Linear-Time Algorithms for Computing Twinless Strong Articulation Points and Related Problems

Loukas Georgiadis
University of Ioannina, Greece
loukas@cs.uoi.gr

Evangelos Kosinas
University of Ioannina, Greece
ekosinas@cs.uoi.gr

Abstract

A directed graph $G = (V, E)$ is twinless strongly connected if it contains a strongly connected spanning subgraph without any pair of antiparallel (or *twin*) edges. The twinless strongly connected components (TSCCs) of a directed graph G are its maximal twinless strongly connected subgraphs. These concepts have several diverse applications, such as the design of telecommunication networks and the structural stability of buildings. A vertex $v \in V$ is a twinless strong articulation point of G, if the deletion of v increases the number of TSCCs of G. Here, we present the first linear-time algorithm that finds all the twinless strong articulation points of a directed graph. We show that the computation of twinless strong articulation points reduces to the following problem in undirected graphs, which may be of independent interest: Given a 2-vertex-connected undirected graph H, find all vertices v for which there exists an edge e such that $H \setminus \{v, e\}$ is not connected. We develop a linear-time algorithm that not only finds all such vertices v, but also computes the number of edges e such that $H \setminus \{v, e\}$ is not connected. This also implies that for each twinless strong articulation point v which is not a strong articulation point in a strongly connected digraph G, we can compute the number of TSCCs in $G \setminus v$.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases 2-connectivity, cut pairs, strongly connected components

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.38

Related Version A full version of the paper is available at [9], `https://arxiv.org/abs/2007.03933`.

Funding Research supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant", Project FANTA (eFficient Algorithms for NeTwork Analysis), number HFRI-FM17-431.

1 Introduction

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* (SCCs) of G are its maximal strongly connected subgraphs. Two vertices $u, v \in V$ are *strongly connected* if they belong to the same strongly connected component of G. We refer to a pair of antiparallel edges, (x, y) and (y, x), of G as *twin edges*. A digraph $G = (V, E)$ is *twinless strongly connected* if it contains a strongly connected spanning subgraph (V, E') without any pair of twin edges. The *twinless strongly connected components* (TSCCs) of G are its maximal twinless strongly connected subgraphs. Two vertices $u, v \in V$ are *twinless strongly connected* if they belong to the same twinless strongly connected component of G. Twinless strong connectivity is motivated by several diverse applications, such as the design of telecommunication networks and the structural stability

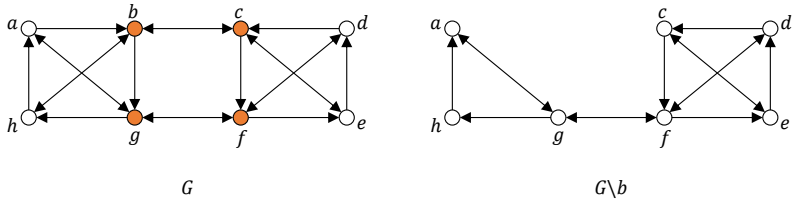

Figure 1 A 2-vertex-connected digraph G that is not twinless 2-vertex-connected. Vertices b, c, g and f are twinless strong articulation points but not strong articulation points; for instance, $G \setminus b$ is strongly connected but not twinless strongly connected.

of buildings [17]. Raghavan [17] provided a characterization of twinless strongly connected digraphs, and, based on this characterization, provided a linear-time algorithm for computing the TSCCs of a digraph.

In this paper, we further explore the notion of twinless strong connectivity, with respect to 2-connectivity in digraphs. An edge (resp., a vertex) of a digraph G is a *strong bridge* (resp., a *strong articulation point*) if its removal increases the number of strongly connected components. Thus, strong bridges (resp., strong articulation points) are 1-edge (resp., 1-vertex) cuts for digraphs. A strongly connected digraph G is 2-*edge-connected* if it has no strong bridges, and it is 2-*vertex-connected* if it has at least three vertices and no strong articulation points. Let $C \subseteq V$. The induced subgraph of C, denoted by $G[C]$, is the subgraph of G with vertex set C and edge set $E \cap (C \times C)$. If $G[C]$ is 2-edge-connected (resp., 2-vertex-connected), and there is no set of vertices C' with $C \subsetneq C' \subseteq V$ such that $G[C']$ is also 2-edge-connected (resp., 2-vertex-connected), then $G[C]$ is a *maximal* 2-*edge-connected (resp., 2-vertex-connected) subgraph of* G. Two vertices $u, v \in V$ are said to be 2-*edge-connected* (resp., 2-*vertex-connected*) 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 every two vertices $u, v \in B$ are 2-edge-connected (resp., 2-vertex-connected). We note that connectivity-related problems for digraphs are known to be much more difficult than for undirected graphs, and indeed many notions for undirected connectivity do not translate to the directed case. See, e.g., [3, 8, 11]. Indeed, it has only recently been shown that all strong bridges and strong articulation points of a digraph can be computed in linear time [13]. Additionally, it was shown very recently how to compute the 2-edge- and 2-vertex-connected components of digraphs in linear time [6, 7], while the best current bound for computing the maximal 2-edge- and the 2-vertex-connected subgraphs in digraphs is not even linear, but it is $O(\min\{m^{3/2}, n^2\})$ [3, 11].

The above notions extend naturally to the case of twinless strong connectivity. An edge $e \in E$ is a *twinless strong bridge* of G if the deletion of e increases the number of TSCCs of G. Similarly, a vertex $v \in V$ is a *twinless strong articulation point* of G, if the deletion of v increases the number of TSCCs of G. A linear-time algorithm for detecting all twinless strong bridges can be derived by combining the linear-time algorithm of Italiano et al. [13] for computing all the strong bridges of a digraph and a linear-time algorithm for computing all the edges which belong to a cut-pair in a 2-edge-connected undirected graph. (See [9] for details.) Previously, Jaberi [16] studied the properties of twinless strong articulation points and some related concepts, and presented an $O(m(n-s))$-time algorithm for their

computation, where s is the number of strong articulation points of G. Hence, this bound is $O(mn)$ in the worst case. Here, we present a linear-time algorithm that identifies all the twinless strong articulation points. Specifically, we show that the computation of twinless strong articulation points reduces to the following problem in undirected graphs, which may be of independent interest: Given a 2-vertex-connected (biconnected) undirected graph H, find all vertices v that belong to a vertex-edge cut-pair, i.e., for which there exists an edge e such that $H \setminus \{v, e\}$ is not connected. We develop a linear-time algorithm that not only finds all such vertices v, but also computes the number of vertex-edge cut-pairs of v (i.e., the number of edges e such that $H \setminus \{v, e\}$ is not connected). This implies that for each twinless strong articulation point v, that is not a strong articulation point in a digraph G, we can also compute the number of twinless strongly connected components of $G \setminus v$. After the submission of this paper we noticed that it is possible to compute the vertices that form a vertex-edge cut-pair by exploiting the structure of the triconnected components of H, represented by an SPQR tree [1, 2] of H. We refer to the full version [9] for the details. In order to construct an SPQR tree, however, we need to know the triconnected components of the graph [10], and efficient algorithms that compute triconnected components are considered conceptually complicated (see, e.g., [4, 10, 12]). Our approach, on the other hand, is conceptually simple and thus likely to be more amenable to practical implementations. Also, we believe that our results and techniques will be useful for the design of faster algorithms for related connectivity problems, such as computing twinless 2-connected components [14, 15].

2 Preliminaries

Let G be a (directed or undirected) graph. We denote by $V(G)$ and $E(G)$, respectively, the vertex set and edge set of G. For a set of edges (resp., vertices) S, we let $G \setminus S$ denote the graph that results from G after deleting the edges in S (resp., the vertices in S and their adjacent edges). We extend this notation for mixed sets S, that may contain both vertices and edges of G, in the obvious way. Also, if S has only one element x, we abbreviate $G \setminus S$ by $G \setminus x$. Let $C \subseteq V(G)$. The induced subgraph of C, denoted by $G[C]$, is the subgraph of G with vertex set C and edge set $\{e \in E(G) \mid \text{both ends of } e \text{ lie in } C\}$.

For any digraph G, the associated undirected graph G^u is the graph with vertices $V(G^u) = V(G)$ and edges $E(G^u) = \{\{u, v\} \mid (u, v) \in E(G) \vee (v, u) \in E(G)\}$. Let H be an undirected graph. An edge $e \in E(H)$ is a *bridge* if its removal increases the number of connected components of H. A connected graph H is 2-edge-connected if it contains no bridges. Raghavan [17] proved the following characterization of twinless strongly connected digraphs.

▶ **Theorem 1** ([17]). *Let G be a strongly connected digraph. Then G is twinless strongly connected if and only if its underlying undirected graph G^u is 2-edge-connected.*

Theorem 1 implies a linear-time algorithm to compute the twinless strongly connected components (TSCCs) of a digraph G. It suffices to compute the strongly connected components, $C_1, \ldots, C_k$, of G, and then the 2-edge-connected components of each underlying undirected graph $G^u[C_i]$, $1 \leq i \leq k$. All these computations take linear time [18].

Another immediate consequence of Theorem 1 is that a twinless strong bridge in a twinless strongly connected graph is either (1) a strong bridge or (2) an edge whose removal destroys the 2-edge connectivity in the underlying graph. All strong bridges can be found in linear time [13]. To compute the edges of type (2), we only have to find all the edges of the underlying graph whose removal destroys the 2-edge connectivity (i.e., all edges which belong to a cut-pair in the underlying graph), which can be done in linear-time by the algorithm of Tsin [20] or the one described in the Appendix of [9]. (We refer to [9] for the details.)

3 Computing twinless strong articulation points

It is an immediate consequence of Theorem 1 that a twinless strong articulation point in a twinless strongly connected digraph G is either (1) a strong articulation point or (2) a vertex whose removal destroys the 2-edge connectivity in the underlying undirected graph G^u. Since all strong articulation points can be computed in linear time [13], it remains to find all vertices of type (2). Note that such a vertex x either (a) entirely destroys the connectivity of the underlying graph G^u with its removal, or (b), upon removal, it leaves us with a graph $G^u \setminus x$ that is connected but not 2-edge-connected. Clearly, the set of vertices with property (a) are a subset of the set of strong articulation points. Therefore, it suffices to find all vertices with property (b). To that end, we process each 2-vertex-connected component of G^u separately, as the following Lemma suggests.

▶ **Lemma 2** ([9]). *Let H be a 2-edge-connected undirected graph. Let v be a vertex that is not an articulation point, and let C be its 2-vertex-connected component. For any edge e, $H \setminus \{v, e\}$ is not connected if and only if e belongs to C and $C \setminus \{v, e\}$ is not connected.*

So, in order to find all twinless strong articulation points, it is sufficient to solve the following problem: *Given a 2-vertex-connected undirected graph G, find all vertices v for which there exists an edge e such that $G \setminus \{v, e\}$ is not connected.* In Section 4 we describe a linear-time algorithm for this problem. Our algorithm utilizes properties of depth-first search (DFS), which are reminiscent of the seminal algorithm of Hopcroft and Tarjan for computing the triconnected components of a graph [12].

Formally, our main technical contribution is summarized in the following theorem:

▶ **Theorem 3.** *Let G be a biconnected undirected graph. There is a linear time algorithm that computes, for every vertex v, the number of edges e such that $G \setminus \{v, e\}$ is not connected.*

Then, Theorem 3 implies the following results:

▶ **Corollary 4.** *Let G be a twinless strongly connected digraph. There is a linear time algorithm that finds all the twinless strong articulation points of G. Moreover, for every twinless strong articulation point v that is not strong a articulation point of G, the algorithm computes the number of TSCCs in $G \setminus v$, in total linear time.*

▶ **Corollary 5** ([9]). *Let G be a biconnected undirected graph. After linear-time preprocessing, we can answer queries of the form: Given a vertex v of G, report all the edges in the set $C(v) = \{e \in E(G) \mid G \setminus \{v, e\}$ is not connected$\}$, in $O(|C(v)|)$ time.*

3.1 Depth-first search, low and high points

Let G be a 2-vertex-connected graph. We consider a DFS traversal of G, starting from an arbitrarily selected vertex r, and let T be the resulting DFS tree [18]. A vertex u is an ancestor of a vertex v (v is a descendant of u) if the tree path from r to v contains u. Thus, we consider a vertex to be an ancestor (and, consequently, a descendant) of itself. We let $p(v)$ denote the parent of a vertex v in T. If u is a descendant of v in T, we denote the set of vertices of the simple tree path from u to v as $T[u, v]$. The expressions $T[u, v)$ and $T(u, v]$ have the obvious meaning (i.e., the vertex on the side of the parenthesis is excluded). Furthermore, we let $T(v)$ denote the subtree of T rooted at vertex v. We identify vertices in G by their DFS number, i.e., the order in which they were discovered by the search. Hence, $u < v$ means that u was discovered before v. The edges in $E(T)$ are called tree-edges; the

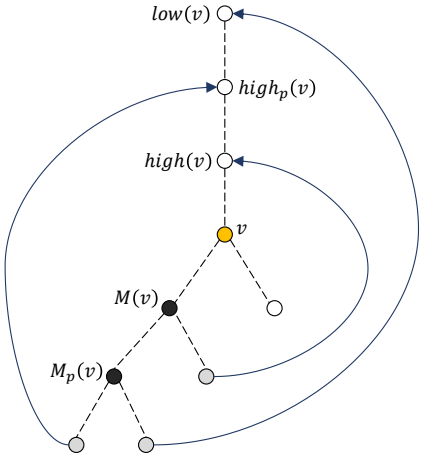

Figure 2 Concepts defined on the structure of the DFS tree that are essential to our algorithm. Dashed lines correspond to DFS tree paths. Back-edges are shown directed from descendant to ancestor.

edges in $E(G) \setminus E(T)$ are called back-edges, as their endpoints have ancestor-descendant relation in T. When we write (u, v) to denote a back-edge, we always mean that $v \leq u$, i.e., u is an descendant of v in T.

Now we describe some concepts that are defined on the structure given by the DFS and are essential to our algorithm. For an illustration, see Figure 2. Let us note beforehand, that, since G is biconnected, all these concepts are well-defined. Define the "low point", $low(v)$, of a vertex $v \neq r$, as the minimum vertex (w.r.t. the DFS numbering) that is connected via a back-edge to a descendant of v, i.e., the minimum vertex in the set $\{u \mid \text{there is a back-edge } (w, u) \text{ such that } w \text{ is a descendant of } v\}$. Define the "high point", $high(v)$, of $v \neq r$, as the maximum proper ancestor of v which is connected with a back-edge to a descendant of v. The notion of low points plays central role in classic algorithms for computing the biconnected components [18] and the triconnected components [12] of a graph. The low points of all vertices can be computed in linear time. (Hopcroft and Tarjan [12] also use a concept of high points, which, however, is different from ours.) Since G is biconnected, r has a unique child vertex c in T. For any vertex $v \notin \{r, c\}$, define $high_p(v)$ to be the maximum proper ancestor of $p(v)$ which is connected with a back-edge to a descendant of v. Finally, for any vertex $v \neq r$, define $M(v)$ as the nearest common ancestor of all descendants of v that are connected with a back-edge to a proper ancestor of v, and, for any vertex $v \notin \{r, c\}$, define $M_p(v)$ as the nearest common ancestor of all descendants of v that are connected with a back-edge to a proper ancestor of $p(v)$.

The following two Lemmata, which combine properties of M, M_p, $high$, and $high_p$, will be useful in analyzing Algorithms 3 and 5 in Sections 4.2.2 and 4.3.2, respectively. Their proof is essentially the same.

▶ **Lemma 6** ([9]). *Let v and v' be two vertices such that v is a descendant of v' with $M(v) = M(v')$ and $high(v) < v'$. Then $high(v) = high(v')$.*

▶ **Lemma 7.** *Let v and v' be two vertices such that v is a descendant of v' with $M_p(v) = M_p(v')$ and $high_p(v) < p(v')$. Then $high_p(v) = high_p(v')$.*

Next, we show how to compute $high(v)$, $high_p(v)$, $M(v)$ and $M_p(v)$, for all vertices v, in total linear time.

3.2 Finding all $high(v)$ and $high_p(v)$ in linear time

The basic idea to compute all $high(v)$ (for $v \neq r$) is to do the following: We process the back-edges (u, v) in decreasing order with respect to their lower end v. When we process (u, v), we ascend the path $T[u, v)$, and for each visited vertex x such that $high[x]$ is still undefined, we set $high[x] \leftarrow v$. It should be clear that this process, which forms the basis of our linear-time algorithm, computes all $high(v)$, for $v \neq r$, correctly.

In order to achieve linear running time, we have to be able, when we consider a back-edge (u, v), to bypass all vertices on the path $T[u, v)$ whose *high* value has been computed. To that end, it suffices to know, for every vertex x in $T[u, v)$, the nearest ancestor of x whose *high* value is still null. We can achieve this by applying a disjoint-set-union (DSU) structure [19]. Algorithm 1 gives a fast algorithm for computing all $high(v)$ (for $v \neq r$).

The DSU structure is implemented by a dynamic forest F, which is a subgraph of T, subject to the following operations:

link(x, y): Adds the edge (x, y) into the forest F.

find(x): Return the root of the tree in F that contains x.

Let F_x denote the tree of F that contains a vertex x. Initially, F contains no edges, so x is the unique vertex in F_x. In our algorithm, the link operation always adds some tree edge $(u, p(u))$ to F, so the invariant that F is a subgraph of T is maintained. This is implemented by uniting the corresponding sets of u and $p(u)$ in the underlying DSU structure, and setting the root of $F_{p(u)}$ as the representative of the resulting set. Then, *find*(u) returns the root of F_u, which will be the nearest ancestor of u in T whose *high* value is still null.

Algorithm 1 FastHigh.

```
1  initialize a forest F with V(F) = V(T) and E(F) = ∅
2  foreach vertex v ≠ r do set high[v] ← null
3  sort the back-edges (u, v) in decreasing order w.r.t. to their lower end v
4  foreach back-edge (u, v) do
5  |  u ← find(u)
6  |  while u > v do
7  |  |  high[u] ← v
8  |  |  next ← find(p(u))
9  |  |  link(u, p(u))
10 |  |  u ← next
11 |  end
12 end
```

The next lemma summarizes the properties of Algorithm 1.

▶ **Lemma 8** ([9]). *Algorithm 1 is correct. Furthermore, it will perform $n - 1$ link and $2m - n + 1$ find operations on a 2-vertex-connected graph with n vertices and m edges.*

Since all the *link* operations we perform are of the type *link*$(u, p(u))$, and the total number of *link* and *find* operations performed is $O(m + n)$, we may use the static tree DSU data structure of Gabow and Tarjan [5] to achieve linear running time.

Finally, we note that the algorithm for computing all $high_p(v)$ is almost identical to Algorithm 1. The only difference is in line 6, where we have to replace "**while** $u > v$" with "**while** $p(u) > v$". The proof of correctness and linearity is essentially the same.

3.3 Finding all $M(v)$ and $M_p(v)$ in linear time

Recall that $M(v)$, for $v \neq r$, is the nearest common ancestor of all descendants of v that are connected with a back-edge to a proper ancestor of v.

Our algorithm for the computation of $M(v)$ works recursively on the children of v. So, let v be a vertex $\neq r$. We define $l(v) = min\{\{v\} \cup \{u \mid \text{there exists a back-edge } (v,u)\}\}$. Now, if $l(v) < v$, we have $M(v) = v$. (This is always the case when v is a leaf, since the graph is biconnected.) Furthermore, if there exist two children c, c' of v such that $low(c) < v$ and $low(c') < v$, then, again, $M(v) = v$. The difficulty arises when $l(v) = v$ and there is only one child c of v with the property $low(c) < v$ (one such child of v must of necessity exist, since the graph is biconnected), in which case $M(v)$ is a descendant of c, and therefore $M(v)$ is a descendant of $M(c)$. (See [9] for a proof of this property of M.) In this case, we repeat the same process in $M(c)$: we test whether $l(M(c)) < v$ or whether there exists only one child d of $M(c)$ such that $low(d) < v$, in which case we repeat the same process in $M(d)$, and so on, until $M(v)$ is finally computed.

Now, we claim that a careful implementation of the above procedure yields a linear-time algorithm for the computation of $M(v)$, for all vertices $v \neq r$. To that end, it suffices to store, for every vertex v that is not a leaf of T, two pointers, $L(v)$ and $R(v)$, on the list of the children of v. Initially, $L(v)$ points to the first child c of v that has $low(c) < v$, and $R(v)$ points to the last child c' of v that has $low(c') < v$. Our algorithm works in a bottom-up fashion. Provided we have computed $M(u)$ for every descendant u of v, we execute Procedure FindM(v).

Procedure FindM(v).

```
1  if l(v) < v then return v
2  if L[v] ≠ R[v] then return v
3  m ← M[L[v]]
4  while M(v) = null do
5  |  if l(m) < v then return m
6  |  while low(L[m]) ≥ v do L[m] ← next child of m
7  |  while low(R[m]) ≥ v do R[m] ← previous child of m
8  |  if L[m] ≠ R[m] then return m
9  |  m ← M[L[m]]
10 end
```

▶ **Lemma 9** ([9]). *By executing Procedure FindM(v), for all vertices $v \neq r$, in bottom-up fashion of T, we can compute all $M(v)$ in linear-time.*

We use a similar algorithm in order to compute all $M_p(v)$. The only change we have to make in Procedure FindM, is to replace every comparison to v with a comparison to $p(v)$.

4 Finding all vertices that belong to a vertex-edge cut-pair

Let $H = (V, E)$ be a 2-vertex-connected undirected graph. For every v in V, we define $count(v) := \#\{e \in E \mid \{v, e\} \text{ is a cut-pair}\}$. We will find all vertices which belong to a vertex-edge cut-pair of H by computing all $count(v)$. We notice that the parameter $count(v)$ is also useful for counting TSCCs, as suggested by the following Lemma (from which Corollary 4 follows).

▶ **Lemma 10** ([9]). *Let G be a twinless strongly connected digraph, and let v be a twinless strong articulation point of G which is not a strong articulation point. Then $count(v) + 1$ (computed in the 2-vertex-connected component of v in G^u) is the number of the TSCCs of $G \setminus v$.*

Now, to compute all $count(v)$, we will work on the tree structure T, with root r, provided by a DFS on H. Then, if $\{v, e\}$ is a vertex-edge cut-pair, e can either be a back-edge, or a tree-edge. Furthemore, in the case that e is a tree-edge, we have the following:

▶ **Lemma 11** ([9]). *If $\{v, e\}$ is a cut-pair such that e is a tree-edge, then e either lies in $T(v)$ or on the simple tree path $T[v, r]$.*

Thus we have three distinct cases in total, and we will compute $count(v)$ by counting the cut-pairs $\{v, e\}$ in each case. We will handle these cases separately, by providing a specific algorithm for each one of them, based on some simple observations like Lemma 11. The linearity of these algorithms will be clear.

Now, we shall begin with the case where e is a back-edge, since this is the easiest to handle. We suppose that all $count(v)$ have been initialized to zero.

4.1 The case where e is a back-edge

▶ **Proposition 12** ([9]). *If $\{v, e\}$ is a cut-pair such that e is a back-edge, then e starts from the subtree $T(c)$ of a child c of v, ends in a proper ancestor of v, and is the only back-edge that starts from $T(c)$ and ends in a proper ancestor of v. Conversely, if e is such a back-edge, then $\{v, e\}$ is a cut-pair.*

This immediately suggests an algorithm for counting all such cut-pairs. We only have to count, for every vertex c ($\neq r$ or the child of r), the number $b_count(c) := \#\{$back-edges that start from $T(c)$ and end in a proper ancestor of $p(c)\}$. To do this efficiently, we define, for every vertex v, $up(v) := \#\{$back-edges that start from v and end in a proper ancestor of $p(v)\}$, and, for every child c of v (if it has any), $down(v, c) := \#\{$back-edges that start from $T(c)$ and end in $v\}$. All $up(v)$ and $down(v, c)$ can be computed easily in linear time. Now, $b\_count(c)$ can be computed recursively: if $d_1, \ldots, d_k$ are the children of c, then $b\_count(c) = up(c) + b\_count(d_1) + \ldots + b\_count(d_k) - down(p(c), c)$; and if c is childless, $b\_count(c) = up(c)$. Finally, the number of vertex-edge cut-pairs $\{v, e\}$ where e is a back-edge, equals the number of children c of v that have $b\_count(c) = 1$.

4.2 The case where e is part of the simple tree path $T[v, r]$

Let $\{v, e\}$ be a vertex-edge cut-pair such that e is part of the simple tree path $T[v, r]$. Then there exists a vertex u which is a proper ancestor of v and such that $e = (u, p(u))$. We observe that all back-edges that start from $T(u)$ and end in a proper ancestor of u must necessarily start from $T(v)$. In other words, $M(u)$ is a descendant of v. Here we further distinguish two cases, depending on whether $M(u)$ is a proper descendant of v.

4.2.1 The case $M(u) = v$

Our algorithm for this case is based on the following observation:

▶ **Proposition 13** ([9]). *Let $c_1, \ldots, c_k$ be the children of v (if it has any), and let $\{v, (u, p(u))\}$ be a cut-pair such that u is an ancestor of v with $M(u) = v$. Then u does not belong to any set of the form $T[high_p(c_i), low(c_i))$, for $i = 1, \ldots, k$. Conversely, given that u is a proper*

ancestor of v such that $M(u) = v$, and given also that u does not belong to any set of the form $T[high_p(c_i), low(c_i))$, for $i = 1, \ldots, k$, we may conclude that the pair $\{v, (u, p(u))\}$ is a cut-pair. (See Figure 3.)

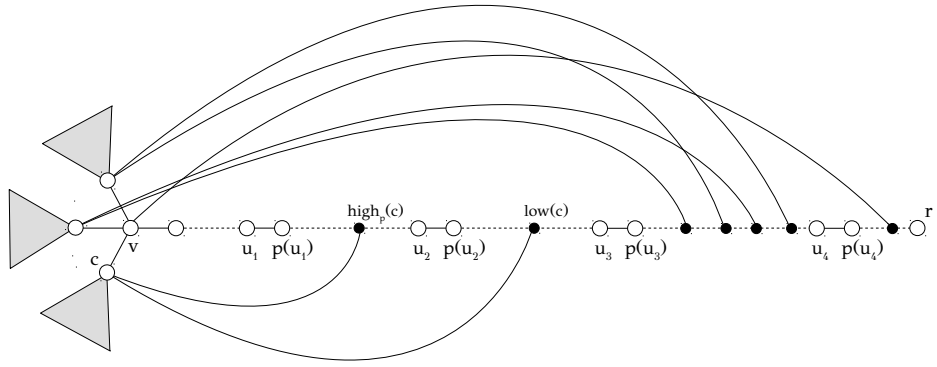

Figure 3 In this DFS-tree structure we have $M(u_1) = M(u_2) = M(u_3) = M(u_4) = v$, and $\{v, (u_1, p(u_1))\}$, $\{v, (u_3, p(u_3))\}$, $\{v, (u_4, p(u_4))\}$ are cut-pairs. $(u_2, p(u_2))$ does not form a cut-pair with v, since $u_2 \in T[high_p(c), low(c))$.

Now, let $c_1, \ldots, c_k$ be the children of v (if it has any). According to Proposition 13, in order to count all cut-pairs of the form $\{v, (u, p(u))\}$, with $M(u) = v$, it is sufficient to find all $u \in M^{-1}(v)$ such that u a proper ancestor of v which does not belong to any set of the form $T[high_p(c_i), low(c_i))$, for any $i \in \{1, \ldots, k\}$. (Of course, if v is a leaf, every $u \in M^{-1}(v) \setminus \{v\}$ satisfies this property.) Now, suppose that $c_1, \ldots, c_k$ are sorted in decreasing order with respect to their $high_p$ point, and let $S = T[high_p(c_1), low(c_1)) \cup \ldots \cup T[high_p(c_k), low(c_k))$. The idea is to find all pairs (x, y) such that $T[x, y)$ is a subset of S maximal with respect to the property of having the form $T[z, w)$ and being a subset of S. If $(x_1, y_1), \ldots, (x_t, y_t)$, with $x_1 > \ldots > x_t$, is the collection of all these pairs, then $T[x_1, y_1) \cup \ldots \cup T[x_t, y_t) = S$, and thus we only have to find all $u \in M^{-1}(m) \setminus \{v\}$ which do not belong to any set of the form $T[x_i, y_i)$, for any $i \in \{1, \ldots, t\}$. It is easy to see that, for every $i \in \{1, \ldots, t\}$, there exist $j \in \{1, \ldots, k\}$ and $j' \in \{j, \ldots, k\}$, such that $x_i = high_p(c_j)$ and $y_i = min\{low(c_j), \ldots, low(c_{j'})\}$. Thus we can efficiently compute all $(x_1, y_1), \ldots, (x_t, y_t)$ (in this order), provided that $c_1, \ldots, c_k$ are sorted in decreasing order with respect to their $high_p$ point. Moreover, provided that $M^{-1}(v)$ is also sorted in decreasing order (all these sortings take linear time with bucket-sort), we can easily find all $u \in M^{-1}(m) \setminus \{v\}$ which do not belong to any $T[x_i, y_i)$, during the computation of the pairs (x_i, y_i), $i \in \{1, \ldots, t\}$.

Algorithm 2 is an implementation of this idea. It essentialy finds, for every vertex v, all cut-pairs of the form $\{v, (u, p(u))\}$, with $M(u) = v$, in total linear time. Let $(x_1, y_1), \ldots, (x_t, y_t)$ be as above, and let $y_0 = p(v)$. The **while** loop in Line 10 counts all $u \in T[y_i, x_{i+1}) \cap M^{-1}(v)$, for $i \in \{0, \ldots, t-1\}$. Lines 14-19 calculate the next y_i, and Line 20 sets u to be the greatest element of $M^{-1}(v)$ which is lower than y_i. Finally, the **while** loop in Line 22 counts all $u \in M^{-1}(v) \setminus \{v\}$ such that $u \leq min\{low(c_1), \ldots, low(c_k)\}$.

4.2.2 The case where $M(u)$ is a proper descendant of v

In this case, $M(u)$ belongs to $T(c)$, for a child c of v, and so we have that $\{p(c), (u, p(u))\}$ is a cut-pair. We base our algorithm for this case on the following observation:

Algorithm 2 $M(u) = v$.

```
1  calculate all lists M^{-1}(v), for all vertices v, and have their elements sorted in
     decreasing order
2  sort the list of the children of every vertex in decreasing order w.r.t. the high_p value
     of its elements
3  foreach vertex v do
4      if M^{-1}(v) = ∅ then continue
5      u ←second element of M^{-1}(v) // the first element of M^{-1}(v) is v
6      c ←first child of v
7      min ← v
8      while u ≠ ∅ and c ≠ ∅ do
9          min ← high_p(c)
10         while u ≠ ∅ and u > min do
11             count[v] ← count[v] + 1
12             u ←next element of M^{-1}(v)
13         end
14         min ← low(c)
15         c ←next child of v
16         while c ≠ ∅ and high_p(c) ≥ min do
17             if low(c) < min then min ← low(c)
18             c ←next child of v
19         end
20         while u ≠ ∅ and u > min do u ←next element of M^{-1}(v)
21     end
22     while u ≠ ∅ do
23         if u ≤ min then count[v] ← count[v] + 1
24         u ←next element of M^{-1}(v)
25     end
26 end
```

▶ **Proposition 14** ([9]). *Let $\{p(c), (u, p(u))\}$ be a cut-pair, such that u is an ancestor of $p(c)$ and $M(u)$ is in $T(c)$. Then $M_p(c) = M(u)$ and $high_p(c) < u$. Conversely, if u is a proper ancestor of $p(c)$ such that $M_p(c) = M(u)$ and $high_p(c) < u$, then the pair $\{p(c), (u, p(u))\}$ is a cut-pair. (See Figure 4.)*

Algorithm 3 shows how we can compute, for every vertex v, the number of cut-pairs of the form $\{v, (u, p(u))\}$, where u is a proper ancestor of v with $M(u)$ in $T(c)$ for a child c of v, in total linear time. To see how and why it works, notice that in order to find the number of these cut-pairs, it is sufficient, according to Proposition 14, to focus our attention on the lists $M_p^{-1}(m)$ and $M^{-1}(m)$, for every vertex m, and count, for every $c \in M_p^{-1}(m)$, the number of $u \in M^{-1}(m)$ which are proper ancestors of $p(c)$ and such that $high_p(c) < u$. Let $U(c)$ denote the collection of those u. Assuming that the lists $M_p^{-1}(m)$ and $M^{-1}(m)$ are sorted in decreasing order, a simple idea to find $\#U(c)$ is to traverse the list $M^{-1}(m)$ from the greatest vertex which is lower than $p(c)$, to the lowest vertex which is greater than $high_p(c)$, and count the number of all elements encountered. Although this is not a linear-time procedure to compute all $\#U(c)$ (since, for some $c, c' \in M_p^{-1}(m)$, we may have that the sets $U(c)$ and $U(c')$ overlap, and so the total number of elements of $M^{-1}(m)$ traversed can be quadratic to the number of vertices), it is the basis for the linear-time algorithm, which is derived by taking advantage of the nested structure of the sets $U(c)$, described in the following Lemma.

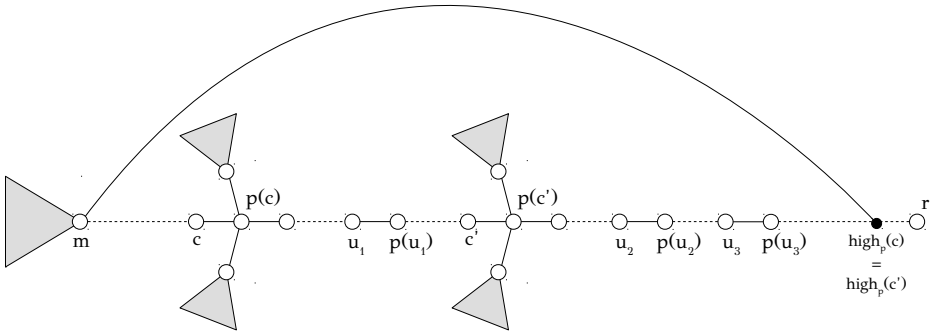

Figure 4 In this DFS-tree structure we have $M_p(c) = M_p(c') = m$ and $M(u_1) = M(u_2) = M(u_3) = m$. The vertex-edge pairs $\{p(c), (u_1, p(u_1))\}$, $\{p(c), (u_2, p(u_2))\}$, $\{p(c), (u_3, p(u_3))\}$, $\{p(c'), (u_2, p(u_2))\}$, and $\{p(c'), (u_3, p(u_3))\}$, are cut-pairs. Following the notation in Lemma 15, $U(c) = \{u_1, u_2, u_3\}$ and $U(c') = \{u_2, u_3\}$.

▶ **Lemma 15.** *Let m be a vertex, c and c' two elements of $M_p^{-1}(m)$ such that $U(c) \cap U(c') \neq \emptyset$, and assume, without loss of generality, that c is a descendant of c'. Then $U(c') = U(c) \cap T(p(c'), high_p(c))$. (See Figure 4.)*

Proof. Let u be an element of both $U(c)$ and $U(c')$. Then u is an ancestor of c' and $high_p(c)$ is a proper ancestor of u (by the definitions of $U(c')$ and $U(c)$, respectively). Thus $high_p(c)$ is a proper ancestor of c', and therefore, since c is a descendant of c' with $M_p(c) = M_p(c')$, it follows from Lemma 7 that $high_p(c) = high_p(c')$. Now we see why $U(c) \cap T(p(c'), high_p(c)) \subseteq U(c')$: every u in $U(c) \cap T(p(c'), high_p(c))$ is a proper ancestor of $p(c')$ with $M(u) = M_p(c) = M_p(c')$ and $high_p(c') = high_p(c) < u$. To prove the reverse inclusion, notice first that, since $high_p(c) = high_p(c')$, by the definition of $U(c')$ we have $U(c') \subseteq T(p(c'), high_p(c))$. Now, if u is in $U(c')$, then u is a proper ancestor of $p(c')$, and therefore a proper ancestor of $p(c)$; furthermore, it satisfies $M(u) = M_p(c') = M_p(c)$ and $high_p(c) = high_p(c') < u$, and therefore it is also in $U(c)$. This shows that $U(c') \subseteq U(c)$. We conclude that $U(c') \subseteq U(c) \cap T(p(c'), high_p(c))$, and the proof is complete. ◀

Thus, if we have established that $U(c)$ and $U(c')$ overlap (Line 8 checks whether $U(c)$ is not empty, and Line 18 checks whether $U(c')$ overlaps with $U(c)$, where c' is a successor of c in $M_p^{-1}(m)$), in order to calculate $\#U(c')$ it is sufficient that we have calculated $\#U(c)$ and the greatest and lowest elements of $U(c)$ - call them $first_c$ and $last_c$, respectively. Then we traverse the list $M^{-1}(m)$ from $first_c$, until we reach the greatest element u of $M^{-1}(m)$ such that $u < p(c)$, and let k be the number of the elements encountered (excluding u). Now we set $first_{c'} \leftarrow u$ and $last_{c'} \leftarrow last_c$, and we have $\#U(c') = \#U(c) - k$.

4.3 The case where e lies in $T(v)$

Let $\{v, (u, p(u))\}$ be a cut-pair where u is a descendant of v. Then u is a proper descendant of a child c of v, and we observe that every back-edge that starts from $T(u)$ and ends in a proper ancestor of u must necessarily end in an ancestor of $p(c)$. In other words, $high(u) \leq v$. Here we distinguish two cases, depending on whether $high(u)$ is a proper ancestor of v.

Algorithm 3 $M(u) > v$.

```
1  calculate all lists M^{-1}(m) and M_p^{-1}(m), for all vertices m, and have their elements
     sorted in decreasing order
2  foreach vertex m do
3      c ← first element of M_p^{-1}(m)
4      u ← first element of M^{-1}(m)
5      while c ≠ ∅ and u ≠ ∅ do
6          while u ≠ ∅ and u ≥ p(c) do u ← next element of M^{-1}(m)
7          if u = ∅ then break
8          if high_p(c) < u then
9              n_edges ← 0
10             first ← u
11             while u ≠ ∅ and high_p(c) < u do
12                 n_edges ← n_edges + 1
13                 u ← next element of M^{-1}(m)
14             end
15             last ← predecessor of u in M^{-1}(m)
16             count[p(c)] ← count[p(c)] + n_edges
17             c ← next element of M_p^{-1}(m)
18             while c ≠ ∅ and p(c) > last do
19                 while first ≥ p(c) do
20                     n_edges ← n_edges − 1
21                     first ← next element of M^{-1}(m)
22                 end
23                 count[p(c)] ← count[p(c)] + n_edges
24                 c ← next element of M_p^{-1}(m)
25             end
26         end
27         else
28             c ← next element of M_p^{-1}(m)
29         end
30     end
31 end
```

4.3.1 The case $high(u) = v$

Our algorithm for this case is based on the following observation:

▶ **Proposition 16** ([9]). *Let $\{v, (u, p(u))\}$ be a cut-pair such that v is a proper ancestor of u with $high(u) = v$, and let c be the child of v of which u is a descendant. Then, either* (1) *$low(u) = p(c)$, or* (2) *$low(u) < p(c)$ and $u \leq M_p(c)$. Conversely, if c is a proper ancestor of u such that $high(u) = p(c)$ and either* (1) *or* (2) *holds, then the pair $\{p(c), (u, p(u))\}$ is a cut-pair.*

It is an immediate application of Proposition 16 that Algorithm 4 correctly computes, for every vertex v, the number of cut-pairs $\{v, (u, p(u))\}$ with the property that u is a descendant of v with $high(u) = v$ (and it essentially finds all of them), in total linear time. Due to the ordering of $high^{-1}(v)$ and of the list of the children of v, we can easily check the ancestry

relation in Line 7 by testing whether c is the last child of v, or whether $c \leq u$ and $c' > u$, where c' is the successor of c in the list of the children of v. Notice that, in Line 8, it is sufficient to check whether $low(u) = v$ or $u \leq M_p(c)$, since $low(u) \leq high(u) = v$.

Algorithm 4 $high(u) = v$.

```
1  calculate all lists high^{-1}(v), for all vertices v, and have their elements sorted in
     increasing order
2  sort the list of the children of every vertex in increasing order
3  foreach vertex v do
4  |  u ← first element of high^{-1}(v)
5  |  c ← first child of v
6  |  while u ≠ ∅ do
7  |  |  while c is not an ancestor of u do c ← next child of v
8  |  |  if u ≠ c and (low(u) = v or u ≤ M_p(c)) then count(v) ← count(v) + 1
9  |  |  u ← next element of high^{-1}(v)
10 |  end
11 end
```

4.3.2 The case $high(u) < v$

Our algorithm for this case is based on the following observation:

▶ **Proposition 17** ([9]). *Let $\{p(c), (u, p(u))\}$ be a cut-pair such that u is a descendant of c with $high(u) < p(c)$. Then $M(u) = M_p(c)$. Conversely, if u is a proper descendant of c such that $M(u) = M_p(c)$ and $high(u) < p(c)$, then the pair $\{p(c), (u, p(u))\}$ is a cut-pair. (See Figure 5.)*

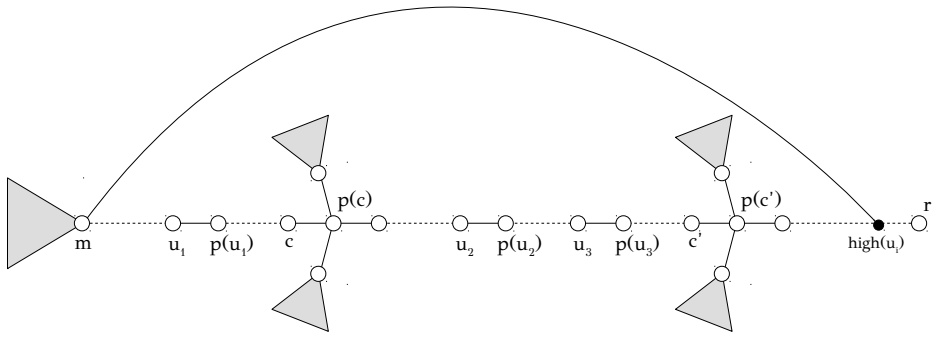

Figure 5 In this DFS-tree structure we have $M_p(c) = M_p(c') = m$ and $M(u_1) = M(u_2) = M(u_3) = m$. The vertex-edge pairs $\{p(c), (u_1, p(u_1))\}$, $\{p(c'), (u_1, p(u_1))\}$, $\{p(c'), (u_2, p(u_2))\}$, and $\{p(c'), (u_3, p(u_3))\}$, are cut-pairs. Following the notation in Lemma 18, $\tilde{U}(c) = \{u_1\}$ and $\tilde{U}(c') = \{u_1, u_2, u_3\}$.

Algorithm 5 shows how we can compute, for every vertex v, the number of cut-pairs of the form $\{v, (u, p(u))\}$, where u is a descendant of v with $high(u) < v$, in total linear time. The explanation of how and why it works is similar to that given in Section 4.2.2 for Algorithm 3

Algorithm 5 $high(u) < v$.

```
1  calculate all lists M^{-1}(m) and M_p^{-1}(m), for all vertices m, and have their elements
     sorted in decreasing order
2  foreach vertex m do
3      u ← first element of M^{-1}(m)
4      c ← first element of M_p^{-1}(m)
5      while u ≠ ∅ and c ≠ ∅ do
6          while c ≠ ∅ and c ≥ u do c ← next element of M_p^{-1}(m)
7          if c = ∅ then break
8          if high(u) < p(c) then
9              n_edges ← 0
10             h ← high(u)
11             while c ≠ ∅ and h < p(c) do
12                 while u ≠ ∅ and c < u do
13                     n_edges ← n_edges + 1
14                     u ← next element of M^{-1}(m)
15                 end
16                 count[p(c)] ← count[p(c)] + n_edges
17                 c ← next element of M_p^{-1}(m)
18             end
19         end
20         else
21             u ← next element of M^{-1}(m)
22         end
23     end
24 end
```

(in both cases the counting is done in an indirect manner, i.e. without finding all cut-pairs explicitly, since their number can be quadratic to the number of vertices). Firstly, we notice that in order to find the number of these cut-pairs, it is sufficient, according to Proposition 17, to focus our attention on the lists $M^{-1}(m)$ and $M_p^{-1}(m)$, for every vertex m, and count, for every $c \in M_p^{-1}(m)$, the number of $u \in M^{-1}(m)$ which are proper descendants of c and such that $high(u) < p(c)$. Let $\tilde{U}(c)$ denote the collection of those u, and assume that the lists $M^{-1}(m)$ and $M_p^{-1}(m)$ are sorted in decreasing order. The next Lemma will be useful in analyzing Algorithm 5. It shows that the sets $\tilde{U}(c)$ which overlap, have a nested structure.

▶ **Lemma 18.** *Let m be a vertex, c and c' two elements of $M_p^{-1}(m)$ such that there exists a w in $\tilde{U}(c) \cap \tilde{U}(c')$, and assume, without loss of generality, that c is a descendant of c'. Then $\tilde{U}(c') = \tilde{U}(c) \cup (T[c, c') \cap M^{-1}(m))$. (See Figure 5.)*

Proof. Let u, u' be two vertices in $\tilde{U}(c)$, and assume, without loss of generality, that u is a proper descendant of u'. By the definition of $\tilde{U}(c)$, we have that u' is a proper descendant of c, $M(u) = M_p(c)$, $M(u') = M_p(c)$, and $high(u) < p(c)$. Since, then, $M(u) = M(u')$ and $high(u) < p(c) < c < u'$, it follows from Lemma 6 that $high(u) = high(u')$. This shows that all vertices in $\tilde{U}(c)$ have the same *high* point. In particular, every $u \in \tilde{U}(c)$ has $high(u) = high(w)$. Now, since w is in $\tilde{U}(c')$, it satisfies $high(w) < p(c')$, and so every u in $\tilde{U}(c)$ has $high(u) < p(c')$. Furthermore, every u in $\tilde{U}(c)$ is a proper descendant of c' (since u is a proper descendant of c and c is a descendant of c') and has $M(u) = M_p(c) = M_p(c')$.

Thus we have demonstrated that $\tilde{U}(c) \subseteq \tilde{U}(c')$. Now let $u \in T[c, c') \cap M^{-1}(m)$. Then u is an ancestor of c and a proper descendant of c'. Furthermore, since $M(u) = m = M_p(c) = M(w)$ and u is an ancestor of w and $high(w) < p(c') < c' < u$, it follows from Lemma 6 that $high(u) = high(w)$, and therefore $high(u) < p(c')$. We conclude that u is in $\tilde{U}(c')$, and thus far we have established that $\tilde{U}(c) \cup (T[c, c') \cap M^{-1}(m)) \subseteq \tilde{U}(c')$. To prove the reverse inclusion, let u be a vertex in $\tilde{U}(c')$, but not in $\tilde{U}(c)$. By the definition of $\tilde{U}(c')$, we have $M(u) = M_p(c')$ and $high(u) < p(c')$. Since, then, $M(u) = M_p(c') = M_p(c)$ and $high(u) < p(c') \leq p(c)$, it cannot be the case that u is a proper descendant of c (for otherwise $u \notin \tilde{U}(c)$ is violated). This shows that u is an ancestor of c (since u and c have a common descendant). By the definition of $\tilde{U}(c')$ we have that u is a proper descendant of c' and has $M(u) = M_p(c') = m$. We conclude that u is in $T[c, c') \cap M^{-1}(m)$, and the proof is complete. ◀

Now, the idea to find all $\#\tilde{U}(c)$ is the following. Firstly, we traverse the list $M^{-1}(m)$ in order to find the greatest $u \in M^{-1}(m)$ with the property that u belongs to a set of the form $\tilde{U}(c)$, for a $c \in M_p^{-1}(m)$, and let c be the greatest vertex in $M_p^{-1}(m)$ such that $u \in \tilde{U}(c)$. (Of course, such a u may not exist, in which case there is not much to do.) Lines 6, 8, and 21, of Algorithm 5, form a routine for this search. If the condition in Line 8 is satisfied, u has the above property. Otherwise, we move to the next element of $M^{-1}(m)$, and we continue the search from the current element of $M_p^{-1}(m)$. (It is easy to see why this works: if the greatest c in $M_p^{-1}(m)$ which is a proper ancestor of u does not satisfy $high(u) < p(c)$, then neither does any ancestor of c, and then, if u' is a successor of u in $M^{-1}(m)$, no c' in $M_p^{-1}(m)$ which is a proper descendant of c is a proper ancestor of u'.) Then we find $\#\tilde{U}(c)$ by traversing the list $M^{-1}(m)$ from u, until we reach an ancestor of c. (By Lemma 6, every u' that we encounter in this traversal satisfies $high(u') = high(u)$, since $u' > c > p(c) > high(u)$; therefore, $\tilde{U}(c)$ consists precisely of those vertices u'.) Now let c' be a successor of c in $M_p^{-1}(m)$. If $high(u) < p(c')$, then $u \in \tilde{U}(c) \cap \tilde{U}(c')$, and therefore, by Lemma 18, we have $\tilde{U}(c') = \tilde{U}(c) \cup (T[c, c') \cap M^{-1}(m))$. This explains the counting procedure of Algorithm 5, in Lines 11-18. When we reach a $c' \in M_p^{-1}(m)$ such that $high(u) \geq p(c')$, we have $\tilde{U}(c') \cap \tilde{U}(c) = \emptyset$ (by Lemma 18), and so we need to find the next u with the property that u belongs to a set of the form $\tilde{U}(c)$, for some $c \in M_p^{-1}(m)$, (if such a u exists), and repeat the same process again.

References

1. G. Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, April 1996.
2. G. Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, October 1996.
3. 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 *Proc. 28th ACM-SIAM Symp. on Discrete Algorithms*, (SODA 2017), pages 1900–1918, 2017.
4. D. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacement. *SIAM J. Comput.*, 22(3):587–616, June 1993. doi:10.1137/0222040.
5. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.
6. L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. *ACM Transactions on Algorithms*, 13(1):9:1–9:24, 2016. Announced at SODA 2015. doi:10.1145/2968448.
7. L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. *Information and Computation*, 261:248–264, 2018. ICALP 2015. doi:10.1016/j.ic.2018.02.007.

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

9 L. Georgiadis and E. Kosinas. Linear-time algorithms for computing twinless strong articulation points and related problems. *ArXiv*, abs/2007.03933, 2020. arXiv:2007.03933.

10 C. Gutwenger and P. Mutzel. A linear time implementation of spqr-trees. In Joe Marks, editor, *Graph Drawing*, pages 77–90, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

11 M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, (ICALP 2015), pages 713–724, 2015.

12 J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

13 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. doi:10.1016/j.tcs.2011.11.011.

14 R. Jaberi. 2-edge-twinless blocks, 2019. arXiv:1912.13347.

15 R. Jaberi. Computing 2-twinless blocks, 2019. arXiv:1912.12790.

16 R. Jaberi. Twinless articulation points and some related problems, 2019. arXiv:1912.11799.

17 S. Raghavan. Twinless strongly connected components. In F. B. Alt, M. C. Fu, and B. L. Golden, editors, *Perspectives in Operations Research: Papers in Honor of Saul Gass' 80th Birthday*, pages 285–304. Springer US, Boston, MA, 2006. doi:10.1007/978-0-387-39934-8_17.

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

19 R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

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