Implementation of Schmidt’s Algorithm for Certifying Triconnectivity Testing

submitted by
Adrian Neumann

March 23, 2011

Supervisor
Prof. Dr. Kurt Mehlhorn

Advisors
Prof. Dr. Kurt Mehlhorn
Arash Farzan, Ph. D.

Reviewers
Prof. Dr. Kurt Mehlhorn
Arash Farzan, Ph. D.
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, den ___________________________ ___________________________
Contents

1 Introduction and Preliminaries 1
   1.1 Introduction .............................................. 1
   1.2 Preliminaries ............................................ 1
      1.2.1 Barnette-Grünbaum construction sequences .......... 2
2 Schmidt’s Algorithm 9
   2.1 Outline ................................................ 9
   2.2 Initial Checks ......................................... 11
   2.3 Finding the Initial Subdivision .......................... 12
   2.4 Chain Decomposition ................................... 18
   2.5 Chain classification ................................... 21
   2.6 A Final Check .......................................... 24
   2.7 Caterpillars ........................................... 26
   2.8 Restrictions ........................................... 28
   2.9 When Individual Chains Are BG-paths ................... 29
   2.10 When Caterpillars Need To Be Decomposed ............... 31
   2.11 Constructing the BG-Decomposition ..................... 36
      2.11.1 Partitioning into segments ......................... 38
      2.11.2 Adding the easy segments .......................... 40
      2.11.3 Adding the hard segments .......................... 45
   2.12 Interlocking Intervals .................................. 52
   2.13 Constructing and Verifying the Certificate ............... 62
3 Experimental Evaluation and Conclusion 71
   3.1 Computational Evaluation ................................. 71
   3.2 Conclusions and Future Work ............................. 76
CONTENTS

A Datastructures 77
  A.1 Chains .................................................. 77
  A.2 Caterpillars ............................................. 78
  A.3 Intervals .................................................. 79
  A.4 Certificates ............................................. 80

B Helper Functions 81
  B.1 Bucket sort ............................................. 81
  B.2 Depth First Search ..................................... 83
  B.3 Merging Nodes .......................................... 84
  B.4 Smoothen ................................................ 85
  B.5 Adding a chain to the subdivision .................... 85
Chapter 1

Introduction and Preliminaries

1.1 Introduction

Determining whether a graph is 3-connected, that is, whether it stays connected upon removal of any two nodes, has many important applications, inter alia in graph drawing and efficient planarity testing.

For the longest time, the algorithm of Hopcroft and Tarjan [6] has been the standard way of determining 3-connectivity in linear time. Unfortunately, it is not trivial to implement and is sufficiently involved for a bug to hide in the original description for several decades before a correction was published by Gutwenger and Mutzel [5] in 2001.

For algorithms that are important in practice but difficult to implement correctly, Mehlhorn and Schweitzer introduced the notion of certifying algorithms that produce an easily verified certificate of correctness alongside their results. For an overview of the topic see [8].

Recently, Schmidt [13] published the first linear time algorithm for certifying 3-connectivity. The main contribution of this thesis is an implementation of said algorithm using LEDA, the Library of Efficient Datatypes and Algorithms [9].

To make the thesis as self-contained as reasonably possible, we will reiterate most of the results needed to prove the correctness of Schmidt’s approach.

1.2 Preliminaries

In this section, we will briefly review the basic results about 3-connected graphs that underlie the workings of Schmidt’s algorithm.

We will show that every 3-connected graph $G$ with more than six edges contains a deletable edge $e$, i.e. deleting $e$ results in a smaller graph that is still 3-connected. Recursively deleting these edges leads to a $K_4$, the complete graph on four nodes,
if and only if the graph is 3-connected. Thus, the deletion sequence provides a linear length certificate for 3-connectivity. Instead of a deletion sequence, Schmidt’s algorithm computes a related construction sequence, i.e. a sequence of operations that preserve 3-connectedness and iteratively construct $G$ from a $K_4$.

We start by introducing some new terms and notation. For a node $v$ with degree two, let $\text{smooth}(v)$ be the operation that removes $v$ from $G$ and connects its two neighbours by a new edge. $\text{smooth}(v)$ does nothing if $v$ has degree greater than two.

For a graph $G$, $\text{smooth}(G)$, is the operation that smoothes all nodes of $G$. Let $\text{deleting}$ an edge $e = (u,v)$ be the operation that removes $e$, followed by a smoothing of $u$ and $v$. On the other hand, let $\text{subdividing}$ an edge be the operation that replaces an edge $e$ by a path of length at least one. Let a $\text{subdivision}$ $S_G$ of a graph $G$ be a graph in which one or more edges of $G$ have been subdivided, such that $\text{smooth}(S_G) = G$. Let a node $v$ of a subdivision $S_G$ be called $\text{real}$ if $v$ also exists in $\text{smooth}S_G$, i.e. if $v$ has degree at least three.

During the discussion we will deal with Depth-First Search (DFS) trees. In such trees the edges can be partitioned into two groups: tree edges and back edges. For tree edges we will write $a \rightarrow b$, and for back edges we write $a \hookrightarrow b$. Note that throughout this text back edges point away from the root and tree edges point towards it. This is exactly opposite to the orientation that is commonly used.

Paths of edges are written as $a \rightarrow^* b$, where a subscript indicates along which structure the path runs, e.g. $a \rightarrow^*_T b$ is a path in the tree $T$. For a path $P = a \rightarrow^* b$, the nodes $V(P)$ are contained in $P$, whereas the nodes $V(P) \setminus \{a,b\}$ are inner nodes of $P$.

We state the following basic result on the existence of certain paths in $k$-connected graphs:

**Theorem 1.2.1** (Menger’s theorem [10]): Let $G$ be a $k$-connected graph, let $x$ be a node of $G$, and let $Y \subseteq V \setminus \{x\}$ be a set of at least $k$ nodes of $G$. Then there are $k$ disjoint paths from $x$ to $Y$ in $G$. □

**1.2.1 Barnette-Grünbaum construction sequences**

Next, we introduce the construction sequence that Schmidt’s algorithm computes. It is based on the Barnette-Grünbaum operations (BG-operations) for constructing any 3-connected graph from a $K_4$ by a series of edge insertions. We first show the results that lead to the existence of a removal sequence and then describe how a sequence of inverse operations can be used to construct a 3-connected graph as shown in [3].

Let $\Pi = \{\pi_1, \ldots, \pi_n\}$ be a family of paths in a graph $G$. We associate a new multigraph $G_{\Pi}$ with $G$ and $\Pi$. The nodes of $G_{\Pi}$ are the nodes of $G$ that occur either as endpoints of some $\pi_i$, or in more than one $\pi_i$.

The edges of $G_{\Pi(i)}$ are defined by the $\pi$ on the new node set. Then, the following holds:
1.2. PRELIMINARIES

Lemma 1.1 ([3, Lemma 4]): Every 3-connected graph $C$ contains a system $T = \{\tau_1, \ldots, \tau_6\}$ of paths such that $C_T$ is $K_4$.

Proof. Let $x, y$ be two distinct nodes of $C$. Since $C$ is 3-connected, there are three disjoint paths $\sigma_1, \sigma_2, \sigma_3$ between $x$ and $y$. At most one of the paths is a single edge, as the graph doesn't contain parallel edges. W.l.o.g. $\sigma_2$ contains another node $z$.

Let $\Sigma$ be the set of all paths having $z$ as endpoint. There must be some $\sigma \in \Sigma$ that contains a $z'$ from $\sigma_1 \cup \sigma_3$. If there is no such path, removing $x$ and $y$ separates the component containing $z$ from $\sigma_1, \sigma_3$. The path $\sigma$ contains a subpath $\sigma_4$ with endpoints $v, w$ s.t. $v \in \sigma_2, w \in \sigma_1 \cup \sigma_3$ and $\sigma_4 \cap \{\sigma_1, \sigma_2, \sigma_3\} = \{v, w\}$. See figure 1.1

We can now use this result to show the existence of a deletable edge in every 3-connected graph.

Theorem 1.2.2 ([3, Theorem 1]): Every 3-connected graph $G$ with more than 6 edges has an edge $e$ such that the graph $G' = G - e$ is still 3-connected.

Proof. We first show how $G$ can be inductively constructed from paths. Consider the system of paths $T = \{\tau_1, \ldots, \tau_6\}$ that maximises the number of nodes while describing a $K_4$ subdivision (note that finding this subdivision involves finding longest paths – an NP-hard problem).

Let $x, y, v, w$ be nodes as in figure 1.1, and let $\pi_0$ be the union of the parts of $T$ that connect $(x, y)$, $(y, v)$ and $(v, w)$. Let $\pi_1 = T \setminus \pi_0$ be the remaining part. The paths $\pi_0$ and $\pi_1$ are disjoint, as lemma 1.1 requires (Figure 1.2).

$G_{\Pi(1)}$ is then, by construction of $\Pi$, $K_4$. If $G_{\Pi(1)}$ is not equal to $G$, we proceed by induction. There are two cases. Either there exists a node $z$ of $G$ which belongs to a $\pi$ in $\Pi(k)$, but is not a node of $G_{\Pi(k)}$ (i.e. $z$ is not part of $\text{smooth}(G_{\Pi(k)})$), or no such node of $G$ exists, but there is a node $z$ of $G_{\Pi(k)}$ that does not have all the incident edges it has in $G$.

Since $G \neq G_{\Pi(k)}$ one of the cases must occur. In the first case we consider the the edge $e = (z_1, \ldots, z_4, z_2)$ in $G_{\Pi(k)}$ that contains $z$. Let $\Sigma$ be the set of all paths in $G$ that start in $z$ and contain neither $z_1$ nor $z_2$. There must be a $\sigma \in \Sigma$ that has a
node $z'$ as endpoint, with $z' \in G_{\Pi(k)} \setminus V(e)$. If there was no such path, removing $v_1$ and $v_2$ disconnects $v$ from the rest of the graph, contradicting the assumption of 3-connectedness.

From $\sigma$ we can construct paths in $G$ that have one endpoint inside of $e$, the other endpoint in $G_{\Pi(k)} \setminus V(e)$ and are disjoint from $G_{\Pi(k)}$ otherwise. Of these paths we use the one that involves most nodes to construct $G_{\Pi(k+1)}$.

In the second case we simply take a longest path from $z$ that involves a missing edge and has only its endpoints in common with $G_{\Pi(k)}$.

Now take a sequence $\{\pi_0, \ldots, \pi_n\}$ of paths that constructs $G$. Since $G \neq K_4$, $n \geq 2$. As $G$ is 3-connected, each vertex has degree at least three. Therefore $\pi_n$ must not contain inner nodes, and hence it consists of just one edge. Deleting the edge results in a graph $G'$ that is isomorphic to $G_{\Pi(n-1)}$ and thus 3-connected.

For finding a construction sequence, we define the following inverse operations of deleting an edge:

**Definition 1.1.** The three BG-operations are as follows:

- adding an edge (possibly parallel)
- subdividing an edge and connecting a third neighbour to the new node
- subdividing two edges and connect the new nodes

See figure 1.3.

From theorem 1.2.2 follows the next theorem that we state without proof.

**Theorem 1.2.3** ([3, Theorem 2]): A graph $G$ is 3-connected if and only if $G$ can be constructed from the $K_4$ using BG-operations.
1.2. PRELIMINARIES

Schmidt’s algorithm uses a different way to characterize the construction sequence that is closer to the paths in theorem 1.2.2. Instead of adding edges to the graph, we will add paths. We show how each BG-operation can be equivalently stated as the addition of a special path to a subdivision of $G$.

Let $K_4, G_5, \ldots, G$ be the construction sequence for $G$ using BG-operations. Each $G_i$ corresponds to a unique $G_i$-subdivision $S_i$ in $G$. The nodes of $G_i$ are the nodes of $S_i$ with degree at least three. We call them real nodes. The links in $S_i$ are the unique paths that contain real end nodes but no other real nodes, i.e. they are edges in $\text{smooth}(G)$. We define the following operations on $S_i$ and show how they correspond to BG-operations:

**Definition 1.2.** A Barnette-Grünbaum path (BG-path) for a subgraph $S_i \subset G$ is a path $P = x \rightarrow^*_G y$ with the following properties:

1. $P$ has only its end nodes in common with $S_i$
   
   \[ S_i \cap P = \{x, y\} \]

2. Every link of $S_i$ that contains both $x$ and $y$ simultaneously, contains them as end nodes

3. If $x$ and $y$ are inner nodes of two links $L_x$ and $L_y$ of $S_i$, respectively, and $|V_{\text{real}}(S_i)| \geq 4$, then the links are not parallel.

**Lemma 1.2 ([14]):** Adding a BG-path to $S_i$ corresponds to a BG-operation on $G_i$ and vice versa.

**Proof.** Let $P = x \rightarrow^*_G y$ be a BG-path. There are several cases

- $x$ and $y$ are contained in one link as end nodes. In this case, adding $P$ to $S_i$ corresponds to adding the edge $(x, y)$ to $G_i$ (operation 1.3(a)).
• \(x\) and \(y\) are contained in two links. Then, depending on whether none, one, or both nodes are real, adding the path corresponds to adding an edge to \(G_l\) (operation 1.3(a)) or subdividing one (operation 1.3(b)), respectively two edges of \(G_l\) to create \(x\) and \(y\) and adding an edge between them. The latter case corresponds to operation 1.3(c), as the links \(L_x\) and \(L_y\) that contain \(x\) and \(y\) are not parallel. In both cases, the subdivision is correct because of BG-properties 2 and 3. See figure 1.4.

(a) Both endnodes of the red BG-path are contained in the same link. The red node is not an endnode of the link.
(b) Both endnodes of the red BG-path are contained in the same link, but both as inner nodes.
(c) The red BG-path subdivides two parallel links.

Figure 1.4: BG-properties 2 and 3 prevent these situations.

It is easy to see that the equivalence also holds in the other direction. \(\square\)

Finding the largest \(K_4\) subdivision, as required by theorem 1.2.2, is a hard problem because it relies on finding longest paths. Luckily, the existence of construction sequences is not dependent on the choice of the subdivision as the following result shows.

**Lemma 1.3** ([14, Theorem 32]): Let \(G\) be a simple 3-connected graph and let \(H \subset G\) be a subdivision of either \(K_{2,3}\), where \(K_{n,m}\) is the complete graph on \(n\) nodes with \(m\) parallel edges between each pair of nodes (figure 1.5), or a 3-connected graph. Then, there is a BG-path for \(H\) on \(G\). If \(H\) is a \(K_{2,3}\) subdivision, there is a BG-path for \(H\) in \(G\) that generates a \(K_4\) subdivision. Moreover, for every link \(L\) in \(H\) of length at least 2, \(L\) or a parallel link of \(L\) contains an inner node on which a BG-path for \(H\) starts.

Figure 1.5: The graph \(K_{2,3}\)
1.2. PRELIMINARIES

Proof. We distinguish two cases.

- First let \( H \neq smooth(H) \), i.e. \( H \) contains a nonreal node. Then, \( H \) must contain a link of length at least two. Let \( L = a \rightarrow^*_H b \) be such a link, let \( x \) be an inner node of \( L \), and let \( I \) be the union of inner nodes of all parallel links of \( L \) (including \( L \)). We show that there is a BG-path for \( H \) that starts at a node in \( I \).

Assume that \( H \) is a subdivision of a 3-connected graph. Since \( H \) contains at least four real nodes and \( G \) is 3-connected, there exists a path \( P = x \rightarrow^*_G \{a,b\} y \) with \( y \in V(H) \setminus I \setminus \{a,b\} \). \( P \) has BG-property 2 because \( x \in L \) and \( y \notin L \). We construct a subpath of \( P \) that also has BG-properties 1 and 3. Take \( x' \) as the last node on \( P \) that is in \( I \), and let \( y' \) be the first node on \( P \) that is in \( V(H) \setminus I \). Then, \( P' = x' \rightarrow^*_P y' \) has no inner nodes in \( H \), hence it fulfills BG-property 1. Consequently, \( x' \) and \( y' \) are not inner nodes of parallel links \( L_{x'} \) and \( L_{y'} \) because \( x' \) is in \( I \) but \( y' \) is not. Therefore, \( P' \) is a BG-path for \( H \) that starts at a node in \( I \).

On the other hand, assume that \( H \) is a \( K_{2,3} \) subdivision. Then, as \( G \) is simple, at least one parallel link \( L' \neq L \) of \( L \) has length at least two and thus an inner node \( y \). It follows that a path \( P = x \rightarrow^*_G \{a,b\} y \) with \( y \in I \setminus V(L) \) exists. As before, BG-property 2 is satisfied because \( x \) and \( y \) are nonreal inner nodes of different links. We truncate the path to make sure that properties 1 and 3 also hold. Let \( x' \) be the last node on \( P \) that is contained in \( L \) and let \( y' \) be the first node in \( P \) that is contained in \( I \setminus V(L) \). Then, the path \( P' = x' \rightarrow^*_P y' \) is a BG-path for \( H \) that generates a \( K_4 \) subdivision.

- Now, let \( H = smooth(H) \), i.e. all nodes in \( H \) are real. Then, all links have length one. Because \( G \) is simple, so is \( H \), and hence it can not be a \( K_{2,3} \) subdivision. Since \( H \subset G \) there must be a node in \( V(G) \setminus V(H) \), or an edge in \( E(G) \setminus E(H) \).

In the case that there is an edge \( e \in E(G) \setminus E(H) \) and \( V(H) = V(G) \), \( e \) must be a BG-path for \( H \) because both endnodes are real. If on the other hand there is a node \( x \in V(G) \setminus V(H) \), then, by the 2-connectivity of \( G \) and lemma 1.2.1, we can find a path \( P = y_1 \rightarrow^*_C x \rightarrow^*_C y_2 \) with \( P \cup H = \{y_1,y_2\} \) that contains \( x \) as inner node. We construct it by concatenating the two paths from \( x \) to \( V(H) \) and truncating them such that only \( y_1 \) and \( y_2 \) are in \( H \). This path is a BG-path because all links in \( H \) have length one.

\( \Box \)

From this result and theorem 1.2.3 the next lemma follows immediately:

**Lemma 1.4** ([14]): The choice of the \( K_4 \) subdivision is not crucial. \( \Box \)

We now have the theoretical basis needed to start the discussion of the algorithm.
Chapter 2

Schmidt’s Algorithm

2.1 Outline

Schmidt’s algorithm finds a construction sequence of BG-paths for a graph $G$ to certify its 3-connectivity. If $G$ is not 3-connected, a separation pair is extracted.

The algorithm doesn’t start by finding a $K_4$ subdivision but instead builds an initial $K_{2,3}$ subdivision out of a DFS tree $T$ of $G$. Because of lemma 1.3 there exists a BG-path that turns the $K_{2,3}$ subdivision into a $K_4$ subdivision, if $G$ is 3-connected.

We construct the initial subdivision from three node disjoint parallel paths $C_0, C_1, C_2$. The latter two paths start at a back edge from the root of $T$ and end at the lowest common ancestor of the back edges’ endpoints. Then, $C_0$ connects the endpoints of $C_1$ and $C_2$. Finding the initial subdivision while performing a DFS is discussed in section 2.3.

After the initial subdivision is found, we proceed to partition the whole graph into paths like $C_1$ and $C_2$ that start at a back edge and continue in the DFS tree until they reach a previously created path. These paths will be called chains. We define a parent relationship on the chains: For a chain $C_i$ the parent is the chain that contains its last node. Chains and the construction of the chain decomposition is discussed in section 2.4.

The chains nearly form an ear decomposition of $G$, i.e. the $i$-th chain $C_i$ has exactly two nodes in common with $\bigcup_{j<i} C_j$: its start and endpoint. The only difference to an ear decomposition is the starting point; instead of starting with a cycle, we start with the chain $C_0$.

That chains have only two nodes in common with other chains suggests that some chains might be BG-paths, and indeed this is the case. Unfortunately, not every chain forms a BG-path. In section 2.5 we show how the chains are grouped into one of five classes: 1, 2a, 2b, 3a, and 3b.

Chains of some classes will be BG-paths themselves, while other chains need to be grouped into larger structures called caterpillars. Those are relatively easy to
decompose into a set of BG-paths. Section 2.7 introduces important properties of caterpillars and section 2.10 discusses how their decomposition works.

The algorithm tries to construct a set of BG-paths from the chains. To structure this process, it is necessary to impose additional restrictions on the construction sequence: We want to add nodes to the subdivision such a way that it stays upwards-closed and modular. A subdivision is upwards-closed if for each node in the subdivision also its parent in $T$ is in the subdivision. It is modular if it is composed of chains. These restrictions and some of their implications are discussed in section 2.8.

With the restrictions to guide us, we can examine under which conditions a single chain or a caterpillar can be added as one or more BG-paths. The results from section 2.9 and 2.10 will be useful throughout the main part of the algorithm.

The actual computation of the construction sequence takes place in a loop that iterates over all chains in the order in which they were created. When a chain is processed, it must already be contained in the subdivision. After its processing is finished, all chains that start at one of its nodes will have been added too, if $G$ is 3-connected. Otherwise, the processing might fail and a separation pair is found.

For each chain $C_i$ we process, we partition the chains that have an endpoint on $C_i$ according to the segments of the current subdivision. Segments are connected components that don’t contain edges from the subdivision. How to find the segments efficiently is discussed in section 2.11.1. Then, one after the other, each segment is added to the subdivision. The order in which they are added is not always arbitrary though, some segments are easy and can be added right away as described in section 2.11.2, while other segments are hard.

Adding the hard segments is the most challenging part of the algorithm, both in conceptual complexity as well as in computational demand. Unlike the easy segments, hard segments can’t be added directly; it is necessary to compute a proper order in which to process them. To find it, we map the problem on the original graph to an auxiliary problem on sets of intervals. This mapping is described in section 2.11.3.

With the intervals at hand, we define a binary relation, interlocking, on them: two intervals $[a, b], [c, d]$ interlock if $a < c < b < d$. This relation induces the interlock graph on the intervals. Finding a proper order in which to add segments reduces to checking whether the interlock graph is connected. As it may have quadratically many edges, the straightforward approach can’t be used. Instead, we employ the algorithm of Olariu and Zomaya [11] to solve this problem in linear time. A full discussion of the algorithm can be found in section 2.12.

Finally, we show in section 2.13 in detail, how a BG-certificate is constructed, and, more importantly, how to verify it in linear time.

All theoretical results in this chapter are taken from Schmidt’s PhD thesis [14], with the exception of the results in section 2.12, which come from Olariu and Zomaya’s
paper [11]. In most cases, we provided additional detail in the proofs to make them easier to understand.

### Algorithm 1 Schmidt’s algorithm for certifying 3-connectivity

```plaintext
procedure SCHMIDT(G=(V,E))
    perform preliminary checks
    INITIAL_DFS(G) ▶ section 2.2
    CHAIN_DECOMPOSITION(G) ▶ section 2.3
    check property B ▶ section 2.4
    for chain $C_i$, $i = 0 \ldots$ number of chains do
        for $D \in children_{12}(C_i)$ do
            if $D$ has type 2a then
                Output $D$ as BG-path
            end if
        end for
        PARTITION INTO SEGMENTS(type3($C_i$)) ▶ section 2.11.1
        for easy segment $H$ do
            for $C_j \in type3(C_i) \cap H$ do
                ADD WITH ANCESTORS($C_j$)
            end for
        end for
        ADD_HARD_SEGMENTS ▶ section 2.11.3
    end for
end procedure
```

## 2.2 Initial Checks

Before we start the actual algorithm, we check the graph for several basic properties: We ensure that it has at least four nodes and that it is connected by doing a DFS. We also check that each node has degree at least three.

```plaintext
Listing 2.1: Initial Checks

1 if (the_graph.number_of_nodes()<4)
2     throw not_triconnected_exception("graph has fewer than four nodes");
3 if (!is_connected(the_graph))
4     throw not_triconnected_exception("graph is not connected");
5 node min_degree_node = the_graph.first_node();
6 { node v;
7     forall_nodes(v, the_graph) {
8         if (the_graph.degree(v) < the_graph.degree(min_degree_node))
9             min_degree_node = v;
10     } }
11 degree_three_or_more(the_graph, min_degree_node);
```
The function degree_three_or_more checks the degree of a node and, if it is too small, extracts a separation pair, respectively an articulation point, from its neighbours.

```cpp
void degree_three_or_more(const ugraph& g, node v)
throw(not_triconnected_exception)
{
    switch (g.degree(v)) {
    case 1: {
        throw not_triconnected_exception(
            "graph contains a degree 1 node",
            opposite(g.first_adj_edge(v), v));
    } break;
    case 2: {
        node a = opposite(g.first_adj_edge(v), v);
        node b = opposite(g.last_adj_edge(v), v);
        throw not_triconnected_exception(
            "graph contains a degree 2 node", a, b
        );
    } break;
    default: break;
    }
}
```

When performance is of greater concern, these checks can be incorporated into the later stages of the algorithm. Connectivity, for example, can be checked by counting the number of chains found during the chain decomposition (see section 2.4). The graph is connected iff \( m - n + 1 \) chains are found [14]. However, profiling indicates that the initial checks only account for negligible amounts of the total runtime.

### 2.3 Finding the Initial Subdivision

To find BG-paths efficiently, we want to decompose the graph into a series of chains. Chains will be discussed in more detail in the next section. For now, they are edge disjoint paths that start at a back edge and continue upwards in the tree until they hit a node that is contained in a previously created chain.

The chain decomposition starts with an initial \( K_{2,3} \) subdivision, in which each subdivided edge is one chain. In this section, we describe how to find that subdivision while performing a DFS on the graph.

To see how we can find the initial subdivision, we note some properties of DFS trees in 3-connected graphs. Let \( T \) be a DFS tree of a 3-connected graph \( G \), let \( r \) be its root, and let \( u \) be the second node that is visited.

**Lemma 2.1** ([14, without proof]): If \( G \) is 3-connected, \( r \) has exactly one child, \( u \), in \( T \). Further, \( u \) has exactly one child too.
2.3. FINDING THE INITIAL SUBDIVISION

Figure 2.1: In a DFS tree of a 3-connected graph, the first two nodes must have exactly one child.

Proof. If \( r \) has at least two children \( c_1, c_2 \), removal of \( r \) disconnects the graph, as there can be no paths between the subtrees of \( c_1 \) and \( c_2 \) that don’t contain \( r \) due to the DFS structure of \( T \). Hence, the graph can not be biconnected.

If on the other hand \( u \) has at least two children \( c_1, c_2 \), the removal of \( r \) and \( u \) disconnects the graph, as there are no paths between the subtrees of \( c_1 \) and \( c_2 \) that don’t cross \( r \) or \( u \), again due to the structure of \( T \). Therefore, the graph can not be 3-connected.

Corollary 2.1: At least two back edges start at \( r \) in \( T \).

Proof. As \( r \) must have degree at least three, but has only one child, at least two back edges must end at \( r \).

Using this corollary, we can construct the initial subdivision as follows: We take two back edges \( r \leftrightarrow a \), \( r \leftrightarrow b \) and find the least common ancestor \( l \) of \( a \) and \( b \) in \( T \). As \( r \) has only one child in \( T \), \( l \) is guaranteed to be distinct from \( r \). We take \( K_{2,3} = C_0 \cup C_1 \cup C_2 \) to be the following paths:

\[
C_0 = l \rightarrow_T^* r \quad C_1 = r \leftrightarrow a \cup a \rightarrow_T^* l \quad C_2 = r \leftrightarrow b \cup b \rightarrow_T^* l
\]

We interleave finding the back edges and \( l \) with a DFS. It starts by a call to the function \texttt{find_cycle_to_root}. To avoid code duplication, the same function will also be used to continue the DFS after the initial subdivision is found. This adds a bit of complexity.

The function takes five arguments:

Listing 2.2: Finding a cycle to the root

```cpp
bool schmidt_triconnectivity::find_a_cycle_to_root(
    node& start_node,
    edge& backedge,
    unsigned int& number_seen,
    edge_array<bool>& seen_edge,
    bool continue_after_found) throw() {
```
1. A reference to a node from which to start a DFS. It will be is set to the last examined node.

2. A reference to an edge that is set to a back edge to \( r \).

3. A reference to an int that is used to assign a Depth-First Index (DFI) to each node.

4. A reference to an edge_array<bool> that keeps track of edges that have already been visited

5. A bool that switches the mode of operation: if it is set to false, the function returns after a back edge to the root has been found; otherwise it continues with the DFS.

The function find_cycle_to_root performs a DFS as usual. It pushes edges on a stack and examines them one after the other until the stack becomes empty. Besides finding cycles to the root, it also assigns a DFI to each node, constructs an edge_array to mark back edges, and computes a parent relation on nodes and edges. It returns true if a cycle to the root is found.

```cpp
stack<edge> stack;

node current_node = start_node;
{ edge e;
  forall_adj_edges(e,current_node) {
    if (seen_edge[e])
      continue;
    stack.push(e);
  }
}

bool found_cycle = false;
while(!stack.empty()) {
  const edge current_edge = stack.pop();
  if (seen_edge[current_edge]) {
    continue;
  }
  seen_edge[current_edge] = true;
}
```

The edges in LEDA have a source and a target even in undirected ugraph objects. To make things easier later on, we will make sure that tree edges point towards the root and back edges are oriented in the opposite direction. At this point in the DFS, the source of the current edge, i.e. the node to be visited next, must always have a lower DFI: Either the node is still unvisited and has DFI 0, or the edge is a back edge and must point away from the root.
2.3. FINDING THE INITIAL SUBDIVISION

```cpp
26 current_node = target(current_edge);
27 node next_node = source(current_edge);
28
29 if (dfi(next_node) > dfi(current_node)) {
30     std::swap(next_node, current_node);
31     the_graph.rev_edge(current_edge);
32 }

Afterwards, we check whether the new edge is a back edge and, if it is, whether it
starts at the root and hence closes a cycle. In that case, we pass the current node and
the current edge to the calling function via the reference parameters. Depending on
the current mode of operation as indicated by continue_after_found, we either
return, or continue with the DFS after a cycle to the root was found.

```cpp
33 if (dfi(next_node)>0) {
34     mark_as_backedge(current_edge);
35     if (dfi(next_node) == 1 && !found_cycle) {
36         start_node = current_node;
37         backedge = current_edge;
38         found_cycle=true;
39         if (!continue_after_found) {
40             return found_cycle;
41         }
42     }
43     continue;
44 }

//continue exploration
46 parent[next_node] = current_edge;
47 current_node = next_node;
48 set_dfi(current_node,number_seen++);
49 {
50     edge next_edge;
51     forall_adj_edges(next_edge,current_node) {
52         if (seen_edge[next_edge]) {
53             continue;
54         }
55         stack.push(next_edge);
56     }
57 }
58
59 }
60 return found_cycle;
```

We proceed to use the function find_cycle_to_root to perform a DFS to find
$C_0$, $C_1$ and $C_2$. First, we call find_cycle_to_root once to find the first back edge
and its endpoint $v$. 

To explore the remaining part of the graph and to locate the least common ancestor, we walk upwards on the path from \( v \) to the root and call \( \text{find_cycle_to_root} \) on each node. The first node from which the search reports a back edge to the root becomes the least common ancestor and thus the endpoint of \( C_1 \) and \( C_2 \). We set \textit{continue_after_found} to \texttt{true} to complete the DFS.
2.3. FINDING THE INITIAL SUBDIVISION

It is not necessary to check whether the root has only one child. Then, \( C_1 \) will form a cycle, and an exception will be thrown during the chain creation. Similarly, we don’t check explicitly that the root’s first child has only one child itself. This case will be detected when we try to construct a BG-decomposition from the chains.

We now have the information needed to construct the initial three chains. \( C_0 \) needs special treatment because it is the only chain that doesn’t start with a back edge. The other chains are built using the function `create_chain` which will be discussed in detail in section 2.4.

We add all three chains to the current subdivision, marking their endpoints as real, and include them as BG-paths in the certificate. The implementation of the certificate will be discussed in section 2.13, and the details of the function `add_to_subdivision` can be studied in appendix B.5.

```c++
    lca == NULL && node_b == NULL)
{  
    lca = current_node;
    node_b = inner_search_cur_node;
    chain2_first_edge = e;
}  
    current_node = parent_node(current_node);
    if (lca==NULL) {
      throw not_triconnected_exception("root on only one cycle",root);
    }
    chains.push_back(new chain());
    chains[0]->number = 0;
    chains[0]->first_edge = parent[lca];
    chains[0]->set_s(lca);
    mark_path(lca, 0);
    classify_chain(0);
    create_chain(chain1_first_edge);
    create_chain(chain2_first_edge);
    add_to_subdivision(chains[0]);
    cert->add_bg_path(chains[0]);
    add_to_subdivision(chains[1]);
    add_to_subdivision(chains[2]);
    cert->add_bg_path(chains[1]);
    cert->add_bg_path(chains[2]);
    }
```
2.4 Chain Decomposition

After the initial \( K_{2,3} \) subdivision is found, we construct a complete path covering of \( G \) by chains.

A chain \( C_i \neq C_0 \) is a path that starts at a back edge \( u \rightarrow v \) and continues on the tree path \( v \rightarrow^* r \) until it reaches an edge \( x \rightarrow y \) that is already part of a chain \( C_k \). Then, \( C_k \) is the parent of \( C_j \), \( u \) is the startnode, \( s(C_i) \); and \( x \) is the endnode, \( t(C_i) \), of \( C_i \).

The nodes \( V(C_i) \setminus \{s(C_i), t(C_i)\} \) are the inner nodes of \( C_i \). Each node \( v \) of the graph is the inner node of exactly one chain (or none, if the graph is not biconnected). However, \( v \) can be contained in up to \( \text{deg}(v) \) many chains, as chains share their endnodes. An edge is contained in a chain if both endpoints are contained in the chain. This implies that all chains are edge disjoint. The parent relation defines an order on the chains. We say \( C_i < C_j \) if \( C_i \) is an ancestor of \( C_j \). The chain \( C_0 \) is special among the chains because it doesn’t have a parent, doesn’t start at a backedge, and contains all its nodes as inner nodes.

![Figure 2.2: A chain \( C_i \) with parent \( C_k \). The node \( s(C_i) \) is shown in green, \( t(C_i) \) in red.](image)

Besides a pointer to their parent and their start- and endnode, chains also store a few more values and flags. We will discuss these at their point of use. For a full reference, see appendix A.1.

The first three chains are already created and form a \( K_{2,3} \) subdivision. All other chains are constructed by iterating over the back edges in ascending order of their starting point’s DFI. This iteration is done in the function \texttt{chain_decomposition}.

### Listing 2.4: Chain Decomposition

```c
void schmidt_triconnectivity::chain_decomposition(void)
  throw(not_triconnected_exception)
{
  for(unsigned int cur_df=1;
      cur_df<(unsigned int)the_graph.number_of_nodes()+1;
```
2.4. CHAIN DECOMPOSITION

When we create a new chain from a back edge \( u \rightarrow v \) in the function `create_chain`, we set the startnode to \( u \). Then, we call the function `mark_path` to mark all edges on the chain with the chain’s number, set the endnode of the chain to the first node on the path \( v \rightarrow T_r \) that is contained in a previously created chain, and set the parent. We will discuss this function in more detail afterwards.

**Listing 2.5: Creating a chain**

```c++
void schmidt_triconnectivity::create_chain(edge e)
throw(not_triconnected_exception)
{
    chain* current_chain = new chain();
    chains.push_back(current_chain);
    const int chain_number = chains.size() - 1;
    belongs_to_chain[e] = chain_number;
    current_chain->set_s(source(e));
    current_chain->first_edge = e;
    mark_path(target(e), chain_number);
}
```

We need to make sure that no chain forms a cycle; otherwise the graph is not biconnected.

**Lemma 2.2 ([14, Lemma 51]):** Let \( T \) with root \( r \) be a DFS tree of a connected graph \( G \) with minimal degree \( \delta(G) \geq 3 \). If a chain \( C_i \) forms a cycle, \( G \) is not biconnected.

**Proof.** We create chains from back edges in ascending order of their starting point’s DFI. This implies that all back edges ending at a node in a subtree \( T(u) \) that is rooted at the last inner node \( u \) of \( C_i \) must start at descendants of \( s(C_i) \). Otherwise they have already been turned into chains before \( C_i \) is created. Then, \( u \) could not be an inner node of \( C_i \) because of the DFS structure of the tree. This implies with \( s(C_i) = t(C_i) \) that there is no edge from the subtree \( T(u) \) to the rest of \( T \), and hence \( s(C_i) \) is an articulation point of \( G \). \(\square\)

Schmidt [14] also shows the other direction, i.e. \( G \) is biconnected if no chain forms a cycle.
if (current_chain->get_s() == current_chain->get_t()) {
    throw not_triconnected_exception(
        "A chain forms a cycle", current_chain->get_s());
}

After the chain is completely constructed, we classify it as described in section 2.5 into one of the classes 1, 2a, 2b, 3a, or 3b.

Chains store a list, type3, of all type three chains that start at their inner nodes and are not yet added to the subdivision. Whenever a new chain $C_i$ is classified as type three, we append it to the list of the chain that contains $s(C_i)$ as inner node. Note that the chains in type3 are ordered.

classify_chain(chain_number);
switch (current_chain->get_type()) {
    case three_a: case three_b: {
        const node s = current_chain->get_s();
        chain* const chain_at_s = chains[inner_of_chain[s]];
        current_chain->add_to_type3_off(chain_at_s);
    }
    default: break;
}

The function mark_path walks upwards on the tree path from the node start and marks all edges and all inner nodes with the number of the current chain. If more than one edge is contained in the chain, we set is_backedge to false.

When the current edge is either NULL or is already contained in a chain, the walk stops(s((aeu and the parent of the chain is set. Note that the only chain whose parent is NULL is $C_0$.

Besides the list type3, chains also store a list, children12, that contains all their children of type one or two that are not yet added to the subdivision. The method set_parent of the chain class updates the new parent’s list.

Listing 2.6: Marking the edges on a chain

```cpp
void schmidt_triconnectivity::mark_path(const node start, const unsigned int the_chain) throw()
{
    chains[the_chain]->number = the_chain;
    node current_node = start;
    for (edge current_edge = parent[start];
        true;
        (current_node = parent_node(current_node),
        current_edge = parent[current_node]))
```
2.5. CHAIN CLASSIFICATION

```cpp
{  
    if (current_edge == NULL || belongs_to_chain[current_edge]>=0) {  
        chains[the_chain]->set_parent(current_edge == NULL ?  
            NULL : chains[belongs_to_chain[current_edge]]);  
        chains[the_chain]->set_t(current_node);  
        break;  
    }

    chains[the_chain]->is_backedge = false;  
    inner_of_chain[current_node] = the_chain;  
    belongs_to_chain[current_edge] = the_chain;  
}
```

The chain decomposition has a few properties that will prove useful in later lemmas. We don’t give proofs here because the lemmas are intuitively true due to the way we constructed the chains.

**Lemma 2.3** ([14, Lemma 56]): Let $C_k \neq C_0$ be a chain with child $C_i$. Then, $k < i$, $s(C_i)$ is a (not necessarily proper) descendant of $s(C_k)$ and $t(C_k)$ is a proper descendant of $t(C_i)$ in $T$.

**Lemma 2.4** ([14, Lemma 57]): The parent relation on the set of chains $C$ defines a tree $U$ with $V(U) = C$ and root $C_0$.

### 2.5 Chain classification

As the chains are constructed, we classify them as one of five types. Chains of some types are, under certain conditions, BG-paths and will subsequently be added to the subdivision. Other chains can’t be added as BG-paths themselves and must be grouped into caterpillars that can be decomposed into BG-paths. Caterpillars will be discussed in detail in section 2.7.

**Listing 2.7: Chain classification**

```cpp
void schmidt_triconnectivity::classify_chain(  
    const unsigned int chain_number) throw()  
{  
    if (chain_number == 0)  
        return; // first chain stays unmarked  
    chain* const the_chain = chains[chain_number];  
    chain* const the_parent = the_chain->get_parent();
```

We begin by checking whether the chain is of type one (Figure 2.3(a)). A chain $C_i$ with parent $C_k$ has type one if the path $t(C_i) \rightarrow^*_T s(C_i)$ is contained in $C_k \setminus s(C_k)$. This is always the case if $C_k = C_0$, as $t(C_i)$ must be an ancestor of $s(C_0)$, and $s(C_i)$ is a proper ancestor of $t(C_i)$ with lemma 2.3. Otherwise, it suffices to check whether
22

CHAPTER 2. SCHMIDT’S ALGORITHM

\[ C_k \]

(a) \( C_i \) is a type one chain: the endpoints are contained in the parent \( C_k \)

\[ C_i \]

(b) \( C_i \) of type 3a: Its endpoints are neither contained in \( C_k \) nor does it start at \( s(C_k) \)

Figure 2.3: Chain classes one and 3a.

\( s(C_i) \neq s(C_k) \) and \( s(C_i) \) is contained in \( C_k \). As \( t(C_i) \) must be contained in \( C_k \) by the definition of the parent relation, \( s(C_i) \) is a descendant of \( s(C_k) \) because of lemma 2.3 and there being no back edges in \( C_i \) besides the first edge.

\[
\text{if } (\text{the\_parent->number} == 0 | |
\text{(the\_chain->get\_s()) != the\_parent->get\_s()} \&\&
\text{contained\_in\_chain(the\_chain->get\_s(), the\_parent)}
\}
\text{the\_chain->set\_type(one);} \text{ return;}
\]

If \( C_i \) starts at the same node as its parent \( C_k \), it is a type two chain. There are two subtypes. If \( C_i \) is a back edge, it gets type 2a (Figure 2.4(a)). Otherwise it is a type 2b chain (Figure 2.4(b)) that has to be grouped into a caterpillar later on. To facilitate this grouping, we mark these chains.

\[
\text{if } (\text{the\_chain->get\_s()} == \text{the\_parent->get\_s()}) 
\text{if } (\text{the\_chain->is\_backedge}) 
\text{the\_chain->set\_type(two\_a);} 
\text{return;}
\text{else} 
\text{the\_chain->set\_type(two\_b);} 
\text{the\_chain->is\_marked = true;} 
\text{return;}
\]
2.5. CHAIN CLASSIFICATION

(a) \( C_i \) of type 2a: it is a back edge that starts at the same node as the parent \( C_k \)

(b) \( C_i, C_j \) are type 2b chains: they start at the same node as their parent without being back edges. \( C_l \) is a type 3b chain: its parent is the (marked) chain \( C_j \). \( C_i, C_j \) and \( C_l \) are grouped into a caterpillar.

Figure 2.4: Chain classes 2a, 2b, and 3b.

The remaining chains get classified as type three. Again, there are two subtypes: Type three chains whose parent is not marked get the type 3a (Figure 2.3(b)); if the parent is marked, the chain is of type 3b (\( C_l \) in figure 2.4(b)). The latter kind of chain starts a caterpillar. Whenever we encounter a 3b chain, we walk upwards in the chain tree \( U \) and collect all marked chains until we reach a chain that is unmarked. Chains that are added to a caterpillar are subsequently unmarked, ensuring that each chain of type 2b is contained in at most one caterpillar.

```c
if (!the_parent->is_marked) {
    the_chain->set_type(three_a);
    return;
} else {
    the_chain->set_type(three_b);
    chain* c_jay = the_parent;
    while(c_jay->is_marked) {
        c_jay->is_marked = false;
        caterpillars[chain_number].append(c_jay);
        c_jay = c_jay->get_parent();
    }
    return;
}
```

Figure 2.5: For property B to be fulfilled, the red back edge must exist. Otherwise $T(x)$ is disconnected from $G$ upon removal of $s(C_i)$ and $t(C_i)$

### 2.6 A Final Check

Recall that we checked whether a chain forms a cycle to ensure that the graph is biconnected (see line 14 in listing 2.5). The argument was that the whole subtree hanging at a cyclic chain $C_i$ becomes unreachable if the attachment node is removed because there can be no back edge that enters the subtree from a proper ancestor of $s(C_i)$.

As we are concerned about 3-connectedness, we must also worry about cases where $s(C_i) \neq t(C_i)$, and the removal of both endpoints disconnects the graph. If $C_i$ is not a cycle, the subtree $T(x)$ of the last inner vertex $x$ on $C_i$ is connected to the remaining graph at least by the endpoints. As argued in lemma 2.2, there can be no back edges from a proper ancestor of $s(C_i)$ entering the subtree. We need to make sure that there is at least one back edge from a proper descendant $s(C_i)$, i.e. an inner node of the path $t(C_i) \rightarrow_T s(C_i)$, that does. Schmidt [14] calls this Property B. A chain $C_i \neq C_0$ that is not a back edge has property B if there is a back edge $e = (u, v)$ with $u \in s(C_i) \rightarrow_T t(C_i)$ that enters the subtree $T(x)$ of the last inner node $x$ of $C_i$. See Figure 2.5. We say a chain decomposition $C$ has property B if all chains in $C$ have property B. Property B will be tremendously useful in later proofs.

The next lemma follows from the above observations.

**Lemma 2.5** ([14, Lemma 60]): If a chain $C_i$ does not have property B, $\{s(C_i), t(C_i)\}$ is a separation pair of $G$. 

Note that property B is not sufficient to prevent $\{s(C_i), t(C_i)\}$ from being a separation pair. If there is no back edge that enters $t(C_i) \rightarrow_T s(C_i)$ from an ancestor of $s(C_i)$, the existence of the back edge $e$ does not help at all: $t(C_i) \rightarrow_T s(C_i)$ is just added to the connected component that is created upon removing $C_i$'s endpoints.
Lemma 61 in [14] explains how to check $C$ for property B. We employ a simplified scheme that is adapted from Prof. Rote’s implementation [12].

The crucial observation is that only type three chains $C_i$ start with the back edges that are relevant for property B, as their starting point lies on a different chain than their endpoint, and their starting point is different from their parent’s starting point. Let $C_k$ be the parent of $C_i$. Because of lemma 2.3 and the type of $C_i$, $s(C_k)$ is proper ancestor of $s(C_i)$. Thus, the parent of a type three chain must have property B. But then also the parent of $C_k$, $C_k$, must have property B as long as $t(C_k)$ is still a descendant of $s(C_i)$, i.e. as long as $C_k$ is not the chain that contains, or ends at, $s(C_i)$. Clearly, this argument allows us to traverse the path in the chain tree $U$ from $C_i$ upwards until such a chain is encountered and mark each chain as having property B. To get a linear runtime we stop the traversal once a chain is reached that is already marked.

The function check_prop_b determines whether each chain satisfies property B. It starts by marking the chains as described above and subsequently checks whether all chains that are not back edges have been marked.

### Listing 2.8: Final Check

```c++
void schmidt_triconnectivity::check_prop_b(void) const
throw(not_triconnected_exception)
{
    int_set marked(chains.size());
    marked.insert(0);

    //chains 1 and 2 have type 1
    for(unsigned int i=3; i<chains.size(); i++) {
        if ((chains[i]->get_type() == three_a ||
             chains[i]->get_type() == three_b)
            { const node s = chains[i]->get_s();
             const unsigned int chain_at_s = (unsigned int)inner_of_chain[s];
             for(chain* c = chains[i]->get_parent();
                 !marked.member(c->number) &&
                 !(c->number == chain_at_s || s == c->get_t());
                 c = c->get_parent())
                 { marked.insert(c->number); }
            } }

    for(unsigned int i=0; i<chains.size(); i++) {
        if (!marked.member(i) && !chains[i]->is_backedge) {
            throw not_triconnected_exception("Property B",
                chains[i]->get_s(), chains[i]->get_t());
        } }
```

**Lemma 2.6:** The function check_prop_b is correct.

**Proof.** We show both directions. First we show that there are no false negatives. Let \( C_i \neq C_0 \) be a contradiction that has property B and is not a back edge, but assume for the sake of contradiction that \( C_i \) was not marked by the procedure. As \( C_i \) has property B, there must be an edge that starts at an inner node of \( t(C_i) \rightarrow T s(C_i) \) and enters the subtree \( T(x) \) of the last inner node \( x \) of \( C_i \). Let \( e = (u, v) \) be the first such edge that was traversed during the chain composition, i.e. \( u \neq s(C_i) \) is minimal, and let \( D_0 \) be the chain that it starts. Let \( C_s \) be the chain that contains \( u \) as an inner node.

As \( e \) enters \( T(x) \), the chain \( C_i \) must be an ancestor of \( D_0 \). Further, it must be a type three chain, because \( u \) lies on \( s(C_i) \rightarrow T s(C_i) \) and \( u \neq s(C_i) \). The only other chains that may enter \( T(x) \) before \( D_0 \) must all start at a node \( v \geq s(C_i) \) because of lemma 2.3. With the minimality of \( u \) we get \( v = s(C_i) \).

Let \( D_0, D_1, \ldots, D_k, C_i \) be the ancestors of \( D_0 \) in \( U \). If \( D_i \neq C_s \) for \( 1 \leq i \leq k \), \( D_0 \) causes \( C_i \) to be marked. But this is obviously the case because of lemma 2.3 and the fact that \( C_i \neq C_s \).

Now let us prove that there are no false positives. Let \( C_i \neq C_0 \) be a chain that is not a back edge and does not have property B, but was marked by the procedure. Let \( e = (u, v) \) be the back edge that initiated the traversal that marked \( C_i \), and let \( D_0 \) be the chain that starts with \( e \). As \( D_0 \) is a descendant of \( C_i \) and has type three, \( s(C_i) < u \). Because \( C_i \) does not have property B, \( u \) is not on \( t(C_i) \rightarrow T s(C_i) \). Hence \( u \) must be a descendant of \( t(C_i) \). But then \( C_i \) is not marked due to the loop condition in line 17.

**Corollary 2.2:** If a chain \( C_i \neq C_0 \) has property B and is not a back edge, there exists a type three chain among its ancestors that starts on an inner node of \( t(C_i) \rightarrow T s(C_i) \).

Schmidt [14] also describes a simplified way to check for property B during a later stage of the algorithm. Note however that the elaborations in [14] contain an minor error. To make the presentation of the algorithm clearer, we decided against implementing the simplification. Profiling shows potential performance gains to be negligible.

### 2.7 Caterpillars

We intent to iteratively add the chains as BG-paths until the whole graph is covered. Unfortunately, not all chains are BG-paths themselves. In this section we deal with those chains that can’t be added as paths themselves, but are instead grouped into caterpillars. As is obvious from the way we create them in listing 2.7, line 29ff, a caterpillar is a list of chains that always starts with a chain of type 3b, followed by one or more chains of type two. Further, every chain of type 2b is contained...
in at most one caterpillar. The parent of a caterpillar is the parent of the minimal chain in the caterpillar. A caterpillar starting from a chain \( C_i \) is called \( L_i \). For implementation details on caterpillars see appendix A.2.

Caterpillars have the following useful properties:

**Lemma 2.7** ([14, Lemma 65]): After the chain classification there are no chains of type 2b that are not contained in a caterpillar if every \( C_i \) has property B.

**Proof.** Let \( C_i \) be a type 2b chain that is not contained in a caterpillar after the chain classification. Let \( x \) be the last inner node on \( C_i \). As \( C_i \) is of type 2b, \( C_i \neq C_0 \). Because it has property B, there is a type three chain that enters \( T(x) \). Let \( C_j \) be the minimal such chain. Then, the other chains that enter \( T(x) \) and are constructed before \( C_j \) must start at \( s(C_i) \) and hence are of type two (cf. lemma 2.6). Of those, only the type 2b chains can be ancestors of \( C_j \) because 2a chains are back edges and have no children. This implies that all ancestors of \( C_i \) up to and including \( C_i \) are marked during the construction of the caterpillar \( L_j \) and hence are added to \( L_j \), contradicting our assumption.

The next lemma follows from these considerations.

**Lemma 2.8** ([14, Lemma 66]): Let \( L_i \) be a caterpillar, and let \( D_k \) be the minimal chain in \( L_i \). Then, \( s(C_i) \) is an inner node of \( t(D_k) \to^* s(D_k) \).

We distinguish two types of caterpillars. The first, good, encompasses caterpillars, for which we will be able to find a BG-decomposition as hinted at in figure 2.6 and described in detail in section 2.10, whereas at a certain point in the algorithm the existence of a bad caterpillar implies a connectivity lower than three.

**Definition 2.1.** Let a caterpillar \( L_i \) with parent \( C_k \) be bad for a subdivision if \( s(C_i) \) is contained in \( C_k \), and \( s(C_i) \to^* s(C_k) \) contains no real inner node.

**Definition 2.2.** Let a caterpillar \( L_i \) with parent \( C_k \) be good for a subdivision if \( s(C_i) \notin V(C_k) \), i.e. \( s(C_i) \in t(C_k) \to^* s(C_k) \), or \( s(C_i) \to^* s(C_k) \) contains a real inner node. See figure 2.6

Note that during the run of the algorithm, a caterpillar may change its status from bad to good because we keep adding real nodes each time we add a chain. In fact, if the graph is 3-connected, all caterpillars must eventually become good, because only good caterpillars can be decomposed to BG-paths. We defer the proof of the following lemma until section 2.10.

**Lemma 2.9:** Let \( L_i \) be a caterpillar with parent \( D \). If \( D \) is in the subdivision and \( L_i \) is good, \( L_i \) can be added.

To save some case distinctions, let a cluster either be a caterpillar or a single chain that is not contained in a caterpillar. We extend the ordering on chains to be applicable to clusters: For two clusters \( C_a, C_b \) let \( C_a < C_b \) if there is a chain \( A \in C_a \) and a chain \( B \in C_b \) such that \( A < B \).
A BG-construction sequence for a graph \( G \) adds paths to a growing subdivision until \( G \) is fully created. We need to impose further restrictions to allow us to handle chains in the order in which they were created, and to make ensuring BG-property 3 easier.

**Definition 2.3.** Let a subdivision \( S_l \subseteq G \) be upwards-closed if for each node in \( S_l \) the edge to its parent is in \( E(S_l) \). Let \( S_l \) be modular if \( S_l \) is the union of chains.

We would like to only have upwards-closed and modular subdivisions during the algorithm, because this means that the chains are added in a preorder on \( U \). Unfortunately, there are cases where no BG-path for a subdivision is a chain. For an example see the graph in figure 2.7. If we take \( S_3 = \{ C_0, C_1, C_2 \} \), node \( a \) doesn’t have enough adjacent edges, and nodes \( b \) and \( d \) aren’t real. We can’t add another parallel edge since \( c \) already has its final degree. Thus the only valid BG-operation is subdividing two edges between \( a \) and \( c \) to create \( b \) and \( d \) and connecting them (an operation of type (c)). There are two corresponding BG-paths: \( b, e, d \) and \( b, f, e, d \), but both contain edges from different chains. Note that the chains \( C_6, C_4, \) and \( C_3 \) form a caterpillar in which the path \( b, f, e, d \) is contained.

Therefore, the modularity of \( S_l \) can’t always be preserved and we have to weaken that requirement. We allow intermediate subdivisions that are neither modular nor
2.9. WHEN INDIVIDUAL CHAINS ARE BG-PATHS

upwards-closed, as long as we have already found a set of $t$ BG-paths that restore upwards-closedness, and whose union is the union of $t$ distinct chains.

We further restrict links of $S_l$ that only consist of tree edges to have no parallel link, except for $C_0$ in $S_3$, to prevent BG-path candidates from violating property 3.

Summarizing, we apply the following restrictions:

(R1) For each upwards-closed and modular subdivision $S_l$, BG-paths are only added as

- single chains of type one, 2a or 3a, with $S_{l+1}$ being again upwards-closed and modular.
- sets of $t > 1$ subsequent BG-paths constructing an upwards-closed, modular subdivision $S_{l+t}$ that differs from $S_l$ in exactly $t$ chains of types 2b and 3b.

(R2) For each upwards-closed and modular subdivision $S_l$, the links of $S_l$ that consist only of tree edges of $T$ have no parallel links, except $C_0$ in $S_3$.

To emphasize the fact that a subdivision is upwards-closed, modular, and consists of exactly $l$ chains, we will from now on write $S_l^R$.

It is not clear that a construction sequence exists under our additional restrictions. This result will follow from the lemmas developed during the discussion of the main algorithm.

2.9 When Individual Chains Are BG-paths

In this section we will examine under which conditions a single chain is a BG-path. The results will be used during the discussion of the main algorithm to show the correctness in the more complicated cases.
First we discuss, how the restrictions from section 2.8 can be used to identify BG-paths more easily.

**Lemma 2.10** ([14, Lemma 72]): Every path \( P \) for \( S^R_l \) that has BG-properties 1 and 2 is a BG-path.

*Proof.* The first two BG-properties require that \( P \cap S^R_l = \{ x, y \} \) and that every link that contains \( x \) and \( y \) simultaneously contains them as end nodes. We need to show that the third BG-property holds, i.e. two links \( L_x \) and \( L_y \) that contain \( x \) and \( y \) respectively as inner nodes are not parallel except if \( S^R_l = S_3 \).

Assume for the sake of contradiction that two parallel links \( L_x \) and \( L_y \) exist and that the BG-property is violated. Then, \( S^R_l \neq S_3 \), and it contains at least four real nodes.

Both links \( L_x \) and \( L_y \) must contain a back edge as restriction (R2) forbids parallel links that only contain tree edges. Let \( C_i \) be the chain in \( S^R_l \) that contains \( L_x \), and let \( C_j \) be the chain that contains \( L_y \). Because \( C_i \) and \( C_j \) are already in \( S^R_l \), their endnodes are real. Since they contain only one back edge, their first edge, \( s(C_i) = s(L_y) = s(L_x) = s(C_j) \).

The inner nodes of \( L_x \) and \( L_y \) must be in different subtrees of \( T \) because they are contained in different chains, but end at the same node. Let \( z \) be that node. Note that \( z \in S^R_l \) because it is contained in two links.

Since the endpoints of \( P \) are inner nodes of \( L_x \) and \( L_y \), \( P \) must connect the two subtrees. Because \( T \) is a DFS tree, such a cross link is only possible via an ancestor \( v \) of \( z \). But \( v \) must be in \( S^R_l \) because of upwards-closedness. This contradicts the assumption that \( P \cap S^R_l = \{ x, y \} \).

Now we proceed to use this knowledge to show under which conditions single chains can be added. The existence of real nodes in the vicinity of the chain will be crucial. Type 3a chains can always be added if their parent is in the subdivision and thus has real endnodes. For the other types, we must assume the existence of real nodes.

**Lemma 2.11** ([14, Lemma 73]): Let \( C_i \) be a chain with parent \( C_k \in S^R_l \). It can be added to the subdivision \( S^R_l \) if one of the following conditions hold:

1. \( C_i \) is of type one, and there is a real inner node on \( t(C_i) \rightarrow t^+ \) s\((C_i) \). (Figure 2.8(a)).
2. \( C_i \) is of type 2a, and \( t(C_i) \) is real, or there is a real inner node on \( t(C_i) \rightarrow t^+_{C_k} \) s\((C_i) \). (Figure 2.8(b) and 2.8(c)).
3. \( C_i \) is of type 3a. (Figure 2.8(d)).

*Proof.* Because \( S^R_l \) is upwards-closed and modular, and \( C_i \) is just single chain, BG-condition 1 holds in all cases, and restriction (R1) is always preserved. We need to
2.10. WHEN CATERPILLARS NEED TO BE DECOMPOSED

show that BG-condition 2 holds to apply lemma 2.10 and that adding \( C_i \) preserves restriction (R2).

Let case 1 hold. The real inner node on \( s(C_i) \rightarrow t(C_i) \) prevents any link that contains both \( s(C_i) \) and \( t(C_i) \) to have them as inner nodes. The link can’t run along the tree path because of the real node, and it can’t skip over the real node by using a back edge because the starting point of a back edge in \( S^R \) must always be real as it starts a chain that has already been added. Therefore, \( C_i \) has BG-property 2, and we can invoke lemma 2.10. Due to the real inner node, we don’t add a parallel link consisting only of tree edges, so (R2) is preserved.

Let case 2 hold. Suppose \( t(C_i) \) is real. As \( C_i \) is a back edge with real endpoints, every link containing both \( s(C_i) \) and \( t(C_i) \) must contain them as endnodes. If on the other hand there is a real inner node on \( t(C_i) \rightarrow s(C_i) \), \( t(C_i) \) must be an inner node of a link that doesn’t contain \( s(C_i) \) as it must end at either \( t(C_k) \), which is real, or the real node on \( t(C_i) \rightarrow s(C_i) \). In both cases, we can invoke lemma 2.10. Since \( t(C_i) \rightarrow s(C_i) \) must contain \( t(C_k) \), restriction (R2) is preserved.

Lastly, let case 3 hold. By the definition of type three, \( s(C_i) \neq s(C_k) \), and \( s(C_i) \) is an inner node of \( t(C_k) \rightarrow s(C_k) \). As the endpoints of \( C_i \) are inner nodes of different chains, each chain that contains both must be a back edge. Hence we can again invoke lemma 2.10. As was the case with chains of type 2a, \( t(C_k) \) is an inner node of \( t(C_i) \rightarrow s(C_i) \), thus restriction (R2) is preserved.

\[ \square \]

2.10 When Caterpillars Need To Be Decomposed

Now that we know how to handle individual chains, we need to examine how to deal with caterpillars.
In the following, we will assume that for a caterpillar $L_i$, the parent $C_k$ is already added to the subdivision, but no chain in $L_i$ has yet been added. We begin by showing that a bad caterpillar can not be decomposed to BG-paths because of the restrictions we impose on the subdivision.

**Lemma 2.12 ([14, lemma 89]):** Let $L_i$ be a caterpillar with parent $C_k \in S^R_l$. If $L_i$ is bad, it can not be added.

**Proof.** If $L_i$ is bad, we have by definition $s(C_i) \in C_k$ and no real inner node on $s(C_i) \rightarrow_{C_i}^* s(C_k)$. Because $C_k \in S^R_l$, $s(C_k)$ is real. Let $y$ be the first real node on $s(C_i) \rightarrow_{C_i}^* t(C_k)$, and let $P = s(C_k) \rightarrow_{C_k}^* y$. Suppose we could add $L_i$, and let $Q_1, Q_2$ be the first two BG-paths that we generate (there must be at least two due to restriction ($R_1$)). $Q_1$ must connect the endnodes of $P$; otherwise BG-property 2 is violated. Since $s(C_i)$ is on $P$ and $L_i$ contains but one incident edge $e$ of $s(C_i)$, $e$ must be in $Q_1$. Then, $Q_2$ can’t contain both end nodes of $P$ and hence has at most one real node because $Q_1$ didn’t add real nodes. As $P$ and $Q_1$ are parallel links, one of the BG-properties must be violated. This implies that $L_i$ can’t be decomposed into at least two paths, therefore ($R_1$) is violated.

Next we show by the means of the function `decompose_to_bg_path`, how good caterpillars can be added. Because the function is called whenever we add a cluster, it also handles the cases when its argument is not a chain of type 3b.

We start by determining which type of good caterpillar we’re dealing with. Good caterpillars $L_i$ with parent $C_k$ are categorized into one of two subtypes: Those where $C_i$ is an inner node of $t(C_k) \rightarrow_{C_k}^* s(C_k)$ are assigned to type one (Figure 2.6(a)), those that have a real inner node on $s(C_i) \rightarrow_{C_i}^* s(C_k)$ get assigned type two (Figure 2.6(b)).

Whether a caterpillar is of type one, can easily be decided by checking whether $s(C_k) < s(C_i)$ and $s(C_i) < t(C_k)$. Because we make sure never to call the function with clusters that can’t be added, a caterpillar must be of type two if it fails this test.

**Listing 2.9: Decomposing a caterpillar to BG-paths**

```cpp
void schmidt_triconnectivity::decompose_to_bg_paths(const chain* a_chain) throw()
{
    switch(a_chain->get_type()) {
        case two_b: break;
        case three_b: {
            caterpillar& p = caterpillars[a_chain->number];
            enum caterpillar_type {
                type1, type2, bad
            } type = bad;
        }
    }
```
2.10. WHEN CATERPILLARS NEED TO BE DECOMPOSED

if (dfi(p.parent->get_s()) < dfi(a_chain->get_s()) &&
    dfi(a_chain->get_s()) < dfi(p.parent->get_t()))

    type = type1;
} else {
    type = type2;
}

We now want to construct the green paths of figure 2.6. To do so, we compute three lists of edges:

- ci, the edges in the chain $C_i$
- tci_y, the edges on the tree path from $t(C_i)$ to the endnode $y$ of the minimal chain in the caterpillar.
- d0, the edges of $C_i$'s the parent up to the node $t(C_i)$.

We will have to determine quickly whether an edge is in the second list to add the remaining part of the caterpillar, so we create a hashmap on_tci_y and insert those edges.

Now the way we oriented the edges during the DFS in listing 2.2 lines 29ff becomes useful because it makes iterating over the paths easier.
Next, we perform a case distinction over the type of the caterpillar and concatenate
the appropriate lists to form the green paths of figure 2.6. For type one caterpillars
ci and tci_y produce the path; otherwise d0 and ci need to be concatenated. In
both cases the remaining path is a BG-path too.
Afterwards, C_i and its parent are completely added to the certificate, and y is a real
node.

```cpp
switch(type) {
    case type1:
        ci.conc(tci_y, leda::behind);
        cert->add_bg_path(ci);
        cert->add_bg_path(d0);
        break;
    case type2:
        ci.reverse();
        d0.conc(ci, leda::behind);
        cert->add_bg_path(d0);
        cert->add_bg_path(tci_y);
        break;
    default: assert(false);
}
```

We can now proceed to add the remaining edges of the other type 2b chains. For
each chain, we generate a list of edges from s(C_k) along the chain until we hit an
edge on tci_y.

```cpp
p.pop(); //remove parent, already added
chain* c;
forall(c,p) {
    list<edge> di;
    for(edge e = c->first_edge; !on_tci_y[e]; e = parent[target(e)]) {
        di.append(e);
    }
    cert->add_bg_path(di);
}
```

**Lemma 2.13 ([14, Lemma 76]):** The function decompose_to_bg_paths is correct.

**Proof.** For chains that are not contained in caterpillars, the correctness follows from
lemma 2.11.
Otherwise, let L_i be a good caterpillar with parent C_k, let y be the endnode of the
minimal chain in L_i, let D be the parent of C_i, and let t > 1 be the number of chains
in L_i.
2.10. WHEN CATERPILLARS NEED TO BE DECOMPOSED

Clearly, if \( L_i \) is added completely, upwards-closedness and modularity are preserved as \( L_i \) is composed of successive ancestors in \( U \). We need to show that the decomposition we compute indeed consists of BG-paths and preserves the restrictions.

We do a case distinction on the type of the caterpillar.

Let \( L_i \) be of type one, i.e., let \( s(C_i) \) be an inner node of \( t(C_k) \to_T^* s(C_k) \) as in figure 2.6(a). In that case we add the following paths:

1. \( s(C_i) \to_{C_i}^* t(C_i) \to_{T}^* y \)
2. \( s(D) \to_D^* t(C_i) \)

The first path obviously fulfills BG-property 1 as only \( s(C_i) \) and \( y \) are in \( S_i^R \). Because the endpoints of \( C_k \) are real, property 2 is also fulfilled. Consequently, the path is a BG-path for \( S_i^R \) with lemma 2.10. The path \( s(D) \to_T^* t(C_i) \) is also a valid BG-path. It has BG-property 1 and 2 because \( s(D) = s(C_k) \) is real, and \( y \) has just become real. The real node \( t(C_k) \) prevents parallel links between \( s(D) \) and \( y \), and thus 3 is also fulfilled.

Otherwise, assume that \( L_i \) is of type two, that is, \( s(C_i) \) is not an inner node of \( t(C_k) \to_{C_k}^* s(C_k) \), but instead lies on \( C_k \), and there is a real inner node on \( s(C_i) \to_{C_k}^* s(C_k) \) as depicted in figure 2.6(b). In this case, we add the following paths:

1. \( s(D) \to_D^* t(C_i) \to_{C_i}^* s(C_i) \)
2. \( t(C_i) \to_T^* y \)

We show that there must be another real node \( b \) on \( t(C_k) \to_T^* s(C_k) \) due to restriction (R2). Assume the contrary for the sake of contradiction. Because \( S_i^R \) is upwards-closed and \( t(C_k) \in S_i^R \), the parent of \( C_k \) must also be in \( S_i^R \). If there is no real inner node on \( t(C_k) \to_T^* S_i^R \), it follows that \( C_k \) is of type one, as otherwise \( t(parent(C_k)) \) would be a real node there. As the endpoints of \( C_k \) are real, \( t(C_k) \to_T^* s(C_k) \) is a link in \( S_i^R \) that consists solely of tree edges. When \( C_k \) was added, it was a BG-path, and hence no real node existed on \( C_k \). But then adding \( C_k \) would have violated restriction (R2) because it created a parallel link. Therefore, there must be a real node \( b \) on \( t(C_k) \to_T^* s(C_k) \).

Because \( a \) and \( b \) are real nodes, the path \( s(D) \to_D^* t(C_i) \to_{C_i}^* s(C_k) \) is a BG-path. It has only two attachment nodes with \( S_i^R \), and \( a \) and \( b \) ensure BG-property 2. Hence lemma 2.10 can be applied.

For the same reasons, \( t(C_i) \to_T^* y \) is also a BG-path. Note that after adding it, upwards-closedness is restored.

Independent of which caterpillar type we’re dealing with, we proceed to add the paths \( s(C_i) \to_{C_i}^* x \), where \( x \) is the first node that lies on \( t(C_i) \to_T^* y \). They
are BG-paths for the same reasons as \( s(D) \xrightarrow{D} t(C_i) \) in the case of a type one caterpillar.

Hence, \( L_i \) can be decomposed into \( t \) BG-paths, and restriction (R1) is preserved. It remains to show that (R2) also holds.

Assume for the sake of contradiction that there are two parallel links \( Q_1, Q_2 \) in \( S_{L+t}^R \), and let \( Q_1 \) consist solely of tree edges. We distinguish two cases.

First, assume that \( Q_1 \) was created during the decomposition of \( L_i \), i.e. \( Q_1 \) is not a link in \( S_{L+t}^R \). Then, \( Q_1 \) is contained in \( L_i \), therefore the parallel link \( Q_2 \) must also be contained in \( L_i \). But as we didn’t add parallel links in the above construction, this is a contradiction. Now suppose \( Q_1 \) is not contained in \( L_i \). In this case, it must be contained in a link of \( S_{L+t}^R \) that was subdivided by at least one real node that was created during the decomposition of \( L_i \). However, we only created the real nodes \( s(C_i) \) and \( y \) in \( S_{L+t}^R \). But as \( t(C_i) \) is already real in \( S_{L+t}^R \), no parallel link \( Q_2 \) can exist.

On the other hand, let \( Q_1 \) be a link in \( S_{L+t}^R \) that is a link in \( S_{L+t}^R \) as well. Then, \( S_{L+t}^R \neq S_{3} \), as \( C_0 \) is the only link in \( S_3 \) that doesn’t contain back edges, and adding \( L_i \) would have created a real inner node on \( C_0 \). As (R2) holds in \( S_{L+t}^R \), \( Q_1 \) does not have a parallel link, and \( Q_2 \) must have been created during the decomposition of \( L_i \). But then \( Q_2 \) can’t be parallel to \( Q_1 \) because every path between two of the nodes \( \{s(C_i), y, s(C_k)\} \) in \( L_i \) must contain a real node in \( S_{L+t}^R \).

Hence (R2) is preserved in \( S_{L+t}^R \).  

\[\square\]

### 2.11 Constructing the BG-Decomposition

After the theoretical work of the previous sections we can at last understand the workings of the algorithm’s main part. We iterate over the chains in the order in which they were created. For each chain \( C_i \), we perform four operations:

1. Add all children \( C_j \) of type 2a that have a real \( t(C_j) \)
2. Partition the chains from \( \text{type3}(C_i) \cup \text{children12}(C_i) \) into segments
3. Add the "easy" segments
4. Add the "hard" segments

The meaning of these steps will become clear in the next sections.

It is important to note that \( C_i \) is already part of the subdivision when we reach it in the iteration. In particular this means that its endpoints are already real. If \( G \) is 3-connected, we will add all chains that have one of their endnodes on \( C_i \), i.e. its children of type one and two as well as the chains in \( \text{type3}(C_i) \). Hence \( \text{type3}(C_j) = \emptyset \) for all proper ancestors of \( C_i \). It will become clear during the presentation that we always terminate should one of these conditions not be fulfilled. For future reference, we summarize them in the following lemma:
2.11. CONSTRUCTING THE BG-DECOMPOSITION

Lemma 2.14: When $C_i$ is processed the following holds:

1. $C_i$ is already added to the subdivision
2. $\text{type}_3(C_i) = \emptyset$ for all proper ancestors $C_j$ of $C_i$

We begin by iterating over $\text{children}_{12}(C_i)$. We add all children of type 2a that have a real endnode and save the remaining children in a list. It is correct to add these chains because of lemma 2.11.

Listing 2.10: The actual algorithm

```cpp
auto_ptr<certificate> schmidt_triconnectivity::certify(void) {
for(unsigned int i=0;
   i<(unsigned int)chains.size() &&
   chains_in_subdivision<(unsigned int)chains.size();
   i++)
{
    chain* current_chain = chains[i];
    slist<chain*> children12;
    int_set set_children12(chains.size());
    foreach(child, current_chain->children12) {
      if (child->get_type()==two_a && is_real[child->get_t()]) {
        add_with_ancestors(child);
        continue;
      }
      children12.append(child);
      set_children12.insert(child->number);
    }
    h_array<unsigned int, slist<node> > attachment_nodes;
    h_array<unsigned int, slist<chain*> > segment_chains;
    h_array<chain*, unsigned int> segment;
    const list<chain*>& type3 = current_chain->type3;
    ...
```
2.11.1 Partitioning into segments

A segment of a subdivision $S_l$ is defined in [14] as the edge induced subgraph of an equivalence class of the following equivalence relation:

**Definition 2.4.** Let $\sim$ be the equivalence relation on $E(G)\setminus E(S_l)$ such that two edges $e, f$ in $E(G)\setminus E(S_l)$ are equivalent $e \sim f$ if $e = f$, or if there is a path in $G$ that contains $e$ and $f$ but no node of $S_l$ as inner node.

Let the **attachment nodes** of a segment $H$ be the nodes in $V(H) \cap V(S_l)$.

We found it useful to think of segments as the connected components that remain after deleting $S_l$ from the graph. This is however just an intuition and not completely correct, as the attachment nodes of a segment lie in $S_l$. For example, a single back edge forms a segment by itself.

We’re dealing with subdivisions $S_l^R$ that are upwards-closed and modular. In this case, a segment is always the union of a subtree of $U$. From this follows directly:

**Lemma 2.15 ([14, Lemma 80]):** Every segment $H$ of $S_l^R$ contains exactly one chain that is a child of a chain in $S_l^R$, namely, the minimal chain in $H$.

To proceed with the algorithm, we want to partition the chains in $children12(C_i) \cup type3(C_i)$ according to the segments to which they belong. In this section, we deal with the segments that contain a chain from $type3(C_i)$. We also check whether a segment is *hard* or *easy*. Easy segments do not contain a chain from $children12(C_i)$.

We identify a segment with the minimal chain that attaches it to $C_i$ and construct two $h$-arrays. One maps from the minimal chain to a list of chains in its segment, the other stores for each chain the segment to which it belongs.
We can find the minimal chain to which a chain $C_j$ in $type3(C_i)$ belongs by traversing the path $C_j \rightarrow \tau^*_i C_0$ until we reach a chain in $S^R$. To get a linear runtime, we mark all chains on that path with the minimal chain and stop in subsequent searches once a marked chain is encountered.

This suggests two passes over the path. We begin by finding the minimal chain:

```
void schmidt_triconnectivity::partition_into_segments(
    const chain* current_chain,
    const list<chain*>& type3,
    h_array<unsigned int>,
    slist<node> & attachment_nodes,
    h_array<unsigned int, slist<chain*> > & segment_chains,
    h_array<chain*, unsigned int> & segment,
    const int_set& children12) throw()
{
    h_array<chain*, unsigned int> minimal_chain;
    chain* t3_chain;
    forall(t3_chain, type3) {
        chain* c = t3_chain;
        while (!c->get_parent()->is_in_subdivision() && !minimal_chain.defined(c)) {
            c = c->get_parent();
        }

        const bool new_segment = c->get_parent()->is_in_subdivision();
        if (new_segment) {
            minimal_chain[c] = c->number;
        }
    }

    const unsigned int m_chain = minimal_chain[c];
    c = t3_chain;
    while (!c->get_parent()->is_in_subdivision() && !minimal_chain.defined(c)) {
        minimal_chain[c] = m_chain;
        c = c->get_parent();
    }

    segment[t3_chain] = m_chain;
    segment_chains[m_chain].append(t3_chain);
}
```

In a second pass over the path, we mark them all with the minimal chain. Then we add the type three chain to the segment.
Afterwards, we check whether the segment is hard, i.e. whether the minimal chain is in $children12(C_i)$. If that’s the case, we also update the attachment nodes of the segment. The attachment nodes are the starting points of type three chains in the segment and the start- and endpoints of the minimal chain.

```cpp
const bool hard = children12.member(m_chain);
if (hard) {
    if (new_segment) {
        const chain* min_chain = chains[m_chain];
        attachment_nodes[m_chain].append(min_chain->get_s());
        attachment_nodes[m_chain].append(min_chain->get_t());
    }
    attachment_nodes[m_chain].append(t3_chain->get_s());
}
}
```

Note that Schmidt [14] describes a slightly different approach to finding the minimal chain. He suggests that one should traverse the tree path $t(C_j) \rightarrow r$ until a node is encountered that is already part of the subdivision. It is easy to see that his method is basically equivalent, but slightly slower, to traversing the chain tree $U$.

### 2.11.2 Adding the easy segments

When we say we "add a segment $H$", we mean adding all ancestors of chains in $(type3(C_i) \cup children12(C_i)) \cap H$. Adding easy segments is indeed not difficult. We simply add each chain in $type3(C_i)$ that is in an easy segment with all its ancestors.

```cpp
void schmidt_triconnectivity::add_easy_segments(
    const list<chain*>& type3,
    h_array<chain*, unsigned int> const & segment,
    const int_set& children12) throw()
{
    chain* t3_chain;
    forall(t3_chain, type3) {
        if (children12.member(segment[t3_chain])) continue;
        add_with_ancestors(t3_chain);
    }
}
```

It is considerably harder to show correctness. Fortunately, we will be able to reuse some of the results for the case of hard segments. First, we examine under which
conditions the minimal type three chain of a segment can be added with all its ancestors. The next lemma then shows how the easyness of a segment implies the preconditions for adding its minimal chain. Lastly, we show how the argument can be iterated to add all type three chains and their ancestors.

**Lemma 2.16** ([14, Lemma 81]): Let $D_0$ be a chain of type three such that $s(D_0)$ is in the subdivision $S^R$ but $D_0$ is not. Let $D_0$ be minimal among the type three chains of its segment $H$.

Let $D_k < \ldots < D_0$ be the ancestors of $D_0$ that are contained in $H$. Then, the clusters of $D_k, \ldots, D_0$ can be added to the subdivision, unless one of the following exceptions hold:

1. $D_0$ is of type 3a, $k = 1$, $D_1$ is of type one, $s(D_0)$ is an inner node of $t(D_1) \to_T s(D_1)$, and there is no real inner node in $t(D_1) \to_T s(D_1)$.

2. $D_0$ is of type 3b, and \{ $D_0, \ldots, D_k$ \} is a bad caterpillar.

3. $D_0$ is of type 3b, \{ $D_0, \ldots, D_{k-1}$ \} is a caterpillar with parent $D_k$ of type one, $s(D_0)$ is an inner node of $t(D_k) \to_T s(D_k)$, and there is no real inner node in $t(D_k) \to_T s(D_k)$.

An example graph of each exception is show in figure 2.9.

![Figure 2.9: The exceptions of lemma 2.16](image)

**Proof.** We do a case distinction and reduce the number of cases through the following observations:
Due to the choice of $D_0$ as the minimal type three chain in $H$, there can be no type three chain in $\{D_1, \ldots, D_k\}$

$D_k$ is the minimal chain in $H$ because segments form subtrees of $U$, and $D_1, \ldots, D_k$ is a maximal path of chains that are in $H$.

$\{D_1, \ldots, D_k\}$ doesn’t contain a chain of type 2a, as those chains are back edges and don’t have descendants.

There are seven cases:

1. Let $D_0$ be of type 3a.
   (a) Let $k = 0$. In this case, $t(D_0)$ is in $S_l$, and $D_0$ can be added according to lemma 2.11
   (b) Let $k > 0$.
      - Let $D_1$ be of type 2b, and let $C_j$ be the chain of type 3b such that $L_j$ contains $D_1$. $C_j$ is contained in $H$, as there obviously is a path to $D_1$, and $C_j$ can’t be in $S^R_l$ because $D_1$ is not in $S^R_l$, and we only add caterpillars as a whole because of restriction ($R_1$).
        Further, $C_j$ must be an ancestor of $D_0$ because otherwise $D_1$ would still be marked during the chain decomposition, and $D_0$ would become a chain of type 3b. This contradicts the minimality of $D_0$.
      - Let $D_1$ be of type one. Then, $s(D_1)$ is a proper ancestor of $s(D_0)$, and $s(D_0)$ is a proper ancestor of $t(D_1)$. Therefore, $s(D_0)$ is an inner node of $t(D_1) \rightarrow^*_s s(D_1)$.
        - Let $k \geq 2$. Because $D_1$ is a type one child of $D_2$, $D_2$ must contain $t(D_1) \rightarrow^* s(D_1)$. Hence the edge $e = (s(D_0), \text{parent}(s(D_0)))$ is in $D_2$. Since $S^R_l$ is upwards-closed and $s(D_0) \in S^R_l$, so must be the parent of $s(D_0)$, contradicting $k \geq 2$ since $D_2 \notin H$.
        - Let $k = 1$. It follows that the parent of $D_1$ is in $S^R_l$. If the path $t(D_1) \rightarrow^*_s s(D_1)$ contains a real inner node, it can be added with lemma 2.11. Otherwise, there is no real node on the path, and exception 1 holds.

2. Let $D_0$ be of type 3b, and let $L$ be the caterpillar that it starts. Because we only add caterpillars as a whole, all chains in $L$ must be in $H$. As $D_1$ is the parent of $D_0$, $D_1$ must be of type 2b and contained in $L$. Let $D_t$, for $1 \leq t \leq k$, be the minimal chain in $L$.
   (a) Let $t = k$. If $L$ is good, it can be added with lemma 2.9. Otherwise, exception 2 holds.
   (b) Let $t < k$.
      - Let $D_{t+1}$ be of type 2b. Then, the caterpillar that contains it must have been created before $L$, contradicting the minimality of $D_0$.
      - Let $D_{t+1}$ be of type one. As it is an ancestor of $D_0$, and $D_0$ is of type 3b, $s(D_{t+1})$ is a proper ancestor of $s(D_0)$. Further, $D_{t+1}$ can’t be a back edge, because it has descendants.
As $D_{t+1}$ must have property $B$, there must be a type three chain starting at an inner node $v$ of $t(D_{t+1}) \to_T s(D_{t+1})$. Because we chose $D_0$ to be minimal, $s(D_0)$ must be an ancestor of $v$ and thus lies on $t(D_{t+1}) \to_T s(D_{t+1})$ too.

Then, the parent $D$ of $D_{t+1}$ must be in $S^R$, because $D$ contains the tree path $t(D_{t+1}) \to_T s(D_{t+1})$ and $s(D_0)$ is in $S^R$. Therefore, $k = t + 1$. If there is a real inner node on $t(D_{t+1}) \to_T s(D_{t+1})$, $D_{t+1}$ can be added with lemma 2.11, and $L$ becomes good due to $t(D_{t+1})$. If there is no such node, exception 3 holds.

Now the preconditions for adding a type three chain and its ancestors in a segment are clear. Next, we examine why they are satisfied if the chain lies in an easy segment.

**Lemma 2.17** ([14, Lemma 84]): Let $C_i$ be the chain that is currently being processed, and let $C_y$ be the first chain in $type3(C_i)$ that is contained in an easy segment $H$. Then, it is possible to add $C_y$ and all its ancestors that are not yet added.

**Proof.** We want to apply lemma 2.16, so we need to show two things.

1. $C_y$ is the minimal type three chain in $H$.
2. None of the exceptions of lemma 2.16 hold.

Let $D_0$ be the minimal type three chain in $H$. We show that it is equal to $C_y$.

According to lemma 2.3, $s(D_0)$ is an ancestor of $s(C_y)$. However, $s(D_0)$ can not be contained in a proper ancestor of $C_i$ because of lemma 2.14. Hence, $D_0$ is in $B$. But as we choose $C_y$ to be the first chain in $type3$ that is in $B$, and that list is ordered, it follows that $D_0 = C_y$.

We proceed to show that $C_y$ is not contained in one of the exceptions of lemma 2.16. First, we show that $C_y$ is not contained in exceptions 1 or 3. Assume the opposite. Then, the parent of $C_y$’s cluster, $D_k$, is of type one. We show that $D_k$ is a child of $C_i$, contradicting the assumption that $H$ is an easy segment.

Both exceptions require $s(C_y)$ to be an inner node of $t(D_k) \to_T s(D_k)$. As $C_y$ is in $type3(C_i)$, we have $s(C_y) \in C_i$. It follows with $D_k$ having type one that it must be in $children12(C_i)$ and $H$ is a hard segment.

Next, we show by contradiction that $C_y$ is not contained in exception 2, i.e. $L_y = \{C_y, \ldots, D_k\}$ is not a bad caterpillar. We show that the parent of $L_y$, $D$, is equal to $C_i$ making $D_k$ a chain in $children12(C_i)$. This contradicts the assumption that $H$ is an easy segment.

Assume then, that $D$ is different from $C_i$. By the definition of bad caterpillars, we have $s(C_y) \in D$. As $C_y$ is of type three and a descendant of $D$, we know that
\( s(D) \neq s(C_y) \). Since \( C_y \in \text{type3}(C_i) \), it follows that \( s(C_y) \) is an inner node of \( C_i \). Summarizing we conclude

\[ s(C_y) \in C_i \cap (D \setminus s(D)). \quad (2.11.1) \]

\( D \) can only contain \( s(C_y) \) as an endnode because it is already an inner node of \( C_i \) and by assumption \( D \neq C_i \). In particular, \( s(C_y) = t(D) \) due to (2.11.1). This implies with the definition of bad caterpillars that \( t(D) \rightarrow s(D) \) does not contain a real node. But as \( D \) must have property B, there must be a chain \( C_z \) of type three that enters the subtree rooted at the last inner node of \( D \). \( C_z \) can’t be in the subdivision because \( D \) does not contain real inner nodes. If \( C_z \) starts in \( C_i \), we have a contradiction to the choice of \( C_y \), since \( s(C_y) = t(D) \) and \( s(C_z) \) is an ancestor of \( t(D) \) (and therefore \( C_z < C_y \)). Otherwise, if \( C_z \) does not start in \( C_i \), it must start in a proper ancestor of \( C_i \) contradicting lemma 2.14.

Hence, \( D = C_i \), and \( D_k \in \text{children12}(C_i) \) yielding a contradiction to the assumption that \( H \) is an easy segment.

Adding a cluster to the subdivision changes the segments. This makes it impossible to use lemma 2.17 again without recalculating them. Obviously, we don’t want to do the calculations again and again if we are to achieve a linear runtime. Fortunately, that is unnecessary. As the next lemma shows, we can add all clusters in a segment if we can add the minimal type three chain. To make the lemma applicable for the case of hard segments too, we make it more general than necessary here by removing the easyness requirement.

**Lemma 2.18** ([14, Lemma 82]): Let \( Y \) be the clusters of ancestors of all chains of type three from a segment \( H \) for a subdivision \( S_R \). If it is possible to add the minimal type three chain \( D_0 \) of \( H \) using lemma 2.16, then it is also possible to add all clusters in \( Y \cap \text{type3}(C_i) \) in ascending order.

**Proof.** Let \( U_Y \) be the subtree of \( U \) that has elements of \( Y \) as nodes. Suppose we already added \( k \) chains resulting in a new subdivision \( S_{i+k}^R \). Let \( J \) be the next cluster to add, and let \( J_0 \) be the minimal chain in the cluster. \( J_0 \) must be the root of a subtree \( U' \) of \( U_Y \) and minimal in its segment \( H' \) because we add clusters in ascending order.

Let \( J_0 \) be the minimal type three chain in \( U' \) that starts in \( S_i^R \) (i.e. the first chain in \( \text{type3}\cap U' \)). Then, \( J_0 \) must also be the minimal type three chain of \( H' \).

Let \( J_s < \ldots < J_0 \) be the ancestors of \( J_0 \) in \( U' \). We want to apply lemma 2.16 to \( J_0 \). As we already know that \( J_0 \) is minimal, we need to show that none of the exceptions hold.

First assume that exception 1 or 3 hold. In this case, \( J_s \) is of type one as \( s(J_0) \) is an inner node of \( t(J_s) \rightarrow t(J_0) \). As \( s(J_0) \in S_i^R \) and \( S_i^R \) is upwards-closed, we get \( s(J_s) \in S_i^R \) because \( s(J_s) \) is an ancestor of \( s(J_0) \).
Let $D_k < \ldots < D_0$ as before be the ancestors of $D_0$ that are not in $S^R_i$. Again $D_k$ is the minimal chain in $H$. According to lemma 2.15, $D_k$ is the only chain in $H$ that is a child of a chain in $S^R_i$ and hence the only chain whose endpoint is in $S^R_i$. Because one of the exceptions holds, $J_s$ is of type one, and hence $s(J_s) \geq t(D_k)$. We also know $s(J_s) \in S^R_i$. Since no chain that is not in $S^R_i$ contains an inner node in $S^R_i$, it must be that $s(J_s) = t(D_k)$. For the same reason and because $s(J_0) \geq s(J_s)$ also $s(J_0) = t(D_k)$. It follows that $s(J_0)$ is not an inner node of $t(J_s) \rightarrow_+ s(J_s)$.

Then, assume that exception 2 holds. In this case $J_0$ is of type 3b and starts a bad caterpillar $L = \{J_0, \ldots, J_k\}$ with parent $D$. $D$ is not in $H'$, hence it must be in $S^R_i$. $D$ must be a descendant of $D_k$ because of the minimality of $D_k$. We show $D = D_k$ and use property B on $I_k$ to derive the existence of a real inner node, contradicting the assumption that $L$ is bad.

As $L$ is a bad caterpillar in $S^R_{i+k'}$, $s(J_0) \in D \setminus s(D)$ as before. Moreover, $s(J_0) \in S^R_i$, and no chain contains a node in the subdivision as inner node, $s(J_0) = t(D)$. Because $D_k$ is the only chain in $H$ that has an endnode in $S^R_i$, $D = D_k$.

We proceed to show that $L$ can’t be bad. The chain $D_k$ must have property B and is not a back edge. Thus, there is a type three chain $C_z$ that enters the subtree of $D_k$’s last inner node. $C_z$ must start on $t(D_k) \rightarrow_+ s(D_k)$ and must be in $H$ but not in $H'$. If it were in $H'$, $J_0$ would not be the minimal type three chain. But if $C_z$ is in $H \setminus H'$, it must be in $S^R_{i+k'}$ and thus there must be a real inner node on $D_k$ due to upwards-closedness. This contradicts the assumption that $L$ is bad.

Schmidt [14] shows how lemma 2.18 can be trivially extended to work with any preorder, not just $<$. 

### 2.11.3 Adding the hard segments

Adding the hard segments is much more complicated than adding the easy segments. Fortunately, proving the correctness of the procedure will be simpler because of the large amounts of preparatory work.

We want to apply lemma 2.18, but we cannot rely on $H \cap \text{children}_{12}(C_i) = \emptyset$ anymore to guarantee the applicability of lemma 2.16.

The idea is to add the minimal chain $C_i$ of a hard segment $H$ by applying lemma 2.11. This creates a new subdivision in which $H$ can be added. The application of lemma 2.11 requires a real node on the dependent path of the cluster.

**Definition 2.5.** Let the dependent path of a cluster $C_i$ with parent $C_k$ be

- the path $t(C_i) \rightarrow_{C_k} s(C_i)$ if $C_i$ is of type one or of type 2a.
- the path $s(C_i) \rightarrow_{C_k} s(C_k)$ if $C_i$ is a caterpillar.
- empty otherwise.
Note that for a type one chain \( C_i \) with parent \( C_k \), the paths \( t(C_i) \rightarrow_{C_i} s(C_i) \) and \( t(C_i) \rightarrow_{T} s(C_i) \) are identical. We define the dependent path of a segment to be the dependent path of its minimal chain.

We begin by showing how the attachment nodes of a segment relate to the dependent path of its parent. Next, we prove the main lemma of this section that shows under which conditions hard segments can be added. Adding \( H \) makes the attachment nodes we computed in listing 2.11, lines 39ff, real. The new real nodes hopefully enable us to add another chain of \( \text{children12}(C_i) \) and its segment, until all chains in \( \text{children12}(C_i) \cup \text{type3}(C_i) \) have been added, and the processing of \( C_i \) is finished.

**Lemma 2.19** ([14, Lemma 86]): Let \( H \) be a segment of \( S_R^t \), and let \( D \) be the minimal chain of \( H \). Let \( C_k \) be the parent of \( D \). If \( D \) is neither of type 3a nor contained in a caterpillar such that \( s(C_i) \notin C_k \), the dependent path \( P \) of \( D \) contains all attachment nodes of \( H \).

**Proof.** We do a case distinction on the type of \( D \).

Let \( D \) be of type one. Then by definition \( P = t(D) \rightarrow_{T} s(D) \). If \( D \) is a back edge, the claim follows directly. Otherwise, let \( x \) be its last inner node. In this case, any chain that enters the subtree \( T(x) \) must start on \( P \) due to lemma 2.3 and the DFS structure of the tree. Hence, all attachment nodes lie on \( P \).

Next, assume that \( D \) is a back edge of type 2a, and let \( C_k \) be its parent. It follows that the dependent path is \( P = t(D) \rightarrow_{C_k} s(D) \), and the only attachment nodes are \( s(D) \) and \( t(D) \). Again, these must be contained in \( P \).

\( D \) can not be of type 3b, as otherwise it and its parent \( C_k \) must be contained in a caterpillar. Restrictions (R1) forbids that \( C_k \) is in \( S_R^t \) but \( D \) is not.

Then, let \( D \) be of type 2b. It must be the minimal chain in a caterpillar \( L_i \). Let \( x \) be the last inner node of \( D \). By the construction of caterpillars, \( C_i \) is the minimal type three chain that enters \( T(x) \) as otherwise \( D \) is contained in a different caterpillar. It follows that all back edges that enter \( T(x) \) must either start at a descendant of \( s(C_i) \) or at a descendant of \( s(D) = s(C_k) \). Because \( s(C_i) \in C_k \) by assumption, all attachment nodes lie on the dependent path \( P = s(C_i) \rightarrow_{C_k} s(C_k) \).

**Lemma 2.20** ([14, Lemma 89]): Let \( S_R^t \) be the subdivision after adding the easy chains. Let \( D \) be a chain in \( \text{children12}(C_i) \), let \( H \) be the segment of \( S_R^t \) that contains it, and let \( P \) be its dependent path.

If \( P \) contains a real inner node, \( D \) and all ancestors of chains from \( \text{type3}(C_i) \cap H \) that are not in \( S_R^t \) can be added in ascending order of \( < \). Conversely, if \( P \) does not contain a real inner node, no cluster in \( H \) can be added.

**Proof.** First assume that \( P \) contains a real inner node. There are two cases: If \( H \cap \text{type3}(C_i) \neq \emptyset \), we can apply lemma 2.18, as the real node prevents the exceptions
from occurring. Otherwise, we just need to add $D$. This is possible due to the real node on the dependent path and lemma 2.11.

Now assume $P$ does not contain a real inner node. We need to show that no cluster of $H$ can be added. Assume for the sake of contradiction that we can add a cluster in $H$. Because we require upwards-closedness, $D$ must be the first chain we add.

According to lemma 2.19, all attachment nodes of $H$ lie on $P$. If we are to add any of the chains in $H$, the endnodes of $P$ must be real as otherwise BG-property 2 is violated. This is obvious for chains of type one, 2a, and caterpillars. Chains of type 3a can’t be added because their parent can not yet be in $S^R$. If the endnodes of $P$ are real, and $P$ does not contain real inner nodes, $P$ is a link in $S^R$ that consists solely of tree edges.

We do a case distinction on the type of $D$.

Let $D$ have type one. In this case, the endnodes of $P$ are also the endnodes of $D$. Therefore, adding $D$ would create a parallel link to $P$ and violate restriction (R2).

Let $D$ have type 2a. Since $D$ is a back edge, the endnodes of $P$ are also endnodes of $D$. But then $t(D)$ is real, and $D$ would already have been added in line 14 of listing 2.10.

Lastly, let $D$ be of type 2b. Then, it must be contained in a caterpillar $L_j$ with parent $C_i$. We show that $L_j$ is bad. If $C_j \in C_i$, it must be that $C_j \in \text{type3}(C_k)$ for a proper ancestor $C_k$ of $C_i$, contradicting our assumptions. Because $s(C_j)$ is an attachment node and thus must lie on $P$, it follows that the path $s(C_j) \rightarrow s(C_i)$ does not contain a real node. Bad caterpillars can’t be added due to lemma 2.12.

The hard part about adding the hard segments is thus finding an order on them that creates the necessary real nodes. We will call an order that adds segments such that these nodes are created proper.

The function add_hard_segments’ primary concern is reducing this problem to a problem on intervals. We map the attachment nodes of each segment and the real nodes on the union of the dependent paths to intervals. Due to lemma 2.19, the dependent path of each segment will also be mapped to an interval. We then want to find an order on these intervals, starting from one of the preexisting real nodes, such that an interval $i$ appears in the order only if one endpoint of a previous interval is contained in $i$. This is explained in more detail in the next section.

We begin by mapping the nodes of $C_i$ to integers from $[1, |C_i|]$, to create a bijection between nodes and numbers.

---

Listing 2.13: Adding the hard segments

```c++
void schmidt_triconnectivity::add_hard_segments(
    const chain* current_chain,
    h_array<unsigned int, slist<node>> const & attachment_nodes,
    h_array<unsigned int, slist<chain>> const & segment_chains,
    slist<chain>> & children12) throw(not_triconnected_exception)
```
CHAPTER 2. SCHMIDT’S ALGORITHM

{ node_array<unsigned int> mapping(the_graph,0);
  h_array<unsigned int, node> reverse_mapping;
  unsigned int nodes_in_chain = 0;
  {
    chain_node_iterator it(current_chain, this);
    for(node n=it.next(); n!=NULL; n=it.next()) {
      nodes_in_chain++;
      mapping[n] = nodes_in_chain;
      reverse_mapping[nodes_in_chain] = n;
    }
  }

Next, we create an interval for each chain in children12(C_i) that is not a minimal chain of a segment that we found while partitioning type3(C_i). Intervals for the other chains in children12(C_i) will be created shortly afterwards, and we want to avoid adding an interval for a chain twice.

Each of these chains must be its own segment and a back edge due to property B and the absence of chains from type3(C_i) in their segment. As such, they only have their endnodes as attachment vertices.

An interval saves its bounds and a pointer to the chain that created it. A detailed discussion of the implementation can be found in appendix A.3.

We also keep two variables min_n and max_n that store the extremal attachment nodes. The path between those is the union of the dependent paths from the segment. We will add intervals for all preexisting real nodes on that path.

unsigned int min_n = nodes_in_chain+1, max_n = 0;
slist<interval<chain* > * > intervals;
{ chain* c;
  forall(c,children12) {
    if (attachment_nodes.defined(c->number))
      continue;
    unsigned int m1,m2;
    m1 = mapping[c->get_s()];
    m2 = mapping[c->get_t()];
    min_n = std::min(min_n,std::min(m1,m2));
    max_n = std::max(max_n,std::max(m1,m2));
    intervals.append(new interval<chain*>(m1, m2,c));
  }
}

Now we add intervals for the remaining segments that contain more than two attachment nodes. Let a_1, . . . , a_k be the attachment nodes of the current segment in sorted order. We add an interval from a_1 to a_k, intervals [a_1, a_j], and intervals [a_j, a_k], for 1 < j < k. We save the information that all these intervals are created from the same segment by storing them as a list in equivalent_intervals.
First, we map all the attachment nodes to numbers and sort them using bucket sort (for implementation details see appendix B.1) while discarding duplicates.

Then we update the endpoints of the longest dependent path and store \(a_1\) in smallest and \(a_k\) in largest.

Afterwards, we begin adding intervals for the segment. First the interval \([a_1, a_k]\), then the remaining ones.
CHAPTER 2. SCHMIDT’S ALGORITHM

```
 CHAPTER 2. SCHMIDT’S ALGORITHM

it = mapped_nodes.succ(it); //skip smallest

for (unsigned int i=1;
 i <= (unsigned int)mapped_nodes.size()-2; //skip largest
 i++)
 {
   interval<chain*>* val;
   const unsigned int a_i = mapped_nodes.contents(it);
   val = new interval<chain*>(smallest, a_i, chains[m_chain]);
   intervals.append(val);
   equivalence_class->append(val);
   val = new interval<chain*>(a_i, largest, chains[m_chain]);
   intervals.append(val);
   equivalence_class->append(val);
   it = mapped_nodes.succ(it);
 }
```

Finally, we add intervals for the preexisting real nodes on \( \min_n \to_{C_i}^* \max_n \). To make sure that only one of their endpoints can lie in another interval, we let them start at 0. The preexisting real nodes generate equivalent intervals. We want to start the order of intervals from the equivalence class of the preexisting real nodes, so we store it in start_interval.

```
{ interval<chain*>* start_interval = NULL;
  slist<interval<chain>*>* equivalence_class =
    new slist<interval<chain>*>();
  { chain_node_iterator it(current_chain, this);
    for (node n = it.next(); n!=NULL; n=it.next()) {
      if (is_real[n] && mapping[n] >= min_n && mapping[n] <= max_n) {
        interval<chain*>* val =
          new interval<chain*>(0, mapping[n], NULL);
        intervals.append(val);
        equivalence_class->append(val);
      }
    }
  }
  start_interval =
    equivalence_class->contents(equivalence_class->last());
  equivalent_intervals.append(equivalence_class);
}
```

As now all intervals are created and the intervals are grouped into appropriate equivalence classes, we can compute an order on them. The algorithm we will use requires the intervals to be provided in a special order: sorted ascending in the left bound and descending in the right, and vice versa in a second list. The reason for this peculiar order will become clear during the discussion in section 2.12.
We use bucket sort to create the two sorted lists.

```cpp

{ 
  unsigned int (*fc)(interval<chain>* const & ) = 
  &interval<chain>::first_component;

  unsigned int (*sc)(interval<chain>* const & ) = 
  &interval<chain>::second_component;

  slist<interval<chain>* >* intervals_asc = 
   bucket_sort<interval<chain>* ,asc, dsc>(
     intervals,fc,sc,0,nodes_in_chain);

  slist<interval<chain>* >* intervals_dsc = 
   bucket_sort<interval<chain>* ,dsc, asc>(
     intervals,sc,fc,0,nodes_in_chain);
}
```

Then we call the algorithm that computes a proper order. Its workings will be discussed in section 2.12. As we shall prove, the graph is not triconnected when no order exists, and a separation pair can be computed from the intervals. In this case an exception with the appropriate bounds will be thrown. We catch the pair and apply the reverse mapping to find the corresponding nodes in G to throw a not_triconnected_exception of our own.

```cpp

std::vector<interval<chain>* >* ordered;

try {
  Order<chain>::compute_order(
    intervals_asc,
    intervals_dsc,
    equivalent_intervals,
    start_interval,
    ordered);
} catch (std::pair<unsigned int, unsigned int> ex) {

  node a = reverse_mapping[ex.first];
  node b = reverse_mapping[ex.second];

  //free resources
  { 
    interval<chain>* val;
    forall(val, intervals) {
      delete val;
    }

    slist<interval<chain>* >* equivalence_class;
    forall(equivalence_class, equivalent_intervals) {
      delete equivalence_class;
    }

    throw not_triconnected_exception(
      "Couldn't add all children cup type3", a,b);
  }
}
```
CHAPTER 2. SCHMIDT’S ALGORITHM

If no exception is thrown, the vector ordered contains an interval for each segment in a proper order. We traverse the vector and add all chains in each segment.

```cpp
for(unsigned int i = 0; i < ordered.size(); i++) {
    if (ordered[i]->cont) {
        add_with_ancestors(ordered[i]->cont);
        if (segment_chains.defined(ordered[i]->cont->number)) {
            chain* chain_in_segment = NULL;
            forall(chain_in_segment, segment_chains[ordered[i]->cont->number]) {
                add_with_ancestors(chain_in_segment);
            }
        }
    }
}
```

Our last duty is to free the resources that we used during the function.

```cpp
{
    interval<chain*>* val;
   forall(val, intervals) {
        delete val;
    }
    slist<interval<chain*>* equivalence_class;
   forall(equivalence_class, equivalent_intervals) {
        delete equivalence_class;
    }
}
```

The correctness of the function add_hard_segments follows from lemma 2.20 and the results we will develop in section 2.12.

2.12 Interlocking Intervals

The primary concern of the function add_hard_segments was the creation of intervals for the attachment nodes of the hard segments and the preexisting real nodes on the union of the dependent paths on $C_i$. We mapped the nodes of $C_i$ to the numbers $1 \ldots |C_i|$ and created an interval $[0, v]$ for each relevant real node. For a segment with attachment nodes $a_1 < a_2 < \ldots < a_k$ we created the intervals

$$
\{[a_1, a_k] \cup \bigcup_{1 < j < k} \{[a_1, a_j], [a_j, a_k]\}\).$$

We will use the graph fragment depicted in figure 2.10 as a running example for this section.

We say two intervals $[a, b]$ and $[c, d]$ interlock iff $a < c < b < d$. The interlock relation can be interpreted as a graph, where the nodes are intervals, and there is an edge between two nodes if the corresponding intervals interlock.
2.12. INTERLOCKING INTERVALS

Figure 2.10: A graph fragment and the intervals for the segments. The intervals are ordered as required by the algorithm of Olariu and Zomaya. Intervals with lower y coordinate come first in the ordering.

We stored the intervals for each segment in a list with the intention of making them equivalent in the sense that it is sufficient for one interval of the segment to interlock with an interval of a different segment to create the necessary real nodes. This equivalence can be mapped onto the interlock graph by merging all nodes that belong to the same interval. In the following we will often use intervals and the segments they represent interchangeably.

Lemma 2.21 ([14, Lemma 90]): There exists a proper order on the segments if and only if the merged interlock graph \( I \) is connected.

Proof. Let \( I \) be connected and let \( S = R, H_1, \ldots, H_k \) be the sequence of merged nodes as they occur in a traversal of \( I \), e.g. by DFS, that starts at \( R \), the node that represents the preexisting real nodes. We show that this order is proper.

Let \( H_i \neq R \) be a node, and let \( H_j \) be a neighbour of \( H_i \) that precedes it in \( S \). Then, there must be an interval in \( H_j \) that interlocks with an interval in \( H_i \). If \( H_j = R \), there must be a preexisting real node on the dependent path of \( H_i \). Otherwise \( H_j \) is a segment with an attachment node on the dependent path of \( H_i \). Since \( H_j \) precedes \( H_i \) in \( S \), the attachment nodes of \( H_j \) are real when \( H_i \) is processed.

On the other hand, let \( H_1, \ldots, H_k \) be a proper order on the segments. We show that the interlock graph is connected by proving that each \( H_j, j > 1 \), is adjacent to some \( H_i, i < j \). Note that \( H_1 \) must have a real node on its dependent path and hence is incident to \( R \). For the other segments, we distinguish several cases. If \( H_j \) has a preexisting real node on its dependent path \( P_j \), it interlocks with \( R \). Otherwise, there are no real nodes on the \( P_j \), and hence an attachment node \( v \) of a segment must have become real before \( H_j \) can be added. Let \( H_i \) be the minimal segment
that has attachment nodes on the dependent path of \( H_j \), and let \( P_i \) be its dependent path. If \( P_i \nsubseteq P_j \), \( P_i \) has an endnode \( u \notin P_j \), and hence either \([u, v]\) or \([v, u]\) must interlock with an interval in \( H_j \). Otherwise \( P_i \subseteq P_j \), and there can be no preexisting real nodes on \( P_i \) if \( H_i \) is minimal.

So to find a proper order on the hard segments, we need to compute a traversal of the interlock graph. Unfortunately, it is not possible to just construct the whole graph, as it might contain quadratically many edges.

Instead, we will apply a sweep line algorithm that has been introduced by Olariu and Zomaya [11]. They describe the algorithm for a CREW PRAM, which complicates the presentation. Olariu and Zomaya assume that all endpoints are distinct. As this is obviously not the case in our application, we use the ordering that was suggested by Lloyd [7]. Note that in our case it is not even guaranteed that all intervals are distinct.

We already generated two lists of intervals in listing 2.13, line 104ff, one sorted by first component ascending and second component descending, and the other one sorted the other way round. The idea of the algorithm in [11] is to generate a spanning tree of the graph by sweeping over the intervals twice, once for each sorted sequence.

The sweeping is done by the function `connect_forest`. We begin by associating each interval with a node in the spanning tree, unless it already has a node.

Listing 2.14: Connecting the forest

```c++
static void connect_forest(
    1  ugraph& g,
    2  const slist<interval<A>*>* input_list,
    3  unsigned int side)
{
    std::vector<interval<A>*> input_array;
    {  
        interval<A>* val=NULL;
        forall(val, input_list) {
            input_array.push_back(val);
            if (val->represented_by == NULL) {
                val->represented_by = g.new_node();
            }
        }
    }
}
```

Assume we sweep over the intervals in ascending order of their left bound, i.e. \( \text{side}=1 \). We want to connect an interval \( I \) by shooting a ray from its right endpoint upwards. The first interlocking interval \( J \) that this ray hits becomes the parent of \( I \) if the intervals interlock. See figure 2.11.

To find \( J \) efficiently, we use the same method that is used to solve an instance of the well known nearest larger value problem (see for example [1]), adapted for
2.12. INTERLOCKING INTERVALS

The observation that enables this approach is that an interval $I$ obstructs the path for any ray to an interval that comes later in the sequence, and whose right bound is smaller than the right bound of $I$.

We keep a stack of intervals. When a new interval $I$ is examined, we first pop all elements from the stack that are obstructed by the new interval. This also makes sure that there are no identical intervals on the stack.

Now we need to check whether the top element actually interlocks with $I$. If it does, we add an edge to the spanning tree. Finally, we push the new interval on the stack.

```cpp
stack<interval<A>*> s;
for(int i=input_array.size()-1; i>=0; i--)
    interval<A>* cur_interval = input_array[i];
    while(!s.empty() &&
        ((side == 1 && s.top()->bounds[1] <= cur_interval->bounds[1]) ||
         (side == 0 && s.top()->bounds[0] >= cur_interval->bounds[0])))
    {
        s.pop();
    }
const bool s1_ta_le_cb =
    s.empty() ||
    (side == 1 && s.top()->bounds[0] < cur_interval->bounds[1]);
const bool s1_ca_le_ta =
    s.empty() ||
    (side == 1 && cur_interval->bounds[0] < s.top()->bounds[0]);
const bool s0_ca_le_tb =
    s.empty() ||
    (side == 0 && cur_interval->bounds[0] < s.top()->bounds[1]);
```

Figure 2.11: The first interlocking interval that is hit by the dashed ray becomes the parent.
**CHAPTER 2. SCHMIDT’S ALGORITHM**

```c
const bool s0_tb_le_cb =
    s.empty() ||
    (side == 0 && s.top()->bounds[1] < cur_interval->bounds[1]);

if (!s.empty() &&
    ((s1_ta_le_cb && s1_ca_le_ta) || (s0_ca_le_tb && s0_tb_le_cb)))
{
    g.new_edge(cur_interval->represented_by, s.top()->represented_by);
}
```

(a) The interlock graph after the first sweep

(b) The interlock graph after the second sweep

It is clear from the algorithm that one sweep over the intervals produces edges that form a subforest of the interlock graph, and that in particular there is only an edge between two intervals if they interlock due to line 39. Further, it is save to assume in the following proofs that all intervals are distinct because the loop in line 17 ensures that no identical intervals are on the stack simultaneously. This way, all identical intervals share at least one neighbour, and it is save to ignore them in the following lemma.

**Lemma 2.22** ([11, Lemma 4.1]): Two sweeps of connect_forest calculate a spanning subgraph of the interlock graph.

**Proof.** Let $E_L$ be the edges from the first and $E_R$ the edges from the second sweep. We show that if two intervals $I, J$ interlock, there is a path between $I$ and $J$ in $E_L \cup E_R$.

Assume otherwise and let $I = [i_a, i_b], J = [j_a, j_b]$ be two interlocking intervals that are not connected by a path in $E_L \cup E_R$ such that $i_a$ in minimal and $i_b$ maximal (i.e. $I$ is the first such interval in the sorted sequence). As $I$ and $J$ interlock, we have

$$i_a < j_a < i_b < j_b.$$  \hfill (2.12.1)
This implies that $I$ can’t be a root in the forest of $E_L$, and $J$ can’t be a root in the forest of $E_R$.

Let $T_L$ be the tree of $I$ in $E_L$, and let $R = [r_a, r_b]$ be its root. Then $i_a < r_a$, and $i_b < r_b$. Equation (2.12.1) together with the fact that $J \not\in T_L$ implies

$$j_b \leq r_b.$$  

Consider the path $I \rightarrow R$ in $T_L$, and let $S = [s_a, s_b]$ be the last interval on the path such that

$$i_a \leq s_a \leq r_a.$$  

(2.12.2)

Note that $S$ is not necessarily different from $R$, and that

$$s_b \leq j_b$$  

(2.12.3)

because $S$ is in $T_L$ but $J$ is not.

Combining (2.12.1), (2.12) and (2.12.3) we get

$$i_a \leq s_a \leq s_b \leq j_b,$$

where one of the inequalities between $i_a$ and $j_a$ must be strict because $I$ and $J$ interlock.

A mirror argument for $T_R$ in $E_R$ provides an interval $Q$ such that

$$q_a \leq i_a < j_a < i_b \leq q_b,$$

where one of the inequalities between $i_b$ and $j_b$ must be strict.

Combining the last two equations we get

$$q_a \leq i_a \leq s_a \leq j_a < i_b \leq q_b \leq j_b \leq s_b$$

Where again there must be at least one strict inequality between $i_a$ and $j_a$ as well as between $i_b$ and $j_b$.

We do a case distinction. Either $q_a = i_a$ and $q_b = i_b$, then $Q = I$, or $q_a = i_a$ while $i_b < q_b$ or $q_a < i_a$. Both cases lead to a contradiction. In the first case, $I$ is connected to $Q = I$, and in the second case $Q$ interlocks with either $S$ or $J$ and comes before $I$ in the sorted sequence, a contradiction to the choice of $I$.  

After we compute a spanning subgraph of the interlock graph, we merge the intervals that correspond to the same segment. We need to make sure that the bounds of the merged intervals are the extreme bounds of their union. Otherwise, we won’t be able to correctly extract a separation pair should the merged interlock graph be disconnected. The function `merge_nodes` is described in appendix B.3.
Listing 2.15: Glueing equivalent nodes

```c++
static void glue_equivalence_classes(
    ugraph& g,
    const slist<slist<interval<A>*>* > equivalent_intervals)
{
    slist<interval<A>*>* a_class;
    forall(a_class, equivalent_intervals) {
        slist<node> nodes_to_merge;
        interval<A>* i;
        unsigned int min_bound = UINT_MAX;
        unsigned int max_bound = 0;
        forall(i, *a_class) {
            min_bound = std::min(min_bound, i->bounds[0]);
            max_bound = std::max(max_bound, i->bounds[1]);
            nodes_to_merge.append(i->represented_by);
        }
        node merged_node = merge_nodes(g, nodes_to_merge);
        forall(i, *a_class) {
            i->bounds[0] = min_bound;
            i->bounds[1] = max_bound;
            i->represented_by = merged_node;
        }
    }
}
```

Combining the above functions, we can build the function compute_order. It performs two sweeps and merges the equivalent nodes. Afterwards it calculates a mapping from nodes to intervals.

Listing 2.16: Checking the interlock graph for connectedness

```c++
static void compute_order(const slist<interval<A>*>* sorted_asc,
    const slist<interval<A>*>* sorted_dsc,
    const slist<slist<interval<A>*>* >& equivalent_intervals,
    const interval<A>* start,
    std::vector<interval<A>* >& output)
throw(std::pair<unsigned int, unsigned int>)
{
    ugraph g;
    connect_forest(g, sorted_asc, 1);
    connect_forest(g, sorted_dsc, 0);
    glue_equivalence_classes(g, equivalent_intervals);

    node_array<interval<A>*> belongs_to(g, NULL);
    { interval<A>* i;
        forall(i, sorted_asc) {
            belongs_to[i->represented_by] = i;
        }
    }
```
2.12. INTERLOCKING INTERVALS

We then proceed to do a DFS on the merged interlock graph. It starts from the node that represents the preexisting real nodes. If the graph is not connected, we extract a separation pair from an unvisited component.

```c
node* ordered = new node[g.number_of_nodes()];
for(unsigned int i=0; i<(unsigned int)g.number_of_nodes(); i++) {
    ordered[i]=NULL;
}
const node startnode = start->represented_by;
bool is_connected = dfs_order(g, startnode, ordered);
```

Lemma 2.23 ([14]): Let \( A \) be a connected component of the merged interlock graph \( I \) that does not contain the node \( r \) that represents the preexisting real nodes. Then, the extreme bounds of the union of intervals in \( A \) are a separation pair in \( G \).

**Proof.** Let \( C_i \) be the chain that is currently processed, and let \( S^R_i \) be the current subdivision. Let \( W \) be the set of segments that is represented by intervals in \( A \). Note that \( W \) must contain a chain \( D \) in \( children12(C_i) \). The union of the dependent paths of \( W \) is a path \( P = x \rightarrow^* y \) that is contained in \( C_i \). The nodes \( x \) and \( y \) are the extreme bounds of the intervals in \( A \). The path \( P \) cannot contain a real inner node due to the choice of \( A \). Hence, every edge in \( E(G) \setminus E(P) \) that is adjacent to an inner node in \( P \) must be contained in some segment \( H \) (if there were real inner nodes on \( P \) the edges might belong to previously added chains and be thus in \( S^R_i \)). \( H \) is either contained in \( W \), or belongs to some other connected component of \( I \) that does not contain \( r \). In both cases, all attachment nodes of \( H \) must be contained in \( P \). Hence, removing \( x, y \) from \( G \) disconnects \( H \) from \( G \).

It remains to show that there is an inner node \( v \) on \( P \). If \( H \) is a back edge, it can’t connect two adjacent nodes on \( P \) because \( G \) is simple. Otherwise, \( H \) contains an inner node \( y \), and, due to property B, there must be a type three chain that starts on \( P \). \( \square \)

To find the separation pair, we do a DFS starting from an unvisited node and iterate over intervals in the connected component to find the extremal bounds. Those are then thrown as an exception.

```c
if (!is_connected) {
    node_array<bool> visited(g,false);
    for(unsigned int i = 0;
        i<(unsigned int)g.number_of_nodes() && ordered[i] != NULL;
        i++)
    {
        visited[ordered[i]] = true;
        ordered[i]=NULL;
    }
    { node n;
```
forall_nodes(n,g) {
    if (!visited[n]) {
        dfs_order(g,n,ordered);
        break;
    }
}

unsigned int min = UINT_MAX;
unsigned int max = 0;
{
    for(unsigned int i = 0; i<(unsigned int)g.number_of_nodes() && ordered[i] != NULL; i++) {
        interval<A>* it = belongs_to[ordered[i]];
        min = std::min(it->bounds[0],min);
        max = std::max(it->bounds[1],max);
    }
}
delete[] ordered;
throw pair<unsigned int,unsigned int>(min,max);

Otherwise, if the merged interlock graph is connected, we push the intervals in the
order of the DFS into a vector that we return to the calling function.

for(unsigned int i=0; i<(unsigned int)g.number_of_nodes(); i++) {
    output.push_back(belongs_to[ordered[i]]);
}
delete[] ordered;

We have already shown in lemma 2.21 and 2.23 that $G$ is not 3-connected if there is
no proper order on the segments. It remains to show that there always is a proper
order for 3-connected graphs.

Lemma 2.24 ([14, Lemma 90]): Let $G$ be 3-connected. Let $C_i$ be the chain that is
currently being processed. Then, there is an order in which the clusters of all
ancestors of the chains in the hard segments $Y$ of $C_i$ can be added.

Proof. If $Y = \emptyset$, the claim follows. We will show by contradiction that $Y \neq \emptyset$
implies the existence of at least one hard segment that can be added.

Suppose $\text{children}12(C_i) \neq \emptyset$ and that the cluster of every chain in $\text{children}12(C_i)$
cannot be added. We examine the possible cases for a chain $D$ in $\text{children}12(C_i)$. Let $H$ be the segment of $D$, and let $P$ be its dependent path.
2.12. INTERLOCKING INTERVALS

1. $D$ is of type one. Thus, $P$ cannot contain a real inner node, or $D$ can be added with lemma 2.11. Further, it must either be a back edge, or be contained in the exceptions 1 or 3 of lemma 2.16, as property B guarantees the existence of a chain from type $3(C_i)$ in $H$.

2. $D$ is of type 2a. Then, there can be no real node on $P\setminus s(D)$ as otherwise lemma 2.11 can be applied.

3. $D$ is of type 2b. In this case, $D$ must be the minimal chain of a bad caterpillar, i.e. $D$ is the chain $D_k$ of exception 2 in lemma 2.16. Note that bad caterpillars don’t have real nodes on their dependent path.

It all cases $P \subseteq C_i$ and does not contain a real node. Therefore, it must be contained in a link $L$. Because $C_i \in S^R$, its endnodes are real, and we have $P \subseteq L \subseteq C_i$. $L$ must have length at least two: if $D$ is a back edge $P$ has length at least two because $G$ is simple; otherwise $P$ must contain an inner node due to property B.

We proceed to show that $L$ must contain an inner node $v$ from which a BG-path $B$ starts. The existence of $B$ will lead to a contradiction.

Because $G$ is 3-connected, lemma 1.3 can be applied, and thus we know that either $L$, or a parallel link $L'$ must contain $v$. We show that $v$ must lie on $L$.

We distinguish two cases on $L$. Suppose it consists solely of tree edges. Then, restriction $(R_2)$ prevents parallel links, unless $S^R = S_3$ and $L = C_0$. Because $G$ is simple, one of the chains $C_1, C_2$, say $C_1$, must contain an inner node. Therefore, the claim follows because $C_1$ must have property B.

If $L$ contains a back edge, $L$ must start at $s(C_i)$ because $L \subseteq C_i$ and $C_i$ contains only one back edge. Let $w$ be the the other end node of $L$. It must be different from $t(C_i)$, because there is a type three chain $C_j$ that starts at a proper ancestor of $C_i$ due to property B. $C_j$ must have already been added according to lemma 2.14. As $C_i$ enters the subtree $T(x)$ of the last inner node $x$ on $C_i$, $C_i$ contains a real inner node because of upwards-closedness.

Now consider a parallel link $L'$ of $L$. We show that $L'$ cannot contain $v$. Note that a link cannot contain a back edge except as its first edge, because all back edges in $S^R$ start with a real node. Since $t(L)$ is a descendant of $s(L)$, the first edge of $L'$ can’t be a tree edge, as otherwise $t(C_i)$ must have been skipped over an inner back edge. This implies that $L'$ must be contained in a chain $C_k$, as it starts with a back edge and can from then on use only tree edges. It has to stop within $C_k$ because $t(C_k)$ is real.

As $L$ is contained in $C_k$, $C_k$ must be a child of $C_i$. Therefore, $C_k$ is of type two. It can’t be of type 2b because otherwise it is part of a caterpillar that has been added completely, and then $C_k$ contains a real inner node. The only possible type for $C_k$ is thus 2a. But then $C_k$ is a back edge, and $L'$ can’t contain any inner nodes. Thus, it must be $L$ that contains an inner node $v$ from which a BG-path $B$ starts.
Because \( B \) is a BG-path, it must be completely contained in a segment \( H' \). We show that the first edge of \( B \) must be contained in a chain of type two and derive a contradiction.

We begin by showing that \( H' \) does not contain any child of \( C_i \). It can’t contain a child of type three as that chain must start at a proper ancestor of \( C_i \) and must thus already be contained in \( S^R_i \), due to lemma 2.14. Suppose then that \( H' \) contains a chain \( D' \in \text{children12}(C_i) \). In this case, \( D' \) must be the minimal chain in \( H' \), and lemma 2.19 implies that the dependent path \( P' \) of \( D' \) contains all attachment nodes of \( H' \). In particular, \( P' \) contains the endpoints of \( B \). Because we assumed that no chain in \( \text{children12}(C_i) \) can be added to the subdivision, \( P' \) may not contain a real inner node. Consequently, \( P' \), and with it the endnodes of \( B \), are contained in a link \( Q \). Since \( v \) is not real in \( S^R_i \), it follows that \( v \) is not an endnode of \( Q \) and thus \( B \) violates BG-property 2. We conclude that \( H' \) does not contain any children of \( C_i \).

We use this result to show that the first edge of \( B \) must be included in a chain of type two by showing that \( H' \) can’t contain chains of types either one or three that start at \( v \).

Assume \( H' \) contains a chain \( C_a \) of type one that starts at \( v \). Let \( C_b \) be the parent of \( C_a \). We have \( C_b \neq C_i \) as otherwise \( C_a \) is a child of \( C_i \). Because \( C_a \) is of type one and starts at \( v \), \( t(C_b) = v \) since \( v \) is already contained in \( C_i \).

Then, \( C_b \) must be a child of \( C_i \). On the other hand, let \( H' \) contain a chain \( C_a \) of type three that starts at \( v \). In this case, \( C_a \in \text{type3}(C_i) \), and one of its ancestors in \( H' \) must be in \( \text{children12}(C_i) \).

Consequently, the chain \( J \) that contains the first edge of \( B \) must be of type two. This finally leads to a contradiction.

Let \( C_a \) be the closest ancestor of \( J \) that is not of type two. Because \( J \) has type two, it follows that \( s(C_a) = s(J) = v \). Since \( v \) is not real, \( C_a \in H' \). This is a contradiction as there can be no chain of type either one or three in \( H' \) that starts at \( v \).

It follows that \( B \) can’t exist, and hence there must be a hard segment that can be added. Iterating this argument over the remaining segments yields a proper sequence. \( \Box \)

### 2.13 Constructing and Verifying the Certificate

During the algorithm, we created BG-paths for the clusters we added to the subdivision (see listing 2.9). In this section, we discuss how the certificate is built from these paths, and how it can be verified.

For each BG-path that we add to the certificate, we need to check all BG-properties. The most practical way to do so is to reconstruct the whole graph from the BG-paths and then delete them in reversed order. This way, the first property can be easily checked when the chain is added: exactly two nodes of the chain must already
exist in the new graph. The other two properties are then checked when the path is deleted.

We will begin by examining the function `add_bg_path`. Besides the version that is shown here, there is an overloaded version that takes a chain as argument and turns it into a list of edges before calling this function.

It adds the necessary nodes and edges to the new graph and updates an `edge_array` for the original graph that keeps track of the edges that we encountered as part of a BG-path. If all entries in the array are true, the original graph and the new graph must be isomorphic. Further, it checks BG-property 1 for each added path. The variable `still_valid` is `true` if all paths so far have the property. For each BG-path we store its edges in the new graph as a list. Each edge has a pointer to its list item to facilitate the deletion of the edges.

When iterating over a list of edges, we see two new nodes for the first edge and only one node for each additional edge. We distinguish these cases.

**Listing 2.17: Adding a BG-path to the certificate**

```cpp
bool certificate::add_bg_path(list<edge> const & edges) throw() {
    if (!still_valid)
        return false;

    node first_node = NULL;
    node last_node = NULL;
    node current_node = NULL;
    paths.push_back(new list<edge>());
    unsigned int prev_created_nodes = 0;
    node last=NULL,nodes[2]={NULL,NULL};
    unsigned int num=0;
    edge e;
    forall(e,edges) {
        edge_accounted_for[e]=true;
        if (last==NULL) {
            nodes[0] = source(e);
            nodes[1] = target(e);
            num = 2;
            last = target(e);
        } else {
            nodes[0] = opposite(last,e);
            nodes[1] = NULL;
            num=1;
            last = opposite(last,e);
        }
    }
    
    We then iterate over the nodes array and create new nodes in the new graph. We also store a mapping from nodes in the original graph to nodes in the new graph, so that we know between which nodes in the new graph we have to add edges when we process an edge from the old graph.
```
for(unsigned int i = 0; i<num; i++) {
    current_node = nodes[i];
    if (first_node == NULL)
        first_node = current_node;

    node new_node;
    if (orig_2_new[current_node] == NULL) {
        new_node = new_graph.new_node();
        orig_2_new[current_node] = new_node;
    } else {
        new_node = orig_2_new[current_node];
        prev_created_nodes++;
    }

    if (last_node != NULL) {
        node prev_node = orig_2_new[last_node];
        edge added_edge = new_graph.new_edge(prev_node, new_node);
        list<edge>::item it = chains.back()->append(added_edge);
        edge_items[added_edge] =
            new pair<list<edge>*, list<edge>::item>(chains.back(), it);
    }
    last_node = current_node;
}

If we already processed one node on this path, i.e. when last_node is not NULL, we add an edge to the new graph and store it in the list for the current path. We also store a pointer to the list item in edge_items.

if (last_node != NULL) {
    node prev_node = orig_2_new[last_node];
    edge added_edge = new_graph.new_edge(prev_node, new_node);
    list<edge>::item it = chains.back()->append(added_edge);
    edge_items[added_edge] =
        new pair<list<edge>*, list<edge>::item>(chains.back(), it);
}
last_node = current_node;

When the path has been completely processed, we check whether it fulfills BG-property 1 and store its endnodes.

still_valid &= chains.size() == 1 || prev_created_nodes == 2;
still_valid &= first_node != last_node;
return still_valid;

After we reconstructed the graph from the paths, we can call the function verify to check BG-properties 2 and 3. First, we check whether the constructed graph is isomorphic to the original graph by iterating over the edge_accounted_for array.

Listing 2.18: Checking a certificate

bool certificate::verify() throw() {
    if (!still_valid)
        return false;
    bool isomorphic = true;
    { edge e;
2.13. CONSTRUCTING AND VERIFYING THE CERTIFICATE

```cpp
forall_edges(e, the_graph) {
    isomorphic &= edge_accounted_for[e];
}

if (!isomorphic)
    return false;
```

Now the actual processing starts. We delete paths in the reverse order in which they were added and smoothen all nodes that become nonreal in the process. This guarantees that each path in a valid certificate has length one when it is deleted (cf. the proof of theorem 1.2.2). Then, we can apply the definition of BG-operations (see definition 1.1 on page 4) to check the correctness in constant time per path.

We begin by determining whether none, one, or both endnodes of the current path will be smoothened away after its deletion. This way, we can distinguish the different BG-operation types that the path might represent.

```cpp
for (int the_path = paths.size()-1; the_path>=3; the_path--) {
    list<edge>* current_path = paths[the_path];

    if (current_path->size() != 1)
        return false;

    const edge bg_edge = current_path->Pop();
    delete(current_path);
    paths[the_path]=NULL;

    const node a = source(bg_edge);
    const node b = target(bg_edge);

    const bool a_smooth = new_graph.degree(a) == 3;
    const bool b_smooth = new_graph.degree(b) == 3;

    unsigned int type = 0;
    if (a_smooth) type++;
    if (b_smooth) type++;

    delete(edge_items[bg_edge]);
    edge_items[bg_edge] = NULL;
    new_graph.hide_edge(bg_edge); //could also delete
}
```

Now we do a case distinction over the type of BG-operation this edge represents. First, we handle the case where both end nodes will be removed from the graph, i.e. BG-operation 1.3(c). We begin by collecting the neighbours of a and b.

```cpp
switch(type) {
    case 2: {
        node a_neighbours[2];
        node b_neighbours[2];
```

Then we check whether the edges we’re subdividing are parallel. This is only allowed for the third path, because it creates a $K_4$ subdivision from the initial $K_{2,3}$ subdivision.

In the next step, we remove the two endpoints from the graph. This means in particular that we need to update the BG-paths in which they are contained. To do so, we look the paths for all adjacent edges of each node in edge_items up and delete those edges. We store the paths on which they lie in lists so that we can afterwards add the edge that is created by the smoothening.
After deleting the obsolete edges, we smoothen both nodes and insert the new edges back into the paths. The function smoothen is discussed in appendix B.4.

```cpp
edge es[2] = {NULL, NULL};
es[0] = smoothen(new_graph, a);
es[1] = smoothen(new_graph, b);
for (unsigned int i = 0; i < 2; i++) {
    list<edge>::item it = lists[i]->append(es[i]);
    edge_items[es[i]] = new pair<list<edge>*, list<edge>::item>(lists[i], it);
}
}
```

The next case we’re handling is simpler. If only one of the endnodes will be deleted, the edge corresponds to BG-operation 1.3(b), i.e. we subdivide one edge and connect the new node to an already existing node. First, we find out which node was created and which node is the third node to which the edge connects.

```cpp
case 1: {
    node third, sub;
    if (a_smooth) {
        third = b;
        sub = a;
    } else {
        third = a;
        sub = b;
    }
}
```

The third node must be different from the endpoints of the subdivided edge. To verify this condition, we collect the neighbours of the created node and compare them to the third node.
The last step is to smoothen the created node. This works quite similar as before. First we delete the adjacent edges:

```cpp
{ 
    list<edge>* l=NULL;
    { 
        edge e;
        forall_adj_edges(e,sub) { 
            pair<list<edge>*, list<edge>::item>* p = edge_items[e];
            p->first->del_item(p->second);
            l = p->first;
            delete(edge_items[e]);
            edge_items[e]=NULL;
        }
    }
}
```

Then we smoothen the node and insert the new edge back into the path.

```cpp
edge e1 = smoothen(new_graph,sub);
list<edge>::item it = l->append(e1);
edge_items[e1] = new pair<list<edge>*, list<edge>::item>(l,it);
}
} break;
```

The remaining BG-operation – connecting two real nodes – needs no additional checks.

```cpp
case 0: { 
} break;
default: 
    assert(false);
}
```

It remains to confirm that the first three paths form a $K_{2,3}$. This is done by comparing their endpoints.
If all BG-paths pass these tests, the certificate is valid and the algorithm’s answer is correct.
Chapter 3

Experimental Evaluation and Conclusion

3.1 Computational Evaluation

For comparison purposes, we also implemented the algorithm of Hopcroft and Tarjan [6]. Our implementation is much faster than the implementation of Gutwenger and Mutzel [5], i.e. with our implementation graphs with \( n + m > 200,000 \) can be processed in about four seconds on a 400MHz Pentium II with 128Mb RAM (Figure 3.1). Most likely, this is because we don’t store the information necessary to construct an SPQR-tree, but instead just check for 3-connectedness.

We tested both algorithms on several kinds of graphs. First, we generated skeletons of 3D polygons by sampling random points on a three dimensional sphere and calculating their convex hull. These graphs are known to be 3-connected [2]. The results of this experiment are shown in figure 3.2.

On the sampled set of polygons our implementation of Schmidt’s algorithm was on average 3.3 times slower than the algorithm of Hopcroft and Tarjan.

We also sampled random graphs with \( m = n(0.5 \ln n + 1.5 \ln \ln n - 0.71) \), an edge density chosen according to the results in Erdős and Rényi [4] such that about half of the graphs were 3-connected. The results are shown in figure 3.3.

On the generated graphs the algorithm of Hopcroft and Tarjan was on average 5.9 times faster than Schmidt’s algorithm for the 3-connected instances, whereas Schmidt’s algorithm was on average faster by a factor of 11.8 on instances with lower connectivity.

Based on this observation, we ran additional tests to determine whether Schmidt’s algorithm is only faster for relatively sparse graphs with lower connectivity. We generated two kinds of graphs with increasing edge density. The first type was constructed by taking two graphs with 20,000 nodes and \( m/2 \) edges each and gluing them together at a pair of nodes. This ensures that the resulting graph
with roughly 40,000 nodes and \( m \) edges is never 3-connected, independent of the edge density. The other kind of graph contains 40,000 nodes and \( m \) edges without ensuring any kind of separation pair. Hence, these graphs are almost always 3-connected for high edge densities. The results of these tests are shown in figure 3.4.

One can clearly see how the connectivity of the graph influences the runtime of both algorithms. Schmidt’s algorithm finds articulation points much faster than our implementation of Hopcroft and Tarjan’s algorithm, most likely because of the preliminary checks we perform. This explains the data in figure 3.3.

For dense 2-connected graphs the average runtime of the two algorithms is virtually identical, but Schmidt’s algorithm exhibits a much wider variance. On the instances where it’s faster, it beats Hopcroft-Tarjan by a factor of 1.7; whereas it is on average slower by a factor of 2 on the other instances. In the case of dense 3-connected graphs Schmidt’s algorithm is much slower than the algorithm of Hopcroft and Tarjan, on average by a factor of 3.7.

Figure 3.1: Running the algorithm of Hopcroft and Tarjan on a 400MHz Pentium II with 128Mb RAM. The graphs are sampled as in figure 3.3
3.1. COMPUTATIONAL EVALUATION

Figure 3.2: The algorithm of Schmidt is shown in red, Hopcroft and Tarjan in blue. Time is on an AMD Opteron with 2.8GHz.
Figure 3.3: Random graphs with $m = n(0.5 \ln n + 1.5 \ln \ln n - 0.71)$. Schmidt’s algorithm in red and orange, Hopcroft and Tarjan in blue and cyan. Orange and cyan triangles result from graphs with connectivity $< 3$. Time is on an AMD Opteron with 2.8GHz.
Figure 3.4: Random graphs with \( n = 40000 \). Schmidt’s algorithm in red, Hopcroft-Tarjan in blue. The upper row shows graphs that are never 3-connected by construction. Black curves indicate (from left to right) the probability for the graph to be 1-, 2-, or 3-connected according to [4]. The dotted vertical line shows the density used for the experiments of figure 3.3. Time is on an AMD Opteron with 2.8GHz
3.2 Conclusions and Future Work

In this thesis we showed an implementation of Schmidt’s algorithm and compared it against the standard algorithm by Hopcroft and Tarjan. Unlike the latter, Schmidt’s algorithm provides a certificate for its answers and thus increases the confidence in its correctness. But the certification comes at a cost, both in runtime as well as in development complexity. Schmidt’s algorithm is on average slower by a factor three to six on 3-connected graphs, while simultaneously having more than twice as many lines of code.

In this context, it would be interesting to see how much the code could be simplified and the runtime improved by removing all parts that are only necessary for the construction of the certificate. But even without these changes the runtime in practice can most likely be improved significantly, for example by replacing linked lists by more cache friendly datastructures where applicable.
Appendix A

Datastructures

A.1 Chains

A chain stores its start- and endnodes, s and t, a pointer to its parent, and its type. The type of a chain is explained in detail in section 2.5. We also store a list of children of types one or two and a list of type three chains whose start nodes lie on $t \rightarrow_\tau^* s$. We want those lists to contain only chains that are not yet added to the subdivision. To facilitate the necessary updating, each chain keeps a pointer to the list in which it is contained. Lastly, a chain stores a few bool variables, the back edge from which it starts, first_edge; and its number.

Listing A.1: Datastructure for a chain

```c
enum chain_type {one, two_a, two_b, three_a, three_b, unmarked};
class chain {
  private:
    chain* parent;
    node s;
    node t;
    list<chain*>* delete_from_list;
    list<chain*>::item delete_from_item;
    bool in_subdivision;
    chain_type the_type;
  
  public:
    unsigned int number;
    list<chain*> children12;
    list<chain*> type3;
    edge first_edge;
    bool is_backedge;
    bool is_marked;

  /* get_ and set_ methods for private members not shown */
}
```
When a chain \( C_i \) of type three is created, we add it to the list type3 of the chain in which \( s(C_i) \) is contained and store a pointer to that list so that we can update it once \( C_i \) is added to the subdivision.

```cpp
inline void add_to_type3_of(chain* c) {
    delete_from_list = &(c->type3);
    delete_from_item = delete_from_list->append(this);
};
```

When we set the type of a chain to one or two, we add the chain to the children12 list of its parent.

```cpp
void set_type(chain_type t) {
    the_type = t;
    if (parent!=NULL) {
        switch(get_type()) {
            case three_a: case three_b: break;
            case unmarked: assert(false);
            default: {
                delete_from_list = &(parent->children12);
                delete_from_item = delete_from_list->append(this);
            } } } );
```

Finally, when a chain is added to the subdivision, it deletes itself either from type3, or from children12 and sets its in_subdivision flag.

```cpp
inline void add_to_subdivision(void) {
    in_subdivision = true;
    if (delete_from_list != NULL) {
        delete_from_list->del_item(delete_from_item);
    }
};
```

## A.2 Caterpillars

We implement caterpillars as singly linked lists of chains. To also keep track of the parent, we extend the slist class that is provided by LEDA.

**Listing A.2: The caterpillar class**

```cpp
class caterpillar : public slist<chain* const> {
public:
    caterpillar(void);
    item append( chain* const & element);
    chain* parent;
};
```
The function append updates the parent pointer if the appended chain’s parent is smaller than the current parent. As only type 2b chains are added to the list, get_parent() can never return NULL.

```cpp
caterpillar::item caterpillar::append(chain* const & element) {
    if (parent==NULL || element->get_parent()->number<parent->number)
        parent=element->get_parent();
    return slist<chain* const>::append(element);
}
```

The list contains only the type 2b chains of the caterpillar. We store the lists in an h_array that is indexed with type 3b chains.

### A.3 Intervals

An interval is essentially a tuple of bounds. It stores a content value and a node by which it is represented in the interlock graph. To facilitate sorting by the bounds, it provides two static functions that can be passed as function pointers to bucket sort.

#### Listing A.3: The interval data structure

```cpp
template<typename A> class interval{
public:
    A cont;
    unsigned int bounds[2];
    leda::node represented_by;
    interval(unsigned int l, unsigned int r, A content) : cont(content) {
        if (l>r)
            std::swap(l,r);
        bounds[0] = l;
        bounds[1] = r;
        represented_by = NULL;
    }

    static unsigned int first_component(interval<A> * i) {
        return i->bounds[0];
    }

    static unsigned int second_component(interval<A> * i) {
        return i->bounds[1];
    }
};
```
A.4 Certificates

The certificate class we used for the algorithm stores a reference to the original graph, a pointer to the decomposition object and several node and edge arrays.

Listing A.4: Datastructure for a certificate

```cpp
class certificate {
private:
  bool still_valid;
  ugraph & the_graph;
  ugraph new_graph;
  schmidt_triconnectivity* decomposition;
  node_array<int> created_by_chain;
  std::vector<list<edge>*> chains;
  edge_array<std::pair<list<edge>*, list<edge>::item>*> edge_items;
  node_array<node> orig_2_new;
  std::vector<std::pair<node,node>*> endnodes;
  edge_array<bool> edge_accounted_for;

certificate(ugraph & graph, schmidt_triconnectivity* d);  
~certificate();
  bool add_bg_path(list<edge> const & edges) throw();
  bool add_bg_path(const chain * a_chain) throw();
  bool verify(void) throw();
};
```

The implementation details of the methods are described in section 2.13.
Appendix B

Helper Functions

B.1 Bucket sort

We perform bucket sort as usual, i.e. by distributing the elements \( a \in A \) into buckets according to some function \( f : A \to \text{int} \). After distributing the elements, we traverse the buckets to build the sorted result. Depending on a type parameter, we either sort ascending or descending.

We provide different versions of bucket sort. The first version doesn’t store duplicate elements.

Listing B.1: Bucketsort

```cpp
enum ord { asc, dsc};

template <typename A, ord o> leda::slist<A> unique_bucket_sort(
  leda::slist<A> const & elems,
  unsigned int (*to_int)(const A&),
  unsigned int start,
  unsigned int end)
{
  A* buckets = new A[end-start+1];
  bool* set = new bool[end-start+1];
  for(unsigned int i = 0; i<end-start+1; i++) {
    set[i] = false;
  }
  A elem;
  forall(elem, elems) {
    const unsigned int elem_v = to_int(elem);
    buckets[elem_v-start] = elem;
    set[elem_v-start] = true;
  }
}
```
leda::slist<A> ret;
switch(o) {
  case asc :
    for(unsigned int i=0; i<(end-start+1); i++) {
      if (set[i]) ret.append(buckets[start+i]);
    } break;
  case dsc:
    for(int i=end-start; i>=0; i--) {
      if (set[i]) ret.append(buckets[start+i]);
    } break;
} delete[] buckets;
delete[] set;
return ret;
}

The second version collects duplicate elements by storing a list of elements at each bucket.

```c++
template<typename A, ord o> leda::slist<A> bucket_sort(
  leda::slist<A> const& elems,
  unsigned int (*to_int)(const A&),
  unsigned int start,
  unsigned int end)
{
  leda::slist<A> buckets = new leda::slist<A>[end-start+1];
  A elem;
  forall(elem, elems) {
    const unsigned int elem_v = to_int(elem);
    buckets[elem_v-start].append(elem);
  }
  leda::slist<A> ret;
  switch(o) {
    case asc :
      for(unsigned int i=0; i<(end-start+1); i++) {
        ret.concat(buckets[start+i]);
      } break;
    case dsc:
      for(int i=end-start; i>=0; i--) {
        ret.concat(buckets[start+i]);
      } break;
  } delete[] buckets;
  return ret;
}
```

And lastly we provide a version that performs two passes of bucket sort on the data.
B.2 Depth First Search

The following functions performs a DFS on a graph $g$ and stores the nodes in the reference parameter ordered in the order as they are visited during the DFS. It returns $\text{true}$ if $g$ is connected.

Listing B.2: Depth First Search

```cpp
bool dfs_order(const ugraph& g, const node startnode, node ordered[]) {
    unsigned int number_of_nodes = g.number_of_nodes();
    node* stack = new node[number_of_nodes];
    node_array<bool> visited(g, false);
    const node* base = stack;
    stack[0] = startnode;
    visited[*stack] = true;
    ordered[0] = *stack;
    unsigned int number_seen = 1;
    unsigned int dfi = 0;
    while(stack>=base) {
        const node current = *stack;
        ordered[dfi++] = current;
        stack--;
        node next_node;
        forall_adj_nodes(next_node, current) {
            if (!visited[next_node]) {
                stack++;
                *stack = next_node;
                visited[next_node] = true;
                number_seen++;
            }
        }
    }
    delete[] base;
    return number_seen==number_of_nodes;
}
```
B.3 Merging Nodes

The function `merge_nodes` merges a list of nodes into one while keeping the graph simple. It works in two phases. First, all edges are collected, and a `node_array` that collects nodes to which no more edge may be added is initialized to the nodes that we want to merge.

```cpp
node merge_nodes(ugraph& g, slist<node> nodes) {
    node new_node = g.new_node();
    slist<node> edge_to;
    node_array<bool> no_more_edges_to(g, false);

    { node u;
        forall(u, nodes) {
            no_more_edges_to[u] = true;
            node v;
            forall_adj_nodes(v, u) {
                edge_to.append(v);
            }
        }
    }

    { node v;
        forall(v, edge_to) {
            if (no_more_edges_to[v]) continue;
            no_more_edges_to[v] = true;
            g.new_edge(new_node, v);
        }
    }

    { node v;
        forall(v, nodes) {
            //g.del_node(v);
            g.hide_node(v);
        }
    }
    return new_node;
}
```

In a second step, a new node is added and connected to all edges we collected. Afterwards, the old nodes are hidden from the graph. In principle one could also delete them, but deletion is a much more costly operation than hiding.
B.4 Smoothen

The function smoothen is very simple. It takes a node of degree two, stores its two neighbours, deletes it from the graph, and inserts a new edge between the neighbours. The new edge is returned.

Listing B.4: Smoothening

```c
1 edge smoothen(ugraph & g, node n) {
2   edge e=NULL;
3   node neighbours[2]=(NULL,NULL);
4   { unsigned int i = 0;
5     edge e;
6       forall_adj_edges(e,n) {
7         neighbours[i++] = opposite(e,n);
8       }
9     }
10   e = g.new_edge(neighbours[0], neighbours[1]);
11   g.del_node(n);
12   return e;
13 }
```

B.5 Adding a chain to the subdivision

The function add_with_ancestors adds the BG-paths for a cluster to the certificate and calls add_to_subdivision on all ancestors that are not yet added.

Listing B.5: "Adding a chain and its ancestors"

```c
void schmidt_triconnectivity::add_with_ancestors(chain* a_chain) throw() {
1   decompose_to_bg_paths(a_chain);
2
3   stack<chain*> ancestors_in_segment;
4   for(chain* cur_chain=a_chain;
5       cur_chain->get_parent() != NULL &&
6       !cur_chain->is_in_subdivision() ;
7       cur_chain = cur_chain->get_parent())
8   {
9     ancestors_in_segment.push(cur_chain);
10   }
11
12   while(ancestors_in_segment.size()>0) {
13     add_to_subdivision(ancestors_in_segment.pop());
14   }
```
The function `add_to_subdivision` does nothing more than calling the method of the same name on the chain and marking the endpoints of the chain as real. We increment `chains_in_subdivision` to enable an early exit of the processing loop in the function `certify` (see listing 2.10).

Listing B.6: Adding to the subdivision

```cpp
void schmidt_triconnectivity::add_to_subdivision(chain* c) throw() {
    c->add_to_subdivision();
    is_real[c->get_s()] = true;
    is_real[c->get_t()] = true;
    chains_in_subdivision++;
}
```
Bibliography


