# Computing Vertex-Edge Cut-Pairs and 2-Edge Cuts in Practice <br> Loukas Georgiadis, Konstantinos Giannis, Giuseppe F Italiano, Evangelos Kosinas 

## - To cite this version:

Loukas Georgiadis, Konstantinos Giannis, Giuseppe F Italiano, Evangelos Kosinas. Computing Vertex-Edge Cut-Pairs and 2-Edge Cuts in Practice. SEA 2021-19th International Symposium on Experimental Algorithms, Jun 2021, Nice, France. pp.1-19, 10.4230/LIPIcs.SEA.2021.20 . hal03395500

HAL Id: hal-03395500
https://inria.hal.science/hal-03395500
Submitted on 22 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Loukas Georgiadis $\square$ (D)<br>Department of Computer Science \& Engineering, University of Ioannina, Greece

Konstantinos Giannis<br>Gran Sasso Science Institute, L'Aquila, Italy<br>Giuseppe F. Italiano $\boxtimes$ (<br>LUISS University, Rome, Italy<br>Evangelos Kosinas<br>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 $G \backslash v$ 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 $G \backslash e$ 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 $\rightarrow$ 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 $G \backslash e$ is not connected. Similarly, a vertex $v \in V$ is an articulation point of $G$ if $G \backslash v$ 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 $G \backslash\{e, f\}$ 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 $G \backslash\{e, f\}$ 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 $G \backslash\{u, v\}$ 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 $G \backslash v$ 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 $G \backslash e$ 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 $h i g h(v)$ and $\operatorname{low}(v)$, 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 $T$ :

- $B(v):=\{(x, y) \in B \mid x \in T(v)$ and $y<v\}$, the set of all back-edges that start from $T(v)$ and end in a proper ancestor of $v$.
- $B_{p}(v):=\{(x, y) \in B \mid x \in T(v)$ 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 $(v, y)$ ).
- $\operatorname{low}(v):=\min \{y \mid(x, y) \in B(v)\}$, the lowest lower end of all back-edges in $B(v)$.
- $\operatorname{high}(v):=\max \{y \mid(x, y) \in B(v)\}$, the highest lower end of all back-edges in $B(v)$.
- $\operatorname{high}_{p}(v):=\max \left\{y \mid(x, y) \in B_{p}(v)\right\}$, the highest lower end of all back-edges in $B_{p}(v)$.
- $M(v):=n c a\{x \mid(x, y) \in B(v)\}$, the nearest common ancestor of the higher ends of all back-edges in $B(v)$.
- $M_{p}(v):=n c a\left\{x \mid(x, y) \in B_{p}(v)\right\}$, 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 $B(v)$.
- $b_{p \_} \operatorname{count}(v):=\# B_{p}(v)$, 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 $v$.
$B(v), l(v), \operatorname{low}(v), \operatorname{high}(v), M(v), b \_\operatorname{count}(v)$, and $u p(v)$ are defined for all vertices $v \neq r$; similarly, $B_{p}(v), \operatorname{high}_{p}(v), M_{p}(v)$, and $b_{p \_c o u n t}(v)$ are defined for all vertices $v \notin\left\{r, r_{c}\right\}$, 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 $G \backslash\{v, e\}$ is not connected. The corresponding values are stored in variables 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 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 $G \backslash\{v, e\}$ 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 $b_{p \_} \operatorname{count}(c)=1$. All $b_{p \_c o u n t}(c)$, 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 $b_{p \_c o u n t}(c)=1$, then $\left\{p(c),\left(M_{p}(c), \operatorname{low}(c)\right)\right\}$ 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 $u \leq l o w(c)$ or $u>\operatorname{high}_{p}(c)$; in other words, $u$ does not belong to any set of the form $T\left[\operatorname{high}_{p}(c), \operatorname{low}(c)\right)$, 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 $T\left[\operatorname{high}_{p}(c)\right.$, low $\left.(c)\right)$, 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 $\operatorname{high}_{p}(c)<u$. (The converse is also true: if $u$ is a proper ancestor of $p(c)$ such that $M_{p}(c)=M(u)$ and $\operatorname{high}_{p}(c)<u$, 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 $\operatorname{high}_{p}(c)<u$. 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 $h i g h_{p}(c)<u$, then, if $c^{\prime}$ is also in $M_{p}^{-1}(m)$ and $c^{\prime} \in T\left[c, h i g h_{p}(c)\right.$ ), we have $U\left(c^{\prime}\right)=U(c) \cap T\left(c^{\prime}, \operatorname{high}_{p}(c)\right)$. (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 $\operatorname{high}(u)$ is an ancestor of $v$, and we further distinguish two cases, depending on whether $\operatorname{high}(u)=v$ or $\operatorname{high}(u)$ 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 $h i g h^{-1}(v)$ that are not children of $v$ and are such that either $\operatorname{low}(u)=v$ or $\operatorname{low}(u)<v$ and $M_{p}(c)$ is in $T(u)$, where $c$ is the child of $v$ which is an ancestor of $u$ (see Algorithm "high $(u)=v$ " in [6]). If $\operatorname{high}(u)$ 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 $\operatorname{high}(u)<p(c)$, 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 $\operatorname{high}(u)<p(c)$. 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 $\operatorname{high}(u)<p(c)$, then all $u$ in $U(c)$ have the same high point, call it $h$, and, if $c^{\prime}$ is also in $M_{p}^{-1}(m)$ and such that $c \geq c^{\prime}$ and $h<p\left(c^{\prime}\right)$, then $U\left(c^{\prime}\right)=U(c) \cup\left(T\left[c, c^{\prime}\right) \cap M^{-1}(u)\right)$.

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 $b_{p \_} \operatorname{count}(v)$, [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 $u p(v)$, as shown in Algorithm 1 (see Appendix A). This works as follows. By definition, $b_{p} \ldots$ 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
 $B(v)\}-\#\{(x, p(v)) \mid x \in T(v)\}$, 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 $b_{p} \ldots \operatorname{count}(v) \leftarrow$ $b_{p \_c o u n t}(v)+b_{p \_c o u n t}(u)$ if $u$ is a child of $v$, or $b_{p \_c o u n t}(v) \leftarrow b_{p \_c o u n t}(v)+1$ if $u$ is an ancestor of $v$ but not its parent. Now, to compute all $u p(v):=\#\{(x, p(v)) \mid x \in T(v)\}$, we keep a variable 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 $u p(\operatorname{tempChild}(v)) \leftarrow u p(\operatorname{temp} \operatorname{Child}(v))+1$.

A second important improvement comes from the fact that [6] uses both the $\operatorname{high}(v)$ and the $\operatorname{high}_{p}(v)$ 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 $\operatorname{high}(u) \leq v=p(c)$; thus we have $\operatorname{high}(u)=\operatorname{high}_{p}(u)$, since $\operatorname{high}(u)<c \leq p(u)$. 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 $\operatorname{high}(u)=\operatorname{high}_{p}(u)$. 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 $u p(u)=0$.

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 $G_{\alpha}$. Each vertex of $G_{\alpha}$ corresponds to a vertex of the original graph $G$. An edge of $G_{\alpha}$ 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 $\alpha$, and the graph $G_{\alpha}$ associated with it, has one of the following types:

- If $\alpha$ is an $S$-node, then $G_{\alpha}$ is a cycle graph with three or more vertices and edges.
- If $\alpha$ is a $P$-node, then $G_{\alpha}$ is a multigraph with two vertices and at least 3 parallel edges.
- If $\alpha$ is a $Q$-node, then $G_{\alpha}$ is a single real edge.
- If $\alpha$ is an $R$-node, then $G_{\alpha}$ 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 $G_{\alpha}$ and the other is an edge in $G_{\beta}$. 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 $G_{\alpha}$ associated with an $R$-node $\alpha$ of $\mathcal{T}$.
(b) $u$ and $v$ are vertices in the graph $G_{\alpha}$ associated with a $P$-node $\alpha$ of $\mathcal{T}$.
(c) $u$ and $v$ are vertices in the graph $G_{\alpha}$ associated with an $S$-node $\alpha$ 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 $G \backslash\{u, v\}$ consists of two components that are represented by two paths of the cycle graph $G_{\alpha}$ associated with the $S$-node $\alpha$ 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 $\alpha$ such that $v, x$ and $y$ are vertices of $G_{\alpha}$ 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 $\operatorname{count}(v) \leftarrow 0$ for all $v \in V$, and process the $S$-nodes of $\mathcal{T}$ individually. For each $S$-node $\alpha$ we count the number $m_{\alpha}$ of the real edges of $G_{\alpha}$. Then, for each vertex $v \in V\left(G_{\alpha}\right)$, we set $\operatorname{count}(v)$ equal to $m_{\alpha}-\mid\{e \in E: e$ is adjacent to $v\} \mid$.

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 $\operatorname{count}(v)$ 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 $b \_\operatorname{count}(v)=1$, 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 $\operatorname{high}(u)=\operatorname{high}(v)$. Now, let $m$ be a vertex and $u_{1}, \ldots, u_{k}$ all the vertices in $M^{-1}(m)$ ordered decreasingly. Then we have $\operatorname{high}\left(u_{1}\right) \geq \ldots \geq \operatorname{high}\left(u_{k}\right)$. 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 $\operatorname{high}(u)=\operatorname{high}(v)$.

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 $(u)=\operatorname{high}(v)$ if and only if $b \_c o u n t(u)=b \_c o u n t(v)$.

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 $y \leq \operatorname{high}(u)=\operatorname{high}(v)<v$, we have that $y$ is a proper ancestor of $v$. This shows that $B(u) \subseteq B(v)$, and so we have $b \_c o u n t(u) \leq b \_c o u n t(v)$. Now, with a reversal of the roles of $u$ and $v$, we also get $b \_\operatorname{count}(v) \leq b \_$count $(u)$. We conclude that $b \_c o u n t(u)=b \_c o u n t(v)$.
$(\Leftarrow)$ 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 $b \_c o u n t(u)=b \_c o u n t(v)$, it must be the case that $B(v)=B(u)$. Thus we have $\operatorname{high}(u)=\operatorname{high}(v)$.

Thus we can test condition (2) by checking whether $b \_$count $(u)=b \_$count $(v)$, instead of $\operatorname{high}(u)=\operatorname{high}(v)$. 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 $b \_\operatorname{count}(v)=1$ ). Now, let $(v, p(v))$ be a tree-edge and $\left(u_{1}, p\left(u_{1}\right)\right), \ldots,\left(u_{k}, p\left(u_{k}\right)\right)$ all the tree-edges which form a cut-pair with $(v, p(v))$. Then (1) implies that every $\left(u_{i}, p\left(u_{i}\right)\right)$, for $i \in\{1, \ldots, k\}$, forms a cut-pair with $\left(u_{j}, p\left(u_{j}\right)\right)$, for every $j \in\{1, \ldots, k\} \backslash\{i\}$. Furthermore, these are all the tree-edges (plus $(v, p(v)))$ with which $\left(u_{i}, p\left(u_{i}\right)\right)$ 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 $\left(u_{1}, p\left(u_{1}\right)\right)$ 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 $\left(u_{k}, p\left(u_{k}\right)\right)$ forms a cut-pair with $(v, p(v))$ (i.e. that satisfies $b \_$count $\left.\left(u_{k}\right)=b \_c o u n t(v)\right)$. 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 $n \operatorname{ext} M(v)$ 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 $E_{c u t}$ that contains all the edges belonging to a cut-pair. In the second part, it processes the edge set $E_{\text {cut }}$ 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 $E_{c u t}$ 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 $\operatorname{stack}(w)$ to determine if we have found a new cut-pair and update $\operatorname{stack}(w)$. When we finish this check for all the children of $v$, we update $\operatorname{stack}(v)$ and backtrack. Finally, when the traversal returns to the root of $T$, all edges belonging in a cut-pair form the set $E_{\text {cut }}$.

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 2.5 GHz processor with 15 MB L3 cache and 96 GB 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 8300 H 4 GHz processor with $8 \mathrm{MB} \mathrm{L3}$ cache and 8 GB DDR4 RAM at $2,666 \mathrm{MHz}$. 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 $\mathrm{C}++11$ compliant compiler and GNU Make. We observed that the OGDF library was not able to compute the SPQR trees of graphs with more than $\sim 45000$ 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 $G^{\prime}$ be the resulting graph. Then, we compute the leaves $B_{1}, \ldots, B_{l}$ of the tree representation of the biconnected components of $G^{\prime}$, 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 | Graph Details |  |  |  |  |  |
| :--- | :--- | ---: | ---: | ---: | ---: | ---: |
| 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 | $\mathbf{0 . 1 0}$ | 0.13 | 0.33 | $\mathbf{0 . 2 5}$ |
| com-amazon | 0.33 | $\mathbf{0 . 2 0}$ | 0.23 | 0.66 | $\mathbf{0 . 5 5}$ |
| com-dblp | 0.26 | $\mathbf{0 . 1 6}$ | 0.18 | 0.54 | $\mathbf{0 . 4 2}$ |
| web-NotreDame | 0.14 | $\mathbf{0 . 0 8}$ | 0.09 | 0.29 | $\mathbf{0 . 1 9}$ |
| web-Stanford | 0.25 | $\mathbf{0 . 1 3}$ | 0.16 | 0.51 | $\mathbf{0 . 3 4}$ |
| Amazon0601 | 0.37 | $\mathbf{0 . 2 1}$ | 0.25 | 0.77 | $\mathbf{0 . 5 4}$ |
| ia-yahoo-messages | 1.22 | $\mathbf{0 . 8 2}$ | 0.88 | 2.57 | $\mathbf{2 . 1 5}$ |
| web-Google | 0.95 | $\mathbf{0 . 5 2}$ | 0.60 | 1.92 | $\mathbf{1 . 3 7}$ |
| ca-cit-HepTh | 1.08 | $\mathbf{0 . 7 0}$ | 0.81 | 2.35 | $\mathbf{1 . 8 0}$ |
| cit-HepTh | 1.08 | $\mathbf{0 . 7 0}$ | 0.82 | 2.34 | $\mathbf{1 . 7 8}$ |
| visualise-us | 1.60 | $\mathbf{0 . 9 9}$ | 1.14 | 3.29 | $\mathbf{2 . 5 4}$ |
| web-BerkStan | 0.52 | $\mathbf{0 . 2 8}$ | 0.30 | 1.12 | $\mathbf{0 . 6 4}$ |
| ca-IMDB | 2.15 | $\mathbf{1 . 3 3}$ | 1.44 | 4.41 | $\mathbf{3 . 4 3}$ |
| ca-cit-HepPh | 1.78 | $\mathbf{1 . 1 6}$ | 1.36 | 3.87 | $\mathbf{3 . 0 1}$ |
| cit-HepPh | 1.78 | $\mathbf{1 . 1 6}$ | 1.33 | 3.84 | $\mathbf{2 . 9 8}$ |
| amazon-ratings | 4.03 | $\mathbf{2 . 4 6}$ | 2.65 | 8.32 | $\mathbf{6 . 4 3}$ |
| hugetrace-00000 | 5.74 | $\mathbf{3 . 5 2}$ | 3.67 | 12.66 | $\mathbf{9 . 3 5}$ |
| rgg-n-2-20-s0 | 3.57 | $\mathbf{2 . 2 2}$ | 2.55 | 7.34 | $\mathbf{5 . 6 4}$ |
| wiki-user-edits-page | 4.42 | $\mathbf{2 . 7 9}$ | 3.24 | 9.37 | $\mathbf{7 . 3 6}$ |
| hugetric-00010 | 8.75 | 5.37 | $\mathbf{5 . 3 3}$ | 21.19 | $\mathbf{1 4 . 3 9}$ |
| delaunay-n22 | 8.55 | $\mathbf{5 . 3 4}$ | 6.41 | 19.49 | $\mathbf{1 4 . 1 2}$ |
| co-papers-dblp | 6.32 | $\mathbf{4 . 0 8}$ | 4.53 | 13.58 | $\mathbf{1 0 . 3 2}$ |
| co-papers-citeseer | 6.51 | $\mathbf{4 . 1 8}$ | 4.73 | 14.15 | $\mathbf{1 0 . 7 5}$ |
| packing-b050 | 8.87 | $\mathbf{5 . 6 6}$ | 6.42 | 7.86 | $\mathbf{5 . 2 1}$ |
| human-Jung2015 | 4.98 | $\mathbf{2 . 6 5}$ | 3.53 | 11.08 | $\mathbf{6 . 1 6}$ |
| rgg-n-2-23-s0 | 19.03 | 11.34 | $\mathbf{1 0 . 5 6}$ | 42.12 | $\mathbf{2 9 . 9 8}$ |

## 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 | $\mathbf{1 . 0 4}$ | 3.18 | 2.63 | $\mathbf{2 9 4} \mathrm{MB}$ | 756 MB | 589 MB |
| com-amazon | $\mathbf{1 . 7 6}$ | 4.94 | 4.05 | $\mathbf{5 8 6} \mathrm{MB}$ | 1.1 GB | 946 MB |
| com-dblp | $\mathbf{1 . 5 3}$ | 4.92 | 3.93 | $\mathbf{4 9 4} \mathrm{MB}$ | 1.2 GB | 887 MB |
| web-NotreDame | $\mathbf{0 . 8 2}$ | 4.08 | 3.32 | $\mathbf{4 3 8} \mathrm{MB}$ | 1.1 GB | 855 MB |
| web-Stanford | $\mathbf{1 . 8 1}$ | 7.76 | 6.45 | $\mathbf{5 8 1} \mathrm{MB}$ | 1.6 GB | 1.2 GB |
| Amazon0601 | $\mathbf{2 . 6 6}$ | 9.21 | 7.71 | $\mathbf{6 9 3} \mathrm{MB}$ | 2 GB | 1.5 GB |
| ia-yahoo-messages | $\mathbf{6 . 6 8}$ | 19.62 | 17.90 | $\mathbf{2 . 3} \mathrm{~GB}$ | 3.8 GB | 3.7 GB |
| web-Google | $\mathbf{5 . 7 4}$ | 19.69 | 16.48 | $\mathbf{1 . 3} \mathrm{~GB}$ | 3.9 GB | 3 GB |
| ca-cit-HepTh | $\mathbf{1 . 0 9}$ | 7.81 | 6.53 | $\mathbf{2 . 5} \mathrm{~GB}$ | 4.7 GB | 4.4 GB |
| cit-hepTh | $\mathbf{1 . 1 1}$ | 8.00 | 6.50 | $\mathbf{2 . 5} \mathrm{~GB}$ | 4.7 GB | 4.4 GB |
| visualise-us | $\mathbf{9 . 9 4}$ | 34.12 | 28.88 | $\mathbf{2 . 9} \mathrm{~GB}$ | 5.8 GB | 5.4 GB |
| web-BerkStan | $\mathbf{4 . 3 7}$ | 27.86 | 20.87 | $\mathbf{1 . 7} \mathrm{~GB}$ | 5.2 GB | 4 GB |
| ca-IMDB | $\mathbf{7 . 7 4}$ | 22.31 | 18.32 | $\mathbf{3 . 2} \mathrm{~GB}$ | 6.8 GB | 6.1 GB |
| ca-cit-HepPh | $\mathbf{1 . 4 5}$ | 10.54 | 8.69 | $\mathbf{4 . 4} \mathrm{~GB}$ | 7.3 GB | 7.1 GB |
| cit-hepPh | $\mathbf{1 . 4 2}$ | 10.50 | 8.68 | $\mathbf{4 . 4} \mathrm{~GB}$ | 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 $\left.C n_{c}+K n_{K}\left(n_{K}-1\right) / 2+2(C+K-1)\right)$. 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 | $\mathbf{0 . 4 4}$ | 0.45 | 1.38 | $\mathbf{1 . 1 2}$ |
| 4500 | 500 | 1000000 | 10859998 | 1.30 | $\mathbf{0 . 7 4}$ | 0.82 | 2.66 | $\mathbf{1 . 7 6}$ |
| 4000 | 1000 | 1000000 | 20709998 | 1.92 | $\mathbf{1 . 0 2}$ | 1.16 | 3.99 | $\mathbf{2 . 3 9}$ |
| 3500 | 1500 | 1000000 | 30559998 | 2.52 | $\mathbf{1 . 3 0}$ | 1.51 | 5.23 | $\mathbf{2 . 9 7}$ |
| 3000 | 2000 | 1000000 | 40409998 | 3.11 | $\mathbf{1 . 6 0}$ | 1.85 | 6.50 | $\mathbf{3 . 5 7}$ |
| 2500 | 2500 | 1000000 | 50259998 | 3.71 | $\mathbf{1 . 8 6}$ | 2.20 | 7.76 | $\mathbf{4 . 1 8}$ |
| 2000 | 3000 | 1000000 | 60109998 | 4.32 | $\mathbf{2 . 1 3}$ | 2.54 | 8.99 | $\mathbf{4 . 7 6}$ |
| 1500 | 3500 | 1000000 | 69959998 | 4.88 | $\mathbf{2 . 4 1}$ | 2.88 | 10.27 | $\mathbf{5 . 3 6}$ |
| 1000 | 4000 | 1000000 | 79809998 | 5.48 | $\mathbf{2 . 6 8}$ | 3.21 | 11.45 | $\mathbf{6 . 0 5}$ |
| 500 | 4500 | 1000000 | 89659998 | 6.05 | $\mathbf{2 . 9 4}$ | 3.53 | 12.74 | $\mathbf{6 . 5 1}$ |
| 0 | 5000 | 1000000 | 99509998 | 6.63 | $\mathbf{3 . 1 9}$ | 3.88 | 13.99 | $\mathbf{7 . 0 9}$ |



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 10 M 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 | $\mathbf{0 . 4 4}$ | 0.47 | 1.31 | $\mathbf{1 . 1 0}$ |  |
| 900 | 100 | 1000000 | 50851998 | 4.12 | $\mathbf{2 . 0 4}$ | 2.32 | 8.41 | $\mathbf{4 . 8 0}$ |  |
| 800 | 200 | 1000000 | 100701998 | 7.52 | $\mathbf{3 . 6 3}$ | 4.33 | 15.39 | $\mathbf{8 . 3 8}$ |  |
| 700 | 300 | 1000000 | 150551998 | 10.92 | $\mathbf{5 . 1 8}$ | 6.25 | 22.84 | $\mathbf{1 2 . 0 4}$ |  |
| 600 | 400 | 1000000 | 200401998 | 14.35 | $\mathbf{6 . 7 4}$ | 8.09 | 30.84 | $\mathbf{1 5 . 5 3}$ |  |
| 500 | 500 | 1000000 | 250251998 | 17.61 | $\mathbf{8 . 2 9}$ | 10.04 | 38.71 | $\mathbf{1 9 . 0 4}$ |  |
| 400 | 600 | 1000000 | 300101998 | 21.10 | $\mathbf{9 . 8 4}$ | 11.88 | 44.27 | $\mathbf{2 2 . 7 0}$ |  |
| 300 | 700 | 1000000 | 349951998 | 24.49 | $\mathbf{1 1 . 4 0}$ | 13.96 | 54.70 | $\mathbf{2 6 . 0 2}$ |  |
| 200 | 800 | 1000000 | 399801998 | 27.77 | $\mathbf{1 2 . 9 4}$ | 15.67 | 63.00 | $\mathbf{2 9 . 8 9}$ |  |
| 100 | 900 | 1000000 | 449651998 | 31.14 | $\mathbf{1 4 . 5 3}$ | 17.67 | 72.01 | $\mathbf{3 3 . 7 2}$ |  |
| 0 | 1000 | 1000000 | 499501998 | 34.47 | $\mathbf{1 6 . 1 5}$ | 19.37 | 80.53 | $\mathbf{3 7 . 0 0}$ |  |

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 | $\mathbf{0 . 4 5}$ | 0.48 | 2.03 | $\mathbf{1 . 7 7}$ |
| 450 | 50 | 1000000 | 100850998 | 8.34 | $\mathbf{4 . 0 3}$ | 4.89 | 19.81 | $\mathbf{1 0 . 4 9}$ |
| 400 | 100 | 1000000 | 200700998 | 16.44 | $\mathbf{8 . 0 8}$ | 9.08 | 37.39 | $\mathbf{1 8 . 0 7}$ |
| 350 | 150 | 1000000 | 300550998 | 24.04 | $\mathbf{1 1 . 6 2}$ | 14.01 | 51.45 | $\mathbf{2 5 . 7 4}$ |
| 300 | 200 | 1000000 | 400400998 | 31.60 | $\mathbf{1 5 . 2 5}$ | 18.18 | 70.57 | $\mathbf{3 3 . 9 4}$ |
| 250 | 250 | 1000000 | 500250998 | 40.41 | $\mathbf{1 8 . 6 5}$ | 21.90 | 78.38 | $\mathbf{4 2 . 1 4}$ |
| 200 | 300 | 1000000 | 600100998 | 46.78 | $\mathbf{2 2 . 4 3}$ | 26.39 | 93.28 | $\mathbf{5 0 . 1 0}$ |
| 150 | 350 | 1000000 | 699950998 | 59.55 | $\mathbf{2 5 . 8 3}$ | 30.79 | 120.87 | $\mathbf{5 8 . 5 2}$ |
| 100 | 400 | 1000000 | 799800998 | 61.69 | $\mathbf{2 8 . 9 3}$ | 33.78 | 124.60 | $\mathbf{6 7 . 2 9}$ |
| 50 | 450 | 1000000 | 899650998 | 69.94 | $\mathbf{3 2 . 4 9}$ | 38.68 | 139.34 | $\mathbf{7 5 . 5 0}$ |
| 0 | 500 | 1000000 | 999500998 | 79.28 | $\mathbf{3 8 . 2 1}$ | 42.14 | 155.44 | $\mathbf{8 3 . 1 3}$ |

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.
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 \(b_{p} \_\operatorname{count}(v)\) and \(u p(v)\) while performing a DFS.
initialize all \(d f s\) labels to \(\emptyset\)
\(d f s \leftarrow 1\)
initialize an array \(p(v)\) with size \(n\)
initialize an array tempChild \((v)\) with size \(n\)
initialize all \(b_{p} \_c o u n t(v)\) to 0
initialize all \(u p(u)\) to \(0 \quad / *\) this is also used in the modified algorithms
    " \(\operatorname{high}(u)=v\) " and " \(\operatorname{high}(u)<v "\), in order to test if \(\operatorname{high}(u)=\operatorname{high}_{p}(u)\) */
DFS( \(r\) )
Function DFS(vertex \(v\) )
begin
    \(d f s(v) \leftarrow d f s\)
    \(d f s \leftarrow d f s+1\)
    foreach vertex \(u\) adjacent to \(v\) do
        if \(d f s(u)=\emptyset\) then
            \(p(u) \leftarrow v\)
            tempChild \((v) \leftarrow u\)
            DFS( \(u\) )
            end
            if \(d f s(u)<d f s(v)\) and \(u \neq p(v)\) then
            \(b_{p \_c o u n t}(v) \leftarrow b_{p \_c o u n t}(v)+1\)
            \(u p(\) tempChild \((u)) \leftarrow u p(\) tempChild \((u))+1\)
            end
            else if \(v=p(u)\) then
                \(b_{p \_c o u n t}(v) \leftarrow b_{p \_c o u n t}(v)+b_{p \_c o u n t}(u)\)
            end
    end
    \(b_{p \_c o u n t}(v) \leftarrow b_{p \_c o u n t}(v)-u p(v)\)
end
```

```
Algorithm 2 Compute the cut-edges and the number of 2-cuts.
1 perform a DFS, and compute all \(\operatorname{low}(v), b \_\operatorname{count}(v), M(v)\) and \(\operatorname{next} M(v)\), for all vertices
    \(v \neq r\)
n2cuts \(\leftarrow 0\)
// case back-edge - tree-edge
foreach \(v \neq r\) do
    if \(b \_\operatorname{count}(v)=1\) then
            mark the edges \((v, p(v))\) and \((M(v), \operatorname{low}(v))\)
            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
            \(u \leftarrow \operatorname{nextM}(v)\)
            \(n\) CutEdges \(\leftarrow 0\)
            while \(u \neq \emptyset\) and \(b \_\operatorname{count}(u)=b \_c \operatorname{count}(v)\) do
                mark the edges \((v, p(v))\) and \((u, p(u))\)
                \(n\) CutEdges \(\leftarrow n\) CutEdges +1
                \(u \leftarrow \operatorname{nextM}(u)\)
            end
            n2cuts \(\leftarrow\) n2cuts + nCutEdges \((n C u t E d g e s+1) / 2\)
            \(v \leftarrow u\)
        end
    end
```

```
Algorithm 3 Compute all \(M(v)\) and \(n e x t M(v)\).
foreach \(v \neq r\) do
    \(L(v) \leftarrow\) first child of \(v\)
    \(R(v) \leftarrow\) last child of \(v\)
    \(\operatorname{nextM}(v) \leftarrow \emptyset\)
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 \(\operatorname{low}(L(m)) \geq v\) do \(L(m) \leftarrow\) next child of \(m\)
            while \(\operatorname{low}(R(m)) \geq v\) 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
            \(n \operatorname{ext} M(c)=v\)
        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 | $\mathbf{4 2} \mathrm{MB}$ | 58 MB | 61 MB | $\mathbf{5 1} \mathrm{MB}$ |
| com-amazon | 81 MB | $\mathbf{7 2} \mathrm{MB}$ | 90 MB | 110 MB | $\mathbf{9 0} \mathrm{MB}$ |
| com-dblp | 69 MB | $\mathbf{6 2} \mathrm{MB}$ | 83 MB | 94 MB | $\mathbf{7 7} \mathrm{MB}$ |
| web-NotreDame | 61 MB | $\mathbf{5 5} \mathrm{MB}$ | 77 MB | 65 MB | $\mathbf{5 3} \mathrm{MB}$ |
| web-Stanford | 74 MB | $\mathbf{6 9} \mathrm{MB}$ | 106 MB | 111 MB | $\mathbf{7 9} \mathrm{MB}$ |
| Amazon0601 | 95 MB | $\mathbf{8 9} \mathrm{MB}$ | 133 MB | 140 MB | $\mathbf{1 0 3} \mathrm{MB}$ |
| ia-yahoo-messages | 394 MB | $\mathbf{3 4 6} \mathrm{MB}$ | 390 MB | 518 MB | $\mathbf{4 4 9} \mathrm{MB}$ |
| web-Google | 192 MB | $\mathbf{1 7 8} \mathrm{MB}$ | 258 MB | 282 MB | $\mathbf{2 0 9} \mathrm{MB}$ |
| ca-cit-HepTh | 378 MB | $\mathbf{3 3 8} \mathrm{MB}$ | 412 MB | 513 MB | $\mathbf{4 2 4} \mathrm{MB}$ |
| cit-HepTh | 373 MB | $\mathbf{3 3 8} \mathrm{MB}$ | 412 MB | 513 MB | $\mathbf{4 2 4} \mathrm{MB}$ |
| visualise-us | 460 MB | $\mathbf{4 1 0} \mathrm{MB}$ | 513 MB | 634 MB | $\mathbf{5 2 2} \mathrm{MB}$ |
| web-BerkStan | $\mathbf{2 0 9} \mathrm{MB}$ | $\mathbf{2 0 9} \mathrm{MB}$ | 332 MB | 314 MB | $\mathbf{2 3 3} \mathrm{MB}$ |
| ca-IMDB | 542 MB | $\mathbf{4 8 5} \mathrm{MB}$ | 597 MB | 738 MB | $\mathbf{6 0 8} \mathrm{MB}$ |
| ca-cit-HepPh | 626 MB | $\mathbf{5 5 6} \mathrm{MB}$ | 665 MB | 841 MB | $\mathbf{7 0 5} \mathrm{MB}$ |
| cit-HepPh | 618 MB | $\mathbf{5 5 6} \mathrm{MB}$ | 665 MB | 841 MB | $\mathbf{7 0 5} \mathrm{MB}$ |
| amazon-ratings | 835 MB | $\mathbf{7 4 6} \mathrm{MB}$ | 917 MB | 1.1 GB | $\mathbf{9 3 6} \mathrm{MB}$ |
| hugetrace-00000 | 986 MB | $\mathbf{8 6 8} \mathrm{MB}$ | 1.1 GB | 1.3 GB | $\mathbf{1 . 1} \mathrm{~GB}$ |
| rgg-n-2-20-s0 | 988 MB | $\mathbf{8 8 3} \mathrm{MB}$ | 1.1 GB | 1.3 GB | $\mathbf{1 . 1} \mathrm{~GB}$ |
| wiki-user-edits-page | 1.2 GB | $\mathbf{1} \mathrm{~GB}$ | 1.2 GB | 1.6 GB | $\mathbf{1 . 3} \mathrm{~GB}$ |
| hugetric-00010 | 1.4 GB | $\mathbf{1 . 2} \mathrm{~GB}$ | 1.5 GB | 1.9 GB | $\mathbf{1 . 6} \mathrm{~GB}$ |
| delaunay-n22 | 1.8 GB | $\mathbf{1 . 6} \mathrm{~GB}$ | 1.9 GB | 2.4 GB | $\mathbf{2} \mathrm{~GB}$ |
| co-papers-dblp | 2.1 GB | $\mathbf{1 . 9} \mathrm{~GB}$ | 2.3 GB | 2.9 GB | $\mathbf{2 . 4} \mathrm{~GB}$ |
| co-papers-citeseer | 2.2 GB | $\mathbf{2} \mathrm{~GB}$ | 2.5 GB | 3.1 GB | $\mathbf{2 . 5} \mathrm{~GB}$ |
| packing-b050 | 2.4 GB | $\mathbf{2 . 2} \mathrm{~GB}$ | 2.7 GB | 3.3 GB | $\mathbf{2 . 7} \mathrm{~GB}$ |
| human-Jung2015 | $\mathbf{1 . 3} \mathrm{GB}$ | $\mathbf{1 . 3} \mathrm{GB}$ | 2.3 GB | 2.2 GB | $\mathbf{1 . 4} \mathrm{~GB}$ |
| rgg-n-2-23-s0 | $\mathbf{2 . 3} \mathrm{GB}$ | $\mathbf{2 . 4} \mathrm{GB}$ | 3.6 GB | 3.7 GB | $\mathbf{2 . 5} \mathrm{~GB}$ |

SEA 2021

