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

Loukas Georgiadis, Giuseppe F Italiano, Evangelos Kosinas

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

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

Giuseppe F. Italiano
LUISS University, Rome, Italy

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

Abstract

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 paper 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.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Connectivity Cuts, Edge Connectivity, Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.60

Related Version *arXiv Version*: `https://arxiv.org/abs/2108.08558` [9]

Supplementary Material *Software (Source Code)*: `https://github.com/4eComponentsALGO/files` archived at `swh:1:dir:f82f28576eba10039b91356977e84253d67c73a3`

Funding *Loukas Georgiadis*: 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.
Giuseppe F. Italiano: Partially supported by MIUR, the Italian Ministry for Education, University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data).
Evangelos Kosinas: 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).

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 classical subject in graph theory, motivated by several application areas (see, e.g., [20]), 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 [17], u and v are k-edge-connected if and only if there are k-edge-disjoint paths between u and v. A *k-edge-connected component* of G is a maximal set $C \subseteq V$ such that there is no $(k-1)$-edge cut in G that disconnects any two vertices $u, v \in C$ (i.e., u and v are in the same connected component of $G \setminus S$ for any $(k-1)$-edge cut S). We can define, analogously, the *vertex cuts* and the *k-vertex-connected components* of G. It is known how to compute the $(k-1)$-edge cuts, $(k-1)$-vertex cuts, k-edge-connected components and k-vertex-connected components of a graph in linear time for $k \in \{2, 3\}$ [6, 12, 19, 23, 26]. The case $k = 4$ has also received significant attention [2, 3, 13, 14], but until very recently, none of the previous algorithms achieved linear running time. In particular, Kanevsky and Ramachandran [13] showed how to test whether a graph is 4-vertex-connected in $O(n^2)$ time. Furthermore, Kanevsky et al. [14] 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 [25]. Using the reduction of Galil and Italiano [6] from edge connectivity to vertex connectivity, the same bounds can be obtained for 4-edge connectivity. Specifically, one can test whether a graph is 4-edge-connected in $O(n^2)$ time, and one can compute the 4-edge-connected components of a 3-edge-connected graph in $O(m + n\alpha(m, n))$ time. Dinitz and Westbrook [3] presented an $O(m + n \log n)$-time algorithm to compute the 4-edge-connected components of a general graph G (i.e., when G is not necessarily 3-edge-connected). Nagamochi and Watanabe [21] gave an $O(m + k^2n^2)$-time algorithm to compute the k-edge-connected components of a graph G, for any integer k.

Very recently, linear-time algorithms for computing the 4-edge-connected components of an undirected graph were presented in [8, 18]. Specifically, Nadara, Radecki, Smulewicz, and Sokołowski [18] gave two linear-time algorithms, a simple randomized and a more complicated deterministic algorithm. Georgiadis, Italiano, and Kosinas [8] 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 [23], 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 linear time, both NRSS [18] and GIK [8] require the use of the static tree disjoint-set-union (DSU) data structure of Gabow and Tarjan [5], 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 [11]. NRSS [18] 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 [8], 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.

Our contribution. We present an improved version of the algorithm of GIK [8] for identifying type-2 cuts, so that it only uses simple data structures. The resulting algorithm relies only on basic properties of depth-first search (DFS) [23], 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.** *The 3-edge cuts of an undirected graph can be computed in linear time on a pointer machine.*

▶ **Corollary 2.** *The 4-edge-connected components of an undirected graph can be computed in linear time on a pointer machine.*

Next, we consider the practical performance of these algorithms. We provide carefully engineered implementations of the randomized algorithm of NRSS [18], the deterministic algorithm of GIK [8], 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 $k < 4$?

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

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 [6, 12, 19, 23, 26]. 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 [2], 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 [10, 26]. 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 [2], 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 [2]. 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 [8] for the details.) It remains to describe how to compute the 3-edge cuts of a 3-edge-connected graph G.

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

In this section we give an overview of the algorithms of NRSS [18] and GIK [8] 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 graph with n vertices and $m \geq 3n/2$ edges, and may have multiple edges. It is well-known that the number of the 3-edge-cuts of G is $O(n)$ [20], which also follows from the definition of the cactus graph [1, 15]). See also [8, 18] 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 Depth-first search and related notions

Here we introduce some concepts and parameters that are used in the algorithms, defined with respect to a depth-first search spanning tree. Let T be the spanning tree of G provided by a depth-first search (DFS) of G [23], 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 $v \leq u < v + ND(v)$ [24].

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** ([8]). *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 $v \neq r$.*

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 $\emptyset$ 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 [23], the triconnected components [12] and the 3-edge-connected components [6, 12, 19, 26] of a graph. (Hopcroft and Tarjan [12] 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

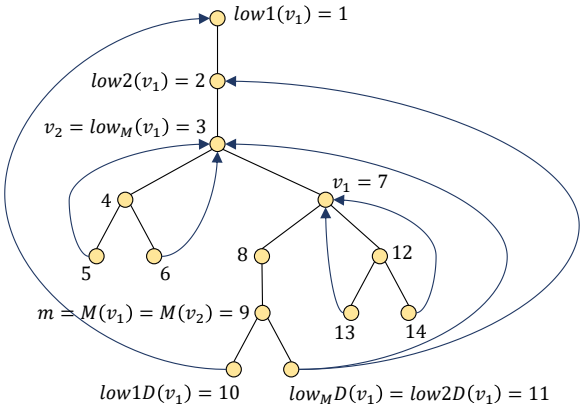

Figure 1 An illustration of the concepts defined on a depth-first search (DFS) spanning tree of an undirected graph. Vertices are numbered in DFS order and back-edges are shown directed from descendant to ancestor in the DFS tree. Vertices v_1 and v_2 have $M(v_1) = M(v_2) = m$, hence $\{v_1, v_2\} \subseteq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

linear-time in pointer machines [8]. For the computation of all $high(v)$ (and $highD(v)$), [8] and [18] gave a linear-time algorithm that uses the static tree DSU data structure of Gabow and Tarjan [5], and thus their computation depends on the power of RAM machines.

3.2 Randomized algorithm of NRSS

We give an overview of the randomized linear-time algorithm of Nadara, Radecki, Smulewicz, and Sokołowski [18]. 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 [5]. (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 spend 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, Italiano, and Kosinas [8]. 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 [8] 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. We need not mention any details of those cases because we can avoid them overall using the idea of NRSS for identifying all type-3 cuts on the contracted graph.

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 (since the only known algorithm for computing all *high* points in linear time depends on the power of RAM machines [8, 18]). 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 the previous subsection). 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 $nextM(v) \neq \emptyset$. 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 $B(v) \setminus B(nextM(v))$. (Refer to Figure 1.) 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 $(x, low_M(v)) \in B(v)$.

All edges $(low_M D(v), low_M(v))$, for all vertices v for which they are defined, can be determined with a linear-time algorithm that uses only elementary data structures and depends on a specific ordering of the adjacency lists of the graph. Now, if we have a pair of vertices $\{u, v\}$ such that $B(u) = B(v) \sqcup \{e\}$, for a back-edge e, and $v = nextM(u)$, then it is certainly true that $e = (low_M D(u), low_M(u)) = (highD(u), high(u))$. This shows how to handle one of the three subcases of the lower type-2 cuts, described in the previous Section. For the other two subcases, as well as for the upper case of type-2 cuts, we can simply provide alternative criteria that characterize them, using most of the DFS notions in Section 3.1 in a way similar to [8] (but without using the *high* points). The algorithms that we get in this way are very similar to those given by GIK.

Due to the space constraints, we refer to [9] for a detailed description of those criteria and their proofs, as well as for an exposition and proof of correctness of the full algorithm for determining type-2 cuts. Throughout the remainder of the paper, we refer to this algorithm as Linear.

4 Empirical Analysis

We implemented our algorithms in `C++`, using `g++ 7.5.0` with full optimization (flag -O4) to compile the code. 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 192GB of RAM memory. The processor has 24.75MB 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 ten different runs.

Implementation details

We tested four implementations, two for the randomized algorithm of Nadara, Radecki, Smulewicz, and Sokołowski [18] (NRSS and NRSS-sort), one for the deterministic algorithm of Georgiadis, Italiano, and Kosinas [8] (GIK), and one for our linear-time pointer machine algorithm (Linear). We did not include an implementation of the deterministic algorithm of [18], because it is more complicated than the routine in GIK for computing type-2 cuts, and it uses data structures for off-line 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 [18]. 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 it 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 the deterministic algorithm of [18].) 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 [18], but it only slightly affects the total running-time, since that 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 [18]), 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 behaviour.

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 $k' \leq k$, 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 [20]. 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. The running times of the algorithms for the original graphs and for their sparse certificates are shown in Table 2 and Table 3 respectively.[1] Figure 2 and 3 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 $k \in \{1, 2, 3, 4\}$.

Table 1 Characteristics of real-world graphs taken from the SNAP [16]; n is the number of vertices, m is the number of edges, and $\#k$CCs 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: social (SN), product (PN), collaboration, (CN), road (RN), and internet topology (IT).

Graph	n	m	#1CCs	#2CCs	#3CCs	#4CCs	m'	type
Deezer-HR	54573	498202	1	2436	5188	7878	194107	SN
Facebook-Artist	50515	819306	1	3220	6425	9144	179349	SN
com-Amazon	548552	925872	213690	244981	284574	335963	884127	PN
Gowalla	196591	950327	65294	112349	135493	145544	325636	SN
com-DBLP	425957	1049866	108878	155981	217842	268497	826192	CN
com-Youtube	1157828	2987624	22939	690029	860753	941713	1999435	SN
roadNet-CA	1971281	5533214	8713	8713	385230	385230	5520326	RN
twitch-gamers	168114	6797557	1	4081	7975	11800	648245	SN
as-Skitter	1696415	11095298	756	232897	493601	680010	5181377	IT
com-LiveJournal	3997962	34681189	38577	860464	1292384	1587631	2519167	SN
com-Orkut	3072627	117185083	187	67981	111261	149632	11953289	SN
com-Friendster	124836180	1806067135	59227815	73213653	79168479	82884698	203718281	SN

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 [23]. For $k = 3$ we use the algorithm of GK, which is the simplest and fastest known for computing 2-edge cuts and 3-edge-connected components [7, 10]. For every 3-edge-connected component that we compute, we construct the auxiliary 3-edge-connected graph that corresponds to it (which essentially maintains its k-edge cuts, for all $k \geq 3$) [2], 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 [18] to compute the cactus tree.

[1] We note that we did not run the algorithms on the original com-Friendster graph, because its number of edges is too big for our implementations, due to our choice of variable types.

Table 2 Running times in seconds of the algorithms for the real-world graphs of Table 1. 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.

Graph	NRSS-sort		NRSS		GIK		Linear	
	total	4ECCs	total	4ECCs	total	4ECCs	total	4ECCs
Deezer-HR	0.163	0.107	0.177	0.119	**0.107**	**0.049**	0.109	0.052
Facebook-Artist	0.223	0.153	0.229	0.158	**0.128**	**0.059**	0.136	0.066
com-Amazon	0.596	0.362	0.640	0.403	**0.416**	**0.177**	0.442	0.204
Gowalla	0.241	0.152	0.249	0.159	**0.158**	**0.065**	0.163	0.074
com-DBLP	0.533	0.322	0.575	0.362	**0.374**	**0.158**	0.387	0.174
com-Youtube	1.424	0.829	1.450	0.844	**0.949**	**0.339**	1.009	0.407
roadNet-CA	3.266	2.128	3.381	2.244	**1.984**	**0.849**	2.058	0.925
twitch-gamers	3.026	2.294	2.325	1.617	**1.265**	**0.521**	1.431	0.706
as-Skitter	5.676	3.918	5.717	3.961	**3.321**	**1.561**	3.645	1.883
com-LiveJournal	30.310	21.203	29.270	20.019	**17.327**	**8.056**	19.805	10.637
com-Orkut	94.435	70.280	85.745	61.550	**43.171**	**19.172**	55.081	30.316

From the running times reported in Tables 2 and 3, and plotted in Figures 2 and 3, 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 7 times faster on the sparse certificate, while for com-LiveJournal, they run about 20 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 4 passes over the edges of the graph) and apply any of those algorithms on it, than apply the algorithm directly on the original graph.

We observe that the running time of the Linear algorithm is very close to that of GIK. Indeed, GIK runs faster than Linear by about 7.5% on the original graphs and by about 3% on the sparse certificates. 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 RAM words for every edge.

Table 3 Running times in seconds of the algorithms for the sparse certificates of the real-world graphs of Table 1. 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.

Graph	NRSS-sort		NRSS		GIK		Linear	
	total	4ECCs	total	4ECCs	total	4ECCs	total	4ECCs
Deezer-HR-SC	0.098	0.061	0.105	0.069	0.070	0.032	**0.069**	**0.032**
Facebook-Artist-SC	0.086	0.054	0.092	0.060	**0.059**	**0.026**	0.060	0.028
com-Amazon-SC	0.585	0.352	0.620	0.388	**0.418**	**0.182**	0.430	0.198
Gowalla-SC	0.124	0.070	0.133	0.079	**0.093**	**0.037**	0.093	0.038
com-DBLP-SC	0.486	0.288	0.508	0.312	**0.343**	**0.141**	0.349	0.153
com-Youtube-SC	1.037	0.544	1.082	0.591	**0.770**	**0.260**	0.791	0.295
roadNet-CA-SC	3.296	2.140	3.403	2.246	**2.018**	**0.856**	2.073	0.922
twitch-gamers-SC	0.307	0.203	0.326	0.221	**0.193**	**0.088**	0.200	0.096
as-Skitter-SC	3.643	2.432	3.830	2.614	**2.336**	**1.106**	2.445	1.236
com-LiveJournal-SC	1.389	0.773	1.419	0.811	**0.958**	**0.338**	0.992	0.383
com-Orkut-SC	16.216	11.422	16.386	11.789	**9.943**	**5.342**	10.597	6.014
com-Friendster-SC	515.693	323.119	551.180	357.877	**376.956**	**184.727**	413.305	220.934

Experimental results for random graphs

Finally, 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 [4] 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. [22] to optimally increase the connectivity by one. The corresponding plots are shown in in Figure 4. In these experiments we notice that the GIK algorithm 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. We also notice that the NRSS-sort algorithm has, in fact, somewhat worse performance than NRSS on dense graphs.

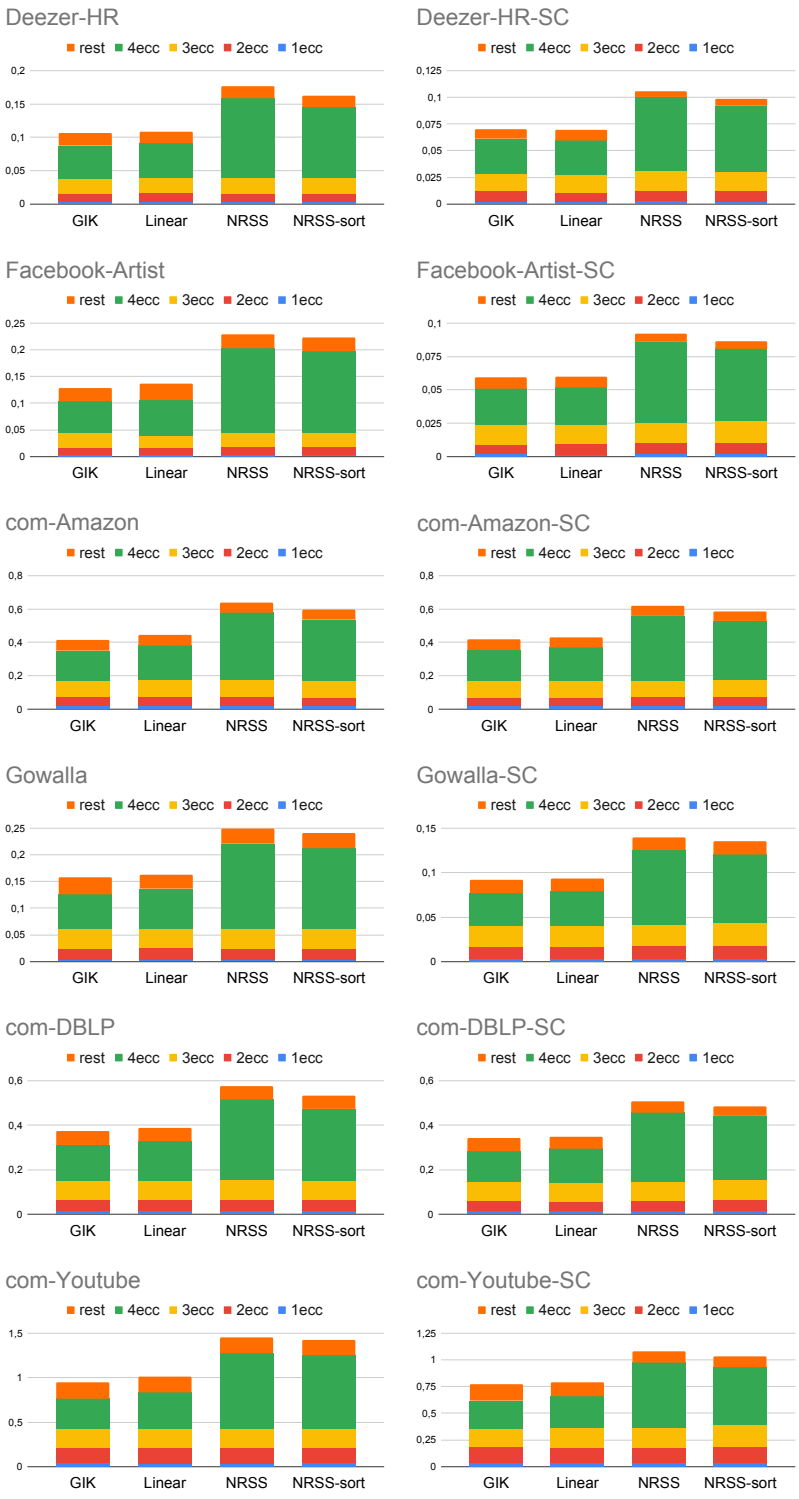

Figure 2 Detailed running times for each part of the 4-edge-connected components algorithms on the first six real-world graphs of Table 1 and their sparse certificates. For each algorithm, we plot the running time for computing the $(k-1)$-edge-cuts and the corresponding k-connected components for $k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

Figure 3 Detailed running times for each part of the 4-edge-connected components algorithms on the last six 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 $k \in \{1, 2, 3, 4\}$.

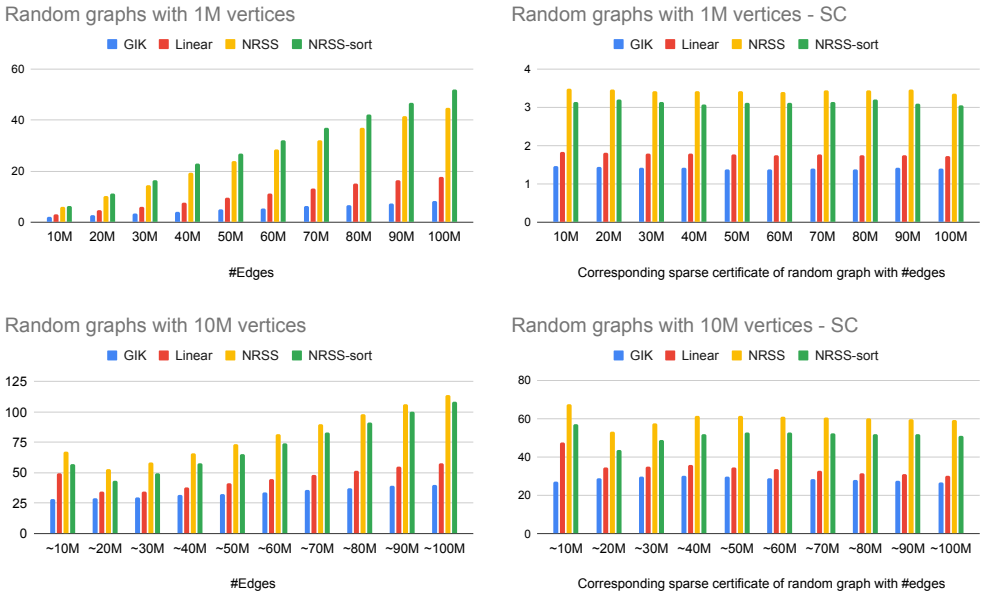

Figure 4 Running times in seconds for random graphs with 1M and 10M vertices and their corresponding sparse certificates.

References

1. E. A. Dinitz, A. V. Karzanov, and M. V. Lomonosov. On the structure of a family of minimal weighted cuts in a graph. *Studies in Discrete Optimization (in Russian)*, pages 290–306, 1976.
2. Y. Dinitz. The 3-edge-components and a structural description of all 3-edge-cuts in a graph. In *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '92, pages 145–157, Berlin, Heidelberg, 1992. Springer-Verlag.
3. Y. Dinitz and J. Westbrook. Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica*, 20:242–276, 1998. doi:10.1007/PL00009195.
4. K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976. doi:10.1137/0205044.
5. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.
6. Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, March 1991. doi:10.1145/122413.122416.
7. L. Georgiadis, K. Giannis, G. F. Italiano, and E. Kosinas. Computing vertex-edge cut-pairs and 2-edge cuts in practice. In David Coudert and Emanuele Natale, editors, *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*, volume 190 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.SEA.2021.20.
8. L. Georgiadis, G. F. Italiano, and E. Kosinas. Computing the 4-edge-connected components of a graph in linear time. In *Proc. 29th European Symposium on Algorithms*, 2021. Full version available at https://arxiv.org/abs/2105.02910.
9. L. Georgiadis, G. F. Italiano, and E. Kosinas. Improved linear-time algorithm for computing the 4-edge-connected components of a graph, 2021. doi:10.48550/ARXIV.2108.08558.
10. L. Georgiadis and E. Kosinas. Linear-Time Algorithms for Computing Twinless Strong Articulation Points and Related Problems. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation (ISAAC 2020)*, volume 181 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ISAAC.2020.38.

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

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

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

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

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

16 J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

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

18 W. Nadara, M. Radecki, M. Smulewicz, and M. Sokołowski. Determining 4-edge-connected components in linear time. In *Proc. 29th European Symposium on Algorithms*, 2021. Full version available at https://arxiv.org/abs/2105.01699.

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

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

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

22 D. Naor, D. Gusfield, and C. Martel. A fast algorithm for optimally increasing the edge connectivity. *SIAM Journal on Computing*, 26(4):1139–1165, 1997.

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

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

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

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