hyperloop nesting forest L from H by contacting each vertex w into c_w.

$w \notin D(u), z \in D(u)$. This contradicts the fact that the only new edge is (x, y) and that either $x, y \in D(u)$ or $x, y \notin D(u)$, as otherwise $D(nca_D(x, y))$ is an ancestor of u in D (which we assume is not). □

□

5 Updating the hyperloop nesting forest after an edge insertion

Let G be a directed graph and let G_s be the flow graph of G with an arbitrary start vertex s. In this section we show how to maintain the hyperloop nesting forest L of G_s under a sequence of edge insertions. We assume that D and L are rooted at s and c_s, respectively. For simplicity, we also assume that all vertices of G are reachable from s, so $m \geq n - 1$. If this is not true, then we can simply recompute D and L from scratch, in linear time, every time a vertex becomes reachable from s after an edge insertion. Since there can be at most $n - 1$ such events, the total running time for these recomputations is $O(mn)$.

Throughout the sequence of edge insertions, we maintain as additional data structures only the dominator tree D (with the incremental dominators algorithm in [18]), the bridge decomposition and the auxiliary components of G_s (with the algorithm in [19]).

Initialization and restarts. To initialize the algorithm, we compute the dominator tree D, bridge decomposition and auxiliary components, which can be done in linear time [3, 19]. We also compute the hyperloop nesting forest L of G_s in linear time, as suggested by Lemma 3.3. The pseudocode for the initialization is given in Algorithm 1.

After the first initialization, in some special cases we initialize our algorithm again, in order to simplify the analysis. We call this a *restart*. We restart our algorithm whenever a bridge $e = (u, v)$ of G_s is canceled after the insertion of a new edge (x, y) but we still have $d'(v) = u$, i.e., (u, v) is no longer a strong bridge in G but the parent of u in the dominator tree D does not change. In this case, we say that the bridge $e = (u, v)$ is *locally canceled*. This is a difficult case to analyze: the incremental dominators algorithm does not spend any time, since there are no D-affected vertices, while the bridge decomposition and the auxiliary components of G_s might change. Fortunately, there are at most $O(n)$ locally canceled bridges throughout a sequence of edge insertions [19]. Hence, we restart our algorithm at most $O(n)$ times. Consequently, the total time spent

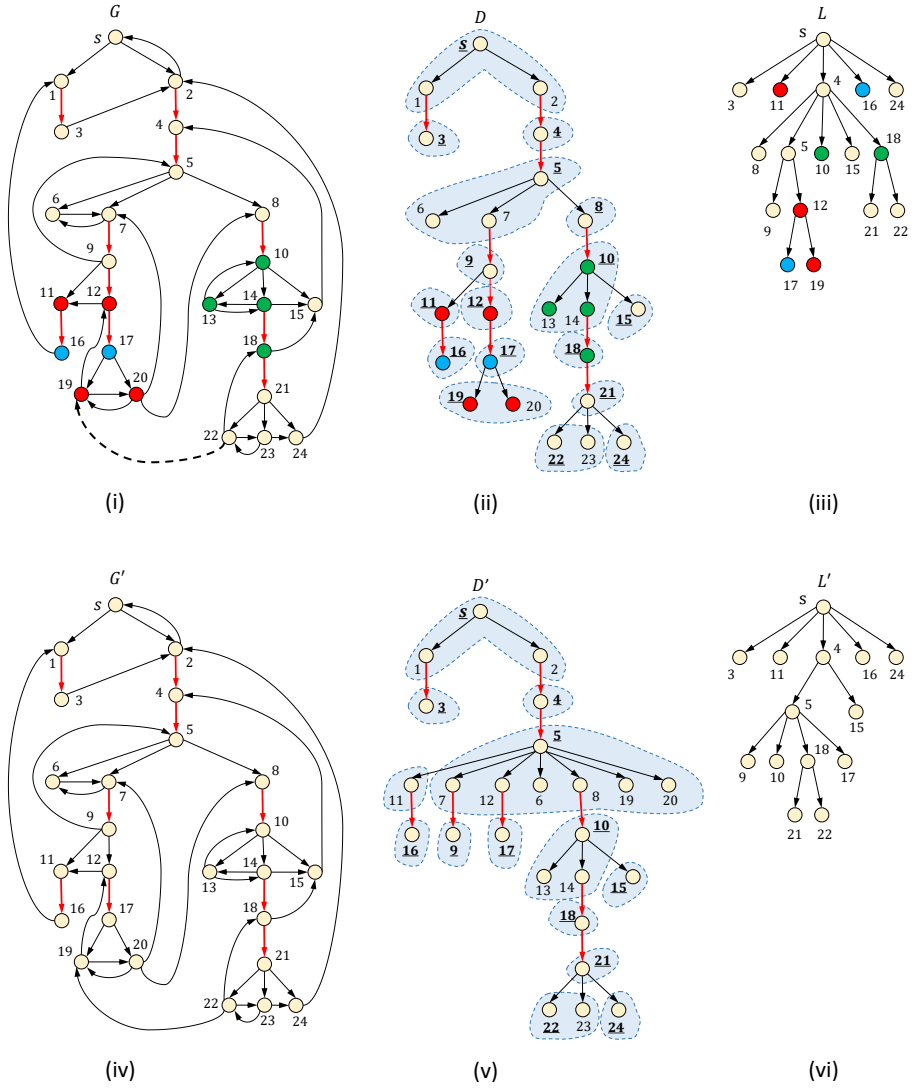

Figure 5: A detailed example demonstrating the different types of vertices that we consider after an edge insertion. In (i), a digraph G before the insertion of $(22, 19)$ (dashed edge). In (ii), the dominator tree of G where vertices in the same auxiliary components are grouped together. In (iii), the hyperloop nesting tree of G. In (i)-(iii) we color the D-affected with red the D-scanned but not D-affected vertices with blue, and the L-affected but not D-scanned vertices with green. Finally, in (iv), (v), and (vi) we represent G', D' where vertices in the same auxiliary components are grouped together, and L'.

in restarts is $O(mn)$.

High-level overview of the update. Let (x, y) be the new edge to be inserted. Similarly to Section 4, for any function f, we use the notation f' to denote the same function after the insertion of (x, y), e.g., we denote by $\ell'(v)$, the parent of a canonical vertex v in the hyperloop nesting forest, after the

Algorithm 2: SCInsertEdge(G, e)

```
Let s be the designated start vertex of G, and let e = (x, y).
Update the dominator trees D. Let S be the set of D-scanned vertices.
Update the bridge decomposition 𝒟, and the auxiliary components of G_s
if a bridge is locally canceled in G_s then
    Execute Initialize(G, s).
else
    Execute Update-D-scanned(𝒟, L, x, y, S) and
      Update-L-affected(𝒟, L, x, y, S).
end
```

insertion of (x, y), and by L' the resulting hyperloop nesting forest. Once again, we denote by *D-scanned* the vertices that decrease their depth in the dominator tree D after an edge insertion. Moreover, we denote by *D-affected* the vertices that change their parent in D and by *L-affected* the vertices for which it holds $\ell'(v') \neq c'_{\ell(v)}$, i.e., when the parent of v' in L' is not in the same auxiliary component as $\ell(v)$ (the parent of v in L). After the insertion of a new edge (x, y), if not involved in a restart, our algorithm performs the following updates, as shown in the pseudocode of Algorithm 2:

1) Compute the new dominator tree D', the corresponding bridge decomposition $\mathcal{D}'$, and the new auxiliary components.

2) Compute $\ell'(v)$ for the D-scanned canonical vertices $v \in D'_y$.

3) Compute $\ell'(v)$ for the D-scanned canonical vertices $v \notin D'_y$.

4) Compute $\ell'(v)$ for the L-affected canonical vertices v that are not D-scanned.

As already mentioned, the dominator tree D, the bridge decomposition and the auxiliary components of G_s can be maintained during edge insertions within our claimed bounds [18, 19]. To complete the algorithm, it remains to show how to update efficiently the parent in the hyperloop nesting forest of the D-scanned and the L-affected vertices. This is non-trivial, and the low-level technical details of the method are spelled out in Sections 5.1 and 5.2, respectively. Before giving the details of our algorithm, we show that we only need to consider the vertices $D'(r'_y)$.

Lemma 5.1. *No canonical vertex $v \notin D'(r'_y)$ is L-affected.*

Proof. We have that $nca_D(x, y) \in D'(r'_y)$. Assume $\ell'(v') = w$. The fact that $v \notin D'(r'_y)$ implies that $r'_w \notin D'(r'_y)$. By Lemma 4.2, $r'_w = r_w$ and $D'(r'_w) = D(r_w)$. Therefore, $G[D(r_w)] = G'[D'(r'_w)] \setminus (x, y)$. As w and v are not strongly connected in $G[D(r_w)] = G[D'(r'_w)]$ but they are strongly connected in $G'[D'(r'_w)] = G[D(r_w)] \setminus (x, y)$, if follows that either all paths from w to v or from v to w in $G'[D'(r'_w)]$ contain (x, y). Thus, all paths from w or from v to y in $G'[D'(r'_w)]$ contain (x, y). As $w, v \notin D'(nca_D(x, y))$, by Lemma 2.1, all paths from w or v to y in $G'[D'(r'_w)]$ contain $nca_D(x, y)$. Note that there is path from $nca_D(x, y)$ to y in $G'[D'(nca_D(x, y))]$ avoiding (x, y) (as $y \in D(nca_D(x, y)) = D'(nca_D(x, y))$,

and $G[D(nca_D(x,y))] = G'[D'(nca_D(x,y))] \setminus (x,y))$. A contradiction to the fact that all paths from w or from v to y in $G'[D'(r'_w)]$ contain (x,y). □

□

5.1 Updating the D-scanned vertices

Let S be the set of D-scanned vertices containing also the D-affected vertices. After the insertion of the edge (x,y), all the D-affected vertices become children of $nca_D(x,y)$ in D', by Lemma 4.1. In this section we deal with the update of the parent in the hyperloop nesting forest $\ell'(v')$, for all canonical vertices $v \in S$. From now on, in order to simplify the notation, we assume without loss of generality that $v = c_v$ for any vertex of interest v; we also denote c'_v by v'.

After the insertion of the edge (x,y) only a subset of the ancestors in L' of an L-affected canonical vertex v changes. In particular, Lemma 5.2 shows that the ancestors w of v in L such that $w \notin D'(r'_y)$ remain ancestors of v' in L'. However, the insertion of (x,y) might create a new path from v to a canonical vertex z such that $v \in D'(r'_z)$, containing only vertices in $D'(r'_z)$. In such a case, z' becomes an ancestor of v' in L'.

Lemma 5.2. *Let $v \in D(r'_y)$ be a canonical vertex in G. For each ancestor z of v in L, such that $z \notin D'(r'_y)$, the canonical vertex z' remains ancestor of v' in L'. Moreover, $D'(r'_{z'}) = D(r_z)$.*

Proof. By Lemma 4.2 and the fact that $r'_{z'}$ is an ancestor of $D'(r'_y)$, it follows that $D'(r'_{z'}) = D(r_z)$ and $(d(r'_{z'}), r'_{z'})$ remains a bridge in G'_s. Then, the fact that z' is an ancestor of v' in L' follows from the fact that v and z are strongly connected in $G'[D'(r'_{z'})]$ and $D'(r'_{z'}) = D(r_z)$. □

□

The following lemma identifies the new parent in L' of y'.

Lemma 5.3. *It holds that $\ell'(y') = w'$, where w is the nearest ancestor of y in L such that $w \notin D'(r'_y)$.*

Proof. By Lemma 5.2, for all ancestors z of y in L, such that $z \notin D'(r'_y)$, vertex z' remains ancestor of y' in L'. We show that all ancestors of y' in L' were ancestors of y in L before the edge insertion. Assume by contradiction that there exists a canonical vertex z' that is an ancestor of y' in L', but z is not an ancestor of y in L. By the definition of hyperloop nesting forest, r'_z must be a proper ancestor of r'_y. Lemma 4.2 and the fact that (x,y) is not locally canceled imply that $dom'(r'_y) = dom(r_y)$, r'_z is an ancestor of r_y in D, and $D'(r'_z) = D(r_z)$. Following our assumption we have that z and y are strongly connected in $G'[D'(r'_z)]$ but not in $G[D(r_z)] = G'[D(r_z)] \setminus (x,y)$. Therefore, there is no path from x to y in $G[D(r_z)]$. By the fact that $y \in D(nca_D(x,y))$, there exists a path from $nca_D(x,y)$ to y in $G[D(nca_D(x,y))]$ avoiding x. As $nca_D(x,y) \in D(r_z)$, there is a path from z to y through $nca_D(x,y)$ in $G[D(r_z)]$ avoiding x; a contradiction. The lemma follows. □

□

We now compute $\ell'(v')$ for each canonical vertex $v' \in S \cap D'_y$, that is, the D-affected vertices that are in the same tree of the canonical decomposition of D' with y. Note that all the new paths that are introduced by the insertion of

the edge (x, y) must contain y. We use this observation to compute $\ell'(v')$, for $v' \in S \cap D'_y$, based on $\ell'(y')$ as shown in the following lemma.

Lemma 5.4. *Let $v \in D'_y$ be D-scanned. If v' and y' are in different auxiliary components then $\ell'(v') = w'$, where w is the nearest ancestor of v in L such that $w \notin D'(r'_y)$.*

Proof. By Lemma 5.2 for each ancestor z of v in L, such that $z \notin D'(r'_y)$, vertex z' remains ancestor of v' in L'. Assume by contradiction that $\ell'(v') = z'$ with $level'(z') > level'(w')$, that is, r'_w is a proper ancestor of r'_z in D'. Since $v \in D'_y$, it follows that $r'_z \notin D'(r'_y)$, and therefore $D'(r'_z) = D(r_z)$ by Lemma 4.2. By our assumption, v and z are strongly connected in $G'[D(r_z)]$. Lemma 2.1 implies that all paths from z to v in $G'[D(r_z)]$ contain $d'(r'_v) = d'(r'_y)$. Let $P_{zv} = P_{zd'(r'_y)} \cdot P_{d'(r'_y)v}$ be such a path, where $P_{zd'(r'_y)}$ is the subpath from z to $d'(r'_y)$ and $P_{d'(r'_y)v}$ the subpath from $d'(r'_y)$ to v. Since, $P_{zd'(r'_y)}$ does not contain (x, y) and we also have that $v \in D(r'_y)$ it follows that z has a path to v in $G[D(r_z)]$. As v and z are not strongly connected in $G[D(r_z)]$, all path from v to z in $G'[D(r_z)]$ contain (x, y). Moreover, due to the facts that $v' \neq y'$, any path from v to z contains a vertex out of $D'(r'_y)$, and therefore, also r'_y by Lemma 2.1. By the last two arguments, all paths from v to z contain both first r'_y and then (x, y). That implies that all paths from r'_y to y in G'_s contain the edge (x, y), and as all paths from s to y in G'_s contain r'_y (by Lemma 2.1) it follows that all paths from s to y contain (x, y). This is sufficient for (x, y) to be a bridge in G'_s, which is a contradiction as y is reachable from s in G_s. Thus, our assumption about z led us to a contradiction. The lemma follows. □

□

With the help of Lemma 5.4, we can iterate over the vertices $v \in S \cap D'_y$ setting $\ell'(v')$ appropriately. Recall that S (the set of D-scanned vertices) is provided to us by the incremental dominators algorithm. Next, we deal with the canonical vertices $v \in S \setminus D'_y$. We begin with the computation of $\ell'(v')$, for the canonical vertices v' in S for which $level'(\ell'(v')) > level'(r'_y)$, that is, their new parent in L is in $D'(r'_y) \setminus D'_y$. Let $G_{scanned}$ be the graph induced by the D-scanned vertices, and let $H_{scanned}$ be the loop nesting forest rooted at y of $G_{scanned}$. (Note that y reaches all vertices in $G_{scanned}$.) By contracting every vertex v into c'_v in $H_{scanned}$, we obtain a forest $\tilde{H}$. Let $\tilde{h}(v)$ be the parent in $\tilde{H}$ of a canonical vertex $v \in S \setminus D'_y$. As stated in Lemma 5.5, the parent in $\tilde{H}$ of each canonical vertex v' such that $\tilde{h}(v) \in S \setminus D'_y$ is the parent of v' in L'.

Lemma 5.5. *Let $\tilde{H}$ be the loop nesting forest rooted at y of $G_{scanned}$ after contracting each vertex v into c'_v. For every canonical vertex v such that $\tilde{h}(v) \in S \setminus D'_y$, it holds that $\ell'(v') = \tilde{h}(v)$.*

Proof. Let $v \in S \setminus D'_y$. As all paths from y to v contain r'_v and v is D-scanned, all vertices in $D'(r'_v)$ are D-scanned. Hence, $G_{scanned}[D'(r'_v)] = G'[D'(r'_v)]$. Therefore, if $\ell'(v') \in S \setminus D'_y$ then v and $\ell'(v')$ are strongly connected in $G'[D'(r'_{\ell'(v)})] = G_{scanned}[D'(r'_{\ell'(v)})]$, which implies that $\ell'(v')$ is an ancestor of v' in $\tilde{H}$. (For a visualization of the relations of $v, y, \ell'(v)$, and s see Figure 6 (i).) To complete the proof we show that if a canonical vertex z is the parent of v' in $\tilde{H}$, where $v' \in S \setminus D'_y$, then z is the parent of v' in L'. Assume by contradiction that $\ell'(v') \neq \tilde{h}(v')$. Then, there is a vertex w such that $level'(\ell'(v')) < level'(w) <$

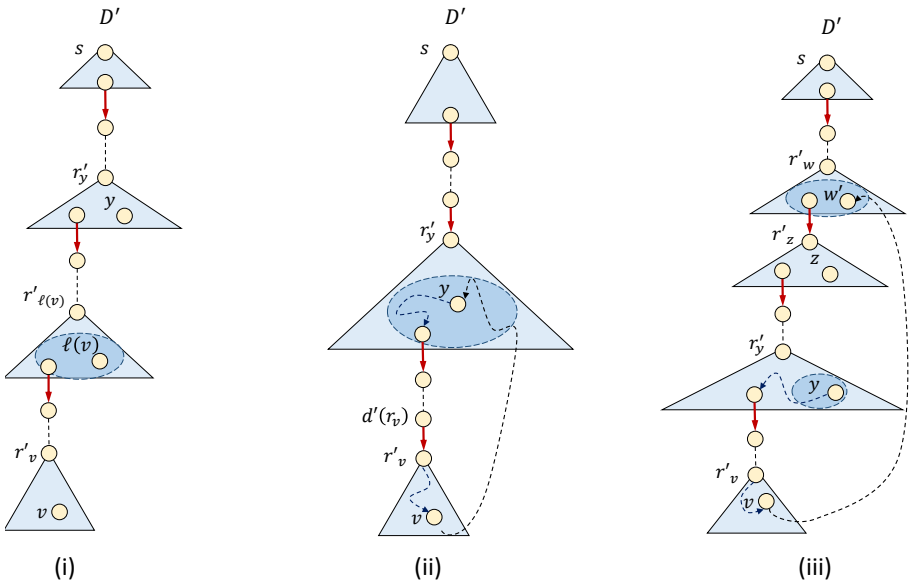

Figure 6: Instances of the updated dominator tree D' after the insertion of (x, y). In (i) v is a D-scanned vertex where $v, \ell'(v) \in S \setminus D'_y$. For all such vertices, we apply Lemma 5.5 to compute $\ell'(v)$. In (ii), v has a path to y in $G'[D'(r'_y)]$ and is D-affected, which is sufficient for y' to be ancestor of v' in L'. In (iii) vertex w is an ancestor of v in L, as w and v are strongly connected in $G[D(r_w)]$, and therefore w' remains an ancestor of v' in L'. The instances visualize the relations of the vertices of interest in the proofs of Lemmas 5.5 and 5.6.

$level'(v)$, and w and v are strongly connected in $G'[D'(r'_w)] = G_{scanned}[D'(r'_w)]$. Note that w contradicts the definition of $\ell'(v')$, and thus the lemma follows. □

□

Finally, for the vertices $v \in S \setminus D'_y$ for which $\ell'(v') \notin S \setminus D'_y$, we compute $\ell'(v')$ according to the following lemma.

Lemma 5.6. *Let $v \notin D'_y$ be a D-scanned vertex such that $level'(\ell'(v')) \leq level'(r'_y)$. If v has a path to y in $G'[D'(r'_y)]$, then $\ell'(v') = y'$ in L'. Otherwise, $\ell'(v') = w'$, where w is the nearest ancestor of v in L such that $level'(w) \leq level'(r'_y)$. If there is no such vertex $\ell'(v') = \emptyset$*

Proof. First, we assume v has a path to y in $G'[D'(r'_y)]$. Since v is D-scanned, it means that y has a path to v in $G'[D'(nca_{D'}(x, y))]$, and therefore, in $G'[D'(r'_y)]$. Hence, y and v are strongly connected in $G'[D'(r'_y)]$. By the definition of the hyperloop nesting forest, y' is an ancestor of v' in L', and since $level'(\ell'(v')) \leq level'(r'_y)$ it follows that $\ell'(v') = y'$. (For a visualization of the relations of v, y, s in D see Figure 6 (ii).) Now we assume v does not contain a path to y in $G'[D'(r'_y)]$. First assume that $w \in D'_y$. In this case v and w were strongly connected in $G[D(r_w)]$ before the insertion of (x, y). Since $D(r_w) \subseteq D'(r'_y)$, if follows that v and w are strongly connected in $G'[D'(r'_y)]$. By the definition of L', w' is an ancestor of v' in L', and by the assumption of the lemma that $level'(\ell'(v')) \leq level'(r'_y)$, and Lemma 3.2, it follows that $\ell'(v') = w'$.

Algorithm 3: Update-D-scanned$(\mathcal{D}, L, x, y, S)$

Set $\ell'(c'_y) = c'_w$, where w is the nearest ancestor of c_y in L such that $w \notin D'(r'_y)$.
Let S be the set of D-scanned vertices, and $G_{scanned} = G'[V(S)]$. Compute the loop nesting forest $H_{scanned}$ of $G_{scanned}$ with start vertex y. Contract every vertex $v \in V(G_{scanned})$ into c'_v in $H_{scanned}$, forming $\tilde{H}$.
foreach *canonical vertex* $v \in V(G_{scanned})$ **do**
  Let $v' = c'_v$ and $y' = c'_y$.
  if $v \in D'_y$ **then**
    if $v' \neq y'$ **then** $\ell'(v') = c'_w$, where w is the nearest ancestor of v in L such that $w \notin D'(r'_y)$.
  else
    if $\tilde{h}(v') \in S \setminus D'_y$ **then** $\ell'(v') = \tilde{h}(v')$
    else if *v has a path to y in $G'[D'(r'_y)]$* **then** $\ell'(v') = \ell'(y')$.
    else $\ell'(v') = c'_w$, where w is the nearest ancestor of v in L such that $level'(w) \leq level'(r'_y)$.
  end
end

Finally, for the rest of the proof we assume $w \notin D'(r'_y)$. By Lemma 5.2, w' remains an ancestor of v' in L'. (For a visualization of the relations of v, y, w', z, s in D see Figure 6 (iii).) Assume, by contradiction, that $\ell'(v') = z'$ and $level'(w') < level'(z') < level'(r'_y)$ and c_z is not an ancestor of v in L. By Lemma 2.1 all paths from z to v contain r'_y. As $v \in D(r'_y)$ there is a path from r'_y to v in $G[D(r'_y)]$. Since $D(r'_y) \subset D'(r'_z)$, there is a path from z to v in $G'[D'(r'_z)] \setminus (x, y)$ $(= G[D(r_z)]$ by Lemma 5.2). This fact, combined with our assumption that c_z is not an ancestor of v in L, implies that there is no path from v to z in $G[D(r_z)]$. Since v and z are strongly connected in $G'[D'(r'_z)]$ but not in $G[D(r_z)] = G'[D'(r'_z)] \setminus (x, y)$ all paths from v to z contain (x, y); let P_{vz} be such a path from v to z and let P_{vx} and P_{yz} be its subpaths from v to x and from y to z, respectively. Note that P_{vx} and P_{yz} exist also in $G[D(r_z)] = G'[D'(r'_z)] \setminus (x, y)$. Since we deal with the case where v does not have a path to y in $G'[D'(r'_y)]$, the path P_{vx} should contain a vertex $u \in D'(r'_z) \setminus D'(r'_y)$ (as otherwise all vertices are in $D'(r'_y)$ and the path $P_{vx} \cdot (x, y)$ is a path from v to y in $G'[D'(r'_y)]$). By Lemma 2.1, P_{vx} contains r'_y; let $P_{vr'_y}$ be the subpath of P_{vx} from v to r'_y. Since $y \in D(r'_y)$ there is a path $P_{r'_y y}$ from r'_y to y in $G[D(r'_y)] \subset G[D(r'_z)]$ avoiding (x, y). Observe that the paths $P_{vr'_y}$,$P_{r'_y y}$, and P_{yz} all exist in $G[D(r'_z)]$. Collectively, we have that $P_{vr'_y} \cdot P_{r'_y y} \cdot P_{yz}$ is a path from v to z in $G[D(r'_z)] = G[D(r_z)]$. As we argued z also has a path to v in $G[D(r_z)]$, and therefore z and v are strongly connected in $G[D(r_z)]$. Since r_z is an ancestor of r_v in D, by the definition of L, c_z is an ancestor of v' in L. This is a contradiction to our assumption that w is the nearest ancestor of v in L such that $w \notin D'(r'_y)$ and $level'(w) < level'(z) < level'(r'_y)$. The lemma follows. □

□

Lemmas 5.3, 5.4, 5.5, and 5.6 provide the tools to update the parent in L' of the D-scanned vertices S. The pseudocode of this update is detailed in Algorithm 3. The challenging part is to determine whether v has a path to y in $G'[D'(r'_y)]$ (Line 9 of Algorithm 3). We show how we can answer the queries of Line 9 of Algorithm 3, and thus how to update the parents in L' of all the canonical D-scanned vertices in time linear in the number of D-scanned vertices and their adjacent edges.

Lemma 5.7. *For all D-scanned vertices v, we can compute $\ell'(v')$ in time $O(|V(S)| + |E(S)|)$.*

Proof. The loop nesting forest of the graph induced by the D-scanned vertices can be computed in linear time to the size of S, i.e., $O(|V(S)| + |E(S)|)$. The nearest ancestor w of each D-scanned canonical vertex v in L, such that $level'(w') \leq level'(r'_y)$, can be computed in total $O(|V(S)|)$ as follows. We find all canonical vertices $v \in S$ such that $level'(\ell(v)) \leq level'(r'_y)$ and we assign to each descendant of v in L the vertex $\ell(v)$. All the necessary tests of Lemmas 5.4, 5.5, and 5.6 can be performed in constant time per vertex.

Now we show that we can determine in time $O(|V(S)| + |E(S)|)$ all vertices $v \in S \setminus D'_y$ that have a path to y in $G'[D'(r'_y)]$, as required by Lemma 5.6. We mark all vertices in S that have outgoing edges to vertices v such that $c'_v = c'_y$. Then, the vertices in S that reach y in $G'[D'(r'_y)]$ are the vertices that can reach a marked vertex in $G'[S]$. These vertices can be determined in time $O(|E(S)|)$, by executing backward traversals from the marked vertices without visiting the same vertices twice. Now we show that the above procedure is correct by showing that a vertex $v \in S \setminus D'_y$ has a path to y in $G'[D'(r'_y)]$ if and only if it has a path to a marked vertex in $G'[S]$. We start with the forward direction. If a vertex $v \in S \setminus D'_y$ has a path to y in $G'[D'(r'_y)]$ using only vertices in S, then clearly v reaches y in $G'[S]$. If on the other hand, there is a path from v that uses vertices outside S, then let w be the first vertex on that path such that $w \notin S$, and let z be its predecessor on the path. Then $c'_w = c'_y$ since: (i) by the fact that $w \notin S$, it holds that either $r'_w \notin D'(r'_y)$ or $r'_w = r'_y$, where only the second case is interesting since in the first it holds that $w \notin G'[D'(r'_y)]$, (ii) w reaches y in $G'[D'(r'_y)]$ (by our assumption that a path from v to y exists in $G'[D'(r'_y)]$ using w) and (iii) y reaches v since v is D-scanned (which means y has a path to v in $G'[D'(r'_y)]$). Therefore, v has a path to the marked vertex $z \in S$. We continue with the reverse direction of the claim, that is, if v does not have a path to y in $G'[D'(r'_y)]$ then there is not path to a marked vertex in $G'[S]$. This is true for the paths containing only vertices in S. Also there is no path from v to a marked vertex z in $G'[S]$, since otherwise, there is a path from v to a vertex w such that $c'_w = c'_y$ and therefore to y in $G'[D'(r'_y)]$ using the edge (z, w), while we assumed that no such path exists. Thus, we showed how to compute all the vertices that have a path to y in $G'[D'(r'_y)]$ in time $O(|E(S)|)$, which concludes the lemma. □

□

5.2 Updating the L-affected vertices that are not D-scanned

Now we consider updating the parent in the hyperloop nesting forest of the vertices that are L-affected but not D-affected. We start with the following lemma that is used throughout the section. The lemma suggests that we can

find all the L-affected vertices via a backward traversal from y vising all vertices in $G'[D'(r'_y)]$ that have a path to y. In the worst case we spend $O(m)$ time to execute the traversal, which our algorithm cannot afford. Later, we show how to speed up this process by exploiting some key properties of the L-affected vertices.

Lemma 5.8. *For every L-affected vertex v that is not D-scanned, every path from v to $\ell'(v')$ in $G'[D'(r'_{\ell'(v')})]$ contains (x, y). Moreover, v has a path to x in $G[D'(r'_y)] = G'[D'(r'_y)] \setminus (x, y)$.*

Proof. Assume by contradiction that there exists a L-affected vertex $v \notin S$ such that v has a path P to $\ell'(v')$ in $G'[D'(r'_{\ell'(v')})] \setminus (x, y)$. Since v is not D-scanned the ancestors of v in D and D' are the same and the edge $(d(r'_{\ell(v)}), r'_{\ell(v)})$ is a strong bridge in G'_s such that $v \in D'(r'_{\ell(v)})$. Therefore, $c'_{\ell(v)}$ is an ancestor of v' in L'. Notice that $level'(\ell'(v')) > level'(\ell(v))$ as v is L-affected and $c'_{\ell(v)}$ remains an ancestor of v' in L' by Lemma 5.2. If $\ell'(v') \notin D'(r'_y)$, by Lemma 5.2, $\ell'(v')$ is an ancestor of v in L, contradicting the fact that $level'(\ell'(v')) > level'(\ell(v))$. Now we assume $\ell'(v') \in D'(r'_y)$. All D-scanned vertices become children of $nca_D(x, y)$, and by Lemma 4.2 no vertex outside of $nca_{D'}(x, y)$ is D-scanned. Therefore, vertices can only move out of $D'(r'_{\ell'(v')})$ and hence $D'(r'_{\ell'(v')}) \subset D(r'_{\ell'(v')})$. Collectively we have that $\ell'(v')$ and v are strongly connected in $G'[D'(r'_{\ell'(v')})] \setminus (x, y) = G[D(r'_{\ell'(v')})]$, and moreover $r'_{\ell'(v')}$ is a proper ancestor of v in D' (as it is in D). As $r'_{\ell'(v')} \in D'(r'_{\ell(v)})$, we have $r'_{\ell'(v')} \in D(r_{\ell(v)})$ as non of the ancestor of v in D are D-scanned. That means $r'_\ell(v')$ is a ancestor of v in L and a descendant of $r_{\ell(v)}$, which contradicts the fact that $level'(\ell'(v')) > level'(\ell(v))$. Thus, all paths from a L-affected vertex $v \notin S$ to $\ell'(v')$ contain (x, y).

Now we prove the second part of the lemma. Assume by contradiction that v does not have a path to x in $G[D'(r'_y)]$. Then, all paths contain a vertex $w \notin D'(r'_y)$, and by Lemma 2.1 also vertices $(d'(r'_y), r'_y)$. Therefore, all paths in $G'[D'(r'_{\ell'(v')})]$ from v to x contain r'_y (clearly avoiding (x, y)); let $P_{vr'_y}$ be the subpath from v to r'_y of any such path. By the fact that $y \in D'_y$ it follows that there is a path from r'_y to y avoiding (x, y); let $P_{r'_y y}$ be such a path. Then, $P_{vr'_y} \cdot P_{r'_y y}$ is a path in $G'[D'(r'_{\ell'(v')})]$ from v to y avoiding (x, y), which contradicts the fact that all paths from v to $\ell'(v')$ contain (x, y). □

□

Notice that we only need to consider the vertices that are in $D'(r'_y)$ and are not D-scanned, by Lemma 5.1.

Lemma 5.9. *Let (x, y) be the newly inserted edge. The canonical vertex v' of a vertex $v \in D'(r'_y)$ that is not D-scanned and has a path to x in $G'[D'(r'_y)]$, changes its parent $\ell'(v')$ as follows:*

(1) Case $v \in D'_y$: if $level'(\ell(v)) < level'(\ell'(y'))$ or $\ell(v) = \emptyset$ then $\ell'(v') = \ell'(y')$. Otherwise, $\ell'(v') = c_{\ell(v)}$.

(2) Case $v \notin D'_y$: let (p, q) the strong bridge such that $p \in D'_y$ and q is an ancestor of v in D'.

(2.1) Case $c'_p = y'$: if $level'(\ell(v)) < level'(c'_p)$ or $\ell(v) = \emptyset$, then $\ell'(v') = c'_p$. Otherwise, $\ell'(v') = c'_{\ell(v)}$.

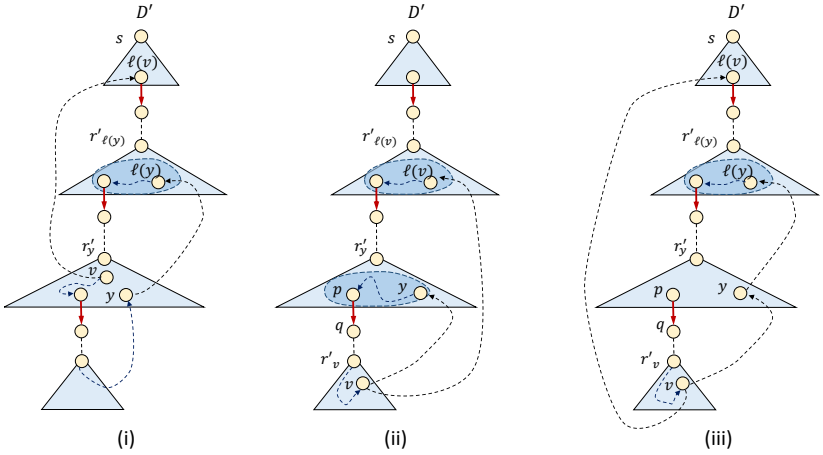

Figure 7: A demonstration of the different case in Lemma 5.9. (i) Case (1) of Lemma 5.9 where $v \in D'_y$ and $level'(\ell(v)) < level'(\ell'(y'))$. Here we have $\ell'(v') = \ell'(y')$. (ii) Case (2.1) of Lemma 5.9 where $v \notin D'_y, c'_p = y'$ and $level'(\ell(v)) < level'(c'_p)$. Now we have $\ell'(v') = c'_p$. (iii) Case (2.2) of Lemma 5.9 where $v \notin D'_y, c'_p \neq y'$ and $level'(\ell(v)) < level'(\ell'(y'))$. In this case $\ell'(v') = \ell'(y')$.

(2.2) Case $c'_p \neq y'$: if $level'(\ell(v)) < level'(\ell'(y'))$ or $\ell(v) = \emptyset$, then $\ell'(v') = \ell'(y')$. Otherwise, $\ell'(v') = c'_{\ell(v)}$.

Proof. First, consider the case where $v = D'_y$. Recall that we consider the update of $\ell'(v')$ for all vertices v that are L-affected but not D-scanned when our algorithm is not involved in a restart. Since no bridge is locally canceled, and no ancestor of v is D-affected, then for every bridge $(d(q), q)$ of G_s for which $v \in D(q)$ we have that $v \in D'(q)$ and $(d(q), q) = (d'(q), q)$ is still a bridge in G'_s. Moreover, no new bridge can appear on the paths from s to v. Hence $(d(r_v), r_v) = (d(r'_v), r'_v) = (d(r'_y), r'_y)$ is a bridge also in G'_s and $r'_y = r_v$. Since $\ell(v) \notin D(r_v)$, it follows that $\ell(v) \notin D'(r'_y)$. Therefore, $z = \ell(v)$ is the canonical vertex with the largest level for which v and z are strongly connected in $G[D(r_z)] = G[D'(r'_z)]$ (the equality holds by Lemma 4.2); that is, without using the edge (x, y). In the case where $\ell(v) = \emptyset$ there is no such vertex. If v has a path to x in $G'[D'(r'_y)]$, then $\ell'(y')$ and v are strongly connected in $G'[D'(r'_{\ell'(y')})]$, as (i) v has a path to $\ell'(y')$ in $G'[D'(r'_{\ell'(y')})]$ through y, and (ii) $\ell'(y')$ has a path to v in $G'[D'(r'_{\ell'(y')})]$ through r'_y (i.e., the very same subpath P_1 from $\ell'(y')$ to r'_y as in the path from $\ell'(y')$ to y, followed by any path from r'_y to v). Therefore, if v has a path to x in $G'[D'(r'_y)]$ then $\ell'(y')$ is an ancestor of v' in L'. That includes the case where $\ell(v) = \emptyset$. Now we show that if v is L-affected then $\ell'(v')$ is an ancestor of y' in L'. By Lemma 5.8, it follows that all paths in $G'[D'(r'_{\ell'(v')})]$ from v' to $\ell'(y')$ contain (x, y), and therefore y has a path to $\ell'(y')$ in $G'[D'(r'_{\ell'(v')})]$ as well. Moreover, $\ell'(v')$ has a path to y in $G'[D'(r'_{\ell'(v')})]$ through r'_y (i.e., the very same subpath from $\ell'(v')$ to r'_v as in the path from $\ell'(v')$ to v, followed by any path from r'_y to y). Collectively, we showed that if v has a path to x in $G'[D'(r'_{\ell'(y')})]$, then $\ell'(y')$ is an ancestor of v' in L', and moreover if v is L-affected then $\ell'(v')$ is an ancestor of y' in L'.

Thus, $\ell'(v') = \ell'(y')$ if $level'(\ell(v)) < level'(r'_{\ell'(y')})$ or $\ell(v) = \emptyset$ and v reaches x in $G'[D'(r'_y)]$, and $\ell'(v') = c'_{\ell(v)}$ otherwise.

Now we prove the case where $v \notin D'_y$. First, we show that if $level'(\ell(v)) \geq level'(y')$, then $\ell'(v') = \ell(v)$. There is no D-scanned vertex $t \in D'(r'_{\ell(v)})$ since otherwise it is an descendant of a D-affected vertex in D'_y and thus, v is also a D-scanned descendant of the same D-affected vertex. As all D-affected vertices become children of $nca_D(x, y)$ in D, for every vertex $z \in D'(r'_y)$ it holds that $D'(r'_z) \subseteq D(r_z)$. As we assume $level'(\ell(v)) \geq level'(y')$, all new ancestors of v' in L' have level greater than $level'(y)$. Moreover, in the case where $level'(\ell(v)) = level'(y')$ it holds that v and $\ell(v)$ are strongly connected in $G'[D'(r'_y)]$ since $D'(r'_y) \supseteq D(r'_{\ell(v)})$. Hence, $c'_{\ell(v)}$ is an ancestor of v' in L'. As mentioned before, $D'(r'_z) \subseteq D(r_z)$ for all ancestors r'_z of v in D' such that $level'(r_z) > level'(y)$ (including q from the statement of the lemma). Therefore, if $\ell'(v') \neq c'_{\ell(v)}$ then $\ell'(v')$ and v are strongly connected in $G[D(r_{\ell'(v')})]$, and hence, $\ell'(v')$ is an ancestor of v in L with $level'(\ell'(v')) > level'(\ell(v))$. This contradicts the definition of $\ell(v)$. Thus, also in this case it follows $\ell'(v) = \ell(v)$.

Finally, we consider the case where $v \in D'(r'_v) \setminus D'_y$ and $level'(\ell(v)) < level'(y')$ or $\ell(v) = \emptyset$. Let (p, q) be the strong bridge such that $p \in D'_y$ and q is an ancestor of v in D'. As mentioned before, $D'(r'_z) \subseteq D(r_z)$ for all ancestors r'_z of v in D' (including q) such that $level'(r'_z) > level'(y)$. Therefore, if $\ell'(v') \neq c_{\ell(v)}$ then $\ell'(v')$ cannot be a descendant of q in D' since $\ell'(v')$ and v is strongly connected in $G[D(r'_{\ell'(v')})]$, and therefore, $\ell'(v')$ is an ancestor of v in L (which contradicts the definition of $\ell(v)$). Notice that if v did not have a path to x before the insertion in $G[D'(r'_y)]$, by Lemma 5.8 v is not L-affected. Next we assume that v has a path to x and y in $G'[D'(r'_y)]$. If y and p are strongly connected in $G'[D'(r'_y)]$ (i.e., $y' = c'_p$), then also v and c'_p are strongly connected in $G'[D'(r'_y)]$, since v has a path to y and p has a path to v in $G'[D'(r'_y)]$. If additionally $r'_{\ell'(v')}$ is an ancestor of r'_p in D' (that is, $level'(\ell(v)) < level'(\ell'(v')) \leq level'(p)$), by the definition of the hyperloop nesting forest $\ell'(v') = c'_p$, including the case where $\ell(v) = \emptyset$. To prove the case where $y' \neq c'_p$ we can use the same argument as in the case where $v \in D'_y$. □

□

Lemma 5.9 shows how to determine the new parent in L' of each canonical vertex $v \in D'(r'_y)$ that is L-affected but not D-scanned. The pseudocode for this update is given in Algorithm 4. In the following we show how we can efficiently answer all the tests of Algorithm 4. The most challenging computation is to determine which vertices have a path to x in $G'[D'(r'_y)]$ as required in Line 1. We show how to compute efficiently those vertices by executing a backward traversal: this runs in time proportional to the sum of the degrees of the L-affected vertices. We start with the following definition of *loop cover* of a vertex, which we use to speed up our backward search.

Definition 5.10. *Let $w \in D'(r'_y) \setminus S$ be a canonical vertex, and let ℓ_{min} be the ancestor of w in L with the lowest level such that $\ell_{min} \in D'(r'_y)$. Moreover, let (p, q) be the bridge such that $p \in D'_{\ell_{min}}$ and q is an ancestor of w. We call q the* loop cover *$lcover(w) = q$ of w in D. If $\ell(w) \notin D'(r'_y)$, then $lcover(w) = \emptyset$.*

We use the loop cover of vertices that are neither D-scanned nor L-affected in order to avoid unnecessary visits to vertices during the search for L-affected

Algorithm 4: Update-L-affected$(\mathcal{D}, L, x, y, S)$

```
foreach canonical vertex v ∈ D'(r'_y), v ∉ S, that has a path to x in
  G'[D'(r'_y)] do
    if v ∈ D'_y then
        if level'(ℓ(v)) < level'(ℓ'(c'_y)) or ℓ(v) = ∅ then ℓ'(c'_v) = ℓ'(c'_y)
    else
        Let (p, q) be the bridge such that p ∈ D'_y and q is an ancestor of v
          in D'.
        if c'_p = c'_y and level'(ℓ(v)) < level'(c'_p) or ℓ(v) = ∅ then
          ℓ'(c'_v) = c'_p
        else if c'_p ≠ c'_y and level'(ℓ(v)) < level'(ℓ'(c'_y)) or ℓ(v) = ∅ then
          ℓ'(c'_v) = ℓ'(c'_y)
    end
end
```

vertices. Whenever we visit a vertex $w \in D'(r'_y) \setminus D'_y$ that is not L-affected, then we do not need to visit any of the vertices in $D'(lcover(w))$. Formally, we have the following lemma.

Lemma 5.11. *Let $w \in D'(r'_y) \setminus D'_y$ be a canonical vertex that is not D-scanned and has a path to x in $G'[D'(r'_y)]$ and $\ell'(w') = c'_{\ell(w)}$. If $lcover(w) \neq \emptyset$, for every canonical vertex $v \in D'(lcover(w))$ such that v has a path to w in $G'[D'(lcover(w))]$, we have that $\ell'(v') = c'_{\ell(v)}$. If $lcover(w) = \emptyset$, for all vertices $v \in D'(r'_y)$ that have a path to w in $G'[D'(r'_y)]$, it holds that $\ell'(v') = c'_{\ell(v)}$.*

Proof. First, assume $lcover(w) \neq \emptyset$. As (i) v has a path to w in $G'[D'(lcover(w))]$, (ii) all paths from $\ell(lcover(w))$ to w in $G'[D'(lcover(w))]$ contain $lcover(w)$, and (iii) $v \in D(lcover(w))$, it follows that $lcover(w)$ and v are strongly connected in $G'[D'(lcover(w))]$. Hence, by definition of hyperloop nesting forest, $c'_{\ell(lcover(w))}$ is an ancestor of v' in L' and $level'(\ell(lcover(w))) \geq level'(y)$. Note that Lemma 5.9 implies that the insertion of (x, y) might only introduce an ancestor t of v' in L' such that $level'(t) \leq level'(y)$. Therefore, by the fact that $c'_{\ell(lcover(w))}$ is an ancestor of v' in L' and $level'(\ell(lcover(w))) \geq level'(y)$, it follows that $\ell'(v') = c'_{\ell(v)}$.

No assume $lcover(w) = \emptyset$ and by contradiction that v is L-affected. We have that $\ell'(w') = c'_{\ell(w)} \notin D'(r'_y)$. Since $D'(r'_y) \subset D'(r'_{\ell'(w')})$, v has a path to w in $G'[D'(r'_{\ell'(w')})]$. Moreover, there is a path from $\ell'(w')$ to v in $G'[D'(\ell'(w'))]$ as all paths from $\ell'(w')$ to w contain r'_y and $v \in D'(r'_y)$. Hence, $\ell'(w')$ is an ancestor of v' in L'. Therefore, $level'(w') < level'(\ell'(v')) \leq level'(r'_y)$ as v is L-affected. By Lemma 4.2 we have that $r'_{\ell'(v')}$ is an ancestor of v in D, and by Lemma 2.1 all paths from $\ell'(v')$ to v contain r'_y and can also avoid (x, y) (as it is not a bridge in G'_s). By Lemma 4.2 we have that $D'(r'_{\ell'(v')}) = D(r_{\ell(v)})$, and therefore, $\ell'(v')$ has a path to v in $G[D(r_{\ell(v)})]$ (which avoids (x, y)). As $\ell'(v')$ is not an ancestor of v before the insertion, by Lemma 5.8, all paths from v' to $\ell'(v')$ in $G'[D'(r'_{\ell'(v')})]$ contain the edge (x, y). Hence, $x, y, \ell'(v')$, and v' are strongly connected in $G'[D'(r'_{\ell'(v')})]$. As w has a path to x in $G'[D'(r'_{\ell'(v')})]$ and $\ell'(v')$ has a path to v' in $G'[D'(r'_{\ell'(v')})]$, it follows that $\ell'(v')$ and w are

strongly connected in $G'[D'(r'_{\ell'(v')})]$. This implies that $\ell'(v')$ is an ancestor of w in L' with $level'(\ell'(v')) > level'(\ell'(w'))$ which contradicts the definition of $\ell'(w')$. The lemma follows. □

□

Lemma 5.12. *If a bridge is not locally canceled by the edge insertion, the set S' of L-affected vertices can be identified and L' can be correctly updated in time $O(V(S') + E(S') + V(S) + E(S) + n)$.*

Proof. By Lemma 5.7, we can compute $\ell'(v')$ for each $v \in S$ in time $O(V(S) + E(S))$. For every vertex $v \in S' \setminus S$, we set its value $\ell'(v')$, according to Lemma 5.9. Notice that in all cases of Lemma 5.9, a vertex $v \in S' \setminus S$, should reach x in $G'[D'(r'_y)]$. Therefore, a straightforward way to test whether a vertex in $D(r'_y)$ changes its parent in L', and compute the new parent, is to start a backward traversal in $G'[D'(r'_y)]$ from x, and for each vertex $v \notin S$ that is visited by the traversal apply Lemma 5.9. This, takes $O(m + n)$ in the worst case. We next present a charging scheme to achieve the claimed bound.

First, we can identify $lcover(v)$ for all canonical vertices v, after every edge insertion, in time $O(n)$ by traversing the forest L from each canonical vertex $z \in D'(r'_y), \ell(z) \notin D'(r'_y)$, and setting for each descendant v of z in L the $lcover(v) = q$ such that (p, q) is the strong bridge for which $p \in D'_z$ and $v \in D'(q)$. Notice that (p, q) is the $level'(q)$-th bridge in the path from s to v in D'. In order to identify for each descendant v of z in L the vertex q, we proceed as follows. Initially, we set to each child of z in L the level of q (which is $level'(z)+1$), that is, each z knows $level'(lcover(v))$. Next, we begin a traversal from s on the tree D' keeping track of the bridges that exist on the path from s to the current vertex v, and once we visit a vertex v that is assigned a value $level'(lcover(v))$, we set $lcover(v)$ to be q where (p, q) is the $level'(lcover(v))$-th bridge on the path from s to v (we keep track of this information during the traversal). Therefore, the total time spent in this computation after all edge insertions is $O(mn)$, where m is the number of edges after all insertions.

Next we start a backward traversal from x in $G'[D'(r'_y)]$; that is, we visit all vertices $v \in D'(r'_y)$ that have an edge to x, and consecutively to each visited vertex. During the traversal we act as follow. Whenever the traversal reaches an unvisited vertex v, we test whether $\ell'(v') \neq c'_{\ell(v)}$: if this is the case, we set $\ell'(v')$ according to Lemma 5.9, we iterate over the incoming edges of v and recursively traverse each vertex w that has an incoming edge to v. If $\ell'(v') = c'_{\ell(v)}$ and $v \notin D'_y$ we do not traverse any incoming edge to v, and instead, we continue the traversal from $lcover(v)$ $(= lcover'(v')$ since $\ell'(v') = c'_{\ell(v)})$ if $lcover(v) \neq \emptyset$, or otherwise we do not visit any vertices from v. It is correct to continue the traversal from $lcover(v)$ in the case where $\ell'(v') = c'_{\ell(v)}$ and $v \notin D'_y$ since by Lemma 5.11 for every vertex $w \in D'(lcover(v))$, such that w reaches v in $G'[D'(lcover(v))]$ we have that $\ell'(c'_w) = c'_{\ell(c_w)}$. If, on the other hand, $\ell'(v') = c'_{\ell(v)}$ and $v \in D'_y$, then we backtrack the traversal from v. This is correct as $level'(\ell(v)) \geq level'(\ell'(y'))$ follows from Lemma 5.9, and therefore for all the vertices $z \in D'(r'_y)$ that have a path to v in $G'[D'(r'_y)]$, $c'_{\ell(v)}$ is an ancestor of c'_z in L' (since z and $\ell(v)$ are strongly connected in $G'[D'(r'_{\ell(v)})]$). Moreover, whenever the traversal reaches a vertex in S, we have time to traverse all the vertices in S and their incoming edges; we continue the traversal without testing

or updating its value $\ell'(v')$. By the above description it is clear that our traversal does not traverse the edges of a vertex that is neither L-affected nor D-scanned (that is, the vertices in $V \setminus S' \cup S$). Collectively, we traverse only D-scanned vertices, L-affected vertices, and vertices for which we test in constant time whether $\ell'(v') = c'_{\ell(v)}$. Thus, we spend time $O(V(S')+E(S')+V(S)+E(S)+n)$.

Next we show that we correctly update the forest L' after an edge insertion. Clearly, the vertices whose edges are traversed by the traversal contain the correct value in L'. Now we argue that all vertices v such that $\ell'(v') \neq c'_{\ell(v)}$ correctly change their parent in L'. Since for all visited vertices we update correctly the value $\ell'(v')$, we need to show that the backwards traversal that we execute visits all vertices that change their value $\ell'(v')$. We already showed in Lemma 5.6 that we do this correctly for the vertices in S. Assume, by contradiction, that this is not true for some vertex $v \notin S$, and therefore, the algorithm fails to set $\ell'(v')$ according to Lemma 5.9. Let z be the correct value $\ell'(v')$ that the algorithm fails to assign. Since v has a path P to z in $G'[D'(r'_z)]$ only after the insertion of (x, y), P goes through (x, y). Let P_{vx} be the subpath of P from v to x. Moreover, let w be the first vertex on P_{vx} that is visited by the traversal, i.e., the traversal did not visited any vertices that appear before w on P_{vx} (recall that we assume the traversal did not visited v). We know that $\ell'(c'_w) = c'_{\ell(c_w)}$ since otherwise the traversal visits the predecessor of w on P_{vx}. Assume first that $w \in D'(r'_y) \setminus D'_y$. Let (p, q) be the first strong bridge on $D'[r'_{\ell'(c'_w)}, w]$. Vertex v is not a descendant of q in D', since otherwise by Lemma 5.11 it holds $\ell'(v') = c'_{\ell(v)}$, which contradicts our assumption. Thus, v is not a descendant of q in D', and therefore, all paths from v to w go through (p, q), by the properties of the dominator tree. That means $p \in P_{vx}$ and p appears earlier than w on P_{vx}. According to the rules of the traversal, if w is visited and $\ell'(c'_w) = c'_{\ell(c_w)}$, then the traversal continues from p, so p must have visited during the traversal. This contradicts the choice of w as the earliest vertex on P_{vx} that is visited. If $w \in D'_y$, then by the assumption that $\ell'(c'_w) = c'_{\ell(c_w)}$ and by Lemma 5.9, we have that $level'(y) > level'(\ell'(c'_w)) \geq level'(\ell'(y'))$. Since v has a path to w in $G'[D'(r'_{y'})] \subset G'[D'(r'_{\ell'(c'_w)})]$ and $\ell'(c'_w)$ has path to v in $G'[D'(r'_{\ell'(c'_w)})]$ (through r'_y; such a path exists since $\ell'(c'_w)$ has a path to w in $G'[D'(r'_{\ell'(c'_w)})]$), it follows that v and $\ell'(c'_w)$ are strongly connected in $G'[D'(r'_{\ell'(c'_w)})]$. Therefore, $level'(\ell'(v')) > level'(\ell'(y'))$, which contradicts the assumption that $\ell'(v') \neq c'_{\ell(v)}$ according to Lemma 5.9. Thus, all vertices v for which $\ell'(v') \neq c'_{\ell(v)}$ are visited by the traversal, and the lemma follows. □

□

Finally, we bound the total time spend over any sequence of m insertions.

Lemma 5.13. *After any sequence of edge insertions in a flow graph G_s, any canonical vertex v changes its parent $\ell'(c'_v)$ in L' at most t times, where $t < n$ is the number of bridges dominating v in D before any insertion. Moreover, L-affected vertices can be identified and correctly updated in a total of $O(mn)$ time for all edge insertions, where m is the number of edges after all edge insertions.*

Proof. We first bound the number of times that a vertex v can be L-affected but not D-scanned, that is $v \in S' \setminus S$. Note that the number of bridges in a flow graph G_s is at most $n - 1$. Therefore, the number of distinct bridges

that appear on the path $D[s, v]$ for each vertex v throughout the course of the algorithms is at most $n-1$. Our strategy is to show that for each strong bridge on $D[s, v]$ for any vertex v a vertex is L-affected and not D-scanned at most once. We first claim that once a vertex w becomes an ancestor of v in L, such that $v \in D(r_w) \setminus D_w$, w' remains an ancestor of c'_v, after any edges insertion, as long as $v' \in D(r'_w) \setminus D'_w$. For every vertex z on the path from w to v in $G[D(r_w)]$ we have that $z \in D'(r'_w)$ as otherwise there is a path from a vertex z to v avoiding r'_w, which means a path from s to v avoiding r'_w, a contradiction to the fact that $v \in D'(r'_w)$. Hence, there is a path from w to v in $G'[D'(r'_w)]$. We use the same argument to show that there is a path from v to w in $G'[D'(r'_w)]$. For every vertex z on the path from v to w in $G[D(r_w)]$ remains in $D'(r'_w)$ as otherwise there is a path from a vertex z to w avoiding r'_w, which means a path from s to w avoiding r'_w, a contradiction to the fact that $w \in D'(r'_w)$. Therefore, there is a path from v to w in $G'[D'(r'_w)]$. Thus, w and v are strongly connected in $G'[D'(r'_w)]$, which means w' is an ancestor of v' in L'.

Every time v is L-affected but not D-scanned we have that $level'(\ell'(v')) > level'(\ell(v))$, as implied by Lemma 5.9. Therefore, v is assigned an ancestor $\ell'(v')$ such that there is no ancestor w of v where $c'_w = \ell'(v')$ (as otherwise v is be L-affected). Moreover, as we shown above, as long as $(d'(r'_{\ell'(v')}), r'_{\ell'(v')})$ is a bridge such that $v \in D'(r'_{\ell'(v')}) \setminus D_{\ell'(v')}$, vertex $\ell'(v')$ remains ancestor of v in L'. Hence, since at most $n-1$ bridges can appear on the path $D[s, v]$, it follows that v can be at most $O(n)$ times L-affected but not D-scanned.

Now we bound the overall running time spent on identifying and updating the L-affected vertices. First note that if an L-affected vertex is also D-scanned, then by Lemma 5.7 we can update their parent in L' in time $O(V(S) + E(S))$ where S is the set of D-scanned vertices. We charge this time to the algorithm for updating the dominator tree, which spends $O(V(S) + E(S))$ time after each edge insertion. Thus, the overall time spent on updating the parent of a vertices that are D-scanned is $O(mn)$. Now, let S' be the set of vertices that are L-affected. By Lemma 5.12, S' can be identified in time $O(V(S') + E(S') + V(S) + E(S) + n)$. We again charge the time $O(V(S) + E(S))$ to the algorithm for updating the dominator tree, which sums to $O(mn)$ after all insertions. As we showed above, a vertex u can be in $S' \setminus S$ at most $n-1$ times. Hence, the time $O(V(S') + E(S'))$ per insertion considers each vertex at most $n-1$ times, and therefore, it takes time $O(mn)$ time. Finally, the time $O(n)$ spent after every edge insertion sums to $O(mn)$ overall, since we have at most m insertions. The bound follows. □

□

Theorem 5.14. *Let G_s be a flow graph with n vertices. We can maintain the hyperloop nesting forest L of G_s through a sequence of edge insertions in $O(mn)$ total time, where m is the number of edges after all insertions.*

6 Answering queries in optimal time

The data structure from [20] computes the strong bridges of G plus four trees: the dominator tree D and the loop nesting tree H of the flow graph G_s, and the dominator tree D^R and the loop nesting tree H^R of the reverse flow graph G_s^R.

This information is sufficient to answer in optimal time all the following types of queries:

(i) Report in $O(1)$ time the total number of SCCs in $G \setminus e$, for a query edge e in G.

(ii) Report in $O(1)$ time the size of the largest and of the smallest SCCs in $G \setminus e$, for a query edge e in G.

(iii) Report in $O(n)$ worst-case time all the SCCs of $G \setminus e$, for a query edge e.

(iv) Test in $O(1)$ time if two query vertices u and v are strongly connected in $G \setminus e$, for a query edge e.

(v) For query vertices u and v that are strongly connected in G, report all edges e such that u and v are not strongly connected in $G \setminus e$, in optimal worst-case time, i.e., in time $O(k+1)$, where k is the number of separating edges.

We note that queries of type (i) and (ii) require additional $O(n)$ preprocessing time, based solely on the same four trees. In particular, the crux of the method is the following theorem, which shows that the information relevant for our queries can indeed be extracted from the strong bridge of G and the four trees D, D^R, H and H^R:

Theorem 6.1 ([20]). *Let $G = (V, E)$ be a strongly connected digraph, s be an arbitrary start vertex in G, and let $e = (u, v)$ be a strong bridge of G. Let C be a SCC of $G \setminus e$. Then one of the following cases holds:*

(a) If e is a bridge in G_s but not in G_s^R then either $C \subseteq D(v)$ or $C = V \setminus D(v)$.

(b) If e is a bridge in G_s^R but not in G_s then either $C \subseteq D^R(u)$ or $C = V \setminus D^R(u)$.

(c) If e is a common bridge of G_s and G_s^R then either $C \subseteq D(v) \setminus D^R(u)$, or $C \subseteq D^R(u) \setminus D(v)$, or $C \subseteq D(v) \cap D^R(u)$, or $C = V \setminus \big(D(v) \cup D^R(u)\big)$.

Moreover, if $C \subseteq D(v)$ (resp., $C \subseteq D^R(u)$) then $C = H(w)$ (resp., $C = H^R(w)$) where w is a vertex in $D(v)$ (resp., $D^R(u)$) such that $h(w) \notin D(v)$ (resp., $h^R(w) \notin D^R(u)$).

Now we show that exactly the same information can be extracted if we replace the loop nesting trees H and H^R with two new trees $\hat{H}$ and $\hat{H}^R$, which (differently from loop nesting trees) can be maintained efficiently throughout any sequence of edge insertions. As a result, the strong bridges of G plus D, D^R, $\hat{H}$ and $\hat{H}^R$ allow us to answer all our queries in optimal time throughout any sequence of edge insertions.

We next define the new trees $\hat{H}$ and $\hat{H}^R$. Without loss of generality, we restrict our attention to $\hat{H}$, as $\hat{H}^R$ is defined in the reverse graph G^R in a completely analogous fashion. We construct $\hat{H}$ starting from the hyperloop nesting tree L, as follows. For every vertex u such that $c_u \neq u$ we set $\hat{h}(u) = c_u$, and for every vertex u where $c_u = u, u \neq s$ we set $\hat{h}(u) = \ell(u)$. Note that, once L is available, the tree $\hat{H}$ can be computed in $O(n)$ time.

As suggested by Theorem 6.1, every SCC C in $G \setminus (u, v)$ is either a subtree of H rooted at a vertex $w \in D(v)$ such that $h(w) \notin D(v)$, or a subtree of H^R rooted at a vertex $z \in D^R(u)$ such that $h^R(z) \notin D^R(u)$, or $C = V \setminus D(v) \cup D^R(u)$.

As a consequence, in order to show that we can safely replace H by $\hat{H}$ and H^R by $\hat{H}^R$, we only need to prove the following lemma, which holds symmetrically also for G_s^R, D^R, H^R and $\hat{H}^R$. First, we start with an intermediate technical lemma that is used in the proof or Lemma 6.3.

Lemma 6.2. *Let w be the nearest common ancestor of two vertices u and v in H, and let z be the nearest common ancestor of c_u and c_v in L. Then, $c_w = z$.*

Proof. It suffices to show that, for every ancestor u of a vertex v in H, c_u is an ancestor of c_v in L. First, note that if $c_u = c_v$ then they are in the same auxiliary component and thus the above statement trivially holds. Now let $c_u \neq c_v$. By Lemma 3.3, v and h_v map to the same vertex in L. Moreover, $c_{h(h_v)}$ is the parent of c_ℓ in L. By repeatedly applying the same argument, it follows that for every ancestor u of v in H, c_u is an ancestor of v in L. □

□

Lemma 6.3. *Let (u, v) be a strong bridge in G_s. For every set $H(w)$ where $w \in D(v)$ and $h(w) \notin D(v)$ there is a vertex $z \in D(v)$ and $\hat{h}(z) \notin D(v)$ such that $\hat{H}(z) = H(w)$. Additionally, for every set $\hat{H}(z)$ where $z \in D(v)$ and $\hat{h}(z) \notin D(v)$ there is a vertex $w \in D(v)$ and $h(w) \notin D(v)$ such that $H(w) = \hat{H}(z)$.*

Proof. We prove the lemma by proving the following two statements:

(i) Let vertex $w \in D(v)$ such that $h(w) \notin D(v)$. Then, for every vertex $p \in H(w)$ there is a vertex $z \in D(v)$ such that $p \in \hat{H}(z)$.

(ii) Let vertex $z \in D(v)$ such that $\hat{h}(z) \notin D(v)$. Then, for every vertex $t \in \hat{H}(z)$ there is a vertex $w \in D(v)$ such that $t \in H(w)$.

We start with statement (i) for $z = c_w$. By Lemma 3.3, for each vertex $p \in H(w) \cap D_v$ we have that $c_p = c_w$. By definition of $\hat{H}$, it follows that p is a child of c_w in $\hat{H}$ (thus, $p \in \hat{H}(c_w)$). For any other vertex $p \in H(w)$, the nearest common ancestor of p and w in H is w. By Lemma 6.2, the nearest common ancestor of c_w and c_p is c_w. Therefore, c_p is a descendant of c_w in $\hat{H}$ and p is a child of c_p in $\hat{H}$ (or $p = c_p$). Thus, $p \in \hat{H}(c_w)$. This proves that for every vertex $p \in H(w)$, $p \in \hat{H}(c_w)$.

We not turn to statement (ii). Since $\hat{h}(z) \notin D(v)$ it holds that z and $\hat{h}(z)$ are not in the same auxiliary component. Therefore, by definition of $\hat{H}$, $c_z = z$. We prove statement (ii) for $w = h_z$. Since $z \in H(w)$, so do vertices t such that $c_t = z$ (including w), since they are children of z in $\hat{H}$. Now let t be a vertex in $\hat{H}(z)$. By definition of $\hat{H}$, t is a child of c_t in $\hat{H}$ (or c_t). Hence, the nearest common ancestor of c_t and c_z in $\hat{H}$, and therefore in L, is z. Thus, by Lemma 6.2, for the nearest common ancestor q of p and z in H, we have that $c_q = z$. This implies that q is a child of z in H and thus $p \in H(w)$. □

□

In summary, our algorithm works as follows. Given a strongly connected digraph G subject to edge insertions, we maintain in a total of $O(mn)$ time the strong bridges of G [19], the dominator trees D and D^R [18], and the hyperloop nesting trees L and L^R, by Theorem 5.14. After each edge insertion, we construct in $O(n)$ time the trees $\hat{H}$ and $\hat{H}^R$ from L and L^R, respectively. Since there can be at most m edge insertions, where m is the final number of edges after all edge insertions, the total time spent on all those computations

is $O(mn)$. By Lemma 6.3, after each update we can answer all our queries in optimal time.

Corollary 6.4. *We can maintain a strongly connected digraph G through any sequence of edge insertions in a total of $O(mn)$ time, where m is the number of edges after all insertion, so as to answer the following queries in optimal time after each insertion:*

(i) Report in $O(1)$ time the total number of SCCs in $G \setminus e$, for a query edge e in G.

(ii) Report in $O(1)$ time the size of the largest and of the smallest SCCs in $G \setminus e$, for a query edge e in G.

(iii) Report in $O(n)$ worst-case time all the SCCs of $G \setminus e$, for a query edge e.

(iv) Test in $O(1)$ time if two query vertices u and v are strongly connected in $G \setminus e$, for a query edge e.

(v) For query vertices u and v that are strongly connected in G, report all edges e such that u and v are not strongly connected in $G \setminus e$, in optimal worst-case time, i.e., in time $O(k+1)$, where k is the number of separating edges.

6.1 Extension to general graphs

In this subsection we extend Corollary 6.4 to general (not necessarily strongly connected) graphs within the same time bounds. The main idea is to maintain the necessary structures for each SCC of the input graph G, rooted at suitable vertices. As edges are inserted into G, several SCCs may merge, so we need to update our structures accordingly. In [19] it is shown how to maintain the dominator tree, the bridge decomposition, and the auxiliary components, all rooted at the same start vertex s, of each SCC of a general graph under any sequence of edge insertions in total time $O(mn)$. Since we always maintain the hyperloop nesting tree rooted at the same start vertex as the dominator tree, we will be using the same start vertices as the algorithm in [19]. Next we briefly review these choices of start vertices.

The algorithm in [19] maintains incrementally the SCCs of the graph using the algorithm from [5]. In each SCC the algorithm maintains an instance of the data structure that we developed for strongly connected graphs. Whenever an edge insertion causes two or more SCCs to merge, a new data structure instance is created for the newly merged SCC as follows. Let $C_1, C_2, \ldots, C_j$ be the SCCs that are merged into C after the insertion of an edge. We choose the start vertex of C to be the start vertex of the largest SCC C_i and we restart the algorithm in C. We refer to the component C_i as the *principal component of C*.

We now show that the choices of start vertices allow our algorithm from Section 5 to run in a total of $O(mn)$ time, when executed in each SCC independently. The total time required to maintain instances of the algorithm in the SCCs of the input graph G is $O(mn)$. The total number of new SCCs that can be created is at most $n-1$. In the insertion of an edge results into a merge between two SCCs, the algorithm restarts in the newly merged component: each

time we restart the algorithm, it takes $O(m)$ time to initialize the data structures. We remark that these restarts are different from the restarts described in Section 5, that are executed whenever a strong bridge is locally canceled. We refer to this new type of restarts as *top-level restarts*. In summary, the total time required to maintain the SCCs and the time spent for the top-level restarts of the algorithm is $O(mn)$.

By Lemma 5.12, after each edge insertion, that does not result to a restart of the algorithm, the hyperloop nesting tree is updated in time $O(V(S')+E(S')+V(S)+E(S)+n')$ where S' and S are the set of L-affected and D-scanned vertices after the edge insertion, and n' is the number of vertices inside the SCCs of both x and y. To bound the running time, we show that each vertex v can be L-affected at most $O(n)$ times and D-scanned at most $O(n)$ times, through the whole execution of the algorithm. This holds despite the fact that a vertex can be part of many SCCs, as they merge while the graph undergoes edge insertions. Let C be the current SCC containing v, and let C' be the new SCC that contains C after a merge caused by an edge insertion. If C is the principal subcomponent of C', then the depth of v may only decrease as we keep the same start vertex in C' as in C. Otherwise, the depth of v may increase. The *effective depth* of v after merging C into C' is defined as follows: it is zero, if C is the principal subcomponent of C', and is equal to the depth of v in the dominator tree of $G_s[C']$ otherwise. To bound the total amount of work needed to maintain the hyperloop nesting tree of all SCCs, we compute the sum of the effective depths of v in all the SCCs that v is contained through the execution of algorithm. We refer to this sum as the *total effective depth of* v.

Lemma 6.5 ([19]). *The total effective depth of any vertex v is $O(n)$.*

Therefore, each vertex can be D-scanned at most $O(n)$ times. Moreover, by Lemma 5.13, a vertex v can be L-affected at most t times, where t is the number of strong bridges that dominate over v before any edge insertion takes place. Clearly, $t \leq n'$, where n' is the number of vertices inside the SCC containing v. Note that, whenever some SCCs merge into one SCC C', if v belongs to the principal component of C', then the number of bridges dominating v in D may only decrease. Otherwise, the number of strong bridges dominating v may increase up to $|V(C')|$, and therefore v may be L-affected $|V(C')|$ times again in the future. By summing these numbers each time some SCCs merge, the number of times that a vertex can be L-affected equals the effective depth of the vertex. Thus, by Lemma 6.5, each vertex can become L-affected at most $O(n)$ times.

In summary, we have shown that each vertex can be D-scanned and L-affected at most $O(n)$ times. By Lemma 5.12, it immediately follows that our algorithm runs in total time $O(mn)$ under any sequence of edge insertions, except for the edge insertions that result to restarts of the algorithm. Each of the restarts takes $O(m)$ time, as we shown previously. The following lemma from [19] shows that, throughout any sequence of edge insertions, at most $O(n)$ strong bridges can appear in the graph, which results to at most $O(n)$ restarts of the algorithm.

Lemma 6.6 ([19]). *During any sequence of edge insertions on a general graph, at most $2(n-1)$ strong bridges can appear.*

Thus, we have the following theorem.

Theorem 6.7. *Let G be a general graph with n vertices. Both the dominator trees D and D^R, the hyperloop nesting trees L and L^R of each SCC C of G, all rooted at the same arbitrary start vertex s, can be maintained in a total of $O(mn)$ time under any sequence of edge insertions, where m is the number of edges after all insertions.*

7 Answering queries under vertex failures

In this section we extend the queries from Section 6, with respect to vertex failures instead of edge failures. More specifically, given a digraph G, we show how we can answer in asymptotically optimal (worst-case) time the following type of queries under vertex failures:

(i) Report in $O(1)$ time the total number of SCCs in $G \setminus v$, for a query vertex $v \in V$.

(ii) Report in $O(1)$ time the size of the largest and of the smallest SCCs in $G \setminus v$, for a query vertex $v \in V$.

(iii) Report in $O(n)$ time all the SCCs of $G \setminus v$, for a query vertex $v \in V$.

(iv) Test in $O(1)$ time if two query vertices u and w are strongly connected in $G \setminus v$, for a query vertex v.

(v) For query vertices u and w that are strongly connected in G, report all vertices v such that u and w are not strongly connected in $G \setminus v$, in time $O(k+1)$, where k is the number of separating vertices.

Let $G = (V, E)$ be a strongly connected digraph and $s \in V$ be an arbitrary start vertex. For any vertex u in G, we let $\widetilde{D}(u)$ (resp., $\widetilde{D}^R(u)$) denote the set of proper descendants of u in D (resp., D^R), i.e., $\widetilde{D}(u) = D(u) \setminus u$ (resp., $\widetilde{D}^R(u) = D^R(u) \setminus u$). Clearly, $\widetilde{D}(u) \neq \emptyset$ (resp., $\widetilde{D}^R(u) \neq \emptyset$) if and only if either u is a nontrivial dominator of G_s (resp., G_s^R) or $u = s$. Analogously, let D^R and H^R be the dominator tree and the loop nesting tree of G^R, respectively.

In order to answer the above queries we use the framework developed in [20], that characterizes the SCCs of $G \setminus v$, for any $v \neq s$ that is a strong articulation point, according to the following lemma.

Theorem 7.1 ([20])**.** *Let u be a strong articulation point of G, and let s be an arbitrary vertex in G. Let C be a SCC of $G \setminus u$. Then one of the following cases holds:*

(a) If u is a nontrivial dominator in G_s but not in G_s^R then either $C \subseteq \widetilde{D}(u)$ or $C = V \setminus D(u)$.

(b) If u is a nontrivial dominator in G_s^R but not in G_s then either $C \subseteq \widetilde{D}^R(u)$ or $C = V \setminus D^R(u)$.

(c) If u is a common nontrivial dominator of G_s and G_s^R then either $C \subseteq \widetilde{D}(u) \setminus \widetilde{D}^R(u)$, or $C \subseteq \widetilde{D}^R(u) \setminus \widetilde{D}(u)$, or $C \subseteq \widetilde{D}(u) \cap \widetilde{D}^R(u)$, or $C = V \setminus \big(D(u) \cup D^R(u)\big)$.

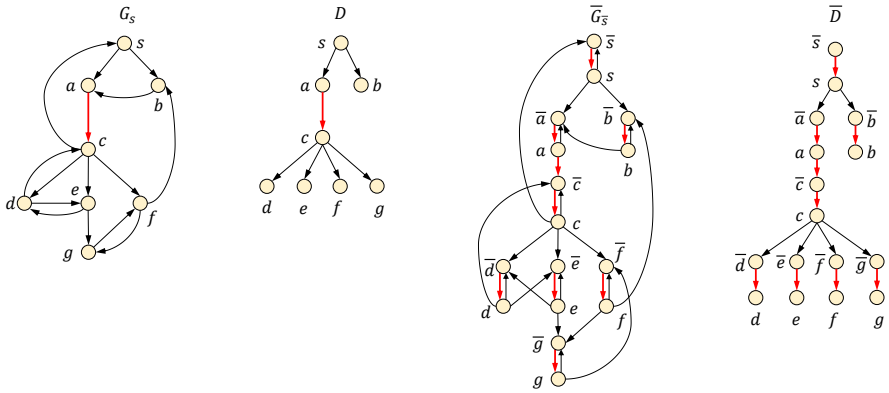

Figure 8: A strongly connected directed flow graph G_s, the dominator tree D of G_s, the graph $\overline{G}_{\overline{s}}$, and the dominator tree $\overline{D}$ of $\overline{G}_{\overline{s}}$. The bridges of G_s and $\overline{G}_{\overline{s}}$ are shown in red.

(d) If $u = s$ then $C \subseteq \widetilde{D}(u)$.
Moreover, if $C \subseteq \widetilde{D}(u)$ (resp., $C \subseteq \widetilde{D}^R(u)$) then $C = H(w)$ (resp., $C = H^R(w)$) where w is a vertex in $\widetilde{D}(u)$ (resp., $\widetilde{D}^R(u)$) such that $h(w) \notin \widetilde{D}(u)$ (resp., $h^R(w) \notin \widetilde{D}^R(u)$).

Next we show a reduction from answering queries under vertex failures to answering queries under edges failures, which allows us to exploit the incremental algorithm from Section 6. For the reduction we built on Lemma 7.1 and on other properties that were shown in [20]. We construct a new graph $\overline{G} = (\overline{V}, \overline{E})$ that results from G by applying the following transformation. For each strong articulation point x of G (i.e., nontrivial dominator in G_s or G_s^R), we add an auxiliary vertex $\overline{x} \in \overline{V}$ and add the auxiliary edges $(\overline{x}, x)$ and $(x, \overline{x})$. Moreover, we replace each edge (u, x) entering x in G with an edge $(u, \overline{x})$ entering $\overline{x}$ in $\overline{G}$. Note that this transformation maintains the strong connectivity of $\overline{G}$. Call the vertices of $V \subset \overline{V}$ ordinary. The resulting graph $\overline{G}$ has n auxiliary vertices and $2n$ auxiliary edges. Hence, $|\overline{V}| = 2n$ and $|\overline{E}| = m + 2n$. Finally, we choose vertex $\overline{s}$ as the start vertex of $\overline{G}$. See Figure 8.

Let $\overline{D}$ and $\overline{D}^R$ be the dominator trees of $\overline{G}_{\overline{s}}$ and $\overline{G}_{\overline{s}}^R$, respectively. The following lemma states the correspondence between the strong articulation points in G (except for s) and the strong bridges in $\overline{G}$. See Figure 8.

Lemma 7.2. *Let $x \neq s$ be a strong articulation point of digraph G. Then the following properties hold:*

(i) Let x be a nontrivial dominator of G_s (resp., G_s^R). Then the auxiliary edge $(\overline{x}, x)$ is a strong bridge of $\overline{G}$ and a bridge of $\overline{G}_{\overline{s}}$ (resp., $\overline{G}_{\overline{s}}^R$).

(ii) For any ordinary vertex $u \in V \setminus x$, we have that $u \in D(x)$ (resp., $u \in D^R(x)$) if and only if $u \in \overline{D}(x)$ (resp., $u \in \overline{D}^R(\overline{x})$).

(iii) All vertices in a SCC of $G \setminus x$ are the ordinary vertices in a SCC of $\overline{G} \setminus (\overline{x}, x)$.

Proof. We prove (i) and (ii) only for the case where x is a nontrivial dominator in G_s since the case where x is a nontrivial dominator in G_s^R is completely

analogous. Note that x has indegree one in $\overline{G}$, and hence $(\overline{x}, x)$ is a strong bridge of $\overline{G}$. By the fact that $\overline{G}$ is strongly connected, there is at least one path from $\overline{s}$ to x. All such paths must contain $(\overline{x}, x)$, so $(\overline{x}, x)$ is a bridge in $\overline{G}_{\overline{s}}$. This implies (i).

In order to prove (ii), we show that $\overline{s}$ has a path to an ordinary vertex v avoiding $(\overline{x}, x)$ in $\overline{G}$ if and only if s has a path to v avoiding x in G. We start with one direction. Let W be the set of vertices such that all paths from $\overline{s}$ to vertices in W contain $(\overline{x}, x)$. Clearly there is no edge $(w, z) \neq (\overline{x}, x)$ such that $w \notin W$, $z \in W$, as that means there is a path from $\overline{s}$ to $z \in W$ avoiding $(\overline{x}, x)$, which contradicts the fact that $z \in W$. Therefore, by the construction of $\overline{G}$ it follows that there is no edge (w, z) in G, such that $w \notin W, w \neq x$ and $z \in W$. Thus all paths from $s \notin W$ to ordinary vertices in W contain x. We now prove the opposite direction of our claim: namely, if x appears in all paths from s to a vertex v in G then $(\overline{x}, x)$ appears in all paths from $\overline{s}$ to v in $\overline{G}$. Let W be the set of vertices such that all paths from s to vertices in W contain x. Clearly there is no edge (w, z) such that $w \notin W, w \neq x$, $z \in W$, as that implies that there is a path from s to $z \in W$ avoiding x, which contradicts the fact that $z \in W$. Therefore, by the construction of $\overline{G}$ it follows that there is no edge $(w, \overline{z})$, such that $w \notin W, w \neq x$ and $z \in W$. Thus all paths from $\overline{s}$ to vertices in W contain $(\overline{x}, x)$. Thus, (i) and (ii) follow.

We finally prove (iii). Assume that two vertices u and v, $u \neq v \neq x$, are in the same SCC in $G \setminus x$ but not in the same SCC in $\overline{G} \setminus (\overline{x}, x)$. The fact that u and v are not strongly connected in $\overline{G} \setminus (\overline{x}, x)$ implies that either all paths from u to v contain $(\overline{x}, x)$ or all paths from v to u contain $(\overline{x}, x)$. Without loss of generality, assume that all paths from u to v in $\overline{G}$ contain the strong bridge $(\overline{x}, x)$. Since u and v are in the same SCC in $G \setminus x$, there is a path π from u to v in G that avoids all edges incoming to x and all edges outgoing to x. That means in $\overline{G} \setminus (\overline{x}, x)$ the corresponding path $\overline{\pi}$ of π avoids all incoming edges to $\overline{x}$ and all outgoing edges from x. This contradicts the fact that all paths from u to v in $\overline{G}$ contain $(\overline{x}, x)$. Now suppose that in a SCC in $\overline{G} \setminus (\overline{x}, x)$, there are two ordinary vertices u and v, $u \neq v \neq x$, that are in different SCC in $G \setminus x$. Therefore, there is a path $\overline{\pi}$ in $\overline{G}$ that avoids $(\overline{x}, x)$, and thus vertices $\overline{x}$ and x by construction. Then the corresponding path π of $\overline{\pi}$ in G avoids x, a contradiction. □

□

Now we combine Lemmas 7.1 and 7.2 to prove the following lemma.

Lemma 7.3. *Let G be a strongly connected graph with start vertex s. For each strong articulation point v it holds that:*

(i) The total number of SCCs in $G \setminus v$, is equal to the total number of SCCs in $\overline{G} \setminus (\overline{v}, v)$ minus 1.

(ii) The size of the largest SCC in $G \setminus v$, is equal to half of the size of the largest SCC in $\overline{G} \setminus (\overline{v}, v)$. The size of the smallest SCC in $G \setminus v$, equal to half of the size of the smallest SCC in $\overline{G} \setminus (\overline{v}, v)$, excluding the singleton SCCs $\overline{v}$ and v.

(iii) The SCCs of $G \setminus v$, are the sets of ordinary vertices of each SCC of $\overline{G} \setminus (\overline{v}, v)$ except of the singleton SCCs v and $\overline{v}$.

(iv) Two query vertices u and w are strongly connected in $G \setminus v$, if and only if u and w are strongly connected in $\overline{G} \setminus (\overline{v}, v)$.

(v) All vertices v such that u and w are not strongly connected in $G \setminus v$, are the ordinary endpoints of strong bridges e (excluding u and w) such that u and w are not strongly connected in $\overline{G} \setminus e$.

Proof. We start with (i). Notice that in $\overline{G} \setminus (\overline{x}, x)$, the vertices $\overline{x}$ and x are singleton SCCs since $(\overline{x}, x)$ is the only outgoing and the only incoming edge of $\overline{x}$ and x, respectively. First, notice that each ordinary vertex $v \neq x$ is in the same SCC together with $\overline{v}$ in $\overline{G}$ since $\overline{G}$ contains $(\overline{v}, v)$ and $(v, \overline{v})$. By Lemma 7.2(iii), all the vertices in a SCC in $G \setminus x$ are the ordinary vertices of a SCC in $\overline{G} \setminus (\overline{x}, x)$. Therefore, (i) follows.

The cases (ii), (iii), and (iv) follows immediately from Lemma 7.2(iii) and the fact that all ordinary vertices are in the same SCC with their auxiliary vertex.

Finally, we consider case (v). By Lemma 7.2(i) and Lemma 7.2(ii) it follows for each vertex x such that u and w are not strongly connected in $G \setminus x$, there exists the strong bridge $(\overline{x}, x)$ such that u and w are not strongly connected in $\overline{G} \setminus (\overline{x}, x)$. Let now $e = (p, \overline{q})$ be a strong bridge in $\overline{G}$ separating w and u, such that $p \neq w \neq u$. We show that $(\overline{p}, p)$ is also a strong bridge in $\overline{G}$ separating w and u. That implies, by Lemma 7.2(iii), p separates w and u in G. Without loss of generality, assume all paths from w to u contain $(p, \overline{q})$. By the fact that the only incoming edge to p is $(\overline{p}, p)$ it follows that all paths from w to u in $\overline{G}$ contain $(\overline{p}, p)$. Thus, $(\overline{p}, p)$ is a strong bridge in $\overline{G}$ separating w and u, and as we mentioned above that implies p is a separating vertex for w and u. Case (v) follows. □

□

Lemma 7.3 allows us to answer strong connectivity queries under vertex failures using the strong connectivity queries under edge failures from Section 6. Similarly to Section 6, we only need to maintain incrementally the dominator trees $\overline{D}$ and $\overline{D}^R$, and the hyperloop nesting trees $\overline{L}$ and $\overline{L}^R$ of $\overline{G}$ and $\overline{G}^R$, respectively. The most challenging query is to report the size of the smallest SCC in $G \setminus v$, for a query vertex v. This happens due to the fact that the size of the smallest SCCs in $\overline{G} \setminus (\overline{v}, v)$ is always one, namely, the singleton SCCs $\overline{v}$ and v. To resolve this problem we apply a minor modification of the algorithm in [20], in order to ignore the singleton SCC v when computing the minimum SCC in $G[D(v)]$, and the singleton SCC $\overline{v}$ when computing the minimum SCC in $G^R[D^R(v)]$. In summary, we have the following theorem.

Theorem 7.4. *We can maintain a digraph G throughout any sequence of edge insertions in a total of $O(mn)$ time, where m is the number of edges after all insertions, and after each edge insertion the following queries can be answered in asymptotically optimal (worst-case) time:*

(i) Report in $O(1)$ time the total number of SCCs in $G \setminus v$, for a query vertex v in G.

(ii) Report in $O(1)$ time the size of the largest and of the smallest SCCs in $G \setminus v$, for a query vertex v in G.

(iii) Report in $O(n)$ time all the SCCs of $G \setminus v$, for a query vertex v.

(iv) Test in $O(1)$ time if two query vertices u and w are strongly connected in $G \setminus v$, for a query vertex v.

(v) For query vertices u and w that are strongly connected in G, report all vertices v such that u and w are not strongly connected in $G \setminus v$, in $O(k+1)$ time, where k is the number of separating vertices.

8 Maintaining the 2-vertex-connected components of a digraph

In this section we present an algorithm for maintaining incrementally the 2-vertex-connected components of a digraph G in a total $O(mn)$ of time. In our algorithm we will use some properties from [20]. However, we need to extend those properties, by substituting them with strong connectivity queries under edge failures, in order to provide necessary and sufficient conditions for two vertices being 2-vertex-connected.

We compute the 2-vertex-connected components of a digraph by computing as an intermediary the following relation. Two vertices x and y are said to be *vertex-resilient* if the removal of any vertex different from x and y leaves x and y in the same strongly connected component. A *vertex-resilient component* of a digraph $G = (V, E)$ is defined as a maximal subset $B \subseteq V$ such that x and y are vertex-resilient for all $x, y \in B$. As a (degenerate) special case, a vertex-resilient component might consist of a singleton vertex only, in which case we have a *trivial vertex-resilient component.* We are interested in computing only the nontrivial vertex-resilient components, and since there is no danger of ambiguity, we will call them simply vertex-resilient components. The following lemma states that we can compute the 2-vertex-connected components starting from the 2-edge-connected components and the vertex-resilient components. Moreover, this computation can be done in $O(n)$ time [16].

Lemma 8.1. ([16]) *For any two distinct vertices x and y, x and y are 2-vertex-connected if and only if x and y are both vertex-resilient and 2-edge-connected.*

Because, of Lemma 8.1, an incremental algorithm for maintaining the 2-vertex-connected components of a digraph can be immediately obtained from incremental algorithms for maintaining the vertex-resilient and 2-edge-connected components. Since we know how to maintain incrementally the 2-edge-connected components of a digraph in a total of $O(mn)$ time [19], in the remainder of this section we will focus on the incremental maintenance of the vertex-resilient components.

Once again, let s be an arbitrary start vertex in G, and let D (resp., D^R) be the dominator tree of the flow graph G_s (resp., G_s^R). For any vertex u, we denote by $C(u)$ (resp. $C^R(u)$) the set of children of u in D (resp. D^R). For any pair of vertices u and v we identify a set of vertices $C(u, v)$ defined as follows. Set $C(u, v)$ contains all vertices in $C(u) \cap C^R(v)$. Also, if $u = v$ or $u \in C^R(v)$ then we include u in $C(u, v)$, and if $v \in C(u)$ then we include v in $C(u, v)$. These sets can be computed in $O(n)$ time [11].

Corollary 8.2 ([20])**.** *Let $G = (V, E)$ be a strongly connected digraph, and let $s \in V$ be an arbitrary start vertex. Any two vertices x and y are vertex-resilient only if they are located in a common set $C(u, v)$.*

Let x and y be two distinct vertices in G. We say that a strong articulation point u *separates* x *and* y (or equivalently that u is a *separating vertex for* x *and* y) if all paths from x to y or all paths from y to x contain vertex u (i.e., x and y belong to different strongly connected components of $G \setminus u$). Clearly, x and y are vertex-resilient if and only if there exists no separating vertex for them. The next two lemmas from [20] imply that only $O(1)$ vertices need to be tested in order to determine whether there exists a separating vertex for x and y.

Lemma 8.3 ([20])**.** *Let u be nontrivial dominator in either D or D^R that is a separating vertex for vertices x and y. Then u must appear in at least one of the paths $D[s,x]$, $D[s,y]$, $D^R[s,x]$, and $D^R[s,y]$. Let u be a strong articulation point that is a separating vertex for vertices x and y. Then u must appear in at least one of the paths $D[s,x]$, $D[s,y]$, $D^R[s,x]$, and $D^R[s,y]$.*

Lemma 8.4 ([20])**.** *Let x and y be vertices that are either siblings or x is the parent of y in D, and let $w = d(x)$. A strong articulation point u that is not a descendant of w in D is a separating vertex for x and y only if w is a separating vertex for x and y.*

The following corollary summarizes Lemmas 8.3 and 8.4.

Corollary 8.5. *Two vertices x and y are vertex-resilient if and only if there are some vertices u, v such that $x, y \in C(u,v)$, and x and y are strongly connected in:*

(a) $G \setminus u$

(b) *(if $u \neq s$) in* $G \setminus d(u)$

(c) $G \setminus v$

(d) *(if $v \neq s$) in* $G \setminus d^R(v)$

Note that, by Corollary 8.5, each vertex-resilient component is contained in a $C(u,v)$ set. Thus, the set $C(u,v)$ defines an initial set of "coarse" blocks that are supersets of the vertex-resilient components. To find the real vertex-resilient components, our algorithm will refine those coarse blocks with the help of the auxiliary components. The sets $C(u,v)$ can be represented by a block forest $\mathcal{F}$ of size $O(n)$ as shown in [20]. The block forest is a bipartite undirected acyclic graph that contains a node for each vertex $v \in V$ and a node for each vertex-resilient component of the graph; it contains an edge α, β if α is in the vertex-resilient component β. In order to refine the initial block forest, we will use the following operation from [16]. Let $\mathcal{B}$ be a set of blocks, let $\mathcal{S}$ be a partition of a set $U \subseteq V$, and let x be a vertex not in U.

refine$(\mathcal{B}, \mathcal{S}, x)$**:** For each block $B \in \mathcal{B}$, substitute B by the sets $B \cap (S \cup \{x\})$ of size at least two, for all $S \in \mathcal{S}$.

As shown in [16], this operation can be executed in time that is linear in the total number of elements in all sets of $\mathcal{B}$ and $\mathcal{S}$. The algorithm needs to locate the blocks that contain a specific vertex, and, conversely, the vertices that are contained in a specific block. Note that $\mathcal{F}$ is bipartite, so the adjacency list of a vertex v stores the blocks that contain v, and the adjacency list of a block

node B stores the vertices in B. Initially F contains one block for each $C(u, v)$ set. Those blocks are refined by means of *refine* operations. The executions of *refine* operations update the block forest F, while maintaining the linear execution time.

Our algorithm, called Inc2VCC, computes after every edge insertion the vertex-resilient components of the digraph in $O(n)$ time, using Corollary 8.5. The pseudocode is shown in Algorithm 5. We first prove the correctness of our algorithm. Next, we show that it runs in $O(n)$ time.

Lemma 8.6. *Algorithm Inc2VCC is correct.*

Proof. We will prove that two vertices x and y are in the same block of the maintained block forest $\mathcal{F}$ after the end of the algorithm if and only if they satisfy Corollary 8.5. Since two vertices are vertex-resilient if and only if they satisfy Corollary 8.5, it follows that the blocks in $\mathcal{F}$ are exactly the vertex-resilient components of the graph.

We first prove one direction of the claim: assume that x and y satisfy Corollary 8.5. The vertices x and y will not be separated in Line 4 since there exists a pair of vertices u, v such that $x, y \in C(u, v) \cup u$ by Corollary 8.5. Now we first assume that x and y are siblings in D. The vertices x, y will not be separated in Line 9 since they are strongly connected in $G \setminus u$ by Corollary 8.5 and $x, y \in C(u) \cup u$. Moreover, the removal carried out in Line 14 cannot remove any of x or y from their block. Now assume, without loss of generality, that $x = d(y) = u$. The vertices x, y will not be separated in Line 9 since $u, y \in C(u)$ and u will be included in the same block C as x by the definition of the *refine* operation. Notice that all vertices in C remain strongly connected in $G \setminus d(u)$ by Lemma 8.4. Therefore, u is strongly connected in $G \setminus d(u)$ with all vertices in C, since u and x are strongly connected by Corollary 8.5 (recall that we assumed x and y satisfy Corollary 8.5). The same arguments can be used to show that x and y are not separated in Lines 24 and 29. Hence, if two vertices satisfy Corollary 8.5, they are contained in the same block of the block forest after the end of the algorithm.

Now we prove the opposite direction of the claim. Namely, we assume that at the end of the algorithm x and y are in the same block of the block forest and we wish to prove that x and y satisfy Corollary 8.5. Since x and y are not separated in Line 4 there exists a pair of vertices u, v such that $x, y \in C(u, v)$. Now assume, without loss of generality, that x and y are siblings in D (both children of some vertex u). Then they are strongly connected in $G \setminus d(x)$ since they are not separated in Line 9. Now assume, without loss of generality, that $x = d(y) = u$. Recall that all vertices in the same block C as y in the block forest, right after the execution of Line 9, are strongly connected in $G \setminus x$. By Lemma 8.4, they are also strongly connected in $G \setminus d(x)$. Since x remains in the same block as y in Line 14, then x is strongly connected with all vertices in the component of y in $G \setminus d(x)$. Therefore, x and y satisfy conditions (a) and (b) in Corollary 8.5. The same arguments can be use to show that x and y satisfy (c) and (d) from Corollary 8.5, and the lemma follows. □

□

In order to collect efficiently the sets of vertices in $C(u)$ (resp., $C^R(u)$), for some vertex u, that are strongly connected in $G \setminus u$, as required in Line 8 (resp., Line 23) of Algorithm Inc2VCC, we use an additional data structure.

Algorithm 5: Inc2VCC

Input: Strongly connected digraph $G = (V, E)$
Output: The vertex-resilient components of G

```
Dominator trees and initialize block forest:
Let D and D^R be the dominator tree of G_s and G_s^R, respectively.
Compute the sets C(u, v).
Initialize the block forest F to contain one block for each set C(u, v) with
  at least two vertices.
Forward direction:
foreach u ∈ V in a bottom-up order of D do
    Find the set of blocks B that contain at least two vertices in C(u).
    Compute the collection of vertex subsets
      S = {S ∩ C(u) : S is an SCC in G \ u}.
    Execute refine(B, S, u).
    if u ≠ s then
        foreach B ∈ B such that u ∈ B do
            Choose an arbitrary vertex v ≠ u in B.
            if u and v are strongly connected in G \ d(u) then
                set B = B \ u
                if |B| = 1 then delete B from F
            end
        end
    end
end
Reverse direction:
foreach u ∈ V in a bottom-up order of D^R do
    Find the set of blocks B that contain at least two vertices in C^R(u).
    Compute the collection of vertex subsets
      S = {S ∩ C^R(u) : S is an SCC in G \ u}.
    Execute refine(B, S, u).
    if u ≠ s then
        foreach B ∈ B such that u ∈ B do
            Choose an arbitrary vertex v ≠ u in B.
            if u and v are strongly connected in G^R \ d^R(u) then
                set B = B \ u
                if |B| = 1 then delete B from F
            end
        end
    end
end
```

Recall the construction of the graph $\overline{G} = (\overline{V}, \overline{E})$ from Section 7. For each strong articulation point x of G, we add an auxiliary vertex $\overline{x} \in \overline{V}$ and add the auxiliary edges $(\overline{x}, x)$ and $(x, \overline{x})$. Moreover, we replace each edge (u, x) entering x in G with an edge $(u, \overline{x})$ entering $\overline{x}$ in $\overline{G}$. We maintain incrementally the flow graph $\overline{G}_s$, with start vertex s, its dominator tree $\overline{D}$, its bridge decomposition $\overline{\mathcal{D}}$, and the auxiliary components of $\overline{G}_s$. This will allow us to execute efficiently

Lines 8 and 23 of Algorithm Inc2VCC, as suggested by the following lemma.

Lemma 8.7. *Let x and y be two vertices that are siblings in D, and let $w = d(x) = d(y)$. Then, x and y are strongly connected in $G \setminus d(x)$ if and only if $r_{\overline{x}} = r_{\overline{y}}$ and $c_{\overline{x}} = c_{\overline{y}}$, where $r_{\overline{x}}$ and $r_{\overline{y}}$ are the roots of the bridge decomposition of $\overline{G}_s$ containing x and y, respectively, and $c_{\overline{x}}$ and $c_{\overline{y}}$ are the auxiliary components of $\overline{G}$ containing x and y, respectively.*

Proof. We begin with the forward direction of the claim: namely, we show that if x and y are strongly connected in $G \setminus d(x)$ then $r_{\overline{x}} = r_{\overline{y}}$ and $c_{\overline{x}} = c_{\overline{y}}$. Notice, that since $\overline{x}$ and $\overline{y}$ are auxiliary vertices, $\overline{d}(\overline{x})$ (resp., $\overline{d}(\overline{y})$) is an ordinary vertex and therefore $e' = (\overline{d}(\overline{d}(\overline{x})), \overline{d}(\overline{x}))$ (resp., $e' = (\overline{d}(\overline{d}(\overline{y})), \overline{d}(\overline{y}))$) is the only edge entering $\overline{d}(\overline{x})$ (resp., $\overline{d}(\overline{y})$) and moreover it is a bridge in G. Now assume that $r_{\overline{x}} \neq r_{\overline{y}}$. Then either $(\overline{d}(\overline{x}), \overline{x})$ or $(\overline{d}(\overline{y}), \overline{y})$ is a strong bridge (or both). Without loss of generality, assume that $(\overline{d}(\overline{x}), \overline{x})$ is a strong bridge and y is not a descendant of $\overline{x}$ in D. By definition of strong bridges, all paths from a vertex $v \notin \overline{D}(\overline{x})$ (which includes y) to $\overline{x}$ contain the strong bridge $(\overline{d}(\overline{x}), \overline{x})$ and therefore the vertex $\overline{d}(\overline{x})$. By Lemma 7.2(ii), $\overline{d}(\overline{x}) = d(x)$. The fact that all paths from y to $\overline{x}$ in $\overline{G}$ contain the vertex $\overline{d}(\overline{x})$ contradicts our initial assumption that x and y are strongly connected in $G \setminus d(x)$, as indicated by Lemma 7.2(ii). Hence, if x and y are strongly connected in $G \setminus d(x)$ then $r_{\overline{x}} = r_{\overline{y}}$. To complete the proof of this case, assume by contradiction that x and y are strongly connected in $G \setminus d(x)$ but $c_{\overline{x}} \neq c_{\overline{y}}$. Since neither $(\overline{d}(\overline{x}), \overline{x})$ nor $(\overline{d}(\overline{y}), \overline{y})$ are strong bridges and $e' = (\overline{d}(\overline{d}(\overline{x})), \overline{d}(\overline{x}))$ is a strong bridge in G, it follows that $r_{\overline{x}} = r_{\overline{x}} = \overline{d}(\overline{x})$. The fact that $c_{\overline{x}} \neq c_{\overline{y}}$ implies that $\overline{x}$ and $\overline{y}$ are not strongly connected in $\overline{G}[\overline{D}(\overline{d}(\overline{x}))]$, and therefore, they are not strongly connected in $\overline{G} \setminus e'$. By Lemma 7.2(iii), all vertices of a strongly connected component in $G \setminus d(x)$ are ordinary vertices of a strongly connected component in $\overline{G} \setminus e'$. This contradicts our assumption that x and y are strongly connected in $G \setminus d(x)$. Thus, if x and y are strongly connected in $G \setminus d(x)$ then it must be $r_{\overline{x}} = r_{\overline{y}}$ and $c_{\overline{x}} \neq c_{\overline{y}}$.

Now we prove the other direction of the claim: namely, if $r_{\overline{x}} = r_{\overline{y}}$ and $c_{\overline{x}} \neq c_{\overline{y}}$ then x and y are strongly connected in $G \setminus d(x)$. Since $r_{\overline{x}} = r_{\overline{y}}$, it follows that neither $(\overline{d}(\overline{x}), \overline{x})$ nor $(\overline{d}(\overline{y}), \overline{y})$ is a strong bridges Moreover, by the fact that $c_{\overline{x}} \neq c_{\overline{y}}$ if follows that $\overline{x}$ and $\overline{y}$ are strongly connected in $\overline{G} \setminus e'$. By Lemma 7.2, x and y must be strongly connected. □

□

We are now ready to analyze the total running time of Algorithm Inc2VCC.

Lemma 8.8. *Algorithm Inc2VCC runs in $O(n)$ time.*

Proof. The computation of all non-empty sets $C(u, v)$ in Line 4 takes $O(n)$ time as it is shown in [11]. Each vertex v is contained in at most 4 different initial blocks, i.e., the blocks $C(u, v), C(d(u), v), C(u, d(v)), C(d(u), d(v))$. This implies that the initial block forest contains $O(n)$ edges, and therefore its initialization in Line 3 takes $O(n)$ time. To find all subsets of vertices of $C(u)$ that are strongly connected in $G \setminus u$ in Line 8, we use Lemma 8.7, which implies that these blocks are the auxiliary components of $\overline{G}_s$ containing vertices $c_{\overline{x}}$, where $x \in C(u)$. Those auxiliary components are maintained by running an instance of the incremental algorithm in [19] on $\overline{G}$ in a total of time $O(mn)$, and they can be easily collected in time $O(|C(u)|)$. As a result, we spend a total of $O(mn)$

time to maintain the auxiliary components of $\overline{G}$, and $O(n)$ time for all queries throughout one execution of Algorithm Inc2VCC.

The initial block forest in Line 4 is a forest since it is defined with respect to the sets $C(u,v) = \{C(u) \cup u\} \cap \{C^R(v) \cup v\}$, where both sets $C(u) \cup u$ and $C^R(v) \cup v$ form trees. We now show that this is preserved by a $refine(\mathcal{B}, \mathcal{S}, u)$ operation. Consider what happens to a block $B \in \mathcal{B}$. This block is represented by a node b in $\mathcal{F}$. B is either partitioned into several disjoint blocks, in which case the tree containing b becomes a forest. Otherwise, B is replaced by sets $B_1, \ldots, B_l$ where all sets share u. In this case, the new adjacency lists of the nodes $b_1, \ldots, b_l$ representing the sets $B_1, \ldots, B_l$ share only the node corresponding to u. Therefore, the tree containing b remains a tree. Since $\mathcal{F}$ is a forest, the sum of the cardinalities of the blocks $\mathcal{B}$ from Line 9 (resp., Line 24) is at most $2|C(u)|$ (resp., $2|C^R(u)|$). To see this, just root each subtree of $\mathcal{F}$ that contains some of the blocks in $\mathcal{B}$ and their vertices: each node has one parent and $|\mathcal{B}| \leq |C(u)|$. Hence, the $refine$ operations in Lines 9 and 24 are executed in time $O(|C(u)|)$ and $O(|C^R(u)|)$, respectively. Thus, all refine operations take overall $O(n)$ time. Finally, notice that we can answer each query in Lines 11 and 26 in constant time by a type (iii) query from Section 7. This gives the lemma. □

□

Lemma 8.9. *We can maintain the vertex-resilient components of a directed graph G through any sequence of edge insertions in a total of $O(mn)$ time, where m is the number of edges after all edge insertions, and linear space.*

Theorem 8.10. *We can maintain the 2-vertex-connected components of a directed graph G through any sequence of edge insertions in a total of $O(mn)$ time, where m is the number of edges after all edge insertions, and linear space.*

Proof. The theorem follows from Lemmas 8.1 and 8.9 and from the fact that we can maintain incrementally the 2-edge-connected components of a directed graph in a total of $O(mn)$ time [19]. □

□

9 Conditional lower bounds

In this section we present conditional lower bounds for both the partially dynamic and the fully dynamic setting. In particular, one of our lower bounds implies that a polynomial improvement over our bounds would have interesting consequences, as such an improvement would disprove widely believed conjectures. More specifically, we show that there is no algorithm that can maintain either incrementally of decrementally a data structure allowing queries of the form "are u and v strongly connected in $G \setminus e$?", where $u, v \in V, e \in E$, and has total update time $O((mn)^{1-\epsilon})$ (for some constant $\epsilon > 0$) and sub-polynomial query time unless the online matrix-vector multiplication (OMv) conjecture [22] is false. Hence, under the OMv conjecture the total running time of our algorithm, for this particular query, is asymptotically optimal.

In the fully dynamic version we show that, unless the strong exponential time hypothesis (SETH) is false, there is no algorithm maintaining a graph and being able to answer any of the queries that we consider, within the same asymptotic

query time, with amortized update time $O(m^{1-\epsilon})$. Finally, we show that in the incremental/decremental model, there is no algorithm that can maintain a data structure answering any of the queries we consider in this paper, in the same asymptotic query time, with worst-case update time $O(m^{1-\epsilon})$, for any $\epsilon > 0$, unless the SETH is false. Therefore, in these two cases, recomputing the data structure from [20] from scratch after every update achieves the best possible asymptotic update time.

9.1 $\Omega(mn)$ conditional lower bound for the total update time in the partially dynamic model

In this section show a conditional lower bound for the total update time of a partially dynamic algorithm that either incrementally or decrementally maintains a data structure that can answer the queries "are u and v strongly connected in $G \setminus e$?", where $u, v \in V$, $e \in E$. Here, we show that there is neither incremental nor decremental algorithm for maintaining a data structure for answering the aforementioned that has total update time $O((mn)^{1-\epsilon})$ (for some constant $\epsilon > 0$) and sub-polynomial query time unless the OMv Conjecture [22] fails. This bound matches the total update time of our algorithms. For our reduction we use a modification of the construction that was used in [15] to prove conditional lower bounds for partially dynamic algorithm for updating the dominator tree. In what follows, we prove the following statement.

Theorem 9.1. *For any constant $\delta \in (0, 1/2]$ and any n and $m = \Theta(n^{1/(1-\delta)})$, there is no algorithm for maintaining a data structure under edge deletions/insertions allowing queries of the form "are u and v strongly connected in $G \setminus e$", where $u, v \in V$, $e \in E$, that uses polynomial preprocessing time, total update time $u(m, n) = (mn)^{1-\epsilon}$ and query time $q(m) = m^{\delta-\epsilon}$ for some constant $\epsilon > 0$, unless the OMv conjecture fails.*

Under this conditional lower bound, the running time of our algorithm is optimal up to sub-polynomial factors. We give the reduction for the incremental version of the problem. The hardness of the decremental problem follows via an analogous reduction.

Hardness assumption. As in [15], we consider the following γ-OuMv problem (for a fixed $\gamma > 0$) and parameters n_1, n_2, and n_3 such that $n_1 = \lfloor n_2^{\gamma} \rfloor$: We are first given a Boolean $n_1 \times n_2$ matrix M to preprocess. After the preprocessing, we are given a sequence of pairs of n_1-dimensional Boolean vectors $(u^{(1)}, v^{(1)}), \ldots, (u^{(n_3)}, v^{(n_3)})$ one by one. For each $1 \leq t \leq n_3$, we have to return the result of the Boolean vector-matrix-vector multiplication $(u^{(t)})^{\intercal} M v^{(t)}$ before we are allowed to see the next pair of vectors $(u^{(t+1)}, v^{(t+1)})$. It has been shown [22] that under the OMv Conjecture as stated above, there is no algorithm for this problem that has polynomial preprocessing time and for processing all vectors spends total time $n_1^{1-\epsilon_1} n_2^{1-\epsilon_2} n_3^{1-\epsilon_3}$ such that all ϵ_i are ≥ 0 and at least one ϵ_i is a constant > 0.

Reduction. We now give the reduction from the γ-OuMv problem with $\gamma = \delta/(1-\delta)$ to the incremental maintainance of a data structure that supports the queries "are u and v strongly connected in $G \setminus e$?", where $u, v \in V$, $e \in E$. In the following we denote by v_i the i-th entry of a vector v and by $M_{i,j}$ the entry at row i and column j of a matrix M.

Consider an instance of the γ-OuMv problem with parameters $n_1 = m^{1-\delta}$, $n_2 = m^{\delta}$, and $n_3 = m^{1-\delta}$. We preprocess the matrix M by constructing a graph $G^{(0)}$ with the set of vertices

$$V = \{s, x_0, x_1, \ldots, x_{n_3}, y_1, \ldots, y_{n_1}, z_1, \ldots, z_{n_2}\}$$

and the following edges:

- an edge (s, x_{n_3}), and, for every $1 \leq t \leq n_3$, an edge (x_t, x_{t-1}) (i.e., a path from s to x_0).
- for every $1 \leq j \leq n_2$, an edge (x_0, z_j).
- for every $1 \leq i \leq n_1$ and every $1 \leq j \leq n_2$, an edge (y_i, z_j) if and only if $M_{i,j} = 1$ (i.e. a bipartite graph between $\{y_1, \ldots, y_{n_1}\}$ and $\{z_1, \ldots, z_{n_2}\}$ encoding the entries of matrix M).
- an edge (z_j, s), for every $1 \leq j \leq n_2$ (this makes the graph strongly connected).

Whenever the algorithm is given the next pair of vectors $(u^{(t)}, v^{(t)})$, we first create a graph $G^{(t)}$ by performing the following edge insertions in $G^{(t-1)}$: If $t > 1$, we first insert from x_{t-1} an edge (x_{t-1}, y_i), for all $1 \leq i \leq n_1$. Then, for every i such that $u_i^{(t)} = 1$ we add the edge from x_t to y_i. Having created $G^{(t)}$, we now, for every j such that $v_j^{(t)} = 1$, check whether s and z_j are strongly connected in $G^{(t)} \setminus (x_t, x_{t-1})$. If this is the case for at least one j we return that $(u^{(t)})^\intercal M v^{(t)}$ is 1, otherwise we return 0.

Correctness. The correctness of our reduction follows from the following lemma.

Lemma 9.2. *For every $1 \leq t \leq n$, the j-th entry of $(u^{(t)})^\intercal M$ is 1 if and only if s and z_j are strongly connected in $G^{(t)} \setminus (x_t, x_{t-1})$.*

Proof. First, notice that there is always a path from z_j to s since there is the edge (z_j, s). If the j-th entry of $(u^{(t)})^\intercal M$ is 1, then there is an i such that $u_i^{(t)} = 1$ and $M_{i,j} = 1$. Thus, $G^{(t)}$ contains the edges (x_t, y_i) and (y_i, z_j) and consequently a cycle containing s and z_j, namely $\langle s, x_{n_3}, \ldots, x_t, y_i, z_j \rangle$. Therefore, s and z_j are strongly connected in $G^{(t)} \setminus (x_t, x_{t-1})$.

If the j-th entry of $(u^{(t)})^\intercal M$ is 0, then there is no i such that $u_i^{(t)} = 1$ and $M_{i,j} = 1$. This implies that there is no path (of length 2) from x_t to z_j avoiding (x_t, x_{t-1}) (via some vertex y_i). Moreover, notice that there is not edge (x_l, y_i) for $t \leq l \leq n_3$ and $1 \leq i \leq n_1$. Thus, every path from s to z_j contains (x_t, x_{t-1}). Thus, s cannot be the strongly connected with z_j in $G^{(t)} \setminus (x_t, x_{t-1})$. □

□

Note that $(u^{(t)})^\intercal M v^{(t)}$ is 1 if and only if there is a j such that both the j-th entry of $u^{(t)} M$ as well as the j-th entry of $v^{(t)}$ are 1. Furthermore, s and z_j are strongly connected in $G^{(t)} \setminus (x_t, x_{t-1})$ if and only if s and z_j are strongly connected in the maintained graph minus the edge (x_t, x_{t-1}). Therefore the lemma establishes the correctness of the reduction.

Complexity. The final graph $G^{(t)}$ has $n := \Theta(n_1 + n_2 + n_3) = \Theta(m^{\delta} + m^{1-\delta}) = \Theta(m^{1-\delta})$ vertices and $\Theta(n_1 n_2 + n_2 n_3) = \Theta(m)$ edges. The total number of

queries is $O(n_1 n_3) = m^{2(1-\delta)}$. Suppose the total update time of the incremental algorithms that supports the required queries is $O(u(m,n)) = (mn)^{1-\epsilon}$ and its query time is $O(q(m)) = m^{\delta-\epsilon}$. Using the reduction above, we can thus solve the γ-OuMv problem for the parameters n_1, n_2, n_3 with polynomial preprocessing time and total update time

$$O(u(m,n) + m^{2(1-\delta)}q(m)) = O(u(m, m^{1-\delta}) + m^{2(1-\delta)}q(m)) = O(m^{2-\delta-\epsilon})\,.$$

Since $n_1 n_2 n_3 = m^{2-\delta}$, this means we would get an algorithm for the γ-OuMv problem with polynomial preprocessing time and total update time $n_1^{1-\epsilon_1} n_2^{1-\epsilon_2} n_3^{1-\epsilon_3}$ where at least one ϵ_i is a constant > 0. This contradicts the OMv Conjecture.

9.2 $\Omega(m)$ conditional lower bound for the amortized update time in the fully dynamic setting

Now we prove a conditional lower bounds for the fully dynamic setting and for the partially dynamic setting with worst-case update time guarantees. More specifically, we show that for those two models, under the assumption of the strong exponential time hypothesis, the trivial algorithm that recomputes from scratch the solution using the static algorithm is asymptotically optimal up to sub-polynomial factors. We base our bounds on the following conjecture that was first stated in [27, 28].

Conjecture 9.3 (Strong Exponential Time Hypothesis (SETH))**.** *For every $\epsilon > 0$, there exists a k, such that SAT on k-CNF formulas on n variables cannot be solved in $O(2^{(1-\epsilon)n} poly(n))$ time.*

As an intermediate step in our reductions, we use the following result from [1].

Theorem 9.4 ([1])**.** *Let G be a digraph with n vertices that undergoes m edge updates from an initially empty graph (until the graph is empty, in the case of a decremental algorithm). If for some $\epsilon > 0$ and $t \in N$, there exists either*

- *a fully dynamic algorithm with preprocessing time $O(n^t)$, amortized update time $O(m^{1-\epsilon})$, and amortized query time $O(m^{1-\epsilon})$, or*
- *an incremental or decremental algorithm with preprocessing time $O(n^t)$, worst-case update time $O(m^{1-\epsilon})$, and worst-case query time $O(m^{1-\epsilon})$*

that can answer either the query "are there more that two or more SCCs in G?" or the query "what is the size of the largest SCC in G?", then Conjecture 9.3 is false.

Now we show the reduction from the problems of Lemma 9.4 to the problem of maintaining a data structure answering any of the queries that we consider in Theorem 7.4. We break our result into two parts. In the first part, we prove that the result for three of the totally five types of queries without any assumptions. In the second part, where we assume that the number of edges is in the graph is superlinear to the number of vertices. This assumption about the density of the graph is necessary as our reduction in this spends $O(n)$ additional time per query. We start with the first case where we make no assumptions about the density of the graph.

Lemma 9.5. *Let $G = (V, E)$ be a digraph with n vertices that undergoes m edge updates from an initially empty graph (until the graph is empty, in the case of a decremental algorithm), $\epsilon > 0$. Assume that there exists a dynamic algorithm that can answer any of the following queries, either incrementally, decrementally, or fully dynamically:*

(i) Report in $O(m^{1-\epsilon})$ time the total number of SCCs in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v) in G.

(ii) Report in $O(m^{1-\epsilon})$ time the size of the largest SCC in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v) in G.

(iii) Report in $O(m^{1-\epsilon})$ time all the SCCs of $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v).

Then, there exists a dynamic algorithm, in the same model (incremental, decremental, or fully dynamic), that can answer either the query "are there more that two SCCs in the graph?" or "what is the size of the largest SCC in the graph?" with the same preprocessing time, the same update time, and can answer queries in $O(m^{1-\epsilon})$ time.

Proof. It is safe to assume that $O(n) \in O(m)$ as if $m \leq n - 3$ we can answer immediately that G has more than two SCCs, as each SCC C has at least $|C|-1$ edges. We construct the following graph $G' = (V', E')$ from $G = (V, E)$. G' consists of the edges and vertices of G and additionally two vertices s_1, s_2, the edge (s_1, s_2), for each $v \in V$ an edge (v, s_1) and an edge (s_2, v). Notice that $|V'| = |V| + 2 \in O(|V|)$ and $|E'| = |E| + 2 \cdot n + 1 \in O(|E|)$. Our overall strategy for proving the lemma is to argue that at any time during the course of the dynamic algorithm we can answer the queries "are there more that two SCCs in G?" or "what is the size of the largest SCC in G?" by answering any one of the queries in the statement of the lemma. That is, we answer those queries independently of the type of updates (i.e., incremental, decremental, or fully dynamic algorithm), and of the type of query bound (i.e., either worst-case per update, or amortized over all updates). Specifically, we show that by answering the queries of the lemma withing the stated bounds, then we can answer the queries "are there more that two SCCs in G?" or "what is the size of the largest SCC in G?" in time $O(m^{1-\epsilon})$.

First, observe that two vertices u and v are strongly connected in $G' \setminus (s_1, s_2)$ or in $G' \setminus s_1$ if and only if they are strongly connected in G. This is true as any new path from u to v or from v to u, that uses any of the new edges that we inserted, contains vertex s_1 and edge (s_1, s_2). Therefore, by deleting either the vertex s_1 or the edge (s_1, s_2) we destroy all new paths from u to v and from v to u.

We begin with type (i) queries. Assume that we can report the number of SCCs in $G' \setminus (s_1, s_2)$ (resp., in $G' \setminus s_1$) in time $O(m^{1-\epsilon})$. Let this number be c. Note that the SCCs among vertices in V remain unchanged, and s_1 and s_2 form singleton SCCs in $G' \setminus (s_1, s_2)$ (resp., s_2 forms a singleton SCC in $G' \setminus s_1$). It follows that the number of SCCs in G is $c - 2$ (resp., $c - 1$). Hence we can answer whether G has more that two SCCs in time $O(m^{1-\epsilon})$. The same argument applies for type (ii) queries that report all SCCs in $G' \setminus (s_1, s_2)$ (resp., in $G' \setminus s_1$) in time $O(m^{1-\epsilon})$. The number of SCCs in G can be extracted in additional $O(n) \in O(m^{1-\epsilon})$ time (recall that we assume $O(n) \in O(m)$).

Now we consider type (iii) queries. Since the SCCs in $G' \setminus (s_1, s_2)$ (resp., in $G' \setminus s_1$) correspond to the SCCs of G plus the singleton SCCs s_1 and s_2 (resp., the singleton SCC s_2), the largest SCC in $G' \setminus (s_1, s_2)$ (resp., in $G' \setminus s_1$) has the same size with the largest SCC in G. Thus, if we can answer the size of the largest SCC in $G' \setminus (s_1, s_2)$ (resp., in $G' \setminus s_1$) in time $O(m^{1-\epsilon})$, then we can answer the size of the largest SCC in G in $O(m^{1-\epsilon})$. □

□

Theorem 9.6. *Let G be a digraph with n vertices that undergoes m edge updates from an initially empty graph (until the graph is empty, in the case of a decremental algorithm). If for some $\epsilon > 0$, there exists an algorithm that can answer the following queries:*

(i) Report in $O(m^{1-\epsilon})$ time the total number of SCCs in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v) in G.

(ii) Report in $O(m^{1-\epsilon})$ time the size of the largest SCC in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v) in G.

(iii) Report in $O(m^{1-\epsilon})$ time all the SCCs of $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v).

while maintaining G fully dynamically with amortized update time $O(m^{1-\epsilon})$ and amortized query time after polynomial time preprocessing, or partially dynamically with worst-case update time $O(m^{1-\epsilon})$ and worst-case query time after polynomial time preprocessing, then Conjecture 9.3 is false.

We now proceed to prove similar results of the last two query types, in graphs where the number of edges is superlinear to the number of vertices.

Lemma 9.7. *Let $G = (V, E)$ be a digraph with n vertices that undergoes $m > n^{1+\delta}$ edge updates from an initially empty graph (until the graph is empty, in the case of a decremental algorithm), $\delta > \epsilon/(1-\epsilon)$, $1/2 > \epsilon > 0$. Assume that there exists a dynamic algorithm that can answer any of the following queries, either incrementally, decrementally, or fully dynamically:*

(iv) Test in $O(m^{1-\epsilon}/n)$ time whether two query vertices u and v are strongly connected in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v).

(v) For any two query vertices u and v that are strongly connected in G, test whether there exists an edge e (resp., vertex v) such that u and v are not strongly connected in $G \setminus e$ (resp., $G \setminus v$) in time $O(m^{1-\epsilon}/n)$.

Then, there exists a dynamic algorithm, in the same model (incremental, decremental, or fully dynamic), that can answer either the query "are there more that two SCCs in the graph?" with the same preprocessing time, the same update time, and can answer queries in $O(m^{1-\epsilon})$ time.

Proof. We use the same construction as in the proof of Lemma 9.5, That is, we construct the following graph $G' = (V', E')$ from $G = (V, E)$. G' consists of the edges and vertices of G and additionally two vertices s_1, s_2, the edge (s_1, s_2), for each $v \in V$ an edge (v, s_1) and an edge (s_2, v). Notice that $|V'| = |V| + 2 \in O(|V|)$ and $|E'| = |E| + 2 \cdot n + 1 \in O(|E|)$. Our overall strategy for proving the lemma is to argue that at any time during the course of the dynamic

algorithm we can answer the queries "are there more that two SCCs in G?" by answering any one of the queries in the statement of the lemma. That is, we answer those queries independently of the type of updates (i.e., incremental, decremental, or fully dynamic algorithm), and of the type of query bound (i.e., either worst-case per update, or amortized over all updates). Specifically, we show that by answering the queries of the lemma withing the stated bounds, then we can answer the queries "are there more that two SCCs in G?" in time $O(m^{1-\epsilon}) \in \omega(n^{1+O(1)})$.

For type (v) queries we claim that for any two $u, v \in V$ there exists a separating edge (resp., vertex) in G' if and only in u and v are not strongly connected in G. We now prove this claim. First, assume that u and v are strongly connected in G. G' contain the paths $\langle u, s_1, s_2, v\rangle$ and $\langle v, s_1, s_2, u\rangle$, and those paths exists if we delete any edge from E. Similarly, if we delete any edge from $E' \setminus E$, the vertices u and v will remain strongly connected as they are strongly connected in (V', E). Second, assume that u and v are not strongly connected in G. Then, either there is no path from u to v or there is no path from v to u in G. Assume, w.l.o.g., that there is no path from u to v in G. Since all new path from u to v contain the edge (s_1, s_2), then (s_1, s_2) (resp., the vertex s_1 and the vertex s_2) is a separating edge (resp., vertex) for u and v. Our claim follows.

We follow a similar approach with type (iv) queries. Assume that we can test for any two query vertices u and v that are strongly connected in G, whether there exists an edge e (resp., vertex v) such that u and v are not strongly connected in $G \setminus e$ (resp., $G \setminus v$) in time $O(\max\{m^{1-\epsilon}/n, 1\})$. We pick an arbitrary vertex x of G and we test whether there exists a separating edge (resp., a separating vertex) for x and v, for every $v \in V \setminus x$. This requires $O(m^{1-\epsilon})$ time in total. Let X be the set of vertices for which there exists no separating edge (resp., vertex) with x. Then we pick an arbitrary vertex y from $V \setminus X$ and we test whether there exists a separating edge (resp., vertex) for y and v, for every $v \in V \setminus \{X \cup y\}$. This requires additional $O(m^{1-\epsilon})$ time. Let Y be the set of vertices for which there exists no separating edge (resp., vertex) with y. If $|X| + |Y| = |V|$, then G has two SCCs, namely the SCC formed by the vertices in X and the SCC formed by the vertices in Y. If, on the other hand $|X| + |Y| < |V|$, then there exists at least one vertex in V that is neither strongly connected to x nor strongly connected to y in G, and hence, G contains at least three SCCs. □

□

Theorem 9.8. *Let G be a digraph with n vertices that undergoes $m > n^{1+\delta}$ edge updates from an initially empty graph (until the graph is empty, in the case of a decremental algorithm), $\delta > \epsilon/(1-\epsilon)$. If for some $\epsilon > 0$ there exists an algorithm that can answer the following queries:*

(iv) Test in $O(\max\{m^{1-\epsilon}/n, 1\})$ time whether two query vertices u and v are strongly connected in $G \setminus e$ (resp., $G \setminus v$), for any query edge e (resp., vertex v).

(v) For any two query vertices u and v that are strongly connected in G, test whether there exists an edge e (resp., vertex v) such that u and v are not strongly connected in $G \setminus e$ (resp., $G \setminus v$) in time $O(\max\{m^{1-\epsilon}/n, 1\})$.

while maintaining G fully dynamically with amortized update time $O(m^{1-\epsilon})$ and amortized query time, or partially dynamically with worst-case update time $O(m^{1-\epsilon})$ and worst-case query time, after arbitrary polynomial time preprocessing, then Conjecture 9.3 is false.

References

[1] A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443, 2014.

[2] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *SODA*, pages 440–452.

[3] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.

[4] J. Aspnes, K. Chang, and A. Yampolskiy. Inoculation strategies for victims of viruses and the sum-of-squares partition problem. *Journal of Computer and System Sciences*, 72(6):1077–1093, 2006.

[5] M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms*, 12(2):14:1–14:22, 2015.

[6] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.

[7] R. Cohen, S. Havlin, and D. ben-Avraham. Efficient immunization strategies for computer networks and populations. *Physical Review Letters*, 91:247901, 2003.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[9] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[10] C. Demetrescu and G. F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008.

[11] W. Di Luigi, L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-connectivity in directed graphs: An experimental study. In *ALENEX*, pages 173–187, 2015.

[12] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, September 1997.

[13] P. G. Franciosa, G. Gambosi, and U. Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Information Processing Letters*, 61(2):113–120, 1997.

[14] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.

[15] L. Georgiadis, T. D. Hansen, G. F. Italiano, S. Krinninger, and N. Parotsidis. Decremental data structures for connectivity and dominators in directed graphs. In *ICALP*, pages 42:1–42:15, 2017.

[16] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *ICALP*, pages 605–616, 2015.

[17] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. *ACM Transactions on Algorithms*, 13(1):9:1–9:24, 2016.

[18] L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *ESA*, pages 491–502, 2012.

[19] L. Georgiadis, G. F. Italiano, and N. Parotsidis. Incremental 2-edge-connectivity in directed graphs. In *ICALP*, pages 49:1–49:15, 2016.

[20] L. Georgiadis, G. F. Italiano, and N. Parotsidis. Strong connectivity in directed graphs under failures, with applications. In *SODA*, pages 1880–1899, 2017.

[21] J. Gunawardena. A linear framework for time-scale separation in nonlinear biochemical systems. *PLoS ONE*, 7(5):e36321, 2012.

[22] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30, 2015.

[23] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *FOCS*, pages 664–672, 1995.

[24] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–536, 1999.

[25] M. R. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Comput.*, 31(2):364–374, February 2002.

[26] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.

[27] R. Impagliazzo and R. Paturi. On the complexity of k-sat. *Journal of Computer and System Sciences*, 62(2):367 – 375, 2001.

[28] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512 – 530, 2001.

[29] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012.

[30] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.

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

[32] C. J. Kuhlman, V. S. Anil Kumar, M. V. Marathe, S. S. Ravi, and D. J. Rosenkrantz. Finding critical nodes for inhibiting diffusion of complex contagions in social networks. In *ECML PKDD*, pages 111–127, 2010.

[33] Jakub Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Transactions on Algorithms*, 9(3):27, 2013.

[34] M. Mihalák, P. Uznański, and P. Yordanov. Prime factorization of the Kirchhoff polynomial: Compact enumeration of arborescences. In *ANALCO*, pages 93–105, 2016.

[35] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961, 2017.

[36] M. Patrascu and M. Thorup. Planning for fast connectivity updates. In *FOCS*, pages 263–271, 2007.

[37] N. Paudel, L. Georgiadis, and G. F. Italiano. Computing critical nodes in directed graphs. In *ALENEX*, pages 43–57, 2017.

[38] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *POPL*, pages 287–296, 1994.

[39] P. Sankowski. Dynamic transitive closure via dynamic matrix inverse: extended abstract. In *FOCS*, pages 509–517, 2004.

[40] Y. Shen, N. P. Nguyen, Y. Xuan, and M. T. Thai. On the discovery of critical links and nodes for assessing network vulnerability. *IEEE/ACM Transactions on Networking*, 21(3):963–973, June 2013.

[41] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–85, 1976.

[42] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *STOC*, pages 343–350, 2000.

[43] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *SWAT*, pages 384–396, 2004.

[44] M. Ventresca and D. Aleman. Efficiently identifying critical nodes in large complex networks. *Computational Social Networks*, 2(1):1–16, 2015.

Symbol	Description
G^R	The graph resulting from G after reversing the direction of all edges.
G_s (resp., G_s^R)	A flow graph G (resp., G^R), with start vertex s, where all vertices are reachable from s (resp., reach s).
$V(G)$ (resp., $E(G)$)	The set of vertices (resp., edges) of G.
$G \setminus v, v \in V$	G after deleting v together with and all its incident edges.
$G \setminus e, e \in E$	The graph resulting from G after deleting edge e.
$G[S], S \subseteq V$	The subgraph of G induced by the vertices in S.
$T(u)$, where T a tree and $u \in V(T)$	The set of vertices in the subtree rooted of T at u.
$T[u, v]$, where T a tree and $u, v \in V(T)$	The path between u and v in T (including u and v).
$nca_T(u, v)$, where T a tree and $u, v \in V(T)$	The nearest common ancestor of u and v in T.
D (resp., D^R) of G_s (resp., of G_s^R)	The dominator tree D (resp., D^R) of G_s (resp., G_s^R).
$d(u)$ (resp., $d^R(u)$)	The parent of u in D (resp., D^R).
$dom(u)$ (resp., $dom^R(u)$)	The set of ancestors of u in D (resp., D^R).
H (resp., H^R) of G_s (resp., of G_s^R)	The loop nesting tree H (resp., H^R) of G_s (resp., G_s^R)
$h(u)$ (resp., $h^R(u)$)	The parent of u in H (resp., H^R).
h_u (resp., h_u^R)	The nearest ancestor of u in the loop nesting tree H (resp., H^R) such that $h_u \in D(r_u)$ and $h(h_u) \notin D(r_u)$ (resp., $h_u^R \in D^R(r_u^R)$ and $h^R(h_u^r) \notin D^R(r_u^R)$).
Bridge decomposition $\mathcal{D}$ (resp., $\mathcal{D}^R$)	The resulting forest after deleting from D (resp., D^R) all bridges of G_s (resp., G_s^R).
r_u (resp., r_u^R)	The root of the tree in $\mathcal{D}$ (resp., $\mathcal{D}^R$) containing u.
c_u (resp., c_u^R)	The canonical vertex of the auxiliary component of u in G_s (resp., G_s^R).
L (resp., L^R) of G_s (resp., of G_s^R)	The hyperloop nesting tree L (resp., L^R) of G_s (resp., G_s^R).
$\ell(u)$ (resp., $\ell^R(u)$)	The parent of u in the hyperloop nesting tree L (resp., L^R).
$\hat{H}$ (resp., $\hat{H}^R$) of G_s (resp., of G_s^R)	The tree resulting from L (resp., $\hat{L}^R$) of G_s (resp., G_s^R) after making each vertex $u \neq c_u$ (resp., $u \neq c_u^R$) child of c_u (resp., c_u^R).
$\hat{h}(u)$ (resp., $\hat{h}^R(u)$)	The parent of u in $\hat{H}$ (resp., $\hat{H}^R$).
$level(u)$ (resp., $level^R(u)$)	The number of strong bridges on $D[s, u]$ (resp., $D^R[s, u]$) where s is the start vertex of the flow graph on which D (resp., D^R) is defined.
f', where f any relation	The relation f after the insertion of an edge (the inserted edge is usually specified, or we refer to any edge insertion).
D-scanned vertices S	The vertices that increase their depth in D after an edge insertion.
$G_{scanned}$	The graph induced by the vertices in S (the set of D-scanned vertices).
$\tilde{H}$	The loop nesting tree of $G_{scanned}$ rooted at y after the deletion of (x, y)
$\tilde{h}(u)$	the parent of u in $\tilde{H}$.
D-affected	The vertices that change parent in D after an edge insertion.
L-affected S'	The set of vertices S that change parent in L after an edge insertion.
$\widetilde{D}(u)$ (resp., $\widetilde{D}^R(u)$)	$D(u) \setminus u$ (resp., $D^R(u) \setminus u$).
$\overline{G} = (\overline{V}, \overline{E})$	For each strong articulation point x of G, we add an auxiliary vertex $\overline{x} \in \overline{V}$ and add the auxiliary edges $(\overline{x}, x)$ and $(x, \overline{x})$. Moreover, we replace each edge (u, x) entering x in G with an edge $(u, \overline{x})$ entering $\overline{x}$ in $\overline{G}$.
$\mathcal{F}$	The block forest, as defined in [16]. A data structure for representing overlapping sets of vertices.

Table 1: The notation that we use throughout the paper. We exclude notation that is used briefly for further definitions (e.g., $loop(u)$ for a vertex u, which is used to define the loop nesting tree of the graph).