Certifying 3-Edge-Connectivity

Kurt Mehlhorn, Adrian Neumann, and Jens M. Schmidt

Max Planck Institute for Informatics, Saarbrücken, Germany

Abstract. We present a linear-time certifying algorithm that tests graphs for 3-edge-connectivity. If the input graph G is not 3-edge-connected, the algorithm returns a 2-edge-cut. If G is 3-edge-connected, the algorithm returns a construction sequence that constructs G from the graph with two nodes and three parallel edges using only operations that (obviously) preserve 3-edge-connectivity.

1 Introduction

Advanced graph algorithms answer complex yes-no questions such as "Is this graph planar?" or "Is this graph k-vertex-connected?". They are not only nontrivial to implement, it is also difficult to test their implementations, as usually only small test sets are available. It is hence possible that bugs persist unrecognized for a long time. An example is the linear time planarity test of Hopcroft and Tarjan [7] in LEDA [13]. A bug was discovered only after two years of intensive use.

Certifying algorithms [12] 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 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. In that case, the solution of *every instance* is *correct by a formal proof* [1].

Our main result is a linear time certifying algorithm for 3-edge-connectivity. Mader [11] 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 [6, 15, 22–24] 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 uses the concept of a *chain decomposition* of a graph introduced in [19]. A chain decomposition is an ear decomposition [10]. It is used in [21] as a common and simple framework for certifying 1- and 2-vertex, as well as 2-edge-connectivity. Further, [20] uses them for certifying 3-vertex-connectivity. Chain decompositions are an example of *path-based* algorithms (see, e.g., Gabow [5]), which use only the simple structure of certain paths in a DFS-tree to compute connectivity information about the graph.

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 [20], but in a simpler form. Hence our paper may also be used as a gentle introduction to the 3-vertex-connectivity algorithm in [20].

Related Work. Deciding 3-edge-connectivity is a well researched problem, with applications in fields such as bioinformatics [4] and quantum chemistry [3]. Consequently, there are many linear time solutions known [6, 15, 22–24]. None of them is certifying.

The paper [12] is a recent survey on certifying algorithms. For a linear time certifying algorithm for 3-vertex-connectivity, see [20] (implemented in [16]). For general k, there is a randomized certifying algorithm for k-vertex connectivity in [9] with expected running time $O(kn^{2.5} + nk^{3.5})$. There is a non-certifying algorithm [8] for deciding k-edge-connectivity in time $O(m \log^3 n)$ w.h.p..

In [6], 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 [20] 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.

2 Preliminaries

We consider finite undirected graphs G with n vertices, m edges, no self-loops, and minimum degree three, and use standard graph-theoretic terminology from [2], unless stated otherwise. We use uv 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. Every vertex in $P \setminus \{s(P), t(P)\}$ 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, let x be an *ancestor* of y and y be a *descendant* of x if $x \in V(r \rightarrow_T y)$. 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 uv* of a graph G be the operation that replaces uv 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 [11]). *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).*

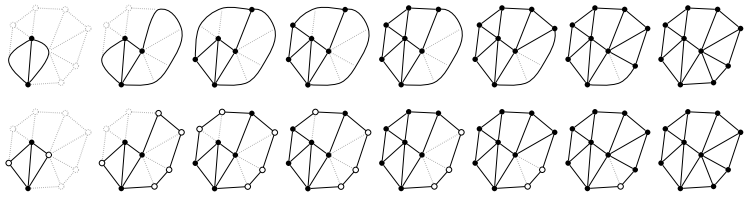

Fig. 1. Two ways of constructing the 3-edge-connected graph shown in the rightmost column. The upper row shows the construction according to Theorem 1. The lower row shows the construction according to Corollary 1. Branch (non-branch) vertices are depicted as filled (non-filled) circles. The black edges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

- *Subdividing an edge xy and connecting the new vertex to any existing vertex.*
- *Subdividing two distinct edges wx, yz and connecting the two new vertices.*

A subdivision G' of a graph G is a graph obtained by subdividing edges 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.

3 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 [21] and 3-vertex-connectivity [20]. In this paper we show that it can also be used to find 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 [18].

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 uv up, i.e., such that $v < u$, and back-edges xy down, that is, such that $x < y$.

We decompose[1] 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[2]. Then, for every back-edge vw with $s(vw) = v$, we traverse $w \to_T r$ until a vertex x is encountered that was visited before; x is a descendant of v. The traversed subgraph $vw \cup (w \to_T x)$ 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.

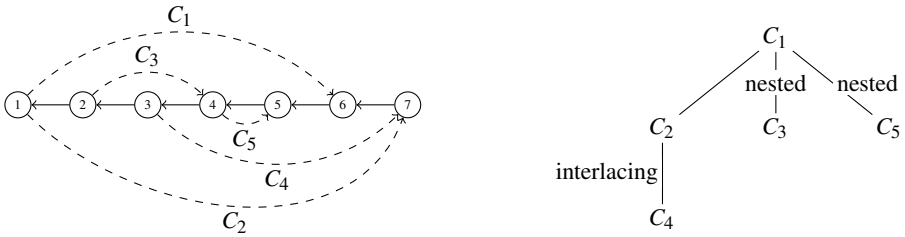

Fig. 2. The left side of the figure shows a DFS tree with a chain decomposition; tree-edges are solid and back-edges are dashed. C_1 is (16,65,54,43,32,21), C_2 is (17,76), C_3 is (24), C_4 is (37), and C_5 is (45). C_3 and C_5 are nested children of C_1 and C_4 is an interlacing child of C_2. Also, $s(C_4)$ s-belongs to C_1.
In the algorithm of Sect. 6, we start with $G_c = C_1 \cup C_2$. In the first phase, we form three segments, 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.

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 ([21]). *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, 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 $vp(v)$. 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 has no parent. For any chain $C \neq C_1$, let the *parent* $p(C)$ of C be the chain to which $t(C)$ s-belongs. We

[1] If G is not 2-edge-connected, there will be edges and maybe vertices not belonging to any chain.
[2] Initially, no vertex is visited.

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 $j < i$.

4 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 assume that the input graph is 2-edge-connected. This is easily checked using Lemma 1. This guarantees that the chain decomposition is a partition of the edge set of the input graph. We will also use the following necessary condition for 3-edge-connectivity.

Proposition 1. *Let T be a DFS tree starting at r for a graph G. If G is 3-edge-connected, then the subtree of every child of r must be connected to r by at least two back-edges.*

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 $C_1 \setminus r$ 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, Proposition 1 exhibits a 2-edge-cut, namely the tree-edge and the back-edge connecting r and T'. 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 $t(C) \to_T s(C)$ 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 $t(C) \to_T s(C)$.

We can now prove that chains can always be added in parent-first order.

Theorem 2. *Let G graph and let G_c be a parent-closed union of chains such that no child of a chain $C \subset G_c$ 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 child 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 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, 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 $z \leq x$ for all $x \in P$, no inner vertex of P belongs to G_c, and the vertex set of G_c is a parent-closed subset of the DFS-tree, z 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 $b \to_T a$ is part of G_c by Lemma 2 and hence no inner vertex of P lies on this path. Let av be the first edge on P. The vertex v must be a descendant of b as otherwise the path $v \to_P b$ would contain a cross-edge, i.e. an edge between different subtrees. Hence av is a back-edge. Let D be the chain that starts with the edge av. 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 as 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 yb be the last edge on the path $t(D) \to_T b$. We claim that yb 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 contain a cross-edge.

Let D^* be the chain containing yb. 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 $t(D^*) = b$. Since $b = t(D^*) \in G_c$, $p(D^*) \in G_c$.

As a and b are not inner vertices of the same link, the path $t(D^*) \to_T s(D^*)$ 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.*

Theorem 2 gives rise to an $O((n+m)\log(n+m))$ algorithm, the Greedy-Chain-Addition Algorithm. Details can be found in the full version of this paper.

5 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 nodes of the chain can be added to the list of addable chains immediately. However, children that have both endpoints as inner nodes 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 $\{C_3, \dots C_{m-n+1}\}$ into two types. Let C be a chain with parent $\widehat{C} = p(C)$. We distinguish two cases[3] 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 two parts corresponding to the two cases above, namely $t(\widehat{C}) \rightarrow_T s(\widehat{C})$, and $(v \rightarrow_T t(\widehat{C}))\backslash t(\widehat{C})$. Depending on which of these paths $s(C)$ lies, it is classified as 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.*

6 A Linear Time Algorithm

According to Lemma 4, interlacing chains whose parent already belongs to the current graph are always Mader-paths and can be added. Adding a chain may create new branching vertices which in turn can turn other chains into Mader-paths. This observation suggests adding interlacing chains as early as possible. Only when there is no interlacing chain to add, we need to consider nested chains. In that case and if the graph is 3-edge-connected, some nested chain must be addable (because a previously added chain created a branching vertex on the tree-path from the sink to the source of the chain). The question is how to find this nested chain efficiently.

The following observation paves the way. Once we add a nested chain, its interlacing children and then their interlacing children etc. become addable. This suggests considering nested chains not in isolation, but to consider them together with their interlacing offspring. We formalize this intuition in the concept of segment below.

[3] In [20], 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. 3;
    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 segments with interlacing root
        Add all segments whose minimal chain is interlacing to G_c;
                                        ▷ Part II of Phase i: add segments with nested root
        Either find an insertion order S_1, ..., S_k on 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
```

Nested chains have both endpoints on their parent chain. Consider the chains nested in chain C_i. Which chains can help their addition by creating branching points on C_i? First, chains nested in C_i and their interlacing offspring, and second, interlacing chains having their source on some C_j with $j < i$. Chains having their source on some C_j with $j > i$ cannot help because they have no endpoint on C_j. These observations suggest an algorithm operating 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, ..., C_{m-n+1} 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.*

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.

We can now define the segments with respect to C_i. 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 $C^* = D^*$, see Figure 2 for an illustration.

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 . 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.*

It is easy to determine the segments with respect to C_i. 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[4]. In the former case, we 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 this segment can be added according to Corollary 7. We have now completed part I of phase i, namely the addition of all segments whose minimal chain is interlacing. We have also determined the segments with nested minimal chain.

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 containing a nested 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 $t(C) \to_T s(C)$, 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. Then all attachment points of S lie on the path $t(C) \to_T s(C)$ and hence on C_i.*

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 $a_2 \to_T a_1$. Further, between two vertices S_i and

[4] Initially, all chains are unmarked

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 $b_1 \leq a_1 \leq b_2 \leq a_2$.

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.*

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 $a_0 \leq a_1 \leq \ldots \leq a_k$ be the set of attachment points of S, i.e., the set of vertices that S has in common with C_i. We associate the intervals $\{[a_0, a_\ell] | 1 \leq \ell \leq k\} \cup \{[a_\ell, a_k] | 1 \leq \ell < k\}$, with S and for every branch vertex v on C_i we define an interval $[0, v]$. See Figure 3 for an example.

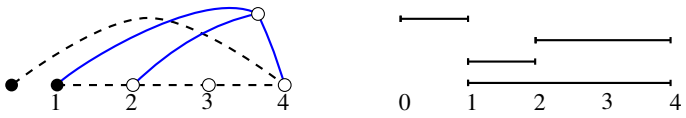

Fig. 3. Intervals for the solid segment with attachment points 1,2,4.

We say two intervals $[a, a'], [b, b']$ *overlap* if $a \leq b \leq a' \leq b'$. 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 belong to the same segment makes H' isomorphic to the overlap graph as required for Lemma 10. Hence we can use H' to find the order on the segments.

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 [17], we can compute a spanning forest of H' in time linear in the number of intervals. The presentation in [17] is for the PRAM and thus needlessly complicated for our purposes. A simpler explanation can be found in the full version of this paper [14].

Since the number of intervals created for a chain C_i is bounded by $|\mathrm{NC}(C_i)| + 2|\mathrm{In}(C_i)| + |V_{\mathrm{branch}}(C_i)|$, where $\mathrm{NC}(C_i)$ are the nested children of C_i, $\mathrm{In}(C_i)$ are the interlacing chains that start on C_i, and $V_{\mathrm{branch}}(C_i)$ is the set of branch vertices on C_i, the total time spent on this procedure for all chains is $O(m)$. From the above discussion follows:

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

7 Verifying the Certificate

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.

Checking the Mader sequence is slightly more involved. We assume that each edge in a Mader-path is doubly linked to the corresponding edge in G. Let G' be a copy of G. We remove the Mader-paths again, in reverse order, 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. There are several things that we need to verify:

- 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.

8 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 [20]. We also provide an implementation in Python, available at `https://github.com/adrianN/edge-connectivity`.

There remain some open problems. Foremost, our algorithm only computes one 2-edge-cut. Is it possible to compute the 3-edge-connected components easily?

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.

References

1. E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. Verification of certifying computations. In *Computer Aided Verification*, pages 67–82. Springer, 2011.
2. J. A. Bondy and U. S. R. Murty. *Graph Theory*. Springer, 2008.

3. J. N. Corcoran, U. Schneider, and H.-B. Schüttler. Perfect stochastic summation in high order feynman graph expansions. *International Journal of Modern Physics C*, 17(11):1527–1549, 2006.
4. F. Dehne, M. Langston, X. Luo, S. Pitre, P. Shaw, and Y. Zhang. The cluster editing problem: Implementations and experiments. *Parameterized and Exact Computation*, pages 13–24, 2006.
5. H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000.
6. Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, 1991.
7. J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.
8. D. R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000.
9. N. Linial, L. Lovász, and A. Wigderson. Rubber bands, convex embeddings and graph connectivity. *Combinatorica*, 8(1):91–102, 1988.
10. L. Lovász. Computing ears and branchings in parallel. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, 1985.
11. W. Mader. A reduction method for edge-connectivity in graphs. In B. Bollobás, editor, *Advances in Graph Theory*, volume 3 of *Annals of Discrete Mathematics*, pages 145–164. 1978.
12. R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.
13. K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
14. Kurt Mehlhorn, Adrian Neumann, and Jens M. Schmidt. Certifying 3-edge-connectivity. *CoRR*, abs/1211.6553, 2012.
15. H. Nagamochi and T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9:163–180, 1992.
16. A. Neumann. Implementation of Schmidt's algorithm for certifying triconnectivity testing. Master's thesis, Universität des Saarlandes and Graduate School of CS, Germany, 2011.
17. S. Olariu and A. Y. Zomaya. A time- and cost-optimal algorithm for interlocking sets – With applications. *IEEE Trans. Parallel Distrib. Syst.*, 7(10):1009–1025, 1996.
18. V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In *Synthesis of Parallel Algorithms*, pages 275–340, 1993.
19. J. M. Schmidt. Contractions, removals and certifying 3-connectivity in linear time. Tech. Report B 10-04, Freie Universität Berlin, Germany, May 2010.
20. J. M. Schmidt. Contractions, removals and certifying 3-connectivity in linear time. *SIAM Journal on Computing*, 42(2):494–535, 2013.
21. J. M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters*, 113(7):241–244, 2013.
22. S. Taoka, T. Watanabe, and K. Onaga. A linear time algorithm for computing all 3-edge-connected components of a multigraph. *IEICE Trans. Fundamentals E75*, 3:410–424, 1992.
23. Y. H. Tsin. A simple 3-edge-connected component algorithm. *Theor. Comp. Sys.*, 40(2):125–142, 2007.
24. Y. H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *J. of Discrete Algorithms*, 7(1):130–146, 2009.