Computing the 4-Edge-Connected Components of a Graph in Linear Time

Loukas Georgiadis
Department of Computer Science & Engineering, University of Ioannina, Greece

Giuseppe F. Italiano
LUISS University, Rome, Italy

Evangelos Kosinas
Department of Computer Science & Engineering, University of Ioannina, Greece

Abstract

We present the first linear-time algorithm that computes the 4-edge-connected components of an undirected graph. Hence, we also obtain the first linear-time algorithm for testing 4-edge connectivity. Our results are based on a linear-time algorithm that computes the 3-edge cuts of a 3-edge-connected graph G, and a linear-time procedure that, given the collection of all 3-edge cuts, partitions the vertices of G into the 4-edge-connected components.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Cuts, Edge Connectivity, Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.47

Related Version *Full Version*: `https://arxiv.org/abs/2105.02910` [6]

Funding Research at the University of Ioannina 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. G. F. Italiano is partially supported by MIUR, the Italian Ministry for Education, University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data).

1 Introduction

Let $G = (V, E)$ be a connected undirected graph with m edges and n vertices. An *(edge) cut* of G is a set of edges $S \subseteq E$ such that $G \setminus S$ is not connected. We say that S is a *k-cut* if its cardinality is $|S|= k$. Also, we refer to the 1-cuts as the *bridges* of G. A cut S is *minimal* if no proper subset of S is a cut of G. The *edge connectivity* of G, denoted by $\lambda(G)$, is the minimum cardinality of an edge cut of G. A graph is *k-edge-connected* if $\lambda(G) \geq k$.

A cut S separates two vertices u and v, if u and v lie in different connected components of $G \setminus S$. Vertices u and v are k-edge-connected, denoted by $u \overset{G}{\equiv}_k v$, if there is no $(k-1)$-cut that separates them. By Menger's theorem [16], u and v are k-edge-connected if and only if there are k-edge-disjoint paths between u and v. A *k-edge-connected component* of G is a maximal set $C \subseteq V$ such that there is no $(k-1)$-edge cut in G that disconnects any two vertices $u, v \in C$ (i.e., u and v are in the same connected component of $G \setminus S$ for any $(k-1)$-edge cut S). We can define, analogously, the *vertex cuts* and the *k-vertex-connected components* of G.

Computing and testing the edge connectivity of a graph, as well as its k-edge-connected components, is a classical subject in graph theory, as it is an important notion in several application areas (see, e.g., [19]), that has been extensively studied since the 1970's. It is known how to compute the $(k-1)$-edge cuts, $(k-1)$-vertex cuts, k-edge-connected components and k-vertex-connected components of a graph in linear time for $k \in \{2,3\}$ [5, 10, 18, 21, 25]. The case $k = 4$ has also received significant attention [2, 3, 11, 12]. Unfortunately, none of the previous algorithms achieved linear running time. In particular, Kanevsky and Ramachandran [11] showed how to test whether a graph is 4-vertex-connected in $O(n^2)$ time. Furthermore, Kanevsky et al. [12] gave an $O(m + n\alpha(m,n))$-time algorithm to compute the 4-vertex-connected components of a 3-vertex-connected graph, where α is a functional inverse of Ackermann's function [23]. Using the reduction of Galil and Italiano [5] from edge connectivity to vertex connectivity, the same bounds can be obtained for 4-edge connectivity. Specifically, one can test whether a graph is 4-edge-connected in $O(n^2)$ time, and one can compute the 4-edge-connected components of a 3-edge-connected graph in $O(m + n\alpha(m,n))$ time. Dinitz and Westbrook [3] presented an $O(m + n\log n)$-time algorithm to compute the 4-edge-connected components of a general graph G (i.e., when G is not necessarily 3-edge-connected). Nagamochi and Watanabe [20] gave an $O(m + k^2n^2)$-time algorithm to compute the k-edge-connected components of a graph G, for any integer k. We also note that the edge connectivity of a simple undirected graph can be computed in $O(m\text{polylog}n)$ time, randomized [8, 13] or deterministic [9, 15]. The best current bound is $O(m\log^2 n \log\log^2 n)$, achieved by Henzinger et al. [9] which provided an improved version of the algorithm of Kawarabayashi and Thorup [15].

Our results and techniques. In this paper we present the first linear-time algorithm that computes the 4-edge-connected components of a general graph G, thus resolving a problem that remained open for more than 20 years. Hence, this also implies the first linear-time algorithm for testing 4-edge connectivity. We base our results on the following ideas. First, we extend the framework of Georgiadis and Kosinas [7] for computing 2-edge cuts (as well as mixed cuts consisting of a single vertex and a single edge) of G. Similar to known linear-time algorithms for computing 3-vertex-connected and 3-edge-connected components [10, 25], Georgiadis and Kosinas [7] define various concepts with respect to a depth-first search (DFS) spanning tree of G. We extend this framework by introducing new key parameters that can be computed efficiently and provide characterizations of the various types of 3-edge cuts that may appear in a 3-edge-connected graph. We deal with the general case by dividing G into auxiliary graphs $H_1, \dots, H_\ell$, such that each H_i is 3-edge-connected and corresponds to a different 3-edge-connected component of G. Also, for any two vertices x and y, we have $x \overset{G}{\equiv}_4 y$ if and only if x and y are both in the same auxiliary graph H_i and $x \overset{H_i}{\equiv}_4 y$. Furthermore, this reduction allows us to compute in linear time the number of *minimal* 3*-edge cuts* in a general graph G. Next, in order to compute the 4-edge-connected components in each auxiliary graph H_i, we utilize the fact that a minimum cut of a graph G separates G into two connected components. Hence, we can define the set V_C of the vertices in the connected component of $G \setminus C$ that does not contain a specified root vertex r. We refer to the number of vertices in V_C as the r*-size* of the cut C. Then, we apply a recursive algorithm that successively splits H_i into smaller graphs according to its 3-cuts. When no more splits are possible, the connected components of the final split graph correspond to the 4-edge-connected components of G. We show that we can implement this procedure in linear time by processing the cuts in non-decreasing order with respect to their r-size.

Due to the space constraints we omit several technical details and proofs. They can be found in the full version of the paper which is available at [6].

2 Concepts defined on a DFS-tree structure

Let $G = (V, E)$ be a connected undirected graph, which may have multiple edges. For a set of vertices $S \subseteq V$, the induced subgraph of S, denoted by $G[S]$, is the subgraph of G with vertex set S and edge set $\{e \in E \mid \text{both ends of } e \text{ lie in } S\}$. Let T be the spanning tree of G provided by a depth-first search (DFS) of G [21], with start vertex r. The edges in T are called tree-edges; the edges in $E \setminus T$ are called back-edges, as their endpoints have ancestor-descendant relation in T. 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). From now on, we identify vertices with their preorder number (assigned during the DFS). Thus, v being an ancestor of u in T implies that $v \leq u$. Let $T(v)$ denote the set of descendants of v, and let $ND(v)$ denote the number of descendants of v (i.e. $ND(v) = |T(v)|$). With all $ND(v)$ computed, we can check in constant time whether a vertex u is a descendant of v, since $u \in T(v)$ if and only if $v \leq u$ and $u < v + ND(v)$ [22].

Whenever (x, y) denotes a back-edge, we shall assume that x is a descendant of y. We let $B(v)$ denote the set of back-edges (x, y), where x is a descendant of v and y is a proper ancestor of v. Thus, if we remove the tree-edge $(v, p(v))$, $T(v)$ remains connected to the rest of the graph through the back-edges in $B(v)$. This implies that G is 2-edge-connected if and only if $|B(v)| > 0$, for every $v \neq r$. Furthermore, G is 3-edge-connected only if $|B(v)| > 1$, for every $v \neq r$. We let $b\_count(v)$ denote the number of elements of $B(v)$ (i.e. $b\_count(v) = |B(v)|$). $low(v)$ denotes the lowest y such that there exists a back-edge $(x, y) \in B(v)$. Similarly, $high(v)$ is the highest y such that there exists a back-edge $(x, y) \in B(v)$.

We let $M(v)$ denote the nearest common ancestor of all x for which there exists a back-edge $(x, y) \in B(v)$. Note that $M(v)$ is a descendant of v. Let m be a vertex and $v_1, \ldots, v_k$ be all the vertices with $M(v_1) = \ldots = M(v_k) = m$, sorted in decreasing order. (Observe that v_{i+1} is an ancestor of v_i, for every $i \in \{1, \ldots, k-1\}$, since m is a common descendant of all $v_1, \ldots, v_k$.) Then we have $M^{-1}(m) = \{v_1, \ldots, v_k\}$, and we define $nextM(v_i) := v_{i+1}$, for every $i \in \{1, \ldots, k-1\}$, and $lastM(v_i) := v_k$, for every $i \in \{1, \ldots, k\}$. Thus, for every vertex v, $nextM(v)$ is the successor of v in the decreasingly sorted list $M^{-1}(M(v))$, and $lastM(v)$ is the lowest element in $M^{-1}(M(v))$.

The following two facts have been proved in [7].

▶ **Fact 1.** *All $ND(v)$, $b\_count(v)$, $M(v)$, $low(v)$ and $high(v)$ can be computed in total linear-time, for all vertices v.*

▶ **Fact 2.** *$B(u) = B(v) \Leftrightarrow M(u) = M(v)$, and $high(u) = high(v) \Leftrightarrow M(u) = M(v)$ and $b\_count(u) = b\_count(v)$.*

Furthermore, [7] implies the following characterization of a 3-edge-connected graph.

▶ **Fact 3.** *G is 3-edge-connected if and only if $|B(v)| > 1$, for every $v \neq r$, and $B(v) \neq B(u)$, for every pair of vertices u and v, $u \neq v$.*

Now let us provide some extensions of those concepts that will be needed for our purposes. Assume that G is 3-edge-connected, and let $v \neq r$ be a vertex of G. By Fact 3, $b\_count(v) > 1$, and therefore there are at least two back-edges in $B(v)$. Thus, there is at least one back-edge $(x, y) \in B(v)$ such that $y = low(v)$. We let $low1(v)$ denote y, and $low1D(v)$ denote x. In other words, $low1(v)$ is the *low* point of v, and $low1D(v)$ is a descendant of v which is connected with a back-edge to its *low* point. (Notice, however, that $low1D(v)$ is not uniquely determined.) Similarly, we let $highD(v)$ denote a descendant of v which is connected with a back-edge to the *high* point of v. Then we define $low2(v) := min\{y' \mid \exists(x',y') \in B(v) \setminus \{(low1D(v), low1(v))\}\}$, and we let $low2D(v)$ denote a descendant of v which is connected with a back-edge to $low2(v)$. Thus, if $v \neq r$, we have that $(low1D(v), low(v))$ and $(low2D(v), low2(v))$ are two distinct back-edges in $B(v)$. It is easy to compute all $low1(v)$, $low1D(v)$, $low2(v)$ and $low2D(v)$ during the DFS. It is also easy to extend the algorithm for the computation of *high* points in order to compute all $highD(v)$. (We refer to [6] for the details.)

We let $l(v)$ denote the lowest y for which there exists a back-edge (v, y), or v if no such back-edge exists. Thus, $low(v) \leq l(v)$. Now let $c_1, \dots, c_k$ be the children of v sorted in non-decreasing order w.r.t. their *low* point. Then we call c_1 the *low1* child of v, and c_2 the *low2* child of v. (Of course, the *low1* and *low2* children of v are not uniquely determined after a DFS on G, since we may have $low(c_1) = low(c_2)$.) We let $\tilde{M}(v)$ denote the nearest common ancestor of all x for which there exists a back-edge $(x, y) \in B(v)$ with x a proper descendant of $M(v)$. Formally, $\tilde{M}(v) := \text{nca}\{x \mid \exists(x,y) \in B(v) \text{ and } x \neq M(v)\}$. If the set $\{x \mid \exists(x,y) \in B(v) \text{ and } x \neq M(v)\}$ is empty, we leave $\tilde{M}(v)$ undefined. We also define $M_{low1}(v)$ as the nearest common ancestor of all x for which there exists a back-edge $(x, y) \in B(v)$ with x being a descendant of the *low1* child of $M(v)$, and also define $M_{low2}(v)$ as the nearest common ancestor of all x for which there exists a back-edge $(x, y) \in B(v)$ with x a descendant of the *low2* child of $M(v)$. Formally, $M_{low1}(v) := \text{nca}\{x \mid \exists(x,y) \in B(v)$ and x is a descendant of the *low1* child of $M(v)\}$ and $M_{low2}(v) := \text{nca}\{x \mid \exists(x,y) \in B(v)$ and x is a descendant of the *low2* child of $M(v)\}$. If the set in the formal definition of $M_{low1}(v)$ (resp. $M_{low2}(v)$) is empty, we leave $M_{low1}(v)$ (resp. $M_{low2}(v)$) undefined.

The following list summarizes the concepts that we use on a DFS-tree; they are defined for all $v \neq r$. (For an illustration, see Figure 1.)

- $B(v) := \{(x,y) \mid x$ is a descendant of v and y is a proper ancestor of $v\}$.
- $l(v) := min(\{y \mid \exists(v,y) \in B(v)\} \cup \{v\})$.
- $low(v) := min\{y \mid \exists(x,y) \in B(v)\}$.
- $low1(v) := low(v)$.
- $low1D(v) :=$ a vertex x such that $(x, low1(v)) \in B(v)$.
- $low2(v) := min\{y' \mid \exists(x',y') \in B(v) \setminus \{(low1D(v), low1(v))\}\}$.
- $low2D(v) :=$ a vertex x such that $(x, low2(v)) \in B(v)$.
- $high(v) := max\{y \mid \exists(x,y) \in B(v)\}$.
- $highD(v) :=$ a vertex x such that $(x, high(v)) \in B(v)$.
- $M(v) := nca\{x \mid \exists(x,y) \in B(v)\}$.
- $\tilde{M}(v) := nca\{x \mid \exists(x,y) \in B(v)$ and x is a proper descendant of $M(v)\}$.
- $M_{low1}(v) := nca\{x \mid \exists(x,y) \in B(v)$ and x is a descendant of the *low1* child of $M(v)\}$.
- $M_{low2}(v) := nca\{x \mid \exists(x,y) \in B(v)$ and x is a descendant of the *low2* child of $M(v)\}$.

In Section 2.1 of the full paper [6], we show how to compute all these concepts in linear time.

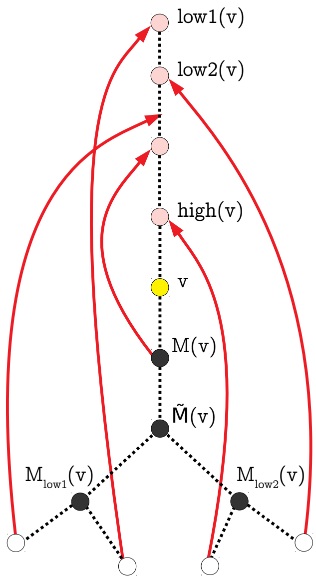

Figure 1 An illustration of some concepts defined on a DFS-tree. The red arrows correspond to the back-edges in $B(v)$. Dashed lines correspond to tree-paths.

3 Computing the 3-cuts of a 3-edge-connected graph

In this section we present a linear-time algorithm that computes all the 3-edge-cuts of a 3-edge-connected graph $G = (V, E)$. It is well-known that the number of the 3-edge-cuts of G is $O(n)$ [19] (e.g., it follows from the definition of the cactus graph [1, 14]), but we provide an independent proof of this fact. Then, in Section 4.1, we show how to extend this algorithm so that it can also count the number of minimal 3-edge-cuts of a general graph. Note that there can be $O(n^3)$ such cuts [2].

Our method is to classify the 3-cuts on the DFS-tree T in a way that allows us to compute them efficiently. If $\{e_1, e_2, e_3\}$ is a 3-cut, we can initially distinguish three cases w.l.o.g.: either e_1 is a tree-edge and both e_2 and e_3 are back-edges, or e_1 and e_2 are two tree-edges and e_3 is a back-edge, or e_1, e_2 and e_3 is a triplet of tree-edges. Then, we divide those cases in subcases based on the concepts we have introduced in the previous section. Figure 2 gives a general overview of the cases we will handle in some detail in the following sections.

3.1 One tree-edge and two back-edges

The following lemma characterizes all 3-cuts consisting of a tree-edge and two back-edges.

▶ **Lemma 4.** *Let $\{(u, p(u)), e, e'\}$ be a 3-cut such that e and e' are back-edges. Then $B(u) = \{e, e'\}$. Conversely, if for a vertex $u \neq r$ we have $B(u) = \{e, e'\}$ where e and e' are back-edges, then $\{(u, p(u)), e, e'\}$ is a 3-cut.*

Thus, to compute all the 3-cuts of this type, we have to find all $u \neq r$ with $b\_count(u) = 2$. For every such u, there are two back-edges e_1, e_2 such that $B(u) = \{e_1, e_2\}$, and so, w.l.o.g., we have $e_1 = (low1D(u), low1(u))$ and $e_2 = (low2D(u), low2(u))$. Then we mark $\{(u, p(u)), e_1, e_2\}$ as a 3-cut.

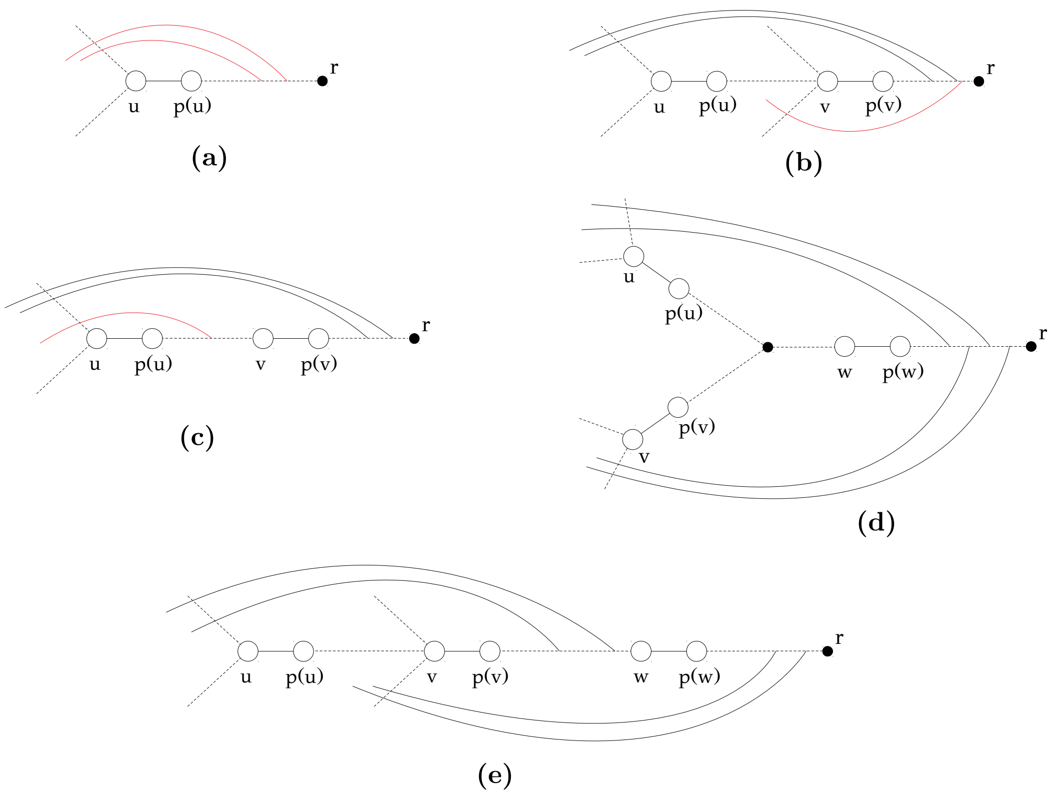

Figure 2 The types of 3-cuts with respect to a DFS-tree. **(a)** One tree-edge $(u, p(u))$ and two back-edges. **(b)** Two tree-edges $(u, p(u))$ and $(v, p(v))$, where u is a descendant of v, and one-back edge in $B(v) \setminus B(u)$. **(c)** Two tree-edges $(u, p(u))$ and $(v, p(v))$, where u is a descendant of v, and one-back edge in $B(u) \setminus B(v)$. **(d)** Three tree-edges $(u, p(u))$, $(v, p(v))$ and $(w, p(w))$, where w is an ancestor of u and v, but u and v are not related as ancestor and descendant. **(d)** Three tree-edges $(u, p(u))$, $(v, p(v))$ and $(w, p(w))$, where u is a descendant of v and v is a descendant of w.

3.2 Two tree-edges and one back-edge

In the case of 3-cuts consisting of two tree-edges and a back-edge, we have the following.

▶ **Lemma 5.** *Let $\{(u, p(u)), (v, p(v)), e\}$ be a 3-cut such that e is a back-edge. Then u and v are related as ancestor and descendant.*

▶ **Proposition 6.** *Let $\{(u, p(u)), (v, p(v)), e\}$ be a 3-cut, where e is a back-edge. Then, either (1) $B(v) = B(u) \sqcup \{e\}$ or (2) $B(u) = B(v) \sqcup \{e\}$. Conversely, if there exists a back-edge e such that (1) or (2) is true, then $\{(u, p(u)), (v, p(v)), e\}$ is a 3-cut.*

We let $V(u)$, for a $u \neq r$, be the set of all v that are ancestors of u such that $B(v) = B(u) \sqcup \{e\}$, for a back-edge e. We also let $U(v)$, for a $v \neq r$, be the set of all u that are descendants of v such that $B(u) = B(v) \sqcup \{e\}$, for a back-edge e. Then, for every 3-cut $\{(u, p(u)), (v, p(v)), e\}$, where e is a back-edge, Proposition 6 implies that either $u \in V(u)$ or $v \in U(v)$.

The following two lemmata imply that the number of 3-cuts consisting of two tree-edges and a back-edge is $O(n)$.

▶ **Lemma 7.** *Let v, v' be two distinct vertices. Then $V(u) \cap V(u') = \emptyset$.*

▶ **Lemma 8.** *Let u, u' be two distinct vertices. Then $U(v) \cap U(v') = \emptyset$.*

Now, every $v \in V(u)$ has either $\tilde{M}(v) = M(u)$, or $M_{low1}(v) = M(u)$, or $M_{low2}(v) = M(u)$, and u is the lowest vertex which is greater than v such that $\tilde{M}(v) = M(u)$, or $M_{low1}(v) = M(u)$, or $M_{low2}(v) = M(u)$, respectively. This suggests a method to compute, for every vertex v, the unique u (if it exists) such that $v \in V(u)$. We process all vertices v, and for every v that we process we have to find the lowest element u of $M^{-1}(x)$ which is greater than v, for every $x \in \{\tilde{M}(v), M_{low1}(v), M_{low2}(v)\}$, and check whether $v \in V(u)$. To perform this efficiently, we have the lists $M^{-1}(x)$, for every vertex x, sorted in decreasing order, and we process the vertices in a bottom-up fashion. Then, for every v that we process, and every $x \in \{\tilde{M}(v), M_{low1}(v), M_{low2}(v)\}$, we search for the lowest u in $M^{-1}(x)$ which is greater than v, by traversing the list $M^{-1}(x)$ starting from the last element of $M^{-1}(x)$ that we considered, which is being stored in a variable $currentVertex[x]$. This is to avoid traversing $M^{-1}(x)$ from the beginning each time we process a vertex v. We can check in constant time whether $v \in V(u)$ thanks to the following lemma.

▶ **Lemma 9.** *Let v be an ancestor of u such that either $\tilde{M}(v) = M(u)$, or $M_{low1}(v) = M(u)$, or $M_{low2}(v) = M(u)$, and let $m = \tilde{M}(v)$, or $m = M_{low1}(v)$, or $m = M_{low2}(v)$, depending on whether $\tilde{M}(v) = M(u)$, or $M_{low1}(v) = M(u)$, or $M_{low2}(v) = M(u)$, respectively. Then, $v \in V(u)$ if and only if u is the lowest element in $M^{-1}(m)$ which is greater than v and such that $high(u) < v$ and $b\_count(v) = b\_count(u) + 1$.*

Finally, for a $v \in V(u)$, we can immediately identify the back-edge (x, y) with $B(v) = B(u) \sqcup \{(x, y)\}$, since we have $x = \tilde{M}(v)$ and $y = l(\tilde{M}(v))$, or $x = M_{low1}(v)$ and $y = l(M_{low1}(v))$, or $x = M_{low2}(v)$ and $y = l(M_{low2}(v))$, depending on whether $\tilde{M}(v) = M(u)$, or $M_{low1}(v) = M(u)$, or $M_{low2}(v) = M(u)$, respectively. Algorithm 1 shows how we can compute all 3-cuts of the form $\{(u, p(u)), (v, p(v)), e\}$, with $B(v) = B(u) \sqcup \{e\}$.

We can use a similar method to compute the 3-cuts of the form $\{(u, p(u)), (v, p(v)), e\}$, with $B(u) = B(v) \sqcup \{e\}$.

3.3 Three tree-edges

The case of 3-cuts consisting of three tree-edges is more involved and is subdivided into several subcases. The following is generally true for all such 3-cuts.

▶ **Lemma 10.** *Let $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ be a 3-cut, and assume, without loss of generality, that $w < min\{v, u\}$. Then w is an ancestor of both u and v.*

First we treat the case that u and v are not related as ancestor and descendant. We have the following characterizations of the 3-cuts of this type.

▶ **Proposition 11.** *Let u and v be two vertices which are not related as ancestor and descendant, and let w be an ancestor of both u and v. Then $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a 3-cut if and only if $B(w) = B(u) \sqcup B(v)$.*

▶ **Lemma 12.** *Let u and v be two vertices which are not related as ancestor and descendant, and let w be an ancestor of both u and v. Then $B(w) = B(u) \sqcup B(v)$ if and only if: $M_{low1}(w) = M(u)$ and $M_{low2}(w) = M(v)$ (or $M_{low1}(w) = M(v)$ and $M_{low2}(w) = M(u)$), and $high(u) < w$, $high(v) < w$, and $b\_count(w) = b\_count(u) + b\_count(v)$.*

Then, as an implication of the following lemma, we see than the pair $\{u, v\}$ with the property that u and v are descendants of w, but are not related as ancestor and descendant, and $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a 3-cut, is uniquely determined by w (and thus the number of those 3-cuts in $O(n)$).

Algorithm 1 Find all 3-cuts $\{(u,p(u)),(v,p(v)),e\}$, where u is a descendant of v and $B(v) = B(u) \sqcup \{e\}$, for a back-edge e.

initialize an array *currentVertex* with n entries
// $m = \tilde{M}(v)$
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $v \leftarrow n$ *to* $v = 1$ **do**
  $m \leftarrow \tilde{M}(v)$
  if $m = \emptyset$ **then continue**
  // *find the lowest* $u \in M^{-1}(m)$ *which is greater than* v
  $u \leftarrow currentVertex[m]$, $prev \leftarrow u$
  while $nextM(u) \neq \emptyset$ ***and*** $nextM(u) > v$ **do** $prev \leftarrow u$, $u \leftarrow nextM(u)$
  $currentVertex[m] \leftarrow prev$
  // *check the condition in Lemma 9*
  if $high(u) < v$ ***and*** $b\_count(v) = b\_count(u) + 1$ **then**
    mark the triplet $\{(u,p(u)),(v,p(v)),(M(v),l(M(v)))\}$
  end
end
// $m = M_{low1}(v)$
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $v \leftarrow n$ *to* $v = 1$ **do**
  $m \leftarrow M_{low1}(v)$
  if $m = \emptyset$ ***or*** $l(M(v)) < v$ **then continue**
  // *find the lowest* $u \in M^{-1}(m)$ *which is greater than* v
  $u \leftarrow currentVertex[m]$, $prev \leftarrow u$
  while $nextM(u) \neq \emptyset$ ***and*** $nextM(u) > v$ **do** $prev \leftarrow u$, $u \leftarrow nextM(u)$
  $currentVertex[m] \leftarrow prev$
  // *check the condition in Lemma 9*
  if $high(u) < v$ ***and*** $b\_count(v) = b\_count(u) + 1$ **then**
    mark the triplet $\{(u,p(u)),(v,p(v)),(M_{low2}(v),l(M_{low2}(v)))\}$
  end
end
// $m = M_{low2}(v)$
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $v \leftarrow n$ *to* $v = 1$ **do**
  $m \leftarrow M_{low2}(v)$
  if $m = \emptyset$ ***or*** $l(M(v)) < v$ **then continue**
  // *find the lowest* $u \in M^{-1}(m)$ *which is greater than* v
  $u \leftarrow currentVertex[m]$, $prev \leftarrow u$
  while $nextM(u) \neq \emptyset$ ***and*** $nextM(u) > v$ **do** $prev \leftarrow u$, $u \leftarrow nextM(u)$
  $currentVertex[m] \leftarrow prev$
  // *check the condition in Lemma 9*
  if $high(u) < v$ ***and*** $b\_count(v) = b\_count(u) + 1$ **then**
    mark the triplet $\{(u,p(u)),(v,p(v)),(M_{low1}(v),l(M_{low1}(v)))\}$
  end
end

▶ **Lemma 13.** *Let $\{(u,p(u)),(v,p(v)),(w,p(w))\}$ be a 3-cut such that u and v are not related as ancestor and descendant and let w is an ancestor of both u and v. By Proposition 11 and Lemma 12, we may assume w.l.o.g. that $M_{low1}(w) = M(u)$ and $M_{low2}(w) = M(v)$, and let $m_1 = M_{low1}(w)$ and $m_2 = M_{low2}(w)$. Then u is the lowest vertex in $M^{-1}(m_1)$ which is greater than w, and v is the lowest vertex in $M^{-1}(m_2)$ which is greater that w.*

This suggests a method to compute those u, v (if they exist) for a particular w. We simply have to find the lowest u in $M^{-1}(M_{low1}(w))$ which is greater than w, and the lowest v in $M^{-1}(M_{low2}(w))$ which is greater than w, and, if they exist, check whether $high(u) < w$, $high(v) < w$, and $b\_count(w) = b\_count(u) + b\_count(v)$. To perform this search efficiently, we have the lists $M^{-1}(x)$, for every vertex x, sorted in decreasing order, we process the vertices w in a bottom-up fashion, and we keep stored in a variable $currentVertex[x]$ the most recent element of $M^{-1}(x)$ that we considered. Algorithm 2 is an implementation of this procedure, for computing all 3-cuts of this type.

Algorithm 2 Find all 3-cuts $\{(u,p(u)),(v,p(v)),(w,p(w))\}$, where w is an ancestor of u and v, and u, v are not related as ancestor and descendant.

---

initialize an array *currentVertex* with n entries
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $w \leftarrow n$ *to* $w = 1$ **do**
  $m_1 \leftarrow M_{low1}(w)$, $m_2 \leftarrow M_{low2}(w)$
  if $m_1 = \emptyset$ ***or*** $m_2 = \emptyset$ **then continue**
  // *find the lowest* u *in* $M^{-1}(m_1)$ *which is greater than* w
  $u \leftarrow currentVertex[m_1]$
  while $nextM(u) \neq \emptyset$ ***and*** $nextM(u) > w$ **do** $u \leftarrow nextM(u)$
  $currentVertex[m_1] \leftarrow u$
  // *find the lowest* v *in* $M^{-1}(m_2)$ *which is greater than* w
  $v \leftarrow currentVertex[m_2]$
  while $nextM(v) \neq \emptyset$ ***and*** $nextM(v) > w$ **do** $v \leftarrow nextM(v)$
  $currentVertex[m_2] \leftarrow v$
  // *check the condition in Lemma 12*
  if $b\_count(w) = b\_count(u) + b\_count(v)$ ***and*** $high(u) < w$ ***and*** $high(v) < w$ **then**
    mark the triplet $\{(u,p(u)),(v,p(v)),(w,p(w))\}$
  end
end

Now we treat the case that u and v are related as ancestor and descendant, and assume w.l.o.g. that u is a descendant of v. We have the following characterization of those 3-cuts.

▶ **Proposition 14.** *Let u, v, w be three vertices such that u is a descendant of v and v is a descendant of w. Then $\{(u,p(u)),(v,p(v)),(w,p(w))\}$ is a 3-cut if and only if $B(v) = B(u) \sqcup B(w)$.*

This implies that $M(v)$ is an ancestor of $M(w)$, and we distinguish two cases, depending on whether $M(v)$ is a proper ancestor of $M(w)$. In the first case we have the following.

▶ **Lemma 15.** *Let u be a descendant of v and v a descendant of w, and $M(v) \neq M(w)$. Then, $\{(u,p(u)),(v,p(v)),(w,p(w))\}$ is a 3-cut if and only if: $M(w) = M_{low1}(v)$ and w is the greatest vertex with $M(w) = M_{low1}(v)$ which is lower than v, $M(u) = M_{low2}(v)$ and u*

is the lowest vertex with $M(u) = M_{low2}(v)$*,* $high(u) < v$ *and* $b\_count(v) = b\_count(u) + b\_count(w)$*.*

This shows that the number of such 3-cuts is $O(n)$, and it immediately suggests an algorithm to compute them efficiently (i.e. we work as in Algorithm 2).

Now, if $M(v) = M(w)$, we distinguish two cases, depending on whether $w = nextM(v)$ or $w < nextM(v)$. In any case, there is a unique u which is a descendant of v such that $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a 3-cut, since by Proposition 14 we have $B(u) = B(v) \setminus B(w)$, and we have assumed that the graph is 3-edge-connected (and so the result follows from Fact 3). The next lemma shows that u satisfies $high(u) = high(v)$ and $nextM(u) = \emptyset$.

▶ **Lemma 16.** *Let* u, v, w *be three vertices such that* u *is a descendant of* v*,* v *is a descendant of* w*, and* $M(v) = M(w)$*. Then,* $B(v) = B(u) \sqcup B(w)$ *only if* $high(u) = high(v)$ *and* $nextM(u) = \emptyset$*.*

Thus, for every vertex h, we seek in the decreasingly sorted list $high^{-1}(h)$ pairs of vertices $\{u, v\}$ that have the potential to provide a 3-cut of the form $\{(u, p(u)), (v, p(v)), (w, p(w))\}$, where u is a descendant of v, v is a descendant of w, and $M(v) = M(w)$. In the case $w = nextM(v)$ we have the following:

▶ **Proposition 17.** *Let* $h = high(v)$ *and* $w = nextM(v)$*, and suppose that the list* $high^{-1}(h)$ *is sorted in decreasing order. Then,* u *is a descendant of* v *such that* $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ *is a 3-cut if and only if* u *is a predecessor of* v *in* $high^{-1}(h)$*,* $nextM(u) = \emptyset$*,* $low(u) \geq w$*,* $b\_count(u) = b\_count(v) - b\_count(w)$*, and all elements of* $high^{-1}(h)$ *between* u *and* v *are ancestors of* u*.*

Thus we traverse the decreasingly sorted list $high^{-1}(h)$ from its first element, and we keep consecutive entries that are related as ancestor and descendant in a stack. When we meet a $v \in high^{-1}(h)$ that has $nextM(v) \neq \emptyset$, we simply check whether there is an entry u in the stack that satisfies $nextM(u) = \emptyset$, $low(u) \geq nextM(v)$ and $b\_count(u) = b\_count(v) - b\_count(nextM(v))$, whence we immediately infer that $\{(u, p(u)), (v, p(v)), (nextM(v), p(nextM(v)))\}$ is a 3-cut. This procedure is shown in Algorithm 3.

The case $w < nextM(v)$ is more complicated, since for a particular $v \in high^{-1}(h)$ there may be many pairs $\{u, w\}$ such that u is a descendant of v and w is a proper ancestor of $nextM(v)$ with $M(w) = M(v)$, and $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a 3-cut. Thus, we keep in a stack $stackU[v]$, for every $v \in high^{-1}(h)$, a set of $u \in high^{-1}(h)$ with the potential to provide such a 3-cut. In particular, let $\tilde{U}(v)$, for a vertex v, denote the set of all descendants u of v with the property that there exists a w with $M(w) = M(v)$ and $w < nextM(v)$, such that $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a 3-cut. Then the stacks $stackU[v]$ are filled with Algorithm 4, and satisfy the following:

▶ **Lemma 18.** *For every vertex* v *we have* $\tilde{U}(v) \subseteq stackU(v)$*, and for every* $v' \neq v$ *we have* $stackU(v) \cap stackU(v') = \emptyset$*. Furthermore, if* u' *is a successor of* u *in* $stackU(v)$*, then* u' *is an ancestor of* u*.*

This implies that the number of 3-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w))\}$, where u is a descendant of v and w is a proper ancestor of $nextM(v)$ with $M(w) = M(v)$, is $O(n)$. The next lemma provides a criterion to determine whether a $u \in stackU(v)$ is in $\tilde{U}(v)$, and a way to compute the vertex w with $M(w) = M(v)$ and $w < nextM(v)$, such that $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a 3-cut.

Algorithm 3 Find all 3-cuts $\{(u,p(u)),(v,p(v)),(w,p(w))\}$, where u is a descendant of v and $w = nextM(v)$.

```
initialize an array A with m entries (where m is the number of edges of the graph)
initialize a stack S
sort the elements of every list high^{-1}(h), for every vertex h, in decreasing order
foreach vertex h do
    u ← first element of high^{-1}(h)
    while u ≠ ∅ do
        z ← next element of high^{-1}(h)
        if z = ∅ then break
        if z is not an ancestor of u then
            while S is not empty do
                u' ← S.pop()
                A[b_count(u')] ← ∅
            end
        end
        if nextM(z) = ∅ then
            S.push(z)
            A[b_count(z)] ← z
        end
        else if nextM(z) ≠ ∅ then
            v ← z, w ← nextM(v)
            if A[b_count(v) − b_count(w)] ≠ ∅ then
                u ← A[b_count(v) − b_count(w)]
                if low(u) ≥ w then
                    mark the triplet {(u,p(u)),(v,p(v)),(w,p(w))}
                end
            end
        end
        u ← z
    end
end
```

▶ **Lemma 19.** *Let u be a vertex in $stackU[v]$ and w a proper ancestor of v such that $M(w) = M(v)$. Then, if $\{(u,p(u)),(v,p(v)),(w,p(w))\}$ is a 3-cut, we have that $b\_count(v) = b\_count(u) + b\_count(w)$ and w is the greatest element of $M^{-1}(M(v))$ such that $w \le low(u)$. Conversely, if $b\_count(v) = b\_count(u) + b\_count(w)$ and $w \le low(u)$, then $\{(u,p(u)),(v,p(v)),(w,p(w))\}$ is a 3-cut.*

Thus, for every $u \in stackU[v]$, we have to find the greatest $w \in M^{-1}(M(v))$ such that $w \le low(u)$ and $b\_count(v) = b\_count(u) + b\_count(w)$. To do this efficiently, we take advantage of the fact that the stack $stackU[v]$ has been filled in such a way, that the successor of every $u \in stackU[v]$ is an ancestor of u, and of the fact that $low(u') \le low(u)$, for every ancestor u' of u. Then we have the lists $M^{-1}(x)$, for every vertex x, sorted in decreasing order, and we process the vertices v from lowest to highest. For every $u \in stackU[v]$, we traverse the list $M^{-1}(M(v))$ in order to find the greatest $w \in M^{-1}(M(v))$ that has $w \le low(u)$.

Algorithm 4 Fill all stacks $stackU[v]$, for all vertices v.

```
initialize a stack S
sort the elements of every list high^{-1}(h), for every vertex h, in decreasing order
foreach vertex v do initialize a stack stackU[v]
foreach vertex h do
    u ← first element of high^{-1}(h)
    while u ≠ ∅ do
        z ← next element of high^{-1}(h)
        if z = ∅ then break
        if z is not an ancestor of u then
            pop out all elements from S
        end
        if nextM(z) = ∅ then
            S.push(z)
        end
        else if nextM(z) ≠ ∅ then
            while low(S.top()) < lastM(z) do S.pop()
            while low(S.top()) < nextM(z) do
                u ← S.pop()
                stackU[v].push(u)
            end
        end
        u ← z
    end
end
```

Using a path-compression method, we can bypass segments of $M^{-1}(M(v))$ that we have already visited. This procedure is shown in Algorithm 5. A detailed proof of correctness and linear complexity is given in the full version of this paper [6].

Algorithm 5 Find all 3-cuts $\{(u, p(u)), (v, p(v)), (w, p(w))\}$, where u is a descendant of v, v is a descendant of w with $M(v) = M(w)$, and $w \neq nextM(v)$.

```
initialize an array lowestW with n entries
foreach vertex v do lowestW[v] ← nextM(v)
for v ← 1 to v ← n do
    while stackU[v].top() ≠ ∅ do
        u ← stackU[v].pop()
        w ← lowestW[v]
        while w > low(u) do w ← lowestW[w]
        lowestW[v] ← w
        if b_count(v) = b_count(u) + b_count(w) then
            mark the triplet {(u, p(u)), (v, p(v)), (w, p(w))}
        end
    end
end
```

4 Computing the 4-edge-connected components in linear time

Now we consider how to compute the 4-edge-connected components of an undirected graph G in linear time. First, we reduce this problem to the computation of the 4-edge-connected components of a collection of auxiliary 3-edge-connected graphs.

4.1 Reduction to the 3-edge-connected case

Given a (general) undirected graph G, we execute the following steps:

1. Compute the connected components of G.
2. For each connected component, we compute the 2-edge-connected components which are subgraphs of G.
3. For each 2-edge-connected component, we compute its 3-edge-connected components $C_1, \ldots, C_\ell$.
4. For each 3-edge-connected component C_i, we compute a 3-edge-connected auxiliary graph H_i, such that for any two vertices x and y, we have $x \overset{G}{\equiv}_4 y$ if and only if x and y are both in the same auxiliary graph H_i and $x \overset{H_i}{\equiv}_4 y$.
5. Finally, we compute the 4-edge-connected components of each H_i.

Steps 1–3 take overall linear time [21, 25]. We describe step 5 in the next section, so it remains to give the details of step 4. Let H be a 2-edge-connected component (subgraph) of G. We can construct a compact representation of the 2-cuts of H, which allows us to compute its 3-edge-connected components $C_1, \ldots, C_\ell$ in linear time [7, 25]. Now, since the collection $\{C_1, \ldots, C_\ell\}$ constitutes a partition of the vertex set of H, we can form the quotient graph Q of H by shrinking each C_i into a single node. Graph Q has the structure of a tree of cycles [2]; in other words, Q is connected and every edge of Q belongs to a unique cycle. Let (C_i, C_j) and (C_i, C_k) be two edges of Q which belong to the same cycle. Then (C_i, C_j) and (C_i, C_k) correspond to two edges (x, y) and (x', y') of G, with $x, x' \in C_i$. If $x \neq x'$, we add a virtual edge (x, x') to $G[C_i]$. (The idea is to attach (x, x') to $G[C_i]$ as a substitute for the cycle of Q which contains (C_i, C_j) and (C_i, C_k).) Now let $\bar{C_i}$ be the graph $G[C_i]$ plus all those virtual edges. Then $\bar{C_i}$ is 3-edge-connected and its 4-edge-connected components are precisely those of G that are contained in C_i [2]. Thus we can compute the 4-edge-connected components of G by computing the 4-edge-connected components of the graphs $\bar{C_1}, \ldots, \bar{C_\ell}$ (which can easily be constructed in total linear time). Since every $\bar{C_i}$ is 3-edge-connected, we can apply Algorithm 6 of the following section to compute its 4-edge-connected components in linear time. Finally, we define the multiplicity $m(e)$ of an edge $e \in \bar{C_i}$ as follows: if e is virtual, $m(e)$ is the number of edges of the cycle of Q which corresponds to e; otherwise, $m(e)$ is 1. Then, the number of minimal 3-cuts of H is given by the sum of all $m(e_1) \cdot m(e_2) \cdot m(e_3)$, for every 3-cut $\{e_1, e_2, e_3\}$ of $\bar{C_i}$, for every $i \in \{1, \ldots, l\}$ [2]. Since the 3-cuts of every $\bar{C_i}$ can be computed in linear time, the minimal 3-cuts of H can also be computed within the same time bound.

4.2 Computing the 4-edge-connected components of a 3-edge-connected graph

Now we describe how to compute the 4-edge-connected components of a 3-edge-connected graph G in linear time. Let r be a distinguished vertex of G, and let C be a minimum cut of G. By removing C from G, G becomes disconnected into two connected components. We let V_C denote the connected component of $G \setminus C$ that does not contain r, and we refer to the number of vertices of V_C as the *r-size* of the cut C. (Of course, these notions are relative to r.)

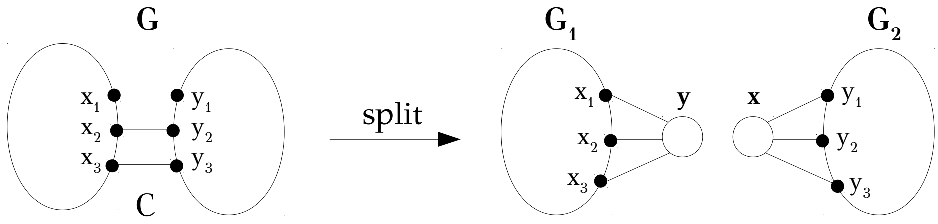

Figure 3 $C = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ is a 3-cut of G, with $\{x_1, x_2, x_3\}$ and $\{y_1, y_2, y_3\}$ lying in different connected components of $G \setminus C$. The split operation of G at C consists of the removal the edges of C from G, and the introduction of two new nodes x, y, and six virtual edges $(x_1, y), (x_2, y), (x_3, y), (x, y_1), (x, y_2), (x, y_3)$. Now, the split graph is made of two connected components, G_1 and G_2. Every 3-cut $C' \neq C$ of G (or more precisely: a 3-cut that corresponds to C') lies entirely within G_1 or G_2. Conversely, every 3-cut of either G_1 or G_2 corresponds to a 3-cut of G. Thus, every 4-edge-connected component of G lies entirely within G_1 or G_2.

Let $G = (V, E)$ be a 3-edge-connected graph, and let $\mathcal{C}$ be the collection of the 3-cuts of G. If the collection $\mathcal{C}$ is empty, then G is 4-edge-connected, and V is the only 4-edge-connected component of G. Otherwise, let $C \in \mathcal{C}$ be a 3-cut of G. By removing C from G, G is separated into two connected components, and every 4-edge-connected component of G lies entirely within a connected component of $G \setminus C$. This observation suggests a recursive algorithm for computing the 4-edge-connected components of G, by successively splitting G into smaller graphs according to its 3-cuts. Thus, we start with a 3-cut C of G, and we perform the splitting operation shown in Figure 3. Then we take another 3-cut C' of G and we perform the same splitting operation on the part which contains (the corresponding 3-cut of) C'. We repeat this process until we have considered every 3-cut of G. When no more splits are possible, the connected components of the final split graph correspond (by ignoring the newly introduced vertices) to the 4-edge-connected components of G.

To implement this procedure in linear time, we must take care of two things. First, whenever we consider a 3-cut C of G, we have to be able to know which ends of the edges of C belong to the same connected component of $G \setminus C$. And second, since an edge e of a 3-cut of the original graph may correspond to two virtual edges of the split graph, we have to be able to know which is the virtual edge that corresponds to e. We tackle both these problems by locating the 3-cuts of G on a DFS-tree T of G rooted at r, and by processing them in increasing order with respect to their r-size. By locating a 3-cut $C \in \mathcal{C}$ on T we can answer in $O(1)$ time which ends of the edges of C belong to the same connected component of $G \setminus C$. And then, by processing the 3-cuts of G in increasing order with respect to their size, we ensure that (the 3-cut that corresponds to) a 3-cut $C \in \mathcal{C}$ that we process lies in the split part of G that contains r.

Now, due to the analysis of the preceding sections, we can distinguish the following types of 3-cuts on a DFS-tree T (see also Figure 2):

- (I) $\{(v, p(v)), (x_1, y_1), (x_2, y_2)\}$, where (x_1, y_1) and (x_2, y_2) are back-edges.
- (IIa) $\{(u, p(u)), (v, p(v)), (x, y)\}$, where u is a descendant of v and $(x, y) \in B(v)$.
- (IIb) $\{(u, p(u)), (v, p(v)), (x, y)\}$, where u is a descendant of v and $(x, y) \in B(u)$.
- (III) $\{(u, p(u)), (v, p(v)), (w, p(w))\}$, where w is an ancestor of both u and v, but u, v are not related as ancestor and descendant.
- (IV) $\{(u, p(u)), (v, p(v)), (w, p(w))\}$, where u is a descendant of v and v is a descendant of w.

Let r be the root of T. Then, for every 3-cut $C \in \mathcal{C}$, V_C is either $T(v)$, or $T(v) \setminus T(u)$, or $T(w) \setminus (T(u) \cup T(v))$, or $T(u) \cup (T(w) \setminus T(v))$, depending on whether C is of type (I), (II), (III), or (IV), respectively. Thus we can immediately calculate the size of C and the ends of its edges that lie in V_C. In particular, the size of C is either $ND(v)$, or $ND(v) - ND(u)$, or $ND(w) - ND(u) - ND(v)$, or $ND(u) + ND(w) - ND(v)$, depending on whether it is of type (I), (II), (III), or (IV), respectively; V_C contains either $\{v, x_1, x_2\}$, or $\{p(u), v, x\}$, or $\{p(u), v, y\}$, or $\{p(u), p(v), w\}$, or $\{u, p(v), w\}$, depending on whether C is of type (I), (IIa), (IIb), (III), or (IV), respectively.

Algorithm 6 shows how we can compute the 4-edge-connected components of G in linear time, by repeatedly splitting G into smaller graphs according to its 3-cuts. When we process a 3-cut C of G, we have to find the edges of the split graph that correspond to those of C, in order to delete them and replace them with (new) virtual edges. That is why we use the symbol v', for a vertex $v \in V$, to denote a vertex that corresponds to v in the split graph. (Initially, we set $v' \leftarrow v$.) Now, if (x, y) is an edge of C with $x \in V_C$, the edge of the split graph corresponding to (x, y) is (x', y'). Then we add two new vertices v_C and $\tilde{v_C}$ to G, and the virtual edges $(x', \tilde{v_C})$ and (v_C, y'). Finally, we let x correspond to v_C, and so we set $x' \leftarrow v_C$. This is sufficient, since we process the 3-cuts of G in increasing order with respect to their size, and so the next time we meet the edge (x, y) in a 3-cut, we can be certain that it corresponds to (v_C, y').

Algorithm 6 Compute the 4-edge-connected components of a 3-edge-connected graph $G = (V, E)$.

Find the collection $\mathcal{C}$ of the 3-cuts of G
Locate and classify the 3-cuts of G on a DFS-tree of G rooted at r
For every $C \in \mathcal{C}$, calculate *size*(C) (relative to r)
Sort $\mathcal{C}$ in increasing order w.r.t. the *size* of its elements
foreach $v \in V$ **do** Set $v' \leftarrow v$
foreach $C = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\} \in \mathcal{C}$ **do**
  Find the ends of the edges of C that lie in V_C // *Let those ends be* x_1*,*x_2 *and* x_3
  Remove the edges (x'_1, y'_1),(x'_2, y'_2),(x'_3, y'_3) from G
  Introduce two new vertices v_C and $\tilde{v_C}$ to G
  Add the edges $(x'_1, \tilde{v_C})$,$(x'_2, \tilde{v_C})$,$(x'_3, \tilde{v_C})$,(v_C, y'_1),(v_C, y'_2),(v_C, y'_3) to G
  Set $x'_1 \leftarrow v_C$, $x'_2 \leftarrow v_C$, $x'_3 \leftarrow v_C$
end
Output the connected components of G, ignoring the newly introduced vertices

Final Remarks

Independently from our work, Nadara et al. [17] also presented a linear-time algorithm for computing the 4-edge-connected components of a graph. Both our algorithm and the algorithm of [17] require the use of the static tree disjoint-set-union data structure of Gabow and Tarjan [4] to achieve linear running time. Also, similar to our algorithm, the main part in the algorithm of Nadara et al. is the computation of the 3-edge cuts of a 3-edge-connected graph G. Both algorithms operate on a depth-first search tree of G, with start vertex r, and distinguish 3 types of cuts $C = \{e_1, e_2, e_3\}$, depending on the number of tree edges in C. These cases are handled in a different manner in [17]. In particular, Nadara et al. [17] show that the case where C consists of three tree edges can be reduced, in linear time, to the other two cases. We note that by applying this idea in our framework, we are able to avoid the use of *high* points. (We achieve this by modifying the algorithm that identifies 3-edge cuts consisting of two tree edges, described in Section 3.2.) This way, we obtain a linear-time algorithm that does not require the Gabow-Tarjan disjoint-set-union data structure, and thus is implementable in the pointer machine computation model [24].

References

1 E. A. Dinitz, A. V. Karzanov, and M. V. Lomonosov. On the structure of a family of minimal weighted cuts in a graph. *Studies in Discrete Optimization (in Russian)*, page 290–306, 1976.

2 Y. Dinitz. The 3-edge-components and a structural description of all 3-edge-cuts in a graph. In *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '92, page 145–157, Berlin, Heidelberg, 1992. Springer-Verlag.

3 Y. Dinitz and J. Westbrook. Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica*, 20:242–276, 1998. doi:10.1007/PL00009195.

4 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.

5 Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, 1991. doi:10.1145/122413.122416.

6 L. Georgiadis, G. F. Italiano, and E. Kosinas. Computing the 4-edge-connected components of a graph in linear time. *CoRR*, abs/2105.02910, 2021. arXiv:2105.02910.

7 L. Georgiadis and E. Kosinas. Linear-Time Algorithms for Computing Twinless Strong Articulation Points and Related Problems. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation (ISAAC 2020)*, volume 181 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ISAAC.2020.38.

8 M. Ghaffari, K. Nowicki, and M. Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 1260–1279, USA, 2020. Society for Industrial and Applied Mathematics.

9 M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020.

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

11 A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. *Journal of Computer and System Sciences*, 42(3):288–306, 1991. doi:10.1016/0022-0000(91)90004-O.

12 A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen. On-line maintenance of the four-connected components of a graph. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science (FOCS 1991)*, pages 793–801, 1991. doi:10.1109/SFCS.1991.185451.

13 D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, January 2000. doi:10.1145/331605.331608.

14 D. R. Karger and D. Panigrahi. A near-linear time algorithm for constructing a cactus representation of minimum cuts. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, page 246–255, USA, 2009. Society for Industrial and Applied Mathematics.

15 K.-I. Kawarabayashi and M. Thorup. Deterministic edge connectivity in near-linear time. *Journal of the ACM*, 66(1), December 2018. doi:10.1145/3274663.

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

17 W. Nadara, M. Radecki, M. Smulewicz, and M. Sokolowski. Determining 4-edge-connected components in linear time. In *Proc. 29th European Symposium on Algorithms*, 2021.

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

19 H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity.* Cambridge University Press, 2008. 1st edition.

20 H. Nagamochi and T. Watanabe. Computing k-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 76(4):513–517, 1993.

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

22 R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

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

24 R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–27, 1979.

25 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.