Algorithmica (2017) 77:309–335
DOI 10.1007/s00453-015-0075-x

Certifying 3-Edge-Connectivity

Kurt Mehlhorn[1] · **Adrian Neumann**[1] ·
Jens M. Schmidt[1]

Received: 4 November 2013 / Accepted: 10 September 2015 / Published online: 22 September 2015

Abstract We present a certifying algorithm that tests graphs for 3-edge-connectivity; the algorithm works in linear time. If the input graph is not 3-edge-connected, the algorithm returns a 2-edge-cut. If it is 3-edge-connected, it returns a construction sequence that constructs the input graph from the graph with two vertices and three parallel edges using only operations that (obviously) preserve 3-edge-connectivity. Additionally, we show how to compute and certify the 3-edge-connected components and a cactus representation of the 2-cuts in linear time. For 3-vertex-connectivity, we show how to compute the 3-vertex-connected components of a 2-connected graph.

Keywords Certifying algorithm · Edge connectivity · Construction sequence

1 Introduction

Advanced graph algorithms answer complex yes-no questions such as "Is this graph planar?" or "Is this graph k-vertex-connected?". These algorithms are not only nontrivial to implement, it is also difficult to test their implementations extensively, as usually only small test sets are available. It is hence possible that bugs persist unrecognized for a long time. An example is the implementation of the linear time planarity test of Hopcroft and Tarjan [10] in LEDA [18]. A bug in the implementation was discovered only after two years of intensive use.

Certifying algorithms [16] approach this problem by computing an additional *certificate* that proves the correctness of the answer. This may, e.g., be either a 2-coloring or an odd cycle for testing bipartiteness, or either a planar embedding or a Kuratowski

✉ Adrian Neumann
aneumann@mpi-inf.mpg.de

[1] Max Planck Institute for Informatics, Saarbrücken, Germany

subgraph for testing planarity. Certifying algorithms are designed such that checking the correctness of the certificate is substantially simpler than solving the original problem. Ideally, checking the correctness is so simple that the implementation of the checking routine allows for a formal verification [1,22].

Our main result is a linear time certifying algorithm for 3-edge-connectivity based on a result of Mader [15]. He showed that every 3-edge-connected graph can be obtained from K_2^3, the graph consisting of two vertices and three parallel edges, by a sequence of three simple operations that each introduce one edge and, trivially, preserve 3-edge-connectivity. We show how to compute such a sequence in linear time for 3-edge-connected graphs. If the input graph is not 3-edge-connected, a 2-edge-cut is computed. The previous algorithms [8,19,28–30] for deciding 3-edge-connectivity are not certifying; they deliver a 2-edge-cut for graphs that are not 3-edge-connected but no certificate in the yes-case.

Our algorithm is path-based [7]. It uses the concept of a *chain decomposition* of a graph introduced in [25] and used for certifying 1- and 2-vertex and 2-edge-connectivity in [27] and for certifying 3-vertex connectivity in [26]. A chain decomposition is a special ear decomposition [14]. We use chain decompositions to certify 3-edge-connectivity in linear time. Thus, chain decompositions form a common framework for certifying k-vertex- and k-edge-connectivity for $k \leq 3$ in linear time. We use many techniques from [26], but in a simpler form. Hence our paper may also be used as a gentle introduction to the 3-vertex-connectivity algorithm in [26].

We state Mader's result in Sect. 3 and introduce chain decompositions in Sect. 4. In Sect. 5 we show that chain decompositions can be used as a basis for Mader's construction. This immediately leads to an $O((m+n)\log(m+n))$ certifying algorithm (Sect. 6). The linear time algorithm is then presented in Sects. 7 and 8. In Sect. 9 we discuss the verification of Mader construction sequences.

The mincuts in a graph can be represented succinctly by a cactus representation [5,6,20]; see Sect. 10. The 3-edge-connected components of a graph are the maximal subsets of the vertex set such that any two vertices in the subset are connected by three edge-disjoint paths. These paths are not necessarily contained in the subset.

Our algorithm can be used to turn any algorithm for computing 3-edge-connected components into a certifying algorithm for computing 3-edge-connected components and the cactus representation of 2-cuts (Sect. 10). An extension of our algorithm computes the 3-edge-connected components and the cactus representation directly (Sect. 11). A similar technique can be used to extend the 3-vertex-connectivity algorithm in [26] to an algorithm for computing 3-vertex-connected components.

2 Related Work

Deciding 3-edge-connectivity is a well researched problem, with applications in diverse fields such as bioinformatics [4] and quantum chemistry [3]. Consequently, there are many different linear time solutions known [8,19,20,28–30]. None of them is certifying. All but the first algorithm also compute the 3-edge-connected components. The cactus representation of a 2-edge-connected, but not 3-edge-connected graph G, can be obtained from G by repeatedly contracting the 3-edge-connected components to single vertices [20].

The paper [16] is a recent survey on certifying algorithms. For a linear time certifying algorithm for 3-vertex-connectivity, see [26] (implemented in [21]). For general k, there is a randomized certifying algorithm for k-vertex connectivity in [13] with expected running time $O(kn^{2.5} + nk^{3.5})$. There is a non-certifying algorithm [12] for deciding k-edge-connectivity in time $O(m \log^3 n)$ with high probability.

In [8], a linear time algorithm is described that transforms a graph G into a graph G' such that G is 3-edge-connected if and only if G' is 3-vertex-connected. Combined with this transformation, the certifying 3-vertex-connectivity algorithm from [26] certifies 3-edge-connectivity in linear time. However, that algorithm is much more complex than the algorithm given here. Moreover, we were unable to find an elegant method for transforming the certificate obtained for the 3-vertex-connectivity of G' into a certificate for 3-edge-connectivity of G.

3 Preliminaries

We consider finite undirected graphs G with $n = |V(G)|$ vertices, $m = |E(G)|$ edges, no self-loops, and minimum degree three, and use standard graph-theoretic terminology from [2], unless stated otherwise. We use $u\,v$ to denote an edge with endpoints u and v.

A set of edges that leaves a disconnected graph upon deletion is called *edge cut*. For $k \geq 1$, let a graph G be *k-edge-connected* if $n \geq 2$ and there is no edge cut $X \subseteq E(G)$ with $|X| < k$. Let $v \rightarrow_G w$ denote a path P between two vertices v and w in G and let $s(P) = v$ and $t(P) = w$ be the source and target vertex of P, respectively (as G is undirected, the direction of P is given by $s(P)$ and $t(P)$). Every vertex in $P \backslash \{s(P), t(P)\}$ is called an *inner vertex* of P and every vertex in P is said to *lie on* P.

Let T be an undirected tree rooted at vertex r. For two vertices x and y in T, x is an *ancestor* of y and y is a *descendant* of x if $x \in V(r \rightarrow_T y)$, where $V(r \rightarrow_T y)$ denotes the vertex set of the path from r to y in T. If additionally $x \neq y$, x is a *proper* ancestor and y is a *proper* descendant. We write $x \leq y$ ($x < y$) if x is an ancestor (proper ancestor) of y. The parent $p(v)$ of a vertex v is its immediate proper ancestor. The parent function is undefined for r. Let K_2^m be the graph on 2 vertices that contains exactly m parallel edges.

Let *subdividing an edge* $u\,v$ of a graph G be the operation that replaces $u\,v$ with a path uzv, where z was not previously in G. All 3-edge-connected graphs can be constructed using a small set of operations starting from a K_2^3.

Theorem 1 (Mader [15]) *Every 3-edge-connected graph (and no other graph) can be constructed from a K_2^3 using the following three operations:*

- *Adding an edge (possibly parallel or a loop).*
- *Subdividing an edge $x\,y$ and connecting the new vertex to any existing vertex.*
- *Subdividing two distinct edges $w\,x$, $y\,z$ and connecting the two new vertices.*

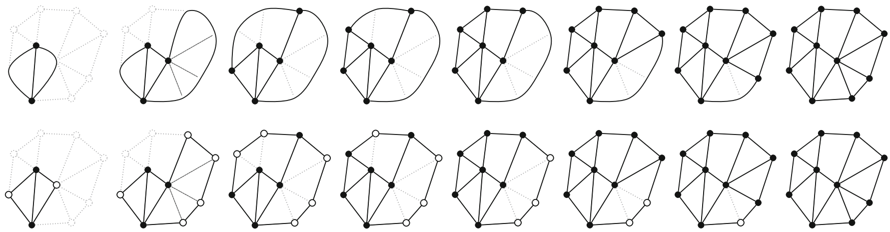

Fig. 1 Two ways of constructing the 3-edge-connected graph shown in the *rightmost column*. The *upper row* shows the construction according to Theorem 1. The *lower row* shows the construction according to Corollary 1. *Branch* (*non-branch*) *vertices* are depicted as *filled* (*non-filled*) *circles*. The *black edges* exist already, while *dotted gray vertices* and edges do not exist yet

A subdivision G' of a graph G is a graph obtained by subdividing edges of G zero or more times. The *branch vertices* of a subdivision are the vertices with degree at least three (we call the other vertices *non-branch*-vertices) and the *links* of a subdivision are the maximal paths whose inner vertices have degree two. If G has no vertex of degree two, the links of G' are in one-to-one correspondence to the edges of G. Theorem 1 readily generalizes to subdivisions of 3-edge-connected graphs.

Corollary 1 *Every subdivision of a* 3*-edge-connected graph (and no other graph) can be constructed from a subdivision of a* K_2^3 *using the following three operations:*

- *Adding a path connecting two branch vertices.*
- *Adding a path connecting a branch vertex and a non-branch vertex.*
- *Adding a path connecting two non-branch vertices lying on distinct links.*

In all three cases, the inner vertices of the path added are new vertices.

Each path that is added to a graph H in the process of Corollary 1 is called a *Mader-path* (*with respect to* H). Note that an ear is always a Mader-path unless both endpoints lie on the same link.

Figure 1 shows two constructions of a 3-edge-connected graph, one according to Theorem 1 and one according to Corollary 1. In this paper, we show how to find the Mader construction sequence according to Corollary 1 for a 3-edge-connected graph in linear time. Such a construction is readily turned into one according to Theorem 1.

4 Chain Decompositions

We use a very simple decomposition of graphs into cycles and paths. The decomposition was previously used for linear-time tests of 2-vertex- and 2-edge-connectivity [27] and 3-vertex-connectivity [26]. In this paper we show that it can also be used to find a Mader's construction for a 3-edge-connected graph. We define the decomposition algorithmically; a similar procedure that serves for the computation of low-points can be found in [24].

Let G be a connected graph without self-loops and let T be a depth-first search tree of G. Let r be the root of T. We orient tree-edges towards the root and back-edges

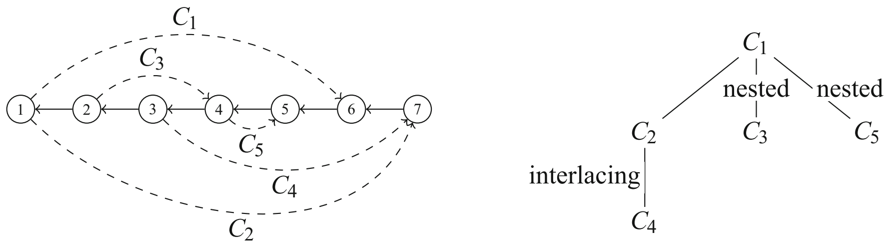

Fig. 2 The *left side* of the figure shows a DFS tree with a chain decomposition; tree-edges are *solid* and *back-edges* are *dashed*. C_1 is (1 6, 6 5, 5 4, 4 3, 3 2,2 1), C_2 is (1 7,7 6), C_3 is (2 4), C_4 is (3 7), and C_5 is (4 5). C_3 and C_5 are nested children of C_1 and C_4 is an interlacing child of C_2. Also, $s(C_4)$ s-belongs to C_1

away from the root, i.e., $v < u$ for an oriented tree-edge $u\,v$ and $x < y$ for an oriented back-edge $x\,y$.

We decompose G into a set $\mathscr{C} = \{C_1, \ldots, C_{|\mathscr{C}|}\}$ of cycles and paths, called *chains*, by applying the following procedure for each vertex v in the order in which they were discovered during the DFS: First, we declare v visited (initially, no vertex is visited), if not already visited before. Then, for every back-edge $v\,w$, we traverse the path $w \to_T r$ until a vertex x is encountered that was visited before; x is a descendant of v. The traversed subgraph $v\,w \cup (w \to_T x)$ forms a new *chain* C with $s(C) = v$ and $t(C) = x$. All inner vertices of C are declared visited. Observe that $s(C)$ and $t(C)$ are already visited when the construction of the chain starts.

Figure 2 illustrates these definitions. Since every back-edge defines one chain, there are precisely $m-n+1$ chains. We number the chains in the order of their construction.

We call $\mathscr{C}$ a *chain decomposition*. It can be computed in time $O(n + m)$. For 2-edge-connected graphs the term decomposition is justified by Lemma 1.

Lemma 1 [27] *Let $\mathscr{C}$ be a chain decomposition of a graph G. Then G is 2-edge-connected if and only if G is connected and the chains in $\mathscr{C}$ partition $E(G)$.*

Since the condition of Lemma 1 is easily checked during the chain decomposition, we assume from now on that G is 2-edge-connected. Then $\mathscr{C}$ partitions $E(G)$ and the first chain C_1 is a cycle containing r (since there is a back-edge incident to r). We say that r *strongly belongs (s-belongs)* to the first chain and any vertex $v \neq r$ *s-belongs* to the chain containing the edge $v\,p(v)$. We use s-belongs instead of belongs since a vertex can belong to many chains when chains are viewed as sets of vertices.

We can now define a parent-tree on chains. The first chain C_1 is the root. For any chain $C \neq C_1$, let the *parent* $p(C)$ of C be the chain to which $t(C)$ s-belongs. We write $C \leq D$ ($C < D$) for chains C and D if C is an ancestor (proper ancestor) of D in the parent-tree on chains.

The following lemma summarizes important properties of chain decompositions.

Lemma 2 *Let $\{C_1, \ldots, C_{m-n+1}\}$ be a chain decomposition of a 2-edge-connected graph G and let r be the root of the DFS-tree. Then*

(1) For every chain C_i, $s(C_i) \leq t(C_i)$.

(2) *Every chain C_i, $i \geq 2$, has a parent chain $p(C_i)$. We have $s(p(C_i)) \leq s(C_i)$ and $p(C_i) = C_j$ for some $j < i$.*
(3) *For $i \geq 2$: If $t(C_i) \neq r$, $t(p(C_i)) < t(C_i)$. If $t(C_i) = r$, $t(p(C_i)) = t(C_i)$.*
(4) *If $u \leq v$, u s-belongs to C, and v s-belongs to D then $C \leq D$.*
(5) *If $u \leq t(D)$ and u s-belongs to C, then $C \leq D$.*
(6) *For $i \geq 2$: $s(C_i)$ s-belongs to a chain C_j with $j < i$.*

Proof (1)–(3) follow from the discussion preceding the Lemma and the construction of the chains. We turn to (4). Consider two vertices u and v with $u \leq v$ and let u s-belong to C and let v s-belong to D. Then $C \leq D$, as the following simple induction on the length of the tree path from u to v shows. If $u = v$, $C = D$ by the definition of s-belongs. So assume u is a proper ancestor of v. Since v s-belongs to D, by definition $v \neq t(D)$ and $v\,p(v)$ is contained in D. Let D' be the chain to which $p(v)$ s-belongs. By induction hypothesis, $C \leq D'$. Also, either $D = D'$ (if $p(v)$ s-belongs to D) or $D' = p(D)$ (if $p(v) = t(D)$) and hence $p(v)$ s-belongs to $p(D)$. In either case $C \leq D$.

Claim (5) is an easy consequence of (4). If $t(D) = r$, $C = C_1$, and the claim follows. If $t(D) \neq r$, $t(D)$ s-belongs to $p(D)$. Thus, $C \leq p(D)$ by (4).

The final claim is certainly true for each C_i with $s(C_i) = r$. So assume $s(C_i) > r$ and let $y = p(s(C_i))$. Since G is 2-edge-connected, there is a back-edge $u\,v$ with $u \leq y$ and $s(C_i) \leq v$. It induces a chain C_k with $k < i$ and hence $s(C_i)y$ is contained in a chain C_j with $j \leq k$. □

5 Chains as Mader-Paths

We show that, assuming that the input graph is 3-edge-connected, there are two chains that form a subdivision of a K_2^3, and that the other chains of the chain decomposition can be added one by one such that each chain is a Mader-path with respect to the union of the previously added chains. We will also show that chains can be added parent-first, i.e., when a chain is added, its parent was already added. In this way the current graph G_c consisting of the already added chains is always *parent-closed*. We will later show how to compute this ordering efficiently. We will first give an $O((n+m)\log(n+m))$ algorithm and then a linear time algorithm.

Using the chain decomposition, we can identify a K_2^3 subdivision in the graph as follows. We may assume that the first two back-edges explored from r in the DFS have their other endpoint in the same subtree T' rooted at some child of r. The first chain C_1 forms a cycle. The vertices in $C_1 \backslash r$ are then contained in T'. By assumption, the second chain is constructed by another back-edge that connects r with a vertex in T'. If there is no such back-edge, the tree edge connecting r and the root of T' and the back edge from r into T' form a 2-edge cut. Let $x = t(C_2)$. Then $C_1 \cup C_2$ forms a K_2^3 subdivision with branch vertices r and x. The next lemma derives properties of parent-closed unions of chains.

Lemma 3 *Let G_c be a parent-closed union of chains that contains C_1 and C_2. Then*

(1) *For any vertex $v \neq r$ of G_c, the edge $v\,p(v)$ is contained in G_c, i.e., the set of vertices of G_c is a parent-closed subset of the DFS-tree.*

(2) $s(C)$ and $t(C)$ are branch vertices of G_c for every chain C contained in G_c.
(3) Let C be a chain that is not in G_c but a child of some chain in G_c. Then C is an ear with respect to G_c and the path $t(C) \rightarrow_T s(C)$ is contained in G_c. C is a Mader-path (i.e., the endpoints of C are not inner vertices of the same link of G_c) with respect to G_c if and only if there is a branch vertex on $t(C) \rightarrow_T s(C)$.

Proof (1): Let $v \neq r$ be any vertex of G_c. Let C be a chain in G_c containing the vertex v. If C also contains $v\, p(v)$ we are done. Otherwise, $v = t(C)$ or $v = s(C)$. In the first case, v s-belongs to $p(C)$, in the second case v s-belongs to some $C' \leq C$ by Lemma 2(4). Hence, by parent-closedness, $v\, p(v)$ is an edge of G_c.

(2): Let C be any chain in G_c. Since C_1 and C_2 form a K_2^3, r and $x = t(C_2)$ are branch vertices. If $s(C) \neq r$, the edge $s(C)\, p(s(C))$ is in G_c by (1), the back-edge $s(C)\, v$ inducing C is in G_c, and the path $v \rightarrow_T s(C)$ is in G_c by (1). Thus $s(C)$ has degree at least three. If $t(C) \notin \{r, x\}$, let $\widehat{C}$ be the chain to which $t(C)$ s-belongs, i.e. $\widehat{C}$ is the parent of C. As G_c is parent-closed $\widehat{C}$ is contained in G_c. By the definition of s-belongs, $t(C)$ has degree two on the chain $\widehat{C}$. Further, it has degree one on the chain C. Since chains are edge-disjoint, it has degree at least three in G_c.

(3) We first observe that $t(C)$ and $s(C)$ belong to G_c. For $t(C)$, this holds since $t(C)$ s-belongs to $p(C)$ and $p(C)$ is part of G_c by assumption. For $s(C)$, this follows from $s(C) \leq t(C)$ and (1). No inner vertex u of C belongs to G_c, because otherwise the edge $u\, p(u)$ would belong to G_c by (1), which implies that C would belong to G_c, as G_c is a union of chains. Thus C is an ear with respect to G_c, i.e., it is disjoint from G_c except for its endpoints. Moreover, the path $t(C) \rightarrow_T s(C)$ belongs to G_c by (1).

If there is no branch vertex on $t(C) \rightarrow_T s(C)$, the vertices $t(C)$ and $s(C)$ are inner vertices of the same link of G_c and hence C is not a Mader-path with respect to G_c. If there is a branch vertex on $t(C) \rightarrow_T s(C)$, the vertices $t(C)$ and $s(C)$ are inner vertices of two distinct links of G_c and hence C is a Mader-path with respect to G_c. □

We can now prove that chains can always be added in parent-first order. For a link L, each edge in L that is incident to an end vertex of L is called an *extremal* edge of L.

Theorem 2 *Let G be a graph and let G_c be a parent-closed union of chains such that no child of a chain $C \in G_c$ is a Mader-path with respect to G_c and there is at least one such chain. Then the extremal edges of every link of length at least two in G_c are a 2-cut in G.*

Proof Assume otherwise. Then there is a parent-closed union G_c of chains such that no child of a chain in G_c is a Mader-path with respect to G_c and there is at least one such chain outside of G_c, but for every link in G_c the extremal edges are not a cut in G.

Consider any link L of G_c. Since the extremal edges of L do not form a 2-cut, there is a path in $G - G_c$ connecting an inner vertex on L with a vertex that is either a branch vertex of G_c or a vertex on a link of G_c different from L. Let P be such a path of minimum length. By minimality, no inner vertex of P belongs to G_c. Note that P is a Mader-path with respect to G_c. We will show that at least one edge of P belongs to a chain C with $p(C) \in G_c$ and that C can be added, contradicting our choice of G_c.

Let a and b be the endpoints of P, and let z be the lowest common ancestor of all points in P. Since a DFS generates only tree- and back-edges, z lies on P. Since $z \leq a$ and the vertex set of G_c is a parent-closed subset of the DFS-tree, z belongs to G_c. Thus z cannot be an inner vertex of P and hence is equal to a or b. Assume w.l.o.g. that $z = a$. All vertices of P are descendants of a. We view P as oriented from a to b.

Since b is a vertex of G_c, the path $b \rightarrow_T a$ is part of G_c by Lemma 2 and hence no inner vertex of P lies on this path. Let $a\,v$ be the first edge on P. The vertex v must be a descendant of b as otherwise the path $v \rightarrow_P b$ would contain a cross-edge, i.e. an edge between different subtrees. Hence $a\,v$ is a back-edge. Let D be the chain that starts with the edge $a\,v$. D does not belong to G_c, as no edge of P belongs to G_c.

We claim that $t(D)$ is a proper descendant of b or D is a Mader-path with respect to G_c. Since v is a descendant of b and $t(D)$ is an ancestor of v, $t(D)$ is either a proper descendant of b, equal to b, or a proper ancestor of b. We consider each case separately.

If $t(D)$ were a proper ancestor of b the edge $b\,p(b)$ would belong to D and hence D would be part of G_c, contradicting our choice of P. If $t(D)$ is equal to b then D is a Mader-path with respect to G_c. This leaves the case that $t(D)$ is a proper descendant of b.

Let $y\,x$ be the last edge on the path $t(D) \rightarrow_T b$ that is not in G_c and let D^* be the chain containing $y\,x$. Then $D^* \leq D$ by Lemma 2(5) (applied with $C = D^*$ and $u = y$) and hence $s(D^*) \leq s(D) \leq a$ by part (4) of the same lemma. Also $t(D^*) = x$. Since $x = t(D^*) \in G_c$, $p(D^*) \in G_c$.

As a and b are not inner vertices of the same link, the path $x = t(D^*) \rightarrow_T b \rightarrow_T a \rightarrow_T s(D^*)$ contains a branch vertex. Thus D^* is a Mader-path by Lemma 3. □

Corollary 2 *If G is 3-edge-connected, chains can be greedily added in parent-first order.*

Proof If we reach a point where not all chains are added, but we can not proceed in a greedy fashion, by Theorem 2 we find a cut in G. □

6 A First Algorithm

Corollary 2 gives rise to an $O((n+m)\log(n+m))$ algorithm, the Greedy-Chain-Addition Algorithm. In addition to G, we maintain the following data structures:

- The current graph G_c. Each link is maintained as a doubly linked list of vertices. Observe that all inner vertices of a link lie on the same tree path and hence are numbered in decreasing order. The vertices in G are labeled *inactive*, *branch*, or *non-branch*. The vertices in $G \backslash G_c$ are called *inactive*. Every non-branch vertex stores a pointer to the link on which it lies and a list of all chains incident to it and having the other endpoint as an inner vertex of the same link.
- A list $\mathscr{L}$ of addable chains. A chain is addable if it is a Mader-path with respect to the current graph.
- For each chain its list of children.

We initialize G_c to $C_1 \cup C_2$. It has three links, $t(C_2) \rightarrow_T r$, $r \rightarrow_{C_1} t(C_2)$, and $r \rightarrow_{C_2} t(C_2)$. We then iterate over the children of C_1 and C_2. For each child, we check in constant time whether its endpoints are inner vertices of the same link. If so, we associate the chain with the link by inserting it into the lists of both endpoints. If not, we add the chain to the list of addable chains. The initialization process takes time $O(n+m)$.

As long as the list of addable chains is non-empty, we add a chain, say C. Let u and v be the endpoints of C. We perform the following actions:

- If u is a non-branch vertex, we make it a branch vertex. This splits the link containing it and entails some processing of the chains having both endpoints on this link.
- If v is a non-branch vertex, we make it a branch vertex. This splits the link containing it, and entails some processing of the chains having both endpoints on this link.
- We add C as a new link to G_c.
- We process the children of C.

We next give the details for each action.

If u is a non-branch vertex, it becomes a branch vertex. Let L be the link of G_c containing u; L is split into links L_1 and L_2 and the set S of chains having both endpoints on L is split into sets S_1, S_2 and S_{add}, where S_i is the set of chains having both endpoints on L_i, $i = 1, 2$, and S_{add} is the set of chains that become addable (because they are incident to u or have one endpoint each in L_1 and L_2). We show that we can perform the split of L in time $O(1 + |S_{\text{add}}| + \min(|L_1| + |S_1|, |L_2| + |S_2|))$. We walk from both ends of L towards u in lockstep fashion. In each step we either move to the next vertex or consider one chain. Once we reach u we stop. Observe that this strategy guarantees the time bound claimed above.

When we consider a chain, we check whether we can move it to the set of addable chains. If so, we do it and delete the chain from the lists of both endpoints. Once, we have reached u, we split the list representing the link into two. The longer part of the list retains its identity, for the shorter part we create a new list header and redirect all pointers of its elements.

Adding C to G_c is easy. We establish a list for the new link and let all inner vertices of C point to it. The inner vertices become active non-branch vertices.

Processing the children of C is also easy. For each child, we check whether both endpoints are inner vertices of C. If so, we insert the child into the list of its endpoints. If not, we add the child to the list of addable chains.

If $\mathscr{L}$ becomes empty, we stop. If all chains have been added, we have constructed a Mader sequence. If not all chains have been processed, there must be a link having at least one inner vertex. The first and the last edge of this link form a 2-edge-cut.

It remains to argue that the algorithm runs in time $O((n+m)\log(n+m))$. We only need to argue about the splitting process. We distribute the cost $O(1 + |S_{\text{add}}| + \min(|L_1| + |S_1|, |L_2| + |S_2|))$ as follows: $O(1)$ is charged to the vertex that becomes a branch vertex. All such charges add up to $O(n)$. $O(|S_{\text{add}}|)$ is charged to the chains that become addable. All such charges add up to $O(m)$. $O(\min(|L_1| + |S_1|, |L_2| + |S_2|))$ is charged to the vertices and chains that define the minimum. We account for these charges with the following token scheme inspired by the analysis of the corresponding recurrence relation in [17].

Consider a link L with k chains having both endpoints on L. We maintain the invariant that each vertex and chain owns at least $\log(|L|+k)$ tokens. When a link is newly created we give $\log(n+m)$ tokens to each vertex of the link and to each chain having both endpoints on the link. In total we create $O((n+m)\log(n+m))$ tokens. Assume now that we split a link L with k chains into links L_1 and L_2 with k_1 and k_2 chains respectively. Then $\min(|L_1|+k_1, |L_2|+k_2) \leq (|L|+k)/2$ and hence we may take one token away from each vertex and chain of the sublink that is charged without violating the token invariant.

Theorem 3 *The Greedy-Chain-Addition algorithm runs in time $O((n+m)\log(n+m))$.*

7 A Classification of Chains

When we add a chain in the Greedy-Chain-Addition algorithm, we also process its children. Children that do not have both endpoints as inner vertices of the chain can be added to the list of addable chains immediately. However, children that have both endpoints as inner vertices of the chain cannot be added immediately and need to be observed further until they become addable. We now make this distinction explicit by classifying chains into two types, interlacing and nested.

We classify the chains $\{C_3, \ldots C_{m-n+1}\}$ into two types. Let C be a chain with parent $\widehat{C} = p(C)$. We distinguish two cases[1] for C.

- If $s(C)$ is an ancestor of $t(\widehat{C})$ and a descendant of $s(\widehat{C})$, C is *interlacing*. We have $s(\widehat{C}) \leq s(C) \leq t(\widehat{C}) \leq t(C)$.
- If $s(C)$ is a proper descendant of $t(\widehat{C})$, C is *nested*. We have $s(\widehat{C}) \leq t(\widehat{C}) < s(C) \leq t(C)$ and $t(C) \rightarrow_T s(C)$ is contained in $\widehat{C}$.

These cases are exhaustive as the following argument shows. Let $s(\widehat{C})\, v$ be the first edge on $\widehat{C}$. By Lemma 2, $s(\widehat{C}) \leq s(C) \leq v$. We split the path $v \rightarrow_T s(\widehat{C})$ into $t(\widehat{C}) \rightarrow_T s(\widehat{C})$ and $(v \rightarrow_T t(\widehat{C})) \setminus t(\widehat{C})$. Depending on which of these paths $s(C)$ lies on, C is interlacing or nested.

The following simple observations are useful. For any chain $C \neq C_1$, $t(C)$ s-belongs to $\widehat{C}$. If C is nested, $s(C)$ and $t(C)$ s-belong to $\widehat{C}$. If C is interlacing, $s(C)$ s-belongs to a chain which is a proper ancestor of $\widehat{C}$ or $\widehat{C} = C_1$. The next lemma confirms that interlacing chains can be added once their parent belongs to G_c.

Lemma 4 *Let G_c be a parent-closed union of chains that contains C_1 and C_2, let C be any chain contained in G_c, and let D be an interlacing child of C not contained in G_c. Then D is a Mader-path with respect to G_c.*

Proof We have already shown in Lemma 3 that D is an ear with respect to G_c, that the path $t(D) \rightarrow_T s(D)$ is part of G_c, and that $s(C)$ and $t(C)$ are branching vertices of G_c.

[1] In [26], three types of chains are distinguished. What we call nested is called Type 1 there and what we call interlacing is split into Types 2 and 3 there. We do not need this finer distinction.

Algorithm 1 Certifying linear-time algorithm for 3-edge connectivity.

```
procedure CONNECTIVITY(G=(V,E))
    Let {C_1, C_2, ..., C_{m-n+1}} be a chain decomposition of G as described in Sect. 4;
    Initialize G_c to C_1 ∪ C_2;
    for i from 1 to m − n + 1 do          ▷ Phase i: add all chains whose source s-belongs to C_i
        Group the chains C for which s(C) s-belongs to C_i into segments;
        Part I of Phase i: Add all segments to G_c whose minimal chain is interlacing;
        Part II of Phase i: Either find an insertion order S_1, ..., S_k of the segments having a nested
            minimal chain or exhibit a 2-edge-cut and stop;
        for j from 1 to k do
            Add the chains contained in S_j parent-first;
        end for
    end for
end procedure
```

Since D is interlacing, we have $s(C) \leq s(D) \leq t(C) \leq t(D)$. Thus $t(D) \rightarrow_T s(D)$ contains a branching vertex and hence D is a Mader-path by Lemma 3(3). □

8 A Linear Time Algorithm

According to Lemma 4, interlacing chains whose parent belongs to the current graph are always Mader-paths and can be added. Nested chains have both endpoints on their parent chain and can only be added once the tree-path connecting its endpoints contains a branching point. Consider a chain nested in chain C_i. Which chains can help its addition by creating branching points on C_i? First, interlacing chains having their source on some C_j with $j \leq i$, and second, chains nested in C_i and their interlacing offspring having their source on C_i. Chains having their source on some C_j with $j > i$ cannot help because they have no endpoint on C_i. This observation shows that chains can be added in phases. In the i-th phase, we try to add all chains having their source vertex on C_i.

The overall structure of the linear-time algorithm is given in Algorithm 1. An implementation in Python is available at https://github.com/adrianN/edge-connectivity. The algorithm operates in phases and maintains a current graph G_c. Let $C_1, C_2, \ldots, C_{m-n+1}$ be the chains of the chain decomposition in the order of creation. We initialize G_c to $C_1 \cup C_2$. In phase i, $i \in [1, m-n+1]$, we consider the i-th chain C_i and either add all chains C to G_c for which the source vertex $s(C)$ s-belongs to C_i to G_c or exhibit a 2-edge-cut. As already mentioned, chains are added parent-first and hence G_c is always parent-closed. We maintain the following invariant:

Invariant: After phase i, G_c consists of all chains for which the source vertex s-belongs to one of the chains C_1 to C_i.

Lemma 5 *For all i, the current chain C_i is part of the current graph G_c at the beginning of phase i or the algorithm has exhibited a 2-edge-cut before phase i.*

Proof The initial current graph consists of chains C_1 and C_2 and hence the claim is true for the first and the second phase. Consider $i > 2$. The source vertex $s(C_i)$ s-belongs to a chain C_j with $j < i$ (Lemma 2(6)) and hence C_i is added in phase j. □

The next lemma gives information about the chains for which the source vertex s-belongs to C_i. None of them belongs to G_c at the beginning of phase i (except for chain C_2 that belongs to G_c at the beginning of phase 1) and they form subtrees of the chain tree. Only the roots of these subtrees can be nested. All other chains are interlacing.

Lemma 6 *Assume that the algorithm reaches phase i without exhibiting a 2-edge-cut. Let $C \neq C_2$ be a chain for which $s(C)$ s-belongs to C_i. Then C is not part of G_c at the beginning of phase i. Let D be any ancestor of C that is not in G_c. Then:*

(1) $s(D)$ s-belongs to C_i.
(2) If D is nested, it is a child of C_i.
(3) If $p(D)$ is not part of the current graph, D is interlacing.

Proof We use induction on i. Consider the i-th phase and let $C \neq C_2$ be chains whose source vertex $s(C)$ s-belongs to C_i. We first prove that C is not in G_c. This is obvious, since in the j-th phase we add exactly the chains whose source vertex s-belongs to C_j.

(1): Let D be any ancestor of C which is not part of G_c. By Lemma 2, we have $s(D) \leq s(C)$ and hence $s(D)$ belongs to C_j for some $j \leq i$. If $j < i$, D would have been added in phase j, a contradiction to the assumption that D does not belong to G_c at the beginning of phase i.

(2): $s(D)$ s-belongs to C_i by (1). If D is nested, $s(D)$ and $t(D)$ s-belong to the same chain. Thus D is a child of C_i.

(3): If $p(D)$ is not part of the current graph, $p(D) \neq C_i$ by Lemma 5 and hence D is not a child of C_i. Hence by (2), D is interlacing. □

We can now define the segments with respect to C_i by means of an equivalence relation. Consider the set $\mathscr{S}$ of chains whose source vertex s-belongs to C_i. For a chain $C \in \mathscr{S}$, let C^* be the minimal ancestor of C that does not belong to G_c. Two chains C and D in $\mathscr{S}$ belong to the same segment if and only if $C^* = D^*$. In Fig. 2 on page 5, if we start with $G_c = C_1 \cup C_2$, we form three segments in the first phase, namely $\{C_4\}$, $\{C_3\}$, and $\{C_5\}$. The first segment can be added according to Lemma 4. Then C_3 can be added and then C_5.

Consider any $C \in \mathscr{S}$. By part (1) of the preceding lemma either $p(C) \in \mathscr{S}$ or $p(C)$ is part of G_c. Moreover, C and $p(C)$ belong to the same segment in the first case. Thus segments correspond to subtrees in the chain tree. In any segment only the minimal chain can be nested by Lemma 6. If it is nested, it is a child of C_i (parts (2) and (3) of the preceding lemma). Since only the root of a segment may be a nested chain, once it is added to the current graph all other chains in the segment can be added in parent-first order by Lemma 4. All that remains is to find the proper ordering of the segments faster than in the previous section. We do so in Lemma 10. If no proper ordering exists, we exhibit a 2-edge-cut.

Lemma 7 *All chains in a segment S can be added in parent-first order if its minimal chain can be added.*

Proof By Lemma 6 all but the minimal chain in a segment are interlacing. Thus the claim follows from Lemma 4. □

We come to part I of phase i, the addition of all segments whose minimal chain is interlacing. As a byproduct, we will also determine all segments with nested minimal chain. We iterate over all chains C whose source $s(C)$ s-belongs to C_i. For each such chain, we traverse the path C, $p(C)$, $p(p(C))$, …until we reach a chain that belongs to G_c or is already marked (initially, all chains are unmarked). We now distinguish cases. If the last chain on the path is nested we mark all chains on the path with the nested chain. If we hit a marked chain we copy the marker to all chains in the path. Otherwise, i.e., all chains are interlacing and unmarked, we add all chains in the path to G_c in parent-first order, as these segments can be added according to Corollary 7.

It remains to compute a proper ordering of the segments in which the minimal chain is nested or to exhibit a 2-edge-cut. We do so in part II of phase i. For simplicity, we will say 'segment' instead of 'segment with nested minimal chain' from now on.

For a segment S let the *attachment points* of S be all vertices in S that are in G_c. Note that the attachment points must necessarily be endpoints of chains in S and hence adding the chains of S makes the attachment points branch vertices. Nested children C of C_i can be added if there are branch vertices on $t(C) \rightarrow_T s(C)$, therefore adding a segment can make it possible to add further segments.

Lemma 8 *Let C be a nested child of C_i and let S be the segment containing C. The attachment points of S consist of $s(C)$, $t(C)$, and the vertices $s(D)$ of the other chains in the segment. All such points lie on the path $t(C) \rightarrow_T s(C)$ and hence on C_i.*

Proof Let D be any chain in S different from C. By Lemma 6, C is the minimal chain in S. Since S is a subtree of the chain tree, we have $C < D$ and hence by Lemma 2 $t(C) \leq t(D)$. Since none of the chains in S is part of G_c, parent-closedness implies that no vertex on the path $(t(D) \rightarrow_T t(C)) \backslash t(C)$ belongs to G_c. In particular, either $t(D) = t(C)$ or $t(D)$ is not a vertex of G_c and hence not an attachment point of S. It remains to show $s(C) \leq s(D) \leq t(C)$. Since $C \leq D$, we have $s(C) \leq s(D)$ by Lemma 2. Since $s(D) \leq t(D)$ and $t(C) \leq t(D)$ we have either $s(D) \leq t(C) \leq t(D)$ or $t(C) < s(D) \leq t(D)$. In the former case, we are done. In the latter case, $s(D)$ is not a vertex of G_c by the preceding paragraph, a contradiction, since $s(D)$ s-belongs to C_i by Lemma 6. □

For a set of segments $S_1, \ldots, S_k$, let the *overlap graph* be the graph on the segments and a special vertex R for the branch vertices on C_i. In the overlap graph, there is an edge between R and a vertex S_i, if there are attachment points $a_1 \leq a_2$ of S_i such that there is a branch vertex on the tree path $a_2 \rightarrow_T a_1$. Further, between two vertices S_i and S_j there is an edge if there are attachment points a_1, a_2 in S_i and b_1, b_2 in S_j, such that $a_1 \leq b_1 \leq a_2 \leq b_2$ or $b_1 \leq a_1 \leq b_2 \leq a_2$. We say that S_i and S_j *overlap*.

Lemma 9 *Let $\mathscr{C}$ be a connected component of the overlap graph H and let S be any segment with respect to C_i whose minimal chain C is nested. Then $S \in \mathscr{C}$ if and only if*

(i) $R \in \mathscr{C}$ and there is a branch vertex on $t(C) \rightarrow_T s(C)$ or
(ii) there are attachments a_1 and a_2 of S and attachments b_1 and b_2 of segments in $\mathscr{C}$ with $a_1 \leq b_1 \leq a_2 \leq b_2$ or $b_1 \leq a_1 \leq b_2 \leq a_2$.

Proof We first show $S \in \mathscr{C}$ if (i) or (ii) holds. For (i) the claim follows directly from the definition of the overlap graph. For (ii), assume $S \notin \mathscr{C}$ for the sake of a contradiction. Then either $R \notin \mathscr{C}$ or there is no branch vertex in $t(C) \to_T s(C)$ by (i). Further, no segment in $\mathscr{C}$ overlaps with S and hence any segment in $\mathscr{C}$ has its attachments points either strictly between a_1 and a_2 or outside the path $a_2 \to_T a_1$. Moreover, both classes of segments are non-empty. However, segments in the two classes do not overlap and R cannot be connected to the segments in the former class. Thus $\mathscr{C}$ is not connected, a contradiction.

If neither (i) nor (ii) hold, there can be no segment in $\mathscr{C}$ overlapping S and either S is not connected to R or no segment in $\mathscr{C}$ is connected to R. □

Lemma 10 *Assume the algorithm reaches phase i. If the overlap graph H induced by the segments with respect to C_i is connected, we can add all segments of C_i. If H is not connected, we can exhibit a 2-edge-cut for any component of H that does not contain R.*

Proof Assume first that H is connected. Let $R, S_1, \ldots, S_k$ be the vertices of H in a preorder, e.g. the order they are explored by a DFS, starting at R, the vertex corresponding to the branch vertices on C_i. An easy inductive argument shows that we can add all segments in this order. Namely, let $k \geq 1$ and let C be the minimal chain of S_k. All attachment points of S_k lie on the path $t(C) \to_T s(C)$ by Lemma 8, and there is either an edge between R and S_k or an edge between S_j and S_k for some $j < k$. In the former case, there is a branch vertex on $t(C) \to_T s(C)$ at the beginning of the phase, in the latter case there is one after adding S_j. Thus the minimal chain of S_k can be added and then all other chains by Lemma 7.

On the other hand, suppose H is not connected. Let $\mathscr{C}$ be any connected component of H that does not contain R, and let $\mathscr{C}_R$ be the connected component that contains R. Let x and y be the minimal and maximal attachment points of the segments in $\mathscr{C}$, and let G_c be the current graph after adding all chains in $\mathscr{C}_R$. We first show that there is no branch vertex of G_c on the path $y \to_T x$. Assume otherwise and let w be any such branch vertex. Observe first that there must be a chain $C \in \mathscr{C}$ with $s(C) \leq w \leq t(C)$. Otherwise, every chain in $\mathscr{C}$ has all its attachment points at proper ancestors of w or at proper descendants of w and hence $\mathscr{C}$ is not connected. Let S be the segment containing C. By Lemma 8, we may assume that C is the minimal chain of S. Since $S \notin \mathscr{C}_R$, $R\,S$ is not an edge of H and hence no branch vertex exists on the path $t(C) \to_T s(C)$ at the beginning of part II of the phase. Hence w is an attachment point of a segment in $\mathscr{C}_R$. In particular $\mathscr{C}_R$ contains at least one segment. We claim that $\mathscr{C}_R$ must also have an attachment point outside $t(C) \to_T s(C)$. This holds since all initial branch vertices are outside the path and since $\mathscr{C}_R$ is connected. Thus $S \in \mathscr{C}_R$ by Lemma 9, a contradiction.

We show next that the tree-edge $x\,p(x)$ and the edge $z\,y$ from y's predecessor z on C_i to y form a 2-edge-cut; $z\,y$ may be a tree-edge or a back-edge. The following argument is similar to the argument in Theorem 2, but more refined.

Assume otherwise. Then, as in the proof of Theorem 2, there is a path $P = a \to b$ such that $a \leq u$ for all $u \in P$, and either a lies on $y \to_T x$ and b does not, or vice versa, and no inner vertex of P is in G_c. Moreover, the first edge $a\,v$ of P is a back-edge and v is a descendant of b. Note that unlike in the proof of Theorem 2, a

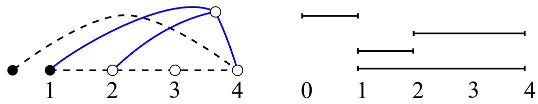

Fig. 3 Intervals for the *solid segment* with respect to the *dashed chain*. It has the attachment points 1, 2, 4. *Filled vertices* are branching points

and b need not lie on different links, as we want to show that $x\,p(x)$ and $z\,y$ form a cut and these might be different from the last edges on the link containing x and y.

Let D be the chain that starts with the edge $a\,v$. D does not belong to G_c, as no edge of P belongs to G_c. In particular, a does not s-belong to C_j for $j < i$ (as otherwise, D would already be added). Since $a \le b$ and one of a and b lies on $y \to_T x$ (which is a subpath of C_i), a s-belongs to C_i. By the argument from the proof of Theorem 2, $t(D)$ is a descendant of b.

Let D^* be the chain that contains the last edge of P. If $t(D) = b$, $D = D^*$. Otherwise, $t(D)$ is a proper descendant of b. Let $u\,b$ be the last edge on the path $t(D) \to_T b$. We claim that $u\,b$ is also the last edge of P. This holds since the last edge of P must come from a descendant of b (as ancestors of b belong to G_c) and since it cannot come from a child different from y as otherwise P would have to contains a cross-edge. Thus $D^* \le D$ by Lemma 2(5) and hence $s(D^*) \le s(D) \le a$ by part (4) of the same lemma.

D and D^* belong to the same segment with respect to C_i, say S, and a and b are vertices in $S \cap G_c$. This can be seen easily. Since a s-belongs to C_i, D belongs to some segment with respect to C_i and since $D^* \le D$, D^* belongs to the same segment. Since $t(D^*) = b$ and b is a vertex of G_c, D^* is the minimal chain in S. Thus D^* is nested and hence b s-belongs to C_i. Hence a and b are attachment points of S.

Thus S overlaps with $\mathscr{C}$ and hence $S \in \mathscr{C}$ by Lemma 9. Therefore x and y are not the extremal attachment points, that is the minimal (or maximal) vertices in $S \cap G_c$, of $\mathscr{C}$, a contradiction. □

It remains to show that we can find an order as required in Lemma 10, or a 2-edge-cut, in linear time. We reduce the problem of finding an order on the segments to a problem on intervals. W.l.o.g. assume that the vertices of C_i are numbered consecutively from 1 to $|C_i|$. Consider any segment S, and let $a_0 \le a_1 \le \ldots \le a_k$ be the set of attachment points of S, i.e., the set of vertices that S has in common with C_i. By Lemma 8, a_0 and a_k are the endpoints of the minimal chain in S and each a_i, $0 < i < k$, is equal to $s(D)$ for some other chain in S. We associate the intervals

$$\{[a_0, a_\ell] | 1 \le \ell \le k\} \cup \{[a_\ell, a_k] | 1 \le \ell < k\},$$

with S and for every branch vertex v on C_i we define an interval $[0, v]$. See Fig. 3 for an example.

We say two intervals $[a, a']$, $[b, b']$ *overlap* if $a \le b \le a' \le b'$. Note that overlapping is different from intersecting; an interval does not overlap intervals in which it is properly contained or which it properly contains. This relation naturally induces a graph H' on the intervals. Contracting all intervals that are associated to the same segment into one vertex makes H' isomorphic to the overlap graph as required for Lemma 10. Hence we can use H' to find the order on the segments. Note that the interval set $\{[a_0, a_\ell]|1 \leq \ell \leq k\}$ for each segment does not suffice without employing a clever tie-breaking rule: If there are two segments with attachments $a < b < c$ and $a < b' < c$, respectively, such that $b' \neq b$, no interval of the first segment overlaps with one of the second.

A naive approach that constructs H', contracts intervals, and runs a DFS will fail, since the overlap graph can have a quadratic number of edges. However, using a method developed by Olariu and Zomaya [23], we can compute a spanning forest of H' in time linear in the number of intervals. The presentation in [23] is for the PRAM and thus needlessly complicated for our purposes. A simpler explanation can be found in the Appendix.

The number of intervals created for a chain C_i is bounded by

$$|\text{NestedChildren}(C_i)| + 2|\text{Interlacing}(C_i)| + |V_{\text{branch}}(C_i)|,$$

where $\text{NestedChildren}(C_i)$ are the nested children of C_i, $\text{Interlacing}(C_i)$ are the interlacing chains that start on C_i, and $V_{\text{branch}}(C_i)$ is the set of branch vertices on C_i. Note that we generate the interval $[s(C), t(C)]$ for each nested child C, and the intervals $[s(C), s(D)]$ and $[s(D), t(C)]$ for each interlacing chain D belonging to a segment with nested minimal chain C. Thus the total time spend the ordering procedure is $O(m)$. From the above discussion, we get:

Theorem 4 *For a 3-edge-connected graph, a Mader construction sequence can be found in time $O(n+m)$.*

9 Verifying the Mader Sequence

The certificate is either a 2-edge-cut, or a sequence of Mader-paths. For a 2-edge-cut, we simply remove the two edges and verify that G is no longer connected.

For checking the Mader sequence, we doubly-link each edge in a Mader-path to the corresponding edge in G. Let G' be a copy of G. We remove the Mader-paths, in reverse order of the sequence, suppressing vertices of degree two as they occur. This can create multiple edges and loops. Let G'_i be the multi-graph before we remove the i-th path P_i. We need to verify the following:

- G must have minimum degree three.
- The union of Mader-paths must be isomorphic to G and the Mader-paths must partition the edges of G. This is easy to check using the links between the edges of the paths and the edges of G.
- The paths we remove must be ears. More precisely, at step i, P_i must have been reduced to a single edge in G'_i, as inner vertices of P_i must have been suppressed if P_i is an ear for G'_i.
- The P_i must not subdivide the same link twice. That is, after deleting the edge corresponding to P_i, it must not be the case that both endpoints are still adjacent (or equal, i.e. P_i is a loop) but have degree two.

- When only two paths are left, the graph must be a K_2^3.

10 The Cactus Representation of 2-Cuts

We review the cactus representation of 2-cuts in a 2-connected but not 3-connected graph and show how to certify it.

A *cactus* is a graph in which every edge is contained in exactly one cycle. Dinits, Karzanov, and Lomonsov [5] showed that the set of mincuts of any graph has a cactus representation, i.e., for any graph G there is a cactus C and a mapping $\phi : V(G) \to V(C)$ such that the mincuts of G are exactly the preimages of the mincuts of C, i.e., for every mincut[2] $A \subseteq V(C)$, $\phi^{-1}(A)$ is a mincut of G, and all mincuts of G can be obtained in this way. The pair (C, ϕ) is called a *cactus representation* of G. Fleiner and Frank [6] provide a simplified proof for the existence of a cactus representation. We will call the elements of $V(G)$ *vertices*, the elements of $V(C)$ *nodes*, and the preimages of nodes of C *blobs*.

In general, a cactus representation needs to include nodes with empty preimages. This happens for example for the K_4; its cactus is a star with double edges where the central node has an empty preimage and the remaining nodes correspond to the vertices of the K_4. For graphs whose mincuts have size two, nodes with empty preimages are not needed, and a cactus representation can be obtained by contracting the 3-edge-connected components into a single node.

Lemma 11 ([20, Section 2.3.5]) *Let G be a 2-edge-connected graph that is not 3-edge-connected. Contracting each 3-edge-connected components of G into a node yields a cactus representation (C, ϕ) of G with the following properties:*

i) The edges of C are in one-to-one correspondence to the edges of G that are contained in a 2-cut.
ii) For every node $c \in V(c)$, $\phi^{-1}(c)$ is a 3-edge-connected component of G.

10.1 Verifying a Cactus Representation

Let G be a graph and let (C, ϕ) be an alleged cactus-representation of its 2-cuts in the sense of Lemma 11. We show how to verify a cactus representation in linear time. We need to check two things. First, we need to ensure that C is indeed a cactus graph, that is, every edge of C is contained in exactly one cycle, that ϕ is a surjective mapping and hence there are no empty blobs, and that every edge of G either connects two vertices in the same blob or is also present in C. Second, we need to verify that the blobs of C are 3-edge-connected components of G. For this purpose, the cactus representation is augmented by a Mader construction sequence for each blob B. The verification procedure from Sect. 9 can then be applied.

[2] For this theorem, a cut is specified by a set of vertices, and the edges in the cut are the edges with exactly one endpoint in the vertex set.

We first verify that C is a cactus. We compute a chain decomposition of C and verify that every chain is a cycle. We label all edges in the i-th cycle by i. We have now verified that C is a cactus.

Surjectivity of ϕ is easy to check. We then iterate over the edges $u\,v$ of G. If its endpoints belong to the same blob, we associate the edge with the blob. If its endpoints do not belong to the same blob, we add the pair $\phi(u)\,\phi(v)$ to a list. Having processed all edges, we check whether the constructed list and the edge list of C are identical by first sorting both lists using radix sort and then comparing them for identity.

We finally have to check that the blobs of C correspond to 3-edge-connected components of G. Our goal is to use the certifying algorithm for 3-edge-connectivity on the substructures of G that represent 3-edge-connected components. Let B be any blob. We already collected the edges having both endpoints in B. We also have to account for the paths using edges outside B. We do so by creating an edge $u\,v$ for a every path in G leaving B at vertex u and returning to B at vertex v. It is straightforward to compute these edges; we look at all edges having exactly one endpoint in the blob. Each such edge corresponds to an edge in C. For each such edge, we know to which cycle it belongs. The outgoing edges pair up so that the two edges of each pair belong to the same cycle.

The maximality of each blob B is given by the fact that every edge of C is contained in a 2-edge-cut of C and hence contained in a 2-edge-cut of G.

Every algorithm for computing the 3-edge-connected components of a graph, e.g. [19,20,28–30], can be turned into a certifying algorithm for computing the cactus representation of 2-cuts. We obtain the cactus C and the mapping ϕ by contraction of the 3-edge-connected components (Lemma 11). Then one applies our certifying algorithm for 3-edge-connectivity to each 3-edge-connected component. The drawback of this approach is that it requires *two* algorithms that check 3-connectivity. In the next section we will show how to extend our algorithm so that it computes the 3-edge-connected components and the cactus representation of 2-cuts of a graph directly.

11 Computing a Cactus Representation

We discuss how to extend the algorithm to construct a cactus representation. We begin by examining the structure of the 2-cuts of G more closely to extend our algorithm such that it finds all 2-cuts of the graph and encodes them efficiently.

We will first show that the two edges of every 2-edge-cut of G are contained in a common chain. This restriction allows us to focus on the 2-edge-cuts that are contained in the currently processed chain C_i only. In the subsequent section, we show how to maintain a cactus for every phase i of the algorithm that represents all 2-edge-cuts of the graph of the branch vertices and links of $C_1 \cup \ldots \cup C_i$ in linear space. The final cactus will therefore represent all 2-edge-cuts in G.

There is one technical detail regarding the computation of overlap graphs: For the computation of a Mader-sequence in Sect. 8, we stopped the algorithm when the first 2-edge-cut occurred, as then a Mader-sequence does not exist anymore. Here, we simply continue the algorithm with processing the next chain C_{i+1}. This does not harm the search for cuts in subsequent chains, as the fact that 2-edge-cuts are only contained in common chains guarantees that every 2-edge-cut that contains an edge e in C_i has its second edge also in C_i.

For simplicity, we assume that G is 2-edge-connected and has minimum degree three from now on. Then all 3-edge-connected components contain at least two vertices.

11.1 2-Edge-Cuts are Contained in Chains and an Efficient Representation of All Cuts in a Chain

In phase i of the algorithm, using Lemma 10, we can find a 2-edge-cut for each connected component of the overlap graph H that does not contain R (R is the special vertex in H that represents the branch vertices on C_i). Lemma 13 shows that the set of edges contained in these cuts is equal to the set of edges contained in any cut on C_i. Lemma 12 states easy facts about 2-edge-cuts, in particular, that the edges of any 2-edge-cut are contained in a common chain. The proofs can be found in many 3-connectivity papers, e.g. [19,28–30]. As in the previous sections, all DFS-tree-edges are oriented towards the root, while back-edges are oriented away from the root.

Lemma 12 *Let T be a DFS-tree of a 2-edge-connected graph G. Every 2-edge-cut $(u\,v, x\,y)$ of G satisfies the following:*

(1) At least one of $u\,v$ and $x\,y$ is a tree-edge, say $x\,y$.
(2) $G - u\,v - x\,y$ has exactly two components. Moreover, the edges $u\,v$ and $x\,y$ have exactly one endpoint in each component.
(3) The vertices u, v, x, and y are contained in the same leaf-to-root path of T.
(4) If $u\,v$ and $x\,y$ are tree-edges and w.l.o.g. $u \leq y$, the vertices in $y \to_T u$ and $\{x, v\}$ are in different components of $G - u\,v - x\,y$.
(5) If $u\,v$ is a back-edge, then $x\,y \in (v \to_T u)$ and, additionally, the vertices in $v \to_T x$ and $y \to_T u$ are in different components of $G - u\,v - x\,y$.

Moreover, let $\mathscr{C}$ be a chain decomposition of G. For every 2-edge-cut $\{u\,v, x\,y\}$ of G, $u\,v$ and $x\,y$ are contained in a common chain $C \in \mathscr{C}$.

Lemma 13 *Let $\mathscr{E}$ be the set of edges that are contained in the 2-edge cuts induced by the connected components of the overlap graph H at the beginning of part II of phase i. Then any 2-edge-cut $\{x\,y, u\,v\}$ on C_i is a subset of $\mathscr{E}$.*

Proof Assume for the sake of contradiction that there is an edge $u\,v$ in the 2-edge-cut that is not in $\mathscr{E}$. We distinguish the following cases.

First assume that both $u\,v$ and $x\,y$ are tree-edges and w.l.o.g. $v < u \leq y < x$. Since G has minimal degree three, every vertex on C_i has an incident edge that is not on C_i. Hence it is either a branch vertex, or belongs to some segment with respect to C_i (incident back-edges start chains in segments w.r.t. C_i, incident tree edges are the last edges of chains in segments w.r.t. C_i). As $s(C_i) \leq v$ is a branch vertex, by Lemma 12(4) the path $y \to_T u$ can not contain a branch vertex. In particular, u is not a branch vertex.

Let S_u be any segment having u as attachment vertex. All segments in the connected component of S_u in H must have their attachment vertices on $y \to_T u$ and the connected component does not contain R. Hence this connected component induces a cut containing $u\,v$.

Now assume that one of $u\,v$ and $x\,y$ is a back-edge. If $u\,v$ is the back-edge, then $u = s(C_i)$ and we have $u < y < x < v$ by Lemma 12. The path $v \rightarrow_T x$ cannot contain a branch vertex. Let S_v be any segment that has v as attachment vertex. All segments in the connected component of S_v must have their attachment vertices on $v \rightarrow_T x$ and the connected component does not contain R. Hence $u\,v$ is contained in a cut induced by this connected component.

If on the other hand $u\,v$ is the tree-edge we have $y < v < u < x$ basically the same argument applies when we replace S_v by a segment S_u containing u. □

We next show how to compute a space efficient representation of all 2-cuts on the chain C_i. Using this technique we can store all 2-cuts in G in linear space. In the next section we will then use this to construct the cactus-representation of all 2-cuts in G.

Number the edges in C_i as $e_1, e_2, \ldots, e_k$. Here e_1 is a back edge and e_2 to e_k are tree edges. We start with a simple observation. Let $h < i < j$. If (e_h, e_i) and (e_i, e_j) are 2-edge-cuts, then (e_i, e_j) is a 2-edge-cut.

Using this observation, we want to group the edges of 2-edge-cuts of C_i such that (i) every two edges in a group form a 2-edge-cut and (ii) no two edges of different groups form a 2-edge-cut. The existence of such a grouping has already been observed in [19,28,30]. We show how to find it using the data structures we have on hand during the execution of our algorithm.

Consider the overlap graph H in phase i of our algorithm. We need some notation. Let I be the set of intervals on C_i that contains for every component of H (except the component representing the branch vertices on C_i) with extremal attachment vertices a and b the interval $[a, b]$. Since the connected components of H are maximal sets of overlapping intervals, I is a *laminar family*, i.e. every two intervals in I are either disjoint or properly contained in each other. In particular, no two intervals in I share an endpoint. The layers of this laminar family encode which edges form pairwise 2-cuts in G, see Fig. 4. We define an equivalence relation to capture this intuition.

For an interval $[a, b]$, $a < b$, let $\ell([a, b])$ and $r([a, b])$ be the edges of C_i directly before and after a and b, respectively. We call $\{\ell([a, b]), r([a, b])\}$ the *interval-cut* of $[a, b]$. For a subset $S \subseteq I$ of intervals, let E_S be the union of edges that are contained in interval-cuts of intervals in S. According to Lemma 13, every 2-edge-cut in C_i consists of edges in E_I.

We now group the edges of E_I using the observation above. Let two intervals $I_1 \in I$ and $I_2 \in I$ *contact* if $r(I_1) = \ell(I_2)$ or $\ell(I_1) = r(I_2)$. Clearly, the transitive closure $\equiv$ of the contact relation is an equivalence relation. Every block B of $\equiv$ is a

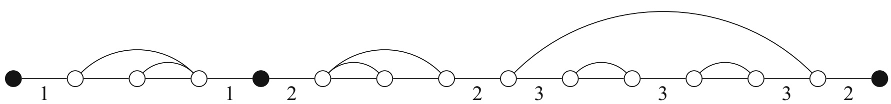

Fig. 4 The intervals induced by the connected components of the overlap graph H form a laminar family. The levels of this family encode which edges form pairwise 2-cuts. Two edges in the figure are labeled with the same number if they form a cut. *Filled vertices* are *branch vertices*

set of pairwise disjoint intervals which are contacting consecutively. This allows us to compute the blocks of $\equiv$ efficiently. We can compute them in time $|I|$ and store them in space $|I|$ by using a greedy algorithm that iteratively extracts the inclusion-wise maximal intervals in I that are contacting consecutively.

Lemma 14 [19,28,30] *Two edges e and e' in C_i form a 2-edge-cut if and only if e and e' are both contained in E_B for some block B of $\equiv$.*

11.2 An Incremental Cactus Construction

In this section we show how to construct a cactus representation incrementally along our algorithm for constructing a Mader sequence. At the beginning of each phase i, we will have a cactus for the graph G^i whose vertices are the branch vertices that exist at this time and whose edges are the links between these branch vertices.

We assume that G is 2-edge-connected but not 3-edge connected and that G has minimum degree three. This ensures that in phase i every vertex on the current chain C_i belongs to some segment or is a branch vertex.

We will maintain a cactus representation (C, ϕ), i.e., for every node v of C, the blob $B = \phi^{-1}(v)$ is the vertex-set of a 3-edge-connected component in G^i. We begin with a single blob that consists of the two branch vertices of the initial K_2^3, which clearly are connected by three edge-disjoint paths.

Consider phase i, in which we add all chains whose source s-belongs to C_i. At the beginning of the phase, the endpoints of C_i and some branch vertices on C_i already exist in G^i. We have a cactus representation of the current graph. The endpoints of C_i are branch vertices and belong to the same blob B, since 2-edge-cuts are contained in chains.

We add all segments that do not induce cut edges and tentatively assign all vertices of C_i to B. If the algorithm determines that C_i does not contain any 2-edge-cut, the assignment becomes permanent, the phase is over and we proceed to phase $i+1$. Otherwise we calculate the efficient representation of 2-edge-cuts on C_i from Sect. 11.1.

Let e_1 be the first edge on C_i in a 2-edge-cut, let A be a block of the contact equivalence relation described in the last section containing e_1 and let $E_A = \{e_1, e_2, \ldots, e_\ell\}$ such that e_j comes before e_{j+1} in C_i for all j. Then every two edges in E_A form a 2-edge-cut. We add a cycle with $\ell - 1$ empty blobs $B_2, \ldots, B_\ell$ to B in C. The ℓ new edges correspond to the ℓ edges in E_A.

For every pair $e_j = (a, b)$, $e_{j+1} = (c, d)$ in E_A we remove the vertices between these edges from B. Since the edges in C_i are linearly ordered, removing the vertices in a subpath takes constant time. We place the end vertices b and c of the path between e_j and e_{j+1} in the blob B_j, add the segments that induced this cut and recurse on the path between b and c. That is, we add all vertices on the path from b to c to B_j, check for cut edges on this path, and, should some exist, add more blobs to the cactus. The construction takes constant time per blob. Figure 5 shows an example.

Graphs that contain nodes of degree two can be handled in the same way, if we add a cycle to each degree two node u. This cycle creates a segment w.r.t. the chain to which u s-belongs and hence the algorithm correctly identifies the two incident edges as cut edges.

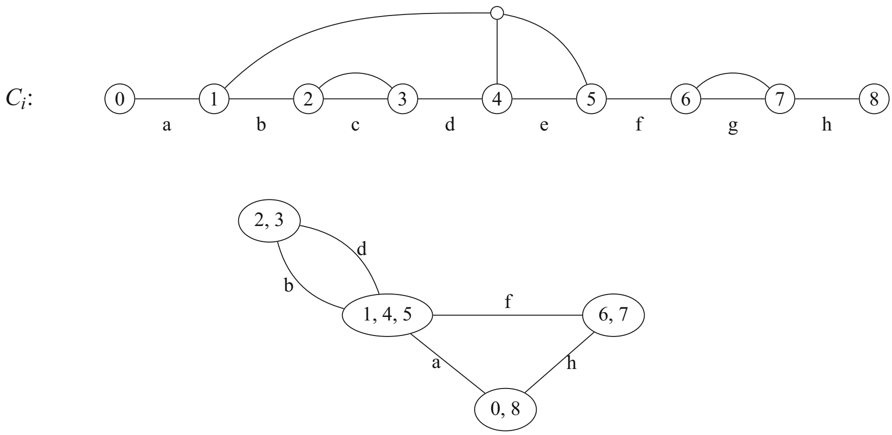

Fig. 5 The segments attached to chain C_i and the corresponding part of the cactus. We first tentatively assign vertices 1–7 to the blob containing the endpoints {0, 8} of C_i. The *top level cuts* are the pairs in the block $\{a, f, h\}$. So we create a cycle with three edges and attach it to the blob containing 0 and 8. We move vertices 1–5 to the blob between a and f, vertices 6–7 to the blob between f and h, and keep vertices 0 and 8 in the parent blob. We then recurse into the first blob. The second level cuts are the pairs in the block $\{b, d\}$. So we create a cycle with two edges and move vertices 2 and 3 to the new blob

Lemma 15 *The above incremental procedure constructs a cactus representation of the 2-edge-cuts in G in linear time.*

Proof Each vertex in G s-belongs to some chain. In the phase in which that chain is treated, all its vertices are added to a blob. Whenever we move a vertex to different blob, we remove it from its previous blob. Therefore each vertex of G is contained in exactly one blob.

Whenever we add edges to the cactus, we do so by adding a cycle that shares exactly one node with the existing cactus. Hence every edge in the cactus lies on exactly one cycle.

Let $\{e_1, e_2\}$ be a 2-edge-cut in G. The two edges must lie on a common cycle in the cactus, since the edges in the cactus are in one-to-one correspondence with edges of G and cutting a cycle in only one place cannot disconnect a graph. As the cycles of the cactus touch in at most one vertex, e_1 and e_2 are a cut in the cactus as well.

Conversely let e_1', e_2' be a cut in the cactus and let e_1, e_2 be the corresponding edges in G. Then e_1', e_2' must lie on some common cycle which, upon their removal, is split into two nonempty parts H_1, and H_2. Assume that $G - e_1 - e_2$ is still connected, then there must be a path from a vertex in the preimage of H_1 to a vertex in the preimage of H_2 in $G - e_1 - e_2$. This path must contain at least one edge $u\,v$ that does not participate in any 2-edge-cut, as otherwise it would be a path in the cactus as well. Moreover, u and v must lie in different blobs B_u and B_v of the cactus.

The one that was created last, say B_u, must be different from the initial blob. Consider the time when B_u was created in the incremental construction of the cactus. We introduced a cycle to some preexisting blob B^* on which all edges were cut edges, in particular the two cut edges incident to B_u. However, the edge $u\,v$ still connects B_u to the rest of graph, since B_v also exists at this time, a contradiction. □

By applying the techniques of this section, the certifying algorithm for 3-vertex-connectivity [26] (which is also based on chain decompositions) can be used to compute the 3-vertex-connected components of a graph. This has been conjectured in [25, p. 18] and yields a linear-time certifying algorithm to construct a SPQR-tree of a graph; we refer to [9,11] for details about 3-vertex-connected components and SPQR-trees. The full construction can be found in the "Appendix 2".

12 Conclusion

We presented a certifying linear time algorithm for 3-edge-connectivity based on chain decompositions of graphs. It is simple enough for use in a classroom setting and can serve as a gentle introduction to the certifying 3-vertex-connectivity algorithm of [26]. We also provide an implementation in Python, available at https://github.com/adrianN/edge-connectivity.

We also show how to extend the algorithm to construct and certify a cactus representation of all 2-edge-cuts in the graph. From this representation the 3-edge-connected components can be readily read off. The same techniques are used to find the 3-vertex-connected components using the algorithm from [26], and thus present a certifying construction of SPQR-trees.

Mader's construction sequence is general enough to construct k-edge-connected graphs for any $k \geq 3$, and can thus be used in certifying algorithms for larger k. So far, though, it is unclear how to compute these more complicated construction sequences. We hope that the chain decomposition framework can be adapted to work in these cases too.

Appendix 1: Computing a Spanning Subgraph of an Overlap Graph

We first assume that all endpoints are pairwise distinct. We will later show how to remove this assumption by perturbation.

For every interval $I = [a, b]$ define its set of left and right neighbors:

$$L(I) = \{I' = [a', b']; a' < a < b' < b\},$$
$$R(I) = \{I' = [a', b']; a < a' < b < b'\}.$$

If the set of left neighbors is nonempty, let the interval $I' \in L(I)$ with the rightmost right endpoint be the immediate left neighbor of I. Similarly, if the set of right neighbors is nonempty, the immediate right neighbor of I is the interval in $R(I)$ with the leftmost left endpoint.

Lemma 16 *The graph G' formed by connecting each interval to its immediate left and right neighbor (if any) forms a spanning subgraph of the overlap graph G and has exactly the same connected components.*

Proof Clearly, every edge of G' is also an edge of G and hence connected components of G' are subsets of connected components of G.

Algorithm 2 Finding a spanning forest of a overlap graph

```
procedure SP(I = {[a_0, a'_0], ..., [a_ℓ, a'_ℓ]})
    stack = [ ]
    sort I lexicographically in descending order
    for [l, r] in I do
        while stack not empty and r > top(stack) right endpoint do
            pop(stack)
        end while
        if stack not empty and r ≥ top(stack) left endpoint then
            connect [l, r], top(stack)
        end if
        push(stack, [l, r])
    end for
    stack = [ ]
    sort I lexicographically in ascending order where the key for [l, r] is [r, l]
    for [l, r] in I do
        while stack not empty and l<top(stack) left endpoint do
            pop(stack)
        end while
        if stack not empty and l ≤ top(stack) right endpoint then
            connect [l, r], top(stack)
        end if
        push(stack, [l, r])
    end for
end procedure
```

For the other direction, assume I and I' are overlapping intervals that are not connected in G'. Then $a < a' < b < b'$, where $I = [a, b] =: I_0$ and $I' = [a', b']$. Let $I_0, I_1, I_2, \ldots$ be such that $I_\ell = [a_\ell, b_\ell]$ is the immediate right neighbor of $I_{\ell-1}$ for all $\ell \geq 1$. Consider the last I_ℓ in this sequence such that $a_\ell < a' < b_\ell < b'$; clearly, such an interval exists, as I_0 is such an interval. Then I' is a right neighbor of I_ℓ, but not the immediate right neighbor of I_ℓ, as otherwise I and I' would be connected in G'. Hence, the immediate right neighbor $I_{\ell+1} =: U =: [c, d]$ of I_ℓ exists, is different from I', and must contain I'. Thus

$$a < c < a' < b < b' < d.$$

Starting from I' and going to immediate left neighbors, we obtain in the same fashion an interval $U' = [c', d']$ with

$$c' < a < a' < b < d' < b'.$$

We conclude that U' and U overlap, but are not connected in G'.

Consider now a particular choice for the overlapping intervals I and I'. We choose them such that the left endpoint of I is as small as possible. However, the left endpoint of U' is to the left of the left endpoint of I, and we have derived a contradiction. □

It is easy to determine all immediate right neighbors by a linear time sweep over all intervals. We sort the intervals in decreasing order of left endpoint and then sweep over the intervals starting with the interval with rightmost left endpoint. We maintain a stack S of intervals, initially empty. If $I_1 = [a_1, b_1], \ldots, I_k = [a_k, b_k]$ are the intervals on the stack with I_1 being on the top of the stack, then $a_1 < a_2 < \ldots < a_k$ and $b_1 < b_2 < \ldots < b_k$, I_1 is the last interval processed, and $I_{\ell+1}$ is the immediate right neighbor of I_ℓ if I_ℓ has right neighbors. If I_ℓ does not have right neighbors, $a_{\ell+1} > b_\ell$. Let $I = [a, b]$ be the next interval to be processed. Its immediate right neighbor is the topmost interval I_ℓ on the stack with $b_\ell > b$ (if any). Hence we pop intervals I_ℓ from the stack while $b > b_\ell$ and then connect I to the topmost interval if $b > a_\ell$, and push I. The determination of immediate left neighbors is symmetric.

It remains to deal with intervals with equal endpoints. We do so by perturbation. It is easy to see that the following rules preserve the reachability by overlaps and eliminate equal endpoints. E.g., in (4), the two intervals are forced to overlap, so reaching one of the two intervals gives a path to the other; the same reasoning motivates (2) and (3).

1. if a left and a right endpoint are at the same coordinate, then the left endpoint is smaller than the right endpoint.
2. if two left endpoints are equal, the one belonging to the shorter interval is smaller.
3. if two right endpoints are equal, the one belonging to the shorter interval is larger.
4. if two intervals are equal, one is slightly shifted to the right.

In other words, the endpoints of an interval $I_i = [a, b]$ are replaced by $((a, -1, b-a, i)$ and $(b, 1, b-a, i))$ and comparisons are lexicographic. The perturbation need not be made explicitly, it can be incorporated into the sorting order and the conditions under which edges are added, as described in Algorithm 2.

Appendix 2: Computing all 3-Vertex-Connected Components

A pair of vertices $\{x, y\}$ is a separation pair of G if $G - x - y$ is disconnected. Similar to the edge-connectivity case, it suffices to compute all vertices that are contained in separation pairs of G in order to compute all 3-vertex-connected components of G. We assume that G is 2-vertex-connected and has minimum degree 3.

For a rooted tree T of G and a vertex $x \in G$, let $T(x)$ be the subtree of T rooted at x. The following lemmas show that separation pairs can only occur in chains. Weaker variants of Lemma 17 can be found in [11,31,32].

Lemma 17 *Let T be a DFS-tree of a 2-connected graph G and $\mathscr{C}$ be a chain decomposition of G. For every separation pair $\{x, y\}$ of G, x and y are contained in a common chain $C \in \mathscr{C}$.*

Proof The following simple observation will be useful. Let r be the root of T and let $x \neq r$ be any vertex. Then for every $t \in T(x) - x$, there is a path P from t to a vertex $s \in G - T(x)$ such that P consists only of vertices in $T(x) - x \cup s$.

We first prove that x and y are *comparable* in T, i.e., contained in a leaf-to-root path of T. Assume they are not. Then $G - x - y$ consists of at most three connected components: one connected component containing the least common ancestor of x and y in T, and the at most two connected components that contain the proper descendants of x and y, respectively. According to the observation above, these components coincide, contradicting that $\{x, y\}$ is a separation pair.

Let x' be the child of x in T that lies on the path $y \rightarrow_T x$. Clearly, if $x' = y$, the chain containing the edge xy is a common chain containing x and y. Otherwise, $x' \neq y$. If $x = r$, then there is a back-edge rt such that $t \in T(y)$, according to the fact that $G - r$ is connected by $T - r$ and due to the observation above. This back-edge rt implies that the first chain C that traverses a vertex of $T(y)$ starts at r and, hence, contains x and y.

In the remaining case, $x' \neq y$ and $x \neq r$. Let st be a back-edge that connects an ancestor s of x with a descendant t of x' (possibly x' itself) such that s is minimal; this edge st exists, since G is 2-vertex-connected. According to [27], C_1 is the only cycle in $\mathscr{C}$ and it follows that $s < x$. If $t \in T(y)$, the first chain C in $\mathscr{C}$ that contains such a back-edge contains x and y and, hence, satisfies the claim. Otherwise, t is a vertex in $T(x') - T(y)$. Due to the back-edge st, $G - x - T(y)$ is contained in one connected component of $G - x - y$. According to the observation above (applied on y), $\{x, y\}$ can form a separation pair only if y has a child y' such that all back-edges that end in $T(y')$ start either in $T(y)$ or at x. Since G is 2-connected, there must be a back-edge from x to $T(y')$. The first chain C in $\mathscr{C}$ containing such a back-edge gives the claim, as it contains x and y. □

Similar to edge-connectivity, the connected components of the overlap graph for C_i represent all vertices in separation pairs that are contained in C_i. The connected components of the overlap graph can be computed efficiently [26, Lemma 51]. After finding all these vertices for C_i, a simple modification allows the algorithm in [26, p. 508] to continue, ignoring all previously found separation pairs: For every separation pair $\{x, y\}$, $x < y$, that has been found when processing C_i, there is a vertex v strictly between x and y in C_i. Furthermore, by doing a preprocessing [26, Property B, p. 508] one can assume that $t(C_i) \rightarrow_T s(C_i)$ also has an inner vertex w. We eliminate every separation pair $\{x, y\}$ after processing C_i by simply adding the new back-edge vw to G. As the new chain containing vw is just an edge, this does not harm future processing steps.

According to Lemma 17, this gives all vertices in the graph that are contained in separation pairs. The 3-vertex-connected components can then be computed in linear time by iteratively splitting separation pairs and gluing together certain remaining structures, as shown in [9,11].

References

1. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, Ch.: A framework for the verification of certifying computations. J. Autom. Reason. **52**(3), 241–273 (2014)
2. Bondy, J.A., Murty, U.S.R.: Graph Theory. Springer, Berlin (2008)
3. Corcoran, J.N., Schneider, U., Schüttler, H.-B.: Perfect stochastic summation in high order feynman graph expansions. Int. J. Mod. Phys. C **17**(11), 1527–1549 (2006)
4. Dehne, F., Langston, M., Luo, X., Pitre, S., Shaw, P., Zhang, Y.: The cluster editing problem: implementations and experiments. In: *Parameterized and Exact Computation*, pp. 13–24 (2006)
5. Dinits, E.A., Karzanov, A.V., Lomonosov, M.V.: On the structure of a family of minimal weighted cuts in graphs. In: Studies in Discrete Mathematics (in Russian), pp. 290–306 (1976)
6. Fleiner, T., Frank, A.: A quick proof for the cactus representation of mincuts. Technical Report QP-2009-03, Egerváry Research Group, Budapest (2009)

7. Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Inf. Process. Lett. **74**(3–4), 107–114 (2000)
8. Galil, Z., Italiano, G.F.: Reducing edge connectivity to vertex connectivity. SIGACT News **22**(1), 57–61 (1991)
9. Gutwenger, C., Mutzel, P.: A linear time implementation of SPQR-trees. In: Proceedings of the 8th International Symposium on Graph Drawing (GD'00), pp. 77–90 (2001)
10. Hopcroft, J., Tarjan, R.: Efficient planarity testing. J. ACM **21**(4), 549–568 (1974)
11. Hopcroft, J.E., Tarjan, R.E.: Dividing a graph into triconnected components. SIAM J. Comput. **2**(3), 135–158 (1973)
12. Karger, D.R.: Minimum cuts in near-linear time. J. ACM **47**(1), 46–76 (2000)
13. Linial, N., Lovász, L., Wigderson, A.: Rubber bands, convex embeddings and graph connectivity. Combinatorica **8**(1), 91–102 (1988)
14. Lovász, L.: Computing ears and branchings in parallel. In: Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS'85) (1985)
15. Mader, W.: A reduction method for edge-connectivity in graphs. In: Bollobás, B. (ed.) Advances in Graph Theory, vol. 3 of Annals of Discrete Mathematics, pp. 145–164 (1978)
16. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
17. Mehlhorn, K.: Nearly optimal binary search trees. Acta Inform. **5**, 287–295 (1975)
18. Mehlhorn, K., Näher, S., Uhrig, C.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (1999)
19. Nagamochi, H., Ibaraki, T.: A linear time algorithm for computing 3-edge-connected components in a multigraph. Jpn. J. Ind. Appl. Math. **9**, 163–180 (1992)
20. Nagamochi, H., Ibaraki, T.: Algorithmic Aspects of Graph Connectivity (Encyclopedia of Mathematics and its Applications). Cambridge University Press, Cambridge (2008)
21. Neumann, A.: Implementation of Schmidt's algorithm for certifying triconnectivity testing. Master's thesis, Universität des Saarlandes and Graduate School of CS, Germany (2011)
22. Noschinski, L., Rizkallah, C., Mehlhorn, K.: Verification of certifying computations through Autocorres and Simpl. In: *NASA Formal Methods, vol. 8430 of LNCS*, pp. 46–61 (2014)
23. Olariu, S., Zomaya, A.Y.: A time- and cost-optimal algorithm for interlocking sets-with applications. IEEE Trans. Parallel Distrib. Syst. **7**(10), 1009–1025 (1996)
24. Ramachandran, V.: Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In: Synthesis of Parallel Algorithms, pp. 275–340 (1993)
25. Schmidt, J.M.: Contractions, removals and certifying 3-connectivity in linear time. Tech. Report B 10-04, Freie Universität Berlin, Germany (2010)
26. Schmidt, J.M.: Contractions, removals and certifying 3-connectivity in linear time. SIAM J. Comput. **42**(2), 494–535 (2013)
27. Schmidt, J.M.: A simple test on 2-vertex- and 2-edge-connectivity. Inf. Process. Lett. **113**(7), 241–244 (2013)
28. Taoka, S., Watanabe, T., Onaga, K.: A linear time algorithm for computing all 3-edge-connected components of a multigraph. IEICE Trans. Fundam. **E75**(3), 410–424 (1992)
29. Tsin, Y.H.: A simple 3-edge-connected component algorithm. Theory Comput. Syst. **40**(2), 125–142 (2007)
30. Tsin, Y.H.: Yet another optimal algorithm for 3-edge-connectivity. J. Discrete Algorithms **7**(1), 130–146 (2009)
31. Vo, K.-P.: Finding triconnected components of graphs. Linear Multilinear Algebra **13**, 143–165 (1983)
32. Vo, K.-P.: Segment graphs, depth-first cycle bases, 3-connectivity, and planarity of graphs. Linear Multilinear Algebra **13**, 119–141 (1983)