Determining 4-Edge-Connected Components in Linear Time

Wojciech Nadara ✉
Institute of Informatics, University of Warsaw, Poland

Mateusz Radecki ✉
University of Warsaw, Poland

Marcin Smulewicz ✉
Institute of Informatics, University of Warsaw, Poland

Marek Sokołowski ✉
Institute of Informatics, University of Warsaw, Poland

Abstract

In this work, we present the first linear time deterministic algorithm computing the 4-edge-connected components of an undirected graph. First, we show an algorithm listing all 3-edge-cuts in a given 3-edge-connected graph, and then we use the output of this algorithm in order to determine the 4-edge-connected components of the graph.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases graphs, connectivity, cuts

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.71

Related Version *Full Version*: `https://arxiv.org/abs/2105.01699` [33]

Funding *Wojciech Nadara*: Supported by ERC Starting Grant CUTACOMBS (Grant Agreement No 714704).
Marcin Smulewicz: Supported by ERC Starting Grant TOTAL (Grant Agreement No 677651).
Marek Sokołowski: Supported by ERC Starting Grant BOBR (Grant Agreement No 948057).

1 Introduction

The connectivity of graphs has always been one of the fundamental concepts of graph theory. The foremost connectivity notions in undirected graphs are the *k-edge-connectedness* and the *k-vertex-connectedness*. Namely, a graph G is *k-edge-connected* for $k \geq 1$ if it is connected, and it remains connected after removing any set of at most $k-1$ edges. Similarly, G is *k-vertex-connected* if it contains at least $k+1$ vertices, and it remains connected after the removal of any set of at most $k-1$ vertices.

These notions can be generalized to the graphs that are not well-connected. Namely, if H is a maximal k-vertex-connected subgraph of G, we say that H is a *k-vertex-connected component* of G. The edge-connected variant is, however, defined differently: we say that a pair of vertices u, v of G is k-edge-connected if it is not possible to remove at most $k-1$ edges from G so that u and v end up in different connected components. This relation of k-edge-connectedness happens to be an equivalence relation; this yields a definition of a *k-edge-connected component* of G as an equivalence class of the relation. We remark that the notions of k-vertex-connected components and k-edge-connected components coincide for $k = 1$ as both simply describe the connected components of G. However, for $k \geq 2$ these definitions diverge; in particular, for $k \geq 3$ the k-edge-connected components of a graph do not even need to be connected.

There has been a plethora of research into the algorithms deciding the k-vertex- and k-edge-connectedness of graphs, and decomposing the graphs into k-vertex- or k-edge-connected components. However, while classical, elementary, and efficient algorithms exist for $k = 1$ and $k = 2$, these problems become increasingly more difficult for the larger values of k. In fact, even for $k = 4$, there were no known linear time algorithms to any of the considered problems. The following description presents the previous work in this area for $k \in \{1, 2, 3, 4\}$, and exhibits the related work for the larger values of k:

k = 1. Here, the notions of k-vertex-connectedness and k-edge-connectedness reduce to that of connectivity and the connected components of a graph. In the static setting, determining the connected components in linear time is trivial. As a consequence, more focus is being laid on dynamic algorithms maintaining the connected components of graphs. In the incremental setting, where the edges can only be added to the dynamic graph, the optimal solution is provided by disjoint-set data structures [38], which solve the problem in the amortized $\mathcal{O}(\alpha(n))$ time per query, where $\alpha(n)$ denotes the inverse of the Ackermann's fast-growing function. The fully dynamic data structures are also considered [44, 7, 9, 18, 19, 20, 40, 27, 22, 25].

k = 2. One step further are the notions of 2-vertex-connectivity (biconnectivity) and 2-edge-connectivity. In the static setting, partitioning of a graph into 2-vertex-connected or 2-edge-connected components are classical problems, both solved in linear time by exploiting the properties of the low function [23]. The incremental versions of both problems are again solved optimally in the amortized $\mathcal{O}(\alpha(n))$ time per query [43]. Significant research has been done in the dynamic setting as well [36, 20, 7, 22, 11, 17, 31].

k = 3. As a next step, we consider 3-vertex-connectivity (triconnectivity) and 3-edge-connectivity. An optimal, linear time algorithm detecting the 3-vertex-connected components was first given by Hopcroft and Tarjan [24]. The first linear time algorithm for 3-edge-connectivity was discovered much later by Galil and Italiano [12], where they present a linear time reduction from the k-edge-connectivity problem to k-vertex-connectivity for $k \geq 3$, showing that in the static setting, the former problem is the easier of the two. This was later followed by a series of works simplifying the solution for 3-edge-connectivity [41, 35, 34, 42]. The incremental setting [32, 21] and the dynamic setting [11, 7] were also considered.

We also mention that in the case of 3-vertex-connectivity, there exists a structure called SPQR-tree which succinctly captures the structure of 2-vertex-cuts in graphs [4, 21]. Its edge-connectivity analogue also exists, but we defer its introduction to the general setting.

k = 4. We move on to the problems of 4-vertex-connectivity and 4-edge-connectivity. A notable result by Kanevsky et al. [26] supports maintaining 4-vertex-connected components in incremental graphs, with an optimal $\mathcal{O}(\alpha(n))$ amortized time per query. Their result also yields the solution for static graphs in $\mathcal{O}(m + n\alpha(n))$ time complexity. By applying the result of Galil and Italiano [12], we derive a static algorithm determining the 4-edge-connected components in the same time complexity. This algorithm is optimal for $m = \Omega(n\alpha(n))$.

Another result by Dinitz and Westbrook [6] supports maintaining the 4-edge-connected components in the incremental setting. Their algorithm processes any sequence of queries in $\mathcal{O}(q + m + n \log n)$ time where q is the number of queries, and m is the total number of inserted edges to the graph.

However, it is striking that the fastest solutions for 4-edge-connectivity and 4-vertex-connectivity for static graphs were derived from the on-line algorithms working in the incremental setting. In particular, no linear time algorithms for $k = 4$ were known before.

$\mathbf{k \geq 5}$. As a side note, we also present the current knowledge on the general problems of k-vertex-connectivity and k-edge-connectivity. A series of results [29, 30, 16, 2] show that it is possible to compute the minimum edge cut of a graph (i.e., determine the edge-connectivity of a graph) in near-linear time. The previously mentioned work by Dinitz and Westbrook [6] maintains the k-edge-connected components of an incremental graph which is assumed to already have been initialized with a $(k-1)$-edge-connected graph. The data structure answers any sequence of on-line queries in $\mathcal{O}(q + m + k^2 n \log(n/k))$ time, where q is the number of queries, and m is the number of edges in the initial graph.

Gomory and Hu [13] proved that for any weighted, undirected graph G there exists a weighted, undirected tree T on the same vertex set such that for any two vertices $s, t \in V(G)$, the value of the minimum s-t edge cut in T is equal to the value of the minimum s-t edge cut in G. Moreover, such a tree can be constructed using $n-1$ invocations of the maximum flow algorithm. In an interesting result by Hariharan et al. [15], the decomposition of any graph into k-edge-connected components is constructed in $\mathcal{O}((m + nk^3) \cdot \mathrm{polylog}(n))$ time, producing a partial Gomory-Hu tree as its result.

Dinitz et el. [5] showed that the set of all minimum edge cuts can be succinctly represented with a *cactus graph*. When the minimum edge cut is odd, this cactus simplifies to a tree (see [8, Corollary 8]). These results imply that if the size of the minimum cut is odd, then the number of minimum cuts is $\mathcal{O}(n)$ and if it is even, then the number of minimum cuts is $\mathcal{O}(n^2)$. The structure of k-vertex-cuts was also investigated [37].

Our results. In this work, we present a linear time, deterministic algorithm partitioning static, undirected graphs into 4-edge-connected components. Even though the area of the dynamic versions of the algorithms for k-edge-connectivity is still thriving, the progress in static variants appears to have plateaued. In particular, both subquadratic algorithms determining the 4-edge-connected components [26, 6] originate from their dynamic incremental equivalents and are almost thirty years old, yet they did not achieve the optimal linear running time. Hence, our work constitutes the first progress in the static setting of 4-edge-connectivity in a long time. As a side result, our algorithm also produces the tree representation of 3-edge-cuts as explained in [8].

Organization of the work. In Section 3 [33], omitted in this abridged version of the work, we show how to reduce the problem of determining 4-edge-connected components to the problem of determining 4-edge-connected components in 3-edge-connected graphs. In Section 4, we show a linear time, randomized Monte Carlo algorithm for listing all 3-edge-cuts in 3-edge-connected graphs. In Section 5, we show how to remove the dependency on the randomness in the algorithm from the previous section, producing a linear time, deterministic algorithm listing all 3-edge-cuts in 3-edge-connected graphs. Then, in Section 6, we construct a tree of 3-edge-cuts in a 3-connected graph, given the list of all its 3-edge-cuts. This tree is then used to determine the 4-edge-connected components of the graph. Finally, in Section 7, we present open problems related to this work.

Section 3 and the proofs of the results marked ★ are deferred to the full version of the work [33] due to the space constraints.

2 Preliminaries

Graphs. In this work, we consider undirected, connected graphs which may contain self-loops and multiple edges connecting pairs of vertices (i.e., multigraphs). The number of vertices of a graph and the number of its edges are usually denoted n and m, respectively.

We use the notions of k-edge-connectedness and k-edge-connected components defined in Section 1. Moreover, we say that a set of k edges of a graph forms a *k-edge-cut* (or a *k-cut* for simplicity) if the removal of these edges from the graph disconnects it.

DFS trees. Consider a run of the *depth-first search* algorithm [39] on a connected graph G. A *depth-first search tree* (or a *DFS tree*) is a spanning tree $\mathcal{T}$ of G, rooted at the source of the search r, containing all the edges traversed by the algorithm. After the search is performed, each vertex v is assigned two values: its *preorder* $\operatorname{pre}(v)$ (also called *discovery time* or *arrival time*) and *postorder* $\operatorname{post}(v)$ (also *finishing time* or *departure time*). Their definitions are standard [3]; it can be assumed that the values range from 1 to $2n$ and are pairwise different.

The edges of $\mathcal{T}$ are called *tree edges*, and the remaining edges are called *back edges* or *non-tree edges*. In this setup, every back edge e connects two vertices remaining in ancestor-descendant relationship in $\mathcal{T}$; moreover, the graph $\mathcal{T} + e$ contains exactly one cycle, named the *fundamental cycle* of e. For a vertex v of G, we define $\mathcal{T}_v$ to be the subtree of $\mathcal{T}$ rooted at v; similarly, for a tree edge e whose deeper endpoint is v, we set $\mathcal{T}_e = \mathcal{T}_v$.

When a DFS tree $\mathcal{T}$ of G is fixed, it is common to introduce directions to the edges of the graph: all tree edges of $\mathcal{T}$ are directed away from the root of $\mathcal{T}$, and all back edges are pointed towards the root of $\mathcal{T}$. Then, uv is a directed edge (either a tree or a back edge) whose *origin* (or *tail*) is u, and whose *destination* (or *head*) is v.

For our convenience, we introduce the following definition: a back edge $e = pq$ *leaps over* a vertex v if $p \in \mathcal{T}_v$, but $q \notin \mathcal{T}_v$; we analogously define *leaping over* a tree edge f.

Moreover, we define a partial order $\leq_{\mathcal{T}}$ on the vertices of G and the tree edges of $\mathcal{T}$ as follows: $x \leq_{\mathcal{T}} y$ if the simple path in $\mathcal{T}$ connecting the root of $\mathcal{T}$ with y also contains x. Then, $\leq_{\mathcal{T}}$ has one minimal element – the root of $\mathcal{T}$ – and each maximal element is a leaf of $\mathcal{T}$. When the tree $\mathcal{T}$ is clear from the context, we may write $\leq$ instead of $\leq_{\mathcal{T}}$. We may also use $x < y$ for $x \leq y \wedge x \neq y$. Using the precomputed preorder and postorder values in $\mathcal{T}$, we can verify if $x \leq_{\mathcal{T}} y$ holds for given $x, y \in V(G) \cup E(\mathcal{T})$ in constant time.

We use the classical low function defined by Hopcroft and Tarjan [23]. However, for our purposes it is more convenient to define it as a function $\operatorname{low} : E(\mathcal{T}) \to (E \setminus E(\mathcal{T})) \cup \{\perp\}$ such that for a tree edge e, $\operatorname{low}(e)$ is the back edge uv leaping over e minimizing the preorder of its head v, breaking ties arbitrarily; or $\perp$, if no such edge exists. This function can be computed for all tree edges in time linear with respect to the size of the graph.

Xors. For sets A and B, by $A \oplus B$ we denote their symmetric difference, which is $(A \cup B) \setminus (A \cap B)$, and we call it a *xor* of A and B. Moreover, for a pair of non-negative integers a and b, by $a \oplus b$ we denote their xor, that is, an integer whose binary representation is a bitwise symmetric difference of the binary representations of a and b. The definitions can be easily generalized to the symmetric differences of multiple sets or integers.

4 Simple randomized algorithm

In this section, we will describe a randomized linear time algorithm listing 3-edge-cuts in 3-edge-connected graphs. In particular, the existence of this algorithm will imply that the number of 3-edge-cuts in any 3-edge-connected graph is at most linear. Since the algorithm is significantly simpler than its deterministic variant, and it already contains most of the core ideas of this work, we believe it serves as a good intermediate step in the explanation.

The overview of the algorithm is as follows. First, we will construct a randomized oracle (Lemma 5) that given two edges e, f of the 3-edge-connected graph G, verifies in constant time whether there exists another edge g such that $\{e, f, g\}$ is a 3-edge-cut of G; and if such an edge exists, it is returned by the oracle. It will be apparent from the description of the oracle that such an edge – if it exists – is unique. Thanks to Lemma 5, the description of our randomized algorithm will be simplified significantly: now, we only need to identify two out of three edges of each 3-edge-cut of G. Then, the remaining edge of each cut can be easily recovered using a single oracle call.

Next, our algorithm will consider an arbitrary DFS tree $\mathcal{T}$ of G, categorize all 3-edge-cuts by the number $t \in \{0, 1, 2, 3\}$ of tree edges of $\mathcal{T}$ in the cut, and find all 3-edge-cuts separately for each value of t. The case $t = 0$ is trivial: $\mathcal{T}$ is a spanning tree of G, hence each edge cut must contain at least one edge of $\mathcal{T}$. The cases $t = 1$ and $t = 2$ will proceed by considering all tree edges of $\mathcal{T}$ separately. For each such tree edge e, we will find all 3-edge cuts containing both e and some non-tree edge leaping over e. By performing a case study, we will show that for each e, there is only a constant number of 3-edge-cuts of this form for $t \in \{1, 2\}$; and moreover, each of them can be located by us in constant time after a linear time preprocessing of $\mathcal{T}$. Finally, the case $t = 3$ will be solved by a recursive call to the algorithm on a properly contracted graph.

We begin the detailed explanation with the description of some auxiliary data structures.

▶ **Theorem 1** ([10])**.** *There exists a data structure for the disjoint set union problem which, when initialized with an undirected tree $\mathcal{T}$ (the "union tree") on n vertices, creates n singleton sets. After the initialization, the data structure accepts the following queries in any order:*

- find(x)*: returns the index of the set containing x,*
- union(x, y)*: if x and y are in different sets, then an arbitrary one of them is replaced with their union and the other one with the empty set. This query can only be issued if xy is an edge of $\mathcal{T}$.*

The data structure executes any sequence of q queries in total $\mathcal{O}(n + q)$ time.

▶ **Lemma 2** (★)**.** *It is possible to enrich the data structure from Theorem 1, so that after rooting it at an arbitrary vertex, we are able to answer the following query in constant time:*

- lowest(x)*: returns the smallest vertex of the set containing x with respect to $\leq_{\mathcal{T}}$.*

Note that each set induces a connected subgraph of $\mathcal{T}$, so its smallest vertex is well-defined.

▶ **Theorem 3** (★)**.** *There exists a deterministic algorithm that takes as input:*

- *an undirected, unrooted tree $\mathcal{T}$ with n vertices,*
- *p weighted paths $P_1, P_2, \dots, P_p$ in the tree, where the path P_i has weight $w_i \in \{0, 1, \dots, C\}$,*
- *and a positive integer k,*

and for each edge e of the tree returns the indices of k paths with the lowest weight containing e, breaking ties arbitrarily; if e is a part of fewer than k paths, all such paths are returned. The time complexity of the algorithm is $\mathcal{O}(nk + p + C)$.

We proceed to the description of our randomized algorithm. Let us choose an arbitrary vertex r of the graph, and perform a depth-first search from r. Let $\mathcal{T}$ be the resulting DFS tree. We are now going to define a *hashing function* $H : E \to \mathcal{P}(E)$, where $\mathcal{P}(E)$ denotes the powerset of E; the value $H(e)$ will be called *a hash* of e. If $e \notin \mathcal{T}$, then we define $H(e) = \{e\}$. Otherwise, we take $H(e)$ as the set of non-tree edges leaping over e. Let us note that the set C_e of edges f such that $e \in H(f)$ forms a cycle – the fundamental cycle of e.

▶ **Lemma 4** (★)**.** *For a connected graph $G = (V, E)$ and a subset A of its edges, the graph $G' := G - A$ is disconnected if and only if there is a nonempty subset $B \subseteq A$ such that xor of the hashes of the edges in B is an empty set.*

Since our graph has no 1-edge-cuts or 2-edge-cuts, no created hashes are empty, and no two edges have equal hashes. Hence, removing a set of three edges disconnects a 3-edge-connected graph if and only if xor of the hashes of all of them is the empty set. Moreover, after removing some 3-edge-cut, the graph disconnects into exactly two components, and no removed edge connects vertices within one component.

As storing hashes as sets of edges would be inefficient, we define *compressed hashes*, resembling the notion of sketches introduced by Ahn et al. [1] and Kapron et al. [28]. We express compressed hashes as b-bit numbers where $b = \lceil 3 \log_2(m) \rceil$, i.e., as a function $CH : E \to \{0, \dots, 2^b - 1\}$. For each non-tree edge e, we draw $CH(e)$ randomly and uniformly from the set of b-bit numbers. For a tree edge e, we define its compressed hash $CH(e)$ as the xor of the compressed hashes of the edges in its hash, i.e., $CH(e) = \bigoplus_{f \in H(e)} CH(f)$. Note that since $b = \mathcal{O}(\log m)$, we can perform arithmetic operations on compressed hashes in constant time. However, this comes at a cost of allowing the collisions of compressed hashes.

For the ease of exposition, in the description of this algorithm we will use hash tables; because of that, the algorithm described in this section will have expected linear instead of worst-case linear running time. Formally, we create a hash table $M : \{0, \dots, 2^b - 1\} \to E \cup \{\bot\}$, which for any b-bit number x returns an edge whose compressed hash is equal to x; or $\bot$ if no such edge exists. In the unlikely event that there are multiple edges whose compressed hashes are equal to x, M returns any of them.

We will now categorize 3-edge-cuts based on the number of tree edges they contain, and show how to handle each case. We remark that each 3-edge-cut must intersect $\mathcal{T}$ as it is a spanning tree of G. Therefore, a 3-edge-cut may intersect $\mathcal{T}$ in one edge (Subsection 4.1), two edges (Subsection 4.2), or three edges (Subsection 4.3).

Throughout the case analysis, we will heavily rely on the following fact:

▶ **Lemma 5.** *Given two edges e and f of some 3-edge-cut in a 3-edge-connected graph, the remaining edge is uniquely identified by its hash: $H(e) \oplus H(f)$.*

Proof. If $\{e, f, g\}$ is a 3-edge-cut, then $H(e) \oplus H(f) \oplus H(g) = \varnothing$, so $H(g) = H(e) \oplus H(f)$. ◀

4.1 One tree edge

Let e be an edge of $\mathcal{T}$ that is the only tree edge of some 3-edge-cut. The two resulting connected components after the removal of such a cut are $\mathcal{T}_e$ and $\mathcal{T} \setminus \mathcal{T}_e$. As e is not a bridge, $\mathrm{low}(e)$ is well-defined and connects $\mathcal{T}_e$ with $\mathcal{T} \setminus \mathcal{T}_e$, so it belongs to the cut as well. By Lemma 5, this uniquely determines the third edge of this cut. Thus, in order to detect all such cuts, we iterate over all tree edges e and for each of them, we look up in M if there exists an edge with compressed hash equal to $CH(e) \oplus CH(\mathrm{low}(e))$. If it exists and if it turns out to be a non-tree edge g, we output the triple $\{e, \mathrm{low}(e), g\}$ as a 3-edge-cut.

4.2 Two tree edges

Let e and f be the edges of $\mathcal{T}$ that form a 3-edge-cut c together with some back edge g. Without loss of generality, we can assume that $e <_{\mathcal{T}} f$, thanks to the following fact:

▶ **Lemma 6 (★).** *The edges e and f are comparable with respect to $\leq_{\mathcal{T}}$.*

The two connected components of $G \setminus c$ are $L \coloneqq \mathcal{T}_e \setminus \mathcal{T}_f$ and $R \coloneqq \mathcal{T} \setminus L$, and the remaining back edge g connects L and R. We distinguish two cases, differing in the location of g (see Figure 2 from the full version of this work): *two tree edges, lower case*, where g connects L with $\mathcal{T}_f$, and *two tree edges, upper case*, where g connects L with $\mathcal{T} \setminus \mathcal{T}_e$.

Two tree edges, lower case. Let A be the set of back edges between $\mathcal{T}_f$ and $\mathcal{T} \setminus \mathcal{T}_f$. It consists of the edge g, connecting L with $\mathcal{T}_f$, and of several edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$. Let B be the set of heads of edges from A (we remind that back edges are directed towards the root r). All elements of B lie on the path from f to the root r, and the head of g is the deepest element of B.

Using this observation, we will use the algorithm from Theorem 3. We initialize an instance of it with the tree $\mathcal{T}$, $k = 1$, $C = 2n$. Then, for each back edge $e = xy$, we create one input path P_e from x to y of weight $w_e := 2n - \mathrm{pre}(y)$. The algorithm determines, for each tree edge f, the back edge leaping over f with the largest preorder of its head; call this edge $\mathrm{MaxUp}(f)$. This back edge is the only candidate for g, given f.

Hence, we can find all such cuts by firstly initializing the data structure, and then iterating over all tree edges f. For each f, we take $g = \mathrm{MaxUp}(f)$. Knowing f and g, we can look up in M whether there exists an edge whose compressed hash is $CH(f) \oplus CH(g)$ (Lemma 5). If it exists and if it is a tree edge, then we call it e and output the triple $\{e, f, g\}$ as a 3-edge-cut.

Two tree edges, upper case. Let A be the set of non-tree edges between $\mathcal{T}_e$ and $\mathcal{T} \setminus \mathcal{T}_e$. It consists of the back edge g, connecting L with $\mathcal{T} \setminus \mathcal{T}_e$, and of several edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$. Let B be the set of the tails of the edges from A. Let $v \in L$ be the tail of g. As $v \notin \mathcal{T}_f$, then we either have $\mathrm{pre}(v) < \min_{u \in \mathcal{T}_f} \mathrm{pre}(u)$, or $\mathrm{pre}(v) > \max_{u \in \mathcal{T}_f} \mathrm{pre}(u)$. In the first case, v is the vertex with the smallest preorder which is a tail of some edge leaping over e; while in the second case v is the analogous vertex with the largest preorder.

Based on this observation, we will again use the algorithm from Theorem 3 again. We initialize one instance of it with the tree $\mathcal{T}$, $k = 1$, and $C = 2n$. Then for each back edge $e = xy$, we create one input path P_e from x to y of weight $w_e := \mathrm{pre}(x)$. We also initialize another instance of this algorithm in the same way, only that we set $w_e := 2n - \mathrm{pre}(x)$ instead. The first instance determines, for each tree edge e, the back edge $\mathrm{MinDn}(e)$ leaping over e with the smallest preorder of its tail; while the second instance determines the analogous edge $\mathrm{MaxDn}(e)$, with the largest preorder of its tail. By our considerations above, if any desired $\{e, f, g\}$ cut exists, then $g \in \{\mathrm{MinDn}(e), \mathrm{MaxDn}(e)\}$.

Hence, we can find all such cuts by firstly initializing both instances of the data structure. Then, we iterate over all tree edges e, and for each e we check both candidates for g. If our hash table contains an edge whose compressed hash is $CH(e) \oplus CH(g)$ and it is a tree edge, then we call it f and output the triple $\{e, f, g\}$ as a 3-edge-cut.

4.3 Three tree edges

Solving this case in a way similar to the previous cases seems intractable. We consider this subsection, together with the time analysis following it, as one of the key ideas of this work.

In this case, we assume that no non-tree edges belong to the cut. Therefore, we can contract all of them simultaneously and recursively list all 3-edge-cuts in the resulting graph. Since 3-edge-cuts in the contracted graph exactly correspond to the 3-edge-cuts consisting solely of tree edges in the original graph, this reduction is sound. The contraction is performed in $\mathcal{O}(n + m)$ time by identifying the vertices within the connected components of $G - E(\mathcal{T})$. Let G' be the graph after these contractions. Since G is 3-edge-connected, G' is 3-edge connected as well, so the assumption about the input graph being 3-edge-connected is preserved. We do not modify the value of b in the subsequent recursive calls, even though the value of m decreases.

4.4 Time and correctness analysis

We will now compute the expected time complexity of our algorithm. The subroutines from Subsections 4.1 and 4.2 clearly take expected linear time. The only nontrivial part of the analysis is the recursion in Subsection 4.3. Let $T(m)$ denote the maximum expected time our algorithm needs to solve any graph with at most m edges. Since our graph is 3-edge-connected, the degree of each vertex in G is at least 3, so $|E(G)| \geq \frac{3}{2}|V(G)| \geq \frac{3}{2}|E(G')| \Rightarrow |E(G')| \leq \frac{2}{3}|E(G)|$. Therefore, we have that $T(m) \leq \mathcal{O}(m) + T\left(\frac{2}{3}m\right)$. The solution to this recurrence is $T(m) = \mathcal{O}(m)$, hence the whole algorithm runs in expected linear time.

We proceed to proving that our algorithm works correctly with sufficiently high probability. The only reason it can output wrong result comes from the compression of hashes – if we used their uncompressed version instead, the algorithm would clearly be correct.

The maximum number of queries to our hash table is linear in terms of n, which follows from a similar argument to the one presented in Subsection 4.4. Hence, there exists some absolute constant c such that the number of queries is bounded by cn. For each query with value q, and for each edge whose hash is not equal to the hash whose compressed version we ask about, there is 2^{-b} probability that compressed version of this hash is equal to q. Hence, the probability that we ever get a false positive is bounded from above by $cnm2^{-b}$. Since $b = \lceil 3\log_2(m) \rceil$ and $n \leq m$, we infer that $cnm2^{-b} \leq \frac{c}{m}$. Therefore, our algorithm works correctly with probability at least $1 - \frac{c}{m}$.

5 Deterministic algorithm

Having established a linear time randomized algorithm producing 3-edge-cuts in 3-edge-connected graphs, we will now determinize it by designing deterministic implementations of the subroutines for each of the cases considered in the randomized algorithm. Three cases need to be derandomized: "one tree edge" (Subsection 4.1), "two tree edges, lower case", and "two tree edges, upper case" (Subsection 4.2). In the following description, we cannot use compressed hashes anymore: the compression is a random process which inevitably results in false positives. Instead, we will exploit additional properties of 3-edge-cuts in order to produce an efficient deterministic implementation of the algorithm.

Recall that in the description of the randomized implementation of the algorithm, we defined the values low(e), MaxUp(e), MinDn(e), and MaxDn(e) for any tree edge e. For the deterministic variant of the algorithm, we generalize these notions: we define low1(e), low2(e), and low3(e) as the three back edges leaping over e with the minimal preorders of their targets; in particular, we set low1(e) := low(e). We analogously define MaxUp1(e), MaxUp2(e), MinDn1(e), MinDn2(e), MaxDn1(e), and MaxDn2(e). We remark that low3(e) might not exist if there are fewer than three edges leaping over e; in this case, we put low3$(e) := \perp$. However, all the other values must exist – otherwise, at most one edge would leap over e, which would mean that this edge, together with e, would form a 2-edge-cut of G.

It is straightforward to compute all the values defined above in linear time; for instance, the generalizations of MaxUp, MinDn, and MaxDn can be determined by passing $k = 2$ to the algorithm in Theorem 3 instead of $k = 1$. Hence, in the following description, we will assume that all the values above have already been computed.

5.1 One tree edge

Recall that in this case, we are to find all 3-edge-cuts intersecting a fixed depth-first search tree $\mathcal{T}$ in a single edge. This is fairly straightforward: if some 3-edge-cut C contains exactly one tree edge e of $\mathcal{T}$, then the border of the cut is exactly $\mathcal{T}_e$; hence, this cut must include all back edges leaping over e. Therefore, e belongs to a 3-edge-cut of this kind if and only if $\mathrm{low3}(e) = \perp$, i.e. if there only exist two back edges leaping over e; in this case, e and these two edges form a 3-edge-cut. Since this check can be easily performed in $\mathcal{O}(n)$ time for all tree edges in $\mathcal{T}$, the whole subroutine runs in $\mathcal{O}(n + m)$ time complexity.

5.2 Two tree edges, lower case

Recall that this case requires us to find all 3-edge-cuts intersecting a fixed depth-first search tree $\mathcal{T}$ in two edges, say e and f, such that $e < f$ and the remaining back edge g connects $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$. Hence, g is the only back edge connecting $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$, no back edges connect $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$, but there may be multiple back edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$.

We shall exploit the fact that in a valid 3-edge-cut of this kind, all back edges leaping over e must originate from $\mathcal{T}_f$. This observation severely limits the set of possible tree edges f. This is formalized by the notion of the *deepest down cut* of e:

▶ **Definition 7** (deepest down cut). *For a tree edge e in $\mathcal{T}$, we define the* deepest down cut *of e, denoted $\mathrm{DeepestDnCut}(e)$, as the deepest tree edge $f \geq e$ for which all back edges leaping over e originate from $\mathcal{T}_f$.*

▶ **Lemma 8** (★). *For every tree edge e, $\mathrm{DeepestDnCut}(e)$ is defined correctly and uniquely. Moreover, every 3-edge-cut $\{e, f, g\}$ of the considered kind satisfies $e < f \leq \mathrm{DeepestDnCut}(e)$.*

▶ **Lemma 9** (★). *For every tree edge e, $\mathrm{DeepestDnCut}(e) = uv$ is the tree edge whose head v is the lowest common ancestor of two vertices: the tails of $\mathrm{MinDn1}(e)$ and $\mathrm{MaxDn1}(e)$.*

Thanks to Lemma 9, we can compute $\mathrm{DeepestDnCut}(e)$ in constant time for each tree edge e as the lowest common ancestor of two vertices can be computed in constant time after linear preprocessing [14].

We now shift our focus to the lower tree edge f. As in Section 4, the only possible candidate g for a back edge of the cut leaping over f is given by $\mathrm{MaxUp1}(f)$. The only problematic part is locating the remaining tree edge e. Previously, we utilized randomness in order to calculate the compressed hash of e given the compressed hashes of f and g. Here, we instead use the following fact:

▶ **Lemma 10** (★). *If $\{e, f, g\}$ is a 3-edge-cut of the considered kind for some tree edges e and $f > e$, then e is the deepest tree edge satisfying $\mathrm{DeepestDnCut}(e) \geq f$.*

Lemmas 8 and 10 naturally lead to the following idea: given a tree edge e, the set of edges f satisfying $\mathrm{DeepestDnCut}(e) \geq f$ is a path P_e connecting the head of e with the head of $\mathrm{DeepestDnCut}(e)$; we assign this path a weight w_e equal to the depth of e in $\mathcal{T}$. We then invoke Theorem 3 with the tree $\mathcal{T}$, the weighted paths $\{P_e \mid e \in E(\mathcal{T})\}$, and $k = 1$. This lets us find, in $\mathcal{O}(n)$ time, for each tree edge $f \in E(\mathcal{T})$, the path P_e of the maximum weight containing f as an edge. This path naturally corresponds to the tree edge e from Lemma 10. This way, for every tree edge f, we have uniquely identified a back edge g and a tree edge e such that the only possible 3-edge-cut containing f as a deeper tree edge is $\{e, f, g\}$.

It only remains to verify that $\{e, f, g\}$ is a 3-edge-cut. Since $\mathrm{DeepestDnCut}(e) \geq f$ guarantees that all back edges leaping over e originate from $\mathcal{T}_f$, we only need to check that exactly one edge (that is, g) connects $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$. As $g = \mathrm{MaxUp1}(f)$, it suffices to verify that the edge $\mathrm{MaxUp2}(f)$, which is a back edge leaping over f whose head is the deepest apart from g, also leaps over e (i.e., it does not terminate in $\mathcal{T}_e$).

It can be easily verified that the implementation of the subroutine is deterministic and runs in linear time with respect to the size of G.

5.3 Two tree edges, upper case

Recall that in this case, we are required to find all 3-edge-cuts intersecting $\mathcal{T}$ in two edges e and f, such that $e < f$ and the remaining back edge g connects $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$. Similarly to the randomized case, we use the fact that given a tree edge e, the tail of g has either the smallest preorder (i.e., $g = \mathrm{MinDn1}(e)$) or the largest preorder (i.e., $g = \mathrm{MaxDn1}(e)$) among all the back edges leaping over e. Without loss of generality, assume that $g = \mathrm{MinDn1}(e)$; the latter case is analogous.

In a similar vein to previous case, observe that if $\{e, f, g\}$ is a 3-edge-cut for some $f > e$, then all back edges leaping over e other than g must originate from $\mathcal{T}_f$.

This leads to the following slight generalization of DeepestDnCut (Definition 7):

▶ **Definition 11.** *For a tree edge e in $\mathcal{T}$, we define the value* $\mathrm{DeepestDnCutNoMin}(e)$ *as the deepest tree edge $f \geq e$ for which all back edges leaping over e other than* $\mathrm{MinDn1}(e)$ *originate from $\mathcal{T}_f$.*

The process of computation of $\mathrm{DeepestDnCut}(e)$ asserted by Lemma 9 can be easily modified to match our needs: we take $\mathrm{DeepestDnCutNoMin}(e)$ as the tree edge whose head is the lowest common ancestor of the tails of $\mathrm{MinDn2}(e)$ and $\mathrm{MaxDn1}(e)$. We refer the reader to Figure 4 from the full version of the work for a helpful picture.

Now, fix the shallower tree edge e of the cut. Let $f_0 := \mathrm{DeepestDnCutNoMin}(e)$. Then, if some tree edge $f > e$ belongs to the 3-edge-cut $\{e, f, g\}$, then $f \leq f_0$ (otherwise, there would be multiple edges connecting $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$). The natural question is then: is $\{e, f_0, g\}$ a 3-edge-cut? The only possible problem is that there may exist back edges connecting $\mathcal{T}_{f_0}$ with $\mathcal{T}_e \setminus \mathcal{T}_{f_0}$. Fortunately, if this is the case, then the back edge $\mathrm{MaxUp1}(f_0)$, dependent only on f_0, is one of these back edges.

Hence, let $h_0 := \mathrm{MaxUp1}(f_0)$. If h_0 leaps over e, we are done, and $\{e, f_0, g\}$ is an edge cut. Otherwise, the subtree $\mathcal{T}_f$ for the sought 3-edge-cut $\{e, f, g\}$ must contain the head of h_0 (or else h_0 would connect $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$). Let then f_1, $e \leq f_1 < f_0$ be the deepest tree edge containing the head of h_0, that is, the edge whose head coincides with the head of h_0. Then, the edge f of the 3-edge-cut $\{e, f, g\}$ must satisfy $e < f \leq f_1$. We can then repeat this procedure: given f_1, we compute $h_1 := \mathrm{MaxUp1}(f_1)$, and either h_1 leaps over e and we are done, or we calculate another edge $f_2 < f_1$ supplying a better bound on the depth of f.

Since the graph is finite, this process terminates in $k = \mathcal{O}(n)$ steps, producing the lowest tree edge $f_k \geq e$ such that no back edge connects $\mathcal{T}_{f_k}$ with $\mathcal{T}_e \setminus \mathcal{T}_{f_k}$, and no back edge other than g connects $\mathcal{T}_e \setminus \mathcal{T}_{f_k}$ with $\mathcal{T} \setminus \mathcal{T}_e$. If $f_k = e$, then no 3-edge-cut $\{e, f, g\}$ exists for $f > e$. Otherwise, $\{e, f_k, g\}$ is naturally a correct 3-edge-cut. Since two edges of a 3-edge-cut uniquely identify the third edge (Lemma 5), we conclude that this is the only 3-edge-cut containing e and g. We refer the reader to Figure 5 from the full version of the paper for a helpful picture depicting the iterative process above.

In order to optimize the algorithm, we seek to optimize the iterative process described above. Indeed, for each $e \in E(\mathcal{T})$, the process is very similar: start with some edge $f > e$, and then repeatedly replace f with a higher edge, until a replacement would result in an edge closer to the root than e. We can model this process with a rooted tree $\mathcal{U}$ defined as follows:

- the vertices of $\mathcal{U}$ are the tree edges of $\mathcal{T}$, and $\perp$,
- $\perp$ is the root of $\mathcal{U}$,
- for a non-root vertex e of $\mathcal{U}$, its parent is the tree edge with the same head as $\mathrm{MaxUp1}(e)$.

If the head of $\mathrm{MaxUp1}(e)$ coincides with the root of $\mathcal{T}$, then the parent of e in $\mathcal{U}$ is $\perp$. Naturally, $\mathcal{U}$ can be constructed in linear time with respect to the size of $\mathcal{T}$; moreover, if an edge p is a parent of another edge q in $\mathcal{U}$, then $p <_{\mathcal{T}} q$.

Now, the iteration is equivalent to the repeated replacement of f_0 with its parent in $\mathcal{U}$ as long as the parent is greater or equal than e with respect to $<_{\mathcal{T}}$. In other words, the final edge $f' := \mathsf{final}(f_0, e)$ is taken as the shallowest ancestor of f_0 in $\mathcal{U}$ for which $f' \geq_{\mathcal{T}} e$.

This leads to the final idea: we simulate a forest of rooted subtrees of $\mathcal{U}$ using a disjoint set union data structure $F_{\mathcal{U}}$ (Theorem 1). Each subtree additionally keeps its root, which can be retrieved from $F_{\mathcal{U}}$ (Lemma 2). Initially, each subtree of $\mathcal{U}$ contains a single vertex.

After the initialization of $F_{\mathcal{U}}$, we iterate e over the tree edges of $\mathcal{T}$ in the decreasing order of depth in $\mathcal{T}$. Throughout the process, we maintain the following invariant on $F_{\mathcal{U}}$: an edge $ef \in E(\mathcal{U})$ for $e < f$ has been added to the forest if and only if e has been considered at any previous iteration as the shallower tree edge of the cut. Hence, at the beginning of the iteration for a given edge e, we add to $F_{\mathcal{U}}$ all tree edges of $\mathcal{U}$ originating from e. At this point of time, for every tree edge f such that $f >_{\mathcal{T}} e$, the edge $\mathsf{final}(f, e)$ is given by $F_{\mathcal{U}}.\mathsf{lowest}(f)$. This reduces the entire iterative process described above to a single lowest query on $F_{\mathcal{U}}$.

We initialized $F_{\mathcal{U}}$ on a tree with n vertices, and we issued $\mathcal{O}(n)$ queries to it in total. Therefore, the whole subroutine runs in time linear with respect to the size of G.

Summing up, we replaced each randomized subroutine with its deterministic counterpart, preserving the linear guarantee on the runtime of the algorithm. We conclude that there exists a deterministic linear time algorithm listing 3-edge-cuts in 3-edge-connected graphs.

6 Reconstructing the structure of 4-edge-connected components

In this section, we show how to build a structure of 4-edge-connected components of a 3-edge-connected graph G, given the set $\mathcal{C}$ of all 3-edge-cuts in G.

First of all, recall what such a structure looks like.

▶ **Theorem 12.** *[8, Corollary 8] For a 3-edge-connected graph $G = (V, E)$, there exists a tree $H = (U, F)$ and functions $\phi : \mathcal{C} \to F$ and $\psi : V \to U$, such that ϕ is a bijection from 3-edge-cuts of G to the edges of H, and ψ maps (not necessarily surjectively) vertices of G to the vertices of H so that whole 4-edge-connected components are mapped to the same vertex.*

Moreover, if a 3-edge-cut c partitions the vertices of G into two parts V_1 and V_2, then $\phi(c)$ partitions the vertices of H into U_1 and U_2 such that $\psi^{-1}(U_1) = V_1$ and $\psi^{-1}(U_2) = V_2$.

We refer the reader to Figure 6 in the full version for an illustrative example of a decomposition postulated by Theorem 12.

The tree H is usually unrooted in the literature. However, we are going to root it. Namely, we take a depth-first search tree $\mathcal{T}$ of G, rooted at some vertex r, and we root H at $\psi(r)$. For a vertex $u \in U$, we let H_u denote the subtree of H rooted at u.

▶ **Definition 13.** *For a 3-edge-cut c, we define $P(c)$ as the set of vertices from the connected component of $G \setminus c$ not containing r.*

We remark that since c is a minimal cut, $G \setminus c$ consists of two connected components, so $P(c)$ is determined uniquely.

▶ **Lemma 14 (★).** *Let $v, u \in U$, and let $e_1, e_2, \ldots, e_k \in F$ be the sequence of edges of H on the path from v and u in H. If v is an ancestor of u, then for each pair of integers i, j such that $1 \leq i < j \leq k$, we have $|P(\phi^{-1}(e_i))| > |P(\phi^{-1}(e_j))|$.*

▶ **Lemma 15 (★).** *Given a 3-edge-connected graph G and the set of all its 3-edge-cuts $\mathcal{C}$, the sizes of $P(c)$ for all $c \in \mathcal{C}$ can be computed in linear time with respect to the size of G.*

To reconstruct H, we will also use the following structural lemma about cuts sharing the same edge of graph G.

▶ **Lemma 16 (★).** *For an edge $(u, v) = e \in E$, let $l(e)$ be the set of all 3-edge-cuts containing e. The image $\phi(l(e))$, i.e., the set of all edges of H corresponding to the edge cuts containing e, forms a path in H between $\psi(u)$ and $\psi(v)$.*

We remark an edge case in Lemma 16: if $\psi(u) = \psi(v)$ for some edge $e = (u, v)$, then the image $\phi(l(e))$ is empty. Moreover, since each 3-edge-cut $c \in \mathcal{C}$ contains at least one tree edge of $\mathcal{T}$, each edge $\phi(c)$ is covered by at least one path $\phi(l(e))$ for $e \in E(\mathcal{T})$. Equivalently, H is a tree, rooted at $\psi(r)$, equal to the union of all paths $\phi(l(e))$ for $e \in E(\mathcal{T})$. This representation of H is the cornerstone of our algorithm reconstructing H from G and $\mathcal{C}$.

▶ **Lemma 17.** *There exists a linear time algorithm which, given a graph G and the list $\mathcal{C}$ of all 3-edge-cuts of G, constructs the tree H, along with the mappings ϕ and ψ.*

Sketch of the proof. For each edge $e \in \mathcal{T}$, create a list $l(e)$ of all 3-edge-cuts c containing e, sorted decreasingly by the size of $P(c)$ using radix sort. Let $e_1, \ldots, e_{n-1} \in E(\mathcal{T})$ be the sequence of edges visited by a depth-first search of $\mathcal{T}$ and let $e_i = u_i v_i$, where u_i is the vertex of $\mathcal{T}$ closer to the root r. We create H iteratively; initially, H is a single vertex $\psi(r)$. We maintain the following invariant after k iterations of the algorithm: H is a connected tree, rooted at $\psi(r)$, equal to the union of all paths $\phi(l(e_i))$ for $i \in \{1, 2, \ldots, k\}$.

It is clear that after $n - 1$ iterations, H will be the required rooted tree. Consider the k-th iteration of the algorithm, $k \in [1, n-1]$, in which we need to add to H the path $\phi(l(e_k))$, originating from $\psi(u_k)$ and terminating at $\psi(v_k)$. It can be proved that $\psi(u_k)$, together with the vertical path connecting it with the root of H, is already in H. Now, adding the path $\phi(l(e_k))$ to H is rather straightforward: first, starting from $\psi(u_k)$, we go up the tree H along the edges of H corresponding to the edge cuts containing e_k. Then, we proceed down the tree: we iterate the list $l(e_k)$ of cuts, excluding the cuts corresponding to the edges visited in the first part of the traversal. For each such cut, we go down the tree along the edge corresponding to the cut (creating it, if necessary). Each 3-edge-cut c is considered in only a constant number of iterations of the algorithm: an edge of H corresponding to c is traversed by the path $\phi(l(e_k))$ if and only if $e_k \in c$. Therefore, the time complexity of all iterations in total is $\mathcal{O}(m + |\mathcal{C}|) = \mathcal{O}(m + n)$. ◀

This concludes the construction of a tree representing all 3-edge-cuts in G. As a result, each vertex of H, as long as it is not empty, contains a single 4-edge-connected component of G. Hence, this algorithm also computes the decomposition of a 3-edge-connected graph G into 4-edge-connected components in total linear time.

7 Open problems

As a natural open problem whose resolving would complement this result nicely, we suggest investigating if it is possible to extend our result to vertex connectivity or to the higher-order edge connectivity.

Problem 1. Given an undirected graph $G = (V, E)$, is it possible to find all 4-vertex-connected components of G in linear time?

Problem 2. Given an undirected graph $G = (V, E)$, is it possible to find all 5-edge-connected components of G in linear time?

We also remark that our algorithm assumes the word RAM model in which we can perform any arithmetic and bitwise operations on pointers and $\mathcal{O}(\log n)$-bit words in constant time; this is required by the linear time data structure of Gabow and Tarjan (Theorem 1, [10]). The natural question is whether this assumption can be avoided.

Problem 3. Given an undirected graph $G = (V, E)$, is it possible to find all 4-edge-connected components of G in linear time in the pointer machine model?

References

1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing Graph Structure via Linear Measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, page 459–467, USA, 2012. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611973099.40.

2 Nalin Bhardwaj, Antonio Molina Lovett, and Bryce Sandlund. A Simple Algorithm for Minimum Cuts in Near-Linear Time. In Susanne Albers, editor, *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands*, volume 162 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SWAT.2020.12.

3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

4 Giuseppe Di Battista and Roberto Tamassia. On-Line Graph Algorithms with SPQR-Trees. In Michael S. Paterson, editor, *Automata, Languages and Programming*, pages 598–611, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

5 Yefim Dinitz, Alexander V. Karzanov, and Michael V. Lomonosov. On the structure of a family of minimum weighted cuts in a graph. *Studies in Discrete Optimization*, page 290–306, 1976.

6 Yefim Dinitz and Jeffery R. Westbrook. Maintaining the Classes of 4-Edge-Connectivity in a Graph On-Line. *Algorithmica*, 20(3):242–276, March 1998. doi:10.1007/PL00009195.

7 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. doi:10.1145/265910.265914.

8 Tamás Fleiner and András Frank. A quick proof for the cactus representation of mincuts. Technical Report QP-2009-03, Egerváry Research Group, Budapest, 2009.

9 Greg N. Frederickson. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *SIAM Journal on Computing*, 14(4):781–798, 1985. doi:10.1137/0214055.

10 Harold N. Gabow and Robert Endre Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

11 Zvi Galil and Giuseppe F. Italiano. Fully Dynamic Algorithms for Edge Connectivity Problems. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 317–327, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/103418.103454.

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

13 Ralph E. Gomory and T. C. Hu. Multi-Terminal Network Flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. URL: http://www.jstor.org/stable/2098881.

14 Dov Harel and Robert Endre Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

15 Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Efficient Algorithms for Computing All Low s-t Edge Connectivities and Related Problems. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, page 127–136, USA, 2007. Society for Industrial and Applied Mathematics.

16 Monika Henzinger, Satish Rao, and Di Wang. *Local Flow Partitioning for Faster Edge Connectivity*, pages 1919–1938. Society for Industrial and Applied Mathematics, 2017. doi:10.1137/1.9781611974782.125.

17 Monika Rauch Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, June 1995. doi:10.1007/BF01189067.

18 Monika Rauch Henzinger and Valerie King. Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '95, page 519–527, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/225058.225269.

19 Monika Rauch Henzinger and Mikkel Thorup. Sampling to Provide or to Bound: With Applications to Fully Dynamic Graph Algorithms. *Random Struct. Algorithms*, 11(4):369–379, 1997.

20 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

21 Jacob Holm and Eva Rotenberg. Worst-Case Polylog Incremental SPQR-Trees: Embeddings, Planarity, and Triconnectivity. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 2378–2397, USA, 2020. Society for Industrial and Applied Mathematics.

22 Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic Bridge-Finding in $\widetilde{O}(\log^2 n)$ Amortized Time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, page 35–52, USA, 2018. Society for Industrial and Applied Mathematics.

23 John Hopcroft and Robert Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, 1973. doi:10.1145/362248.362272.

24 John Hopcroft and Robert Tarjan. Dividing a Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135–158, 1973. doi:10.1137/0202012.

25 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully Dynamic Connectivity in $O(\log n(\log\log n)^2)$ Amortized Expected Time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, page 510–520, USA, 2017. Society for Industrial and Applied Mathematics.

26 Arkady Kanevsky, Roberto Tamassia, Giuseppe Di Battista, and Jianer Chen. On-Line Maintenance of the Four-Connected Components of a Graph. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 793–801, 1991. doi:10.1109/SFCS.1991.185451.

27 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1131–1142, USA, 2013. Society for Industrial and Applied Mathematics.

28 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1131–1142, USA, 2013. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611973105.81.

29 David R. Karger. Minimum Cuts in Near-Linear Time. *J. ACM*, 47(1):46–76, 2000. doi:10.1145/331605.331608.

30 Kenichi Kawarabayashi and Mikkel Thorup. Deterministic Global Minimum Cut of a Simple Graph in Near-Linear Time. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 665–674, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2746539.2746588.

31 Johannes A. La Poutré and Jeffery Westbrook. Dynamic 2-Connectivity with Backtracking. *SIAM J. Comput.*, 28(1):10–26, 1999. doi:10.1137/S0097539794272582.

32 Johannes A. La Poutré, Jan van Leeuwen, and Mark H. Overmars. Maintenance of 2- and 3-edge-connected components of graphs I. *Discrete Mathematics*, 114(1):329–359, 1993. doi:10.1016/0012-365X(93)90376-5.

33 Wojciech Nadara, Mateusz Radecki, Marcin Smulewicz, and Marek Sokołowski. Determining 4-edge-connected components in linear time. *CoRR*, abs/2105.01699, 2021. arXiv:2105.01699.

34 Hiroshi Nagamochi and Toshihide Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9(2):163, June 1992. doi:10.1007/BF03167564.

35 Nima Norouzi and Yung H. Tsin. A simple 3-edge connected component algorithm revisited. *Information Processing Letters*, 114(1):50–55, 2014. doi:10.1016/j.ipl.2013.09.010.

36 Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. Optimal Offline Dynamic 2, 3-Edge/Vertex Connectivity. In Zachary Friggstad, Jörg-Rüdiger Sack, and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *Lecture Notes in Computer Science*, pages 553–565. Springer, 2019. doi:10.1007/978-3-030-24766-9_40.

37 Seth Pettie and Longhui Yin. The Structure of Minimum Vertex Cuts, 2021. arXiv:2102.06805.

38 Robert E. Tarjan and Jan van Leeuwen. Worst-Case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, March 1984. doi:10.1145/62.2160.

39 Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

40 Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 343–350, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/335305.335345.

41 Yung H. Tsin. A Simple 3-Edge-Connected Component Algorithm. *Theory of Computing Systems*, 40(2):125–142, February 2007. doi:10.1007/s00224-005-1269-4.

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

43 Jeffery Westbrook and Robert E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1):433–464, June 1992. doi:10.1007/BF01758773.

44 Christian Wulff-Nilsen. Faster Deterministic Fully-Dynamic Graph Connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1757–1769, USA, 2013. Society for Industrial and Applied Mathematics.