Computing Vertex-Edge Cut-Pairs and 2-Edge Cuts in Practice

Loukas Georgiadis, Konstantinos Giannis, Giuseppe F Italiano, Evangelos Kosinas

Computing Vertex-Edge Cut-Pairs and 2-Edge Cuts in Practice

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

Konstantinos Giannis
Gran Sasso Science Institute, L'Aquila, Italy

Giuseppe F. Italiano
LUISS University, Rome, Italy

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

Abstract

We consider two problems regarding the computation of connectivity cuts in undirected graphs, namely identifying vertex-edge cut-pairs and identifying 2-edge cuts, and present an experimental study of efficient algorithms for their computation. In the first problem, we are given a biconnected graph G and our goal is to find all vertices v such that $G \setminus v$ is not 2-edge-connected, while in the second problem, we are given a 2-edge-connected graph G and our goal is to find all edges e such that $G \setminus e$ is not 2-edge-connected. These problems are motivated by the notion of twinless strong connectivity in directed graphs but are also of independent interest. Moreover, the computation of 2-edge cuts is a main step in algorithms that compute the 3-edge-connected components of a graph. In this paper, we present streamlined versions of two recent linear-time algorithms of Georgiadis and Kosinas that compute all vertex-edge cut-pairs and all 2-edge cuts, respectively. We compare the empirical performance of our vertex-edge cut-pairs algorithm with an alternative linear-time method that exploits the structure of the triconnected components of G. Also, we compare the empirical performance of our 2-edge cuts algorithm with the algorithm of Tsin, which was reported to be the fastest one among the previously existing for this problem. To that end, we conduct a thorough experimental study to highlight the merits and weaknesses of each technique.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases 2-Connectivity, Graph Algorithms, Split Components

Digital Object Identifier 10.4230/LIPIcs.SEA.2021.20

Supplementary Material The source code, together with some sample input instances is available at *Software (Source Code)*: `https://github.com/KoGi89/VECuts`
archived at `swh:1:snp:1f050f2c505784ef144a790438e918ddb790fb41`

Funding Research at the University of Ioannina supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant", Project FANTA (eFficient Algorithms for NeTwork Analysis), number HFRI-FM17-431. G. F. Italiano is partially supported by MIUR, the Italian Ministry for Education, University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data)

1 Introduction

Let $G = (V, E)$ be a connected undirected graph with m edges and n vertices. An edge $e \in E$ is a *bridge* of G if $G \setminus e$ is not connected. Similarly, a vertex $v \in V$ is an *articulation point* of G if $G \setminus v$ is not connected. Graph G is *biconnected* (resp., 2-*edge-connected*) if it has no articulation points (resp., no bridges). Note that if a graph is biconnected then it is necessarily 2-edge-connected. A 2-*edge cut* of G is a pair of edges e and f such that $G \setminus \{e, f\}$ is not connected. A 3-*edge-connected component* of G is a maximal set $C \subset V$ such that there is no 2-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 $G \setminus \{e, f\}$ for any 2-edge cut $\{e, f\}$). A *separation pair* of G is a 2-vertex cut of G, i.e., a pair of vertices u and v such that $G \setminus \{u, v\}$ is not connected. The *triconnected components* of a biconnected graph $G = (V, E)$ is a collection of smaller graphs that describe all the separation pairs in G, as well as the partition of the vertex set V induced by each separation pair [8].

Here we consider two problems regarding the computation of connectivity cuts in undirected graphs, namely identifying vertex-edge cut-pairs and identifying 2-edge cuts, and present an experimental study of efficient algorithms for their computation. In the first problem, we are given a biconnected graph G and our goal is to find all vertices v such $G \setminus v$ is not 2-edge-connected, while in the second problem, we are given a 2-edge-connected graph G and our goal is to find all edges e such that $G \setminus e$ is not 2-edge-connected. These problems are motivated by the notion of twinless strong connectivity in directed graphs [6, 9, 14] but are also of independent interest. Moreover, the computation of 2-edge cuts is a main step in algorithms that compute the 3-edge-connected components of a graph [5, 12, 13, 17]. In this paper, we present streamlined versions of two recent linear-time algorithms of Georgiadis and Kosinas [6] that compute all vertex-edge cut-pairs and all 2-edge cuts, respectively. We note that both algorithms of [6] are based on a common framework applied on a depth-first search (DFS) tree structure T of G that yields algorithms that are conceptually simple, asymptotically optimal, and fast in practice. Furthermore, we believe that it may prove useful in solving efficiently other connectivity problems. We compare the empirical performance of our vertex-edge cut-pairs algorithm with an alternative linear-time method that exploits the structure of the triconnected components of G, that can be represented efficiently by an SPQR tree [1, 2]. Since SPQR trees can be constructed in linear time [7], this approach implies an alternative linear-time algorithm for computing the vertex-edge cut-pairs of G. In order to construct an SPQR tree, however, we need to know the triconnected components of the graph [7], and efficient algorithms that compute triconnected components are considered conceptually complicated, and thus difficult to implement (see, e.g., [4, 7, 8]). Also, we compare the empirical performance of our 2-edge cuts algorithm with the algorithm of Tsin [17], which was previously reported to be the fastest one among the previously existing for this problem. To that end, we conduct a thorough experimental study to highlight the merits and weaknesses of each technique.

2 Preliminaries

Recall that a graph is 2-edge-connected if it contains no bridges. For such a graph G, we say that an edge e is a *cut-edge* if it forms a 2-edge cut together with some other edge. The framework of Georgiadis and Kosinas [6] relies on a classification of the elements we want to compute (e.g. cut-edges or vertices which belong to a vertex-edge cut), applied on a depth-first search (DFS) tree structure T of G. We let $T(v)$ denote the subtree of T rooted at vertex v. This classification is based on the distribution of the back-edges of T, represented by the sets $B(v)$ of the back-edges that start from $T(v)$ and end in an ancestor of v. To see why these sets are useful in determining connectivity relations of G, observe that if we remove a vertex v (which is not a leaf or the root) or the tree-edge $(v, p(v))$, that connects v to its parent $p(v)$ in T, from G, then $T(v)$ remains connected with the rest of the graph only through the back-edges in $B(v)$. Now, we can capture the connectivity information we want from the sets $B(v)$ by considering the higher ends and the lower ends of all back-edges in $B(v)$. Thus we define the nearest common ancestor of the higher ends of all back-edges in $B(v)$, denoted by $M(v)$, and the maximum and minimum lower ends of all back-edges in $B(v)$, denoted by $high(v)$ and $low(v)$, respectively. Using those (and similar) concepts, we can classify the elements we want to compute in such a way that we can provide necessary and sufficient conditions that characterize them and allow us to compute them efficiently.

Concepts defined on a DFS-tree

We consider a DFS traversal of G, starting from an arbitrarily selected vertex r, and let T be the resulting DFS tree [16]. A vertex u is an ancestor of a vertex v (v is a descendant of u) if the tree path from r to v contains u. Thus we consider a vertex to be an ancestor (and, also, a descendant) of itself. We let $p(v)$ denote the parent of a vertex v in T. If u is a descendant of v in T, we denote the set of vertices of the simple tree path from u to v as $T[u, v]$. The expressions $T[u, v)$ and $T(u, v]$ have the obvious meaning (i.e., the vertex on the side of the parenthesis is excluded from the tree path). Recall that $T(v)$ denotes the subtree of T rooted at vertex v. We identify vertices in G by their DFS number, i.e., the order in which they were discovered by the search. Hence, $u \leq v$ means that vertex u was discovered before v. The edges of T are called tree-edges, and the edges of G that are not tree-edges are called back-edges, as their endpoints are related as ancestor and descendant on T. We denote the collection of all back-edges as B. When we write (u, v) to denote a back-edge, we always mean that $v \leq u$, i.e., u is an descendant of v in T. The framework of Georgiadis and Kosinas [6], referred to as GK hereafter, uses the following key concepts that are defined on T:

- $B(v) := \{(x, y) \in B \mid x \in T(v) \text{ and } y < v\}$, the set of all back-edges that start from $T(v)$ and end in a proper ancestor of v.
- $B_p(v) := \{(x, y) \in B \mid x \in T(v) \text{ and } y < p(v)\}$, the set of all back-edges that start from $T(v)$ and end in a proper ancestor of $p(v)$.
- $l(v) := min\{\{v\} \cup \{y \mid (v, y) \in B(v)\}\}$, the lowest vertex that is connected with a back-edge with v (or v if there is no back-edge (v, y)).
- $low(v) := min\{y \mid (x, y) \in B(v)\}$, the lowest lower end of all back-edges in $B(v)$.
- $high(v) := max\{y \mid (x, y) \in B(v)\}$, the highest lower end of all back-edges in $B(v)$.
- $high_p(v) := max\{y \mid (x, y) \in B_p(v)\}$, the highest lower end of all back-edges in $B_p(v)$.
- $M(v) := nca\{x \mid (x, y) \in B(v)\}$, the nearest common ancestor of the higher ends of all back-edges in $B(v)$.
- $M_p(v) := nca\{x \mid (x, y) \in B_p(v)\}$, the nearest common ancestor of the higher ends of all back-edges in $B_p(v)$.
- $b\_count(v) := \#B(v)$, the number of elements of $B(v)$.
- $b_p\_count(v) := \#B_p(v)$, the number of elements of $B_p(v)$.
- $up(v) := \#\{(x, p(v)) \mid x \in T(v)\}$, the number of back-edges that start from $T(v)$ and end in the parent of v.

$B(v)$, $l(v)$, $low(v)$, $high(v)$, $M(v)$, $b\_count(v)$, and $up(v)$ are defined for all vertices $v \neq r$; similarly, $B_p(v)$, $high_p(v)$, $M_p(v)$, and $b_p\_count(v)$ are defined for all vertices $v \notin \{r, r_c\}$, where r_c is the unique child of r, if G is biconnected. Except for $B(v)$ and $B_p(v)$, all these parameters can be computed in total linear-time, for all the vertices on which they are defined (see [6], and also Algorithms 1 and 3 in Appendix A).

3 Computing vertex-edge cut-pairs in linear time

Here we present an overview of linear-time algorithms for computing the vertex-edge cut-pairs of a biconnected graph $G = (V, E)$. All algorithms compute, for each vertex $v \in V$, the number of edges $e \in E$ such that $G \setminus \{v, e\}$ is not connected. The corresponding values are stored in variables $\mathit{count}(v)$. We first describe the algorithm of Georgiadis and Kosinas [6], which operates on a DFS tree of T. Next, we provide a streamlined version that enhances its performance in practice. Finally, we describe how to compute the $\mathit{count}(v)$ values using a SPQR tree of G, and describe a simplification of this approach that only computes the relevant nodes of the SPQR tree.

3.1 Computing vertex-edge cut-pairs via the GK framework

We give an overview of the algorithm in [6] for computing all vertices that belong to a vertex-edge cut in a biconnected graph G. This algorithm computes, for every vertex v, the number of edges e such that $G \setminus \{v, e\}$ is not connected. It works by classifying the vertex-edge cuts on the DFS tree in such a way that we can provide an efficient method to count the number of vertex-edge cuts of each type.

Let T be a DFS-tree of G rooted at r, and let $\{v, e\}$ be a vertex-edge cut-pair. We distinguish three cases, depending on the location of e relative to v on T: e can either be a back-edge, or a tree-edge of the form $(u, p(u))$, with u a proper ancestor of v, or a tree-edge of the form $(u, p(u))$, with u a proper descendant of a child of v.

If e is a back-edge, then there exists a child c of v, such that e connects $T(c)$ with $T(v, r]$ and is the only back-edge with this property. Thus, for every vertex v, the number of cut-pairs of the form $\{v, e\}$, where e is a back-edge, cannot be greater than the number of children of v, and we can find all these cut-pairs explicitly: We only have to count, for every vertex v, the number of its children c that have $b_p\_count(c) = 1$. All $b_p\_count(c)$, for every vertex c, can be computed during the depth-first search (see Algorithm 1 in Appendix A). If for a vertex c we have $b_p\_count(c) = 1$, then $\{p(c), (M_p(c), \mathit{low}(c))\}$ is a cut-pair.

Now, if e is a tree-edge of the form $(u, p(u))$, with u a proper ancestor of v, then every back-edge that starts from $T(u)$ and ends in a proper ancestor of u must start from a descendant of v. This means that $M(u)$ is a descendant of v, and we further distinguish two cases, depending on whether $M(u) = v$ or $M(u)$ is a proper descendant of v. In the first case, u has the property that, for every child c of v, either $u \leq \mathit{low}(c)$ or $u > \mathit{high}_p(c)$; in other words, u does not belong to any set of the form $T[\mathit{high}_p(c), \mathit{low}(c))$, for any child c of v. (And conversely: if u has this property, and $M(u) = v$, then $\{v, (u, p(u))\}$ is a cut-pair.) Thus, we can find all vertex-edge cut-pairs of this type explicitly: we only have to find, for every vertex v, all elements in $M^{-1}(v)$ that do not belong to any set of the form $T[\mathit{high}_p(c), \mathit{low}(c))$, for any child c of v. Now, if $M(u)$ is a proper descendant of v, it is a descendant of a child c of v. In this case, we have $M_p(c) = M(u)$, and every back-edge that starts from $T(c)$ and ends in a proper ancestor of $v = p(c)$ must end in a proper ancestor of u, and therefore $\mathit{high}_p(c) < u$. (The converse is also true: if u is a proper ancestor of $p(c)$ such that $M_p(c) = M(u)$ and $\mathit{high}_p(c) < u$, then $\{p(c), (u, p(u))\}$ is a cut-pair.) Thus, to count all vertex-edge cut-pairs of this type, it is sufficient to focus our attention on the lists $M_p^{-1}(m)$ and $M^{-1}(m)$, for every vertex m, and count all pairs $(c, u) \in M_p^{-1}(m) \times M^{-1}(m)$, such that u is a proper ancestor of $p(c)$ with $\mathit{high}_p(c) < u$. To do this efficiently, we exploit the following fact: If c is in $M_p^{-1}(m)$, and $U(c)$ is the set of all vertices u in $M^{-1}(m)$ such that u is a proper ancestor of $p(c)$ with $\mathit{high}_p(c) < u$, then, if c' is also in $M_p^{-1}(m)$ and $c' \in T[c, \mathit{high}_p(c))$, we have $U(c') = U(c) \cap T(c', \mathit{high}_p(c))$. (For details, see Algorithm "$M(u) > v$" in [6].)

Finally, if e is a tree-edge of the form $(u, p(u))$, with u a proper descendant of a child of v, then every back-edge that starts from $T(u)$ and ends in a proper ancestor of u must end in an ancestor of v. This means that $\mathit{high}(u)$ is an ancestor of v, and we further distinguish two cases, depending on whether $\mathit{high}(u) = v$ or $\mathit{high}(u)$ is a proper ancestor of v. In the first case, we can find all cut-pairs explicitly: we only have to find, for every vertex v, all u in $\mathit{high}^{-1}(v)$ that are not children of v and are such that either $\mathit{low}(u) = v$ or $\mathit{low}(u) < v$ and $M_p(c)$ is in $T(u)$, where c is the child of v which is an ancestor of u (see Algorithm "$\mathit{high}(u) = v$" in [6]). If $\mathit{high}(u)$ is a proper ancestor of v, then $M(u) = M_p(c)$, where c is the child of v which is an ancestor of u. (Conversely: if u and c are such that u is a proper descendant of c with $M(u) = M_p(c)$ and $\mathit{high}(u) < p(c)$, then $\{p(c), (u, p(u))\}$ is a cut-pair.) Thus, to count all vertex-edge cut-pairs of this type, it is sufficient to focus our attention on the lists $M_p^{-1}(m)$ and $M^{-1}(m)$, for every vertex m, and count all pairs $(c, u) \in M_p^{-1}(m) \times M^{-1}(m)$, such that u is a proper descendant of c with $\mathit{high}(u) < p(c)$. To do this efficiently, we exploit the following fact: If c is in $M_p^{-1}(m)$ and $U(c)$ is the set of all vertices u in $M^{-1}(m)$ such that u is a proper descendant of c with $\mathit{high}(u) < p(c)$, then all u in $U(c)$ have the same high point, call it h, and, if c' is also in $M_p^{-1}(m)$ and such that $c \geq c'$ and $h < p(c')$, then $U(c') = U(c) \cup (T[c, c') \cap M^{-1}(u))$.

We refer to the algorithm of [6] as GK-VE.

Streamlined version

Now we describe our improvements that both simplify the algorithm of [6] and also make it faster in practice. First, in order to compute all $b_p\_count(v)$, [6] suggests a recursive algorithm, which depends upon a specific sorting of the list of back-edges. Here, we observe that we can compute these values directly during the DFS of G, together with all $\mathit{up}(v)$, as shown in Algorithm 1 (see Appendix A). This works as follows. By definition, $b_p\_count(v)$ is the number of back-edges of the form (x, y), where x is a descendant of v and y is a proper ancestor of $p(v)$. Therefore, $b_p\_count(v) = b_p\_count(c_1) + \ldots + b_p\_count(c_k) + \#\{(v, y) \in B(v)\} - \#\{(x, p(v)) \mid x \in T(v)\}$, where $c_1, \ldots, c_k$ are the children of v (if it has any). Thus, when we process a vertex v and u is in the adjacency list of v, we set $b_p\_count(v) \leftarrow b_p\_count(v) + b_p\_count(u)$ if u is a child of v, or $b_p\_count(v) \leftarrow b_p\_count(v) + 1$ if u is an ancestor of v but not its parent. Now, to compute all $\mathit{up}(v) := \#\{(x, p(v)) \mid x \in T(v)\}$, we keep a variable $\mathit{tempChild}(v)$, for every vertex v, which is set to be the child of v in which we have currently descended during the DFS. Then, when we process a vertex x and v is in the adjacency list of x, and also v is an ancestor of x but not its parent, we set $\mathit{up}(\mathit{tempChild}(v)) \leftarrow \mathit{up}(\mathit{tempChild}(v)) + 1$.

A second important improvement comes from the fact that [6] uses both the $\mathit{high}(v)$ and the $\mathit{high}_p(v)$ values, while we can observe that it suffices to use only the latter. Indeed, if $\{v, (u, p(u))\}$ is a vertex-edge cut-pair such that u is a descendant of v, then u is a proper descendant of a child c of v and $\mathit{high}(u) \leq v = p(c)$; thus we have $\mathit{high}(u) = \mathit{high}_p(u)$, since $\mathit{high}(u) < c \leq p(u)$. Then, we only have to make sure that every time we discover a vertex-edge cut $\{v, (u, p(u))\}$ of this type, we have $\mathit{high}(u) = \mathit{high}_p(u)$. This is the case if and only if there is no back-edge $(x, p(u))$ with $x \in T(u)$, which can be checked simply by testing whether $\mathit{up}(u) = 0$.

We refer to the above version as GK-VE-S.

3.2 Computing vertex-edge cut-pairs via SPQR trees

Here we describe how to compute the number of vertex-edge cut-pairs in linear time via SPQR trees [1, 2]. An SPQR tree $\mathcal{T}$ for a biconnected graph G represents the triconnected components of G. Each node $\alpha \in \mathcal{T}$ is associated with an undirected graph or multigraph G_α. Each vertex of G_α corresponds to a vertex of the original graph G. An edge of G_α is either a *virtual edge* that corresponds to a separation pair of G, or a *real edge* that corresponds to an edge of the original graph G. The node α, and the graph G_α associated with it, has one of the following types:

- If α is an S-node, then G_α is a cycle graph with three or more vertices and edges.
- If α is a P-node, then G_α is a multigraph with two vertices and at least 3 parallel edges.
- If α is a Q-node, then G_α is a single real edge.
- If α is an R-node, then G_α is a simple triconnected graph.

Each edge $\{\alpha, \beta\}$ between two nodes of the SPQR tree is associated with two virtual edges, where one is an edge in G_α and the other is an edge in G_β. If $\{u, v\}$ is a separation pair in G, then one of the following cases applies (see, e.g., [18]):

(a) u and v are the endpoints of a virtual edge in the graph G_α associated with an R-node α of $\mathcal{T}$.

(b) u and v are vertices in the graph G_α associated with a P-node α of $\mathcal{T}$.

(c) u and v are vertices in the graph G_α associated with an S-node α of $\mathcal{T}$, such that either u and v are not adjacent, or the edge $\{u, v\}$ is virtual.

In case (c), if $\{u, v\}$ is a virtual edge, then u and v also belong to a P-node or an R-node. If u and v are not adjacent then $G \setminus \{u, v\}$ consists of two components that are represented by two paths of the cycle graph G_α associated with the S-node α and with the SPQR tree nodes attached to those two paths. Let $e = \{x, y\}$ be an edge of G such that $\{v, e\}$ is a vertex-edge cut-pair of G. Then, $\mathcal{T}$ must contain an S-node α such that v, x and y are vertices of G_α and $\{x, y\}$ is not a virtual edge.

The above observation implies that we can use $\mathcal{T}$ to identify (and count) all vertex-edge cut-pairs of G. We do that as follows. We initialize $count(v) \leftarrow 0$ for all $v \in V$, and process the S-nodes of $\mathcal{T}$ individually. For each S-node α we count the number m_α of the real edges of G_α. Then, for each vertex $v \in V(G_\alpha)$, we set $count(v)$ equal to $m_\alpha - |\{e \in E : e \text{ is adjacent to } v\}|$.

Gutwenger and P. Mutzel [7] showed that an SPQR tree can be constructed in linear time, by extending the triconnected components algorithm of Hopcroft and Tarjan. Moreover, given $\mathcal{T}$ it is straightforward to compute $count(v)$ in $O(n)$ time, for all vertices v. We refer to this algorithm as SPQR-VE.

We also considered a variant that avoids the computation of a complete SPQR tree. Since we only care about S-nodes, it suffices to compute only these nodes from the partition of the graph into *split components*. These components are formed during the execution of the Hopcroft-Tarjan algorithm as follows. When the algorithm finds a pair of separating vertices u and v, it splits the graph at these two vertices into two smaller graphs, and adds the virtual edge $\{u, v\}$ in both graphs. The split components of G are the graphs that are formed when we repeat this process until no more separating pairs exist. (Note that this partition is not uniquely defined.) A split component can be of three types: a P-component that consist of two vertices joined with multiple edges, an S-component that forms a triangle, or an R-component that is any other split component. Then, the S-nodes of the SPQR tree are formed by merging S-components that share a virtual edge. After we have computed just the S-nodes of the SPQR tree, we can identify the vertex-edge cut-pairs of G as above. We refer to this algorithm as Split-VE.

4 Finding all cut-edges and computing the number of 2-cuts

Here we present an overview of linear-time algorithms for computing the 2-edge cuts of a 2-edge-connected graph $G = (V, E)$. These algorithms compute the cut-edges of G (i.e., the edges that form a 2-edge cut together with some other edge). First, we describe the algorithm of [6], and our streamlined version of it. Then, we give an overview of the algorithm of Tsin [17], which was previously reported to be the fastest one among the previously existing for this problem.

4.1 Computing 2-edge cuts via the GK framework

The algorithm in [6] works on a DFS tree T of G rooted at r. It distinguishes two types of cut-pairs: those consisting of a back-edge and a tree-edge, and those consisting of two tree-edges. The first case is very easy to handle, since we only have to find the vertices $v \neq r$ that have $b\_count(v) = 1$, and mark the edges $(v, p(v))$ and $(M(v), low(v))$ as cut-edges. In the case where $(u, p(u))$, $(v, p(v))$ are two tree-edges, [6] proved the following condition: (1) $\{(u, p(u)), (v, p(v))\}$ is a cut-pair if and only if $M(u) = M(v)$ and $high(u) = high(v)$. Now, let m be a vertex and $u_1, \dots, u_k$ all the vertices in $M^{-1}(m)$ ordered decreasingly. Then we have $high(u_1) \geq \dots \geq high(u_k)$. Thus, by (1), it is sufficient it is sufficient to traverse the decreasingly sorted list $M^{-1}(m)$ from the greatest to the lowest element, and mark the edges $(u, p(u))$ and $(v, p(v))$ that satisfy the following condition: (2) u and v are consecutive vertices in $M^{-1}(m)$ such that $high(u) = high(v)$.

We refer to this algorithm of [6] as GK-2E. We also note that a simple extension of this algorithm computes the 3-edge-connected components of G as in [17].

Streamlined version

Algorithm GK-2E depends on the computation of the *high* points of all vertices $v \neq r$, so that it can check condition (2). Here, however, we observe that we can skip this computation thanks to the following Lemma.

▶ **Lemma 1.** *Let u, v be two vertices ($\neq r$) with $M(u) = M(v)$. Then $high(u) = high(v)$ if and only if $b\_count(u) = b\_count(v)$.*

Proof. ($\Rightarrow$) Let (x, y) be a back-edge such that x is a descendant of u and y is a proper ancestor of u. Since $M(u) = M(v)$, we have that x is a descendant of v. Furthermore, since $y \leq high(u) = high(v) < v$, we have that y is a proper ancestor of v. This shows that $B(u) \subseteq B(v)$, and so we have $b\_count(u) \leq b\_count(v)$. Now, with a reversal of the roles of u and v, we also get $b\_count(v) \leq b\_count(u)$. We conclude that $b\_count(u) = b\_count(v)$.

($\Leftarrow$) Since u and v have a common descendant, we can assume, without loss of generality, that u is a descendant of v. Let (x, y) be a back-edge such that x is a descendant of v and y is a proper ancestor of v. Since $M(u) = M(v)$, we have that x is a descendant of u. Furthermore, since v is an ancestor of u, we have that y is a proper ancestor of u. This shows that $B(v) \subseteq B(u)$. Since $b\_count(u) = b\_count(v)$, it must be the case that $B(v) = B(u)$. Thus we have $high(u) = high(v)$. ◀

Thus we can test condition (2) by checking whether $b\_count(u) = b\_count(v)$, instead of $high(u) = high(v)$. In this way, we can avoid the computation of all *high* points (during which we have to process all the back-edges), and so we get an algorithm which is about twice as fast as the original.

Now, in order to compute the number of 2-edge cuts, we first observe that those consisting of a back-edge and a tree-edge can be found explicitly, since their number can be at most $n-1$ (as they correspond to the vertices $v \neq r$ that have $b\_count(v) = 1$). Now, let $(v, p(v))$ be a tree-edge and $(u_1, p(u_1)), \ldots, (u_k, p(u_k))$ all the tree-edges which form a cut-pair with $(v, p(v))$. Then (1) implies that every $(u_i, p(u_i))$, for $i \in \{1, \ldots, k\}$, forms a cut-pair with $(u_j, p(u_j))$, for every $j \in \{1, \ldots, k\} \setminus \{i\}$. Furthermore, these are all the tree-edges (plus $(v, p(v))$) with which $(u_i, p(u_i))$ forms a cut-pair. Thus, to count the number of those cut-pairs efficiently, we can work as follows. We traverse the decreasingly sorted list $M^{-1}(m)$, for every vertex m, until we reach a vertex v such that $(v, p(v))$ and $(u_1, p(u_1))$ is a cut-pair, where u_1 is the successor of v in $M^{-1}(m)$. Then we keep traversing the list $M^{-1}(m)$, until we reach the lowest vertex u_k such that $(u_k, p(u_k))$ forms a cut-pair with $(v, p(v))$ (i.e. that satisfies $b\_count(u_k) = b\_count(v)$). Then we add the quantity $k(k+1)/2$ to the number of 2-edge cuts, and we repeat the same process until we reach the end of $M^{-1}(m)$.

The entire algorithm which computes the cut-edges and the number of 2-edge cuts is shown in Algorithm 2 (see Appendix A). Variable $nextM(v)$ denotes the successor of v in the decreasingly sorted list $M^{-1}(M(v))$ (or the end-of-list symbol $\emptyset$). It can be computed during the calculation of all $M(v)$, as shown in Algorithm 3 (see Appendix A).

We refer to the above version as GK-2E-S.

4.2 Tsin's algorithm

Tsin's algorithm [17] finds the 3-edge-connected components of a 2-edge-connected graph G. The algorithm consists of two parts, and the first one determines the set E_{cut} that contains all the edges belonging to a cut-pair. In the second part, it processes the edge set E_{cut} in order to form the 3-edge-connected components of G.

Here we describe the first part of the algorithm that is relevant to our problem. The algorithm performs a depth-first traversal in G, and it creates a DFS tree T while separating the edges of G into two sets, the tree-edges belonging in T and the set of back-edges which contains all other edges. Tsin provides the following key definition. A *generator* is a cut-edge $e = (x, y)$, where e is either a back-edge or a tree-edge, and there is no other tree edge in $T(y)$ or back-edge having an end-vertex in $T(y)$ that forms a cut-pair with e. It is shown that every edge in E_{cut} belongs to a cut-pair containing a generator and therefore it suffices to determine the subset of cut-pairs that contain a generator. Moreover, [17] shows that the cut-pairs have a nesting structure, which allows the algorithm to use stacks in order to determine the cut-pairs. Specifically, each vertex v is associated with a stack *stack(v)* that stores entries of the form $[(x, y), T[q, p]]$, where the edge (x, y) is a generator or a potential generator and $T[q, p]$ is a path with edges that may form cut-pairs with (x, y).

The algorithm distinguishes two cases depending on whether the current vertex v that we traverse is a leaf of T or not. In the former case (where v is a leaf), we determine if v is an ending point of an edge that is a generator, and we push a corresponding entry to *stack(v)* if needed. Otherwise, if v is not a leaf, when the traversal returns from a child w of v, we check the top of *stack(w)* to determine if we have found a new cut-pair and update *stack(w)*. When we finish this check for all the children of v, we update *stack(v)* and backtrack. Finally, when the traversal returns to the root of T, all edges belonging in a cut-pair form the set E_{cut}.

We refer to this algorithm as Tsin-2E.

5 Empirical Analysis

We wrote our implementations in `C++`, using `Visual Studio Compiler x64` with maximum optimization to favor speed (flag `/O2`) to compile the code. For computing the SPQR tree in algorithm SPQR-VE, we use the linear-time implementation of Gutwenger and Mutzel [7], which is available within the Open Graph Drawing Framework (OGDF) [3]. Similarly, for computing the split components in algorithm Split-VE, we use the implementation of the Hopcroft-Tarjan algorithm [8] provided in OGDF. In order to make a fair comparison among algorithms GK-VE-S, SPQR-VE and Split-VE, we also provide an OGDF-based implementation of GK-VE-S that we refer to as GK-VE-SF. Specifically, GK-VE-SF uses the OGDF representation of graphs, as well basic data structures such as arrays, lists and stacks.

Our main experiments were performed using the following setting: (I) A Dell Precision Tower 7820 CTO Base machine running Red Hat Enterprise 6, equipped with an Intel Xeon E5-2430 2.5GHz processor with 15MB L3 cache and 96GB DDR4 RAM at 2666 MHz. For the OGDF-based implementations we used a less powerful setting: (II) A Dell G5 15 Laptop running Windows 10 (Home 64 bit), equipped with an 8th Generation Intel Core i5 8300H 4GHz processor with 8 MB L3 cache and 8GB DDR4 RAM at 2,666 MHz. We used setting (II) because we did not have physical access to the server of (I) in order to install OGDF. OGDF also requires `CMake` (version 3.1+) a C++11 compliant compiler and GNU Make. We observed that the OGDF library was not able to compute the SPQR trees of graphs with more than ~ 45000 vertices. (This problem was also reported in [10].) This is due to the use of recursion in three routines (DFS, pathFind, and pathSearch) in the implementation of the Hopcroft-Tarjan triconnected components algorithm (which is contained in file "Triconnectivity.cpp"). To fix this, we replaced the recursion with stacks. After this modification, we were able to handle graphs with millions of vertices and edges.

We did not use any parallelization in either setting, and each algorithm ran on a single core. We report CPU times measured with the `std::chrono::steady_clock::now` function.

Real-world graphs

Table 1 shows some statistics of the graphs used in our experimental evaluation. We include both undirected and directed graphs, since one of our motivating applications (twinless strong connectivity) deals with directed graphs. We convert these directed graphs to undirected by ignoring edge directions. Furthermore, we augment these graphs so that they become biconnected, by applying the following procedure. Let G be the input graph. Firstly, we compute the connected components $C_1, \ldots, C_k$ of G, and we join them in a path, by adding an edge connecting a vertex in C_i to a vertex in C_{i+1}, for every $i \in \{1, \ldots, k-1\}$. Let G' be the resulting graph. Then, we compute the leaves $B_1, \ldots, B_l$ of the tree representation of the biconnected components of G', and we connect them in a path. To that end, for every $i \in \{1, \ldots, l-1\}$, we add an edge connecting a vertex $x_i \in B_i$ to a vertex $x_{i+1} \in B_{i+1}$, such that neither x_i nor x_{i+1} is an articulation point.

The corresponding experimental results for setting (I) are given in Table 2 and plotted in Figure 1, while Table 3 shows the results using setting (II). (The memory consumption of the algorithms tested in setting (I) is reported in Table 7 in Appendix B.) First, we make some remarks about the performance of our improved versions of the algorithms of [6]. In Table 2 we observe that our streamlined version GK-VE-S is consistently faster than GK-VE, and improves its running time by 16% up to 44%. The improvement obtained by our streamlined version is even more prominent in the case of 2-edge cuts. Specifically, GK-2E-S is consistently faster than GK-2E by 33% up to 47%.

Table 1 Graph instances used in the experiments. The original graphs are taken from [11] and [15], and were converted to biconnected graphs by adding m_e extra edges. The graph categories are: web graph (WG), network with ground truth communities (NGT), dynamic network (DN), collaboration network (CLN), heterogeneous network (HN), recommendation network (RN), dimacs10 (D10), interaction network (IN) and brain network (BN) Undirected graphs are indicated by U and directed graphs are indicated by D. The number of vertices n and edges m refer to the produced instances; n_c is the number of vertices that form a vertex-edge cut pair with at least one edge.

		Graph Details				
Graph	**Type**	n	m	m_e	n_c	**Reference**
Amazon0302	WG (D)	262111	906735	6946	12438	SNAP [11]
com-amazon	NGT (U)	548551	1168192	242323	267685	SNAP [11]
com-dblp	NGT (U)	425957	1219125	169262	166497	SNAP [11]
web-NotreDame	WG (D)	325729	1252708	162601	16475	SNAP [11]
web-Stanford	WG (D)	281903	2008593	15960	25658	SNAP [11]
Amazon0601	WG (D)	403394	2455710	12305	27772	SNAP [11]
ia-yahoo-messages	DN (U)	3157315	3745264	3157299	3073682	NR [15]
web-Google	WG (D)	916428	4523768	201720	127081	SNAP [11]
ca-cit-HepTh	CLN (D)	2673133	5117848	2673049	2665102	NR [15]
cit-HepTh	CLN (U)	2673133	5117859	2673060	2665080	NR [15]
visualise-us	HN (U)	3247673	6495338	3247664	2669435	NR [15]
web-BerkStan	WG (D)	685230	6693612	44143	50673	SNAP [11]
ca-IMDB	CLN (U)	3782456	7564896	3782448	2902605	NR [15]
ca-cit-HepPh	CLN (D)	4596803	7745139	4596691	4584870	NR [15]
cit-HepPh	CLN (U)	4596803	7745148	4596700	4584710	NR [15]
amazon-ratings	RN (U)	5838027	11581127	5837994	3708476	NR [15]
hugetrace-00000	D10 (U)	6879133	13758157	6879023	2307134	NR [15]
rgg-n-2-20-s0	D10 (U)	6891620	13783038	6891417	5859427	NR [15]
wiki-user-edits-page	IN (U)	8998641	14571201	8998616	6930358	NR [15]
hugetric-00010	D10 (U)	9885854	19771559	9885704	3309476	NR [15]
delaunay-n22	D10 (U)	12582869	25165521	12582651	8404913	NR [15]
co-papers-dblp	D10 (U)	15245729	30491160	15245430	14721451	NR [15]
co-papers-citeseer	D10 (U)	16036720	32073082	16036361	15618987	NR [15]
packing-b050	D10 (U)	17488243	34975965	17487763	26	NR [15]
human-Jung2015	BN (U)	1827166	52455284	1827112	1463704	NR [15]
rgg-n-2-23-s0	D10 (U)	8388608	71889991	8388597	2	NR [15]

Next, we compare GK-2E-S to Tsin-2E. Here, we observe that the two algorithm have similar performance, with GK-2E-S being 4% up to 24% faster than Tsin-2E on all but two instances. Moreover, in all instances GK-2E-S uses less memory than Tsin-2E. (See Table 7 in Appendix B.)

Now we turn to the OGDF-based implementations evaluated in setting (II). In Table 3 we report the running times and memory consumption of the corresponding algorithms. Notice that since setting (II) had limited RAM memory (8GB), we only included instances such that the memory consumption of all algorithms did not exceed the available capacity. First, we compare the performance of SPQR-VE and Split-VE. Here we notice that the computation of the full SPQR tree incurs a small overhead over the computation of just the split components (followed by merging S-components that share a virtual edge to form the S-nodes). Indeed, Split-VE is about 17% faster than SPQR-VE and consumes about 14% less memory on average. On the other hand, it is evident that both SPQR-VE and Split-VE are not competitive against GK-VE-SF. Indeed, GK-VE-SF is faster than Split-VE (resp., SPQR-VE) by a factor larger than 4 (resp., 4.5) and requires 1.9 (resp., 2.26) times less memory on average.

Finally, by comparing Table 3 to Table 7 (given in Appendix B), it is worth noticing that GK-VE-SF requires 6 times more memory space than GK-VE-S on average. This is because GK-VE-SF is implemented using the dynamic data structures of OGDF, in order to make a fair comparison with SPQR-VE and Split-VE. On the other hand, our implementation of GK-VE-S (as well as of all other algorithms tested in setting (I)), uses a much more compact representation of the input graph with just 2 static arrays. We remark that it is not possible to employ this compact representation in SPQR-VE and Split-VE, since for the computation of the split components we need to manipulate the adjacency lists and insert virtual edges.

Table 2 Running times in seconds for the graphs of Table 1 in experimental setting (I). The best results in each row are marked in bold.

	2-edge cuts			vertex-edge cuts	
Graph	GK-2E	GK-2E-S	Tsin-2E	GK-VE	GK-VE-S
Amazon0302	0.16	**0.10**	0.13	0.33	**0.25**
com-amazon	0.33	**0.20**	0.23	0.66	**0.55**
com-dblp	0.26	**0.16**	0.18	0.54	**0.42**
web-NotreDame	0.14	**0.08**	0.09	0.29	**0.19**
web-Stanford	0.25	**0.13**	0.16	0.51	**0.34**
Amazon0601	0.37	**0.21**	0.25	0.77	**0.54**
ia-yahoo-messages	1.22	**0.82**	0.88	2.57	**2.15**
web-Google	0.95	**0.52**	0.60	1.92	**1.37**
ca-cit-HepTh	1.08	**0.70**	0.81	2.35	**1.80**
cit-HepTh	1.08	**0.70**	0.82	2.34	**1.78**
visualise-us	1.60	**0.99**	1.14	3.29	**2.54**
web-BerkStan	0.52	**0.28**	0.30	1.12	**0.64**
ca-IMDB	2.15	**1.33**	1.44	4.41	**3.43**
ca-cit-HepPh	1.78	**1.16**	1.36	3.87	**3.01**
cit-HepPh	1.78	**1.16**	1.33	3.84	**2.98**
amazon-ratings	4.03	**2.46**	2.65	8.32	**6.43**
hugetrace-00000	5.74	**3.52**	3.67	12.66	**9.35**
rgg-n-2-20-s0	3.57	**2.22**	2.55	7.34	**5.64**
wiki-user-edits-page	4.42	**2.79**	3.24	9.37	**7.36**
hugetric-00010	8.75	5.37	**5.33**	21.19	**14.39**
delaunay-n22	8.55	**5.34**	6.41	19.49	**14.12**
co-papers-dblp	6.32	**4.08**	4.53	13.58	**10.32**
co-papers-citeseer	6.51	**4.18**	4.73	14.15	**10.75**
packing-b050	8.87	**5.66**	6.42	7.86	**5.21**
human-Jung2015	4.98	**2.65**	3.53	11.08	**6.16**
rgg-n-2-23-s0	19.03	11.34	**10.56**	42.12	**29.98**

Artificial graphs

In order to test the robustness of our algorithms and to obtain a better view of their relative performance, we also conducted experiments with artificial graphs that we produced in the following manner. We construct a biconnected graph by connecting cyclic and complete graphs in a tree-like structure. This allows us to control the number of vertex-edge and 2-edge cuts, as well as the density of the graph. Our generator receives as inputs the number of cyclic and complete graphs, denoted by C and K respectively, and the number of vertices

Table 3 Running times in seconds and memory usage of the OGDF-based implementations for the graphs of Table 1 in experimental setting (II). The best results in each row are marked in bold.

Graphs	**Running Times**			**RAM memory consuption**		
	GK-VE-SF	SPQR-VE	Split-VE	GK-VE-SF	SPQR-VE	Split-VE
Amazon0302	**1.04**	3.18	2.63	**294** MB	756 MB	589 MB
com-amazon	**1.76**	4.94	4.05	**586** MB	1.1 GB	946 MB
com-dblp	**1.53**	4.92	3.93	**494** MB	1.2 GB	887 MB
web-NotreDame	**0.82**	4.08	3.32	**438** MB	1.1 GB	855 MB
web-Stanford	**1.81**	7.76	6.45	**581** MB	1.6 GB	1.2 GB
Amazon0601	**2.66**	9.21	7.71	**693** MB	2 GB	1.5 GB
ia-yahoo-messages	**6.68**	19.62	17.90	**2.3** GB	3.8 GB	3.7 GB
web-Google	**5.74**	19.69	16.48	**1.3** GB	3.9 GB	3 GB
ca-cit-HepTh	**1.09**	7.81	6.53	**2.5** GB	4.7 GB	4.4 GB
cit-hepTh	**1.11**	8.00	6.50	**2.5** GB	4.7 GB	4.4 GB
visualise-us	**9.94**	34.12	28.88	**2.9** GB	5.8 GB	5.4 GB
web-BerkStan	**4.37**	27.86	20.87	**1.7** GB	5.2 GB	4 GB
ca-IMDB	**7.74**	22.31	18.32	**3.2** GB	6.8 GB	6.1 GB
ca-cit-HepPh	**1.45**	10.54	8.69	**4.4** GB	7.3 GB	7.1 GB
cit-hepPh	**1.42**	10.50	8.68	**4.4** GB	7.3 GB	7.1 GB

in every cyclic and complete graph, n_C and n_K respectively. Then it processes these graphs in random order, and it connects every one of them to a graph that was already processed, by inserting two edges that stem from two different vertices and end in two different vertices (thus ensuring biconnectivity). By carefully selecting the number and the sizes of the cyclic and complete graphs, we can determine the density of the resulting graph (since the number of edges of such a graph is given by $Cn_c + Kn_K(n_K - 1)/2 + 2(C + K - 1)$). In particular, we note that the graphs produced by our generator may contain a lot of cut-pairs (depending on C, n_C and K), even if they are very dense.

Table 4 Running times in seconds for artificial graphs in experimental setting (I). Parameters C and K correspond to the number of cyclic and complete graphs, respectively, where each such graph has $n_C = n_K = 200$ vertices. The number of vertices n and edges m refer to the produced instances. The best results in each row are marked in bold.

Graph details				2-edge cuts			vertex-edge cuts	
C	K	n	m	GK-2E	GK-2E-S	Tsin-2E	GK-VE	GK-VE-S
5000	0	1000000	1009998	0.68	**0.44**	0.45	1.38	**1.12**
4500	500	1000000	10859998	1.30	**0.74**	0.82	2.66	**1.76**
4000	1000	1000000	20709998	1.92	**1.02**	1.16	3.99	**2.39**
3500	1500	1000000	30559998	2.52	**1.30**	1.51	5.23	**2.97**
3000	2000	1000000	40409998	3.11	**1.60**	1.85	6.50	**3.57**
2500	2500	1000000	50259998	3.71	**1.86**	2.20	7.76	**4.18**
2000	3000	1000000	60109998	4.32	**2.13**	2.54	8.99	**4.76**
1500	3500	1000000	69959998	4.88	**2.41**	2.88	10.27	**5.36**
1000	4000	1000000	79809998	5.48	**2.68**	3.21	11.45	**6.05**
500	4500	1000000	89659998	6.05	**2.94**	3.53	12.74	**6.51**
0	5000	1000000	99509998	6.63	**3.19**	3.88	13.99	**7.09**

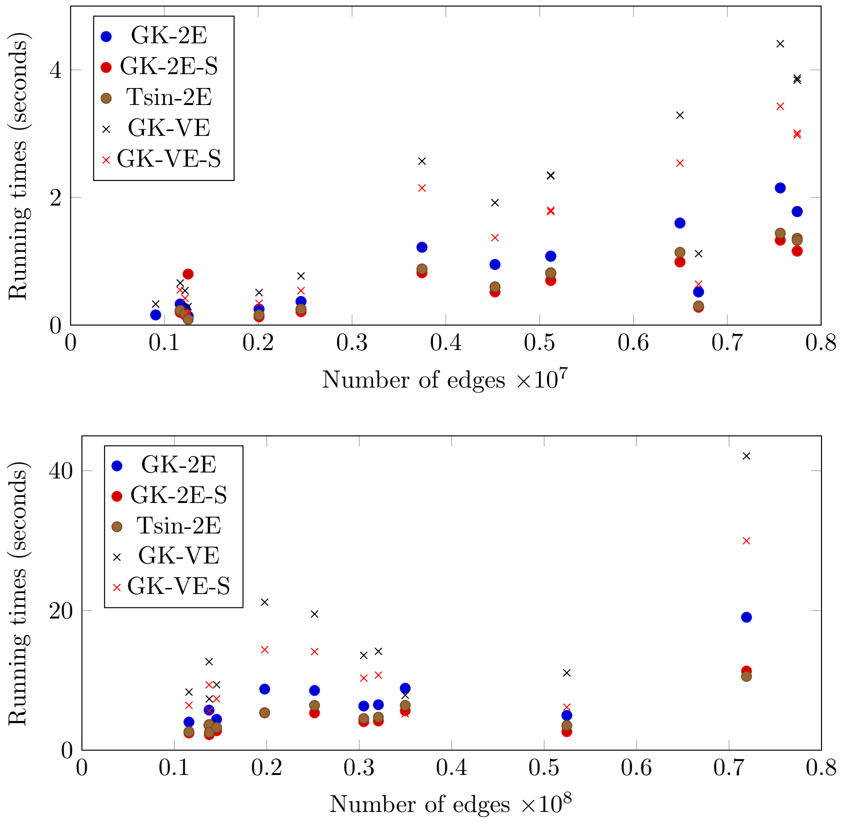

Figure 1 Running times for the graphs of Table 1 in experimental setting (I). The top plot shows the running times for graphs with less than 10M edges, while the bottom plot shows the running times for graphs with more than 10M edges.

Table 5 Running times in seconds for artificial graphs in experimental setting (I). Parameters C and K correspond to the number of cyclic and complete graphs, respectively, where each such graph has $n_C = n_K = 1000$ vertices. The number of vertices n and edges m refer to the produced instances. The best results in each row are marked in bold.

Graph details				2-edge cuts			vertex-edge cuts	
C	K	n	m	GK-2E	GK-2E-S	Tsin-2E	GK-VE	GK-VE-S
1000	0	1000000	1001998	0.68	**0.44**	0.47	1.31	**1.10**
900	100	1000000	50851998	4.12	**2.04**	2.32	8.41	**4.80**
800	200	1000000	100701998	7.52	**3.63**	4.33	15.39	**8.38**
700	300	1000000	150551998	10.92	**5.18**	6.25	22.84	**12.04**
600	400	1000000	200401998	14.35	**6.74**	8.09	30.84	**15.53**
500	500	1000000	250251998	17.61	**8.29**	10.04	38.71	**19.04**
400	600	1000000	300101998	21.10	**9.84**	11.88	44.27	**22.70**
300	700	1000000	349951998	24.49	**11.40**	13.96	54.70	**26.02**
200	800	1000000	399801998	27.77	**12.94**	15.67	63.00	**29.89**
100	900	1000000	449651998	31.14	**14.53**	17.67	72.01	**33.72**
0	1000	1000000	499501998	34.47	**16.15**	19.37	80.53	**37.00**

Table 6 Running times in seconds for artificial graphs in experimental setting (I). Parameters C and K correspond to the number of cyclic and complete graphs, respectively, where each such graph has $n_C = n_K = 2000$ vertices. The number of vertices n and edges m refer to the produced instances. The best results in each row are marked in bold.

Graph details				2-edge cuts			vertex-edge cuts	
C	K	n	m	GK-2E	GK-2E-S	Tsin-2E	GK-VE	GK-VE-S
500	0	1000000	1000998	0.68	**0.45**	0.48	2.03	**1.77**
450	50	1000000	100850998	8.34	**4.03**	4.89	19.81	**10.49**
400	100	1000000	200700998	16.44	**8.08**	9.08	37.39	**18.07**
350	150	1000000	300550998	24.04	**11.62**	14.01	51.45	**25.74**
300	200	1000000	400400998	31.60	**15.25**	18.18	70.57	**33.94**
250	250	1000000	500250998	40.41	**18.65**	21.90	78.38	**42.14**
200	300	1000000	600100998	46.78	**22.43**	26.39	93.28	**50.10**
150	350	1000000	699950998	59.55	**25.83**	30.79	120.87	**58.52**
100	400	1000000	799800998	61.69	**28.93**	33.78	124.60	**67.29**
50	450	1000000	899650998	69.94	**32.49**	38.68	139.34	**75.50**
0	500	1000000	999500998	79.28	**38.21**	42.14	155.44	**83.13**

The corresponding experimental results are given in Tables 4, 5 and 6, and plotted in Figure 2. We choose the values of parameters C, K, n_C and n_K so that all produced instances have $n = 1000000$ vertices, and vary their density and structure. Also, for simplicity, we choose $n_C = n_K$ in all instances. Then, it is easy to observe that as we decrease the value of C (and correspondingly increase K so that we maintain the total number of vertices fixed), the number of vertex-edge cut-pairs as well as the number of 2-edge-cuts decrease, while the graph gets more dense. From the experimental results, however, we observe that the number of cuts does not affect the performance of the algorithms. Indeed, similarly to our previous experiments, GK-2E-S is consistently faster than GK-2E by more than 50% on average. Also, with respect to Tsin-2E, GK-2E-S is faster on all instances by 15% on average.

Regarding the performance of the algorithms for computing vertex-edge cut-pairs, we note that in this experiment, our streamlined version GK-VE-S achieves higher speed-ups with respect to GK-VE. Specifically, GK-VE-S is faster than GK-VE by 16% up to 54% (more than 45% on average).

6 Concluding remarks

We presented streamlined versions of two linear-time algorithms of [6] that compute the vertex-edge cut-pairs of a biconnected graph G and the 2-edge cuts of a 2-edge-connected graph, respectively. Although we can compute these cuts in linear time using previously known techniques, the new algorithms have a major advantage: Both algorithms are based on a common framework, which results in conceptually simple and easy to implement algorithms, especially for computing vertex-edge cut-pairs, where the alternative linear-time algorithms exploit the structure of the triconnected components of G. Furthermore, our experimental results showed that our new algorithms perform significantly better in practice both in terms of running time and of space requirements.

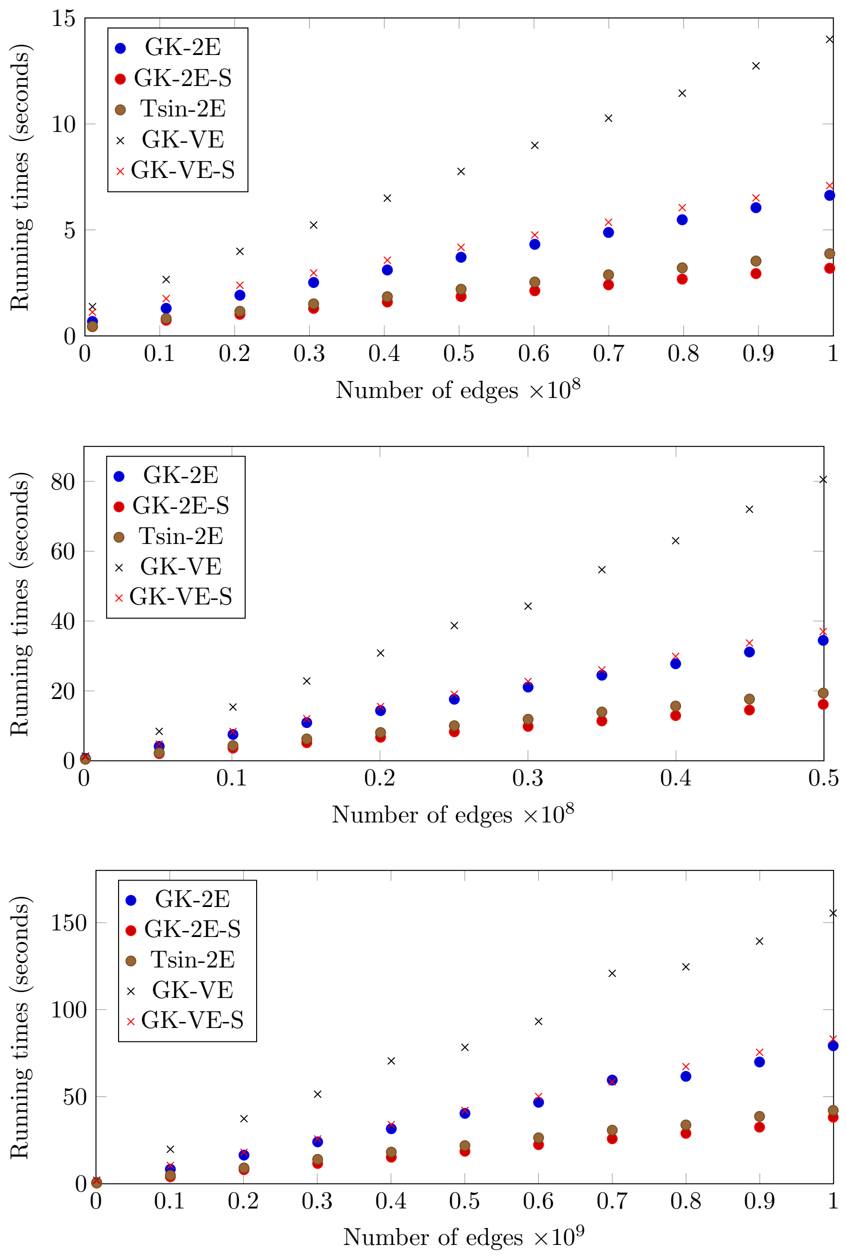

Figure 2 Running times for artificial graphs in experimental setting (I). The plots, from top to bottom, show the running times for the graphs of Tables 4, 5, and 6, respectively.

References

1 G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, April 1996.

2 G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996.

3 M. Chimani, C. Gutwenger, M. Junger, G. W. Klau, K. Klein, and P. Mutzel. The open graph drawing framework. In *Handbook of Graph Drawing and Visualization*, pages 543–570. CRC Press, 2013.

4 D. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacement. *SIAM J. Comput.*, 22(3):587–616, June 1993. doi:10.1137/0222040.

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

6 L. Georgiadis 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.

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

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

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

10 Z. Jiang. An empirical study of 3-vertex connectivity algorithms. Master's thesis, University of Windsor, 2013. Electronic Theses and Dissertations, paper 4980.

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

12 K. Mehlhorn, A. Neumann, and J. M. Schmidt. Certifying 3-edge-connectivity. *Algorithmica*, 77(2):309–335, February 2017. doi:10.1007/s00453-015-0075-x.

13 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.

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

15 R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: http://networkrepository.com.

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

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

18 Wikipedia contributors. SPQR tree — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=SPQR_tree&oldid=951256273, 2020.

A Omitted algorithms

Algorithm 1 compute all $b_p\_count(v)$ and $up(v)$ while performing a DFS.

```
initialize all $dfs$ labels to $\emptyset$
$dfs \leftarrow 1$
initialize an array $p(v)$ with size $n$
initialize an array $tempChild(v)$ with size $n$
initialize all $b_p\_count(v)$ to 0
initialize all $up(u)$ to 0            /* this is also used in the modified algorithms
  "$high(u) = v$" and "$high(u) < v$", in order to test if $high(u) = high_p(u)$ */
DFS($r$)
Function $DFS$(vertex $v$)
begin
    $dfs(v) \leftarrow dfs$
    $dfs \leftarrow dfs + 1$
    foreach $vertex\ u\ adjacent\ to\ v$ do
        if $dfs(u) = \emptyset$ then
            $p(u) \leftarrow v$
            $tempChild(v) \leftarrow u$
            DFS($u$)
        end
        if $dfs(u) < dfs(v)$ and $u \neq p(v)$ then
            $b_p\_count(v) \leftarrow b_p\_count(v) + 1$
            $up(tempChild(u)) \leftarrow up(tempChild(u)) + 1$
        end
        else if $v = p(u)$ then
            $b_p\_count(v) \leftarrow b_p\_count(v) + b_p\_count(u)$
        end
    end
    $b_p\_count(v) \leftarrow b_p\_count(v) - up(v)$
end
```

Algorithm 2 Compute the cut-edges and the number of 2-cuts.

```
perform a DFS, and compute all $low(v)$, $b\_count(v)$, $M(v)$ and $nextM(v)$, for all vertices $v \neq r$
$n2cuts \leftarrow 0$
// case back-edge - tree-edge
foreach $v \neq r$ do
    if $b\_count(v) = 1$ then
        mark the edges $(v, p(v))$ and $(M(v), low(v))$
        $n2cuts \leftarrow n2cuts + 1$
    end
end
// case tree-edge - tree-edge
foreach $v \neq r$ that has $M(v) = v$ do
    while $v \neq \emptyset$ do
        $u \leftarrow nextM(v)$
        $nCutEdges \leftarrow 0$
        while $u \neq \emptyset$ and $b\_count(u) = b\_count(v)$ do
            mark the edges $(v, p(v))$ and $(u, p(u))$
            $nCutEdges \leftarrow nCutEdges + 1$
            $u \leftarrow nextM(u)$
        end
        $n2cuts \leftarrow n2cuts + nCutEdges(nCutEdges + 1)/2$
        $v \leftarrow u$
    end
end
```

Algorithm 3 Compute all $M(v)$ and $nextM(v)$.

```
foreach $v \neq r$ do
    $L(v) \leftarrow$ first child of $v$
    $R(v) \leftarrow$ last child of $v$
    $nextM(v) \leftarrow \emptyset$
end
foreach $v \neq r$, in a bottom-up fashion do
    $c \leftarrow v$
    $m \leftarrow v$
    while true do
        if $l(m) < v$ then $M(v) \leftarrow m$ break
        while $low(L(m)) \geq v$ do $L(m) \leftarrow$ next child of $m$
        while $low(R(m)) \geq v$ do $R(m) \leftarrow$ previous child of $m$
        if $L(m) \neq R(m)$ then $M(v) \leftarrow m$ break
        $c \leftarrow L(m)$
        $m \leftarrow M(c)$
    end
    if $c \neq v$ then
        $nextM(c) = v$
    end
end
```

B Omitted experimental results

Table 7 Memory consumption for the graphs of Table 1 in experimental setting (I). The best results in each row are marked in bold.

	2-edge cuts			vertex-edge cuts	
Graph	GK-2E	GK-2E-S	Tsin-2E	GK-VE	GK-VE-S
Amazon0302	46 MB	**42** MB	58 MB	61 MB	**51** MB
com-amazon	81 MB	**72** MB	90 MB	110 MB	**90** MB
com-dblp	69 MB	**62** MB	83 MB	94 MB	**77** MB
web-NotreDame	61 MB	**55** MB	77 MB	65 MB	**53** MB
web-Stanford	74 MB	**69** MB	106 MB	111 MB	**79** MB
Amazon0601	95 MB	**89** MB	133 MB	140 MB	**103** MB
ia-yahoo-messages	394 MB	**346** MB	390 MB	518 MB	**449** MB
web-Google	192 MB	**178** MB	258 MB	282 MB	**209** MB
ca-cit-HepTh	378 MB	**338** MB	412 MB	513 MB	**424** MB
cit-HepTh	373 MB	**338** MB	412 MB	513 MB	**424** MB
visualise-us	460 MB	**410** MB	513 MB	634 MB	**522** MB
web-BerkStan	**209** MB	**209** MB	332 MB	314 MB	**233** MB
ca-IMDB	542 MB	**485** MB	597 MB	738 MB	**608** MB
ca-cit-HepPh	626 MB	**556** MB	665 MB	841 MB	**705** MB
cit-HepPh	618 MB	**556** MB	665 MB	841 MB	**705** MB
amazon-ratings	835 MB	**746** MB	917 MB	1.1 GB	**936** MB
hugetrace-00000	986 MB	**868** MB	1.1 GB	1.3 GB	**1.1** GB
rgg-n-2-20-s0	988 MB	**883** MB	1.1 GB	1.3 GB	**1.1** GB
wiki-user-edits-page	1.2 GB	**1** GB	1.2 GB	1.6 GB	**1.3** GB
hugetric-00010	1.4 GB	**1.2** GB	1.5 GB	1.9 GB	**1.6** GB
delaunay-n22	1.8 GB	**1.6** GB	1.9 GB	2.4 GB	**2** GB
co-papers-dblp	2.1 GB	**1.9** GB	2.3 GB	2.9 GB	**2.4** GB
co-papers-citeseer	2.2 GB	**2** GB	2.5 GB	3.1 GB	**2.5** GB
packing-b050	2.4 GB	**2.2** GB	2.7 GB	3.3 GB	**2.7** GB
human-Jung2015	**1.3** GB	**1.3** GB	2.3 GB	2.2 GB	**1.4** GB
rgg-n-2-23-s0	**2.3** GB	**2.4** GB	3.6 GB	3.7 GB	**2.5** GB