Faster Algorithms for Computing Maximal 2-Connected Subgraphs in Sparse Directed Graphs*

Shiri Chechik[†] Tel Aviv University, Israel schechik@cs.tau.ac.il

Giuseppe F. Italiano[§] Università di Roma "Tor Vergata", Italy giuseppe.italiano@uniroma2.it Thomas Dueholm Hansen[‡]
Aarhus University, Denmark
tdh@cs.au.dk

Veronika Loitzenbauer University of Vienna, Vienna, Austria vl@cs.univie.ac.at

Nikos Parotsidis Università di Roma "Tor Vergata", Italy nikos.parotsidis@uniroma2.it

Abstract

Connectivity related concepts are of fundamental interest in graph theory. The area has received extensive attention over four decades, but many problems remain unsolved, especially for directed graphs. A directed graph is 2-edge-connected (resp., 2-vertex-connected) if the removal of any edge (resp., vertex) leaves the graph strongly connected. In this paper we present improved algorithms for computing the maximal 2-edge- and 2-vertex-connected subgraphs of a given directed graph. These problems were first studied more than 35 years ago, with $\widetilde{O}(mn)$ time algorithms for graphs with m edges and n vertices being known since the late 1980s. In contrast, the same problems for undirected graphs are known to be solvable in linear time. Henzinger et al. [ICALP 2015] recently introduced $O(n^2)$ time algorithms for the directed case, thus improving the running times for dense graphs. Our new algorithms run in time $O(m^{3/2})$, which further improves the running times for sparse graphs.

The notion of 2-connectivity naturally generalizes to k-connectivity for k > 2. For constant values of k, we extend one of our algorithms to compute the maximal k-edge-connected in time $O(m^{3/2} \log n)$, improving again for sparse graphs the best known algorithm by Henzinger et al. [ICALP 2015] that runs in $O(n^2 \log n)$ time.

1 Introduction

Connectivity is one of the most well-studied notions in graph theory. The literature covers many different aspects of connectivity related problems. In this paper we study the problem of computing the maximal k-connected subgraphs of directed graphs.

^{*}A preliminary version of this paper appeared in [2]

 $^{^{\}dagger}$ Work partially done while visiting the Università di Roma "Tor Vergata". This research was supported by the ISRAEL SCIENCE FOUNDATION (grant No. 1528/15).

[‡]Work partially done while visiting the Università di Roma "Tor Vergata". Supported by the Carlsberg Foundation, grant no. CF14-0617.

[§]Partially supported by MIUR, the Italian Ministry of Education, University and Research, under Project AMANDA (Algorithmics for MAssive and Networked DAta).

Work partially done while visiting the Università di Roma "Tor Vergata" and the University of Michigan. The research leading to these results has received funding from the Marshall Plan Foundation and the European Research Council under the European Unions Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

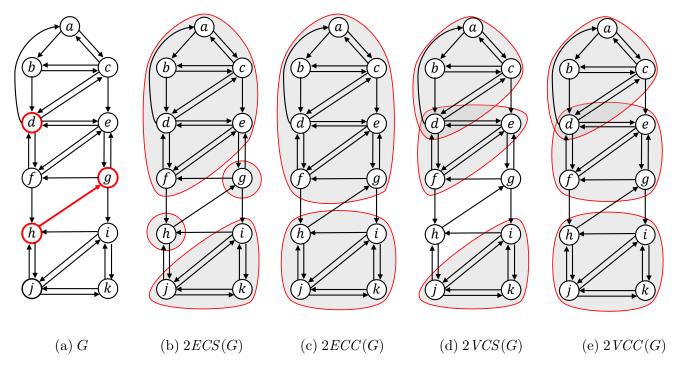


Figure 1: (a) A strongly connected digraph G; strong articulation points and strong bridges are shown in red. (b) The 2-edge-connected subgraphs of G. (c) The 2-edge-connected components of G. (d) The 2-vertex-connected subgraphs of G.

Problem definition and related concepts. Strong connectivity. Let G = (V, E) be a directed graph (digraph) with m = |E| edges and n = |V| vertices. The digraph G is said to be strongly connected if there is a directed path from each vertex to every other vertex. The strongly connected components (SCCs) of G are its maximal strongly connected subgraphs. Two vertices $u, v \in V$ are strongly connected if they belong to the same strongly connected component of G.

2-edge connectivity. An edge of G is a strong bridge if its removal increases the number of strongly connected components. Let G be a strongly connected graph. We say that G is 2-edge-connected if it has no strong bridges. Two vertices v and w are 2-edge-connected if there are two edge-disjoint paths from v to w and two edge-disjoint paths from w to v. A 2-edge-connected component of G is a maximal subset of vertices such that any pair of distinct vertices is 2-edge-connected. For a set of vertices $C \subseteq V$ its induced subgraph G[C] is a maximal 2-edge-connected subgraph of G if G[C] is a 2-edge-connected graph and no superset of G has this property. The 2-edge-connected components of G might be very different from the maximal 2-edge-connected subgraphs of G because the two edge-disjoint paths between a pair of vertices of a 2-edge-connected component might use vertices that are not in the 2-edge-connected component. (See Figure 1 for an example.)

2-vertex connectivity. Analogous definitions can be given for 2-vertex connectivity. In particular, a vertex is a strong articulation point if its removal increases the number of strongly connected components of G. Let G be a strongly connected graph. The graph G is 2-vertex-connected if it has at least three vertices and no strong articulation points. Note that the condition on the minimum number of vertices disallows for degenerate 2-vertex-connected graphs consisting of two mutually adjacent vertices (i.e., two vertices v and w and the two edges (v, w) and (w, v)). Two vertices v and v are 2-vertex-connected if there are two internally vertex-disjoint paths from v to v and two internally vertex-disjoint paths from v

to v, i.e., the paths meet at v and w but not in-between (see also [12]). A 2-vertex-connected component of G is a maximal subset of vertices such that any distinct pair of vertices is 2-vertex-connected. For a set of vertices $C \subseteq V$ its induced subgraph G[C] is a maximal 2-vertex-connected subgraph of G if G[C] is a 2-vertex-connected graph and no superset of C has this property. Note that the 2-vertex-connected components of G might be very different from the maximal 2-vertex-connected subgraphs of G.

k-connectivity. The notions of 2-edge and 2-vertex connectivity extend naturally to k-edge and k-vertex connectivity. Given a directed graph G = (V, E), a set of edges S is an $edge\ cut$ of size |S| if its removal increases the number of strongly connected components of G. A strongly connected graph is k-edge-connected if it has no edge cut of size less than k. Two vertices v and w are k-edge-connected if there are k edge-disjoint paths from v to w and k edge-disjoint paths from w to v. A k-edge-connected component of G is a maximal subset of vertices such that any pair of distinct vertices is k-edge-connected. For a set of vertices $C \subseteq V$ its induced subgraph G[C] is a $maximal\ k$ -edge-connected $maximal\ k$ -edge-connected graph and no superset of $maximal\ k$ -edge-connected $maximal\ k$ -edge-connected maxima

Undirected graphs. In undirected graphs a set of edges S is an edge cut of size |S| if its removal increases the number of connected components of the graph. An undirected connected graph is k-edge-connected if it has no edge cut of size less than k. The definitions of a vertex cut and of a k-vertex-connected graph are analogous. The remaining definitions follow immediately from the definitions for directed graphs.

Throughout the paper, we usually omit the word maximal when referring to maximal k-edge- or k-vertex-connected subgraphs.

Our results. In this paper we present $O(m^{3/2})$ time algorithms for computing the maximal 2-edge-connected subgraphs and the maximal 2-vertex-connected subgraphs of a given directed graph with m edges and n vertices. This is an improvement over the existing $O(n^2)$ time algorithms [16] whenever m is $o(n^{4/3})$. The algorithm for 2-edge-connected subgraphs is extended to compute the maximal k-edge-connected subgraphs for any constant $k \geq 2$ and runs in time $O(m^{3/2} \log n)$, improving over the existing $O(n^2 \log n)$ time algorithm [16]. The maximal k-edge-connected (and k-vertex-connected) subgraphs are defined for undirected graphs as they are for directed graphs. We also show how to adjust the algorithm to compute the maximal k-edge-connected subgraphs for undirected graphs in time $O((m+n\log n)\sqrt{n})$, where k is again viewed as a constant. For the special case where k=3, the running time for computing the 3-edge-connected subgraphs on undirected graphs is $O(m\sqrt{n})$.

Related work. In the literature the terms "components" and "blocks" have both been used to mean either k-connected components, as defined above, or the maximal (induced) k-connected subgraphs; therefore we explicitly use the term subgraphs for the latter in order to avoid further confusion.

Undirected graphs. It has been known for over 40 years how to compute the 2-edge- and 2-vertex-connected components of undirected graphs in linear time [28]. While the 2-edge-connected (resp., 2-vertex-connected) components are equal to the 2-edge-connected (resp., 2-vertex-connected) subgraphs in undirected graphs, this is no longer the case for k > 2. The first algorithm for computing the 3-vertex-connected components in linear (in the number of edges) time was by Hopcroft and Tarjan [19]. Later, Galil and Italiano [10] reduced the computation of the 3-edge-connected components to 3-vertex-connected components, thus obtaining a linear time algorithm for this case as well. Kanevsky

and Ramachandran [22] showed how to test whether a graph is 4-vertex-connected in $O(n^2)$ time. Over 20 years ago, Nagamochi and Watanabe [27] presented an algorithm for computing the k-edge-connected components for k > 3 in $O(m + k^2 n^2)$ time. The best known algorithm for this problem runs in expected $\widetilde{O}(m + nk^3)$ time and was presented by Hariharan et al. [15]. Their algorithm additionally computes a partial version of the Gomory-Hu tree [14], that represents the edge-connectivity of the pairs whose edge-connectivity is less than k; the k-edge-connected components are contracted into singleton vertices in the tree. Karger [23] showed how to determine with high probability whether an undirected graph is k-edge-connected in $\widetilde{O}(m)$ time. In a recent breakthrough, Kawarabayashi and Thorup [24] presented a deterministic algorithm with similar time bounds; Henzinger et al. [17] improve the running time even beyond the randomized algorithm. There is no study that explicitly considers the computation of the k-edge-connected or the k-vertex-connected subgraphs of undirected graphs, however, the problem can be reduced to the problem on directed graphs in a straightforward manner. Furthermore, for undirected graphs the running time (which is implied by [7], see below) of the basic algorithm for k-edge-connected subgraphs for constant k can be reduced to $O(n^2 \log n)$ by additionally maintaining a sparse certificate [3, 26, 30].

k-connected components in digraphs. Very recently Georgiadis et al. [13, 12] showed that the 2-edge-connected and the 2-vertex-connected components of a directed graph can be computed in linear time. Nagamochi and Watanabe [27] gave an O(kmn) time algorithm for computing the k-edge-connected components in directed graphs.

k-edge-connected subgraphs in digraphs. A simple algorithm for computing the maximal 2-edge-connected subgraphs is to remove at least one strong bridge of a strongly connected component of the graph and repeat on the resulting graph. It is known since 1976 how to compute a strong bridge [29] in $O(m + n \log n)$ time, and since 1985 in O(m) time [9], resulting in an O(mn) time algorithm for computing the 2-edge-connected subgraphs of a directed graph. Recently, Italiano et al. [20] gave a linear time algorithm for computing all strong bridges of a directed graph in O(m) time, of which there can be O(n) many. A similar idea can be used to compute the k-edge-connected subgraphs. In this case, in each iteration we remove the minimum edge cut of each strongly connected component of the graph, if its size does not exceed k-1. Since an edge cut of size k can be computed in time $O(km \log n)$ [7], and in each iteration we disconnect at least one pair of vertices, this algorithm runs in $O(kmn \log n)$ time. Recently, Henzinger et al. [16] presented an $O(n^2)$ time algorithm for computing the 2-edge-connected subgraphs of a directed graph and an $O(n^2 \log n)$ time algorithm for the k-edge-connected subgraphs for any constant k. Their algorithm uses a sparsification technique introduced in [1, 18] that can be used, under appropriate structural properties, to replace a factor of m in the running time of an algorithm by n.

k-vertex-connected subgraphs in digraphs. 2-vertex-connected subgraphs were first studied in 1980 by Erusalimskii and Svetlov [4], but they did not analyze the running time of their algorithm. Very recently, Jaberi [21] showed that their algorithm runs in $O(m^2n)$ time and presented an O(mn) time algorithm. Prior to Jaberi, Makino [25] gave an algorithm for computing the maximal k-vertex-connected subgraphs of a directed graph in time $O(n \cdot S)$, where S is the running time for computing a single vertex cut of size at most k-1. Since one strong articulation point [11], or even all the strong articulation points [20], can be computed in linear time, Makino's algorithm can be implemented so as to compute the 2-vertex-connected subgraphs of a directed graph in time O(mn). Combined with Gabow's algorithm for identifying k-vertex cuts [8], Makino's algorithm yields a running time of $O(mn \cdot (n + \min\{k^{5/2}, kn^{3/4}\}))$ for k-vertex-connected subgraphs; an $O(kmn^2)$ time algorithm is already implied by combining it with [5]. The recent algorithm of Henzinger et al. [16] computes the 2-vertex-connected subgraphs in time $O(n^2)$ and extends to the k-vertex-connected subgraphs for constant k with a running time of $O(n^3)$.

Key Ideas. We next outline the main ideas behind our approach. The basic algorithm for 2-edge-connected subgraphs can be seen as maintaining a partition of the vertices that is iteratively refined by identifying parts that cannot be in the same 2-edge-connected subgraph, which are then separated from each other in the maintained partition. In the basic algorithm these parts are identified by computing bridges and SCCs. The main technical contribution of this work is a subroutine that can identify a "small" part that can be separated from the rest of the graph by local depth-first searches that, starting from one given vertex, explore only the edges in this small part and a proportional number of edges outside of it.

For 2-edge-connected subgraphs we call the subgraphs identified in this way 1-edge-out and 1-edge-in components¹. A k-edge-out (resp., k-edge-in) component of a vertex u is a subgraph (induced by some set of vertices) that contains u and has at most k edges from (resp., to) the subgraph to (resp., from) the rest of the graph. We start the searches for these subgraphs from all vertices that have lost edges since the last time bridges and SCCs were computed and only recompute bridges and SCCs when no 1-edge-out or 1-edge-in component with at most \sqrt{m} edges exists².

The intuition for the local depth-first searches for edge connectivity can be better understood in terms of maximum flow in uncapacitated graphs. Assume there is a 1-edge-out component of a vertex u. Since this subgraph has at most one outgoing edge to the rest of the graph, the vertex u can send at most one unit of flow to any vertex outside of the subgraph. Thus if we find a path along which we can send one unit of flow to some vertex outside of the subgraph and then look at the residual graph given this flow, then there is no edge from the subgraph to the rest of the graph in the residual graph. We find such a flow using depth-first search and then use a second search to explore the subgraph that is still reachable from u in the residual graph.

Finding k-1 paths to send flow out of a (k-1)-edge-out component is more difficult for k>2. We show that one can exploit the properties of depth-first search to find a set of O(k) paths of which at least one of them leaves the (k-1)-edge-out component. As we have to do this for k many searches, each conducted in the residual graph after the previous search, this yields an exponential dependence on k. For any constant k>2 we compute the k-edge-connected subgraphs in time $O(m^{3/2} \log n)$ time, where the additional factor of $\log n$ compared to k=2 is due to the increased cost of computing cuts with at most k-1 edges.

The notion of a k-edge-out (resp., k-edge-in) component of a vertex u is adjusted to vertex connectivity as follows. A k-vertex-out (resp., k-vertex-in) component S of a vertex u is a subgraph that contains u and at most k vertices in the subgraph have edges from (resp., to) the subgraph to (resp., from) the rest of the graph.

For vertex connectivity some additional difficulties arise. First, the 2-vertex-connected subgraphs partition the edges rather than the vertices (apart from degenerate cases), i.e., when we find a strong articulation point and run our algorithm recursively on the subgraphs that it separates, the strong articulation point is included in each of these subgraphs. Second, the intuition of flows and residual graphs cannot be applied directly; instead, we let one depth-first search "block" specific vertices (those whose DFS subtree is adjacent to many edges) and let a second search "unblock" vertices such that it can explore the 1-vertex-out component but not the remaining graph.

It seems that the algorithm for computing k-edge-connected subgraphs can be extended to k-vertex-connected subgraphs by using the connection between flows and vertex connectivity shown in [6]. Additional details will appear in the full version of the paper.

Outline. After the preliminaries in Section 2, we present our algorithm for 2-edge-connected subgraphs

¹A similar notion called 2-isolated set was introduced in [16].

²A similar overall algorithmic structure was used in [16, Appendix B] and, for a different problem, e.g., in [1].

in Section 3. We then describe the algorithm for 2-vertex-connected subgraphs in Section 4. Finally, in Section 5, we state the results about k-edge-connectivity.

2 Preliminaries

For a directed graph G we denote by V(G) its set of vertices and by E(G) its set of edges. The reverse graph of a directed graph G = (V, E), denoted by $G^R = (V, E^R)$, is the directed graph that results from G after reversing the direction of all edges. By $G \setminus S$ and $G \setminus Q$ we denote the graph G after the deletion of a set S of vertices and after the deletion of a set Q of edges, respectively. We refer to the subgraph of G induced by the set of vertices S as G[S]. Let G be a strongly connected graph, or a strongly connected component of some larger graph. We say that deleting a set of edges G (resp., set of vertices G) disconnects G, if G is not strongly connected. Given a set of vertices G, we say that a set of edges G (resp., a set of vertices G) disconnected. Given a set of the graph if there is no pair of vertices G, a set of vertices G disconnected in $G \setminus G$ (resp., $G \setminus G$). For the sake of simplicity, we write G instead of G

We use the term tree to refer to a rooted tree with edges directed away from the root. Given a tree T, a vertex u is an ancestor (resp., descendant) of a vertex v if there is a directed path from u to v (resp., from v to u) in T. We denote by T[u,v] the path from u to v in T. We use T(u) to denote the set of vertices that are descendants of u in T.

There is a natural connection between edge cuts and maximum flow in unweighted graphs. The maximum flow that can be sent from a source vertex s to a target vertex t in directed graphs with uncapacitated edges is equal to the number of edge-disjoint paths directed from s to t. Therefore, the existence of a cut consisting of k edges directed from a set of vertices A to a set of vertices B implies that the maximum flow that can be pushed from any vertex in A to any vertex in B is at most k. Throughout the paper we implicitly use this connection between edge cuts and max flow. We further assume that the reader is familiar with depth-first search (DFS), see, e.g., [28].

3 Maximal 2-edge-connected subgraphs of a digraph

In this section we first show how to identify 1-edge-out components that contain at most Δ edges in time proportional to Δ . Applied to the reverse graph, the same algorithm finds 1-edge-in components. We then use this subroutine with $\Delta = \sqrt{m}$ to obtain an $O(m^{3/2})$ algorithm for computing the maximal 2-edge-connected subgraphs of a given directed graph.

3.1 1-edge-out and 1-edge-in components

DEFINITION 3.1. Let G = (V, E) be a digraph and $u \in V$ be a vertex. A k-edge-out component of u is a minimal subgraph S of G that contains u and has at most k outgoing edges to $G \setminus S$.

We similarly define a k-edge-in component of u.

DEFINITION 3.2. Let G = (V, E) be a digraph and $u \in V$ be a vertex. A k-edge-in component of u is a minimal subgraph S of G that contains u and has at most k incoming edges from $G \setminus S$.

See Figure 2 for an example of a k-edge-cut component and Figure 3 for an example of a k-edge-in with k = 1. Note that u may have more than one k-edge-out (resp., k-edge-in) component. Also note that for k' < k, every k'-edge-out component of u is a k-edge-out component of u as well. For the

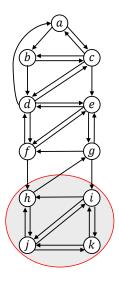


Figure 2: An example of a 1-edge-out component of j.

case when k = 1, the outgoing (resp., incoming) edge of a 1-edge-out (resp., 1-edge-in) component S is either a strong bridge or an edge between strongly connected components of the graph. Moreover, each 2-edge-connected subgraph is either completely contained in S or in $G \setminus S$ (see also [16]).

We next present an algorithm that takes as input a graph G, a vertex $u \in V(G)$, and a parameter $\Delta < m/2$, and that spends time at most $O(\Delta)$ to search for a 1-edge-out component of u in G. The algorithm may fail to find such a component, and we therefore prove the following guarantees about its outcome:

- If u has a 1-edge-out component with at most Δ edges, then the algorithm returns a 1-edge-out component for u with at most 2Δ edges.
- If every 1-edge-out component of u has more than Δ edges, then the algorithm may return a 1-edge-out component for u with at most 2Δ edges, but it may also return the empty set (i.e., fail to find a 1-edge-out component for u).

Note that by using exponential search in Δ , the algorithm can find a 1-edge-out component for a given vertex u in time that is linear in the number of edges of the smallest 1-edge-out component that contains u. For our purpose, however, it suffices to distinguish between small and large 1-edge-out components and only use one fixed choice of Δ (see Section 3.2). We use the algorithm to quickly find a small 1-edge-out component S, given a vertex u in S.

For the rest of this section, we assume that the starting vertex u can reach at least $2\Delta + 1$ edges. Notice that if u cannot reach $2\Delta + 1$ edges, then the reachable subgraph from u defines a 0-edge-out component of u containing at most 2Δ edges. In this case, the algorithm returns this 0-edge-out component. We use exactly the same algorithm executed on the reverse graph for 1-edge-in components and therefore only describe the algorithm for 1-edge-out components. First, we provide the following supporting lemmas.

LEMMA 3.1. Let (x,y) be the outgoing edge of a 1-edge-out component 1EOut(u) of a vertex u. Then u has a path to every vertex $v \in 1EOut(u)$ that is contained entirely within the subgraph 1EOut(u). Moreover, u has two edge-disjoint paths to x within 1EOut(u).

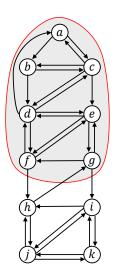


Figure 3: An example of a 1-edge-in component of f.

Proof. We begin by showing that u has a path to every vertex $v \in 1EOut(u)$ that is contained entirely within the subgraph 1EOut(u). Assume for the sake of contradiction that there is a set of vertices $C \subset 1EOut(u)$ such that the vertices of C are unreachable from u in 1EOut(u). Then there is no edge (w, z) with $w \in 1EOut(u) \setminus C$ and $z \in C$ and thus the only possible outgoing edge from $1EOut(u) \setminus C$ is (x, y). Thus, $1EOut(u) \setminus C$ is a 1-edge-out component of u, which contradicts the minimality of 1EOut(u).

We now show that u has two edge-disjoint paths to x in 1EOut(u). First, we note that all simple paths from u to x contain only vertices in 1EOut(u) since there is no edge $(x',y') \neq (x,y)$ leaving 1EOut(u). Assume, for the sake of contradiction, that all paths from u to x in 1EOut(u) share a common edge (w,z). Then, u does not have a path to z in $1EOut(u) \setminus (w,z)$. Let $C \subset 1EOut(u)$ be the set of vertices that become unreachable from u in $1EOut(u) \setminus (w,z)$. (Notice that $|C| \geq 1$ since $z \in C$.) Clearly, there is no edge (w',z') such that $w' \in V(1EOut(u)) \setminus C$ and $z' \in C$. Hence, the only outgoing edge from $1EOut(u) \setminus C$ is (w,z). Thus, $1EOut(u) \setminus C$ is a 1-edge-out component of u, which again contradicts the minimality of 1EOut(u).

Our algorithm starts a DFS traversal F_1 from u. We charge to a visited vertex its outgoing edges that were discovered by F_1 . We stop F_1 when the number of traversed edges reaches $2\Delta + 1$. Let T be the DFS tree constructed by the DFS traversal. We define the weight of a vertex v, denoted by w(v), to be the total number of edges charged to the descendants of v in T (including v). Assume u is contained in a 1-edge-out component C with at most Δ edges. Then the DFS has to leave C via its only outgoing edge in order to reach more than Δ edges. Note that for any vertex $v \neq u$ whose DFS subtree only explores edges inside C we have $w(v) < \Delta$. The following two lemmata show that the vertices with $w(v) \geq \Delta$ form a path from u to a vertex outside of C that we can then use to block the outgoing edge such that the second traversal explores exactly C.

Lemma 3.2. Let 1EOut(u) be a 1-edge-out component of u with outgoing edge (x,y) such that $|E(1EOut(u))| \leq \Delta$, and let T be a DFS tree of a DFS traversal from u that visits $2\Delta + 1$ edges.

Then $w(v) \ge \Delta$ for each vertex v on the path from u to y in T, i.e., $v \in T[u, y]$, and $w(v) < \Delta$ for each $v \in 1EOut(u) \setminus T[u, x]$.

Proof. Since $|E(1EOut(u))| \leq \Delta$, the DFS traversal has to visit vertices outside of 1EOut(u) to reach more than Δ edges. The DFS traversal can leave 1EOut(u) only by using the edge (x,y) and it can do so only once. As the DFS traversal visits at least $2\Delta + 1$ edges, it visits at least Δ edges in the subtraversal from y (i.e., subsequent to exploring the DFS tree edge (x,y)). Therefore, for each $v \in T[u,y]$ it holds that $w(v) \geq \Delta$. Moreover, any subtraversal that does not visit vertices outside of 1EOut(u) cannot include more than Δ edges. None of the subtraversals from vertices $v \in 1EOut(u) \setminus T[u,y]$ can visit vertices outside of 1EOut(u), since these vertices are either visited after the edge (x,y), or their subtraversal can not visit x (i.e., the vertex v cannot reach x without using backedges w.r.t. the DFS tree). Thus, for each vertex $v \in 1EOut(u) \setminus T[u,x]$, it holds that $w(v) < \Delta$.

LEMMA 3.3. Let F be a DFS traversal that visited $2\Delta + 1$ edges and let T be the DFS tree generated by F. The edges $e = (x, y) \in T$ with $w(y) \geq \Delta$ form a path in T.

Proof. Assume by contradiction that there are two distinct tree edges $e_1 = (x_1, y_1)$ and $e_2 = (x_2, y_2)$ with $w(y_1) \geq \Delta$ and $w(y_2) \geq \Delta$ that do not have an ancestor-descendant relation in T (i.e., y_1 is not an ancestor of x_2 and y_2 is not an ancestor of x_1). Since also the edges (x_1, y_1) and (x_2, y_2) are visited by T, this contradicts the fact that the traversal visited $2\Delta + 1$ edges. Therefore, all edges $e = (x, y) \in T$ with $w(y) \geq \Delta$ form a path in T.

After the execution of the first DFS F_1 , by Lemma 3.3, there is a path P of T such that we have $w(y) \geq \Delta$ for every edge e = (x, y) of P. We call this path the heavy path of F_1 , and the edges contained in the heavy path the heavy edges of F_1 . Note that (1) the heavy path has to leave a 1-edgeout component of u with at most Δ edges for the search to reach more than Δ edges and (2) the heavy path cannot enter the component again after leaving it because the subtree of any incoming edge of the component cannot contain Δ or more edges as the only outgoing edge of the component was already used. We construct the residual graph G' formed from G by reversing the direction of the heavy edges of F_1 . The residual graph will be used as follows. If there exists a 1-edge-out component 1EOut(u) of u containing at most Δ edges, then the heavy path P can be interpreted as sending one unit of flow out of 1EOut(u) and in the residual graph with respect to this flow no additional unit of flow can be sent out of 1EOut(u). That means that no other search from u is able to have an outgoing path from 1EOut(u). Next, we execute a second traversal F_2 from u (not necessarily a depth-first search) on G'. We show that if there exists a 1-edge-out component 1EOut(u) of u containing at most Δ edges, this second traversal has two main properties: (i) it never visits edges outside of G'[V(1EOut(u))], and (ii) it visits all the edges in G'[V(1EOut(u))]. Whenever F_2 traverses more than Δ edges, we terminate the search and conclude that any 1-edge-out component of u contains more than Δ edges.

LEMMA 3.4. Let G' be the residual graph obtained from G by reversing the direction of the heavy edges of F_1 . The traversal F_2 reaches at most Δ edges in G' if and only if there exists a 1-edge-out component 1EOut(u) of u containing at most Δ edges. Moreover, if F_2 traverses at most Δ edges, then the subgraph in G induced by the vertices traversed by F_2 defines 1EOut(u).

Proof. Let us first assume that there exists a 1-edge-out component 1EOut(u) of u that contains at most Δ edges and has one outgoing edge (x, y). By Lemma 3.2, the edge (x, y) is reversed in the

residual graph G'. Moreover, the lemma implies that no incoming edge to 1EOut(u) is reversed in G'because each incoming edge (v,z) either has $w(z) < \Delta$ or $z \in T[u,x]$; in the latter case (v,z) cannot be a DFS tree edge as T[u,x] is contained in 1EOut(u), thus $v \notin T[u,x]$, and hence (v,z) being a DFS tree edge would generate a cycle in the DFS tree. Thus, G'[V(1EOut(u))] has no outgoing edges to $G'[V(G) \setminus V(1EOut(u))]$. Therefore, F_2 cannot visit more than Δ edges. We now show that F_2 visits all vertices in G'[V(1EOut(u))] using only paths internal to G'[V(1EOut(u))]. Notice that this does not trivially follow from Lemma 3.1 since we are operating on the residual graph G', where the direction of some edges of 1EOut(u) is reversed. Assume by contradiction that u cannot visit all vertices in G'[V(1EOut(u))]. Then, there is a set of vertices $C \subset V(1EOut(u))$ that has no incoming edge from $V(1EOut(u)) \setminus C$ in the residual graph G'. By Lemma 3.3 the edges that are reversed in the residual graph G' form a path P in the DFS tree of F_1 . The path P contains an incoming edge to C in G since otherwise $1EOut(u) \setminus C$ is a 1-edge-out component of u, contradicting the minimality of 1EOut(u). Since C has no incoming edges from $1EOut(u) \setminus C$ in G', we have that P has no outgoing edges from C to $1EOut(u) \setminus C$. Therefore $T[u, y] \in P$ implies $x \in C$. Since P does not enter 1EOut(u) after leaving through (x, y), only one edge incident to C was reversed in G'. As there is no edge incident to C in G', this is a contradiction to Lemma 3.1, which says that u has two edge-disjoint paths to x. Hence no such set C exists and F_2 traverses all vertices of 1EOut(u).

Now we show the opposite direction. Assume that F_2 visits at most Δ edges in the residual graphs. We will show that there exists a 1-edge-out component 1EOut(u) of u that contains at most Δ edges and that is given by the subgraph induced by the vertices traversed by F_2 . Let C be the subgraph that F_2 traversed in the residual graph. Then C has no outgoing edges in G', since otherwise their neighbors would also be traversed by F_2 . Since F_1 visited $2\Delta + 1$ edges, there is at least one edge e^* incoming to C in G' that was reversed. Note that there cannot exist more than one incoming edge to C in G' that was reversed after F_1 , since that would imply the existence of an outgoing edge from C since the set of reversed edges forms a path by Lemma 3.3. Hence u has no path to any of the vertices in $V \setminus C$ in the residual graph G', and has only one outgoing edge in the original graph G. Therefore, after restoring the reversed edges, C forms a 1-edge-out component of u that contains at most Δ edges, with the only outgoing edge being e^* . Notice that the vertices of C were all traversed by F_2 . It remains to show that there is no 1-edge-out component 1EOut'(u) of u with one outgoing edge (x',y') and such that $1EOut'(u) \subset 1EOut(u)$. Assume by contradiction that there exists such a component. By Lemma 3.2 the traversal F_1 reversed (x', y'), and there is no other outgoing edge from 1EOut'(u) in the residual graph. Therefore, F_2 cannot visit vertices outside 1EOut'(u). A contradiction to the fact that F_2 visited all the edges and vertices in 1EOut(u). \square

Recall that we assumed in the beginning of this section that u reaches at least $2\Delta + 1$ edges. If this is not satisfied, we return the set of reachable vertices from u, which is a 0-edge-out component of u with at most 2Δ edges. Otherwise the first DFS search F_1 is able to visit $2\Delta + 1$ edges. After the execution of the second traversal F_2 on the residual graph G', we can answer whether there exists a 1-edge-out component of u with at most Δ edges, as shown in Lemma 3.4. The pseudocode of our algorithm is illustrated in Procedure 1EdgeOut. The following lemma summarizes the result of this section.

LEMMA 3.5. Procedure 1EdgeOut computes a 1-edge-out (resp., 1-edge-in) component of u with at most 2Δ edges or decides that there is no 1-edge-out (resp., 1-edge-in) component of u with at most Δ edges. Moreover, Procedure 1EdgeOut runs in $O(\Delta)$ time.

3.2 Computing the 2-edge-connected subgraphs. Let G = (V, E) be a digraph. A straightforward algorithm for computing the 2-edge-connected subgraphs is to recursively remove, from G, one

Procedure 1EdgeOut(G, u, Δ)

```
Input: Digraph G = (V, E), a vertex u, and an integer \Delta
   Output: Either a 1-edge-out component of u with at most 2\Delta edges or \emptyset; if \emptyset is returned, then
              every 1-edge-out component that contains u has more than \Delta edges
 1 Execute DFS F_1 from u for up to 2\Delta + 1 edges
 2 Let S_1 be the vertices reached by F_1
 3 if F_1 cannot reach 2\Delta + 1 edges then
       return G[S_1] as 1-edge-out component of u
 5 else
       Let P be the heavy path of F_1
 6
       Let G' be G after reversing the direction of the edges of P
 7
       Execute DFS F_2 from u on G' for up to \Delta + 1 edges
 8
       Let S_2 be the vertices reached by F_2
 9
       if F_2 cannot reach \Delta + 1 edges then
10
        return G[S_2] as 1-edge-out component of u
11
       else
12
          return \emptyset
13
```

strong bridge of each strongly connected component of G until no strong bridges can be found. In each recursive call at least one vertex becomes disconnected from the rest of the graph. Since computing the strongly connected components and one strong bridge (or all strong bridges) of a digraph can be done in linear time, this simple algorithm runs in O(mn) time.

In our algorithm we build on the simple algorithm described above. The high-level idea of our approach is to (a) find subgraphs with at most \sqrt{m} edges that are not 2-edge-connected to the rest of the graph in total time $O(m\sqrt{m})$ and by this (b) limit the maximum recursion depth to \sqrt{m} by only making recursive calls when large subgraphs will be disconnected from each other or the remaining graph has at most $O(\sqrt{m})$ edges. This is done as follows. We use the terms small and large components to refer to subgraphs that contain at most and more than \sqrt{m} edges, respectively. We first identify all the small components that can be disconnected from the rest of the graph by a single edge deletion. In each recursive call of the algorithm we maintain a list L of vertices for which we want to identify small 1-edgeout and 1-edge-in components. Initially, we set the list L to contain all vertices in order to find all small components that can be separated by at most one edge. We search for such small subgraphs using the algorithm from Section 3.1. We compute 1-edge-in components by executing $1EdgeOut(G^R, u, \sqrt{m})$, where G^R is the reverse graph of G. Whenever we find a small 1-edge-out or 1-edge-in component, we remove all its incident edges and search for more small 1-edge-out or 1-edge-in components in the remaining graph. We do that by inserting the endpoints of the deleted edges into the list L. If, on the other hand, we cannot find new small components, we conclude that either the remaining graph is 2-edge-connected or there are at least two large sets of vertices that will get disconnected by either recomputing SCCs or by the removal of a strong bridge. In a final phase of each recursive call we compute the SCCs of the graph and for each SCC we remove one strong bridge and then recursively call the algorithm on every resulting SCC. Before each recursive call, we initialize the lists L to contain the vertices that lost an edge during the last phase of the parent recursive call. We keep this list in order to restrict the total number of searches for small separable components to O(m+n) since, after initially adding all vertices to the list of the initial call, we only add the endpoints of deleted edges into

Input: A strongly connected digraph G = (V, E) and a list of vertices L (initially L = V) Output: The 2-edge-connected subgraphs of G1 Let m_0 be number of edges of initial graph 2 if G has no strong bridge then 3 \lfloor return $\{G\}$ as 2-edge-connected subgraph 4 while $L \neq \emptyset$ and G has more than $2\sqrt{m_0}$ edges do 5 \lfloor Extract a vertex u from L6 \rfloor $S \leftarrow 1$ EdgeOut $(G, u, \sqrt{m_0})$

```
8 If either S or S<sup>R</sup> is not empty, remove from G all edges incident to one non-empty set of S and S<sup>R</sup> and add their endpoints to L
9 Compute SCCs C<sub>1</sub>,..., C<sub>c</sub> of G
10 U ← ∅
```

```
11 foreach C_i, 1 \le i \le c do

12 Remove one strong bridge from C_i (if one exists)

13 Recompute SCCs and delete the edges between them

14 foreach SCC C' do

15 Initialize L' with the vertices of C' that are endpoints of newly deleted edges

16 U \leftarrow U \cup 2ECS(C', L')
```

17 return U

7

Algorithm 1: 2ECS(G, L)

 $S^R \leftarrow 1 \text{EdgeOut}(G^R, u, \sqrt{m_0})$

the lists (which are O(m) many). Algorithm 1 contains the pseudocode of our algorithm.

The following is a key property that allows us to find small sets that are not strongly connected to the rest of the graph, or that can be disconnected by deleting a single edge, or to conclude that there are no such small sets. Every new 1-edge-out component that appears in the graph throughout the algorithm must have lost an outgoing edge. Respectively, every new 1-edge-in component that appears must have lost an incoming edge. Therefore, we use the list L to keep track of the vertices that have lost an edge and for each such vertex u we search for new small 1-edge-out or 1-edge-in components of u. If no such small components exist in a set of vertices C, then we know that either C is a 2-edge-connected subgraph or either recomputing SCCs or the deletion of any strong bridge disconnects at least two large components. These properties are summarized in the following lemma.

LEMMA 3.6. Let C be a set of vertices of G. Every 1-edge-out or 1-edge-in component (of some vertex $u \in C$) in G[C] that is not such a component in G must contain an endpoint of an edge incident to G[C]. Moreover, if there is no 1-edge-out or 1-edge-in component containing at most Δ edges for any vertex $u \in C$ in G[C], then one of the following holds:

- (a) G[C] is a 2-edge-connected subgraph of G.
- (b) There are two sets $A, B \subset C$ with $|E(G[A])|, |E(G[B])| > \Delta$ such that A and B are in different strongly connected components of G[C].
- (c) For each strong bridge of G[C] there are two sets $A, B \subset C$ with $|E(G[A])|, |E(G[B])| > \Delta$ that get disconnected by the deletion of the strong bridge.

Proof. We first show that every 1-edge-out component 1EOut(u) of some vertex $u \in C$ that is no 1-edge-out component in G must contain a vertex $x \in 1EOut(u)$ such that there is an edge (x,y) with $y \notin C$. Assume, by contradiction, that 1EOut(u) exists but there is no such edge (x,y) in G with $x \in 1EOut(u)$ and $y \notin C$. In this case we have that the very same component 1EOut(u) is a 1-edge-out component of u in G. The same argument on the reverse graph shows that every new 1-edge-in component (of some vertex $u \in C$) in G[C] must contain an endpoint of an edge incident to G[C] in G.

We now turn to the second part of the lemma. If G[C] is strongly connected and does not contain a strong bridge, then G[C] is 2-edge-connected and thus (a) holds. If G[C] is not strongly connected, then it contains (at least) two disjoint sets $A, B \subset C$ such that both G[A] and G[B] are strongly connected components of G[C] and G[A] has no outgoing edge in G[C] (i.e., G[A] is a sink in the DAG of SCCs of G[C]) and G[B] has no incoming edge in G[C] (i.e., G[B] is a source in the DAG of SCCs of G[C]). That is, in G[C] we have that G[A] is or contains a 1-edge-out component (and is a 0-edge-out component) of some $u \in C$ and G[B] contains a 1-edge-in component of some $u' \in C$. Both can have the same property in G or contains (resp. be) new such components in G[C] compared to G. In any case it contradicts the assumptions if one of them has at most G[C] deges and otherwise statement (b) holds. If G[C] is strongly connected and contains a strong bridge G[C] and G[C] in this case G[C] is two disjoint sets G[C] by considering the DAG of SCCs of $G[C] \setminus G[C]$. In this case G[C] is the only incoming edge of G[C] and the only outgoing edge of G[C]. Thus we have that case (c) holds if the assumptions of the lemma are satisfied. G[C]

LEMMA 3.7. Algorithm 2ECS runs in $O(m\sqrt{m})$ time.

Proof. First notice that each time we search for a 1-edge-out or a 1-edge-in component, we are searching for a component with one outgoing (resp., incoming) edge containing at most \sqrt{m} edges or with no outgoing (resp., incoming) edges and at most $2\sqrt{m}$ edges. We can identify if such a component containing a given vertex u exists in time $O(\sqrt{m})$ by using the algorithm of Section 3.1. We initiate such a search from each vertex that appears in the list L of some recursive call of the algorithm. Initially, we place all vertices in the list L. Throughout the algorithm we insert into L only vertices that are endpoints of deleted edges. Therefore, the number of vertices that are added to the lists L throughout the algorithm is O(m). Identifying which edges to delete (and thus which vertices to add to L) can be done in time proportional to the deleted edges and the edges in the 1-edge-out or 1-edge-in component. Hence, the total time spent on these searches (and the subsequent operations) is $O(m\sqrt{m})$.

Consider now the time spend in each recursive call without the searches for 1-edge-out and 1-edge-in components. Let G' be the graph for which the recursive call is made and let $m_{G'} = |E(G')|$. In each recursive call the algorithm spends $O(m_{G'})$ time searching for strong bridges in G' in lines 2 and 12 and computing SCCs in lines 9 and 13. Since the subgraphs of different recursive calls at the same recursion depth are disjoint, the total time spent at each level of the recursion is O(m). We now bound the recursion depth with $O(\sqrt{m})$.

We show that the graph passed to each recursive call has at most $\max\{m_{G'} - \sqrt{m}, 2\sqrt{m}\}$ edges, or G' is a 2-edge-connected subgraph and thus the recursion stops. This implies a recursion depth of $O(\sqrt{m})$ as follows. If the graph passed to a recursive call has at most $2\sqrt{m}$ edges, then also the number of vertices of this graph is at most $2\sqrt{m}$. Therefore, even if the algorithm only removes one strong bridge from every strongly connected component in each recursive call, the total recursion depth is at most $O(\sqrt{m})$. On the other hand, the number of times that the graph passed to a recursive call has at least \sqrt{m} fewer edges than G' is at most \sqrt{m} . Overall, this implies that the recursion depth is bounded by $O(\sqrt{m})$.

It remains to show the claimed bound on the size of the graph passed to a recursive call in line 16. For every 1-edge-out or 1-edge-in component with at most $2\sqrt{m}$ edges that is discovered throughout the algorithm, its incident edges are removed and therefore it will be in a separate strongly connected component with at most $2\sqrt{m}$ edges. Let C be the set of vertices that were not included in any 1-edge-out or 1-edge-in component. By Lemma 3.6 the subgraph G'[C] either is a 2-edge-connected subgraph or there are two sets A and B with $|E(A)|, |E(B)| > \sqrt{m}$ that will be separated in Line 12. Thus, every graph passed to the recursive call will have at most $\max\{|E(G')| - \sqrt{m}, 2\sqrt{m}\}$ edges. The lemma follows.

Lemma 3.8. Let G = (V, E) be a strongly connected digraph. Algorithm 2ECS(G, V) returns the maximal 2-edge-connected subgraphs of G.

Proof. First note that by assumption the initial call to the algorithm is on a strongly connected graph and that recursive calls are only made on strongly connected subgraphs. Thus whenever the algorithm reports a 2-edge-connected subgraph in line 3, then it is a strongly connected subgraph that does not contain any strong bridges, which is by definition a 2-edge-connected subgraph. Thus it suffices to show that the algorithm reports all the maximal 2-edge-connected subgraphs. Notice that this also implies that the reported 2-edge-connected subgraphs are maximal. Let C be a maximal 2-edge-connected subgraph. We show that the vertices of C do not get separated by the algorithm, and therefore C is reported eventually as a 2-edge-connected subgraph. Since there are two edge-disjoint paths between every pair of vertices in C, any search for either a 1-edge-out or a 1-edge-in component of a vertex u (lines 6-7) either returns a superset of C or fails to identify such a set containing a subset of the vertices of C. Furthermore, notice that any deletion of an edge that does not have both endpoints in C does not affect the fact that C is 2-edge-connected. That is, unless an edge with both endpoints in C is deleted, no strong bridge appears in C. Thus, it remains to show that no edge (x,y) such that $x,y \in C$ is ever deleted throughout the algorithm. The edges deleted in line 8 of the algorithm are incident to a 1-edge-out or a 1-edge-in component. Since C is always fully inside or fully outside of such a set, no edge from C is deleted. The edges deleted in line 12 are strong bridges and the edges deleted in line 13 before the recursive calls are between separate strongly connected components. Since C is 2-edge-connected, no edges from C are deleted. Finally, notice that at each level of recursion at least one of the strong bridges of each strongly connected component of the graph is deleted and the algorithm is recursively executed in each resulting strongly connected component. Thus, finally there will be a recursive call for each strongly connected subgraph that does not contain strong bridges, including C. \square

Algorithm 2ECS can be applied to an arbitrary, i.e., not necessarily strongly connected, digraph by taking the union of the 2-edge-connected subgraphs of the SCCs of the input graph. We have shown the following theorem.

Theorem 3.1. The maximal 2-edge-connected subgraphs of a digraph can be computed in $O(m^{3/2})$ time.

4 Maximal 2-vertex-connected subgraphs in directed graphs

In this section we first introduce a procedure for identifying 1-vertex-out components containing at most Δ edges in time proportional to Δ . The same algorithm applied to the reverse graph identifies 1-vertex-in components. We then use this subroutine with $\Delta = \sqrt{m}$ to obtain a $O(m^{3/2})$ algorithm for computing the maximal 2-vertex-connected subgraphs of a given directed graph.

4.1 1-vertex-out and 1-vertex-in components. We begin with the definition of k-vertex-out and k-vertex-in components of a vertex u. In algorithms for (k+1)-vertex-connected subgraphs we want to detect when for some subgraph (induced by a vertex set) the number of vertices with outgoing (resp. incoming) edges has decreased to at most k. This can only happen when some vertex has lost adjacent edges. Intuitively, the vertex u for which we search for a k-vertex-out or a k-vertex-in component is a candidate for a vertex that is contained in such a subgraph and has lost edges adjacent to the subgraph. Thus it is sufficient to search for k-vertex-out and k-vertex-in components for which the set of vertices with outgoing resp. incoming edges does not include the starting vertex u. This is reflected in the definitions below and implies in particular that a k-vertex-out (resp. k-vertex-in) component contains all vertices that have an edge from (resp. to) u.

DEFINITION 4.1. Let G = (V, E) be a digraph and $u \in V$ be a vertex. A k-vertex-out component of u is a minimal subgraph S of G that contains u and has at most k vertices $X \subset V(S)$, $u \notin X$, with outgoing edges to $G \setminus S$.

DEFINITION 4.2. Let G = (V, E) be a digraph and $u \in V$ be a vertex. A k-vertex-in component of u is a minimal subgraph S of G that contains u and has at most k vertices $X \subset V(S)$, $u \notin X$, with incoming edges from $G \setminus S$.

As in the case of k-edge-out (resp., k-edge-in) components, a vertex u may have more than one k-vertex-out (resp., k-vertex-in) component. Also note that for k' < k, every k'-vertex-out component of u as well. For the case when k = 1, the only vertex x that has outgoing (resp., incoming) edges from a 1-vertex-out (resp., 1-vertex-in) component S is either a strong articulation point or a vertex that has outgoing (resp., incoming) edges to vertices that belong to different strongly connected components than x. Moreover, each 2-vertex-connected subgraph is either completely contained in S or in $(G \setminus S) \cup \{x\}$.

For a given vertex u and a parameter $\Delta < m/2$, we present an algorithm for computing a 1-vertexout component of u that runs in time $O(\Delta)$ and has the following guarantees:

- If there exists a 1-vertex-out component of u with at most Δ edges, then it returns a 1-vertex-out component of u with at most 2Δ edges.
- If no 1-vertex-out component with at most Δ edges exists, it might either return a 1-vertex-out component of u with at most 2Δ edges or the empty set.

As mentioned earlier, our algorithm identifies a 1-vertex-out component of u in time proportional to its size (i.e., its number of edges). In Section 4.2 we will use this algorithm to determine quickly whether there exist 1-vertex-out (resp., 1-vertex-in) components of small size (namely, containing at most a predefined number of edges Δ), or conclude that all 1-vertex-out (resp., 1-vertex-in) components have large size. We show that this is sufficient to bound the total running time of our algorithm for computing the 2-vertex-connected subgraphs.

For the rest of this section, we assume that we are given a starting vertex u that can reach at least $2\Delta + 1$ edges. If this is not the case, then the reachable subgraph from u defines a valid 1-vertex-out component of u that contains at most 2Δ edges and has no outgoing edges. The exactly same algorithm executed on the reverse graph computes a 1-vertex-in component of u that contains at most 2Δ edges, or we conclude that there is no 1-vertex-in component of u with at most Δ edges. Since the algorithm for computing a 1-vertex-in component of u when executed on the reverse graph, we only describe the algorithm for finding 1-vertex-out components. The following lemma provides intuition for the properties that we (implicitly) exploit in our algorithm.

LEMMA 4.1. Let 1VOut(u) be a 1-vertex-out component of a vertex u that has outgoing edges and let $x \neq u$ be the only vertex that has outgoing edges from 1VOut(u). It holds that u has a path to every vertex $v \in 1VOut(u)$ that is contained entirely within the subgraph 1VOut(u). Moreover, either there is an edge from u to x or u has two internally vertex-disjoint paths to x in 1VOut(u).

Proof. We begin by showing that u has a path to every vertex $v \in 1VOut(u)$ that is contained entirely within the subgraph 1VOut(u). Assume, for the sake of contradiction, that there is a set of vertices C such that the vertices of C are unreachable from u in 1VOut(u). Then there is no edge (w, z) where $w \in 1VOut(u) \setminus C$ and $z \in C$ and thus the outgoing edges from the vertex x are the only possible outgoing edges from $1VOut(u) \setminus C$. Thus, $1VOut(u) \setminus C$ is a 1-vertex-out component of u, which contradicts the minimality of 1VOut(u).

We now show that if there is no edge from u to x, then u has two internally vertex-disjoint paths to x in 1VOut(u). First, we note that all simple paths from u to x contain only vertices in 1VOut(u) since there is no other vertex $x' \neq x$ such that $x' \in 1VOut(u)$ and x' has edges leaving 1VOut(u). Assume, for the sake of contradiction, that all paths from u to x in 1VOut(u) share a common vertex w that is different from both u and x. Then, u does not have a path to x in $1VOut(u) \setminus \{w\}$. Let C be the set of vertices that become unreachable from u in $1VOut(u) \setminus \{w\}$. (Notice that $|C| \geq 1$ since $x \in C$.) Clearly, there is no edge (w', z') such that $w' \in 1VOut(u) \setminus (C \cup \{w\})$ and $z' \in C$, since otherwise z' would be reachable from u in $1VOut(u) \setminus \{w\}$. Hence, the only vertex that has edges leaving $1VOut(u) \setminus C$ is w. Thus, $1VOut(u) \setminus C$ is a 1-vertex-out component of u, which contradicts the minimality of 1VOut(u). The lemma follows.

Our algorithm for identifying 1-vertex-out components begins with a DFS traversal F_1 from u. As for 1-edge-out components, the idea is that the first DFS traversal "uses and blocks" the only vertex that has edges out of a 1-vertex-out component if such a component of size at most Δ exists, and then a second traversal explores exactly the 1-vertex-out component. In the DFS traversal F_1 we charge to a visited vertex its outgoing edges that were traversed. We stop F_1 when the number of the traversed edges reaches $2\Delta + 1$. Let T be the DFS tree constructed by the DFS traversal. We define the weight of a vertex v, denoted by w(v), to be the total number of edges charged to the descendants of v in T (including v).

Assume that u has a 1-vertex-out component C containing at most Δ edges and exactly one vertex x with outgoing edges to $V \setminus C$. It is easy to see that F_1 is guaranteed to traverse at least $\Delta + 1$ edges outside of C (since it visits at least $2\Delta + 1$ edges and $|E(C)| \leq \Delta$), and therefore, since x is the only vertex with outgoing edges from C, we have $w(x) \geq \Delta + 1$. Moreover, for any vertex $v \neq u$ whose DFS subtree explores only vertices inside C, we have $w(v) < \Delta$. The following two lemmata show that the vertices with weight more than Δ form a path from u to a vertex outside of C that we can then use to block the only vertex with outgoing edges for the second traversal starting from u.

LEMMA 4.2. Let 1VOut(u) be a 1-vertex-out component of u such that $|E(1VOut(u))| \leq \Delta$, let x be the only vertex that has edges leaving 1VOut(u), and let T be a DFS tree generated by a DFS traversal from u that visits $2\Delta + 1$ edges. Then, for each $v \in T[u,x]$ it holds that $w(v) \geq \Delta + 1$ and for each $v \in 1VOut(u) \setminus T[u,x]$ it holds that $w(v) \leq \Delta$.

Proof. Since 1VOut(u) contains at most Δ edges, the only way a DFS traversal can visit $2\Delta + 1$ edges is by visiting at least $\Delta + 1$ edges outside of 1VOut(u). By the fact that x is the only vertex that has edges leaving 1VOut(u), it follows that $w(x) \geq \Delta + 1$, and therefore, for each $v \in T[u, x]$ it holds that $w(v) \geq \Delta + 1$. Note that the DFS reaches each vertex and in particular x only once (i.e., each vertex

of T except u has exactly one incoming edge in T), and that any traversal from u that does not visit vertices $v \notin 1VOut(u)$ cannot be charged more than Δ edges. None of the subtraversals from vertices $v \in 1VOut(u) \setminus T[u,x]$ can visit vertices outside of 1VOut(u) since either v is visited after x or the subtraversal can not reach x. In both cases the subtraversal from v can not use the outgoing edges of x to visit more than Δ edges. Thus, for each vertex $v \in 1VOut(u) \setminus T[u,x]$, it holds that $w(v) \leq \Delta$. \Box

After the traversal F_1 , we say that a vertex v is blocked if $w(v) \geq \Delta + 1$. Next, we start a second traversal F_2 from u (not necessarily a depth-first search) as follows. The traversal F_2 can only visit the vertex u and vertices that are not blocked. We say that the traversal reaches a vertex v whenever it traverses an edge incoming to v; thus F_2 can reach blocked vertices but not visit them and all vertices that are visited are also reached by F_2 . Whenever F_2 reaches a blocked vertex v, we unblock all blocked vertices on $T[u,v] \setminus v$. (Notice that v itself is not unblocked.) Assuming that there exists a 1-vertex-out component of u with at most Δ edges for which $x \neq u$ is the only vertex with outgoing edges. Then this second traversal F_2 has two main properties: (i) it never unblocks x, and (ii) it reaches all vertices in 1VOut(u). Since we are interested only in computing a 1-vertex-out component of u containing at most Δ edges (recall that we assumed in the beginning that u can reach at least $2\Delta + 1$ edges), we terminate F_2 whenever it visits $\Delta + 1$ edges. If the traversal F_2 visits $\Delta + 1$ edges, we conclude that there is no 1-vertex-out component of u containing at most Δ edges. Before proving the above claim, we first show the following supporting lemma, which says that the blocked vertices form a path in the DFS tree; we call this path the heavy path of F_1 .

LEMMA 4.3. Let F be a DFS traversal that visits $2\Delta + 1$ edges and let T be its DFS tree. The vertices v with $w(v) \ge \Delta + 1$ form a path in T.

Proof. Assume, by contradiction, that the vertices v with $w(v) \geq \Delta + 1$ do not form a path on T. That means, there are two vertices x and y with $w(x), w(y) \geq \Delta + 1$ that do not have an ancestor-descendant relation in T, i.e., $T(x) \cap T(y) = \emptyset$. This is a contradiction to the fact that F visits only $2\Delta + 1$ edges. Therefore, the vertices v with $w(v) \geq \Delta + 1$ form a path in T.

LEMMA 4.4. Let G be a graph where the vertices v with $w(v) \geq \Delta + 1$ are blocked after the DFS traversal F_1 . If there exists a 1-vertex-out component of u containing at most Δ edges, then F_2 traverses at most Δ edges. Moreover, if F_2 traverses at most Δ edges, the subgraph induced by the vertices reached by F_2 (including a reached but not unblocked vertex) defines a 1-vertex-out component of u that contains at most $\Delta + 1$ vertices and at most 2Δ edges.

Proof. Let us first assume that there exists a 1-vertex-out component 1VOut(u) of u that contains at most Δ edges and that all edges leaving 1VOut(u) share a common source x. By Lemma 4.2, x is blocked. The traversal F_2 cannot visit more than Δ edges, since u cannot visit vertices $v \notin 1VOut(u)$ avoiding x, and hence, F_2 cannot unblock x.

Now we show the opposite direction. Assume that F_2 visits at most Δ edges and thus reaches at most $\Delta + 1$ vertices. We will show that there exists a 1-vertex-out component 1VOut(u) of u that is induced by the vertices reached by F_2 and has at most 2Δ edges. If F_2 unblocks the whole path $P_{blocked}$, then it would visit at least $2\Delta + 1$ edges, since F_1 did so. Hence, there is at least one vertex that remains blocked after the traversal of F_2 ; let v^* be such a vertex. Let C be the set of vertices that were reached by F_2 . Then, C has at most one blocked vertex, which is v^* , since whenever two vertices

Procedure 1VertexOut(G, u, Δ)

```
Input: Digraph G = (V, E), a vertex u, and an integer \Delta
   Output: Either a 1-vertex-out component of u with at most 2\Delta edges or \emptyset; if \emptyset is returned,
              then no 1-vertex-out component of u with at most \Delta edges exists
 1 Execute DFS F_1 from u for up to 2\Delta + 1 edges
 2 Let S_1 be the vertices reached by F_1
 3 if F_1 cannot reach 2\Delta + 1 edges then
      return G[S_1] as 1-edge-out component of u
 5 else
       Block the vertices on the heavy path of F_1
 6
       Execute a DFS F_2 from u for up to \Delta + 1 edges and whenever a blocked vertex v is reached:
 7
        unblock vertices from u to the predecessor of v in F_1 and continue the DFS without v
       Let S_2 be the vertices reached by F_2 (including reached but not unblocked vertices)
 8
       if F_2 cannot reach \Delta + 1 edges then
 9
        return G[S_2] as 1-vertex-out component of u
10
       else
11
          return \emptyset
12
```

of the path $P_{blocked}$ are reached, reaching the vertex further away from u on $P_{blocked}$ unblocks all the blocked vertices on the tree path from u. Notice that all edges leaving C are from v^* . Moreover, v^* might have at most Δ edges to vertices in C that were not traversed. Thus the subgraph induced by C contains u, has only one vertex $v^* \neq u$ with edges out of the subgraph, and contains at most 2Δ edges. Notice that all vertices in C were reached by F_2 . To show that C induces a 1-vertex-out component of u it remains to show that there is no proper subset C' of C that contains u and has at most one vertex $x' \neq u$ with edges out of C'. Assume by contradiction that there exists such a vertex set C'. By Lemma 4.2 the traversal F_1 would have blocked x', and there is no other outgoing edge from a vertex of C' to a vertex in $V \setminus C$. Therefore, F_2 cannot visit vertices outside of C since it cannot unblock x'. This is a contradiction to the fact that F_2 visits all the vertices of C.

After the execution of the traversal F_2 , we can either return a 1-vertex-out component of u with at most 2Δ edges or decide that all 1-vertex-out components of u contain more than Δ edges, as shown in Lemma 3.4. The pseudocode of our algorithm is illustrated in Procedure 1VertexOut. The following lemma summarizes the result of this section.

LEMMA 4.5. Procedure 1VertexOut computes in $O(\Delta)$ time a 1-vertex-out component of a vertex u containing at most 2Δ edges or decides that there is no 1-vertex-out component of u containing at most Δ edges.

4.2 Computing the 2-vertex-connected subgraphs. In this section we present an $O(m\sqrt{m})$ time algorithm for computing the 2-vertex-connected subgraphs of a directed graph. We begin with a simple algorithm and then show how we can improve its running time. Recall that the 2-vertex-connected subgraphs of a graph are subgraphs that do not contain any strong articulation points, that is, they cannot get disconnected by the deletion of any single vertex. In contrast to 2-edge-connected subgraphs, the 2-vertex-connected subgraph do not define a partition of the vertices of the input graph.

More specifically, any two 2-vertex-connected subgraphs might share up to one common vertex. This introduces an additional challenge since the existence of a strong articulation point x that disconnects a vertex set S guarantees that no vertex of S appears in the same 2-vertex-connected subgraph as a vertex of $V \setminus (S \cup x)$ but does not provide information on whether x itself appears in a 2-vertex-connected subgraph with vertices from S or $V \setminus (S \cup x)$.

A simple algorithm for computing the 2-vertex-connected subgraphs of a directed graph works as follows. Assume the input graph is strongly connected (or consider each SCC separately). We repeatedly find a strong articulation point x that disconnects the graph into two sets of vertices S and $V \setminus (S \cup x)$, i.e., there is no pair of vertices u and v that are strongly connected in $G \setminus x$ such that $u \in S$ and $v \in V \setminus (S \cup x)$. We recursively execute the same algorithm on the strongly connected components of the subgraphs $G[S \cup x]$ and $G[V \setminus S]$ that contain at least three vertices. If a recursive call fails to identify a strong articulation point in a strongly connected subgraph, then it reports the subgraph as 2-vertex-connected. The correctness and the running time of this simple algorithm can easily be verified along the following lines. First, since at each recursive call we identify a strong articulation point that separates two (non-empty) sets of vertices, we know that the 2-vertex-connected subgraphs of these two sets are disjoint apart from possibly the articulation point itself. Moreover, we restrict the recursive calls to the strongly connected components of the resulting subgraphs since every 2-vertex-connected subgraph is also strongly connected. Therefore, all the 2-vertex-connected subgraphs are preserved at each recursive call. The algorithm reports a 2-vertex-connected subgraph once it recurses on a subgraph that does not contain a strong articulation point, which is correct by definition. Second, we bound the running time. The maximum recursion depth is O(n) since every recursive call is executed on a graph that contains at least one vertex less than the parent call. Although at each recursive call the strong articulation point is included in both sets that it separates, the set of edges is partitioned between the two subgraphs. Therefore, at each level of recursion the total number of edges in all instances is at most m, and the total time to compute a strong articulation point and the strongly connected components at the end of each recursive call is O(m), which leads to an overall running time of O(mn).

The high-level idea of our algorithm for computing the 2-vertex-connected subgraphs is similar to the algorithm of Section 3.2 for computing the 2-edge-connected subgraphs. We additionally define the following operation to construct the subgraphs on which the algorithm recurses. Let G be a digraph, x a vertex, and N a subset of neighbors of x. The operation split(x, N) is executed as follows. First, we create an additional vertex x' in G, that serves as a copy of x. Second, for every edge (x, y) with $y \in N$ we remove (x, y) from G and add the edge (x', y). Analogously, for every edge (y, x) with $y \in N$ we remove (y, x) from G and add the edge (y, x'). This operation can be implemented to take time proportional to the number of neighbors of the vertices in N by traversing their edges and change every edge that is incident to x to be incident to x'.

Lemma 4.6. A split operation preserves the number of edges in the graph. The maximum number of auxiliary vertices after any sequence of split operations is 2m - n.

Proof. By definition, no edges are added or deleted when performing the *split* operation. Since every edge has two endpoints, in the worst case each vertex is adjacent to only one edge. Notice that the original n vertices always exist in the graph. Therefore, the total number of auxiliary vertices cannot exceed 2m - n.

LEMMA 4.7. Let G be a directed graph, x a strong articulation point, and let the sets N_1 , N_2 be a partition of the vertices adjacent to x such that all paths from vertices in N_1 to vertices in N_2 go

through x. There is a one-to-one correspondence between the 2-vertex-connected subgraphs in G and in the graph resulting from G through the execution of either $split(x, N_1)$ or $split(x, N_2)$.

Proof. W.l.o.g., we assume that the *split* operation is $split(x, N_1)$. Let C be a 2-vertex-connected subgraph before the execution of the split operation. If the split operation is not executed on a vertex of C, then C remains a 2-vertex-connected subgraph by the definition of x. Now assume that the split operation is executed on a vertex $x \in C$. Then all neighbors of x that are in C are strongly connected in $C \setminus x$, and therefore they are either all included in $C \setminus x$ or none of them is. Thus, all the edges between the vertices of C are preserved.

Now we prove the opposite direction. Let C be a 2-vertex-connected subgraph after the execution of the split operation. Then, either all edges between the vertices of C existed before the split operation, or there is an auxiliary vertex $x' \in C$ such that all edges between vertices of $C \setminus x'$ existed before the operation and all edges between vertices of $C \setminus x'$ and x' were between $C \setminus x'$ and a vertex x before the split operation (where x is the vertex on which the split operation was executed). That is, no additional paths among the vertices of C were introduced through the split operation and thus in both cases C was a 2-vertex-connected subgraph before the split operation. \Box

We are ready to describe our algorithm for computing the 2-vertex-connected subgraphs of a directed graph G. We build on the simple recursive algorithm that is described at the beginning of this section. To distinguish the input graph from the graphs in the recursive calls, we refer to the original input graph as $G_0 = (V_0, E_0)$. We use the terms small components and large components to refer to subgraphs that contains at most and more than $\sqrt{m_0}$ edges, respectively, where $m_0 = |E_0|$. (We allow small components to contain up to $2\sqrt{m_0}$ edges.) Our algorithm begins by identifying all the small 1-vertex-out and 1vertex-in components of any vertex in G_0 , using the algorithm from Section 4.1. Throughout the algorithm we maintain a list L of the vertices for which we then start a search for a small 1-vertex-out or 1-vertex-in component. We show that it is sufficient to search from the vertices that are inserted into L throughout the algorithm in order to find all the small 1-vertex-out and 1-vertex-in components of all the vertices in the graph. In the initial call to the algorithm we set $L=V_0$ (i.e., this is not done for every recursive call). At each recursive call the algorithm first tests whether the given strongly connected graph contains less than 3 vertices; in this case the empty set is returned as every 2-vertex-connected subgraph has to contain at least 3 vertices. Then it is tested whether the given strongly connected graph is also 2-vertex-connected, which is the case (by definition) if it does not contain a strong articulation point; in this case the algorithm outputs the graph as a 2-vertex-connected subgraph. Then, while L is not empty, we extract a vertex u from L and search for a small 1-vertex-out or a small 1-vertex-in component of u (containing at most $2\sqrt{m}$ edges).

After identifying a 1-vertex-out component 1VOut(u) whose outgoing edges originate from a common vertex x, the algorithm executes split(x, N), where N are the neighbors of x in 1VOut(u). Furthermore, for every edge e = (w, z) incident to 1VOut(u) that is not adjacent to x (i.e., the incoming edges of 1VOut(u)), we insert both w and z into L and remove e from the graph. We treat every identified small 1-vertex-in component in an analogous way.

If, on the other hand, the graph is not 2-vertex-connected but we cannot find a new small 1-vertexout or 1-vertex-in component, we conclude that either there are at least two large sets of vertices that are in different strongly connected components or for every strong articulation point there exist two large sets of vertices that get disconnected by the removal of the strong articulation point. To exploit that, we perform the following steps in the final phase of each recursive call: First we compute the strongly connected components C_1, C_2, \ldots, C_c of the graph that results from the *split* operations. Each of the SCCs that does not contain a strong articulation point and contains at least 3 vertices is 2-vertex-connected and added to the set of 2-vertex-connected subgraphs. For each of the other SCCs C_i we first execute split $(v, N_{C'})$ on some strong articulation point v, where $N_{C'}$ are the neighbors of v that are contained in a singe arbitrary strongly connected component C' in $G[C_i] \setminus v$, and then recursively call the algorithm on each strongly connected component of the resulting graph. Before each recursive call we initialize the lists L to contain the vertices that lost an edge during the last phase of the parent recursive call. After initially adding all vertices to the list of the initial call, we only add the endpoints of deleted edges to the lists, thus the total number of searches for small 1-vertex-out and 1-vertex-in components is bounded by O(m+n). Algorithm 2 contains the pseudocode of our algorithm. This formulation of the algorithm has the advantage that we can bound the size of the subgraphs passed to the recursive calls: either the subgraph is a small 1-vertex-out or 1-vertex-in component and thus contains at most $2\sqrt{m}$ edges or two large sets are separated and therefore the number of edges for each subgraph at the subsequent level of recursion is reduced by at least \sqrt{m} , which can happen at most \sqrt{m} times.

Similarly to Algorithm 1 from Section 3.2, we now show the key property that allows us to either find small sets that can be separated by a single vertex deletion or conclude that there are at least two large vertex sets that are either not strongly connected to each other or become disconnected by the deletion of a single vertex. Every new 1-vertex-out component that appears in the graph throughout the algorithm must contain a vertex that has lost all its outgoing edges that led to vertices not in the component as otherwise the component would have been a 1-vertex-out component before. Note that this vertex that has lost outgoing edges cannot be equal to the only vertex that siil has outgoing edges from the 1-vertex-out component. Analogously, every new 1-vertex-in component that appears must have lost an incoming edge to a vertex other than the separating vertex of the 1-vertex-in component. Therefore, we use the list L to keep track of the vertices that have lost an edge and for each such vertex u we search for new small 1-vertex-out or 1-vertex-in components of u. If no such small components exist in a set of vertices C, then we know that either (i) C is a 2-vertex-connected subgraph or (ii) we are guaranteed that either two large sets of vertices are in separate strongly connected components of the graph, or that every strong articulation point separates two large sets of vertices. This property is summarized in the following lemma.

LEMMA 4.8. Let C be a set of vertices in G. Each 1-vertex-out component (of some vertex $u \in C$) in G[C] for which x is the only vertex that has outgoing edges to $V \setminus C$ and that is not a 1-vertex-out component in G must contain an endpoint z of an edge incident to G[C], such that $z \neq x$. Moreover, if there is no 1-vertex-out or 1-vertex-in component containing at most Δ edges for any vertex $u \in C$ in G[C], then one of the following holds.

- (a) G[C] is a 2-vertex-connected subgraph.
- (b) There are two sets $A, B \subset C$ with $|E(G[A])|, |E(G[B])| > \Delta$ that are disjoint strongly connected components.
- (c) For every strong articulation point x, there are two sets $A, B \subset C$ with $|E(G[A])|, |E(G[B])| > \Delta$ that are separated in $G[C] \setminus x$.

Proof. We first show that every 1-vertex-out component 1VOut(u) of some vertex $u \in C$ for which x is the only vertex that has outgoing edges to $V \setminus C$ and that is no 1-vertex-out component in G must contain a vertex $w \in 1VOut(u) \setminus x$ such that there is an edge $(w, y) \in G$ with $y \notin C$. Assume, by contradiction, that 1VOut(u) exists but there is no such edge (w, y) in G with $w \in 1VOut(u) \setminus x$ and

 $y \notin C$. In this case, the very same component 1VOut(u) is a 1-vertex-out component of u in G, since x is the only vertex having outgoing edges to $V \setminus C$. The same argument on the reverse graph shows that every 1-vertex-in component (of some vertex $u \in C$) in G[C] must contain an endpoint of an edge incident to G[C].

Now we turn to the second part of the lemma. If G[C] is strongly connected and does not contain an articulation point, then G[C] is 2-vertex-connected, i.e., case (a) holds. If G[C] is not strongly connected, then it contains (at least) two disjoint sets $A, B \subset C$ such that both G[A] and G[B] are strongly connected components of G[C] and G[A] has no outgoing edge in G[C] (i.e., G[A] is a sink in the DAG of SCCs of G[C]) and G[B] has no incoming edge in G[C] (i.e., G[B] is a source in the DAG of SCCs of G[C]). That is, in G[C] we have that G[A] is or contains a 1-vertex-out component of some $u \in C$ and G[B] is or contains a 1-vertex-in component of some $u' \in C$. Both can have the same property in G or contains (resp. be) new such components in G[C] compared to G. In any case it contradicts the assumptions if one of them has at most Δ edges and otherwise case (b) holds. If G[C] is strongly connected and contains an articulation point v^* , an analogous argument can be made for two disjoint sets $A, B \subset C$ by considering the DAG of SCCs of $G[C] \setminus v^*$. In this case v^* is the only vertex with incoming edges of B and the only vertex with outgoing edges of A in G[C]. Thus in this case (c) is satisfied if the assumptions of the lemma hold.

Lemma 4.9. Algorithm 2VCS is correct.

Proof. First note that by assumption the initial call to the algorithm is on a strongly connected graph and that recursive calls are only made on strongly connected subgraphs. Thus whenever Algorithm 2VCS reports a 2-vertex-connected subgraph, then this is a 2-vertex-connected subgraph, since it is strongly connected, does not have any strong articulation points, and contains at least 3 vertices. It suffices to show that 2VCS reports all the maximal 2-vertex-connected subgraphs. Notice that this also implies that the reported 2-vertex-connected subgraphs are maximal. Let C be a maximal 2-vertex-connected subgraph. We show that C does not get disconnected by the algorithm, since this will ensure that the algorithm eventually will recurse on C and report it as a 2-vertex-connected subgraph. Since there is no vertex whose deletion separates any pair of vertices in C, any search for either a 1-vertex-out or a 1-vertex-in component (of some vertex u), either returns a superset of C, or it fails to identify such a set containing a subset of the vertices of C. Furthermore, note that any deletion of an edge that does not have both endpoints in C does not affect the fact that C is 2-vertex-connected. That is, unless an edge with both endpoints in C is deleted, no strong articulation points appear in C. Thus, it is left to show that no edge (x,y) such that $x,y \in C$ is ever deleted throughout the algorithm. The edges that are deleted are either edges between strongly connected components, or between two sets of vertices A, B that get disconnected by a strong articulation point, or edges incident to a 1-vertex-out or a 1-vertex-in component found during the course of the algorithm. Since C is always fully included in such a component, no edge of C is deleted. Finally, notice that in each recursive call, unless the graph that is passed to the recursion is 2-vertex-connected, at least one strong articulation point that separates at least one pair of vertices is computed and the algorithm recurses on each strongly connected component (possibly containing a copy of the strong articulation point) after its removal. Thus, the algorithm makes progress in each iteration and at some point there will be a recursive call for each strongly connected subgraph that does not contain strong articulation points, including C. \square

Algorithm 2: 2VCS(G, L)**Input:** A strongly connected digraph G = (V, E) and a list of vertices L (initially L = V) **Output:** The 2-vertex-connected subgraphs of G 1 Let m_0 be the number of edges of the initial graph 2 if $|V| \leq 2$ then return \emptyset // removing degenerate subgraphs 3 if G has no strong articulation point then **return** $\{G\}$ as 2-vertex-connected subgraph **5 while** $L \neq \emptyset$ and G has more than $2\sqrt{m_0}$ edges do Extract a vertex u from L6 $S \leftarrow 1 \text{VertexOut}(G, u, \sqrt{m_0})$ 7 $S^R \leftarrow 1 \text{VertexOut}(G^R, u, \sqrt{m_0})$ 8 Pick non-empty set of S and S^R if it exists 9 Let x be the common vertex in S resp. S^R of all outgoing resp. incoming edges (if it exists) 10 and let N be the neighbors of x inside the set Execute split(x, N) (if x exists) 11 Delete all edges incident to the selected set that are not adjacent to x and add their 12endpoints to L13 Compute strongly connected components C_1, \ldots, C_c of G14 $U \leftarrow \emptyset$ 15 foreach $C_i, 1 \leq i \leq c$ do if C_i contains a strong articulation point v then 16 execute $split(v, N_{C'})$, where $N_{C'}$ are the edges between v and the vertices of an arbitrary **17** strongly connected component C' of $C_i \setminus v$. for each SCC C of C_i do 18 Initialize L' with the vertices of C that are endpoints of newly deleted edges 19 $U \leftarrow U \cup 2VCS(C, L')$ 20 else $\mathbf{21}$ if $|V(C_i)| \geq 3$ then // C_i is 2-vertex-connected 22 $U \leftarrow U \cup \{C_i\}$ 23

Proof. Let $G_0 = (V_0, E_0)$ be the input graph for the initial call to the algorithm. Let $n_0 = |V_0|$ and $m_0 = |E_0|$. First, notice that each time we search for a 1-vertex-out (or a 1-vertex-in component by searching on the reverse graph), we are searching either for a component where all outgoing edges have a common source or for a component with no outgoing edges; in both cases we search for a component with at most $2\sqrt{m_0}$ edges. We can identify if such components exist in time $O(\sqrt{m_0})$ by using the algorithm of Section 4.1. We start a search from every vertex that is added to the list L in some recursive call. Notice that initially we add all vertices to L, and throughout the course of the algorithm we insert the two endpoints of every deleted edge into the corresponding list L. By Lemma 4.6 the number of edges does not increase by the split operations. Therefore, the total time spent on these calls is $O((m_0 + n_0)\sqrt{m_0}) = O(m_0\sqrt{m_0})$. For every 1-vertex-out or 1-vertex-in component S (with at most $2\sqrt{m_0}$ edges) that is discovered throughout the algorithm, the component has either no outgoing (resp., incoming) edges or we execute the split operation on the only vertex x that has outgoing (resp.,

24 return U

incoming) edges. We can execute the operation split in time proportional to the edges incident to the neighbors of x in S, and we can charge this time, as well as the time for identifying the edges to delete, to the process of identifying the set S (that covers for the edges in G'[S]), and to the edges deleted from the graph.

Let G' = (V', E') be the graph passed to a recursive call. The algorithm spends O(|E'|) time to test whether there are strong articulation points in the graph (line 3), and additionally O(|E'|) time to compute the strong articulation points, to execute the *split* operation on an arbitrary strong articulation point in each strongly connected component, and to recompute strongly connected components (lines 13–17). Since the recursive calls are executed on subgraphs whose sets of edges are disjoint (since the *split* operator simply partitions the edges incident to the vertex on which the operation is executed, and moreover, all the strongly connected components are disjoint), it follows that the total time spend for the above procedures in all instances at each recursion depth is $O(m_0)$. Notice that the number of vertices does not exceed $2m_0$, by Lemma 4.6, after any sequence of split operations, and thus this time bound holds for every recursion depth.

Let G' be the graph at some recursive call. We show that the graph passed to each subsequent recursive call has at most $\max\{|E(G')| - \sqrt{m_0}, 2\sqrt{m_0}\}$ edges, or G' is a 2-vertex-connected subgraph and thus the recursion stops. This implies a recursion depth of $O(\sqrt{m_0})$ as follows. If a graph passed to a recursive call has at most $2\sqrt{m_0}$ edges, it means that also the number of vertices is at most $2\sqrt{m_0}$. Therefore, even if the algorithm simply identifies a strong articulation point, executes the *split* operation, and recurses on each strongly connected component of the resulting graph, the total recursion depth is at most $O(\sqrt{m_0})$. On the other hand, there can be at most $\sqrt{m_0}$ cases where the graph that is passed in a recursive call has $\sqrt{m_0}$ fewer edges that G'. Overall, this proves that the recursion depth is bounded by $O(\sqrt{m_0})$.

It remains to show the claimed bound on the size of the graph passed to a recursive call in line 20. By Lemma 4.7 every 1-vertex-out (resp., 1-vertex-in) component will be in a separate strongly connected component with at most $2\sqrt{m_0}$ edges. Now, let C be the set of vertices that were not included in any 1-vertex-out or any 1-vertex-in component. This set did not contain any 1-vertex-out or any 1-vertex-in component S with less than $\sqrt{m_0}$ edges in G' since otherwise such a set S would contain a vertex x that lost an edge (and thus was added to L) and the algorithm would search for a 1-vertex-out or a 1-vertex-in component of x, identifying S in this way. This means, by Lemma 4.8, that C either is a 2-vertex-connected subgraph, or there are two disjoint sets in $A, B \subset C$, $|E'(A)|, |E'(B)| > \sqrt{m_0}$ that are either not strongly connected to each other or separated by at most one strong articulation point in G'[C]. If the later holds, A and B will be separated in line 13, and every graph passed to a subsequent recursive call has at most max $\{|E'(G')| - \sqrt{m_0}, 2\sqrt{m_0}\}$ edges. \square

The following theorem summarizes the result of this section.

Theorem 4.1. The maximal 2-vertex-connected subgraphs of a digraph can be computed in $O(m^{3/2})$ time.

5 k-edge-connected subgraphs

In this section we extend our algorithm for 2-edge-connected subgraphs to k-edge-connected subgraphs for k > 2. At the end of the section we discuss how to obtain better running time bounds for undirected graphs.

Let G = (V, E) be a digraph and $u \in V$ be a vertex. Let for this section k' = k - 1. We define a k'-edge-out component S of u to be a subgraph of G with $\widetilde{k} \leq k'$ outgoing edges such that S contains u and there is no component $S' \subset S$ containing \widetilde{k} or fewer outgoing edges. We denote a k'-edge-out

component of u by k'EOut(u). Analogously, we define a k'-edge-in component S of u to be a subgraph of G with $\widetilde{k} \leq k'$ incoming edges, such that S contains u and there is no component $S' \subset S$ containing \widetilde{k} or fewer incoming edges.

We would like to extend the algorithm for maximal 2-edge-connected subgraphs to maximal k-edgeconnected subgraphs for $k \geq 2$. For this we need to identify k'-edge-out components (and therefore k'-edge-in components by using the reverse graph) for a given vertex u in time proportional to their size, potentially with an additional factor depending on k. The first idea would be to simply start k = k' + 1 depth-first searches from u. Assume for now that the first k' searches $F_1, \ldots, F_{k'}$ each visited $\ell(k',\Delta)$ edges, for some function $\ell(k',\Delta)$ of order $O(k'\Delta)$ specified later, and let $T_1,\ldots,T_{k'}$ denote the DFS trees generated by the searches, respectively. Further assume that there exists a k'-edge-out component k'EOut(u) that contains at most Δ edges. Suppose we have for each $1 \leq i \leq k'$ a path P_i in T_i that starts at u and ends outside of k'EOut(u). The idea of the k=k'+1 searches from u is that the first k' searches should each reduce the number of outgoing edges of k'EOut(u) by one by reversing the direction of the edges on P_i after the i-th such search, and the (k'+1)-st search should explore exactly the edges of k'EOut(u) (and does not have to be a DFS). However, since there are multiple edges leaving k'EOut(u), each DFS F_i might enter and leave k'EOut(u) multiple times and thus we cannot determine such paths P_i that end outside of k'EOut(u) so easily. Note that if we reverse a path that ends inside of k'EOut(u), then the number of outgoing edges of k'EOut(u) remains the same and no progress was made by this search. We first show that under the assumption that we have for all $1 \le i \le k'$ a path P_i from u that ends outside of k'EOut(u), the strategy of reversing the path P_i before conducting the search F_{i+1} works, i.e., we extend Section 3.1 to this case. We then construct O(k') paths for each of the searches of which one of them is guaranteed to end outside of the k'-edge-out component of u (if such a component with at most Δ edges exists). Based on this, we provide an algorithm for computing the maximal k-edge-connected subgraphs with exponential dependence on k = k' + 1.

5.1 (k-1)-edge-out components Recall k' = k - 1. Given an integer Δ , we assume that the starting vertex u can reach at least $\ell(k', \Delta)$ edges for some $\ell(k', \Delta) \in O(k'\Delta)$ with $\ell(k', \Delta) < m$. Notice that if u cannot visit $\ell(k', \Delta)$ edges, then the reachable subgraph from u defines a k'-edge-out component of u containing less than $\ell(k', \Delta)$ edges. Let F_1 denote a depth-first search for up to $\ell(k', \Delta)$ edges started from u in $G_1 = G$. Let P_1 be a path from u in F_1 . Let G_{i+1} for $1 \le i \le k'$ be defined as G_i with the edges of P_i reversed, where F_i is the depth-first search conducted on G_i (from u, for up to $\ell(k', \Delta)$ edges) and P_i is a path from u in F_i . We can interpret the graph G_{i+1} as residual graph after sending one unit of flow along the path P_i in the graph G_i . The following lemma supports this interpretation by showing that if \widetilde{k} paths end in a vertex set T with $u \notin T$, then the number of edges from $S = V \setminus T$ to T in G_{i+1} is reduced by \widetilde{k} compared to G.

LEMMA 5.1. Let S be a set of vertices containing u. Let $T = V \setminus S$ be a set of vertices that contains $0 \le \widetilde{k} \le i$ of the endpoints of the paths P_1, \ldots, P_i for some $i \le k'$. Then there are \widetilde{k} fewer edges from S to T in G_{i+1} than in G.

Proof. Consider the (multi-)graph G' that is constructed from G by contracting the vertices of S to a single vertex s and the vertices of T to a single vertex t. Applying the contraction to the paths P_j , we obtain for each $1 \leq j \leq i$ a set of edges E'_j between s and t that represent the contracted path, where we keep the direction of the edges as in P_j . Let $G'_1 = G'$ and let G'_{j+1} for $1 \leq j \leq i$ be the (multi-)graph obtained from G'_j by reversing the edges of E'_j , i.e., the graph G'_j can be obtained from G_j by contracting S and T, respectively. By definition, the graphs G'_j differ from G' only in the direction of the edges between s and t. Further we have that if P_j ends at a vertex of T (case 1), then the number of edges from s to t in E'_j is one more than the number of edges

from t to s; in contrast, if P_j ends at a vertex of S (case 2), there are as many edges from s to t as from t to s in E'_j . In case 1 the number of edges from s to t in G'_{j+1} is one lower than in G'_j , while in case 2 the number of edges from s to t is the same in G'_j and G'_{j+1} . Let $0 \leq \tilde{k} \leq i$ be the number of paths of $\{P_1, \ldots, P_i\}$ that end in T. We have that the number of edges from s to t in G'_{i+1} , and therefore from S to T in G_{i+1} , is equal to the number of paths from S to T in G minus \tilde{k} . \square

The following lemma shows that for the first of the k'+1 searches that cannot reach $\ell(k', \Delta)$ edges from u, we have that the subgraph traversed by this search induces a k'-edge-out component of u.

LEMMA 5.2. Let the i-th search of the searches $F_1, \ldots, F_{k'+1}$ be the first one that visits less than $\ell(k', \Delta)$ edges if such a search with i > 1 exists. Then there exists an (i-1)-edge-out component of u with less than $\ell(k', \Delta)$ edges and the subgraph induced by the vertices traversed by F_i defines this component.

Proof. Let S be the set of vertices traversed by F_i and let T be $V \setminus S$. Note that T contains at least one vertex that was traversed by some search F_1, \ldots, F_{i-1} but not by F_i because F_i could traverse $\ell(k', \Delta)$ edges if it could traverse the same vertices as the other searches. Notice further that S contains u. By the definition of S and the assumption that F_i traverses less than $\ell(k', \Delta)$ edges, we have that there are no edges from S to T in G_i . Thus by Lemma 5.1 the number of edges from S to S to S in S is equal to the number S of paths of S, and S is equal to the number S of paths of S, and S is equal to the number S is edges in S.

To show that S is an (i-1)-edge-out component of u, it remains to show that S does not contain a proper subset \hat{S} that contains u and has \tilde{k} or less outgoing edges. Assume by contradiction such a set \hat{S} exists and let $\hat{T} = V \setminus \hat{S}$. By $\hat{T} \supseteq T$ at least \tilde{k} paths of the paths P_1, \ldots, P_{i-1} end in \hat{T} . Thus by Lemma 5.1 there are at least \tilde{k} edges from \hat{S} to \hat{T} in G. If there were exactly \tilde{k} edges from \hat{S} to \hat{T} in G, then there would be no edges from \hat{S} to \hat{T} in G_i . Recall that the search F_i is conducted in the graph G_i . Thus this is a contradiction to S being the set of vertices explored by F_i from u.

The following lemma shows that identifying a k'-edge-out component of u reduces to identifying paths $P_1, \ldots, P_{k'}$ that all end outside of the component.

LEMMA 5.3. Let k'EOut(u) be a k'-edge-out component of u with $|k'EOut(u)| \leq \Delta$. Assume all paths $P_1, \ldots, P_{k'}$ end outside of k'EOut(u) and all searches $F_1, \ldots, F_{k'}$ have visited $\ell(k', \Delta)$ edges. Then the subgraph traversed by $F_{k'+1}$ is k'EOut(u).

Proof. Let S denote the set of vertices of k'EOut(u) and let $T = V \setminus S$. By definition S has at most k' outgoing edges in G and no proper subset of S has k' or less outgoing edges. By assumption all paths $P_1, \ldots, P_{k'}$ end in T. Thus by Lemma 5.1 there are no edges from S to T in $G_{k'+1}$ and hence $F_{k'+1}$ traverses a subset of S.

It remains to show that $F_{k'+1}$ traverses exactly the vertices of S. Assume by contradiction it traverses a proper subset S' of S. By Lemma 5.2 this implies that the subgraph induced by S' is a k'-edge-out component of u, a contradiction to the minimality of S.

5.1.1 Finding outgoing DFS-tree paths. Assume a k'-edge-out component k'EOut(u) of u with $|k'EOut(u)| \leq \Delta$ exists. We show next how to find a set of O(k) paths \mathcal{P}_1 from u in the DFS tree T_1 such that at least one of the paths ends outside of k'EOut(u). We also specify the number of edges $\ell(k', \Delta)$ for which we conduct each depth-first search in order to ensure this property. As we do not know which

of the paths of \mathcal{P}_1 ends outside, we define O(k) variants of the graph G_2 , one for each of the paths in \mathcal{P}_1 , i.e., the j-th variant of G_2 is equal to $G_1 = G$ with the j-th path of \mathcal{P}_1 reversed. Provided that the DFS F_1 has visited $\ell(k', \Delta)$ edges, we start a second DFS search F_2 on each of the variants of G_2 . In the same manner, we start O(k) third DFS searches for each path in \mathcal{P}_i for each variant of G_2 and so on, that is, we have $O(k^2)$ variants of G_3 and $O(k^k)$ variants of $G_{k'+1}$. Thus instead of k = k' + 1 DFS searches, we perform $O(k^k)$ DFS searches in total in order to ensure that at least in one variant all the paths $P_1, \ldots, P_{k'}$ end outside of k'EOut(u) and thus the (k'+1)-st search in this variant of $G_{k'+1}$ explores k'EOut(u) by Lemma 5.3.

To identify a path from u that ends outside of k'EOut(u), we conduct each DFS from u in chunks of $\Delta+1$ edges. After each chunk we identify one candidate path from u in the DFS tree and make a recursive call that starts the next DFS on the graph with the candidate path reversed. We show that either the candidate path indeed ends outside of k'EOut(u) or the DFS has either traversed one additional edge of the outgoing edges of k'EOut(u) or retracted along one of these outgoing edges. Recall that whenever a DFS traversal retracts from some vertex v to a proper ancestor of v, then it will not visit v or any of its outgoing edges ever again. Thus in particular we have for every outgoing edge of k'EOut(u) that it is explored only once and that the DFS retracts along the edge at most once. Hence both cases together can happen in at most 2k' of the chunks and thus by continuing the DFS for 2k'+1 chunks we are guaranteed to have identified at least one path from u to a vertex outside of k'EOut(u). By this argument we implicitly set $\ell(k', \Delta)$ to $(2k'+1)(\Delta+1)$. The following lemma shows that this strategy indeed either makes progress by using an outgoing edge or identifies a path ending outside. For each chunk the candidate path is given by the path from u to the highest vertex in the DFS tree (i.e., the vertex closest to the root u) that is visited during this chunk.

LEMMA 5.4. Let F denote a (possibly empty) DFS started at u. Assume there are at least $\Delta + 1$ edges reachable from u that where not explored by F. Let F' be the DFS obtained by extending F by $\Delta + 1$ edges. At least one of the following statements is true.

- (a) The nearest common ancestor (NCA) in the DFS tree of all vertices visited by $F' \setminus F$ is not in k'EOut(u).
- (b) The DFS F' explored at least one more outgoing edge of k'EOut(u) than the DFS F.
- (c) The DFS retracted along an outgoing edge of k'EOut(u) while extending F to F'.

Proof. Let s be the vertex where the DFS F stopped and the extension of F starts. Let h be the nearest common ancestor in the DFS tree of all vertices visited during extending F to F'. Note that h was already explored by F. If the DFS retracts to an ancestor of s at some point during the extension, then h is equivalent to the highest vertex the DFS retracts to; otherwise we have h = s.

First consider the case where s is not in k'EOut(u). Then either also h is not in k'EOut(u) and thus (a) is satisfied or the DFS retracts back to some ancestor of s that is inside of k'EOut(u), which satisfies (c).

Assume now $s \in k'EOut(u)$. Since $|k'EOut(u)| \leq \Delta$ and the DFS F' visits $\Delta + 1$ additional edges, it follows that F' visits at least one additional edge that is not in k'EOut(u). To reach edges outside of k'EOut(u) from s, the DFS F' can either (1) use one of the outgoing edges of k'EOut(u) that were not traversed by F or (2) complete the DFS traversal from s and retract to some ancestor of s in the current DFS tree. In case (1) the statement (b) holds. In case (2) we distinguish three sub-cases. Recall that h is the highest vertex the DFS retracts to and the nearest common ancestor of all vertices visited during the extension of the DFS.

- (i) If h is in k'EOut(u) but some vertex on the tree path from h to s is not in k'EOut(u), then (c) is satisfied.
- (ii) If all vertices on the path from h to s in the DFS tree are in k'EOut(u), then the DFS has to use at least one outgoing edge of k'EOut(u) to visit edges outside of k'EOut(u) and thus (b) holds.
- (iii) If neither (i) nor (ii) holds, then h is not in k'EOut(u) and hence (a) holds.

In Procedure kEOut we combine the results of this section to an algorithm that returns a k'-edge-out component of u whenever one with at most Δ edges exists, and might return the empty set if no such component exists. The soundness of the algorithm is given by Lemma 5.2, i.e., whenever one of the subsequent k'+1 DFS searches can visit less than $\ell(k',\Delta)=(2k'+1)(\Delta+1)$ edges, then the set of vertices visited by this search induces a k'-edge-out component of u. The completeness of the algorithm is as follows: by Lemma 5.3 the (k'+1)-st search identifies k'-edge-out component of u with at most Δ edges given that the k' paths $P_1, \ldots, P_{k'}$ all end outside of the component (where a path P_i is identified in the graph constructed from G_{i-1} by reversing the edges of P_{i-1}); and by Lemma 5.4, and the observation that the cases (b) and (c) of Lemma 5.4 can each happen at most k' times if a k'-edge-out component of u with at most Δ edges exists, this property is satisfied by the paths constructed in at least one of the sequences of depth-first searches initialized by the recursive calls of the algorithm. One DFS search takes time $(2k'+1)(\Delta+1)$ and makes at most 2k'+1 recursive calls. In Procedure kEOut the recursion depth is explicitly bounded by k=k'+1 (line 6). Thus we have shown the following lemma.

LEMMA 5.5. We can compute in $O((2k)^{k+1} \cdot \Delta)$ time a (k-1)-edge-out component of a vertex u that contains less than $(2k-1)(\Delta+1)$ edges, or otherwise we conclude that there is no (k-1)-edge-out component of u containing at most Δ edges.

5.2 Computing the k-edge-connected subgraphs of a digraph. Our algorithm to compute the maximal k-edge connected subgraphs of a given digraph follows the same structure as the algorithm given in Section 3.2 for k = 2. The main difference lies in the different subroutine we use to determine (k-1)-edge-out components with at most Δ edges. For k > 2 we use Procedure kEOut, which has a running time exponential in k and linear in Δ . The increased time to determine an edge cut of at most k-1 edges (i.e., lines 2 and 12 take time $O(m \log n)$ [7]) leads to an additional factor of $\log n$ in the running time. Thus we obtain for any constant k > 2 an $O(m^{3/2} \log n)$ time algorithm.

The basic algorithm for maximal k-edge connected subgraphs finds for each strongly connected component of the input graph a (directed) cut of at most k-1 edges if one exists, removes the cut edges from the graph, and recurses on each strongly connected component of the remaining graph. A cut of at most k-1 edges can be found with Gabow's algorithm [7] in $O(km\log n)$ time and whenever a cut is found at least two vertices seize to be strongly connected, thus this algorithm takes time $O(kmn\log n)$ time.

To improve upon the basic algorithm for constant k and sparse graphs with $\sqrt{m} < n$, we search for (k-1)-edge-out components with at most $\Delta = \sqrt{m}$ edges from all vertices that have lost adjacent edges in a prior iteration of the algorithm, using Procedure kEOut. Analogously, we search for (k-1)-edge-in components with at most \sqrt{m} edges from these vertices by applying Procedure kEOut on the reverse graph. Note that whenever a cut with at most k-1 edges exists, then one side of the cut contains a (k-1)-edge-out component and the other side of the cut contains a (k-1)-edge-in component, and vice versa. Further, if a subgraph is a (k-1)-edge-out component in a recursive call of the algorithm

Procedure $kEOut(G, u, \Delta)$

```
Input: Digraph G = (V, E), a vertex u, and an integer \Delta
```

Output: For k' = k - 1 either a vertex set inducing a k'-edge-out component of u with less than $(2k' + 1)(\Delta + 1)$ edges or the empty set; if the empty set is returned, no k'-edge-out component of u with less than Δ edges exists

```
1 Initialize DFS F starting from u
 2 for 2k' + 1 times do
       Extend the DFS F for at most \Delta + 1 edges
 3
       if at most \Delta edges added to F then
 4
          return the set of vertices explored by F
 5
       else if the recursion depth is at most k' then
 6
           Let h be the NCA in the DFS tree of the vertices visited during the extension
 7
           Let G' be G with the DFS tree path from u to h reversed
 8
           S \leftarrow \text{kEOut}(G', u)
 9
           if S \neq \emptyset then
10
              return S
11
12 return \emptyset
```

but was not before the recursive call, then this subgraph must have had at least one additional outgoing edge before the recursive call. Thus by searching from vertices that have lost adjacent edges, we ensure to find all (k-1)-edge-out components with at most \sqrt{m} edges in each recursive call. Hence if no (k-1)-edge-out or (k-1)-edge-in component is found by these searches, we know that every cut of at most k-1 edges divides the graph into two subgraphs with more than \sqrt{m} edges each. In this case we execute one iteration of the basic algorithm. Since subgraphs of more than \sqrt{m} edges can be removed from G at most \sqrt{m} times, we can bound the recursion depth of the algorithm by \sqrt{m} as in Section 3. The following lemmata formalize this discussion and are straightforward generalizations of the k=2 case.

LEMMA 5.6. Let C be a set of vertices in G. Every (k-1)-edge-out or (k-1)-edge-in component (of some vertex $u \in C$) in G[C] that is not such a component in G must contain an endpoint of an edge incident to G[C]. Moreover, if there is no (k-1)-edge-out or (k-1)-edge-in component containing at most Δ edges for any vertex $u \in C$ in G[C], then one of the following holds:

- (a) G[C] is a k-edge-connected subgraph of G.
- (b) There are two sets $A, B \subset C$ with $|E(G[A])|, |E(G[B])| > \Delta$ such that A and B are in different strongly connected components of G[C].
- (c) For each cut of size at most k-1 in G[C] there are two sets $A, B \subset C$ with $|E(G[A])|, |E(G[B])| > \Delta$ that get disconnected by the deletion of the cut edges.

Proof. Let k' = k - 1. We first show that every k'-edge-out component k'EOut(u) of some vertex $u \in C$ that is not a k'-edge-out component in G must contain an edge (x, y) with $x \in k'EOut(u)$ and $y \notin C$. Assume, by contradiction, that k'EOut(u) exists but there is no such edge (x, y) in G with

Algorithm 3: kECS(G, L)

```
Input: Strongly connected digraph G = (V, E) and a list of vertices L (initially L = V)
   Output: The k-edge-connected subgraphs of G
 1 Let m_0 be the number of edges in the initial graph
 2 if G has no cut of less than k edges then
       return \{G\} as k-edge connected subgraph
   while L \neq \emptyset and G has more than 2k\sqrt{m_0} edges do
       Extract a vertex u from L
 5
       S \leftarrow \text{kEOut}(G, u, \sqrt{m_0})
 6
       S^R \leftarrow \text{kEOut}(G^R, u, \sqrt{m_0})
       If either S or S^R is not empty, remove from G all the incident edges to one non-empty set of
        S and S^R and add their endpoints to L
 9 Compute SCCs C_1, \ldots, C_c of G
10 U \leftarrow \emptyset
11 foreach C_i, 1 \leq i \leq c do
       Remove a (k-1)-cut from G[C_i] (if it exists)
12
       Recompute SCCs and delete the edges between them
13
       foreach SCC C' do
14
           Insert into L' the vertices of C' that are endpoints of newly deleted edges
15
           U \leftarrow U \cup kECS(C', L')
16
17 return U
```

 $x \in k'EOut(u)$ and $y \notin C$. In this case we have that the very same component k'EOut(u) is a k'-edge-out component of u in G. The same argument on the reverse graph shows that every new k'-edge-in component (of some vertex $u \in C$) in G[C] must contain an endpoint of an edge incident to G[C].

Now we turn to the second part of the lemma. If G[C] is strongly connected and does not contain a cut of size at most k', then G[C] is k-edge-connected and thus (a) holds. If G[C] is not strongly connected, then it contains (at least) two disjoint sets $A, B \subset C$ such that both G[A] and G[B] are strongly connected components of G[C] and G[A] has no outgoing edge in G[C] (i.e., G[A] is a sink in the DAG of SCCs of G[C]) and G[B] has no incoming edge in G[C] (i.e., G[B] is a source in the DAG of SCCs of G[C]). That is, in G[C] we have that G[A] contains a k'-edge-out component of some $u \in C$ and G[B] contains a k'-edge-in component of some $u' \in C$. Both G[A] and G[B] may have the same property in G or be new such components in G[C] compared to G. In any case it contradicts the assumptions if one of them has at most Δ edges and otherwise statement (b) holds. If G[C] is strongly connected and contains a cut of with at most k' edges, an analogous argument can be made for two disjoint sets $A, B \subset C$ by considering the DAG of SCCs of G[C] with the cut edges removed. If the number of cut edges is minimal, we have that the cut edges are the only incoming edges of B and the only outgoing edges of A in G[C]. We have that case (c) holds if the assumptions of the lemma are satisfied. \Box

Lemma 5.7. The algorithm kECS is correct.

Proof. Whenever the algorithm kECS reports a k-edge-connected subgraph in line 3, then it is a strongly connected subgraph that does not contain any cut with at most k-1 edges, which is

by definition a k-edge-connected subgraph. Thus it suffices to show that kECS reports all the maximal k-edge-connected subgraphs. Notice that this also implies that the reported k-edge-connected subgraphs are maximal. Let C be a maximal k-edge-connected subgraph. We show that the vertices of C do not get separated by the algorithm, and therefore C is reported eventually as a k-edge-connected subgraph. Since there are k edge-disjoint paths between every pair of vertices in C, any search for either a (k-1)-edge-out or a (k-1)-edge-in component of a vertex u (lines 6-7) either returns a superset of C or fails to identify such a set containing a subset of the vertices of C. Furthermore, notice that any deletion of an edge that does not have both endpoints in C does not affect the fact that C is k-edge-connected. That is, unless an edge with both endpoints in C is deleted, no cut with at most k-1 edges appears in C. Thus, it remains to show that no edge (x,y) such that $x,y\in C$ is ever deleted throughout the algorithm. The edges deleted in line 8 of the algorithm are incident to a (k-1)-edge-out or a (k-1)-edge-in component. Since C is always fully inside or fully outside of such a set, no edge from C is deleted. The edges deleted in line 12 are cuts with at most k-1 edges and the edges deleted in line 13 before the recursive calls are between separate strongly connected components. Since C is k-edge-connected, no edges from C are deleted. Finally, notice that at each level of recursion at least cut with at most k-1 edges of each strongly connected component of the graph is deleted and the algorithm is recursively executed on each resulting strongly connected component. Thus, the recursive calls finally consider all strongly connected subgraphs that do not contain cuts with at most k-1 edges, including C.

LEMMA 5.8. The algorithm kECS runs in $O(m\sqrt{m}\log n)$ time for constant k>2.

Proof. Let $\ell(k,\Delta) = (2k-1)(\Delta+1)$ and $\Delta = \sqrt{m}$, where m is the number of edges in the input graph. First notice that each time we search for a (k-1)-edge-out or a (k-1)-edge-in component, we are searching for a component with k-1 outgoing (resp., incoming) edges containing at most \sqrt{m} edges or with less than k-1 (resp., incoming) edges and less than $\ell(k,\Delta)$ edges. We can identify if such a component containing a given vertex u exists in time $O(\sqrt{m})$ for constant k by using the algorithm of Section 5.1. We initiate such a search from each vertex that appears in the list L of some recursive call of the algorithm. Initially, we place all vertices into the list L. Throughout the algorithm we insert into L only vertices that are endpoints of deleted edges. Therefore, the number of vertices that are added to the lists L throughout the algorithm is O(m). Hence, the total time spent on these searches is $O(m\sqrt{m})$ for constant k.

Consider now the time spend in each recursive call without the searches for (k-1)-edge-out and (k-1)-edge-in components. Let G' be the graph for which the recursive call is made and let $m_{G'} = |E(G')|$. In each recursive call the algorithm spends $O(km_{G'}\log n)$ time to search cuts with at most k-1 edges in G' in lines 2 and 12 and $O(m_{G'})$ to compute SCCs in lines 9 and 13. Since the subgraphs of different recursive calls at the same recursion depth are disjoint, the total time spent at each level of the recursion is O(m). We now bound the recursion depth for constant k with $O(\sqrt{m})$.

We show that the graph passed to each recursive call has at most $\max\{m_{G'} - \sqrt{m}, 2k\sqrt{m}\}$ edges, or it is a k-edge-connected subgraph and thus the recursion stops. This implies a recursion depth of $O(k\sqrt{m})$, i.e., $O(\sqrt{m})$ for constant k, as follows. If the graph passed to a recursive call has at most $2k\sqrt{m}$ edges, then also the number of vertices of this graph is at most $2k\sqrt{m}$. Therefore, even if the algorithm only removes one cut from every strongly connected component in each recursive call, the total recursion depth is at most $O(\sqrt{m})$ for constant k. On the other hand, the number of times that the graph passed to a recursive call has \sqrt{m} fewer edges than G' is at most \sqrt{m} . Overall, this implies that the recursion depth is bounded by $O(\sqrt{m})$.

It remains to show the claimed bound on the size of the graph passed to a recursive call in line 16. For every (k-1)-edge-out or (k-1)-edge-in component with at most $2k\sqrt{m}$ edges that is discovered throughout the algorithm, its incident edges are removed and therefore it will be in a separate strongly connected component with at most $2k\sqrt{m}$ edges. Let C be the set of vertices that were not included in any (k-1)-edge-out or (k-1)-edge-in component. By Lemma 5.6 the subgraph G'[C] either is a k-edge-connected subgraph or there are two sets A and B with $|E(A)|, |E(B)| > \sqrt{m}$ that will be separated in line 12. Thus, every graph passed to the recursive call will have at most $\max\{|E(G')| - \sqrt{m}, 2k\sqrt{m}\}$ edges. The lemma follows.

5.3 k-edge-connected subgraphs for undirected graphs. The problems of computing the k-edge-connected subgraphs of an undirected graph can be reduced to the equivalent problem for directed graphs in a straightforward way. More specifically, for a given undirected graph we construct a directed graph with the same vertex set, and replace every undirected edge with two bidirectional edges. On the resulting digraph the set of vertices of the k-edge-connected subgraphs are equivalent to the set of vertices of the k-edge-connected subgraphs in the original undirected graph.

The complexity of our algorithms is determined by the choice of the parameter Δ in the algorithm that searches for (k-1)-edge-out and the (k-1)-edge-in components of a vertex. The parameter Δ determines both the depth of the recursion, which is $O(m/\Delta)$, and the time we spend searching for small components, which is $O((m+n)\Delta)$ in total.

The second factor that affects the complexity is the time spent identifying a cut at every depth of the recursion. Note that the time spent searching for a cut will dominate the O(m) time it takes to compute the strongly connected components before executing the recursive call. This factor is multiplied by the maximum recursion depth in the time complexity of the algorithm. The digraph on which we executed our algorithm originates from an undirected graph, and we can use this to search for edge cuts of size at most k-1 faster. Thus, the time complexity of our algorithms is $O(t \cdot (m/\Delta) + n\Delta)$, where t is the time required to identify a cut of size at most k-1 in an undirected graph.

The edge cuts of size at most 2 can be identified in linear time [10, 19]. It is easy to verify that the optimal choice of Δ is therefore m/\sqrt{n} for k=3. For constant k, we can compute an edge cut of size at most (k-1) in time $O(m+n\log n)$ [7]. We choose $\Delta=m/\sqrt{n}$ for k-edge-connected subgraphs as well as for 3-edge-connected subgraphs. We obtain the following result.

THEOREM 5.1. The maximal k-edge-connected subgraphs of an undirected graph can be computed in $O((m + n \log n)\sqrt{n})$ time on a undirected graph with m edges and n vertices. For the maximal 3-edge-connected subgraphs, our algorithm runs in $O(m\sqrt{n})$ time.

References

- [1] K. Chatterjee and M. Henzinger. Efficient and dynamic algorithms for alternating Büchi games and maximal end-component decomposition. *Journal of the ACM*, 61(3):15:1–15:40, 2014. Announced at SODA'11 and SODA'12.
- [2] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster Algorithms for Computing Maximal 2-Connected Subgraphs in Sparse Directed Graphs. In *SODA*, pages 1900–1918, 2017.
- [3] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, September 1997. Announced at FOCS'92.

- [4] Y. M. Erusalimskii and G. G. Svetlov. Bijoin points, bibridges, and biblocks of directed graphs. *Cybernetics and Systems Analysis*, 16(1):41–44, 1980.
- [5] S. Even. An algorithm for determining whether the connectivity of a graph is at least k. SIAM Journal on Computing, 4(3):393–396, 1975.
- [6] S. Even and Robert E. Tarjan. Network Flow and Testing Graph Connectivity. SIAM Journal on Computing, 4(4):507–518, 1975.
- [7] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995.
- [8] H. N. Gabow. Using expander graphs to find vertex connectivity. *Journal of the ACM (JACM)*, 53(5):800–844, 2006.
- [9] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.
- [10] Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. SIGACT News, 22(1):57–61, March 1991.
- [11] L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint s-t paths in digraphs. In Automata, Languages and Programming, 37th Int'l. Coll., ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I, pages 738–749, 2010.
- [12] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc.* 42nd Int'l. Coll. on Automata, Languages, and Programming, pages 605–616, 2015.
- [13] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. *ACM Trans. Algorithms*, 13(1):9:1–9:24, 2016. Announced at SODA'15.
- [14] Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.
- [15] Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Efficient algorithms for computing all low st edge connectivities and related problems. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 127–136. Society for Industrial and Applied Mathematics, 2007.
- [16] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 713–724, 2015. Full version available at http://arxiv.org/abs/1412.6466.
- [17] M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. In *Proc. 28th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 1919–1938, 2017.
- [18] M. R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.
- [19] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. SIAM Journal on Computing, 2(3):135–158, 1973.
- [20] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012.
- [21] R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164–172, 2016.
- [22] Arkady Kanevsky and Vijaya Ramachandran. Improved algorithms for graph four-connectivity. *Journal of Computer and System Sciences*, 42(3):288–306, 1991.
- [23] D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000. Announced at STOC'96.
- [24] K. Kawarabayashi and M. Thorup. Deterministic global minimum cut of a simple graph in near-linear time. In Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015, pages 665-674, 2015.
- [25] S. Makino. An algorithm for finding all the k-components of a digraph. *Int'l Journal of Computer Mathematics*, 24(3-4):213–221, 1988.
- [26] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. Algorithmica, 7(5&6):583–596, 1992.
- [27] H. Nagamochi and T. Watanabe. Computing k-edge-connected components of a multigraph. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E76–A(4):513–517, 1993.

- [28] R. E. Tarjan. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2):146–160, 1972.
- [29] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. Acta Informatica, 6(2):171–85, 1976.
- [30] M. Thorup. Fully-dynamic min-cut. Combinatorica, 27(1):91–127, 2007. Announced at STOC'01.