Strong Connectivity in Directed Graphs under Failures, with Applications

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

Abstract

Let G be a directed graph (digraph) with m edges and n vertices, and let $G \setminus e$ (resp., $G \setminus v$) be the digraph obtained after deleting edge e (resp., vertex v) from G. We show how to compute in $O(m+n)$ worst-case time:

- The total number of strongly connected components in $G \setminus e$ (resp., $G \setminus v$), *for all edges* e (resp., *for all vertices* v) in G.
- The size of the largest and of the smallest strongly connected components in $G \setminus e$ (resp., $G \setminus v$), *for all edges* e (resp., *for all vertices* v) in G.

Let G be strongly connected. We say that edge e (resp., vertex v) separates two vertices x and y, if x and y are no longer strongly connected in $G \setminus e$ (resp., $G \setminus v$). We also show how to build in $O(m+n)$ time $O(n)$-space data structures that can answer in optimal time the following basic connectivity queries on digraphs:

- Report in $O(n)$ worst-case time all the strongly connected components of $G \setminus e$ (resp., $G \setminus v$), for a query edge e (resp., vertex v).
- Test whether an edge or a vertex separates two query vertices in $O(1)$ worst-case time.
- Report all edges (resp., vertices) that separate two query vertices in optimal worst-case time, i.e., in time $O(k)$, where k is the number of separating edges (resp., separating vertices). (For $k = 0$, the time is $O(1)$).

All our bounds are tight and are obtained with a common algorithmic framework, based on a novel compact representation of the decompositions induced by 1-edge and 1-vertex cuts in digraphs, which might be of independent interest. With the help of our data structures we can design efficient algorithms for several other connectivity problems on digraphs and we can also obtain in linear time a strongly connected spanning subgraph of G with $O(n)$ edges that maintains the 1-connectivity cuts of G and the decompositions induced by those cuts.

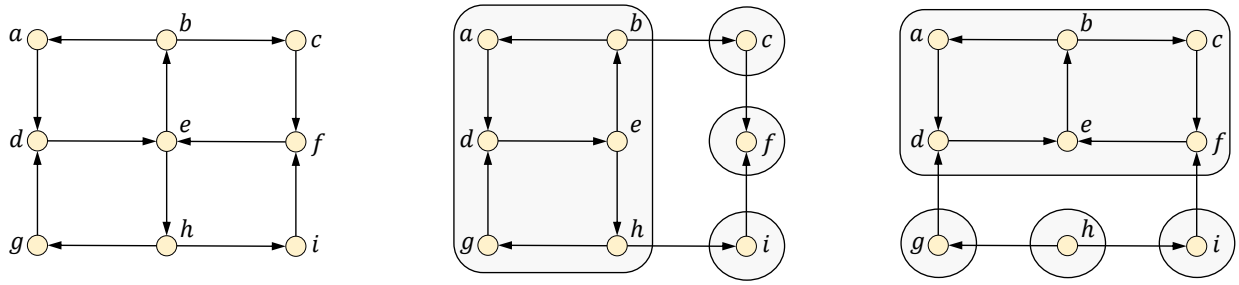
[1]University of Ioannina, Greece. E-mail: loukas@cs.uoi.gr.

[2]Università di Roma "Tor Vergata", Roma, Italy. E-mail: {giuseppe.italiano, nikos.parotsidis}@uniroma2.it. Partially supported by MIUR, the Italian Ministry of Education, University and Research, under Project AMANDA (Algorithmics for MAssive and Networked DAta).

1 Introduction

In this paper, we investigate some basic connectivity problems in directed graphs. Before defining precisely the problems considered, we need few definitions. Let $G = (V, E)$ be a directed graph (digraph), with m edges and n vertices. 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. The *size* of a SCC is given by its number of vertices. An edge (resp., a vertex) of G is a *strong bridge* (resp., a *strong articulation point*) if its removal increases the number of SCCs. Note that strong bridges (resp., strong articulation points) are 1-edge (resp., 1-vertex) cuts for digraphs. Let G be strongly connected. Two vertices $u, v \in V$ are said to be 2-*edge-connected* (resp., 2-*vertex-connected*), and we denote this relation by $u \leftrightarrow_{2e} v$ (resp., $u \leftrightarrow_{2v} v$), if there are two edge-disjoint (resp., two internally vertex-disjoint) directed paths from u to v and two edge-disjoint (resp., two 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- or vertex-disjoint). A 2-*edge-connected component* (resp., 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 $u, v \in B$. Let $G \setminus e$ (resp., $G \setminus v$) denote the digraph obtained after deleting edge e (resp., vertex v together with all its incident edges). We say that edge e (resp., vertex v) *separates* vertices x and y, if x and y are no longer strongly connected in $G \setminus e$ (resp., $G \setminus v$). In the literature the terms "components" and "blocks" have both been used to refer to k-connected components as defined above; here we explicitly use the term components in order to avoid further confusion.

Connectivity-related problems for digraphs are notoriously harder than for undirected graphs, and indeed many notions for undirected connectivity do not translate to the directed case. As shown in [6, 9, 12, 13, 17], in digraphs edge and vertex connectivity have a much richer and more complicated structure than in undirected graphs. For instance, an undirected graph is



(a) G (b) $G \setminus (f, e)$ (c) $G \setminus (e, h)$

Figure 1: An example illustrating the complicated structure of 1-edge cuts in digraphs. (a) A strongly connected digraph G. (b) The SCCs in $G \setminus (f, e)$. (c) The SCCs in $G \setminus (e, h)$. Note that a SCC in $G \setminus (f, e)$ and a SCC in $G \setminus (e, h)$ are neither disjoint nor nested. In fact, all edges are strong bridges, and the deletion of each edge creates many non-disjoint and non-nested sets in the resulting partitions.

naturally decomposed by bridges (resp., articulation points) into a tree of 2-edge- (resp., 2-vertex-) connected components, known as the bridge-block (resp., block) tree (see, e.g., [31]). In digraphs, the decomposition induced by strong bridges (resp., strong articulation points) becomes much more complicated (see Figure 1): in general, it was shown by Benczúr that in digraphs there can be no "cut" tree for various connectivity concepts [3]. For undirected graphs, it has been known for over 40 years how to compute the analogous notions (bridges, articulation points, 2-edge- and 2-vertex-connected components) in linear time, by simply using depth first search [26]. In the case of digraphs, however, the same problems revealed to be much more challenging: although these problems have been investigated for quite a long time (see, e.g., [9, 22, 24]), obtaining fast algorithms for digraphs has been an elusive goal for many years. Indeed, it has been shown only recently that all strong bridges, strong articulation points, and 2-edge- and 2-vertex-connected components of a digraph can be computed in linear time [12, 13, 18].

In this paper, we are interested in computing efficiently some properties of $G \setminus e$ and of $G \setminus v$, for all possible edges e and vertices v, such as their SCCs, the number of their SCCs, or the largest / smallest size of their SCCs. We are also interested in finding an edge or a vertex whose deletion minimizes / maximizes those properties. Those problems are not only theoretically interesting, but they also arise in a variety of application areas, including network analysis and computational biology. For instance, in several critical networked infrastructures, one is interested in identifying vertices and edges, whose removal results in a specific degradation of the network global pairwise connectivity [8]. In social networks, finding vertices whose deletion optimizes some connectivity properties is related to identifying key players whose removal fragments / disrupts the underlying network [4, 30]. Subsequently to this work, Paudel et al. [25] applied the framework presented here to devise an algorithm for identifying the critical nodes in digraphs. Applications in computational biology include the computation of steady states on digraphs governed by Laplacian dynamics, which include the symbolic derivation of kinetic equations and steady state expressions for biochemical systems [15, 23]. In particular, Mihalák et al. [23] presented recently a recursive deletion-contraction algorithm for such applications, whose efficient implementation needs to find repeatedly the edge of a strongly connected digraph whose deletion maximizes quantities such as the number of resulting SCCs or minimizes their maximum size.

Before stating our new bounds, we review some simple-minded solutions to the problems considered. A trivial approach to find an edge whose deletion minimizes / maximizes the number, or the largest or the smallest size of the resulting SCCs, would be to recompute the SCCs of $G \setminus e$, for each edge e in G, which requires $O(m^2)$ time in the worst case. This trivial bound can be easily improved by observing that if an edge e is not a strong bridge then, by definition, $G \setminus e$ remains strongly connected. Hence, we can first compute all strong bridges of G in $O(m + n)$ time [18] and then consider only the case where the edge to be deleted is a strong bridge, by recomputing the SCCs of $G \setminus e$ for each strong bridge e. Let b the total number of strong bridges in G: this yields a total time of $O(mb)$, which is $O(mn)$ in the worst case, since $b = O(n)$ (see Property 2.1). Similar bounds apply to the corresponding problems on vertices: we can find a vertex whose deletion minimizes / maximizes the number, or the largest or the smallest size of the resulting SCCs in $O(mp)$ time, where p is the total

number of strong articulation points in G. Since $p \leq n$, this can be $O(mn)$ in the worst case.

Our results. In this paper, we present new algorithms and data structures for computing in $O(m+n)$ worst-case time:

- The total number of SCCs in $G \setminus e$ (resp., $G \setminus v$), *for all edges* e (resp., *for all vertices* v) in G, thus improving the trivial bound of $O(mn)$. Our bound is asymptotically tight.
- The size of the largest and of the smallest SCCs in $G \setminus e$ (resp., $G \setminus v$), *for all edges* e (resp., *for all vertices* v) in G, thus improving the trivial bound of $O(mn)$. Our bound is again asymptotically tight.

Note that this gives immediately an algorithm for finding in linear time an edge (resp., a vertex) whose deletion minimizes / maximizes the total number or the largest / smallest size of the resulting SCCs in the resulting digraph, improving over the previous $O(mn)$ bounds. We can also build $O(n)$-space data structures that, after $O(m+n)$-time preprocessing, are able to answer in asymptotically optimal time the following basic 2-edge and 2-vertex connectivity queries on digraphs:

- Report in $O(n)$ worst-case time all the SCCs of $G \setminus e$ (resp., $G \setminus v$), for a query edge e (resp., vertex v), improving the trivial bound of $O(m+n)$. Note that those bounds are asymptotically tight, as one needs $O(n)$ time to output the SCCs of a digraph.
- Test whether an edge or a vertex separates two query vertices in $O(1)$ worst-case time, improving the trivial bound of $O(m)$.
- Report all the edges (resp., vertices) that separate two query vertices in optimal worst-case time, i.e., in time $O(k)$, where k is the number of separating edges (resp., separating vertices). (For $k = 0$, the time is $O(1)$). This improves the trivial bound of $O(mn)$.

With our approach, we can design efficient algorithms for several other connectivity problems on digraphs. After $O(m+n)$-time preprocessing, we can answer in $O(1)$ time for each edge e (resp., vertex v) queries such as the number of SCCs in $G \setminus e$ (resp., $G \setminus v$), or the maximum / minimum size of a SCC in $G \setminus e$ (resp., $G \setminus v$). We can further output all the SCCs in $G \setminus e$, for all edges e in G, in total $O(m+nb)$ worst-case time, and all the SCCs in $G \setminus v$, for all vertices v in G, in total $O(m+np)$ worst-case time, improving over previous $O(mb)$ and $O(mp)$ bounds. All those bounds are asymptotically tight. Note that $O(m+n)$, $O(m+nb)$ and $O(m+np)$ are all $O(n^2)$ in the worst case, while $O(mb)$ and $O(mp)$ are $O(n^3)$. Thus, our data structures are able to improve one order of magnitude over previously known bounds. Furthermore, our approach is able to provide alternative linear-time algorithms for computing the 2-edge-connected and 2-vertex-connected components of a digraph, which are much simpler than the algorithms presented in [12, 13], and thus are likely to be more amenable to practical implementations. Finally, we show how to obtain in linear time, a strongly connected spanning subgraph of G with $O(n)$ edges that maintains: (i) the 1-connectivity cuts of G (i.e., 1-edge cuts given by strong bridges, and 1-vertex cuts given by strong articulation points) and the decompositions induced by those cuts, and (ii) the 2-edge-connected and 2-vertex-connected components of G.

Related work. The problem of preprocessing a digraph G so that one can quickly answer queries under edge or vertex failures is not new. For instance, it has been investigated for reachability [19] and shortest paths [7]. Specifically, King and Sagert [19] showed how to process a directed acyclic graph G so that, for any pair of query vertices x and y, and a query edge e, one can test in constant time if there is a path from x to y in $G \setminus e$. Demetrescu et al. [7] considered the problem of preprocessing an edge-weighted digraph G to answer queries that ask for the shortest distance from any given vertex x to any other vertex y avoiding an arbitrary failed vertex or edge. They provide an oracle that answers such queries in constant time using $O(n^2 \log n)$ space. Our framework allows us to answer in asymptotically optimal time and space various queries related to the SCCs of a digraph G under an arbitrary edge or vertex failure. We can not only compute the SCCs that remain in G after the deletion of an edge or a vertex, but we can also report various statistics such as the number of SCCs in *constant time* per query (failed) edge or vertex. We can also extend our framework to compute a class of functions over the number of SCCs again in constant time per query edge or vertex. Our framework also supports constant-time path queries under failures, such as: "are there paths from x to y and from y to x after the deletion of a given failed edge or vertex"?

Key ideas. All our results are obtained with a common algorithmic framework, based on a novel combination of dominance relations [1, 21] and loop nesting forests [29]. We remark that the 1-connectivity cuts of a digraph can be found efficiently with the use of two dominator trees [18]. However, these dominator trees alone do not reveal enough information in order to determine the SCCs after the deletion of a single edge or vertex in these cuts. One of our key observations is that we can complement the dominance information with loop nesting information, and obtain a new compact representation of the

decompositions induced by the 1-connectivity cuts of digraphs, which consists simply of four trees: two dominator trees and two loop nesting trees. Still, combining these four trees in order to extract the decompositions induced by the 1-connectivity cuts turns out to be a non-trivial task. One of the main technical difficulties is to locate or count the vertices that are common in different subtrees of these four trees. To overcome this obstacle, we develop a novel technique, that takes advantage of relations between dominator and loop nesting trees. As a matter of fact, our techniques can be generalized so that we can compute several functions defined on the decompositions induced by the 1-connectivity cuts. We believe that the framework developed in this paper may be of independent interest and may prove to be useful for other problems as well. In particular, after this work, we have been able to apply it to the incremental maintenance of 2-edge connectivity on digraphs [14].

Due to lack of space, some details are omitted from this extended abstract. We refer the interested reader to the full version of the paper[1].

Organization of the paper. The remainder of the paper is organized as follows. In Section 2 we introduce some preliminary definitions and terminology. Section 3 describes our new framework for representing the structure of strong bridges and the decomposition they induce in a digraph. We use this framework to perform computations on the SCCs that could be obtained after an edge deletion, such as reporting all SCCs, counting the number of SCCs, and computing the size of the largest and of the smallest SCC. In Section 4 we deal with pairwise 2-edge connectivity queries. Section 5 states all the results for which the details are omitted from this extended abstract.

2 Preliminaries

We assume that the reader is familiar with standard graph terminology. All graphs in this paper are directed, i.e., an edge $e = (u, v)$ in digraph G is directed from u, the *tail* of e, to v, the *head* of e. We also assume that at this point the reader is familiar with the definitions of 2-edge and 2-vertex connectivity on digraphs already given in the introduction and contained in more detail in [12, 13].

Let T be a rooted tree. Throughout the paper, we assume that the edges of T are directed away from the root. For each directed edge (u, v) in T, we say that u is a parent of v (and we denote it by $t(v)$) and that v is a child of u. Every vertex except the root has a unique parent. If there is a (directed) path from vertex v to vertex w in T, we say that v is an ancestor of w and that w is a descendant of v, and we denote this path by $T[v, w]$. If $v \neq w$, we say that v is a proper ancestor of w and that w is a proper descendant of v, and denote by $T(v, w]$ the path in T to w from the child of v that is an ancestor of w. The ancestor/descendant relationship can be naturally extended to edges. Namely, let w be a vertex and (u, v) be an edge of T. If w is ancestor (resp., proper ancestor) of u (and thus also of v), we say that w is ancestor (resp., proper ancestor) of edge (u, v) and that (u, v) is descendant (resp., proper descendant) of w. Similarly, if w is descendant (resp., proper descendant) of v (and thus also of u), we say that w is descendant (resp., proper descendant) of edge (u, v) and that (u, v) is ancestor (resp., proper ancestor) of w. Let (u, v) and (w, z) be two edges in T. If v is ancestor of w, we say that (u, v) is ancestor of (w, z) and that (w, z) is descendant of (u, v). Also, for a rooted tree T, we let $T(v)$ denote the subtree of T rooted at v, and we also view $T(v)$ as the set of descendants of v.

Let T be a depth first search (dfs) tree of a digraph G, starting from a given vertex s. Edge (v, w) of the digraph G is a *tree edge* if $v = t(w)$, a *forward edge* if v is a proper ancestor of $t(w)$ in T, a *back edge* if v is a proper descendant of w in T, and a *cross edge* if v and w are unrelated in T. A *preorder* of T is a total order of the vertices of T such that, for every vertex v, the descendants of v are ordered consecutively, with v first. It can be obtained by a depth-first traversal of T, by ordering the vertices in the order they are first visited by the traversal. The following lemma is an immediate consequence of depth-first search.

LEMMA 2.1. (Path Lemma [26]) *Let T be a dfs tree of a digraph G, and let $pre(v)$ denote the preorder number of vertex v in T. If v and w are vertices such that $pre(v) < pre(w)$, then any path from v to w must contain a common ancestor of v and w in T.*

2.1 Flow graphs, dominators, and bridges.

A *flow graph* is a directed graph with a distinguished *start vertex* s such that every vertex is reachable from s. Let $G = (V, E)$ be a strongly connected graph. The *reverse digraph* of G, denoted by $G^R = (V, E^R)$, is the digraph that results from G by reversing the direction of all edges. Throughout the paper we let s be a fixed but arbitrary start vertex of G. Since G is strongly connected, all vertices are reachable from s and reach s, so we can view both G and G^R as flow graphs with start vertex s. To avoid ambiguities, we denote those flow graphs respectively by G_s and G_s^R.

Let G_s be a flow graph with start vertex s. A vertex u is a *dominator* of a vertex v (u *dominates* v) if every path from s to v in G_s contains u; u is a *proper dominator* of v if u dominates v and $u \neq v$. Let $dom(v)$ be the set of dominators of v. Clearly, $dom(s)$

[1] https://arxiv.org/abs/1511.02913

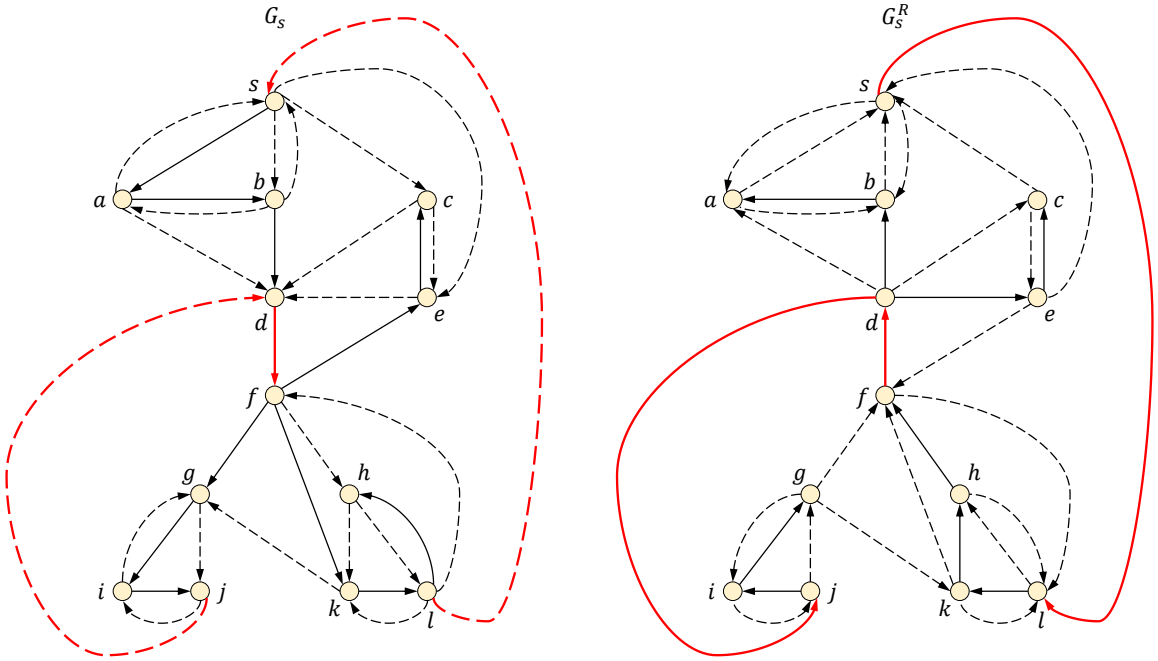


Figure 2: A flow graph G_s and its reverse G_s^R. The solid edges are the edges of depth first search trees with root s. The corresponding digraph G is strongly connected. Strong bridges of G and G^R are shown red.

$= \{s\}$ and for any $v \neq s$ we have that $\{s, v\} \subseteq dom(v)$: we say that s and v are the *trivial dominators* of v in the flow graph G_s. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree, the *dominator tree* D: u dominates v if and only if u is an ancestor of v in D. If $v \neq s$, the parent of v in D, denoted by $d(v)$, is the *immediate dominator* of v: it is the unique proper dominator of v that is dominated by all proper dominators of v. 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. We also denote the immediate dominator of v in G_s^R by $d^R(v)$. Throughout the paper, we let N (resp., N^R) denote the set of nontrivial dominators of G_s (resp., G_s^R). Lengauer and Tarjan [20] presented an algorithm for computing dominators in $O(m\alpha(m, n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [28]. Subsequently, several linear-time algorithms were discovered [2, 5, 10, 11].

An edge (u, v) is a *bridge* of a flow graph G_s if all paths from s to v include (u, v).[2] The following properties were proved in [18].

PROPERTY 2.1. ([18]) *Let s be an arbitrary start vertex of G. An edge $e = (u, v)$ is strong bridge of G if and only if it is a bridge of G_s (so $u = d(v)$) or a bridge of G_s^R (so $v = d^R(u)$) or both.*

PROPERTY 2.2. ([18]) *Let s be an arbitrary start vertex of G. A vertex $v \neq s$ is a strong articulation point of G if and only if v is a nontrivial dominator in G_s or a nontrivial dominator in G_s^R or both.*

As a consequence of Property 2.1, all the strong bridges of the digraph G can be obtained from the bridges of the flow graphs G_s and G_s^R, and thus there can be at most $(2n - 2)$ strong bridges in a digraph G. Figure 2 illustrates a strongly connected graph G and its reverse graph G^R, while Figure 3 contains the corresponding dominator trees D and D^R. Let $e = (u, v)$ be a strong bridge of G such that (v, u) is a bridge of G_s^R. Since there is no danger of ambiguity, with a little abuse of notation we will say that e is a bridge of G_s^R (although it is actually the reverse edge (v, u) that is a bridge of G_s^R). We refer to the edges that are bridges in both G_s and G_s^R as *common bridges*. We will use the following lemma from [13] that hold for a flow graph G_s of a strongly connected digraph G.

LEMMA 2.2. ([13]) *Let G be a strongly connected digraph and let (u, v) be a strong bridge of G. Also, let D and D^R be the dominator trees of the corresponding flow graphs G_s and G_s^R, respectively, for an arbitrary start vertex s.*

(a) Suppose $u = d(v)$. Let w be any vertex that is not a descendant of v in D. Then there is a path

[2] 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.

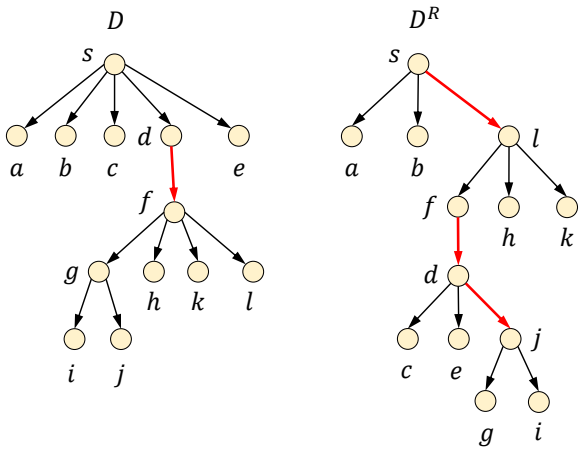


Figure 3: The dominator trees D and D^R of the flow graph in Figure 2. Bridges of G_s and G_s^R in D and D^R are shown red. The bridge decomposition of D and D^R is obtained after deleting the red edges.

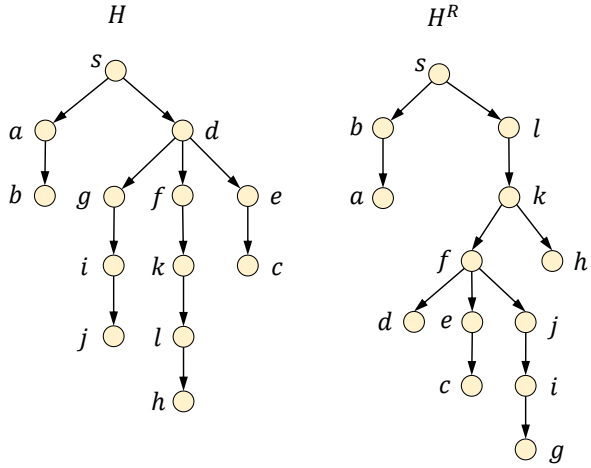


Figure 4: The loop nesting trees H and H^R of the flow graphs G_s and G_s^R of Figure 2 respectively, with respect to the dfs trees shown in Figure 2.

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 $(d(v), v)$.

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

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 $\mathcal{D}^R$. 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).

2.2 Loop nesting forests. Let $G = (V, E)$ be a digraph. A *loop nesting forest* represents a hierarchy of strongly connected subgraphs of G [29], and is defined with respect to a dfs tree T of G as follows. For any vertex u, the *loop* of u, denoted by $loop(u)$, 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. Vertex u is the *head* of $loop(u)$. 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 (i.e., one contains the other). The above property allows us to define the *loop nesting forest* H of G, with respect to T, as 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 will also denote as $H(u)$ (the subtree of H rooted at vertex u). Since T is a dfs tree, every cycle contains a back edge [26]. More generally, every cycle C contains a vertex u that is a common ancestor of all other vertices v of T in the cycle [26], which means that any $v \in C$ is in $loop(u)$. Hence, every cycle of G is contained in a loop. A loop nesting forest can be computed in linear time [5, 29]. Since here we deal with strongly connected digraphs, each vertex is contained in a loop, so H is a tree. Therefore, we will refer to H as the *loop nesting tree* of G. Figure 4 shows the loop nesting trees H and H^R of the flow graphs G_s and G_s^R given in Figure 2. The following lemma will be useful throughout the paper.

LEMMA 2.3. *Let x be any vertex in G. Let $h(x)$ be the parent of x in the loop nesting tree H and $r_{h(x)}$ be the root of the tree $D_{h(x)}$ in the bridge decomposition $\mathcal{D}$. Then $r_{h(x)}$ is an ancestor of x in D.*

Proof. Assume by contradiction that $r_{h(x)}$ is not an ancestor of x in D. Let T be the dfs tree based on which the loop nesting tree H was built. By the definition of dominators, all paths from s to $h(x)$ contain $r_{h(x)}$, and therefore $r_{h(x)}$ is an ancestor of $h(x)$ in T. By Lemma 2.2(a) all paths from x to $h(x)$ in G contain the strong bridge $(d(r_{h(x)}), r_{h(x)})$. But this is a contradiction to the fact that there is a path from x to $h(x)$ containing only descendants of $h(x)$ in T, since $r_{h(x)}$ is an ancestor of $h(x)$ in T. □

3 SCCs in $G \setminus e$

In this section, we describe our compact representation of the structure of all the 1-edge cuts (given by strong

bridges) of a strongly connected digraph G. Let G^R be the reverse digraph of G, and let s be any vertex in G. Let G_s be the flow graph with start vertex s and let G_s^R be the reverse flow graph with start vertex s. Let D and D^R be the dominator trees of G_s and G_s^R, and let H and H^R be the loop nesting trees of G_s and G_s^R generated from arbitrary dfs trees. We show that the four trees D, D^R, H and H^R are sufficient to encode efficiently the decompositions that the strong bridges induce in G, i.e., all the SCCs of $G \setminus e$, *for all strong bridges* e in G. In particular, let e be a strong bridge in G. We will show how the four trees D, D^R, H and H^R can be effectively exploited for solving efficiently the following problems:

- Compute all the SCCs of $G \setminus e$;
- Count the number of SCCs of $G \setminus e$;
- Find the size of the smallest/largest SCC of $G \setminus e$.

Throughout this section, we assume without loss of generality that the input digraph G is strongly connected (otherwise, we apply our algorithms to the SCCs of G). If G is strongly connected, then $m \geq n$, where m and n are respectively the number of edges and vertices in G, which will simplify some of the bounds. Since all our algorithms are based on dominator trees and loop nesting forests, we fix arbitrarily a start vertex s in G. We also restrict our attention to the strong bridges of G, since only the deletion of a strong bridge of G can affect its SCCs.

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. We will first prove some general properties of the SCCs of $G \setminus e$. We will then exploit those properties in Sections 3.1, 3.2 and 3.3 to design efficient solutions for our problems. Consider the dominator relations in the flow graphs G_s and G_s^R. By Property 2.1, one of the following cases must hold:

(a) e is a bridge in G_s but not in G_s^R, so $u = d(v)$ and $v \neq d^R(u)$.

(b) e is a bridge in G_s^R but not in G_s, so $u \neq d(v)$ and $v = d^R(u)$.

(c) e is a common bridge, i.e., a bridge both in G_s and in G_s^R, so $u = d(v)$ and $v = d^R(u)$.

We will show how to compute the SCCs of $G \setminus e$ in each of these cases. Consider $u = d(v)$, i.e., when either (a) or (c) holds. Case (b) is symmetric to (a). By Lemma 2.2, the deletion of the edge $e = (u, v)$ separates the descendants of v in D, denoted by $D(v)$, from $V \setminus D(v)$. Therefore, we can compute separately the SCCs of the subgraphs of $G \setminus e$ induced by the vertices in $D(v)$ and by the vertices in $V \setminus D(v)$. We begin with some lemmata that allow us to compute the SCCs in the subgraph of $G \setminus e$ that is induced by the vertices in $D(v)$. We recall here that, given the loop nesting tree H of G_s and a vertex w in H, we denote by $h(w)$ the parent of w in H and by $H(w)$ the set of descendants of w in H.

LEMMA 3.1. *Let $e = (u, v)$ be a strong bridge of G that is also a bridge in the flow graph G_s (i.e., $u = d(v)$). For any vertex $w \in D(v)$ the vertices in $H(w)$ are contained in a SCC C of $G \setminus e$ such that $C \subseteq D(v)$.*

Proof. Let $x \in D(v)$ be a vertex such that $w = h(x) \in D(v)$. We claim that w and x are strongly connected in $G \setminus e$. Let T be the dfs tree that generated H. Since $w = h(x)$ and $x, w \in D(v)$, the path π_1 from w to x in the dfs tree T avoids the edge $e = (u, v)$. To show that w and x are strongly connected in $G \setminus e$, we exhibit a path π_2 from x to w that avoids the edge e. Indeed, by the definition of the loop nesting forest, there is a path π_2 from x to w that contains only descendants of w in T. Note that π_2 cannot contain the edge e since $d(v)$ is a proper ancestor of v in T and all descendants of w in T are descendants of v in T, since all paths from s to w contain v. Assume by contradiction that either π_1 or π_2 contains a vertex $z \notin D(v)$. Lemma 2.2 implies that the either the subpath of π_1 from z to x or the subpath of π_2 from z to w contains e, a contradiction. This implies that every pair of vertices in $H(w)$ are strongly connected in $G \setminus e$. Let C be the SCC of $G \setminus e$ that contains $H(w)$. The same argument implies that all vertices in C are descendants of v in D. □

LEMMA 3.2. *Let $e = (u, v)$ be a strong bridge of G that is also a bridge in the flow graph G_s (i.e., $u = d(v)$). For every SCC C in $G \setminus e$ such that $C \subseteq D(v)$ there is a vertex $w \in C$ that is a common ancestor in H of all vertices in C.*

Proof. Let C be a SCC of $G \setminus e$ that contains only descendants of v in D. Let T be the dfs tree that generated H and let *pre* be the corresponding preorder numbering of the vertices. Define w to be the vertex in C with minimum preorder number with respect to T. Consider any vertex $z \in C \setminus w$. Since C is a SCC, there is a path π from w to z that contains only vertices in C. By the choice of w, $pre(w) < pre(z)$, so Lemma 2.1 implies that π contains a common ancestor q of w and z in T. Also $q \in C$, since π contains only vertices in C. But then $q = w$, since otherwise $pre(q) < pre(w)$ which contradicts the choice of w. Hence w is also an ancestor of z in H. □

The following lemma follows from Lemmata 3.1 and 3.2.

LEMMA 3.3. *Let $e = (u, v)$ be a strong bridge of G that is also a bridge in the flow graph G_s (i.e., $u = d(v)$). Let w be a vertex such that $w \in D(v)$ and $h(w) \notin D(v)$. Then, the subgraph induced by $H(w)$ is a SCC of $G \setminus e$.*

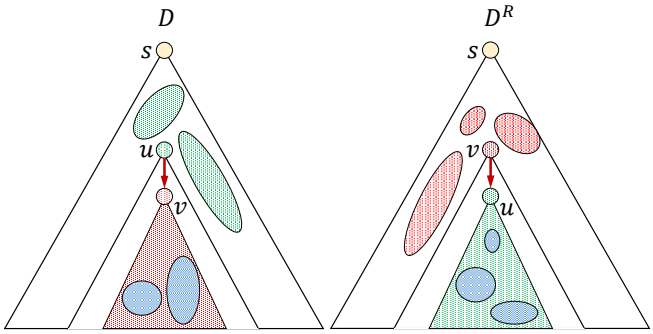


Figure 5: An illustration of the sets $D(v)$, $D^R(u)$, $D(v) \cap D^R(u)$, and $C = V \setminus \big(D(v) \cup D^R(u)\big)$ that correspond to a common strong bridge (u, v), as they appear with respect to the dominator trees D and D^R. The vertices in $D(v) \setminus D^R(u)$ are shown in red, the vertices in $D^R(u) \setminus D(v)$ are shown in green, and the vertices in $D(v) \cap D^R(u)$ are shown in blue. The remaining vertices are in $C = V \setminus \big(D(v) \cup D^R(u)\big)$.

Next we consider the SCCs of the subgraph of $G \setminus e$ induced by $V \setminus D(v)$.

LEMMA 3.4. *Let $e = (u, v)$ be a strong bridge of G that is a bridge in G_s but not in G_s^R (i.e., such that $u = d(v)$ and $v \neq d^R(u)$). Let $C = V \setminus D(v)$. Then, the subgraph induced by C is a SCC of $G \setminus e$.*

Proof. By Lemma 2.2(a), a vertex in C cannot be strongly connected in $G \setminus e$ to a vertex in $D(v)$. Thus, it remains to show that the vertices in C are strongly connected in $G \setminus e$. Note that by the definition of C we have that $s \in C$. Now it suffices to show that for any vertex $w \in C$, digraph G has a path π from s to w and a path π' from w to s containing only vertices in C. Assume by contradiction that all paths in G from s to w contain a vertex in $D(v)$. Then, Lemma 2.2 implies that all paths from s to w contain e, which contradicts the fact that $w \notin D(v)$. We use a similar argument for the paths from w to s. If all such paths contain a vertex in $D(v)$, then by Lemma 2.2 we have that all paths from w to s contain e. Hence, also all paths from u to s must contain e, which contradicts the fact that $v \neq d^R(u)$. □

Finally we deal with the more complicated case (c). We refer the reader to Figure 5 for an illustration of the sets involved in the lemma.

LEMMA 3.5. *Let $e = (u, v)$ be a strong bridge of G that is a common bridge of G_s and G_s^R (i.e., such that $u = d(v)$ and $v = d^R(u)$). Let $C = V \setminus \big(D(v) \cup D^R(u)\big)$. Then, the subgraph induced by C is a SCC of $G \setminus e$.*

Proof. The fact that e is a strong bridge and Lemma 2.2 imply that the following properties hold in G:

(1) There is a path from s to any vertex in $V \setminus D(v)$ that does not contain e.

(2) There is a path from any vertex in $V \setminus D^R(u)$ to s that does not contain e.

(3) There is no edge $(x, y) \neq e$ such that $x \notin D(v)$ and $y \in D(v)$. In particular, since $C \subseteq V \setminus D(v)$, there is no edge $(x, y) \neq e$ such that $x \in C$ and $y \in D(v)$.

(4) Symmetrically, there is no edge $(x, y) \neq e$ such that $x \in D^R(u)$ and $y \notin D^R(u)$. In particular, since $C \subseteq V \setminus D^R(u)$, there is no edge $(x, y) \neq e$ such that $x \in D^R(u)$ and $y \in C$.

Let K be a SCC of $G \setminus e$ such that $K \cap C \neq \emptyset$. By properties (3) and (4), K contains no vertex in $V \setminus C = D(v) \cup D^R(u)$. Thus $K \subseteq C$. Let G_C be the subgraph of G induced by the vertices in C. We will show that for any vertex $x \in K$ digraph G_C contains a path from s to x and a path from x to s. This implies that all vertices in C are strongly connected in G_C, and hence also in $G \setminus e$. Moreover, since $K \subseteq C$ is a SCC of $G \setminus e$, we have that $K = C$ and that it induces a SCC in $G \setminus e$.

First we argue about the existence of a path from s to $x \in C$ in G_C. Let π be a path in G from s to x that does not contain e. Property (1) guarantees that such a path exists. Also, Lemma 2.2 implies that π does not contain a vertex in $D(v)$, since otherwise π would include e. It remains to show that π also avoids $D^R(u)$. Assume by contradiction that π contains a vertex in $z \in D^R(u)$. Choose z to be the last such vertex in π. Since $x \notin D^R(u)$ we have that $z \neq x$. Let w be the successor of z in π. From the fact that π does not contain vertices in $D(v)$ and by the choice of z it follows that $w \in C$. But then, edge $(z, w) \neq e$ violates property (4), which is a contradiction. We conclude that path π also exists in G_C as claimed.

The argument for the existence of a path from $x \in C$ to s in G_C is symmetric. Let π be a path in G from x to s that does not contain e. From property (2) and Lemma 2.2 we have that such a path π exists and does not contain a vertex in $D^R(u)$. Now we show that π also avoids $D(v)$. Assume by contradiction that π contains a vertex in $z \in D(v)$. Choose z to be the first such vertex in π. Since $x \notin D(v)$ we have that $z \neq x$. Let w be the predecessor of z in π. From the fact that π does not contain vertices in $D^R(u)$ and by the choice of z it follows that $w \in C$. So, edge $(w, z) \neq e$ violates property (3). Hence path π also exists in G_C. □

LEMMA 3.6. *Let $e = (u, v)$ be a strong bridge of G that is a common bridge of G_s and G_s^R (i.e., such that $u = d(v)$ and $v = d^R(u)$). Let C be a SCC of $G \setminus e$ that contains a vertex in $D(v) \cap D^R(u)$. Then, $C \subseteq D(v) \cap D^R(u)$.*

Proof. Consider any two vertices x and y such that $x \in D(v) \cap D^R(u)$ and $y \notin D(v) \cap D^R(u)$. We claim that x and y are not strongly connected in $G \setminus e$, which implies the lemma. To prove the claim, note that by Lemma 3.5, x is not strongly connected in $G \setminus e$ with any vertex in $V \setminus (D(v) \cup D^R(u))$. Hence, we can assume that $y \in D(v) \setminus D^R(u)$ or $y \in D^R(u) \setminus D(v)$. In either case, x and y are not strongly connected in $G \setminus e$ by Lemma 2.2. □

The following theorem summarizes the results of Lemmata 3.3, 3.4, 3.5, and 3.6.

THEOREM 3.1. *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., $h^R(w) \notin D^R(u)$).

3.1 Finding all SCCs of $G \setminus e$. We now show how to exploit Theorem 3.1 in order to find efficiently the SCCs of $G \setminus e$, for each edge e in G. In particular, we present an $O(n)$-space data structure that, after $O(m)$-time preprocessing, given a strong bridge e of G can report in $O(n)$ time all the SCCs of $G \setminus e$. This is a sharp improvement over the naive solution, which computes from scratch the SCCs of $G \setminus e$ in $O(m)$ time. Furthermore, our bound is asymptotically tight, as one needs $O(n)$ time to output the SCCs of a digraph. With our data structure we can output in a total of $O(m+nb)$ worst-case time the SCCs of $G \setminus e$, for each edge e in G, where b is the total number of strong bridges in G.

We next describe our data structure. First, we process the dominator trees D and D^R in $O(n)$ time, so that we can test the ancestor/descendant relation in each tree in constant time [27]. To answer the query about a strong bridge $e = (u, v)$, we execute a preorder traversal of the loop nesting trees H and H^R. During those traversals, we will assign a label $scc(v)$ to each vertex v that specifies the SCC of v: that is, at the end of the preorder traversals of H and H^R, each SCC will consist of vertices with the same label. More precisely, if e is a bridge of G_s but not of G_s^R, then the preorder traversal of H will identify the SCCs of all vertices. Similarly, if e is a bridge of G_s^R but not of G_s, then we only execute a preorder traversal of H^R. Finally, if e is a common bridge, then the preorder traversal of H will identify the SCCs of all vertices except for those in $D^R(u) \setminus D(v)$ (i.e., the green vertices in Figure 5). The SCCs of these vertices will be discovered during the subsequent preorder traversal of H^R.

We do this as follows. We initialize $scc(v) = v$ for all vertices v. During our preorder traversals of H and H^R we update $scc(v)$ for all vertices $v \neq s$. Throughout, we always have $scc(s) = s$. Suppose $e = (u, v)$ is a bridge only in G_s. We do a preorder traversal of H, and when we visit a vertex $w \neq s$ we test if the condition $(w \in D(v) \wedge h(w) \notin D(v))$ holds. If it does, then the label of w remains $scc(w) = w$, otherwise the label of w is updated as $scc(w) = scc(h(w))$. We handle the case where e is a bridge only in G_s^R symmetrically. Suppose now that e is a common bridge. During the preorder traversal of H, when we visit a vertex $w \notin D^R(u) \setminus D(v)$, $w \neq s$, we test if the condition $(w \in D(v) \wedge h(w) \notin D(v))$ holds. As before, if this condition holds then the label of w remains $scc(w) = w$, otherwise we set $scc(w) = scc(h(w))$. Note that this process assigns $scc(x) = s$ to all vertices $x \in C = V \setminus \big(D(v) \cup D^R(u)\big)$. Also, all vertices $x \in H(w)$, where $w \in D(v)$ and $h(w) \notin D(v)$ are assigned $scc(x) = w$. Finally, we need to assign appropriate labels to the vertices in $D^R(u) \setminus D(v)$. We do that by executing a similar procedure on H^R. This time, when we visit a vertex $w \in D^R(u) \setminus D(v)$, $w \neq s$, we test if the condition $(w \in D^R(u) \wedge h^R(w) \notin D^R(u))$ holds. If it does, then the label of w remains $scc(w) = w$, otherwise we set $scc(w) = scc(h^R(w))$. At the end of this process we have that all vertices $x \in H^R(w)$, such that $w \in D^R(u)$ and $h^R(w) \notin D^R(u)$, are assigned $scc(x) = w$.

In every case, Theorem 3.1 implies that the above procedure assigns correct labels to all vertices. We remark that, during the execution of a query, each condition can be tested in constant time, since it involves computing the parent of a vertex in a loop nesting tree or checking the ancestor/descendant relationship in a dominator tree. This yields the following theorem.

THEOREM 3.2. *Let G be a strongly connected digraph with n vertices and m edges. We can preprocess G in $O(m)$ time and construct an $O(n)$-space data structure, so that given an edge e of G we can report in $O(n)$ time all the SCCs of $G \setminus e$.*

COROLLARY 3.1. *Let G be a strongly connected digraph with n vertices, m edges and b strong bridges. We can output the SCCs of $G \setminus e$, for all strong bridges e in G, in a total of $O(m + nb)$ worst-case time.*

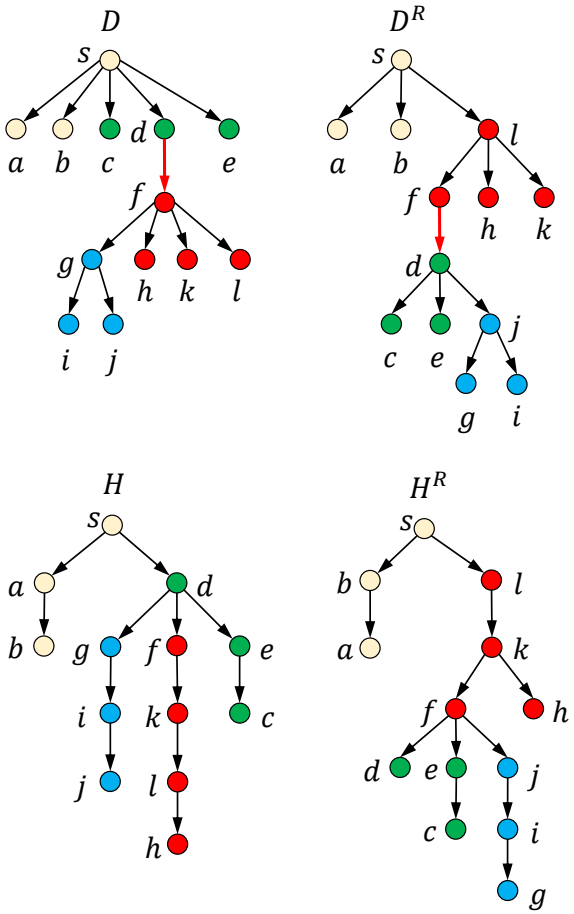


Figure 6: The dominator trees D and D^R and the loop nesting trees H and H^R of the flow graphs G_s and G_s^R of Figure 2, respectively. The edge (d, f) in red is a common bridge. The vertices in $D(f) \setminus D^R(d)$ are shown in red. The vertices in $D^R(d) \setminus D(f)$ are shown in green, and the vertices in $D(f) \cap D^R(d)$ are shown in blue. The remaining vertices are in $C = V \setminus \big(D(f) \cup D^R(d)\big)$.

Figure 6 highlights the vertices of different sets (with respect to the strong bridge (d, f)) in the dominator trees D and D^R and the loop nesting trees H and H^R of the flow graph G_s and G_s^R given in Figure 2. Figure 7 shows the result of a reporting query for the digraph of Figure 2 and for the edge (d, f).

3.2 Counting the number of SCCs of $G \setminus e$. In this section we consider the problem of computing the total number of SCCs obtained after the deletion of a single edge in a strongly connected digraph $G = (V, E)$. In particular, we describe a data structure which, after $O(m)$-time preprocessing, is able to answer the following aggregate query in worst-case time $O(n)$: "Find the total number of SCCs in $G \setminus e$, for all edges e." This provides a linear-time algorithm for computing the total

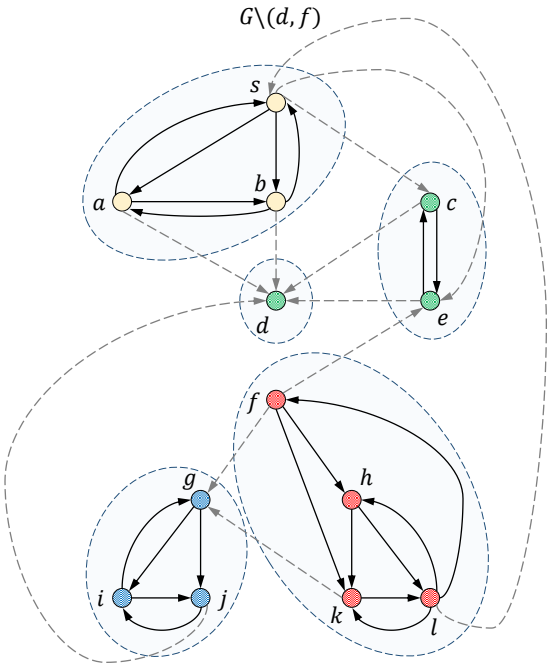


Figure 7: The SCCs of $G \setminus (d, f)$ where G is the graph of Figure 2.

number of SCCs obtained after the deletion of a single edge, for all edges in G. Note that we need to answer this query only for edges that are strong bridges in G; indeed if e is not a strong bridge, then $G \setminus e$ has exactly the same SCCs as G. Our bound is tight, and it improves sharply over the naive $O(mn)$ solution, which computes from scratch the SCCs of $G \setminus e$ for each strong bridge e of G.

Let $S \subseteq V$ be a subset of vertices of G. We denote by $\#SCC(S)$ the number of the SCCs in the subgraph of G induced by the vertices in S. Also, for an edge e of G, we denote by $\#SCC_e(V)$ the number of the SCCs in $G \setminus e$. Our goal is to compute $\#SCC_e(V)$ for every strong bridge e in G. Theorem 3.1 yields immediately the following corollary.

COROLLARY 3.2. *Let $e = (u, v)$ be a strong bridge of G. Then one of the following cases holds:*

(a) If e is a bridge in G_s but not in G_s^R then $\#SCC_e(V) = \#SCC(D(v)) + 1$.

(b) If e is a bridge in G_s^R but not in G_s then $\#SCC_e(V) = \#SCC(D^R(u)) + 1$.

(c) If e is a common bridge of G_s and G_s^R then $\#SCC_e(V) = \#SCC(D(v) \cup D^R(u)) + 1 = \#SCC(D(v)) + \#SCC(D^R(u)) - \#SCC(D(v) \cap D^R(u)) + 1$.

Moreover, let C be a SCC of $G \setminus e$. 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., $h^R(w) \notin D^R(u)$).

Corollary 3.2 shows that in order to compute the number of SCCs in $G \setminus e$ it is enough to compute $\#SCC(D(v))$, $\#SCC(D^R(u))$ and $\#SCC(D(v) \cap D^R(u))$. We will first show how to compute $\#SCC(D(v))$ and $\#SCC(D^R(u))$. Next, we will present an algorithm for computing $\#SCC(D(v) \cap D^R(u))$.

Computing $\#SCC(D(v))$ and $\#SCC(D^R(u))$. As suggested by Corollary 3.2, we can compute $\#SCC(D(v))$ (resp., $\#SCC(D^R(u))$) by counting the number of distinct vertices w in $D(v)$ (resp., in $D^R(u)$) for which $h(w) \notin D(v)$ (resp., $h^R(w) \notin D^R(u)$). We do this with the help of the bridge decomposition $\mathcal{D}$ (resp., $\mathcal{D}^R$) of D (resp., D^R) defined in Section 2.1. We recall that we denote by D_x (resp., D_x^R) the tree in $\mathcal{D}$ (resp., $\mathcal{D}^R$) containing vertex x, and by r_x (resp., r_x^R) the root of the tree D_x (resp., D_x^R).

To compute $\#SCC(D(v))$, we maintain a counter SCC_e for each bridge $e = (u, v)$ in G_s. SCC_e counts the number of SCCs in $G \setminus e$ that are subsets of $D(v)$ encountered so far. Rather than processing the bridges of G_s one at the time, we update simultaneously all counters SCC_e for all bridges e while visiting the bridge decomposition of the dominator tree D in a bottom-up fashion. To update the counters SCC_e, we exploit the loop nesting tree H: we increment the counter SCC_e of a bridge $e = (u, v)$ in G_s whenever we find a vertex w in $D(v)$ such that $h(w) \notin D(v)$, since $H(w) \subseteq D(v)$ is a SCC of $G \setminus e$ by Corollary 3.2. After the bridge decomposition of D has been processed, for each strong bridge $e = (u, v)$ in G we have that $SCC_e = \#SCC(D(v))$, i.e., the counter SCC_e stores exactly the number of SCCs of $G \setminus e$ containing only vertices in $D(v)$. In particular, by Corollary 3.2(a) we can compute $\#SCC_e(V) = SCC_e + 1$ for each strong bridge $e = (u, v)$ which is a bridge in G_s but not in G_s^R.

We can compute $\#SCC(D^R(u))$ in a completely similar way by considering the flow graph G_s^R, its dominator tree D^R, and its loop nesting tree H^R, and by swapping the roles of u and v.

Note that if $e = (u, v)$ is a common bridge of G_s and G_s^R, then by Corollary 3.2(c) we have that $\#SCC_e(V) = SCC_e + SCC_e^R - \#SCC(D(v) \cap D^R(u)) + 1$. Thus, to complete the description of our algorithm we have still to show how to compute $\#SCC(D(v) \cap D^R(u))$, i.e., the number of SCCs in the subgraph induced by the vertices in $D(v) \cap D^R(u)$. We will deal with this issue later.

As previously mentioned, a crucial task for updating the counter SCC_e (resp., SCC_e^R) for each strong bridge $e = (u, v)$ is to check for a vertex w in $D(v)$ (resp., in $D^R(u)$) such that $h(w) \notin D(v)$ (resp., $h^R(w) \notin D^R(u)$). An efficient method to perform this test hinges on the following lemma.

LEMMA 3.7. *For each vertex w in G, the following holds:*

(a) $H(w)$ (resp., $H^R(w)$) induces a SCC of $G \setminus e$, for any bridge e of G_s (resp., G_s^R) in the path $D[r_{h(w)}, w]$ (resp., $D^R[r^R_{h^R(w)}, w]$) from $r_{h(w)}$ to w in D (resp., from $r^R_{h^R(w)}$ to w in D^R).

(b) $H(w)$ (resp., $H^R(w)$) does not induce a SCC of $G \setminus e$, for any bridge e of G_s (resp., G_s^R) in the path $D[s, r_{h(w)}]$ (resp., $D^R[s, r^R_{h^R(w)}]$) from s to $r_{h(w)}$ in D (resp., from s to $r^R_{h^R(w)}$ in D^R).

Proof. We only prove the lemma for bridges of G_s, as the case for bridges of G_s^R is completely analogous. Let $e = (u, v)$ be any bridge of G_s in the path $D[r_{h(w)}, w]$: since $w \in D(v)$ and $h(w) \notin D(v)$, (a) follows immediately from Lemma 3.3. Now we turn to (b). Let $e = (u, v)$ be any bridge of G_s in the path $D[s, r_{h(w)}]$, and let z be the nearest ancestor of w in the loop nesting tree H such that $z \in D(v)$ and $h(z) \notin D(v)$. Note that $z \neq w$ since $h(w)$ is a descendant of v in D. Hence z is a proper ancestor of w in H. By Theorem 3.1, $H(z)$ induces a SCC of $G \setminus e$. Since z is a proper ancestor of w in H, then $H(w) \subset H(z)$. As a consequence, $H(w)$ does not induce a maximal strongly connected subgraph in $G \setminus e$ and thus it cannot induce a SCC of $G \setminus e$. □

We are now ready to describe our algorithm. We do not maintain the counters SCC_e and SCC_e^R for each strong bridge e explicitly. Instead, for the sake of efficiency, we distribute this information along some suitably chosen vertices in the dominator trees D and D^R. We first compute a compressed tree $\widehat{D}$ of the dominator tree D as follows. Let x be any vertex of G, and let D_x be the tree cointaining x in the bridge decomposition $\mathcal{D}$: then $\widehat{D}$ is obtained by contracting all the vertices of D_x into its tree root r_x. Let x be a vertex in the dominator tree D such that $h(x) \notin D(r_x)$: then by Lemma 2.3 $r_{h(x)}$ is an ancestor of x in D and by Lemma 3.7 we need to increment the counter of all bridges that lie in the path $D[r_{h(x)}, x]$ from $r_{h(x)}$ to x in D. By construction, those bridges correspond exactly to all the edges in the path $\widehat{D}[r_{h(x)}, r_x]$ from $r_{h(x)}$ to r_x in the compressed tree $\widehat{D}$. We refer to such a path $\widehat{D}[r_{h(x)}, r_x]$ as a *bundle starting from vertex $r_{h(x)}$ and ending at vertex r_x*. For each bridge e in the bundle starting from $r_{h(x)}$ and ending at r_x, by Lemma 3.7(a) there is a SCC $H(x)$ in $G \setminus e$, where x is in the tree D_x rooted at r_x of the bridge decomposition $\mathcal{D}$.

Algorithm 1: SCCsDescendants

Input: Strongly connected digraph $G = (V, E)$
Output: For each strong bridge (u, v) the numbers $\#SCC(D(v))$ and $\#SCC(D^R(u))$

Select an arbitrary start vertex $s \in V$.
Compute the dominator trees D and D^R, the loop nesting trees H and H^R, and the bridge decompositions $\mathcal{D}$ and $\mathcal{D}^R$.
Compute the compressed trees $\widehat{D}$ and $\widehat{D}^R$ of the dominator trees D and D^R, respectively. Initialize $bundle(x) = 0$ for each $x \in \widehat{D}$ and $bundle^R(x) = 0$ for each $x \in \widehat{D}^R(x)$.
foreach *vertex* $x \in V$ **do**
  if $h(x) \notin D(r_x)$ **then**
    Find the roots r_x and $r_{h(x)}$ in $\mathcal{D}$
    Set $bundle(r_x) = bundle(r_x) + 1$ and $bundle(r_{h(x)}) = bundle(r_{h(x)}) - 1$
  end
  if $h^R(x) \notin D^R(r_x^R)$ **then**
    Find the root r_x^R and $r^R_{h^R(x)}$ in $\mathcal{D}^R$
    Set $bundle^R(r_x^R) = bundle^R(r_x^R) + 1$ and $bundle^R(r^R_{h^R(x)}) = bundle^R(r^R_{h^R(x)}) - 1$
  end
end
foreach *vertex* $z \in \widehat{D}$, $z \neq s$, *in a bottom-up fashion* **do**
  Set $\#SCC(D(z)) = \#SCC(D(z)) + bundle(z)$
  Set $\#SCC(D(r_{d(z)})) = \#SCC(D(r_{d(z)})) + \#SCC(D(z))$
end
foreach *vertex* $z \in \widehat{D}^R$, $z \neq s$, *in a bottom-up fashion* **do**
  Set $\#SCC(D^R(z)) = \#SCC(D^R(z)) + bundle^R(z)$
  Set $\#SCC(D^R(r^R_{d^R(z)})) = \#SCC(D^R(r^R_{d^R(z)})) + \#SCC(D^R(z))$
end

For each vertex $z \in \widehat{D}$ we maintain a value $bundle(z)$ which stores the number of bundles that end at z, minus the number of bundles that start at z. For each vertex $z \in \widehat{D}$, $\sum_{y \in \widehat{D}(z)} bundle(y)$ equals the total number of bundles that start from a proper ancestor of z in $\widehat{D}$ and end at a descendant of z in $\widehat{D}$. For sake of space, the proof of the following lemma is omitted.

LEMMA 3.8. *Let (u, v) be a bridge of G_s. Then (u, v) corresponds to the edge (r_u, v) in $\widehat{D}$ and $\#SCC(D(v)) = \sum_{y \in \widehat{D}(v)} bundle(y)$.*

We next show how to compute $bundle(z)$ for each vertex $z \in \widehat{D}$. We process all vertices in G in any order. Whenever we find a vertex x in the digraph G such that $h(x) \notin D(r_x)$, we increment $bundle(r_x)$ and decrement $bundle(r_{h(x)})$. Indeed, by Lemma 3.7(a) there is a bundle $\widehat{D}[r_{h(x)}, r_x]$ starting from $r_{h(x)}$ and ending at r_x: for each bridge e in $\widehat{D}[r_{h(x)}, r_x]$, $H(x)$ induces a SCC of $G \setminus e$.

Once $bundle(z)$ is computed for each vertex z in $\widehat{D}$, we can compute for each bridge (u, v) in G_s the value $\#SCC(D(v))$. Recall that each vertex $z \in \widehat{D}$ is the root of a tree in the bridge decomposition $\mathcal{D}$ and $(d(z), z)$ is a bridge in G_s. From Lemma 3.8, we have that $\#SCC(D(v)) = bundle(v) + \sum_y \#SCC(D(y))$, where the sum is taken for all children y of v in $\widehat{D}$. So we can compute the $\#SCC(D(v))$ values by visiting the compressed tree $\widehat{D}$ in a bottom-up fashion as follows. For each vertex z visited in $\widehat{D}$, we set $\#SCC(D(z)) = \#SCC(D(z)) + bundle(z)$, and we increment $\#SCC(D(r_{d(z)}))$ by the value $\#SCC(D(z))$.

We can compute $\#SCC(D^R(u))$ for each bridge (u, v) in G_s^R in a completely analogous fashion.

LEMMA 3.9. *Given the dominator trees D and D^R, the loop nesting trees H and H^R, and the strong bridges of G, Algorithm SCCsDescendants runs in $O(n)$ time.*

Proof. Each dominator tree has $n - 1$ edges and therefore we construct the bridge decomposition of D and D^R and the compressed trees $\widehat{D}$ and $\widehat{D}^R$ in $O(n)$ time. The lemma follows from the fact that in each loop the algorithm performs $O(1)$ computations. □

Computing $\#SCC(D(v) \cap D^R(u))$. To complete the description of our algorithm, we still need to describe how to compute the quantity $\#SCC(D(v) \cap D^R(u))$ for each common bridge $e = (u, v)$ of G_s and G_s^R. Before doing that, we need to introduce some new terminology. Let $e = (u, v)$ be a common bridge. We say that a vertex w is a *common descendant* of (u, v) if $w \in D(v)$ and $w \in D^R(u)$; in this case we also say that (u, v) is a *common bridge ancestor* of w. Previously, we have been working with the bridge decomposition $\mathcal{D}$ and $\mathcal{D}^R$ of the dominator trees D and D^R, as defined in Section 2.1. Since we need to deal now with the common bridges of G_s and G_s^R, we define a coarser partition of D and D^R, as follows. After deleting all the common bridges from the dominator trees D and D^R, we obtain the *common bridge decomposition* of D and D^R into forests $\breve{\mathcal{D}}$ and $\breve{\mathcal{D}}^R$ (see Figure 8). We denote by $\breve{D}_u$ (resp., $\breve{D}_u^R$) the tree in $\breve{\mathcal{D}}$ (resp., $\breve{\mathcal{D}}^R$) that contains vertex u, and by $\breve{r}_u$ (resp., $\breve{r}_u^R$) the root of $\breve{D}_u$ (resp., $\breve{D}_u^R$). Lemma 3.10 extends Lemma 2.3.

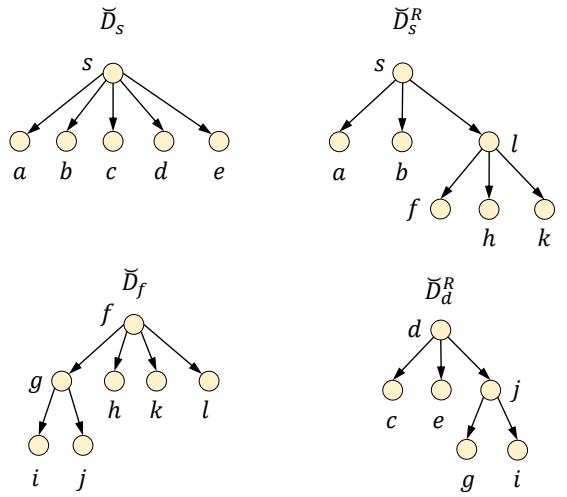


Figure 8: The common bridge decomposition of the dominator trees D and D^R of Figure 2. In this example the only common bridge is edge (d, f).

LEMMA 3.10. *Let x be a vertex in G. Then $\breve{r}_{h(x)}$ is an ancestor of x in D.*

Proof. Omitted, similar to the proof of Lemma 2.3. □

Let e and e' be two common bridges. If e is an ancestor of e' in D (resp., D^R) then we use the notation $e \overset{D}{\to} e'$ (resp., $e \overset{D^R}{\to} e'$) to denote this fact.

LEMMA 3.11. *Let e, e', and e'' be distinct common bridges such that $e \overset{D}{\to} e'' \overset{D}{\to} e'$ and $e' \overset{D^R}{\to} e$. Then $e' \overset{D^R}{\to} e'' \overset{D^R}{\to} e$.*

Proof. Let $e = (u, v)$ and $e' = (w, z)$. The fact that $e \overset{D}{\to} e'' \overset{D}{\to} e'$ implies that all paths from u to z in G contain e, e'' and e' in that order. If e'' is not in the path from z to u in D^R then there is path from u to z in G that avoids e'', a contradiction. □

We will use Lemma 3.11 to identify the common bridge ancestors of each vertex from a sequence of common strong bridges $e_1 \overset{D}{\to} e_2 \overset{D}{\to} \ldots \overset{D}{\to} e_\ell$. In order to have a compact representation of the relations in Lemma 3.11 we define the *common bridge forest* $\mathcal{Q}$ as follows (see Figure 9). Forest $\mathcal{Q}$ contains a node $\varphi(e)$ for each common bridge e. We also define the reverse map φ^{-1} from the nodes of $\mathcal{Q}$ to the common bridges of G, i.e., for any node α of $\mathcal{Q}$, $\varphi^{-1}(\alpha)$ is the corresponding common bridge represented by α. Let α and β be two distinct nodes of $\mathcal{Q}$. Let $\varphi^{-1}(\alpha) = (x, y)$ and $\varphi^{-1}(\beta) = (w, z)$. Then $\mathcal{Q}$ contains the edge (α, β) if and only if $\breve{D}_y = \breve{D}_w$ and $\varphi^{-1}(\beta) \overset{D^R}{\to} \varphi^{-1}(\alpha)$. That is, there is an edge from node $\varphi(e)$ to $\varphi(e')$, where

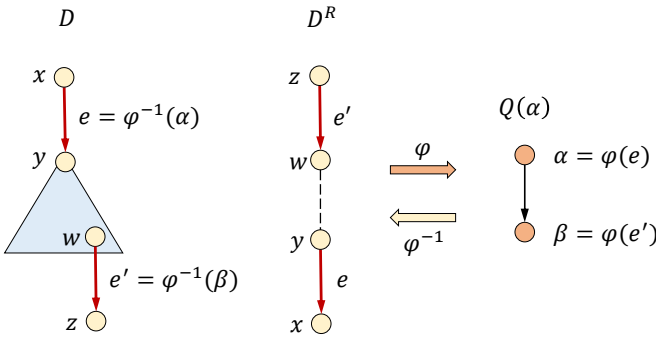


Figure 9: An illustration of the definition of the common bridge forest $\mathcal{Q}$.

$e = (x, y)$ and $e' = (w, z)$ are common bridges, if e is the common bridge that enters $\breve{D}_y = \breve{D}_w$ and e' is an ancestor of e in D^R. To see that $\mathcal{Q}$ is indeed a forest, consider the tree D' obtained from D by contracting each subtree $\breve{D} \in \breve{\mathcal{D}}$ into its root. Then, the edges of D' are the common bridges. Note that $\mathcal{Q}$ contains a node for each common bridge, and two nodes can be adjacent only if the corresponding common bridges are adjacent in D'. We use the notation $Q(\alpha)$ to denote the tree in the common bridge forest $\mathcal{Q}$ that contains node α.

LEMMA 3.12. *Let $\mathcal{Q}$ be the common bridge forest of G. Suppose that vertex x is a common descendant of a common bridge e and let e' be the nearest common bridge that is an ancestor of x in D such that $\varphi(e') \in Q(\varphi(e))$. Let $\pi = \langle \varphi(e) = \alpha_1, \alpha_2, \ldots, \alpha_\ell = \varphi(e') \rangle$ be the path from $\varphi(e)$ to $\varphi(e')$ in $Q(\varphi(e))$. Then x is a common descendant of every bridge $\varphi^{-1}(\alpha_i)$, $1 \le i \le \ell$.*

Proof. All common bridges that are represented by nodes in π are ancestors of x in D, since for each edge (α_i, α_{i+1}) in $\mathcal{Q}$ we have that $\varphi^{-1}(\alpha_i) \overset{D}{\to} \varphi^{-1}(\alpha_{i+1})$. Moreover, for each edge (α_i, α_{i+1}) we have $\varphi^{-1}(\alpha_{i+1}) \overset{D^R}{\to} \varphi^{-1}(\alpha_i)$: by the fact that $e = \varphi^{-1}(\alpha_1)$ is an ancestor of x in D^R (since we assumed that e is a common bridge ancestor of x), it follows that all common bridges that are represented by nodes in π are ancestors of x in D^R, and thus are common bridge ancestors of x. □

To compute the number $\#SCC(D(v) \cap D^R(u))$ for each common bridge $e = (u, v)$ we follow an approach similar to Algorithm SCCsDescendants. Namely, instead of computing the quantities $\#SCC(D(v) \cap D^R(u))$ one at the time for each common bridge $e = (u, v)$, we find for each vertex w its common bridge ancestors $e = (u, v)$ such that $H(w)$ induces a SCC in $G \setminus e$ and we increment the counter of $\#SCC(D(v) \cap D^R(u))$ for all those common bridges $e = (u, v)$. The following lemma, combined with Lemma 3.12, will allow us to accomplish this task efficiently.

LEMMA 3.13. *Let $e_1, e_2, \ldots, e_k$ be the common bridges in the path $D[\breve{r}_{h(x)}, x]$ from vertex $\breve{r}_{h(x)}$ to vertex x in D, in order of appearance in that path. If e_1 is not a common bridge ancestor of x then no e_j, $1 < j \leq k$, is. Otherwise, let j be the largest index $1 \leq j \leq k$ such that vertex x is a common descendant of e_j. Then $e_j \overset{D^R}{\rightarrow} e_{j-1} \overset{D^R}{\rightarrow} \ldots \overset{D^R}{\rightarrow} e_1$ and x is a common descendant of every common bridge e_i for $1 \leq i \leq j$.*

Proof. It suffices to show that if a common strong bridge e_ℓ, for any $1 < \ell \leq j$, is a common ancestor of x then $e_\ell \overset{D^R}{\rightarrow} e_{\ell-1}$ and $e_{\ell-1}$ is a common ancestor of x. Let $e_{\ell-1} = (u, u')$ and $e_\ell = (v, v')$. Assume by contradiction that the above statement is not true, i.e., either e_ℓ is not an ancestor of $e_{\ell-1}$ in D^R or $e_{\ell-1}$ is not a common ancestor of x. Then, in either case there must be a path π in G from x to v avoiding $e_{\ell-1}$. If π contained a vertex $x' \notin D(u')$, then the subpath of π from x' to v would include $e_{\ell-1}$, due to the fact that u is an ancestor of x in D and $v \in D(u')$. Thus, all vertices in π are descendants of u' in D. Let T be the dfs tree that generated H, and let pre be the preorder numbering in T. We claim that there is a vertex w in π such that all vertices in π are descendants of w in T. The claim implies that $x \in H(w)$, so $h(x)$ is a descendant of u' in D. But this contradicts the fact that $h(x)$ is not a descendant of u'. Hence, the lemma will follow.

To prove the claim, choose w to be the vertex in π such that $pre(w)$ is minimum. Then Lemma 2.1 implies that w is an ancestor of v in T. Let z be any vertex in π. We argue that $pre(w) \leq pre(z) < pre(w) + |T(w)|$, hence z is a descendant of w in π. By the choice of w we have $pre(w) \leq pre(z)$, so it remains to prove the second inequality. Suppose $pre(z) \geq pre(w) + |T(w)|$. Since x is descendant of v in D it is also a descendant of v in T. So $pre(x) < pre(z)$. By Lemma 2.1, path π contains a common ancestor q of x and z in T. Vertex q is an ancestor of w in T, since $x \in T(w)$ and $z \notin T(w)$. But then $pre(q) < pre(w)$, which contradicts the choice of w. Therefore, any path π from x to v must contain the common bridge $e_{\ell-1}$. We conclude that $e_\ell \overset{D^R}{\rightarrow} e_{\ell-1}$. □

We are now ready to describe our algorithm for computing the number $\#SCC(D(v) \cap D^R(u))$ for each common bridge $e = (u, v)$. That is, the number of SCCs in the subgraph induced by $D(v) \cap D^R(u)$, which, by Corollary 3.2, is equal to the number of SCCs in $G \setminus e$ that are composed only by common descendants of the common bridge e. Throughout the algorithm, we maintain a list of pairs L of the form $\langle x, y \rangle$, where x and y are vertices in G: the pair $\langle x, y \rangle$ will correspond to a path in the common bridge forest $\mathcal{Q}$, pinpointed by x and y, on which we wish to identify the common bridge ancestors of vertex y. The common bridge ancestors of any vertex z in G will be computed with the help of Lemma 3.13. Specifically, we use $\mathcal{Q}$ to locate the

Algorithm 2: SCCsCommonDescendants

Input: Strongly connected digraph $G = (V, E)$
Output: For each strong bridge $e = (x, y)$ the number $\#SCC(D(y) \cap D^R(x))$

Initialization:
Select an arbitrary start vertex $s \in V$.
Compute the dominator trees D and D^R, the loop nesting trees H and H^R, the common bridge decomposition forests $\breve{\mathcal{D}}$, and the common bridge forest $\mathcal{Q}$.
Initialize an empty list of pairs L.
forall the *nodes* $\alpha \in \mathcal{Q}$ **do** set $end(\alpha) = start(\alpha) = 0$
Construct list of pairs:
foreach *tree* $\breve{D}_r$ *with root* $r \neq s$ **do**
  foreach $x \in \breve{D}_r$ **do**
    if $h(x) \notin \breve{D}_r$ **then**
      $L = L \cup \langle h(x), x \rangle$
    end
  end
end
Process pairs:
foreach *pair* $\langle x, y \rangle \in L$ **do**
  Let $e = (w, z)$ be the common bridge such that $w \in \breve{D}_x$ and w is ancestor of y in D
  if w *is an ancestor of* y *in* D^R **then**
    Let e' be nearest common bridge that is an ancestor of y in D with $\varphi(e') \in Q(\varphi(e))$
    Set $end(\varphi(e')) = end(\varphi(e')) + 1$ and $start(\varphi(e)) = start(\varphi(e)) + 1$
  end
end
foreach *tree* Q *in* $\mathcal{Q}$ **do**
  foreach *node* $\alpha \in Q$, *in a bottom-up fashion* **do**
    Let $e = \varphi^{-1}(\alpha) = (u, v)$
    Set $\#SCC(D(v) \cap D^R(u)) = end(\alpha)$
    foreach *edge* (α, β) *in* Q **do**
      Let $e' = \varphi^{-1}(\beta) = (u', v')$
      Set $\#SCC(D(v) \cap D^R(u)) = \#SCC(D(v) \cap D^R(u)) + \#SCC(D(v') \cap D^R(u')) - start(\beta)$
    end
  end
end

endpoints of each sequence of common bridges that satisfy Lemma 3.12, and will propagate this information to the whole sequence in a second phase. Note that, as a special case, such a sequence may consist of only one common bridge. To accomplish this, we will maintain for each node α of $\mathcal{Q}$ (corresponding to common bridge $\varphi^{-1}(\alpha)$ of G_s and G_s^R) two values, denoted respectively by $start(\alpha)$ and $end(\alpha)$, defined as follows. The value $start(\alpha)$ (resp., $end(\alpha)$) stores the number of SCCs that contain only common descendants of a bridge contained in a sequence (or sequences) of common bridges starting (resp., ending) at bridge $\varphi^{-1}(\alpha)$.

The pseudocode of the algorithm is given in Algorithm 2. We first consider all vertices of G one at the time (lines 7–13). For each vertex x such that $h(x) \notin \breve{D}_{\breve{r}_x}$ we add the pair $\langle h(x), x\rangle$ to the list L to indicate that for each common bridge e that is both in the path $D[\breve{r}_{h(x)}, x]$ from $\breve{r}_{h(x)}$ to x in the dominator tree D and it is a common ancestor of x, the set of vertices $H(x)$ induces a SCC in $G \setminus e$. By Theorem 3.1(c) all vertices in the SCC $H(x)$ in $G \setminus e$ are also common descendants of e, i.e., $H(x) \subseteq D(v) \cap D^R(u)$. After traversing all the vertices, we process the pairs that were inserted in the list L. Note that there can be at most $n-1$ pairs in L, since at most one pair is inserted for each vertex $x \neq s$, and no pair is inserted for $x = s$. Furthermore, by construction, all the pairs are of the form $\langle h(x), x\rangle$. So let $\langle h(x), x\rangle$ be a pair extracted from L. We first identify, the common bridge (w, z) such that $\breve{D}_w = \breve{D}_{h(x)}$ and w is an ancestor of x in D. If $e = (w, z)$ is a common ancestor of x, then by Lemma 3.12, so are all common bridges that correspond to the nodes in the path π from $\varphi(e)$ to $\varphi(e')$ in $Q(\varphi(e))$, where $e' = (y, d^R(y))$ is the last common bridge in the path from w to x in D such that $\varphi(e') \in Q(\varphi(e))$. That means that $H(x)$ is a SCC in $G \setminus e''$ for each common bridge e'' such that $\varphi(e'') \in \pi$. To account for this, we increment $end(\varphi(e'))$ and $start(\varphi(e))$ accordingly. In the final loop (lines 22–31), we visit all trees in the common bridge forest $\mathcal{Q}$. We process the nodes of each tree Q of the forest $\mathcal{Q}$ in a bottom-up fashion. When we visit a node $\alpha \in Q$ we compute $\#SCC(D(v) \cap D^R(u))$, where $e = (u, v) = \varphi^{-1}(\alpha)$, as follows. We first initialize $\#SCC(D(v) \cap D^R(u))$ to $end(\alpha)$ (line 25). Next (lines 27–28), for each child β of α, where $e' = \varphi^{-1}(\beta) = (u', v')$, we increment $\#SCC(D(v) \cap D^R(u))$ by the quantity $\#SCC(D(v') \cap D^R(u')) - start(\beta)$.

At the end of the algorithm, $\#SCC(D(v) \cap D^R(u))$ contains the number of the SCCs in $G \setminus e$ that include only common descendants of the common bridge $e = (u, v)$, as shown by the following lemma.

LEMMA 3.14. *Algorithm* SCCsCommonDescendants *is correct.*

Proof. For each vertex x such that $h(x) \notin \breve{D}_x$ the set $H(x)$ is a SCC that contains only vertices that are common descendants of a strong bridge e only if e lies in the path $D[\breve{r}_{h(x)}, x]$ and e is a common bridge ancestor of x. We prove that we correctly identify each such common bridge $e = (u, v)$ for each vertex x and we account for $H(x)$ in $\#SCC(D(v) \cap D^R(u))$. If $h(x) \notin \breve{D}_x$ then by Lemma 3.7, $H(x)$ induces a SCC in $G \setminus e'$ for all bridges e' in $D[\breve{r}_{h(x)}, x]$. Therefore, to identify for which common bridges $e = (u, v)$ the set $H(x)$ induces a SCC in $G \setminus e$ and $H(x) \subseteq D(v) \cap D^R(u)$, it is sufficient to check which of the common bridges in $D[\breve{r}_{h(x)}, x]$ are common bridge ancestors of vertex x. By Lemma 3.13 if there is a bridge $e' = (w, z)$ in $D[\breve{r}_{h(x)}, x]$ that is a common bridge ancestor of x then all the bridges in the subpath $D[\breve{r}_{h(x)}, z]$ are common bridge ancestors of x, and there is a path in $\mathcal{Q}$ from $\varphi(e'')$ to $\varphi(e')$ containing all common bridges in $D[\breve{r}_{h(x)}, z]$, where e'' is the first bridge in $D[\breve{r}_{h(x)}, x]$. Let $e_1, e_2, \ldots, e_k$, be the common bridges in $D[\breve{r}_{h(x)}, x]$, and let e_ℓ be the nearest common bridge that is an ancestor of x in D such that $\varphi(e_\ell) \in Q(\varphi(e_1))$. If e_1 is a common bridge ancestor of x, then by Lemmata 3.12 and 3.13 so are all the common bridges in the path from $\varphi(e_1)$ to $\varphi(e_\ell)$. If e_1 is not a common bridge ancestor of x, then by Lemma 3.13 no other common bridge in $D[\breve{r}_{h(x)}, x]$ is a common bridge ancestor of x. Thus, by simply testing whether e_1 is a common bridge ancestor of x we can determine whether all the common bridges $e_1, e_2, \ldots, e_\ell$, are common bridge ancestors of x and update their counter $\#SCC(D(v_i) \cap D^R(u_i))$, for $1 \leq i \leq \ell$, where $e_i = (u_i, v_i)$. We mark this relation by adding the pair $\langle h(x), x\rangle$ to the list L: the related update of the counters $\#SCC(D(v_i) \cap D^R(u_i))$ will be done during the second phase (lines 15–31) when the pairs in the list L will be processed.

To complete the proof, we need to show that the pairs in L are handled correctly during the second phase of the algorithm. Consider a pair $\langle x, y\rangle \in L$. Then, it must be $x = h(y)$, and by Lemma 3.10 we have that $\breve{r}_x$ is an ancestor of y in D. Now we have to locate the common bridge ancestors of y that are in the path $D[\breve{r}_x, y]$, and increase the count of the SCCs that contain only common descendants by one, since $H(y)$ induces a SCC after deleting each one of them. To find the common bridge ancestors of y, the algorithm locates the bridge $e = (w, z)$ that is an ancestor of y in D such that $\breve{D}_w = \breve{D}_x$. If e is also an ancestor of y in D^R, it follows that e is a common bridge ancestor of y. Hence, Lemma 3.12 and Lemma 3.13 imply that the common bridge ancestors we seek correspond to the nodes in the path $Q[\varphi(e), \varphi(e')]$, where e' is the nearest common bridge that is an ancestor of y in D

such that $\varphi(e') \in Q(\varphi(e))$. This common bridge e' is located, and then we need to increase by one the count of the SCCs that contain only common descendants of the common bridges in $Q[\varphi(e), \varphi(e')]$. We do this by incrementing $end(\varphi^{-1}(e'))$ and $start(\varphi^{-1}(e))$ by one. The actual number of SCCs that contain only common descendants of a common bridge $e = (u, v)$, i.e., $\#SCC(D(v) \cap D^R(u))$ is computed in the bottom-up traversal of Q. When we visit a node $\alpha \in \mathcal{Q}$, where $e = \varphi^{-1}(\alpha) = (u, v)$, we first initialize its number $\#SCC(D(v) \cap D^R(u))$ to $end(\alpha)$. Next, for each child $\varphi(e')$ of α in $\mathcal{Q}$, we increment this number by $\#SCC(D(v') \cap D^R(u')) - start(\varphi(e'))$. This way, we increase by one the count of SCCs that contain only common descendants of a common bridge if and only if it corresponds to a node in the path $Q[\varphi(e), \varphi(e')]$, as required. □

LEMMA 3.15. *Given the dominator trees D and D^R, the loop nesting trees H and H^R, and the strong bridges of G, Algorithm* SCCsCommonDescendants *runs in $O(n)$ time.*

Proof. Since each dominator tree has $n - 1$ edges, we can locate the common bridges on D and construct the common bridge decomposition $\breve{\mathcal{D}}$ in $O(n)$ time. The common bridge forest $\mathcal{Q}$ can also be constructed in $O(n)$ time, since it only requires a constant-time ancestor/descendant test for D^R [27] in order to identify the common bridges that are adjacent in $\mathcal{Q}$. If the dominator trees, the loop nesting trees, and the strong bridges are available, then the initialization phase of Algorithm SCCsCommonDescendants (lines 1–4) can be implemented in a total of $O(n)$ time.

Now we turn to the main loop of the algorithm (lines 6–14) where we visit all vertices and insert the appropriate pairs into the list L. It requires $O(n)$ time since it visits every vertex once and performs constant-time computations per vertex. Here again, we use a constant-time ancestor/descendant test for both D and D^R. Next, we consider the last phase (lines 14–31). As we already mentioned, L contains at most $n - 1$ pairs. The foreach loop in lines 15–21 requires $O(n)$ time, except for lines 16 and 18, which we account later. The last foreach loop (lines 22–31) takes $O(n)$ time since it visits each node of $\mathcal{Q}$ only once and iterates over its children, performing only constant-time computations per child.

To complete the proof, we need to specify how to compute efficiently on line 16 and on line 18 the appropriate common bridges e and e', respectively. We identify the common bridge $e = (w, d^R(w))$ that is an ancestor of y in D such that $\breve{D}_w = \breve{D}_x$. (Recall that $h(y) = x$.) In order to locate efficiently the appropriate common bridge for every pair, we compute them together in a pre-processing step as follows. First, we perform a preorder traversal of D and assign to each vertex u a preorder number $pre(u)$. Then, we create two lists of triples, A and B. List A contains the triple $\langle \breve{r}_x, pre(y), 1\rangle$ for each pair $\langle x, y\rangle \in L$. List B contains the triple $\langle \breve{r}_u, pre(u), 0\rangle$ for each common bridge (u, v). We sort both lists in increasing order in $O(n)$ time by bucket sort and then merge them. (Without loss of generality, we can assume that vertices of G are integers from 1 to n). Let C be the resulting list. We divide C into sublists $C(r)$, where r is the first vertex of each triple in $C(r)$. Consider a triple $\langle r, pre(y), 1\rangle$ that corresponds to the pair $\langle x, y\rangle \in L$, where $\breve{r}_x = r$. The desired common bridge $(w, d^R(w))$ corresponds to the last triple of the form $\langle r, pre(w), 0\rangle$ that precedes $\langle r, pre(y), 1\rangle$.

Now, given a vertex y and the common bridge $e = (w, z)$ computed above for a pair $\langle x, y\rangle \in L$, we wish to find the node $\varphi(e') \in \mathcal{Q}$ such that e' is the nearest common bridge that is an ancestor of y in D and $\varphi(e') \in Q(\varphi(e))$. Again we create two lists of triples, A' and B'. List A' contains the triple $\langle \alpha, pre(y), 1\rangle$ for each pair $\langle x, y\rangle \in L$, where α is the root of $Q(\varphi((w, z)))$, where $e = (w, z)$ is the common bridge computed above. List B' contains the triple $\langle \alpha, pre(u), 0\rangle$ for each common bridge (u, v), where α is the root of $Q(\varphi((u, v)))$. We bucket sort both lists A' and B' in increasing order and merge them in a list C'. As before, we divide C' into sublists $C'(\alpha)$, where α is the first vertex of each triple in $C'(\alpha)$. Consider a triple $\langle \alpha, pre(y), 1\rangle$. The desired nearest common bridge $(z, d^R(z))$ for y corresponds to the last triple of the form $\langle \alpha, pre(z), 0\rangle$ that precedes $\langle \alpha, pre(y), 1\rangle$. □

THEOREM 3.3. *Given the dominator trees D and D^R, the loop nesting trees H and H^R, and the strong bridges of a strongly connected digraph G, number of SCCs after the deletion of a strong bridge in $O(n)$ time for all strong bridges.*

COROLLARY 3.3. *Given a directed graph $G = (V, E)$, we can find in worst-case time $O(m+n)$ a strong bridge e in G that maximizes / minimizes the total number of SCCs of $G \setminus e$.*

Extension to more general family of functions. The algorithm given in this section can be extended to a more general family of functions defined over the sizes of the SCCs of $G \setminus e$, for each edge e. We summarize the result with the following theorem. More details can be found in the full version of the paper.

THEOREM 3.4. *Let $\odot$ be an associative and commutative binary operation such that its inverse operation $\odot^{-1}$ is defined and both $\odot$ and $\odot^{-1}$ are computable in*

constant time. Let $f(x)$ be a function defined on positive integers which can be computed in constant time. Given a strongly connected digraph G, we can compute in $O(m+n)$ time, for all edges e, the function $f(|C_1|) \odot f(|C_2|) \odot ... \odot f(|C_k|)$, where $C_1, C_2, ..., C_k$ are the SCCs in $G \setminus e$.

3.3 Finding all the smallest and all the largest SCCs of $G \setminus e$. Let G be a strongly connected digraph, with m edges and n vertices. Since G is strongly connected, $m \geq n$. In this section we consider the problem of answering the following aggregate query: "Find the size of the largest / smallest SCC of $G \setminus e$, for all edges e." We recall here that the size of a SCC is given by its number of vertices, so the largest component (resp., smallest component) is the one with maximum (resp., minimum) number of vertices. In the following, we restrict ourselves to the computation of the largest SCCs of $G \setminus e$, since the smallest SCCs can be computed in a completely analogous fashion. Note again that the naive solution is to compute the SCCs of $G \setminus e$ for all strong bridges e of G, which takes $O(mn)$ time. We will be able to provide a linear-time algorithm for this problem, which also gives an asymptotically optimal $O(m)$-time algorithm for the motivating biological application discussed in the introduction, i.e., finding the strong bridge e that minimizes the size of the largest SCC in $G \setminus e$. Once we find such a strong bridge e, we can report the actual SCCs of $G \setminus e$ in $O(n)$ additional time by using the algorithm of Section 3.1.

Let $S \subseteq V$ be a subset of vertices of G. We denote by $LSCC(S)$ the size of the largest SCC in the subgraph of G induced by the vertices in S. Also, for an edge e of G, we denote by $LSCC_e(V)$ the size of the largest SCC in $G \setminus e$. Our goal is to compute $LSCC_e(V)$ for every strong bridge e in G. Then, Theorem 3.1 immediately implies the following:

COROLLARY 3.4. *Let $e = (u, v)$ be a strong bridge of G and let s be an arbitrary vertex in G. The cardinality of the largest SCC of $G \setminus e$ is equal to*

(a) $\max\{LSCC(D(v)), |V \setminus D(v)|\}$ *when e is a bridge in G_s but not in G_s^R.*

(b) $\max\{LSCC(D^R(u)), |V \setminus D^R(u)|\}$ *when e is a bridge in G_s^R but not in G_s.*

(c) $\max\{LSCC(D(v)), LSCC(D^R(u)), |V \setminus (D(v) \cup D^R(u))|\}$ *when e is a bridge of both G_s and G_s^R.*

Moreover, $LSCC(D(v)) = \max_w\{|H(w)| : w \in D(v)$ *and* $h(w) \notin D(v)\}$ *and* $LSCC(D^R(u)) = \max_w\{|H^R(w)| : w \in D^R(u)$ *and* $h^R(w) \notin D^R(u)\}$.

Now we develop an algorithm that applies Corollary 3.4. Our algorithm uses the dominator and the loop nesting trees of G_s and its reverse G_s^R, with respect to an arbitrary start vertex s, and computes for each strong bridge e of G the size of the largest SCC of $G \setminus e$, denoted by $LSCC_e(V)$.

In order to get an efficient implementation of our algorithm, we need to specify how to compute efficiently the following quantities:

(a) $LSCC(D(v))$ for every vertex v such that the edge $(d(v), v)$ is a bridge in flow graph G_s.

(b) $LSCC(D^R(u))$ for every vertex u such that the edge $(u, d^R(u))$ is a bridge in flow graph G_s^R.

(c) $|D(v) \cup D^R(u)|$ for every strong bridge (u, v) of G that is a common bridge of flow graphs G_s and G_s^R.

We deal with computations of type (a) first. The computations of type (b) are analogous. We precompute for all vertices v the number of their descendants in the loop nesting tree H, and we initialize $LSCC(D(v)) = 0$. Then we process D in a bottom-up order. For each vertex v, we store the nearest ancestor w of v in D such that $(d(w), w)$ is a bridge in G_s. Recall that we use the notation r_v to refer to the vertex w with this property, and we have $r_v = s$ if no such w exists for v. For every vertex v in a bottom-up order of D we update the current value of $LSCC(D(r_v))$ by setting

$$LSCC(D(r_v)) = \max\{LSCC(D(r_v)), |H(v)|\}.$$

If $(d(v), v)$ is also a bridge in G_s, we update the current value of $LSCC(D(r_{d(v)}))$ by setting

$$LSCC(D(r_{d(v)})) = \max\{LSCC(D(r_{d(v)})), LSCC(D(v))\}.$$

These computations take $O(n)$ time in total.

Finally, we need to specify how to compute the values of type (c), that is, we need to compute the cardinality of the union $D(v) \cup D^R(u)$ for each strong bridge (u, v) that is a common bridge in both flow graphs G_s and G_s^R. Since $|D(v) \cup D^R(u)| = |D(v)| + |D^R(u)| - |D(v) \cap D^R(u)|$, it suffices to compute the cardinality of the intersections $D(v) \cap D^R(u)$ for all common bridges (u, v). We describe next how to compute these values in $O(n)$ time. This gives an implementation of Algorithm LSCC which runs in $O(m+n)$ time in the worst case, or in $O(n)$ time in the worst case once the dominator trees, loop nesting trees and strong bridges are available.

Computing common descendants of strong bridges. We now consider the problem of computing the number of common descendants, i.e., the cardinality of the intersections $D(v) \cap D^R(u)$, for all common bridges $e = (u, v)$. Let x be a common descendant of a strong bridge $e = (u, v)$ (i.e., $x \in D(v)$ and $x \in D^R(u)$), and let C be the SCC containing x in $G \setminus e$. By Theorem

3.1(c), $C \subseteq D(v) \cap D^R(u)$, i.e., all vertices in the same SCC C of $G \setminus e$ as x must also be common descendants of e. Therefore, the number of common descendants of a strong bridge e can be computed as the sum of the sizes of the SCCs containing a vertex that is a common descendant of e. This observation allows us to solve efficiently our problem with a simple variation of Algorithm SCCsCommonDescendants from Section 3.2.

Recall that Algorithm SCCsCommonDescendants maintains for each strong bridge $e = (u, v)$ a counter $\#SCC(D(v) \cap D^R(u))$ for the number of SCCs in $G \setminus e$ that contain only vertices in $D(v) \cap D^R(u)$. The high-level idea behind the new algorithm is to maintain for each strong bridge $e = (u, v)$ a counter for *the sum of the sizes* of the SCCs in $G \setminus e$ that contain only vertices in $D(v) \cap D^R(u)$. We can maintain those counters by proceeding as in Algorithm SCCsCommonDescendants. Exactly as before, we identify for each vertex x all the common bridges e after whose deletion the set $H(x)$ induces a SCC containing only vertices that are common descendants of e. However, this time we need to compute for each common bridge $e = (u, v)$ the total number of common descendants (i.e., vertices in $D(v) \cap D^R(u)$) rather than the number of SCCs in $D(v) \cap D^R(u)$: when we process $H(x)$ we increase the counter of the common bridge e by $|H(x)|$ (rather than by 1 as before). This is correct since by Theorem 3.1 all vertices in $H(x)$ are common descendants of e.

We next describe the two small modifications needed in the pseudocode of Algorithm SCCsCommonDescendants to deal with our problem. Recall that Algorithm SCCsCommonDescendants added a pair $\langle h(x), x\rangle$ to the list L (lines 7–13) to indicate that the set $H(x)$ induces a SCC in $G \setminus e$, for each common bridge e that lies in the path $D[\breve{r}_{h(x)}, x]$ from $\breve{r}_{h(x)}$ to x in the dominator tree D and that is also a common ancestors of x. This time, we will insert a triple of the form $\langle h(x), x, |H(x)|\rangle$, to denote that the set $H(x)$ contributes $|H(x)|$ common descendants to all common bridges in $D[\breve{r}_{h(x)}, x]$ that are common ancestors of x. Algorithm SCCsCommonDescendants used the information stored in the pairs $\langle h(x), x\rangle$ while processing the list L in lines 15–21: for each pair $\langle h(x), x\rangle$, the algorithm incremented by 1 the values $start(\varphi(e))$ and $end(\varphi(e'))$, where e and e' were respectively the first and last bridges in the sequence of common strong bridges in $D[\breve{r}_{h(x)}, x]$ that are common ancestors of x. The new algorithm will process a triple in similar fashion: when a triple $\langle h(x), x, |H(x)|\rangle$ is extracted from the list L, the algorithm will increment by $|H(x)|$ the corresponding values $start(\varphi(e))$ and $end(\varphi(e'))$. Clearly these small modifications do not affect the $O(n)$ time complexity of Algorithm SCCsCommonDescendants:

LEMMA 3.16. *Given the dominator trees D and D^R, the loop nesting trees H and H^R, and the common bridges of G, we can compute the number of the common descendants of all common strong bridges in $O(n)$ time.*

Hence, we have the following results.

THEOREM 3.5. *Given the dominator trees D and D^R, the loop nesting trees H and H^R, and the strong bridges of G, we can compute the size of the largest / smallest SCC in $G \setminus e$ in $O(n)$ time for all strong bridges e.*

COROLLARY 3.5. *Given a strongly connected directed graph G, we can find in worst-case time $O(m + n)$ a strong bridge e that minimizes / maximizes the size of the largest / smallest SCC in $G \setminus e$.*

4 Pairwise 2-edge connectivity queries

Let $G = (V, E)$ be a strongly connected digraph, let x and y be any two vertices of G, and let $e = (u, v)$ be a strong bridge of G. Recall that we say that e is a *separating edge for x and y* (or equivalently that e *separates x and y*) if x and y are not strongly connected in $G \setminus e$. In this section we show how to extend our data structure to answer in asymptotically optimal time the following types of queries:

(a) Test if two query vertices x and y are 2-edge-connected; if not, report a separating edge for x and y.

(b) Test whether a given edge e separates two query vertices x and y.

(c) Report all the separating edges for a given pair of vertices x and y.

We note that the data structure of [13] only supports, in constant time, the queries of type (a). Our framework allows us to answer such queries, also in constant time (see full version of the paper for the details). Next we deal with queries of type (b) and (c).

LEMMA 4.1. *Let x and y be two vertices, and let w and w^R be their nearest common ancestors in the loop nesting trees H and H^R, respectively. Then, a strong bridge $e = (u, v)$ is a separating edge for x and y if and only if one of the following conditions holds:*

(1) The edge e is an ancestor of x or y in D and w is not a descendant of e in D.

(2) The edge e is an ancestor of x or y in D^R and w^R is not a descendant of e in D^R.

Proof. Omitted due to lack of space. □

Note that, by Lemma 2.2, a bridge e of G_s (resp., G_s^R) is not a separating edge for x and y only if it is

an ancestor of both x and y in D (resp., D^R). This fact, combined with Lemma 4.1 suggests the following algorithm for computing, in an online fashion, all the separating edges for a given pair of query vertices x and y. First, we preprocess the loop nesting trees H and H^R in $O(n)$ time so that we can compute nearest common ancestors in constant time [16]. To answer a reporting query for vertices x and y, we compute their nearest common ancestor w in H, and visit the bridges of G_s that are ancestors of x and y in D in a bottom-up order, as follows. Starting from x and y, we visit the bridges that are ancestors of x and y in D until we reach a bridge $e = (u, v)$ such that v is an ancestor of w in D, or until we reach s if no such bridge e exists. (As we showed above, if there is a bridge e that is an ancestor of x or y in D and is not a separating vertex for x and y then e is an ancestor of both x and y in D.) If $e = (u, v)$ exists then the bridges in $D[v, x] \cup D[v, y]$ are separating edges for x and y. Otherwise the bridges in $D[s, x] \cup D[s, y]$ are separating edges for x and y. We do the same for the reverse direction, i.e., compute the nearest common ancestor w^R of x and y in H^R, and visit the bridges of G_s^R in a bottom-up order in D^R, starting from x and y, until we reach a bridge $e = (u, v)$ such that v is an ancestor of w^R in D^R, or until we reach s if no such e exists. We finally return the union of the separating edges that we found in both directions. To speed up this process, we can compute in a preprocessing step compressed versions of D and D^R that are formed by contracting, respectively, each vertex $u \in D$ into r_u, and each vertex $u \in D^R$ into r_u^R.

With the same data structure we can test in constant time, for any pair of query vertices x and y and a query edge e, if e is a separating edge for x and y. By Lemma 4.1, it suffices to find their nearest common ancestors, w in H and w^R in H^R, and then test whether e is an ancestor of x or y and w is not a descendant of e in D, or whether e is an ancestor of x or y and w^R is not a descendant of e in D^R. This gives the following theorem.

THEOREM 4.1. *After $O(m)$-time preprocessing, we can build an $O(n)$-space data structure, that can:*

- *Test if two query vertices x and y are 2-edge-connected and if not report a corresponding separating edge.*
- *Report all edges that separate two query vertices x and y in $O(k)$ time, where k is the total number of separating edges reported. For $k = 0$, the time is $O(1)$.*
- *Test in constant time if a query edge is a separating edge for a pair of query vertices.*

5 Other results

Our approach extends naturally to provide efficient algorithms for computing the corresponding problems with respect to vertex failures. The interested reader is referred to the full paper for all the details of the algorithms and their analysis.

THEOREM 5.1. *We can preprocess a digraph G in $O(m+n)$ time and construct an $O(n)$-space data structure, so that given a vertex v of G we can report in $O(n)$ time all the SCCs of $G \setminus v$.*

THEOREM 5.2. *Given the dominator trees D and D^R, the loop-nesting trees H and H^R we can compute the number of the SCCs or the size of the largest / smallest SCC in $G \setminus v$ in $O(n)$ time for all strong articulation points v.*

COROLLARY 5.1. *Given a strongly connected directed graph G, we can find in worst-case time $O(m + n)$ a strong articulation point v that minimizes / maximizes the size of the largest / smallest SCC in $G \setminus e$.*

As a side result, we can answer in optimal asymptotic time basic connectivity queries:

THEOREM 5.3. *Let G be a strongly connected digraph with m edges and n vertices. After $O(m + n)$ preprocessing, we can build an $O(n)$-space data structure, that can report all vertices that separate two query vertices u and v in $O(k)$ time, where k is the total number of separating vertices reported. For $k = 0$, the time is $O(1)$. Moreover, we can test in constant time if a query vertex is a separating vertex for a pair of query vertices.*

Our new techniques provide also alternative linear-time algorithms for computing the 2-edge-connected and 2-vertex-connected components of a digraph. The new algorithms are much simpler than the algorithms presented in [12, 13], and thus are likely to be more amenable to practical implementations. Furthermore, they require only $O(n)$ time once the dominator trees, loop nesting trees and the strong bridges are available.

THEOREM 5.4. *Given the dominator trees D and D^R, the loop-nesting trees H and H^R and the strong bridges of a strongly connected digraph G, we can compute in worst-case time $O(n)$ the 2-edge-connected and the 2-vertex-connected components of G.*

Using our framework we can also compute in linear time a spanning subgraph of a directed graph G with $O(n)$ edges that has the same 1-connectivity cuts, and the same decompositions induced by those cuts, as G.

THEOREM 5.5. *Given a strongly connected directed graph G with m edges and n vertices, we can compute in $O(m+n)$ time a sparse certificate containing $O(n)$ edges*

that maintains the same 1*-connectivity cuts and the same decompositions induced by the* 1*-connectivity cuts of* G*, together with the* 2*-edge and* 2*-vertex-connected components of* G*.*

Acknowledgments. We are indebted to Peter Widmayer for suggesting the problem of finding the edge whose deletion yields the smallest largest strongly connected components in a strongly connected digraph, and to Matúš Mihalák, Przemysław Uznański and Pencho Yordanov for useful discussions on the practical applications of this problem and for pointing out references [15, 23].

References

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling. Vol. 2, Compiling.* Prentice-Hall, Englewood Cliffs, N.J, 1972.

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

[3] A. A. Benczúr. Counterexamples for directed and node capacitated cut-trees. *SIAM J. on Computing*, 24:505–510, 1995.

[4] S. P. Borgatti. Identifying sets of key players in a social network. *Computational & Mathematical Organization Theory*, 12(1):21–34, 2006.

[5] 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 J. on Computing*, 38(4):1533–1573, 2008.

[6] S. Chechik, T. D. Hansen, G. F. Italiano, V. Loitzenbauer, and N. Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In *SODA*, 2017. (To appear).

[7] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. on Computing*, 37(5):1299–1318, January 2008.

[8] T. N. Dinh, Y. Xuan, M. T. Thai, P. M. Pardalos, and T. Znati. On new approaches of assessing network vulnerability: Hardness and approximation. *IEEE/ACM Trans. Networking*, 20(2):609–619, 2012.

[9] Y. M. Erusalimskii and G. G. Svetlov. Bijoin points, bibridges, and biblocks of directed graphs. *Cybernetics*, 16(1):41–44, 1980.

[10] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *J. of Discrete Algorithms*, 23:2–20, 2013.

[11] H. N. Gabow. The minset-poset approach to representations of graph connectivity. *ACM Trans. Algorithms*, 12(2):24:1–24:73, February 2016.

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

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

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

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

[16] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13(2):338–55, 1984.

[17] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *ICALP*, 2015.

[18] 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.

[19] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. of Computer and System Sciences*, 65(1):150 – 167, 2002.

[20] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. on Programming Languages and Systems*, 1(1):121–41, 1979.

[21] E. S. Lorry and V. W. Medlock. Object code optimization. *Comm. of the ACM*, 12(1):13–22, 1969.

[22] S. Makino. An algorithm for finding all the k-components of a digraph. *Int. J. of Computer Mathematics*, 24(3–4):213–221, 1988.

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

[24] H. Nagamochi and T. Watanabe. Computing k-edge-connected components of a multigraph. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E76–A.4:513–517, 1993.

[25] N. Paudel, L. Georgiadis, and G. F. Italiano. Computing critical nodes in directed graphs. In *Proc. 19th Workshop on Algorithm Engineering and Experiments*, 2017. (To appear).

[26] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. on Computing*, 1(2):146–160, 1972.

[27] R. E. Tarjan. Finding dominators in directed graphs. *SIAM J. on Computing*, 3(1):62–89, 1974.

[28] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

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

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

[31] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(5&6):433–464, 1992.