Computing the 4-Edge-Connected Components of a Graph: An Experimental Study

LOUKAS GEORGIADIS, Department of Computer Science & Engineering, University of Ioannina, Ioannina, Greece
GIUSEPPE F. ITALIANO, LUISS University, Rome, Italy
EVANGELOS KOSINAS, Department of Computer Science and Engineering, University of Ioannina, Ioannina, Greece

The notions of edge-cuts and k-edge-connected components are fundamental in graph theory with numerous practical applications. Very recently, the first linear-time algorithms for computing all the 3-edge-cuts and the 4-edge-connected components of a graph have been introduced. In this article, we present carefully engineered implementations of these algorithms and evaluate their efficiency in practice, by performing a thorough empirical study using both real-world graphs taken from a variety of application areas, as well as artificial graphs. To the best of our knowledge, this is the first experimental study for these problems, which highlights the merits and weaknesses of each technique. Furthermore, we present an improved algorithm for computing the 4-edge-connected components of an undirected graph in linear time. The new algorithm uses only elementary data structures, and is implementable in the pointer-machine model of computation.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Theory of computation → Graph algorithms analysis**;

Additional Key Words and Phrases: Connectivity Cuts, Edge Connectivity, Graph Algorithms

ACM Reference format:
Loukas Georgiadis, Giuseppe F. Italiano, and Evangelos Kosinas. 2025. Computing the 4-Edge-Connected Components of a Graph: An Experimental Study. *ACM Trans. Algor.* 22, 1, Article 1 (October 2025), 34 pages. https://doi.org/10.1145/3757919

An extended abstract of this work appeared in the *Proceedings of the 30th Annual European Symposium on Algorithms,* Article 60, 1–16 (2022).
L. Georgiadis was 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. Italiano was partially supported by the Italian Ministry of University and Research under PRIN Project n. 2022TS4Y3N—EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data. E. Kosinas was supported by the project "Dioni: Computing Infrastructure for Big-Data Processing and Analysis." (MIS No. 5047222) which is implemented under the Action "Reinforcement of the Research and Innovation Infrastructure," funded by the Operational Programme "Competitiveness, Entrepreneurship and Innovation" (NSRF 2014–2020) and co-financed by Greece and the European Union (European Regional Development Fund).
Authors' Contact Information: Loukas Georgiadis, Department of Computer Science & Engineering, University of Ioannina, Ioannina, Greece; e-mail: loukas@cs.uoi.gr; Giuseppe F. Italiano, LUISS University, Rome, Italy; e-mail: gitaliano@luiss.it; Evangelos Kosinas (corresponding author), Department of Computer Science and Engineering, University of Ioannina, Ioannina, Greece; e-mail: ekosinas@cs.uoi.gr.

ACM 1549-6333/2025/10-ART1
https://doi.org/10.1145/3757919

1 Introduction

Determining or testing the edge connectivity of a graph $G = (V, E)$, as well as computing notions of connected components or subgraphs, is a classic subject in graph theory, motivated by several application areas (see, e.g., [27]), that has been extensively studied since the 1970's. 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 $|S| = k$. 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 [24], 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). One can also consider the *maximal k-edge-connected* subgraphs of a graph G [1, 3, 9, 31, 32].[1] These are the (inclusion-wise) maximal (w.r.t. the vertex set) subgraphs of G that are k-edge-connected. It is easy to see that the vertex sets of the maximal k-edge-connected subgraphs of G provide a partition of its vertex set. Although the k-edge-connected components and the maximal k-edge-connected subgraphs are defined similarly, these concepts do not necessarily coincide when $k \geq 3$. Thus, in general, the vertex set of a maximal k-edge-connected subgraph is a strict subset of a k-edge-connected component.

We can define, analogously, the *vertex cuts* and the *k-vertex-connected components* of G. Until recently, it was 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 $k \in \{2, 3\}$ [11, 12, 16, 26, 33, 34, 37, 38]. The case $k = 4$ has also received significant attention [5, 6, 17, 18], but none of the previous algorithms achieved linear running time. In particular, Kanevsky and Ramachandran [17] showed how to test whether a graph is 4-vertex-connected in $O(n^2)$ time. Furthermore, Kanevsky et al. [18] 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 [36]. Using the reduction of Galil and Italiano [11] 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 $O(m+n\alpha(m, n))$ 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 [6] 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 [28] gave an $O(m + k^2n^2)$-time algorithm to compute the k-edge-connected components of a graph, for any integer k.

Recently, linear-time algorithms for computing the 4-edge-connected components of an undirected graph were presented in [13, 25]. Specifically, Nadara et al. [25] gave two linear-time algorithms, a simple randomized and a more complicated deterministic algorithm. Georgiadis et al. [13] also presented a linear-time deterministic algorithm that requires less machinery but a more detailed analysis. The main part in these algorithms is the computation of the 3-edge-cuts of a 3-edge-connected graph G. The algorithms operate on a **depth-first search (DFS)** tree T of G [34], with start vertex r, and compute three types of 3-edge-cuts $C = \{e_1, e_2, e_3\}$, depending on the number of tree-edges in C. We refer to a cut C that consists of t tree-edges of T as a *type-t cut of G*. The challenging cases are when C is a type-2 or type-3 cut. To handle type-2 cuts in

[1] We note that some of those papers use different terminology for the maximal k-edge-connected subgraphs, and some even call them k-edge-connected components.

linear time, both NRSS [25] and GIK [13] require the use of the static tree **disjoint-set-union (DSU)** data structure of Gabow and Tarjan [10], which is quite sophisticated and not amenable to simple implementations. Moreover, the deterministic algorithm of NRSS also employs a linear-time algorithm for computing offline nearest common ancestors [15]. NRSS [25] provided an elegant way to handle type-3 cuts. Specifically, they showed that computing all type-3 cuts can be reduced, in linear time, to computing type-1 and type-2 cuts, by contracting the edges of $G \setminus E(T)$. The algorithm of GIK [13], on the other hand, operates directly on the original graph, but requires a more involved case analysis and, as for type-2 cuts, uses the Gabow-Tarjan DSU data structure.

In more recent work, Kosinas [22] extended the framework of [13] and presented the first linear-time algorithm for computing the 5-edge-connected components of an undirected graph. Furthermore, in a significant recent breakthrough, Korhonen [21] introduced an algorithm with running time $k^{O(k^2)}m$ for computing the k-edge-connected components of an undirected graph, which is linear in the number of edges for any fixed k.

Our Contribution. We present an improved version of the algorithm of GIK [13] for identifying type-2 cuts, that uses only simple data structures. The resulting algorithm relies only on basic properties of DFS [34], and on parameters carefully defined on the structure of a DFS spanning tree (see Section 3.3). As a consequence, it is simple to describe and to implement, and it does not require the power of the RAM model of computation, thus implying the following *new results*:

Theorem 1.1. *The 3-edge-cuts of an undirected graph can be computed in linear time on a pointer machine.*

Corollary 1.2. *The 4-edge-connected components of an undirected graph can be computed in linear time on a pointer machine.*

We note that it is also possible to obtain Theorem 1.1 and Corollary 1.2 by combining the framework of GIK [13] with the linear-time pointer-machine algorithm for interval analysis in [2]. Still, the latter is based on much more complicated techniques, and thus it is unlikely to be competitive with our new algorithm in practice.

Next, we consider the practical performance of these algorithms. We provide carefully engineered implementations of the randomized algorithm of NRSS [25], the deterministic algorithm of GIK [13], as well as our new linear-time pointer-machine algorithm. Then, we conduct a thorough empirical study to highlight the merits and weaknesses of each technique. In particular, our experimental evaluation addresses the following questions:

- How well does the randomized algorithm perform with respect to its deterministic counterparts?
- How efficient is the graph contraction technique in practice?
- How does our new linear-time pointer-machine algorithm compare against the linear-time RAM algorithms?
- How fast can we compute the 4-edge-connected components of a graph compared to computing the k-edge-connected components for $k < 4$?

Furthermore, by utilizing the linear-time algorithms for computing all 3-cuts of a 3-edge-connected graph, we can implement $O(nm)$-time algorithms for computing the maximal 4-edge-connected subgraphs of a graph with n vertices and m edges. This enables us to address the following questions:

- How fast can we compute the maximal 4-edge-connected subgraphs in real-world graphs by using a simple cut-removal approach, and how much slower is this computation compared to that of the 4-edge-connected components?
- How does the structure of the maximal 4-edge-connected subgraphs in real-world graphs compare to that of the 4-edge-connected components?

Organization of the Article. The remainder of the article is organized as follows. In Section 2, we outline the basic steps that we need to perform to compute the 4-edge-connected components of a graph. This gives a reduction to the computation of the 4-edge-connected components of a collection of auxiliary 3-edge-connected graphs. To that end, we need to compute the 3-cuts of a 3-edge-connected graph. We describe several algorithms for this task in Section 3. Then, in Section 4 we provide an extensive empirical analysis using both real-world graphs taken from a variety of application areas, as well as artificial graphs. Finally, in Appendix A we present some experimental results on the number of k-edge-connected components in random graphs, for $k \in \{1, 2, 3, 4\}$.

2 Linear-Time Algorithms for Computing the 4-Edge-Connected Components

Let $G = (V, E)$ be the input graph, which may have multiple edges. To compute the 4-edge-connected components of G, we can perform the following steps:

(1) Compute the connected components of G.
(2) For each connected component, compute the subgraphs induced by its 2-edge-connected components (which are 2-edge-connected subgraphs of G).
(3) For each 2-edge-connected component, compute its 3-edge-connected components $C_1, \ldots, C_\ell$.
(4) For each 3-edge-connected component C_i, 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, compute the 4-edge-connected components of each H_i.

For Steps 1–3, we can compute the 1-edge- and 2-edge-cuts, and the 2-edge- and 3-edge-connected components a graph in linear time by [11, 12, 16, 26, 33, 34, 37, 38]. It can be easily demonstrated that the (subgraphs induced by the) k-edge-connected components of G, for $k = 1, 2$, have the same k'-edge-connected components as G, for all $k' > k$. However, the analogous property does not hold for $k = 3$. Thus we use the construction provided by Dinitz [5], which extends the 3-edge-connected components of G, by adding some extra edges to them, so that the resulting auxiliary graphs have the same k'-edge-connected components as G, for all $k' > 3$. To perform Step 4, we use the fact that we can construct a compact representation of the 2-cuts of any 2-edge-connected component H of G, which allows us to compute its 3-edge-connected components $C_1, \ldots, C_\ell$ in linear time [14, 38]. We let $G[C_i]$ denote the subgraph of G that is induced by the vertices in C_i. By shrinking each 3-edge-connected component C_i of H into a single node, we obtain a quotient graph Q of H which has the structure of a tree of cycles [5], i.e., 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 $G[C_i]$, which acts as a substitute for the cycle of Q that contains (C_i, C_j) and (C_i, C_k). Now let H_i be the graph $G[C_i]$ plus all those virtual edges. Then H_i is 3-edge-connected and its 4-edge-connected components are precisely those of G that are contained in C_i [5]. Thus we can compute the 4-edge-connected components of G by computing the 4-edge-connected components of the graphs H_i.

Hence, we have reduced the problem of computing the 4-edge-connected components of a (general) graph, to the computation of the 4-edge-connected components of a collection of auxiliary 3-edge-connected graphs. So, from now on, we let G be a 3-edge-connected graph. We can compute its 4-edge-connected components by successively splitting G 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 refer to [13] for the details.) It remains to describe how to compute the 3-edge-cuts of a 3-edge-connected graph G.

We note that Tsin [39] recently presented a linear-time algorithm that directly constructs the auxiliary subgraphs corresponding to the 3-edge-connected components of a connected multigraph (i.e., the auxiliary graphs H_i from Step 4) in a single pass, without first computing the 2- and 3-edge-connected components. While this approach may enable a more efficient computation of the 4-edge-connected components in practice, in our experiments we follow the five-step procedure described above in order to facilitate a consistent comparison of the running times required to compute the k-edge-connected components for all $1 \leq k \leq 4$.

3 Computing the 3-Cuts of a 3-Edge-Connected Graph

In this section we give an overview of the algorithms of NRSS [25] and GIK [13] for computing the 3-edge-cuts of a 3-edge-connected graph G. Then, we also present our new linear-time pointer-machine algorithm. Throughout this section, we assume that $G = (V, E)$ is a 3-edge-connected multigraph with n vertices and m edges. It is well-known that the number of the 3-edge-cuts of G is $O(n)$ [27], which also follows from the definition of the cactus graph [4, 19]). See also [13, 25] for an independent proof of this fact.

For a graph H, we let $V(H)$ and $E(H)$ denote the set of vertices and edges of H, respectively. If H is a subgraph of G then $G \setminus E(H)$ denotes the graph that results from G after deleting the edges of H.

3.1 DFS and Related Notions

Here we introduce some concepts and parameters that are used in the algorithms, defined with respect to a DFS spanning tree. Let T be the spanning tree of G provided by a DFS of G [34], with start vertex r. A vertex u *is an ancestor of a vertex* v (v *is a descendant of* u) in T if the tree path from r to v contains u. Thus, we consider a vertex to be both an ancestor and a descendant of itself. The edges in T are called *tree-edges*; the edges in $E(G) \setminus E(T)$ are called *back-edges*, as their endpoints have ancestor-descendant relation in T. 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). We identify vertices with their preorder number assigned during the DFS. Thus, if v is an ancestor of u in T, then $v \leq u$. Let $T(v)$ denote the set of descendants of v, and let $ND(v) = |T(v)|$ denote the number of descendants of v. Then, vertex u is a descendant of v (i.e., $u \in T(v)$) if and only if $v \leq u < v + ND(v)$ [35].

Whenever (x, y) denotes a back-edge, we shall assume that x is a descendant of y. We say that a back-edge (x, y) leaps over a vertex v if x is a descendant of v and y is a proper ancestor of v. We let $B(v)$, for a vertex $v \neq r$, denote the set of all back-edges that leap over v, and we let $bcount(v) = |B(v)|$ denote the number of elements of $B(v)$. 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 $B(v)$, which implies the following property:

Property 3.1 ([13]). *A connected graph G is 2-edge-connected if and only if $bcount(v) > 0$, for every $v \neq r$. Furthermore, G is 3-edge-connected only if $bcount(v) > 1$, for every $v \neq r$, and $B(u) \neq B(v)$ for every two distinct vertices u and v.*

This suggests that the sets $B(v)$ can be used to determine various connectivity relations in graphs. In fact, we can consider many useful notions extracted from those sets by considering the distribution of the ends of the back-edges that are contained in them. First, consider the lower ends of the back-edges in $B(v)$. We define the *low* point of vertex v, denoted by $low(v)$, as the minimum vertex y such that there exists a back-edge $(x, y) \in B(v)$. Formally, $low(v) := min\{y \mid \exists (x, y) \in B(v)\}$. We also let $lowD(v)$ be x. That is, $lowD(v)$ is a descendant of v from which stems a back-edge that provides $low(v)$. (Notice that $lowD(v)$ is not uniquely determined.) We denote the back-edge $(lowD(v), low(v))$ as MinUp(v). Similarly, we define the *high* point of v, denoted by $high(v)$, as the maximum vertex y such that there exists a back-edge $(x, y) \in B(v)$. Formally, $high(v) := max\{y \mid \exists (x, y) \in B(v)\}$. We also let $highD(v)$ be x. That is, $highD(v)$ is a descendant of v from which stems a back-edge that provides $high(v)$. (Again, $highD(v)$ is not uniquely determined.) We denote the back-edge $(highD(v), high(v))$ as MaxUp(v). Finally we let $l_1(v)$ be the minimum y such that there exists a back-edge of the form (v, y) (or v, if no such back-edge exists). Formally, $l_1(v) := min(\{y \mid \exists (v, y) \in B(v)\} \cup \{v\})$. Furthermore, we define $l_2(v) := min(\{y \mid \exists (v, y) \in B(v) \setminus \{(v, l_1(v)\}\} \cup \{v\})$.

Now we consider the higher ends of the back-edges in $B(v)$. First, we let MinDn(v) (resp. MaxDn(v)) denote a back-edge in $B(v)$ with minimum (resp. maximum) higher end. We then define the *maximum* point $M(v)$ of v as the maximum vertex z such that $T(z)$ contains the higher ends of all back-edges in $B(v)$. In other words, $M(v)$ is the nearest common ancestor of all x for which there exists a back-edge $(x, y) \in B(v)$. See Figure 1. (Clearly, $M(v)$ 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 $prevM(v_i) := v_{i-1}$, for every $i \in \{2, \ldots, k\}$. Thus, for every vertex v, $nextM(v)$ is the successor of v in the decreasingly sorted list $M^{-1}(M(v))$, and $prevM(v)$ is the predecessor of v in the decreasingly sorted list $M^{-1}(M(v))$.

Now let v be a vertex and let $u_1, \ldots, u_k$ be the children of v sorted in non-decreasing order w.r.t. their *low* point. We let $c_i(v)$ be u_i if $i \in \{1, \ldots, k\}$, and $\perp$ if $i > k$. (Note that $c_i(v)$ is not uniquely determined, since some children of v may have the same *low* point.) Then we call $c_1(v)$ the *low1* child of v, and $c_2(v)$ the *low2* child of v. 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 $M(v)$. We leave $\tilde{M}(v)$ undefined if no such proper descendant x of $M(v)$ exists. 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 $M(v)$. We leave $M_{low1}(v)$ (resp., $M_{low2}(v)$) undefined if no such proper descendant x of the *low1* (resp., *low2*) child of $M(v)$ exists.

We note that it is easy to compute all $l_1(v)$, $l_2(v)$, $low(v)$, $lowD(v)$, and $bcount(v)$ during the DFS. In particular, the notion of low points plays central role in classic algorithms for computing the biconnected components [34], the triconnected components [16], and the 3-edge-connected components [11, 16, 26, 37] of a graph. (Hopcroft and Tarjan [16] also use a concept of high points, which, however, is different from ours.) Furthermore, all $M(v)$, $\tilde{M}(v)$, $M_{low1}(v)$ and $M_{low2}(v)$, for all vertices v for which they are defined, can be computed in linear-time in pointer machines [13]. For the computation of all $high(v)$ (and $highD(v)$), [13] and [25] gave a linear-time algorithm that uses

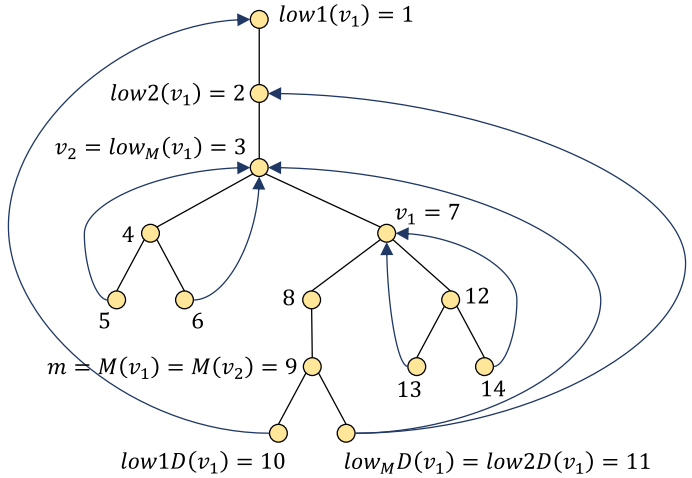

Fig. 1. An illustration of the concepts defined on a DFS spanning tree of an undirected graph. Vertices are numbered in DFS order and back-edges are shown directed from descendant to ancestor in the DFS tree. Vertices v_1 and v_2 have $M(v_1) = M(v_2) = m$, hence $\{v_1, v_2\} \subseteq M^{-1}(m)$, $nextM(v_1) = v_2$, and $prevM(v_2) = v_1$. Moreover, $(11, 3) \notin B(v_2)$, so $low_M(v_1) = 3$.

the static tree DSU data structure of Gabow and Tarjan [10], and thus their computation depends on the power of the RAM model of computation. We note, however, that all $high(v)$ and $highD(v)$, and the respective MinUp(v) edges, can be computed in linear time in pointer machines (see the Interval Analysis problem in [2]), but the algorithm is not trivial to be implemented, and it is not expected to be efficient in practice.

3.2 Randomized Algorithm of NRSS

We give an overview of the randomized linear-time algorithm of Nadara et al. [25]. Let $H : E \mapsto 2^E$ be a function defined as follows. If e is a back-edge, $H(e) = \{e\}$. Otherwise, $H(e) = B(u)$, for the unique vertex $u \neq r$ such that $e = (u, p(u))$. Now NRSS provide an elegant characterization of 3-cuts: a triple of edges $\{e_1, e_2, e_3\}$ is a 3-cut if and only if $H(e_1) \oplus H(e_2) \oplus H(e_3) = \emptyset$ (where $\oplus$ denotes the symmetric set difference). Then we get a useful equivalent characterization of 3-cuts by representing the sets $H(e)$ as bit-vectors. To be precise, we consider the composition of $\chi : 2^E \mapsto \{0, 1\}^E$ (the characteristic function on E) with $H : E \mapsto 2^E$; then we have that $\{e_1, e_2, e_3\}$ is a 3-cut if and only if $\chi_{H(e_1)} \oplus \chi_{H(e_2)} \oplus \chi_{H(e_3)} = 0$ (where $\oplus$ here denotes the logical XOR function). Thus, if for an edge e we have a candidate e' that may provide a 3-cut of the form $\{e, e', e''\}$, for a third edge e'', then e'' is uniquely determined by the expression $\chi_{H(e)} \oplus \chi_{H(e')}$.

Since we cannot compute all $\chi_{H(e)}$, for all edges e, in linear time, the idea of NRSS is to compress the bits of those vectors into a fixed number of RAM words. (We assume that a RAM cell can store $O(\log m)$ bits.) In particular, we select $b = \lceil 3 \lg m \rceil$ random bits to represent the characteristic function of $H(e)$, for every back-edge e, and we call this value $CH(e)$. (We may select a bigger multiple of $\lg m$ for higher precision; however, $\lceil 3 \lg m \rceil$ is enough for all practical purposes.) Then $CH(e)$, for a tree-edge e, corresponds to $\bigoplus_{e' \in H(e)} CH(e')$.

Now, for type-1 and type-2 cuts, we can find good candidate edges that may yield 3-cuts. Specifically, let $\{(u, p(u)), e, e'\}$ be a type-1 cut (i.e., e and e' are back-edges). Then one of e, e' is the back-edge that provides the *low* point of u. Thus, without loss of generality, we may assume that $e = \text{MinUp}(u)$. Then we can retrieve e' with high probability by determining $CH^{-1}(CH((u, p(u))) \oplus$

$CH(e)$). Now let $\{(u, p(u)), (v, p(v)), e\}$ be a type-2 cut (where e is a back-edge). In this case u and v are related as ancestor and descendant, and thus we may assume, without loss of generality, that u is a descendant of v. Then we have that either e leaps over u but not over v, or that e leaps over v but not over u. In the first case, e is the back-edge that provides the *high* point of u (that is, $e = \text{MaxUp}(u)$). Thus we can retrieve $(v, p(v))$ with high probability by determining $CH^{-1}(CH((u, p(u))) \oplus CH(\text{MaxUp}(u)))$. In the second case, e is either $\text{MinDn}(v)$ or $\text{MaxDn}(v)$. Thus we can retrieve $(u, p(u))$ with high probability from $CH((v, p(v)) \oplus \text{MinDn}(v))$ or $CH((v, p(v)) \oplus \text{MaxDn}(v))$. NRSS provide linear-time algorithms for computing all $\text{MaxUp}(v)$, $\text{MinDn}(v)$ and $\text{MaxDn}(v)$, for all vertices $v \neq r$, using the static-tree DSU data structure of Gabow and Tarjan [10]. (The inverses CH^{-1} can be computed efficiently by using (e.g., hash tables.)

Finally, for type-3 cuts, NRSS provided a simple linear-time reduction to computing type-1 and type-2 cuts. First, compute the connected components $C_1, \ldots, C_k$ of $G \setminus E(T)$. Then, form a reduced graph G' by contracting each component C_i into a single vertex c_i. In this construction, we maintain parallel edges between two vertices but remove self-loops. Then, recursively compute the type-1 and type-2 cuts of G'. Assuming that G is 3-edge-connected, it is easy to observe that G' satisfies the following properties: (i) G' is also 3-edge-connected, and (ii) $|E(G')| \leq \frac{2}{3}|E(G)|$. The latter inequality follows from the fact that the minimum vertex degree in G is 3, and that $|E(G')| \leq |V(G)|$ due to the contractions. Then, the total running time spent during the recursive calls is bounded by $O(m)$.

3.3 Deterministic Algorithm of GIK

Here we provide an overview of the deterministic linear-time algorithm of Georgiadis et al. [13]. The idea is to distinguish various types of 3-cuts on a DFS tree of the graph, and then provide an algorithm specifically adapted for each particular case. The classification of GIK is heavily guided by some DFS parameters that can be extracted from the sets $B(v)$ of the back-edges that leap over a vertex v, for $v \neq r$, without computing the sets $B(v)$ explicitly.

First, let $\{(u, p(u)), e, e'\}$ be a type-1 cut. Then we obviously have $B(u) = \{e, e'\}$. Thus, in order to identify all those cuts, we need to have computed, for every vertex $u \neq r$, two back-edges that leap over u. We take as one of them a back-edge e that provides the *low* point of u. To get one more back-edge, we extend the definition of *low* points: we let $low2(u)$ be the lowest lower end of all back-edges in $B(u) \setminus \{e\}$, and let $low2D(u)$ be a descendant of u such that $(low2D(u), low2(u)) \in B(u) \setminus \{e\}$. Let $low1(u) := low(u)$ and $low1D(u) := lowD(u)$. (Refer to Figure 1.) Then, for every $u \neq r$, we have that $\{(low1D(u), low1(u)), (low2D(u), low2(u))\} \subseteq B(u)$, and these two back-edges form a 3-cut with $(u, p(u))$ if and only if $bcount(u) = 2$. We note that all back-edges $(low2D(v), low2(v))$ can be computed easily during the DFS, and thus all type-1 cuts can be computed in linear time in a pointer machine without using sophisticated data structures.

Now let $\{(u, p(u)), (v, p(v)), e\}$ be a type-2 cut (where e is a back-edge). Then we have that u and v are related as ancestor and descendant, and we may assume in what follows, without loss of generality, that u is a descendant of v. Then we have that either $B(u) = B(v) \sqcup \{e\}$ or $B(v) = B(u) \sqcup \{e\}$. In either case, we can exploit the similarity of the sets $B(u)$ and $B(v)$ to find good candidate pairs $\{u, v\}$ that may provide type-2 cuts of the form $\{(u, p(u)), (v, p(v)), e\}$, for a back-edge e. Take the case $B(u) = B(v) \sqcup \{e\}$ as an example. Then e must be the back-edge that provides the *high* point of u, and so $e = (highD(u), high(u))$. Depending on the location of $highD(u)$, we can relate $M(v)$ with a maximum point of u. Specifically, if $highD(u)$ is a descendant of $M(v)$, then $M(v) = M(u)$, and v is the greatest ancestor of u with this property (i.e., $v = nextM(u)$). If $highD(u)$ is an ancestor of $M(v)$, then $M(v) = \tilde{M}(u)$ and v is the greatest ancestor of u with this property. Finally, if $highD(u)$ is not related as ancestor or descendant with $M(v)$, then $M(v) = M_{low1}(u)$, and

v is again the greatest ancestor of u with this property. These considerations are the basis of an efficient algorithm for determining, for a vertex u, a proper ancestor v of u, and a back-edge e, that may yield a type-2 cut of the form $\{(u, p(u)), (v, p(v)), e\}$. (In fact, once such a v has been found, we only have to check whether $bcount(u) = bcount(v) + 1$ to establish the existence of this cut.) The case $B(v) = B(u) \sqcup \{e\}$ can be handled in a similar manner. We refer to [13] for more details.

Finally, with a much more involved subdivision of type-3 cuts into more cases, and with extensive use of DFS parameters extracted from the sets of leaping back-edges, GIK provide linear-time algorithms to identify all those cuts. These algorithms use the *high* points in a way that seems difficult to avoid them.

In the next section we will show how to handle the cases of type-2 cuts without the use of *high* points, thus providing the first linear-time pointer-machine algorithm for determining all 3-cuts of 3-edge-connected graphs.

3.4 A New Linear-Time Pointer-Machine Algorithm

Our goal is to provide a method to compute all 3-cuts in linear time without using the *high* points. In the algorithm of [13], this is the only parameter that depends on the power of RAM to be computed in linear time (and it is also basically used in the algorithm of [25]). Actually, we notice that this parameter can be computed in pointer machines in linear time [2]. However, this is achieved with the use of sophisticated techniques, that are neither easy to implement, nor expected to run efficiently in practice. Here we show that we can completely avoid the computation of the *high* points. Instead, we use other DFS-based concepts, that can be easily computed, and that we can use in order to provide alternative criteria for testing the existence of 3-cuts, and retrieve the edges that constitute them.

We achieve this through the following three improvements over the GIK algorithm. First, we introduce two new parameters, $low_M D(v)$ and $low_M(v)$, that yield the back-edge $(highD(u), high(u))$ that is needed in the case $B(u) = B(v) \sqcup \{e\}$, $M(u) = M(v)$ (see Section 3.3). Second, we replace all conditions provided by GIK for determining type-2 cuts with equivalent ones that avoid the invocation and the computation of the *high* points. And finally, we use the idea of NRSS to deal with type-3 cuts by reducing them to type-1 and type-2 cuts.

Let v be a vertex such that $nextM(v) \neq \perp$. Then we have $B(nextM(v)) \subset B(v)$. Thus we can define $low_M(v)$ as the lowest lower end of all back-edges in $B(v) \setminus B(nextM(v))$, and we let $low_M D(v)$ be a vertex such that $(low_M D(v), low_M(v))$ is a back-edge in $B(v) \setminus B(nextM(v))$. Formally, we have:

- $low_M(v) := min\{y \mid \exists (x, y) \in B(v) \setminus B(nextM(v))\}$.
- $low_M D(v) :=$ a vertex x such that $(x, low_M(v)) \in B(v)$.

Now we describe how to compute $(low_M D(v), low_M(v))$ for every vertex v such that $nextM(v) \neq \perp$. To do this efficiently, we process the vertices in a bottom-up fashion. For every vertex v that we process, we check whether $u = prevM(v) \neq \perp$. If that is the case, then $low_M(u)$ is defined and it lies on the simple tree path $T(u, v]$. Thus we descend the path $T(u, v]$, starting from v, following the *low1* children of the vertices on the path; for every vertex y that we encounter we check whether there exists a back-edge (x, y) with $x \in T(M(v))$. The first y with this property is $low_M(u)$, and we set $(low_M D(u), low_M(u)) \leftarrow (x, y)$. To achieve linear running time, we let $In[y]$ denote the list of all vertices x for which there exists an incoming back-edge to y with higher end x. In other words, $In[y]$ contains all vertices x for which there exists a back-edge (x, y). Furthermore, we have the elements of $In[y]$ sorted in increasing order (this can be done easily in linear time with bucket-sort). When we process a vertex y as we descend $T(u, v]$, during the processing of v, we traverse $In[y]$ starting from the element we accessed the last time we traversed $In[y]$ (or, if this is the first time we traverse $In[y]$, from the first element of $In[y]$). Thus, we need a variable

Algorithm 1: Calculate $(low_M D(v), low_M(v))$, for Every Vertex v with $nextM(v) \neq \perp$

```
1  calculate In[y], for every vertex y, and have its elements sorted in increasing order
2  foreach vertex y do currentBackEdge[y] ← first element of In[y]
3  foreach vertex v do low_M[v] ← ⊥
4  for v ← n to v = 1 do
5      if prevM(v) = ⊥ then continue
6      u ← prevM(v)
7      y ← v
8      while low_M(u) = ⊥ do
9          while currentBackEdge[y] ≠ ⊥ do
10             x ← currentBackEdge[y]
11             if x < M(u) then
12                 currentBackEdge[y] ← next element of In[y]
13             end
14             else
15                 if x < M(u) + ND(M(u)) then
16                     (low_M D(u), low_M(u)) ← (x, y)
17                 end
18                 break
19             end
20         end
21         if low_M(u) = ⊥ then
22             if prevM(c_1(y)) = ⊥ then y ← c_1(y)
23             else y ← low_M(prevM(c_1(y)))
24         end
25     end
26 end
```

$currentBackEdge[y]$ to store the element of $In[y]$ that we accessed the last time we traversed $In[y]$. Now, for every $x \in In[y]$ that we meet, we check whether $x \in T(M(v))$. If that is the case, then we set $(low_M D(u), low_M(u)) \leftarrow (x, y)$; otherwise, we move to the next element of $In[y]$. If we reach the end of $In[y]$, then we descend the path $T(u, v]$ by moving to the *low1* child of y. In fact, if $prevM(c_1(y)) \neq \perp$, then we may descend immediately to $low_M(prevM(c_1(y)))$. This ensures that $In[y]$ will not be accessed again. Algorithm 1 shows how to compute all pairs $(low_M D(v), low_M(v))$, for all vertices v with $nextM(v) \neq \perp$, in total linear time.

Proposition 3.2. *Algorithm 1 correctly computes all pairs* $(low_M D(v), low_M(v))$*, for all vertices* v *with* $nextM(v) \neq \perp$*, in total linear time.*

Proof. Let v be a vertex with $prevM(v) = u \neq \perp$. We will prove inductively that $(low_M D(u), low_M(u))$ will be computed correctly, and that $low_M D(u)$ will be the lowest vertex which is a descendant of $M(u)$ such that $(low_M D(u), low_M(u))$ is a back-edge. So let us assume that we have run Algorithm 1 and we have correctly computed all pairs $(low_M D(u'), low_M(u'))$, for all vertices $v' > v$ with $prevM(v') = u' \neq \perp$, and $low_M D(u')$ is the lowest vertex in $T(M(u'))$ such that $(low_M D(u'), low_M(u'))$ is a back-edge. Suppose also that we have currently descended the path $T(u, v]$, we have reached y, and $low_M(u) \geq y$.

Let us assume, first, that $low_M(v) = y$, and let (x, y) be the back-edge such that $x \in T(M(v))$ and x is minimal with this property. The *while* loop in line 9 will search the list of incoming back-edges to y, starting from $currentBackEdge[y]$. If $currentBackEdge[y]$ is the first element of $In[y]$, then it is certainly true that x will be found. Otherwise, let $x' = currentBackEdge[y]$. Due to the inductive hypothesis, we have that $(x', y) = (low_M D(u'), low_M(u'))$, for a vertex u' with $nextM(u') = v' > v$. Then, y is in $T(u', v']$, but also in $T(u, v]$, and thus it is a common descendant of v and v'. This means that v and v' are related as ancestor and descendant. In particular, since $v' > v$, we have that v is an ancestor of v'. Furthermore, since y is an ancestor of u, it is also an ancestor of $M(u) = M(v)$; therefore, since v' is an ancestor of y, it is also an ancestor of $M(v)$. Since v is an ancestor of v', this implies that $M(v')$ is an ancestor of $M(v)$. Since $M(v') = M(u')$ and $M(v) = M(u)$, we thus have that $M(u')$ is an ancestor of $M(u)$, and therefore $M(u') \leq M(u)$. Thus, since x' is the lowest descendant of $M(u')$ such that (x', y) is a back-edge, and x is the lowest descendant of $M(u)$ such that (x, y) is a back-edge, we have $x' \leq x$. This shows that x will be accessed during the *while* loop in line 9.

Now let us assume that $low_M(v) \neq y$. This means that $low_M(u)$ is greater than y, and we have to descend the path $T(u, y)$ to find it. First, let c be the child of y in the direction of u. Then we have $low1(c) < v$ (since $M(v) = M(u)$ is a descendant of u, and therefore a descendant of c, and we have $low1(M(v)) < v$). If there was another child c' of y with $low1(c') < v$, this would imply that $M(v) = y$, which is absurd, since y is a proper ancestor of u, and therefore a proper ancestor of $M(u) = M(v)$. This means that c is the *low1* child of v, and thus we may descend to $c_1(y) = y'$. Now we have $low_M(u) \geq y'$. If $prevM(y') = \perp$, then we simply traverse the list of incoming back-edges to y', in line 9, and repeat the same process. Otherwise, let $u' = prevM(y')$. Due to the inductive hypothesis, we know that $low_M(u')$ has been computed correctly. Since y' is an ancestor of u, it is also an ancestor of $M(u) = M(v)$. Furthermore, y' is a descendant of v. Thus, $M(y')$ is an ancestor of $M(v)$, and therefore $M(u')$ is an ancestor of $M(u)$ (since $M(y') = M(u')$ and $M(v) = M(u)$). This means that u' is an ancestor of $M(u)$. Now we see that $low_M(u)$ lies on $T(u, low_M(u')]$. (For otherwise, $(low_M D(u), low_M(u))$ would be a back-edge in $B(u')$ with $low_M(u) \geq y' = nextM(u')$ and $low_M(u) < low_M(u')$, contradicting the minimality of $low_M(u')$). Thus we may descend immediately to $low_M(u')$. Then we traverse the list of incoming back-edges to $low_M(u')$, in line 9, and repeat the same process. Eventually we will reach $low_M(u)$ and have it computed correctly.

Since we have established the correctness of Algorithm 1, it remains to explain why it runs in linear time. Consider a vertex v with $prevM(v) \neq \perp$, and let $u = prevM(v)$. Then the algorithm will compute $low_M(u)$ by descending the tree path $T(u, v]$ (where it may possibly skip some vertices through shortcuts), until it reaches the lowest vertex on it which has an incoming back-edge from $T(M(u))$. Specifically, for every vertex y processed on this path, there are two cases: either y will be determined to be $low_M(u)$, or not. In the first case, we can charge the processing of y to the processing of v. Furthermore, each entry of the list $In[y]$ of the incoming back-edges to y that was traversed during the processing of v – except possibly the last one met—will not be accessed again. In the second case, we will show that we will not meet y again during the descension of a tree path. It should be clear that this establishes the linear-time complexity of Algorithm 1.

So let us suppose the contrary. That is, there exist two vertices v and v', with $v \neq v'$ and $u := prevM(v) \neq \perp$ and $u' := prevM(v') \neq \perp$, such that, when the algorithm descended the tree paths $T(u, v]$ and $T(u', v']$ (in order to compute $low_M(u)$ and $low_M(u')$, respectively), it met y both times, where the first time it was determined that y does not provide the low_M point. We may assume, w.l.o.g., that $v < v'$. Thus, since we process the vertices in decreasing order, we have that v' was processed before v, and we determined that $y \neq low_M(u')$. Furthermore, during the descension of the tree path $T(u', v']$, the algorithm had to descend further in order to reach $low_M(u')$, and so we have that $low_M(u')$ is a proper descendant of y.

Now, since y is a common descendant of v and v', we have that v and v' are related as ancestor and descendant, and therefore v is a proper ancestor of v'. Since the algorithm met y during the descension of the tree path $T(u, v]$, it cannot have met $p(v')$ during this descension, because otherwise it would have used the shortcut from $p(v')$ to $low_M(prevM(v')) = low_M(u')$, and so it would have moved to a proper descendant of y, without ever meeting y. Thus, during the descension of $T(u, v]$, the algorithm skipped the processing of $p(v')$. The only explanation for this, is that there is a proper ancestor v'' of v', such that $u'' := prevM(v'') \neq \perp$, and the algorithm met $p(v'')$ during the descension of $T(u, v]$, and after processing it it moved directly to $low_M(u'')$, where $low_M(u'')$ is a proper descendant of $p(v')$ (so that $p(v')$ was avoided), but also an ancestor of y (so that y was eventually met).

Now we will show that this is an impossible situation. First, there is a back-edge of the form $(x, low_M(u''))$, where x is a descendant of $M(u'')$. Then, since $low_M(u'') \in T[y, v]$, we have that x is not a descendant of u', because otherwise we would have $low_M(u') \leq low_M(u'')$ (due to the minimality of $low_M(u')$), in contradiction to the fact that $low_M(u')$ is a proper descendant of y. Now let (z, w) be a back-edge in $B(v'')$. Then, we have that w is a proper ancestor of v'', and therefore a proper ancestor of v'. Furthermore, since $M(v'') = M(u'')$, we have that z is a descendant of u'', and therefore a descendant of $low_M(u'')$, and therefore a descendant of v' (since $low_M(u'')$ is a proper descendant of $p(v')$ in the direction of v'). This shows that $(z, w) \in B(v')$. Due to the generality of $(z, w) \in B(v'')$, this implies that $B(v'') \subseteq B(v')$, and therefore we have that $M(v'')$ is a descendant of $M(v')$. Then, since $M(v'') = M(u'')$ and $M(v') = M(u')$, we have that $M(u'')$ is a descendant of $M(u')$. But since x is a descendant of $M(u'')$, this implies that x is a descendant of $M(u')$, and therefore a descendant of u'. Thus, we have arrived at a contradiction. □

Now, given these new parameters, we show how to compute all 3-cuts of type-2 (consisting of two tree-edges and one back-edge) without using the *high* points of [13]. We use the following characterization of such cuts.

Lemma 3.3 ([13]). *Let u, v be two vertices with $v < u$. Suppose that $\{(u, p(u)), (v, p(v)), e\}$ is a 3-cut, where e is a back-edge. Then v is an ancestor of u, and 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.*

In the following, for any vertex u, we let $V(u)$ denote the set of all vertices v that are ancestors of u and such that $B(v) = B(u) \sqcup \{e\}$, for a back-edge e. Also, for any vertex v, we let $U(v)$ denote the set of all vertices u that are descendants of v and such that $B(u) = B(v) \sqcup \{e\}$, for a back-edge e. In [13] it is shown that $V(u) \cap V(u') = \emptyset$ (resp. $U(v) \cap U(v') = \emptyset$) for every two vertices $u \neq u'$ (resp. $v \neq v'$). Thus, in order to find all type-2 cuts, it is sufficient to find, for every vertex u (resp. every vertex v) the unique vertex v (resp. u), if it exists, such that $u \in U(v)$ (resp. $v \in V(u)$), and then identify the back-edge e such that $\{(u, p(u)), (v, p(v)), e\}$ is a 3-cut. The following two lemmas show how to identify e.

Lemma 3.4. *Let u, v be two vertices such that u is a descendant of v and $B(v) = B(u) \sqcup \{e\}$, for a back-edge $e = (x, y)$. Then we have $M(u) \in \{\tilde{M}(v), M_{low1}(v), M_{low2}(v)\}$. In particular, we have that either* (1) $M(u) = \tilde{M}(v)$ *and $e = (M(v), l_1(M(v)))$, or* (2) $M(u) = M_{low1}(v)$ *and $e = (M_{low2}(v), l_1(M_{low2}(v)))$, or* (3) $M(u) = M_{low2}(v)$ *and $e = (M_{low1}(v), l_1(M_{low1}(v)))$.*

Proof. First we will show that $M(v)$ is a proper ancestor of $M(u)$. Obviously, $M(v)$ is an ancestor of $M(u)$, since $B(v) = B(u) \sqcup \{e\}$. Furthermore, since $e \in B(v)$, x is a descendant of $M(v)$, and y is a proper ancestor of v, and therefore a proper ancestor of u. Thus, it cannot be the case that $M(u) = M(v)$, for otherwise we would have $e \in B(u)$. This shows that $M(v)$ is a proper

ancestor of $M(u)$. Now we will show that $low1(c_k(M(v))) \geq v$, for every $k > 2$. Suppose for the sake of contradiction, that there exists a $k > 2$ such that $low1(c_k(M(v))) < v$. Then we have $low1(c_{k'}(M(v))) < v$, for every $k' \leq k$, and so $B(v) \cap B(c_{k'}(M(v))) \neq \emptyset$, for every $k' \leq k$. Then, since $B(u) = B(v) \setminus \{e\}$, we have that $B(u) \cap B(c_{k'}(M(v))) \neq \emptyset$, for all $k' \leq k$, except possibly a $\tilde{k} \in \{1, \ldots, k\}$. Thus, $M(u)$ is a common ancestor of all $\{c_1(M(v)), \ldots, c_k(M(v))\}$, except possibly $c_{\tilde{k}}(M(v))$, and so, since $k > 2$, we conclude that $M(u)$ is an ancestor of $M(v)$, which is absurd. Thus we have shown that $low1(c_k(M(v))) \geq v$, for every $k > 2$.

Now, there are two cases to consider: either $x = M(v)$, or x is a descendant of a child of $M(v)$. First take the case $x = M(v)$. Then $(M(v), l_1(M(v)))$ is obviously a back-edge in $B(v)$. Furthermore, since $M(u)$ is not an ancestor of $M(v)$, we also have $(M(v), l_1(M(v))) \notin B(u)$. Thus $e = (M(v), l_1(M(v)))$. Since every other back-edge of the form $(M(v), y')$ with $y' < v$ must have $(M(v), y') \in B(v) \setminus B(u)$, we conclude that e is the unique back-edge of the form $(M(v), y')$ with $y' < v$. Since $B(u) = B(v) \setminus \{e\}$, this means that $M(u) = nca(\{x' \mid \exists (x', y') \in B(v)\} \setminus \{M(v)\}) = \tilde{M}(v)$. Next, consider the case that x is a descendant of a child of $M(v)$. Then we have that x is either a descendant of $c_1(M(v))$, or a descendant of $c_2(M(v))$ (since $low1(c_k(M(v))) \geq v$, for every $k > 2$). We will consider only the case that x is a descendant of $c_1(M(v))$, since the other case can be treated in a similar manner. So let x be a descendant of $c_1(M(v))$. Then we must have $low1(c_2(M(v))) < v$, for otherwise there would exist a back-edge e' of the form $(M(v), y')$ with $y' < v$, and so we would have two distinct back-edges $e, e' \in B(v) \setminus B(u)$, which is absurd. Thus, $B(v) \cap B(c_2(M(v))) \neq \emptyset$, and therefore, since $B(u) = B(v) \setminus \{e\}$, we have $B(u) \cap B(c_2(M(v))) \neq \emptyset$. Thus, we must necessarily have $B(u) \cap B(c_1(M(v))) = \emptyset$, for otherwise $M(u)$ would be an ancestor of both $c_1(M(v))$ and $c_2(M(v))$, and therefore an ancestor of $M(v)$, which is absurd. Since $B(v) = B(u) \sqcup \{e\}$ and $low1(c_1(M(v))) < v$, this means that $B(v) \cap B(c_1(M(v))) = \{e\}$, and therefore $e = (M_{low1}(v), l_1(M_{low1}(v)))$. □

Lemma 3.5. *Let u, v be two vertices such that u is a descendant of $v \neq r$ and $B(u) = B(v) \sqcup \{e\}$, for a back-edge $e = (x, y)$. Then we have $M(v) \in \{M(u), \tilde{M}(u), M_{low1}(u)\}$. In particular, we have that either (1) $M(v) = M(u)$ and $e = (low_M D(u), low_M(u))$, or (2) $M(v) = \tilde{M}(u)$ and $e = (M(u), l_1(M(u)))$, or (3) $M(v) = M_{low1}(u)$ and $e = (M_{low2}(u), l_1(M_{low2}(u)))$.*

Proof. $B(u) = B(v) \sqcup \{e\}$ implies that $M(u)$ is an ancestor of $M(v)$. If $M(u) = M(v)$, then $v = nextM(u)$. (For otherwise, there exists a vertex v' with $M(v') = M(u)$ and $v < v' < u$, and so we have $B(v) \subset B(v') \subset B(u)$—which is impossible, since $B(u) = B(v) \sqcup \{e\}$.) Since $e \in B(u) \setminus B(v)$ and $|B(u) \setminus B(v)| = 1$ and $(low_M D(u), low_M(u))$ is a back-edge in $B(u) \setminus B(nextM(u))$, we conclude that $e = (low_M D(u), low_M(u))$.

Now let's assume that $M(u)$ is a proper ancestor of $M(v)$. There are two cases to consider: either $x = M(u)$, or x is a descendant of a child of $M(u)$. If $x = M(u)$, then $(M(u), l_1(M(u)))$ is a back-edge in $B(u)$. Furthermore, $(M(u), l_1(M(u))) \notin B(v)$ (for otherwise we would have that $M(v)$ is an ancestor of $M(u)$). This shows that $e = (M(u), l_1(M(u)))$. We also see that $(M(u), y') \notin B(v)$, for any back-edge $(M(u), y')$ with $y' < u$. Thus, since $B(v) = B(u) \setminus \{e\}$, we have $M(v) = nca(\{x' \mid \exists (x', y') \in B(u)\} \setminus \{M(u)\}) = \tilde{M}(u)$. Finally, let's assume that x is a descendant of a child of $M(u)$, i.e., x is a descendant of $c_k(M(u))$, for some k. We will show that $low1(c_{k'}(M(u))) < v$, for every $k' < k$. So suppose for the sake of contradiction, that $low1(c_{k'}(M(u))) \geq v$, for some $k' < k$. Since $e \in B(c_k(M(u)))$, we have $low1(c_k(M(u))) < u$; therefore, since $low1(c_{k'}(M(u))) \leq low1(c_k(M(u)))$, we have $low1(c_{k'}(M(u))) < u$. This shows that there exists a back-edge $e' \in B(u)$ which is also in $B(c_{k'}(M(u)))$. But since, $low1(c_{k'}(M(u))) \geq v$, it cannot be the case that $e' \in B(v)$. Thus we have provided two distinct back-edges $e, e' \in B(u) \setminus B(v)$, which is absurd. This shows that $low1(c_{k'}(M(u))) < v$, for every $k' < k$. If $k > 2$, this implies that $M(v)$ is a common ancestor of at least two children of $M(u)$, which is absurd. Thus we have that $k \leq 2$. Now suppose for the

sake of contradiction, that $k = 1$. It cannot be the case that $low1(c_2(M(u))) < v$, for otherwise $low1(c_1(M(u))) < v$ also, which would imply that $M(v)$ is an ancestor of $M(u)$, which is absurd. Now, it cannot be the case that $low1(c_2(M(u))) < u$, for otherwise there would exist two distinct back-edges $e', e \in B(u) \setminus B(v)$, which is also absurd. Thus, $c_1(M(u))$ is the only child of $M(u)$ with $low1(c_1(M(u))) < u$, which means that there must exist a back-edge $(M(u), y')$ with $y' < u$. Now if $y' < v$, $M(v)$ is an ancestor of $M(u)$, which is absurd. And if $y \geq v$, then we have two distinct back-edges $e, e' \in B(u) \setminus B(v)$, which is also absurd. Thus the assumption $k = 1$ cannot hold, and we conclude that $k = 2$, i.e., x is a descendant of $c_2(M(u))$. Now it cannot be the case that $low1(c_2(M(u))) < v$, for this would imply $low1(c_1(M(u))) < v$, and so we would have that $M(v)$ is an ancestor of $M(u)$. Thus $v \leq low1(c_2(M(u))) < u$, and so $B(u) \cap B(c_2(M(u))) = \{e\}$ (since $|B(u) \setminus B(v)| = 1$). This shows that $e = (M_{low2}(u), l_1(M_{low2}(u)))$. Then, since $B(v) = B(u) \setminus \{e\}$, we have that $B(v) = (B(u) \cap B(c_1(M(u)))) \sqcup \{(x, y') \in B(u) \mid x = M(u)\}$. But since $M(u)$ is a proper ancestor of $M(v)$, we conclude that $B(v) = B(u) \cap B(c_1(M(u)))$, and therefore $M(v) = M_{low1}(u)$. □

The next two lemmas are the basis for a linear-time algorithm to compute all 3-cuts of type-2.

Lemma 3.6 ([13]). *Let v be an ancestor of u such that $m = M(u) \in \{\tilde{M}(v), M_{low1}(v), M_{low2}(v)\}$. Then, $v \in V(u)$ if and only if: u is the minimum vertex in $M^{-1}(m)$ greater than v, and is such that $high(u) < v$ and $bcount(v) = bcount(u) + 1$.*

Lemma 3.7 ([13]). *Let u be a descendant of v such that $m = M(v) \in \{M(u), \tilde{M}(u), M_{low1}(u)\}$. Then, $u \in U(v)$ if and only if: v is the maximum vertex in $M^{-1}(m)$ less than u, and is such that $bcount(u) = bcount(v) + 1$.*

Now let $\{(u, p(u)), (v, p(v)), e\}$ be a 3-cut (of type-2), where e is a back-edge and $v < u$. By Lemma 3.3, we have that u is a descendant of v and that either $B(v) = B(u) \sqcup \{e\}$ or $B(u) = B(v) \sqcup \{e\}$. We will handle these cases in turn. In the following, we let $e = (x, y)$ be the back-edge of the 3-cut of type-2.

▶ Case $B(v) = B(u) \sqcup \{e\}$: By Lemma 3.4, one of the following cases holds (see Figure 2: either (1) $x = M(v)$, $y = l_1(M(v))$ and $M(u) = \tilde{M}(v)$, or (2) $x = M_{low2}(v)$, $y = l_1(M_{low2}(v))$ and $M(u) = M_{low1}(v)$, or (3) $x = M_{low1}(v)$, $y = l_1(M_{low1}(v))$ and $M(u) = M_{low2}(v)$. In any case, by Lemma 3.6 we have that u is the minimum vertex in $M^{-1}(m)$ greater than v, where $m = \tilde{M}(v)$, or $m = M_{low1}(v)$, or $m = M_{low2}(v)$, depending on whether (1), or (2), or (3) is true, respectively. Thus, we can compute all those 3-cuts by finding, for every vertex v, and every $m \in \{\tilde{M}(v), M_{low1}(v), M_{low2}(v)\}$, the minimum vertex u in $M^{-1}(m)$ greater than v, and then check whether $B(v) = B(u) \sqcup \{e\}$, for a back-edge e. This last condition is equivalent to $bcount(v) = bcount(u) + 1$ and $high(u) < v$. Note that $high(u) < v$ is necessary to ensure $B(u) \subseteq B(v)$; then, $bcount(v) = bcount(u) + 1$ implies the existence of a back-edge e such that $B(v) = B(u) \sqcup \{e\}$. We will now show how to check this condition without using the *high* points.

Lemma 3.8 (Case (1)). *Let $m = \tilde{M}(v)$ and let u be the minimum vertex in $M^{-1}(m)$ strictly greater than v. Then there exists a back-edge e such that $B(v) = B(u) \sqcup \{e\}$ if and only if: $bcount(v) = bcount(u) + 1$, $l_2(M(v)) \geq v$, and either $M(v)$ has no low2 child, or $low1(c_2(M(v))) \geq v$.*

Proof. ($\Rightarrow$) $bcount(v) = bcount(u) + 1$ is an immediate consequence of $B(v) = B(u) \sqcup \{e\}$. Furthermore, $B(v) = B(u) \sqcup \{e\}$ implies that $M(v)$ is an ancestor of $M(u)$. But it cannot be the case that $M(v) = M(u)$ (for otherwise, v being an ancestor of u would imply that $B(v)$ is a subset of $B(u)$); thus, $M(v)$ is a proper ancestor of $M(u)$. Since $M(u) = \tilde{M}(v)$, this implies that there is no back-edge (x, y) with x a descendant of a child c of $M(v)$, with $c \neq c_1(M(v))$, and y a proper

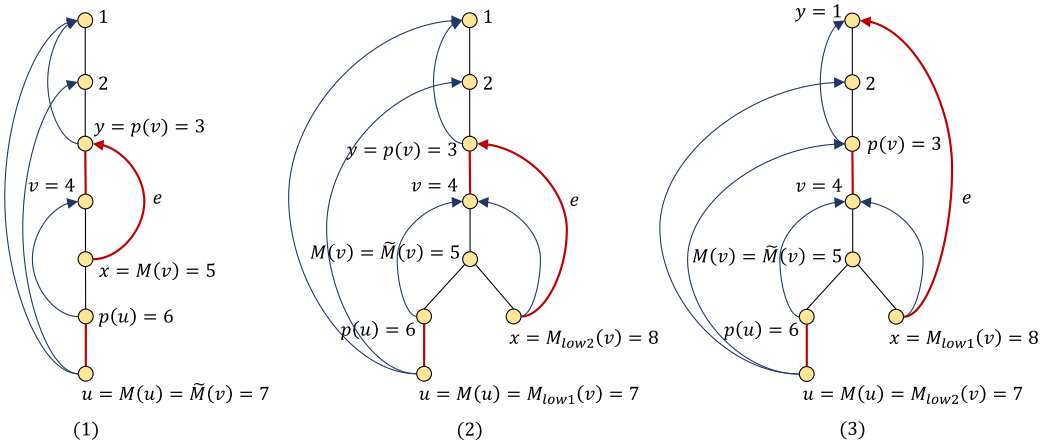

Fig. 2. An illustration of the three cases for type-2 cuts $\{(u, p(u)), (v, p(v)), e\}$ (shown with red edges) where v is an ancestor of u, e is a back-edge, and $B(v) = B(u) \sqcup \{e\}$. Vertices are numbered in DFS order and back-edges are shown directed from descendant to ancestor in the DFS tree.

ancestor of v (otherwise, we would have $\tilde{M}(v) = M(v)$). This means that $low1(c) \geq v$, for every child c of $M(v)$ with $c \neq c_1(M(v))$. In other words, either $M(v)$ has no *low2* child, or $low1(c_2(M(v))) \geq v$. Since $\tilde{M}(v) \neq \perp$, this also means that there exists a back-edge $\tilde{e} = (M(v), l_1(M(v)))$. Obviously, $\tilde{e} = e$, since $\tilde{e} \in B(v) \setminus B(u)$. Now, if we had $l_2(M(v)) < v$, then there would exist a back-edge $e' = (M(v), l_2(M(v))) \neq e$. But then we would have $e' \in B(v) \setminus B(u)$, contradicting $B(v) = B(u) \sqcup \{e\}$. ($\Leftarrow$) Let $(x, y) \in B(v)$. Since $M(v)$ either has no *low2* child, or $low1(c_2(M(v))) \geq v$, we have that x is either $M(v)$ or a descendant of $\tilde{M}(v)$. Let $\tilde{B}(v) = \{(x, y) | x \in T(\tilde{M}(v)) \text{ and } y < v\}$. Then we have $B(v) = \tilde{B}(v) \sqcup \{(M(v), z) | z < v\}$. Now, if $(x, y) \in \tilde{B}(v)$, then x is a descendant of $\tilde{M}(v)$, and therefore a descendant of $M(u)$. Furthermore, y is an ancestor of v, and therefore an ancestor of u. This means that $\tilde{B}(v) \subseteq B(u)$. Now, since $l_2(M(v)) \geq v$, there is at most one back-edge $e = (M(v), z)$ with $z < v$ (which thus satisfies $z = l_1(M(v))$). But such a back-edge must necessarily exist, for otherwise we would have $B(v) = \tilde{B}(v) \subseteq B(u)$, contradicting $bcount(v) = bcount(u) + 1$. Now, since $B(v) = \tilde{B}(v) \sqcup \{(M(v), z) | z < v\}$ and $|\{(M(v), z) | z < v\}| = 1$ and $\tilde{B}(v) \subseteq B(u)$ and $bcount(v) = bcount(u) + 1$, we conclude that $\tilde{B}(v) = B(u)$, and therefore $B(v) = B(u) \sqcup \{(M(v), l_1(M(v)))\}$. □

Lemma 3.9 (Case (2)). *Let $l_1(M(v)) \geq v$, $m = M_{low1}(v)$, and u be the minimum vertex in $M^{-1}(m)$ strictly greater than v. Then there exists a back-edge e such that $B(v) = B(u) \sqcup \{e\}$ if and only if: $bcount(v) = bcount(u) + 1$, $low2(M_{low2}(v)) \geq v$, and either $M(v)$ has no low3 child, or $low1(c_3(M(v))) \geq v$.*

Proof. ($\Rightarrow$) $bcount(v) = bcount(u) + 1$ is an immediate consequence of $B(v) = B(u) \sqcup \{e\}$. Now, $l_1(M(v)) \geq v$ implies that every back-edge $(x, y) \in B(v)$ has $x \in T(c)$, where c is a child of $M(v)$. Take a back-edge $(x, y) \in B(v)$, with $x \notin T(c_1(M(v)))$. Then $(x, y) \notin B(u)$, and therefore $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) = e$. This means that $x \in T(c_2(M(v)))$, $low2(M_{low2}(v)) \geq v$, and either $M(v)$ has no *low3* child, or $low1(c_3(M(v))) \geq v$.
($\Leftarrow$) $l_1(M(v)) \geq v$ implies that every back-edge $(x, y) \in B(v)$ has $x \in T(c)$, where c is a child of $M(v)$. Furthermore, it implies that there exists at least one back-edge $(x, y) \in B(v)$ with $x \in T(c_2(M(v)))$. Now let $B_{low1}(v) = \{(x, y) | x \in T(c_1(M(v)))\}$. Then, since $low2(M_{low2}(v)) \geq v$, and either $M(v)$ has no *low3* child, or $low1(c_3(M(v))) \geq v$, we have that $B(v) = B_{low1}(v) \sqcup \{(M_{low2}(v), l_1(M_{low2}(v)))\}$. Since $M_{low1}(v) = M(u)$ and v is an ancestor of u, we have that

$B_{low1}(v) \subseteq B(u)$. Now, from $B(v) = B_{low1}(v) \sqcup \{(M_{low2}(v), l_1(M_{low2}(v)))\}$, $B_{low1}(v) \subseteq B(u)$, and $bcount(v) = bcount(u) + 1$, we conclude that $B(v) = B(u) \sqcup \{(M_{low2}(v), l_1(M_{low2}(v)))\}$. □

Lemma 3.10 (Case (3)). *Let $l_1(M(v)) \geq v$, $m = M_{low2}(v)$, and u be the mimimum vertex in $M^{-1}(m)$ strictly greater than v. Then there exists a back-edge e such that $B(v) = B(u) \sqcup \{e\}$ if and only if: $bcount(v) = bcount(u) + 1$, $low2(M_{low1}(v)) \geq v$, and either $M(v)$ has no low3 child, or $low1(c_3) \geq v$, where c_3 is the low3 child of $M(v)$.*

Proof. ($\Rightarrow$) $bcount(v) = bcount(u) + 1$ is an immediate consequence of $B(v) = B(u) \sqcup \{e\}$. Now, $l_1(M(v)) \geq v$ implies that every back-edge $(x, y) \in B(v)$ has $x \in T(c)$, where c is a child of $M(v)$. Take a back-edge $(x, y) \in B(v)$, with $x \notin T(c_2(M(v)))$. Then $(x, y) \notin B(u)$, and therefore $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) = e$. This means that $x \in T(c_1(M(v))$, $low2(M_{low1}(v)) \geq v$, and either $M(v)$ has no *low3* child, or $low1(c_3(M(v))) \geq v$.
($\Leftarrow$) $l_1(M(v)) \geq v$ implies that every back-edge $(x, y) \in B(v)$ has $x \in T(c)$, where c is a child of $M(v)$. Furthermore, it implies that there exists at least one back-edge $(x, y) \in B(v)$ with $x \in T(c_1(M(v)))$. Now let $B_{low2}(v) = \{(x, y) | x \in T(c_2(M(v)))\}$. Then, since $low2(M_{low1}(v)) \geq v$, and either $M(v)$ has no *low3* child, or $low1(c_3(M(v))) \geq v$, we have that $B(v) = B_{low2}(v) \sqcup \{(M_{low1}(v), l_1(M_{low1}(v)))\}$. Since $M_{low2}(v) = M(u)$ and v is an ancestor of u, we have that $B_{low2}(v) \subseteq B(u)$. Now, from $B(v) = B_{low2}(v) \sqcup \{(M_{low1}(v), l_1(M_{low1}(v)))\}$, $B_{low2}(v) \subseteq B(u)$, and $bcount(v) = bcount(u) + 1$, we conclude that $B(v) = B(u) \sqcup \{(M_{low1}(v), l_1(M_{low1}(v)))\}$. □

Algorithm 2 shows how we can determine all 3-cuts of this type. The idea is to handle cases (1), (2) and (3) separately, and our goal is to find, for every vertex v, the minimum vertex u in $M^{-1}(m)$ (where $m \in \{\tilde{M}(v), M_{low1}(v), M_{low2}(v)\}$) which is strictly greater than v, and then check whether $B(v) = B(u) \sqcup \{e\}$, for a back-edge e. We can perform this search in linear time by processing the vertices in a bottom-up fashion, and keep in a variable $currentVertex[m]$ the lowest element of $M^{-1}(m)$ that we accessed so far. Then we can easily check in constant time whether a pair of vertices u, v such that u is the minimum vertex in $M^{-1}(m)$ which is strictly greater than v satisfies $B(v) = B(u) \sqcup \{e\}$, for a back-edge e, and also identify this back-edge.

▶ Case $B(u) = B(v) \sqcup \{e\}$: Now let u, v be two vertices such that v is an ancestor of u with $B(u) = B(v) \sqcup \{e\}$, for a back-edge e. By Lemma 3.5, one of the following cases holds (see Figure 3): either (1) $M(v) = M(u)$ and $e = (low_M D(u), low_M(u))$, or (2) $M(v) = \tilde{M}(u)$ and $e = (M(u), l_1(M(u)))$, or (3) $M(v) = M_{low1}(u)$ and $e = (M_{low2}(u), l_1(M_{low2}(u)))$. In any case, by Lemma 3.7 we have that v is the maximum vertex in $M^{-1}(m)$ less than u, where $m = M(u)$, or $m = \tilde{M}(u)$, or $m = M_{low1}(u)$, depending on whether (1), or (2), or (3) is true, respectively. Thus, we can compute all those 3-cuts by finding, for every vertex u, and every $m \in \{M(v), \tilde{M}(u), M_{low1}(u)\}$, the maximum vertex v in $M^{-1}(m)$ less than u, and then check whether $B(u) = B(v) \sqcup \{e\}$, for a back-edge e. This last condition is equivalent to $bcount(u) = bcount(v) + 1$.

Algorithm 3 shows how we can determine all 3-cuts of this type. Similar to Algorithm 2, the idea is to handle cases (1), (2) and (3) separately, and our goal is to find, for every vertex u, the maximum vertex v in $M^{-1}(m)$ (where $m \in \{M(u), \tilde{M}(u), M_{low1}(u)\}$) which is strictly less than u, and then check whether $B(u) = B(v) \sqcup \{e\}$, for a back-edge e. We can perform this search in linear time by processing the vertices in a bottom-up fashion, and keep in a variable $currentVertex[m]$ the lowest element of $M^{-1}(m)$ that we accessed so far. Then we can easily check in constant time whether a pair of vertices u, v such that v is the maximum vertex in $M^{-1}(m)$ which is strictly less than u satisfies $B(u) = B(v) \sqcup \{e\}$, for a back-edge e, and also identify this back-edge.

Algorithm 2: 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
// Case (1): $m = \tilde{M}(v)$
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $v \leftarrow n$ *to* $v = 1$ **do**
  $m \leftarrow \tilde{M}(v)$
  if $m = \perp$ **then continue**
  // find the minimum $u \in M^{-1}(m)$ that is greater than v
  $u \leftarrow currentVertex[m]$
  while $nextM(u) \neq \perp$ ***and*** $nextM(u) > v$ **do** $u \leftarrow nextM(u)$
  $currentVertex[m] \leftarrow u$
  // check the condition in Lemma 3.8
  if $bcount(v) = bcount(u) + 1$ ***and*** $l_2(M(v)) \geq v$ ***and*** ($c_2(M(v)) = \perp$ ***or*** $low1(c_2(M(v))) \geq v$) **then**
    mark the triplet $\{(u, p(u)), (v, p(v)), (M(v), l_1(M(v)))\}$ as a 3-edge cut
  end
end
// Case (2): $m = M_{low1}(v)$
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $v \leftarrow n$ *to* $v = 1$ **do**
  $m \leftarrow M_{low1}(v)$
  if $m = \perp$ ***or*** $l_1(M(v)) < v$ **then continue**
  // find the minimum $u \in M^{-1}(m)$ that is greater than v
  $u \leftarrow currentVertex[m]$
  while $nextM(u) \neq \perp$ ***and*** $nextM(u) > v$ **do** $u \leftarrow nextM(u)$
  $currentVertex[m] \leftarrow u$
  // check the condition in Lemma 3.9
  if $bcount(v) = bcount(u) + 1$ ***and*** $low2(M_{low2}(v)) \geq v$ ***and*** ($c_3(M(v)) = \perp$ ***or*** $low1(c_3(M(v))) \geq v$) **then**
    mark the triplet $\{(u, p(u)), (v, p(v)), (M_{low2}(v), l_1(M_{low2}(v)))\}$ as a 3-edge cut
  end
end
// Case (3): $m = M_{low2}(v)$
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $v \leftarrow n$ *to* $v = 1$ **do**
  $m \leftarrow M_{low2}(v)$
  if $m = \perp$ ***or*** $l_1(M(v)) < v$ **then continue**
  // find the minimum $u \in M^{-1}(m)$ that is greater than v
  $u \leftarrow currentVertex[m]$
  while $nextM(u) \neq \perp$ ***and*** $nextM(u) > v$ **do** $u \leftarrow nextM(u)$
  $currentVertex[m] \leftarrow u$
  // check the condition in Lemma 3.10
  if $bcount(v) = bcount(u) + 1$ ***and*** $low2(M_{low1}(v)) \geq v$ ***and*** ($c_3(M(v)) = \perp$ ***or*** $low1(c_3(M(v))) \geq v$) **then**
    mark the triplet $\{(u, p(u)), (v, p(v)), (M_{low1}(v), l_1(M_{low1}(v)))\}$ as a 3-edge cut
  end
end

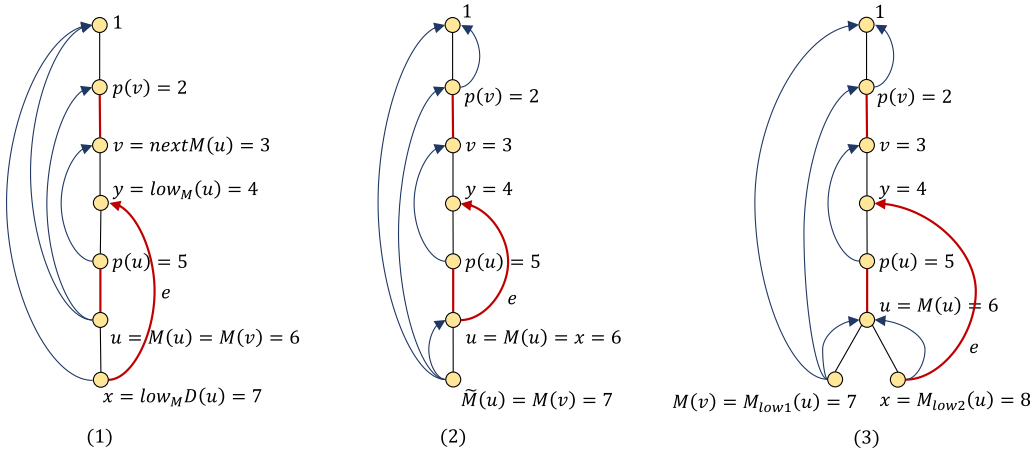

Fig. 3. An illustration of the three cases for type-2 cuts $\{(u, p(u)), (v, p(v)), e\}$ (shown with red edges) where v is an ancestor of u, e is a back-edge, and $B(u) = B(v) \sqcup \{e\}$. (Note that in the examples of Cases (2) and (3) the input graphs have parallel edges.) Vertices are numbered in DFS order and back-edges are shown directed from descendant to ancestor in the DFS tree.

3.5 An Improved Linear-Time Algorithm

In the previous section we showed that we can compute all 3-cuts in linear time in pointer machines, using only elementary data structures. This involves the computation of the edge $(low_M D(v), low_M(v))$, for every vertex $v \neq r$ with $nextM(v) \neq \perp$. Observe that we only need this edge in Line 5 of Algorithm 3, because this is the only edge in $B(v) \setminus B(nextM(v))$ in this case. Here we consider an alternative way to retrieve the DFS numbers of the endpoints of this edge, that avoids the computation of the $low_M D$ and low_M values.

For every vertex $v \neq r$, we consider the values $XorDesc(v)$ and $XorAnc(v)$, which are defined as the XORs of the DFS numbers of the higher ends and the lower ends, respectively, of the back-edges in $B(v)$. In other words, $XorDesc(v) := \oplus\{x \mid \exists(x, y) \in B(v)\}$ and $XorAnc(v) := \oplus\{y \mid \exists(x, y) \in B(v)\}$. These values can be computed easily during the DFS traversal. Thus, we initialize every $XorDesc(v)$ to 0, and the computation is performed in a bottom-up fashion. If we have computed $XorDesc$ for every proper descendant of v, then we set (1) $XorDesc(v) \leftarrow XorDesc(v) \oplus XorDesc(c)$ for every child c of v, (2) $XorDesc(v) \leftarrow XorDesc(v) \oplus v$ for every back-edge (v, y), and (3) $XorDesc(y) \leftarrow XorDesc(y) \oplus v$ for every back-edge (v, y). With (1) we rely on the computation of the previous $XorDesc$ values, and (2) captures the back-edges that stem from v. The purpose of (3) is to cancel out, beforehand, all v that correspond to the back-edges of the form (v, y), because these are captured in $XorDesc(c)$ and will be XORed with $XorDesc(y)$ in (2) when the DFS backtracks to y, where c is the child of y that is an ancestor of v. $XorAnc(v)$ is computed similarly.

Now, if we know that $B(v) \setminus B(nextM(v))$ consists of only one back-edge $e = (x, y)$, then, since $B(nextM(v)) \subset B(v)$ is generally true, we have that $x = XorDesc(nextM(v)) \oplus XorDesc(v)$ and $y = XorAnc(nextM(v)) \oplus XorAnc(v)$. We note that, in pointer machines, the pair of values (x, y) does not coincide with the edge e that has those endpoints,[2] and we have to scan the list of the

[2]This is because, in pointer machines, we can gain access to the data only through pointers, but pointer arithmetic is not supported. Thus, since the pair (x, y) was the result of applying a XOR operation, this is just a pair of numbers that does not correspond to a pointer, and there is no RAM available for reference.

Algorithm 3: Find All 3-Cuts $\{(u, p(u)), (v, p(v)), e\}$, Where u Is a Descendant of v and $B(u) = B(v) \sqcup \{e\}$, for a Back-Edge e

initialize an array $currentVertex$ with n entries
// Case (1): $m = M(v)$; just check whether the condition of Lemma 3.7 is satisfied for $nextM(u)$
foreach *vertex* $u \neq r$ **do**
  if $bcount(u) = bcount(nextM(u)) + 1$ **then**
    mark the triplet $\{(u, p(u)), (nextM(u), p(nextM(u))), (low_M D(u), low_M(u))\}$ as a 3-edge cut
  end
end
// Case (2): $m = \tilde{M}(u)$
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $u \leftarrow n$ *to* $u = 1$ **do**
  $m \leftarrow \tilde{M}(u)$
  if $m = \perp$ ***or*** $m = M(u)$ **then continue**
  // find the maximum $v \in M^{-1}(m)$ that is smaller than u
  $v \leftarrow currentVertex[m]$
  while $v \neq \perp$ ***and*** $v \geq u$ **do** $v \leftarrow nextM(v)$
  $currentVertex[m] \leftarrow v$
  // check the condition in Lemma 3.7
  if $bcount(u) = bcount(v) + 1$ **then**
    mark the triplet $\{(u, p(u)), (v, p(v)), (M(u), l_1(M(u)))\}$ as a 3-edge cut
  end
end
// Case (3): $m = M_{low1}(u)$
foreach *vertex* x **do** $currentVertex[x] \leftarrow x$
for $u \leftarrow n$ *to* $u = 1$ **do**
  $m \leftarrow M_{low1}(u)$
  if $m = \perp$ ***or*** $l_1(M(u)) < u$ **then continue**
  // find the maximum $v \in M^{-1}(m)$ that is smaller than u
  $v \leftarrow currentVertex[m]$
  while $v \neq \perp$ ***and*** $v \geq u$ **do** $v \leftarrow nextM(v)$
  $currentVertex[m] \leftarrow v$
  // check the condition in Lemma 3.7
  if $bcount(u) = bcount(v) + 1$ **then**
    mark the triplet $\{(u, p(u)), (v, p(v)), (M_{low2}(u), l_1(M_{low2}(u)))\}$ as a 3-edge cut
  end
end

edges of the graph in order to find it. In practice, however, where we use RAM programs, we can use this pair of values as an edge, and so this algorithm is expected to run faster than the previous one that uses the $low_M D$ and low_M values.

4 Empirical Analysis

We implemented our algorithms in C++, using g++ 7.5.0 with full optimization (flag -O3) to compile the code.[3] The reported running times were measured on a GNU/Linux machine, with Ubuntu (18.04.6 LTS): a Dell Precision Tower 7820 server 64-bit NUMA machine with an Intel(R) Xeon(R) Gold 5220R processor and 192 GB of RAM. The processor has 24.75 MB of cache memory and 18 cores. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the high_resolution_clock function of the standard library chrono, averaged over 10 different runs. We used valgrind (version 3.20.0) in order to measure the memory usage for every run of the algorithms.

4.1 Implementation Details

We tested five implementations, two for the randomized algorithm of Nadara et al. [25] (NRSS and NRSS-sort), one for the deterministic algorithm of Georgiadis et al. [13] (GIK), one for our linear-time pointer-machine algorithm (Linear), and one for its variant that avoids the use of the low_M points (Linear+). We did not include an implementation of the deterministic algorithm of [25], because it is more complicated than the routine in GIK for computing type-2 cuts, and it uses data structures for offline nearest common ancestors (in addition to the static tree DSU data structure of Gabow and Tarjan).

In NRSS we made use of a hash table in order to store the *CH* values of the tree-edges and to be able to retrieve them efficiently. An alternative suggestion for this problem is to use radix sort [25]. However, the latter is only significant for the theoretical analysis, since in practice we observed that this part of the algorithm does not play a significant role in the total running time (as this is dominated by the computation of other parameters). Furthermore, we made the following improvements in this algorithm. First, we handle the case of type-1 cuts using the same idea as in GIK and in our linear-time pointer-machine algorithm. Specifically, we detect those cuts by using the values *bcount*, *low1D*, *low1*, *low2D* and *low2*. (This method is also suggested in the deterministic algorithm of [25].) This confers an obvious advantage, since (1) these parameters can be computed easily and fast during the DFS, and (2) we thus avoid having to store $O(m)$ *CH* values and pointers to the corresponding back-edges. Second, we implemented a simpler and faster linear-time algorithm for computing all MinDn and MaxDn values, that uses only elementary data structures. We observed that this algorithm is several times faster than the one suggested by [25], but it only slightly affects the total running time, since this is dominated by the computation of *high* and *highD* points (i.e., the MaxUp edges). Finally, we used two different algorithms for MaxUp, since the total running time is heavily dependent on the way this computation is performed. The first algorithm (as in GIK) simply traverses the adjacency lists. This is enough for GIK, but for NRSS we then have to sort the adjacency lists, in order to keep pointers to the MaxUp edges, and this incurs a significant overhead. The second implementation of NRSS uses bucket-sort on the back-edges (as suggested by [25]), and it is called NRSS-sort. In both GIK and NRSS(-sort) we implemented the DSU data structure by using path compression and union by size. Although this is not theoretically optimal, in practice it works efficiently and we get a linear-time behavior.

4.2 Experimental Results for Real-World Graphs

For the experimental evaluation, we considered several real-world (undirected) graphs, taken from various application areas, as well as random graphs. For each graph G, we also compute a corresponding sparse 4-edge-connectivity certificate of G. A k-edge-connectivity certificate of G is a spanning subgraph H of G such that, for any two vertices u and v and any positive integer $k' \leq k$,

[3] Our code, together with some sample input instances is available at https://github.com/EvanK20/4eComponents.

Table 1. Characteristics of Real-World Graphs Taken from the SNAP [23]

Graph	n	m	#1CCs	#2CCs	#3CCs	#4CCs	m'	type
Deezer-HR	54,573	498,202	1	2,436	5,188	7,880	194,107	SN
Facebook-Artist	50,515	819,306	1	3,220	6,425	9,146	179,349	SN
com-Amazon	548,552	925,872	213,690	244,981	284,574	336,545	884,127	PN
Gowalla	196,591	950,327	65,294	112,349	135,493	145,544	325,636	SN
com-DBLP	425,957	1,049,866	108,878	155,981	217,842	268,721	826,192	CN
com-Youtube	1,157,828	2,987,624	22,939	690,029	860,753	942,232	1,999,435	SN
roadNet-CA	1,971,281	5,533,214	8,713	8,713	385,230	385,230	5,520,326	RN
twitch-gamers	168,114	6,797,557	1	4,081	7,975	11,800	648,245	SN
as-Skitter	1,696,415	11,095,298	756	232,897	493,601	680,604	5,181,377	IT
com-LiveJournal	3,997,962	34,681,189	38,577	860,464	1,292,384	1,588,620	2,519,167	SN
com-Orkut	3,072,627	117,185,083	187	67,981	111,261	149,635	11,953,289	SN
com-Friendster	124,836,180	1,806,067,135	59,227,815	73,213,653	79,168,479	82,886,290	203,718,281	SN

n is the number of vertices, m is the number of edges, and #kCCs is the number of k-edge-connected components; m' is the number of edges in a 4-edge-connectivity certificate of G. The network types are: SN, PN, CN, RN, and IT.
CN, collaboration; IT, internet topology; PN, product; RN, road; SN, social.

u and v are k'-edge-connected in H if and only if they are k'-edge-connected in G. Such a subgraph H has at most $k(n-1)$ edges and can be computed in linear time [27]. The reason why we use both the original graphs and their sparse certificates is twofold. First, we would like to observe if sparsity affects the relative performance of the algorithms. Second, we would like to see how more efficient is the computation of the 4-edge-connected components on the sparse certificates rather than on the original graphs.

Table 1 shows some statistics about the real-world graphs used in our experimental evaluation, and Table 2 provides more details about their 4-edge-connected components. The running times of the algorithms for the original graphs and for their sparse certificates are shown in Table 3 and Table 4, respectively.[4] Figures 5–7 illustrate the detailed running times for each part of the 4-edge-connected components algorithms on the real-world graphs and their sparse certificates. For each algorithm, it depicts the running time for computing the $(k-1)$-edge-cuts and the corresponding k-connected components for $k \in \{1, 2, 3, 4\}$. Figure 4 illustrates the memory usage for each algorithm applied on some real-world graphs and their sparse certificates.

The algorithms GIK, Linear, NRSS and NRSS-sort compute the 4-edge-connected components, report the number of k-edge-connected components, for $k \leq 4$, and the corresponding times to compute them, as well as the number of minimal k-edge-cuts, for $k \leq 3$. (We note that the number of minimal k-edge-cuts can be as large as $O(n^2)$ and $O(n^3)$ for $k = 2, 3$, respectively.) For computing the k-edge-connected components for $k < 4$, all algorithms use the same routines. For $k = 1$ we simply perform a BFS; for $k = 2$ we implement the classic algorithm of Tarjan that uses the *low* points [34]. For $k = 3$ we use the algorithm of GK, which is the simplest and fastest known for computing 2-edge-cuts [12, 14]. Due to the organization of the 2-edge-cuts that this algorithm provides, it is straightforward to compute, for every 3-edge-connected component, the auxiliary 3-edge-connected graph that is derived from it [5], by adding some virtual edges, and then we run the corresponding 3-edge-cut algorithm on this graph. Finally, after we have computed all 3-edge-cuts of each such graph (the number of those cuts is $O(n)$ in total), we compute its 4-edge-connected components by using the algorithm of [25] to compute the cactus tree.

[4]We note that we did not run the algorithms on the original com-Friendster graph, because it has so many edges that the RAM of our system was not enough to support the computation of its 4-edge-connected components. We only had enough memory to compute its sparse certificate.

Table 2. Statistics about the 4-Edge-Connected Components of the Real-World Graphs of Table 1

Graph	4ECC					
	#vertices	#4ECCs	#trivial	%trivial	size of largest	average size of non-trivial
Deezer-HR	54,573	7,880	7,879	99.987	46,694	46,694.00
Facebook-Artist	50,515	9,146	9,141	99.945	41,362	8,274.80
com-Amazon	548,552	336,545	333,271	99.027	191,787	65.75
Gowalla	196,591	145,544	145,543	99.999	51,048	51,048.00
com-DBLP	425,957	268,721	267,206	99.436	151,248	104.78
com-Youtube	1,157,828	942,232	941,505	99.922	214,271	297.55
roadNet-CA	1,971,281	385,230	381,188	98.950	1,566,101	393.39
twitch-gamers	168,114	11,800	11,799	99.991	156,315	156,315.00
as-Skitter	1,696,415	680,604	680,010	99.912	1,014,845	1,711.11
com-LiveJournal	3,997,962	1,588,620	1,585,952	99.832	2,431,816	918.51
com-Orkut	3,072,627	149,635	149,633	99.998	2,922,992	1,461,497.00
com-Friendster	124,836,180	82,886,290	82,884,482	99.997	41,945,644	23,203.37

For every graph we can see the number of vertices, the number of 4-edge-connected components, the number of trivial (i.e., singleton) 4-edge-connected components, the percentage of the trivial ones, the size of the largest 4-edge-connected component, and the average size of the non-trivial ones.

Table 3. Running Times in Seconds of the Algorithms for the Real-World Graphs of Table 1

Graph	NRSS-sort		NRSS		GIK		Linear		Linear+	
	total	4ECCs	total	4ECCs	total	4ECCs	total	4ECCs	total	4ECCs
Deezer-HR	0.210	0.139	0.220	0.148	**0.130**	**0.060**	0.142	0.071	0.140	0.067
Facebook-Artist	0.293	0.204	0.285	0.194	0.165	0.089	0.181	0.089	**0.164**	**0.077**
com-Amazon	0.744	0.445	0.783	0.487	**0.529**	**0.236**	0.572	0.269	0.562	0.263
Gowalla	0.321	0.203	0.320	0.205	**0.202**	**0.084**	0.219	0.102	0.206	0.091
com-DBLP	0.682	0.407	0.725	0.453	**0.483**	**0.202**	0.515	0.237	0.504	0.223
com-Youtube	1.946	1.110	1.985	1.114	**1.356**	**0.478**	1.476	0.599	1.409	0.530
roadNet-CA	4.220	2.698	4.421	2.895	**2.693**	**1.203**	2.881	1.354	2.901	1.386
twitch-gamers	4.107	3.012	3.490	2.395	**1.842**	**0.739**	2.184	1.083	1.949	0.852
as-Skitter	7.605	5.133	7.599	5.144	**4.659**	**2.185**	5.172	2.686	4.959	2.463
com-LiveJournal	38.894	26.623	38.143	25.917	**23.208**	**10.857**	26.400	14.101	24.804	12.489
com-Orkut	122.825	89.019	115.752	81.932	**61.300**	**27.627**	75.299	41.080	65.711	32.075

For each algorithm we report the total running time and the time required to compute the 3-edge-cuts and the corresponding 4-edge-connected components. Best running times for each graph are marked in bold.

From the running times reported in Tables 3 and 4, and plotted in Figures 5–7, we observe that running time of all algorithms is dominated by the computation of the 3-edge-cuts and the 4-edge-connected components. Indeed, computing the k-edge-connected components takes about twice as much as computing the $(k-1)$-edge-connected components. On average, the computation of the 3-edge-cuts and the 4-edge-connected components constitutes more than 60% of the running time of the NRSS algorithms and more than 40% of the running time of the GIK and Linear. Furthermore, as expected, all algorithms are executed faster on the sparse certificates, sometimes significantly, depending on the density of the original graph. (For instance, for twitch-gamers, all algorithms run about 10 times faster on the sparse certificate.) Thus, for computing the 4-edge-connected components and the 3-cuts of a dense graph, it is more efficient to produce the sparse certificate (even with a straightforward algorithm that performs four passes over the edges of the graph) and apply any of those algorithms on it, then apply the algorithm directly on the original graph.

Table 4. Running Times in Seconds of the Algorithms for the Sparse Certificates of the Real-World Graphs of Table 1

Graph	NRSS-sort		NRSS		GIK		Linear		Linear+	
	total	4ECCs	total	4ECCs	total	4ECCs	total	4ECCs	total	4ECCs
Deezer-HR-SC	0.115	0.069	0.128	0.081	**0.085**	**0.037**	0.087	0.042	0.090	0.043
Facebook-Artist-SC	0.100	0.061	0.112	0.071	**0.074**	**0.033**	0.076	0.036	0.077	0.037
com-Amazon-SC	0.720	0.430	0.789	0.488	**0.539**	**0.242**	0.566	0.270	0.554	0.261
Gowalla-SC	0.158	0.084	0.167	0.096	**0.122**	**0.045**	0.126	0.050	0.127	0.051
com-DBLP-SC	0.647	0.394	0.725	0.453	**0.445**	**0.185**	0.461	0.207	0.458	0.204
com-Youtube-SC	1.449	0.723	1.535	0.796	**1.135**	**0.377**	1.183	0.438	1.156	0.408
roadNet-CA-SC	4,187	2,664	4,403	2,881	**2,707**	**1,203**	2,893	1,370	2,949	1,368
twitch-gamers-SC	0.403	0.266	0.423	0.289	**0.262**	**0.121**	0.273	0.137	0.267	0.131
as-Skitter-SC	5.226	3.294	5.582	3.633	**3.649**	**1.679**	3.856	1.913	3.827	1.861
com-LiveJournal-SC	20.873	13.410	21.899	14.411	**14.859**	**7.240**	16.466	8.839	15.418	7.881
com-Orkut-SC	19,069	13.142	20.333	14.310	**13.355**	**7.084**	14.172	7.953	13.682	7.415
com-Friendster-SC	573.353	360.488	607.523	398.601	**401.717**	**196.158**	437.486	234.625	413.392	208.695

For each algorithm we report the total running time and the time required to compute the 3-edge-cuts and the corresponding 4-edge-connected components. Best running times for each graph are marked in bold.

We observe that the running time of the Linear(+) algorithm is very close to that of GIK. This means that the contraction operation does not incur a significant overhead in the total running time. One could expect the GIK algorithm to be worse than Linear, because it implements various sub-algorithms to handle the various subcases of type-3 cuts. However, all those sub-algorithms (except one, that needs an array of size $O(m)$) take time $O(n)$, because they rely on parameters already computed. Furthermore, although the Linear algorithm avoids the computation of *high* points, it uses as a replacement the $low_M D$ and low_M points, whose computation rely on a specific sorting of the adjacency lists.

We also observe that the NRSS algorithms are consistently slower than GIK and Linear, by a significant factor, in both the original graphs and their sparse certificates. Specifically, GIK is about 40% faster than NRSS-sort on the original graphs, and about 35% faster on the sparse certificates. This is somewhat expected, because all these algorithms use the same parameters (or similar methods to compute those that they need), but the NRSS algorithms need also to maintain pointers to the edges that correspond to those parameters, and they also compute the *CH* values that use at least three times the number of bits that are needed in order to encode an edge. The memory usage for some real-world graphs, plotted in Figure 4, demonstrates a similar pattern as that we get from the running times.

Comparison with the Maximal 4-Edge-Connected Subgraphs. Each of the five linear-time algorithms for computing the 3-cuts in 3-edge-connected graphs provides a corresponding $O(nm)$-algorithm for computing the maximal 4-edge-connected subgraphs in (general) graphs. Thus, we get five algorithms "max-4ecs-NRSS-sort," "max-4ecs-NRSS," "max-4ecs-GIK," "max-4ecs-Linear," and "max-4ecs-Linear+" for computing the maximal 4-edge-connected subgraphs. These algorithms work by repeatedly removing all mincuts of every connected component that is not 4-edge-connected. (Their only difference is in the subroutine for finding all 3-cuts in a component which is 3-edge-connected.) Furthermore, for every one of those algorithms, we implement two extra variants, that incorporate some simple but very effective heuristics. The first heuristic performs the following preprocessing (trimming) on the graph: repeatedly remove any vertex that has degree less than 4 (every such vertex is a trivial maximal 4-edge-connected subgraph). The second heuristic performs the trimming operation described in the first, not only in the preprocessing phase, but also every time that the small mincuts are removed from a connected component of the graph.

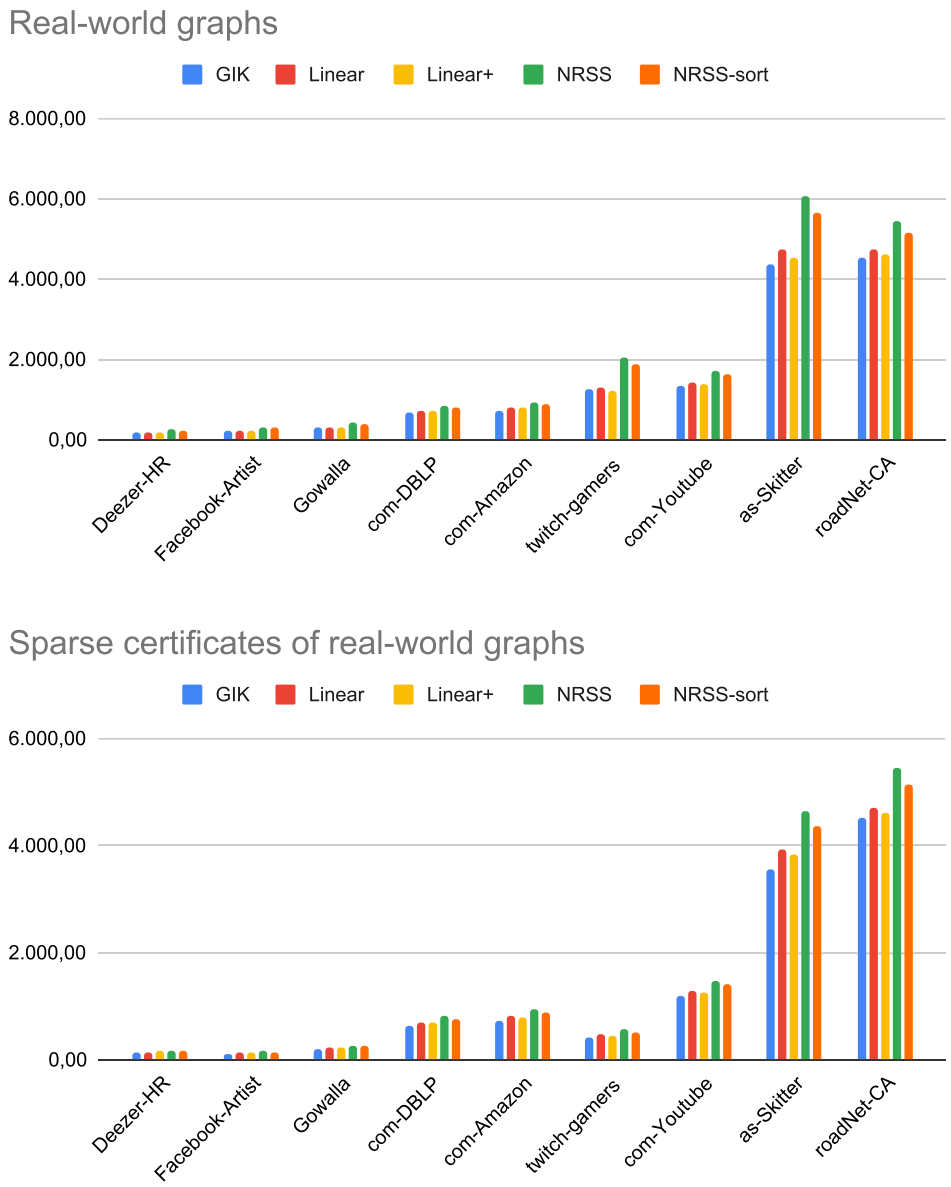

Fig. 4. Memory usage in MB for every instance of the 4-edge-connected components algorithms on real-world graphs and their sparse certificates.

In Table 5, we see some statistics about the maximal 4-edge-connected subgraphs of the real-world graphs of Table 1.[5] Similar to Table 2 that concerns the 4-edge-connected components, we report, for every graph, the number of maximal 4-edge-connected subgraphs, the number of the trivial ones, the percentage of the trivial, the size of the largest maximal 4-edge-connected subgraph (in terms of number of vertices), and the average size of the non-trivial ones. By comparing the statistics from Tables 2 and 6, we infer that the structure of the maximal 4-edge-connected subgraphs

[5]Except for com-Friendster, which was forbiddingly large for our RAM.

Fig. 5. Detailed running times for each part of the 4-edge-connected components algorithms on the first three real-world graphs of Table 1 and their sparse certificates. For each algorithm, we plot the running time for computing the $(k-1)$-edge-cuts and the corresponding k-connected components for $k \in \{1, 2, 3, 4\}$.

is very close to that of the 4-edge-connected components: that is, most maximal 4-edge-connected subgraphs are trivial, and the largest ones are induced by the largest 4-edge-connected components after removing a small part of them. This indicates that, if two vertices are 4-edge-connected in a real-world graph, then they are probably part of a 4-edge-connected subgraph S (thus, there are four edge-disjoint paths that connect them, that are also contained in S).

In Table 6, we report the times for computing the maximal 4-edge-connected subgraphs using the three variants of the five algorithms that we implemented. Without using any heuristics, we observe that in general it takes much more time to compute the maximal 4-edge-connected subgraphs in comparison to the 4-edge-connected components (e.g., for the graph com-Youtube, the computation is more than 40 times slower). However, it seems that the (worst-case) $O(nm)$ time-bound is very pessimistic, since we can get those subgraphs within a very reasonable amount of time in practice. Furthermore, our two heuristics provide a significant improvement, probably because most of the maximal 4-edge-connected subgraphs are indeed trivial. We notice that the second heuristic (that repeatedly removes vertices with degree less than 4 after the removal of all small mincuts of a connected component), is generally better than the first one, except in the graph roadNet-CA, where

Fig. 6. Detailed running times for each part of the 4-edge-connected components algorithms on the 4th to 8th real-world graph of Table 1 and their sparse certificates. For each algorithm, we plot the running time for computing the $(k-1)$-edge-cuts and the corresponding k-connected components for $k \in \{1, 2, 3, 4\}$.

Fig. 7. Detailed running times for each part of the 4-edge-connected components algorithms on the last four real-world graphs of Table 1 and their sparse certificates. For each algorithm, we plot the running time for computing the $(k-1)$-edge-cuts and the corresponding k-connected components for $k \in \{1, 2, 3, 4\}$.

it is slightly worse. This is explainable, because the trimming operation, if it manages to remove vertices in many recursion levels, is faster than searching for small cuts. Otherwise, it only slightly burdens the running time due to an extra search for vertices with small degree. Finally, we note that we get a better performance whenever we use a more efficient algorithm for computing 3-cuts in 3-edge-connected graphs (thus, e.g., max-4ecs-GIK has better performance than max-4ecs-NRSS).

Table 5. Statistics about the Maximal 4-Edge-Connected Subgraphs of the Real-World Graphs of Table 1

Graph	Max-4ECS					
	#vertices	#Max-4ECSs	#trivial	%trivial	size of largest	average size of non-trivial
Deezer-HR	54,573	8,715	8,714	99.988	45,859	45,859.00
Facebook-Artist	50,515	9,600	9,598	99.979	40,912	20,458.50
com-Amazon	548,552	387,652	384,366	99.152	129,516	49.96
Gowalla	196,591	146,205	146,204	99.999	50,387	50,387.00
com-DBLP	425,957	284,495	283,114	99.514	133,768	103.43
com-Youtube	1,157,828	977,637	977,509	99.986	179,481	1,408.74
roadNet-CA	1,971,281	385,230	381,188	98.950	1,566,101	393.39
twitch-gamers	168,114	11,956	11,955	99.991	156,159	156,159.00
as-Skitter	1,696,415	745,326	745,296	99.995	950,790	31,703.96
com-LiveJournal	3,997,962	1,666,493	1,665,139	99.918	2,355,360	1,751.40
com-Orkut	3,072,627	150,996	150,995	99.999	2,921,632	2,921,632.00

Table 6. Running Times in Seconds of the Algorithms for Computing the Maximal 4-Edge-Connected Subgraphs of the Real-World Graphs of Table 1

Graph	max-4ecs-NRSS-sort			max-4ecs-NRSS			max-4ecs-GIK			max-4ecs-Linear			max-4ecs-Linear+		
	-	pre	rep	-	pre	rep	-	pre	rep	-	pre	rep	-	pre	rep
Deezer-HR	7.1	0.1	0.1	7.5	0.2	0.2	4.7	0.1	0.1	3.9	0.1	0.1	3.8	0.1	0.1
Facebook-Artist	8.9	2.4	0.6	8.7	2.4	0.6	5.8	1.6	0.4	5.5	1.5	0.4	5.3	1.3	0.4
com-Amazon	34.5	15.1	5.8	36.6	16.1	6.29	24.0	10.8	3.8	18.1	8.6	3.8	18.1	8.6	3.7
Gowalla	7.8	0.2	0.2	7.7	0.2	0.2	5.1	0.1	0.1	5.2	0.1	0.1	4.9	0.1	0.1
com-DBLP	21.8	9.5	5.3	22.4	10.0	5.7	14.5	5.7	3.5	15.0	6.6	3.9	14.7	6.5	3.8
com-Youtube	93.5	16.7	10.3	94.4	16.9	10.8	65.5	11.0	7.6	56.2	9.2	7.4	54.3	8.6	6.9
roadNet-CA	4.9	4.1	4.2	5.1	4.3	4.4	3.5	2.9	3.0	3.5	2.8	2.9	3.4	2.8	2.9
twitch-gamers	118.0	3.2	3.2	113.1	2.8	2.8	74.2	1.3	1.3	80.0	1.6	1.6	70.0	1.4	1.4
as-Skitter	582.0	65.0	49.6	595.0	66.0	50.8	450.2	46.3	36.8	459.7	39.8	36.8	444.5	37.4	35.0
com-LiveJournal	3,002	767	554	2,992	759	552	2,253	499	363	2,188	487	417	2,080	448	392
com-Orkut	5,055	94.8	94.0	4,858	91.8	89.7	3,226	43.9	43.5	2,972	53.5	52.5	2,786	45.5	44.5

Each of the five algorithms has three variants: "-" simply removes k-cuts, for $k < 4$; "pre" does a preprocessing of repeatedly removing all vertices with degree less than 4; and "rep" also repeatedly removes all vertices with degree less than 4 after removing every k-cut, for $k < 4$.

4.3 Experimental Results for Artificial Graphs

Next, we explore further the relative performance of the four algorithms for computing 3-edge-cuts (in 3-edge-connected graphs). Thus we created random graphs with a fixed number of vertices and an increasing number of edges, and we applied four algorithms on those graphs and on their sparse certificates. We augmented the graphs, whenever needed, in order to become 3-edge-connected, by adding a relatively small number of new edges. Specifically, we computed their connected components, and we connected them on a path. Then we applied the algorithm of Eswaran and Tarjan [8] to introduce the smallest number of edges that are needed to make them 2-edge-connected. And finally we applied the algorithm of Naor et al. [29] to optimally increase the connectivity by one.

The corresponding plots are shown in Figures 8 and 9. In these experiments we notice that GIK can be as much as 50% faster than Linear, and more than twice as fast as NRSS(-sort). This fact is not easily observable in the full algorithms for computing the 4-edge-connected components, because there are other unrelated operations taking place, and also the sizes of the 3-edge-connected components are most probably not large enough to make this difference manifest. Indeed, in the sparse certificates of the random graphs, we observe that Linear has only slightly worse performance

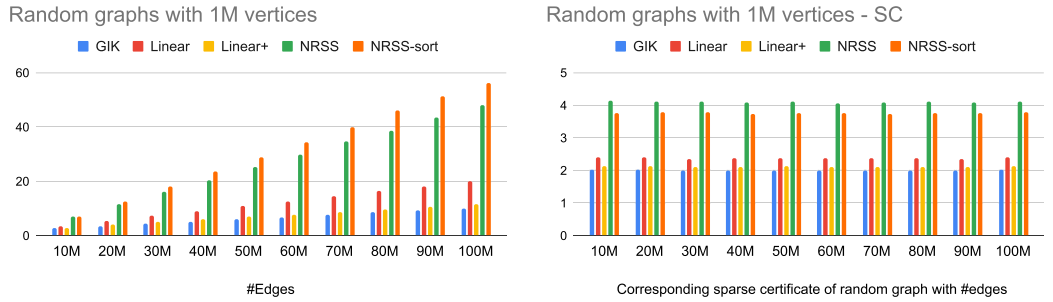

Fig. 8. Running times in seconds for random graphs with 1M vertices and their corresponding sparse certificates.

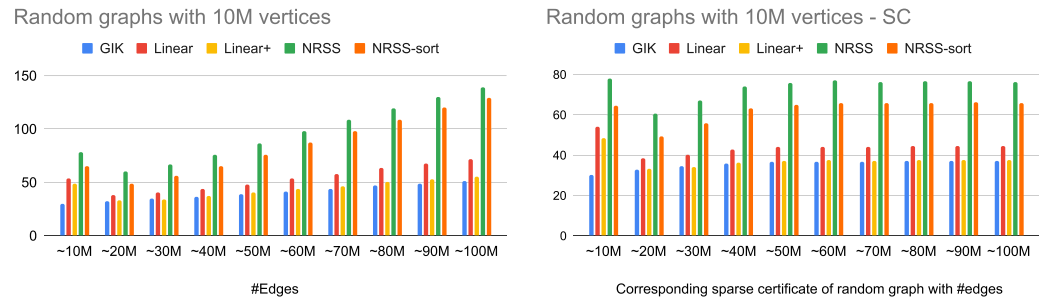

Fig. 9. Running times in seconds for random graphs with 10M vertices and their corresponding sparse certificates.

Table 7. Running Times in Seconds for Computing the Sparse Certificates of the Random Graphs

vertices/edges	**10M**	**20M**	**30M**	**40M**	**50M**	**60M**	**70M**	**80M**	**90M**	**100M**
1M	0.708	1.093	1.490	1.879	2.269	2.612	2.991	3.383	3.821	4.180
10M	3.425	4.854	7.374	9.986	11.473	12.782	13.964	15.303	16.366	17.682

than GIK. Linear+ is everywhere almost as fast as GIK. We notice that the NRSS-sort algorithm has somewhat worse performance than NRSS on dense graphs, but it performs better than NRSS in sparse graphs.

Table 7 shows the times that are needed in order to compute the sparse certificates of the random graphs. In combination with the times that are shown in Figures 8 and 9, we can see that for sufficiently sparse graphs it is better to apply any of the algorithms for computing 3-cuts directly on the graph, whereas for dense enough graphs it is better to apply the algorithms on a sparse certificate of the graph. The precise point at which it is better to first compute the sparse certificate depends on (a) the algorithm that we apply, (b) the number of vertices of the graph, and (c) the number of edges of the graph. In general, it seems that GIK and Linear(+) are good enough even if applied directly on the graph, whereas NRSS(-sort) perform better if applied on the sparse certificate.

In the plot for the graphs with 10M vertices and ~10M edges, we observe that the algorithms Linear(+) and NRSS(-sort) have significantly higher running time than expected. This is due to the contraction that is used in order to compute type-3 cuts. More specifically, the graphs generated are very sparse, and therefore the back-edges do not induce very large connected components. Thus, the contracted graph at the first iteration has very large size in relation to the original graph.

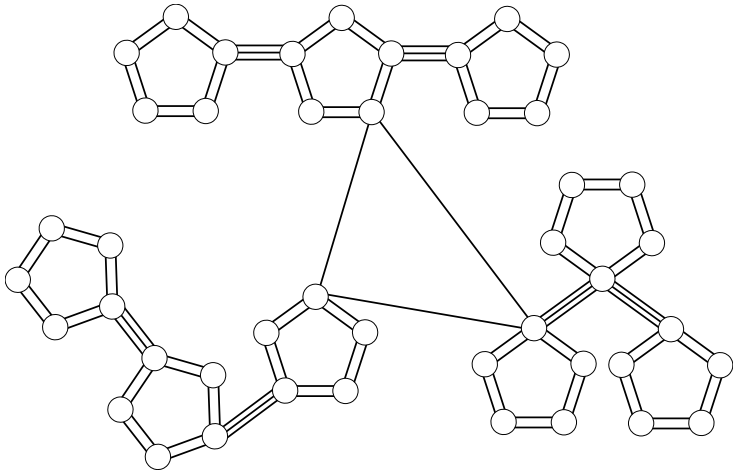

Fig. 10. An example of an artificial graph with $N = 5$, $N_4 = 3$ and $N_3 = 3$.

In particular, the contracted graphs at ∼10M edges have about 40% the number of vertices of the original graph. At higher densities, the running times have a clear linear-time behavior. (E.g., in the case of ∼20M edges, the contracted graph at the first iteration has about 8% the number of vertices of the original graph, and in the case of ∼30M edges about 1.5%.)

In conclusion, we note that the running times of the algorithms for computing 3-cuts do not seem to depend much on the structure of the graphs on which they are applied (e.g., on their number of cuts), but mainly on their number of vertices and edges. (This can also be inferred from the structure of the algorithms.) We remark that the algorithms Linear+ and NRSS(-sort) are the easiest to implement. Linear+ seems to be the best choice, since it is deterministic and has almost the best performance. We also report that the NRSS(-sort) algorithm, although it is Monte Carlo randomized, it always produced the correct answer in our experiments.

Controlling the Number of 3-Edge- and 4-Edge-Connected Components. Finally, we explore how the structure of the 3-edge- and the 4-edge-connected components affects the performance of the algorithms for computing the 4-edge-connected components. This study is motivated by Figures 5–7. Specifically, from those figures we observe that the heaviest work of the algorithms—which determines the difference in their performance—is done during the computation of the 4-edge-connected components (of the auxiliary 3-edge-connected graphs). This difference is overshadowed to some extent in the total running time, because this also involves the computation of the k-edge-connected components for $k < 4$. Furthermore, as we can see from Tables 1 and 2, the number of the 3-edge-connected components is very close to that of the 4-edge-connected components in the real-world graphs from our dataset, and more than 99% of the 4-edge-connected components are trivial. Thus, most of the 3-edge-connected components of those graphs are also trivial, and therefore the total running time on those graphs does not capture very well the difference in the performance of the algorithms.

In order to control precisely the number of 3-edge- and 4-edge-connected components, we generated artificial graphs with a fixed number of vertices, but with a specified structure for the 3-edge- and the 4-edge-connected components. Specifically, let N_3, N_4 and N, denote the desired number of 3-edge-connected components, the number of 4-edge-connected components within every 3-edge-connected component, and the number of vertices of every 4-edge-connected component, respectively. (Thus, the total number of vertices will be $N_3 \cdot N_4 \cdot N$.) First, we create

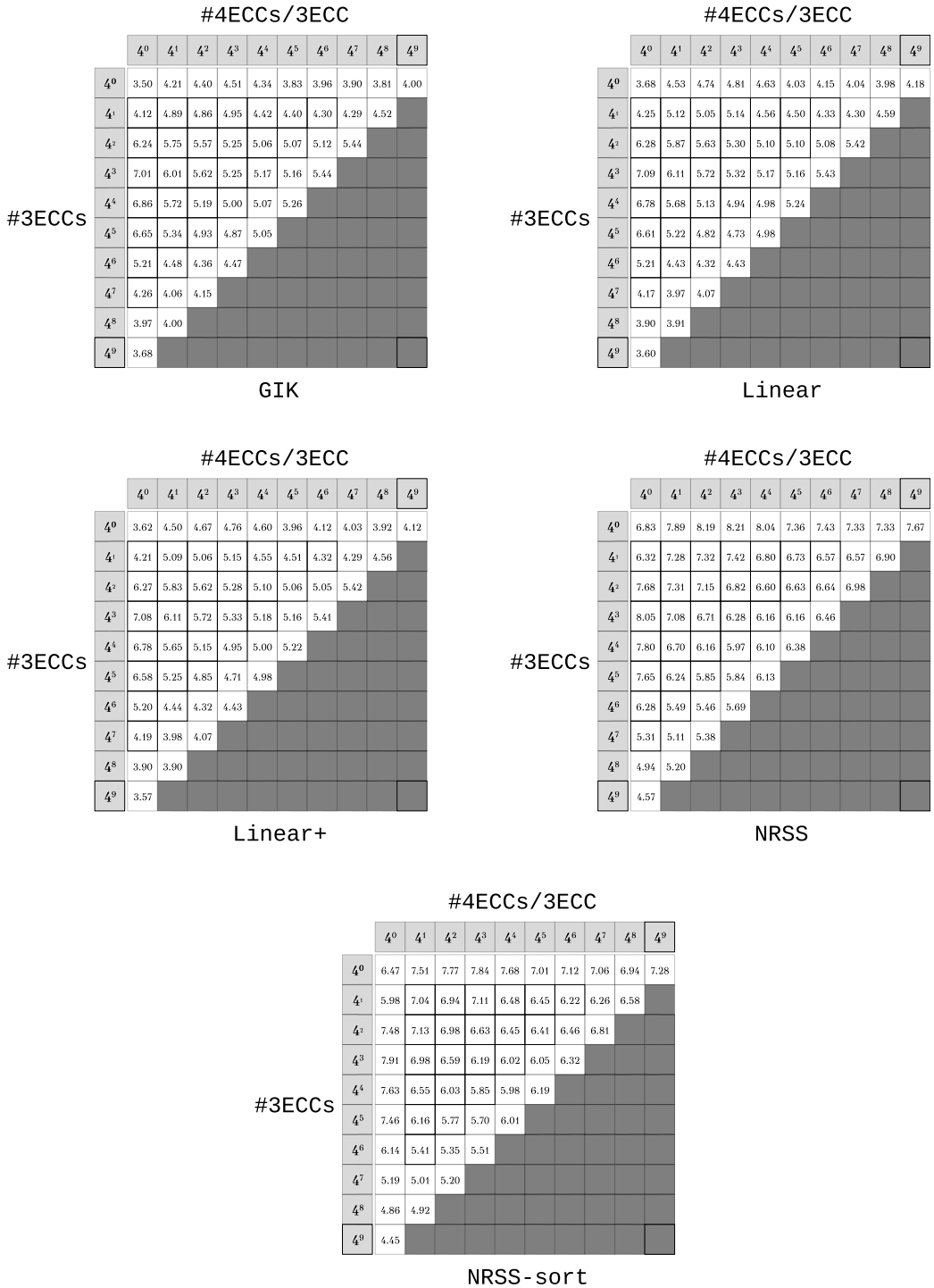

Fig. 11. Times in seconds for computing the 4-edge-connected components in artificial graphs with 4^{11} = 4,194,304 vertices and specified numbers of 3-edge-connected components and 4-edge-connected components within every 3-edge-connected component.

$N_3 \cdot N_4$ 4-edge-connected graphs, each with N vertices, by forming a cyclic graph on those N vertices, where we use two parallel edges between every two consecutive vertices on the cycle. (Thus, every such graph has $2 \cdot N$ edges.) Then we partition those graphs into N_3 sets of N_4 graphs, and for every such set we create a 3-edge-connected graph by joining its graphs in a path using three parallel edges (all three of which having the same endpoints) between any two consecutive graphs. Finally, we create a 2-edge-connected graph from those N_3 3-edge-connected graphs, by forming a cyclic graph with them (where we use the same vertex from every such graph as an endpoint for the two incident edges to it). Thus, the total number of edges is $2 \cdot N \cdot N_4 \cdot N_3 + 3 \cdot N_3 \cdot (N_4 - 1) + 2 \cdot N_3$. An example of such an artificial graph is shown in Figure 10.

In our experiments, we used artificial graphs with $4^{11} = 4{,}194{,}304$ vertices, and all possible combinations of values $N_3 = 4^0, \ldots, 4^9$ and $N_4 = 4^0, \ldots, 4^9$, such that $N_3 \cdot N_4 \leq 4^9$ (where N is adapted accordingly so as to keep the number of vertices fixed, i.e., we set $N = 4^{11}/(N_3 \cdot N_4)$). Thus, we get 55 different types of artificial graphs in total. Every such graph has the property that its 4-edge-connected components have the same number of vertices (i.e., N), and N ranges from 4^2 to 4^{11}. In order to introduce some randomness in the artificial graphs, we use a random seed which determines the numbering of the vertices and the endpoints for the various edges. In Figure 11 we can see the times reported for the algorithms that compute the 4-edge-connected components (averaged over five different instances, which showed no significant variation). Every non-empty cell of the matrices in Figure 11 corresponds to a particular (type of) artificial graph. The noticeable variation in the reported times per matrix cannot be attributed to the difference in the number of edges, but it is mainly due to the different structure of the 3-edge- and 4-edge-connected components of the corresponding graphs. This is because the number of edges in the 55 different artificial graphs ranges from 8,388,610 to 9,175,039.

We can make the following two observations from Figure 11. First, the greatest difference in the performance of the algorithms is revealed in the first row, which basically corresponds to 3-edge-connected graphs with 4^{11} vertices. Thus, we can see e.g., that GIK takes about half the time compared with NRSS and NRSS-sort on the upper-left corner graph. This difference in the performance was anticipated from our results on the random graphs (which are 3-edge-connected). Furthermore, in those sparse graphs, we can see that the number of the 4-edge-connected components has an observable impact on the running time. (E.g., compare the times in the cells $(4^0, 4^0)$ and $(4^0, 4^3)$ of the NRSS matrix.) More precisely, it is the relation between the number and the sizes of the 4-edge-connected components that makes this observable difference. And second, in each of the first few columns of every matrix, there is a somewhat consistent pattern of an initial increase in the running time, followed by a steady decrease. It is perhaps easier to explain the decreasing part: this is because the sizes of the 3-edge-connected components shrink exponentially, and therefore the computation of the 3-cuts makes a much lower contribution to the total running time. The initial increase in the running times can be attributed to the overhead of handling the increasing number of the 3-edge-connected components.

References

[1] T. Akiba, Y. Iwata, and Y. Yoshida. 2013. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM '13)*. ACM, 909–918.

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

[3] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, 205–216.

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

[5] Y. Dinitz. 1992. 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)*. Springer-Verlag, 145–157.
[6] Y. Dinitz and J. Westbrook. 1998 Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica* 20 (1998), 242–276.
[7] P. Erdös and J. Kennedy. 1987. k-connectivity in random graphs. *European Journal of Combinatorics* 8, 3 (1987), 281–286.
[8] K. P. Eswaran and R. E Tarjan. 1976. Augmentation problems. *SIAM Journal on Computing* 5, 4 (1976), 653–665.
[9] S. Forster, D. Nanongkai, L. Yang, T. Saranurak, and S. Yingchareonthawornchai. 2020. Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA '20)*. SIAM, 2046–2065.
[10] H. N. Gabow and R. E. Tarjan. 1985. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences* 30, 2 (1985), 209–221.
[11] Z. Galil and G. F. Italiano. 1991. Reducing edge connectivity to vertex connectivity. *ACM SIGACT News* 22, 1 (Mar. 1991), 57–61.
[12] L. Georgiadis, K. Giannis, G. F. Italiano, and E. Kosinas. 2021. Computing vertex-edge cut-pairs and 2-edge cuts in practice. In *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA '21)*. LIPIcs, Vol. 190, Schloss Dagstuhl Leibniz-Zentrum für Informatik, Article 20, 1–19.
[13] L. Georgiadis, G. F. Italiano, and E. Kosinas. Computing the 4-edge-connected components of a graph in linear time. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA '21)*. Leibniz International Proceedings in Informatics (LIPIcs), Vol. 204, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 47, 1–17.
[14] L. Georgiadis and E. Kosinas. Linear-time algorithms for computing twinless strong articulation points and related problems. In *Proceedings of the 31st International Symposium on Algorithms and Computation (ISAAC '20)*. Leibniz International Proceedings in Informatics (LIPIcs), Vol. 181, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Article 38, 1–16.
[15] D. Harel and R. E. Tarjan. 1984. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing* 13, 2 (1984), 338–355.
[16] J. E. Hopcroft and R. E. Tarjan. 1973. Dividing a graph into triconnected components. *SIAM Journal on Computing* 2, 3 (1973), 135–158.
[17] A. Kanevsky and V. Ramachandran. 1991. Improved algorithms for graph four-connectivity. *Journal of Computer and System Sciences* 42, 3 (1991), 288–306.
[18] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen. 1991. On-line maintenance of the four-connected components of a graph. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science (FOCS '91)*, 793–801.
[19] D. R. Karger and D. A. Panigrahi. Near-linear time algorithm for constructing a cactus representation of minimum cuts. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*. Society for Industrial and Applied Mathematics, 246–255.
[20] R. M. Karp and R. E. Tarjan. 1980. Linear expected-time algorithms for connectivity problems. *Journal of Algorithms* 1, 4 (1980), 374–393.
[21] T. Korhonen. 2025. Linear-time algorithms for k-edge-connected components, k-lean tree decompositions, and more. In *Proceedings of the 57th ACM Symposium on Theory of Computing (STOC '25)*.
[22] E. Kosinas. 2024. Computing the 5-edge-connected components in linear time. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms (SODA '24)*. SIAM, 1887–2119.
[23] J. Leskovec and A. Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved June 2014 from http://snap.stanford.edu/data
[24] K. Menger. 1927. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae* 10, 1 (1927), 96–115.
[25] W. Nadara, M. Radecki, M. Smulewicz, and M. Sokołowski. 2021. Determining 4-edge-connected components in linear time. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA '21)*. Leibniz International Proceedings in Informatics (LIPIcs), Vol. 204, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 71, 1–15.
[26] H. Nagamochi and T. Ibaraki. 1992. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics* 9, 2 (1992), 163.
[27] H. Nagamochi and T. Ibaraki. 2008. *Algorithmic Aspects of Graph Connectivity* (1st ed.). Cambridge University Press.
[28] H. Nagamochi and T. Watanabe. 1993. Computing k-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 76, 4 (1993), 513–517.
[29] D. Naor, D. Gusfield, and C. Martel. 1997. A fast algorithm for optimally increasing the edge connectivity. *SIAM Journal on Computing* 26, 4 (1997), 1139–1165.
[30] J. H. Reif, and, P. G. 1985. k-connectivity in random undirected graphs. *Discrete Mathematics* 54, 2 (1985), 181–191.
[31] T. Saranurak and W. Yuan. 2023. Maximal k-edge-connected subgraphs in almost-linear time for small k. In *Proceedings of the 31st Annual European Symposium on Algorithms (ESA '23)*. LIPIcs, Vol. 274, Schloss Dagstuhl Leibniz-Zentrum für Informatik, Article 92, 1–9.

[32] H. Sun, J. Huang, Y. Bai, Z. Zhao, X. Jia, F. He, and Y. Li. 2016. Efficient k-edge connected component detection through an early merging and splitting strategy. *Knowledge-Based Systems* 111 (2016), 63–72.
[33] S. Taoka, T. Watanabe, and K. Onaga. 1992. A linear-time algorithm for computing all 3-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 75, 3 (1992), 410–424.
[34] R. E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1, 2 (1972), 146–160.
[35] R. E. Tarjan. 1974. Finding dominators in directed graphs. *SIAM Journal on Computing* 3, 1 (1974), 62–89.
[36] R. E. Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the ACM* 22, 2 (1975), 215–225.
[37] Y. H. Tsin. 2007. A simple 3-edge-connected component algorithm. *Theory of Computing Systems* 40, 2 (Feb. 2007), 125–142.
[38] Y. H. Tsin. 2009. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms* 7, 1 (2009), 130–146.
[39] Y. H. Tsin. 2023. A simple linear-time algorithm for generating auxiliary 3-edge-connected subgraphs. arXiv:2309.13827. Retrieved from https://arxiv.org/abs/2309.13827

Appendix

A Number of k-Edge-Connected Components in Random Graphs, for $k \in \{1, 2, 3, 4\}$

Since various k-connectivity properties of random graphs have been studied in the literature (see, e.g., [7, 20, 30]), here we conduct the following experiment in order assess the number of k-edge-connected components in random graphs. We created random multigraphs with $n = 100{,}000$ vertices and m edges, where we select every edge with equal probability, for $m \in \{0, 1{,}000, 2{,}000, 3{,}000, \ldots, 1{,}000{,}000\}$. For each m we created 10 random multigraphs and counted the number of k-edge-connected components, for each $k \in \{1, 2, 3, 4\}$. The average values are depicted in Figure A1. We note that the deviation was negligible, and for this reason it is not presented in the plot. We notice that the average number of k-edge-connected components in a random multigraph with fixed number of vertices follows a similar pattern as m increases, for every k.

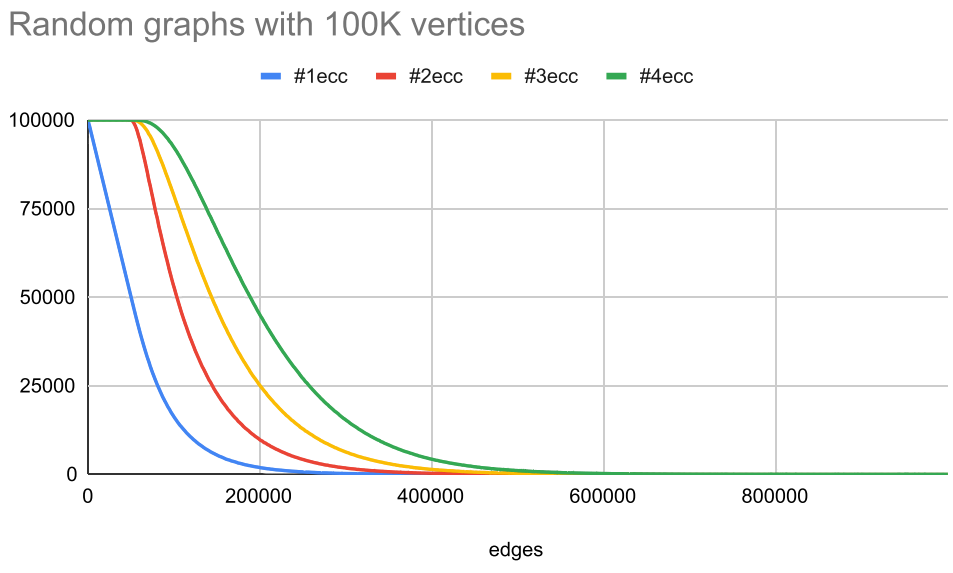

Fig. A1. Expected number of k-connected components, for $k \in \{1, 2, 3, 4\}$, in random graphs with 100K vertices.

Received 20 April 2023; revised 13 June 2025; accepted 21 July 2025