technische universität
dortmund

Algorithms for Planar Graph Augmentation

Bernd Thomas Zey

Algorithm Engineering Report
TR09-1-009
Dec. 2009
ISSN 1864-4503

Faculty of Computer Science
Algorithm Engineering (Ls11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

technische universität
dortmund

Diplomarbeit

Algorithms for Planar Graph Augmentation

Bernd Thomas Zey

15. Dezember 2008

Gutachter:
Prof. Dr. Petra Mutzel
Dipl.-Inform. Carsten Gutwenger

Fakultät für Informatik
Algorithm Engineering (Ls11)
Technische Universität Dortmund
http://ls11-www.cs.uni-dortmund.de

Acknowledgement

The reader may forgive me for writing this in German:

An dieser Stelle möchte ich mich bei all jenen bedanken, die mich während der Erstellung dieser Diplomarbeit und während des gesamten Studiums unterstützt und begleitet haben.

Als erstes danke ich meinen beiden Betreuern, Prof. Petra Mutzel und Carsten Gutwenger, für die vielen guten Gespräche, die hilfreichen Ideen und Korrekturen. Martin Gronemann danke ich für die unzähligen Diskussionsrunden und für die gemeinsame Studienzeit. Ein großer Dank geht an meine Eltern, die mich immer unterstützt und mein Studium überhaupt ermöglicht haben. Außerdem möchte ich meinen Großeltern für die tägliche Versorgung mit Kaffee und Kuchen danken.

Short Abstract

This diploma thesis deals with several special cases of the *Planar Augmentation Problem*. Here, we search for a minimum number of edges whose addition biconnects the graph while planarity is preserved. In general, this optimization problem is $\mathcal{NP}$-hard and the previously best known approximation algorithm achieves a ratio of $\frac{5}{3}$. However, we present a counter-example which shows that its ratio is only two. By constructing a polynomial-time reduction from the *Planar Vertex Cover Problem*, we prove that the problem remains $\mathcal{NP}$-hard even in the restricted case where all cutvertices belong to one biconnected component and the related SPQR-tree (without Q-nodes) of this subgraph has height one. Furthermore, we present a new polynomial-time approximation algorithm for this special case with ratio $\frac{5}{3}$. The approach relies on the decomposition of the biconnected core into its triconnected components by the SPQR-tree. Another special case considers the planar augmentation on graphs with additionally fixed embedding. Here, an optimal augmenting set can be computed efficiently. The developed algorithm works on the BC-tree and runs in time $\mathcal{O}(|V| + |E| + \alpha(|V|)|V|)$.

Zusammenfassung

Diese Diplomarbeit behandelt mehrere Spezialfälle des *Planaren Augmentierungsproblems*. Dabei wird eine minimale Kantenmenge gesucht deren Hinzufügen den gegebeneen Graphen zweizusammenhängend macht und die Planarität beibehält. Im allgemeinen Fall ist dieses Problem $\mathcal{NP}$-schwierig und der beste bekannte Approximationsalgorithmus erreicht eine Güte von $\frac{5}{3}$. Allerdings präsentieren wir ein Gegenbeispiel, welches zeigt, dass die Güte lediglich zwei beträgt. Durch eine polynomielle Reduktion von dem *Planar Vertex Cover*-Problem können wir zeigen, dass das Problem sogar für den eingeschränkten Fall $\mathcal{NP}$-schwierig bleibt, bei dem alle Schnittknoten zu derselben Zweizusammenhangskomponente gehören und der zugehörige SPQR-Baum (ohne Q-Knoten) eine Höhe von eins hat. Darüber hinaus präsentieren wir für diesen Spezialfall einen polynomiellen Algorithmus mit Güte $\frac{5}{3}$. Der Ansatz basiert auf der Zerlegung des zweizusammenhängenden Kerns in die Dreizusammenhangskomponenten durch den SPQR-Baum. Ein weiterer Spezialfall betrachtet die planare Augmentierung auf Graphen bei denen die Einbettung zusätzlich fixiert ist. Hierfür kann eine optimale Lösungsmenge effizient berechnet werden. Der entwickelte Algorithmus arbeitet auf dem BC-Baum und hat eine Laufzeit von $\mathcal{O}(|V| + |E| + \alpha(|V|)|V|)$.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, sowie Zitate kenntlich gemacht habe.

_____

(Bernd Thomas Zey)

Contents

Chapter 1

Introduction

Graphs are a mathematical structure that represent objects and their relations. In classical graph theory, the objects are called *vertices* or *nodes* and a relation between two objects is called an *edge*. Graphs are utilized to visualize informations in a wide field of applications, not only in mathematics and computer science, but also in everyday life. Roadmaps, genealogical trees and organigrams are only three examples. Figure 1.1 illustrates a typical subway map, in this case of London. The stations are the vertices and the rail tracks between the stations correspond to the edges.

Of course, graphs play a very important role in various fields of computer science, for example:

- *Software Engineering*: Graphs and diagrams illustrate workflows and relationships between objects. Examples are UML (Unified Modeling Language) diagrams like class, activity, sequence, and state diagrams.
- *Database Design*: (S)ER ((Structured) Entity Relationship) diagrams model the relationship between the different tables of a database.
- *VLSI (Very-Large-Scale Integration) – Chip Design*: Chip designers use graphs to visualize the wiring scheme.
- *General computer science*: BBD's (Binary Decision diagrams) are a graph-based data structure for boolean formulas and petri nets describe discrete distributed systems.

The main advantage of representing information as graphs is the possibility of visualizing graphs by drawing them into the plane and making the information readable and easier accessible. But not every drawing of a graph is "good". Simple graphs with few objects are easy to draw by hand, but since graphs grow in complexity and size, it becomes more difficult to find an appropriate layout. The problem of computing a nice drawing is known under the term *(automatic) graph drawing*, which is an important research field in computer science. Introductory information about graph drawing algorithms can be found in [1], [29], and [34].

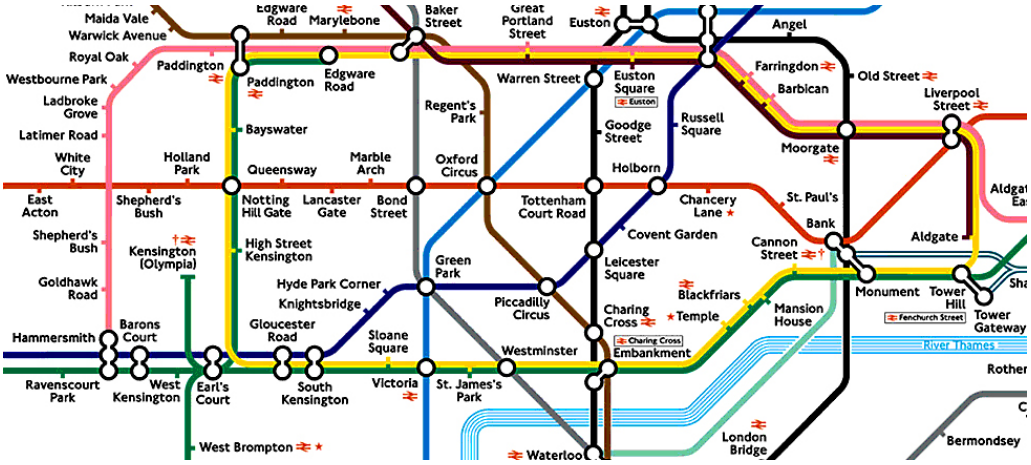


Figure 1.1: An excerpt of London's subway map.

Like mentioned above, not all drawings are nice and easy to read. There do exist certain *aesthetic criteria*, which should, if they are optimized, lead to good drawings.

- *Minimum number of crossings*: Crossings reduce the readability of a drawing, since they make it more difficult to follow the edges and understand the structure of the graph. This criterion is often considered to be the most important one. In VLSI design it is very important that the number of crossings is minimized, since a crossing of two wires increases the number of layers of the chip.

- *Minimum number of bends*: The number of bends is important for orthogonal drawings and the minimization of this criterion leads to simpler drawings.

- *Minimum drawing area*: Usually, diagrams are presented on paper or an output device, which have limited visible area. A large drawing has to be scaled down to fit the output size, thus decreasing its readability.

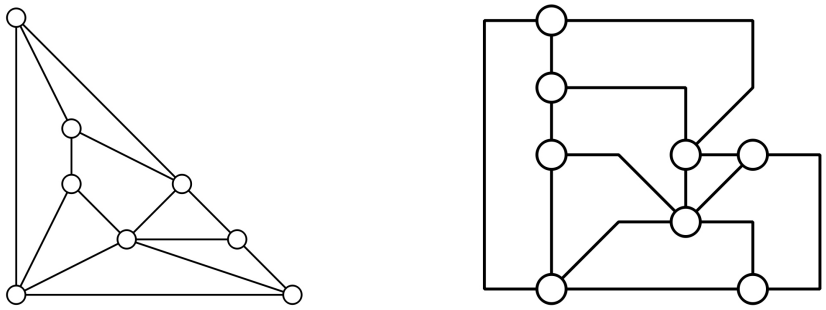
- *Maximum angle*: Adjacent edges should have an adequate angle at the common vertex because otherwise it is difficult to distinguish between single edges.

- *Small edge-length*: Unnecessary long distances between adjacent vertices complicate the identification of relations.

- *Symmetries*: If a graph contains symmetries then the drawing should make them visible.

In general, it is not possible to optimize all criteria. The minimization of crossings is actually an $\mathcal{NP}$-hard problem, see [17]. Furthermore, the optimization of two criteria sometimes works conversely. For example, a layout that emphasizes symmetric structures might have more crossings than a non-symmetric layout.

There are many different approaches for graph drawing algorithms. One of them is the class of straight-line grid drawings for planar graphs which are based on an



(a) (b)

Figure 1.2: (a) An example for the layout computed by a straight-line algorithm and (b) the same graph layouted by the mixed-model algorithm.

ordering of vertices, namely the *canonical ordering*. De Fraysseix, Pach, and Pollack introduced the ordering for maximal planar graphs, see [14]. Kant generalizes the ordering in [30] to work on triconnected planar graphs, too. A typical result of a straight-line layout is presented in Figure 1.2 (a), whereas Figure 1.2 (b) illustrates a drawing computed by the mixed-model algorithm.

These algorithms require planar and triconnected graphs. In [20], Gutwenger and Mutzel modified the first phase of the algorithms to work also for biconnected planar graphs. Both authors also improved the mixed-model algorithm in [21].

To apply such an algorithm on general graphs, they have to be made biconnected by adding new edges. Afterwards, an appropriate layout can be computed and the inserted edges can be removed. Since the added edges have a great influence on the layout, we search for the minimum number of edges to be added. Furthermore, it is necessary that the inserted edges preserve planarity. The optimization problem of inserting the minimum number of edges into a planar graph for obtaining a planar and biconnected graph is called the *Planar Augmentation Problem*. It has been introduced by Kant and Bodlaender in [31] and is investigated further within this work.

This thesis is organized as follows. In Chapter 2, we introduce the fundamentals needed throughout this thesis. We will define the central terms around *embeddings*, *planar graphs* and *biconnectivity*, as well as two data structures for graph decomposition, the *BC-* and the *SPQR-tree*. Chapter 3 gives an introduction to augmentation problems with focus on the *Planar Augmentation Problem*. Furthermore, we present complexity results, previous ideas, and approaches for several augmentation problems. Chapter 4 deals with a restricted version of the *Planar Augmentation Problem*. Here, the embedding of the graph is fixed and thereby, an optimum solution becomes efficiently solvable. Another special case of the *Planar Augmentation Problem* is considered in Chapter 5. There, the input graphs have a restricted biconnected structure, that is all cutvertices belong to one biconnected component. Although this problem seems not as complex as the general case, we show that it is also $\mathcal{NP}$-hard. This new polynomial-time reduction implies that the problem remains $\mathcal{NP}$-hard even in case the SPQR-tree (without Q-nodes) has only height

one. We introduce a new approximation algorithm based on SPQR-trees for this special case with approximation ratio $\frac{5}{3}$. Finally, the last chapter summarizes the results and gives an outlook on possible future work.

Chapter 2

Preliminaries

In this chapter we introduce the fundamentals required throughout this thesis. We start in Section 2.1 with basic terms and results from graph theory. In Section 2.2, we consider *planar graphs* and their *combinatorial embeddings*. Further, we give a detailed introduction to BC-trees (Section 2.3) and SPQR-trees (Section 2.4), which are the two central data structures of the algorithms presented in the following chapters. In the last section, we take a short look at the complexity of algorithmic problems and define the terms concerning $\mathcal{NP}$*-completeness*.

2.1 Graph Theory

The following definitions are based on Diestel [9].

Definition 2.1 (**Graph**). *A graph $G = (V, E)$ is a pair, consisting of the finite set V and the finite multiset E. The elements of V are the vertices (or nodes), the elements of E are its edges. An edge e is a pair of distinct vertices $v, w \in V$, denoted by $e = (v, w)$.*

Notice that the above definition prohibits *self-loops*, i.e. there is no vertex related to itself. A graph is considered *simple*, if and only if the edge set E is not a multiset, hence every edge is unique. Let $G = (V, E)$ be a graph. A graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is called a *subgraph* of G.

Definition 2.2 (**(Un)Directed Graph**). *A graph is directed if its edges are defined as an ordered pair, otherwise the graph is undirected.*

In this thesis, we will consider undirected graphs unless stated otherwise.

Let $e = (v, w)$ be an edge. We say e *connects* the two vertices v and w, which are called its *endpoints*. The vertices v and w are *adjacent* to each other and *incident* to e. On the other hand, e is *incident* to v and w. Two edges $e_1 \neq e_2$ are adjacent if they share a common endpoint.

The *degree* of a vertex v is its number of incident edges and is denoted by $\deg(v)$.

In a *complete graph* every pair of vertices is adjacent. If the vertex set of a graph can be divided into two disjoint subsets $V_1, V_2 \subset V$, with $V_1, V_2 \neq \emptyset$ and the property that no edge has both endpoints in one set, then the graph is called *bipartite*.

A *path* $P = (V, E)$ is a directed or undirected graph with distinct vertices $x_0, ..., x_k$ and edge set $E = \{e_0, ..., e_{k-1}\}$ such that e_i connects x_i and x_{i+1} ($i = 0, ..., k-1$). The *length* of a path is its number of edges. A path augmented by one edge between x_k and x_0 is a *cycle*.

We denote by $G - v$ and $G - X$ the graph that results from G by deleting a single vertex v or a subset of vertices X, respectively, including the incident edges. A graph G is *connected* if there exists a path between each pair of vertices. Instead of calling a graph 'not connected', it is called *disconnected*. The maximal connected subgraphs of G are called the *connected components* of G.

Definition 2.3 (**k-Connectivity**). *An undirected graph $G = (V, E)$ is called k-connected, for $k \in \mathbb{N}$, if $|V| > k$ and $G - X$ is still connected, for any subset $X \subset V$ with $|X| < k$.*

An equivalent definition of k-connectivity is, that there have to exist at least k different paths between any pair of vertices v, w that are pairwise vertex-disjoint except for their endpoints.

For the cases $k = 2$ and $k = 3$, a graph is called *biconnected* and *triconnected*, respectively. The biconnected subgraphs are called the *biconnected components* or short *blocks* (compare Section 2.3).

We say that a subset of vertices $X \subset V$ separates a connected graph G, if $G - X$ is disconnected. A separating set of size one is a *cutvertex* and of size two a *separation pair*. If a separation pair $\{v, w\}$ is adjacent then the corresponding edge (v, w) is a *bridge*. Thus, the bridges of a graph are those edges that do not lie on any cycle. The *bridge-connected components* of a graph are the connected components that arise after deleting all bridges. A graph is *bridge-connected* if it does not contain any bridge.

Definition 2.4 (**Forest, Tree**). *An acyclic graph, one not containing any cycle, is called a forest. A tree T is an acyclic and connected graph.*

We will refer to the vertices of trees as nodes.

The nodes of degree one in a tree are its *leaves*, whereas all other nodes are *inner nodes*. A tree can have a designated node, the *root*. We then speak of a *rooted tree* and every edge of the tree has an orientation, that is they are all directed towards or away from the root. A node v is the *parent* of node w if they are adjacent and v lies on the unique path from w to the root. Then w is a *child* of v.

The *height* of a rooted tree is the length of the longest path from the root to a leaf.

2.2 Planar Graphs

Graphs are basically utilized to represent the relations between certain objects. By drawing graphs on the plane, they get a graphical representation and a certain topological structure. Furthermore, *drawings* of graphs lead to an important type of graphs, namely the *planar graphs*.

The following definitions are based on [43].

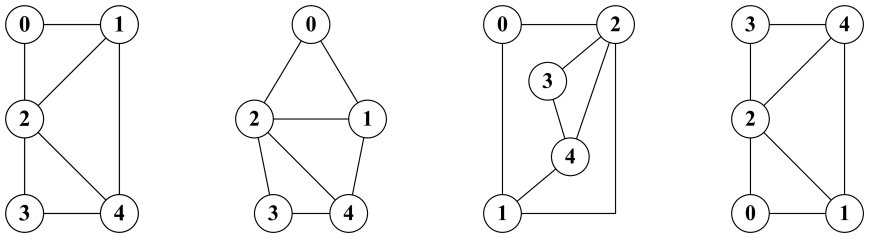


Figure 2.1: Four different drawings of the same planar graph.

Definition 2.5 (**Drawing**). *A drawing* Γ *of a graph* $G = (V, E)$ *is a function that maps each vertex* $v \in V$ *of* G *to a point* $\Gamma(v)$ *in the plane and each edge* $e = (v, w) \in E$ *to a curve* $\Gamma(e)$ *with endpoints* $\Gamma(v)$ *and* $\Gamma(w)$*. A drawing is planar if no two curves* $\Gamma(e_1)$ *and* $\Gamma(e_2)$ *cross each other, except in their endpoints.*

Definition 2.6 (**Planar Graph**). *A graph* G *is called planar if there exists a planar drawing of* G*.*

A planar drawing of a graph partitions the plane into connected regions, bounded by the curves of the edges. These regions are called *faces*. The *boundary* of a face f consists of the vertices and edges whose image forms the boundary of f in the drawing. The unbounded face enclosing the graph is called the *external face*. An *outerplanar graph* is a planar graph where all vertices lie on the external face.

A drawing Γ is the description of a graphical representation of a graph. Two drawings of the same graph can look very different but have the same topological structure. To define the topological equivalence, we introduce two equivalence classes. We say that two planar drawings Γ_1 and Γ_2 of a graph $G = (V, E)$ are *weakly equivalent*, if for every vertex $v \in V$, the circular order of the incident edges around v in clockwise order is the same in Γ_1 and Γ_2. Furthermore, two planar drawings are *strongly equivalent*, if they are weakly equivalent and the external face is the same in both drawings.

Definition 2.7 (**Combinatorial Embedding**). *The combinatorial embeddings of a planar graph are the equivalence classes of its drawings with respect to the weak equivalence relation.*

Definition 2.8 (**Planar Embedding**). *The planar embeddings of a planar graph are the equivalence classes of its drawings with respect to the strong equivalence relation.*

Hence, the cyclic order of incident edges of every vertex defines a combinatorial embedding. An additionally given external face defines a planar embedding of the graph.

We will use the term embedding both for combinatorial and planar embedding, if it is clear from context which is the desired one.

Figure 2.1 illustrates a graph with four different drawings. The drawings in (a) and (b) are strongly equivalent, since only the positions of the vertices are slightly

modified, hence they have the same planar embedding. Drawing (c) looks different to the first two ones, but is weakly equivalent to them, because the cyclic order of the incident edges is the same. Only the external face differs, since in (a) and (b) the face bounded by vertices $\{0, 1, 4, 3, 2\}$ is the external one, whereas in (c) the vertex set $\{0, 2, 1\}$ forms the external face. Drawing (d) looks quite similar to (a), but the cyclic order of the incident edges around the vertices is different. In fact, the sequences of incident edges of each vertex are mirrored. Therefore, the drawings are not weakly equivalent.

Deciding whether a graph is planar or not is a well-studied problem. The first linear-time algorithm is due to Hopcroft and Tarjan [25]. In [5], Boyer and Myrvold presented a simpler approach based on Depth-First-Search, that also returns an embedding of the graph, if planar.

One sufficient criteria for planarity is the well-known theorem by Kuratowski [35]. Let K_n denote the complete graph on n vertices and $K_{n,m}$ the complete bipartite graph $G = (V_1 \dot\cup V_2, E)$ with $|V_1| = n$ and $|V_2| = m$. A *subdivision* of a graph $G = (V, E)$ emerges from a series of *split-operations* on the edges of E. A *split-operation* on edge $e = (v, w)$ replaces e by two edges (v, w') and (w', w). Subdivisions of K_5 and $K_{3,3}$ are called *Kuratowski subdivisions.*

Theorem 2.1. *A graph is planar if and only if it does not contain a subgraph that is a K_5– or $K_{3,3}$–subdivision.*

In [6], Chimani, Mutzel, and Schmidt presented a linear time algorithm that extracts the Kuratowski subdivisions, based on the algorithm by Boyer and Myrvold [5].

An important fact on planar graphs is its bounded size in the number of vertices. The upper bound can be concluded from Euler's formula for polyhedra.

Theorem 2.2. *Let $G = (V, E)$ be a planar connected graph with $n := |V| > 1$ and $m := |E|$, Π a planar embedding of G and f the number of faces. Then the following equation is true:*

$$n - m + f = 2$$

Corollary 2.3. *A planar graph $G = (V, E)$ with $|V| \geq 3$ has at most $3|V| - 6$ edges, hence the number of edges is linearly bounded by the number of vertices.*

2.3 BC-Tree

A *block-cutvertex-tree*, or short *BC-tree*, represents the biconnected structure of a graph.

Definition 2.9 (BC-Tree). *For a connected graph $G = (V, E)$ the corresponding BC-tree $bc(G) = (V_{bc}, E_{bc})$ is defined as follows:*

- *The node set V_{bc} is the union of two disjoint sets V_b and V_c.*
- *Each block in G is represented by one b-node in V_b and each cutvertex by one c-node in V_c.*

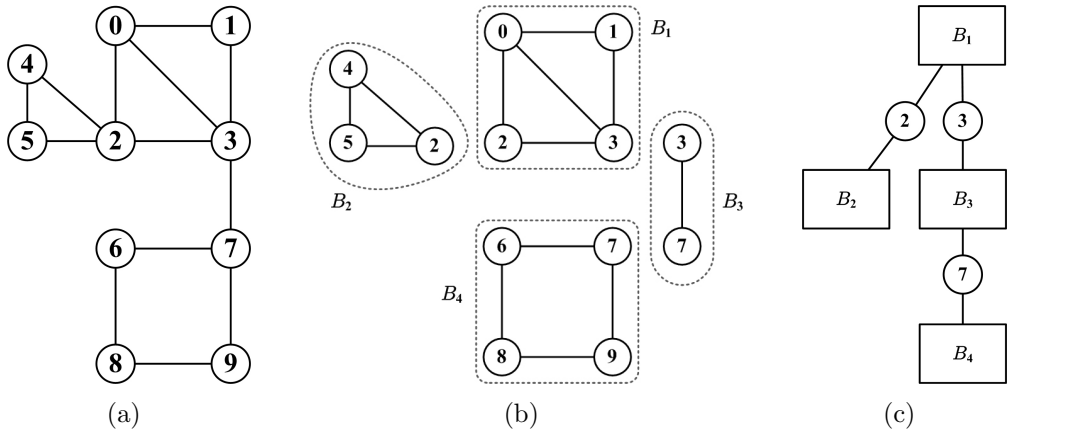


Figure 2.2: (a) A connected graph, (b) its blocks B_1, B_2, B_3, B_4, and (c) the corresponding BC-tree rooted at the b-node B_1; the cutvertices are 2,3, and 7.

- *A b-node $b \in V_b$ and a c-node $c \in V_c$ are adjacent, if and only if the cutvertex represented by c belongs to the block represented by b.*

A BC-tree has a designated root that is always an inner node, except for the trivial case where the graph is already biconnected. Figure 2.2 presents an example of a graph and its related BC-tree. In this thesis, we will represent b-nodes by rectangles and c-nodes by circles.

Since cutvertices belong to at least two blocks, c-nodes have degree ≥ 2. Therefore, a c-node cannot be a leaf and hence, all leaves in a BC-tree are b-nodes.

Each induced subgraph of a leaf-block always consists of at least one vertex that is no cutvertex. We refer to these vertices of the graph as *simple vertices*.

Let $G = (V, E)$ be a connected graph and $bc(G) = (V_{bc}, E_{bc})$ its related BC-tree. Obviously, the number of c- and b-nodes in V_{bc} is $\mathcal{O}(|V|)$. The number of edges in a tree with n nodes is always $n - 1$. Therefore the total size of a BC-tree is linear in the number of vertices of the underlying graph.

A BC-tree can be computed by using a well-known Depth-First-Search approach, see [39]. The running time is also linear in the input size of the graph. We gather the last two results in the following corollary:

Corollary 2.4. *A BC-tree of a graph $G = (V, E)$ can be computed in $\mathcal{O}(|V| + |E|)$ time and uses $\mathcal{O}(|V|)$ space.*

2.4 SPQR-Tree

The SPQR-tree data structure represents the decomposition of a biconnected graph into its triconnected components. The data structure was first introduced by Di Battista and Tamassia [2]. Their definition of the decomposition tree is based on the ideas of Bienstock and Monma [4] and is related to the classical decomposition of biconnected graphs into triconnected components by Tutte [40] and Hopcroft and Tarjan [24]. In [2], SPQR-trees were originally used for incremental planarity

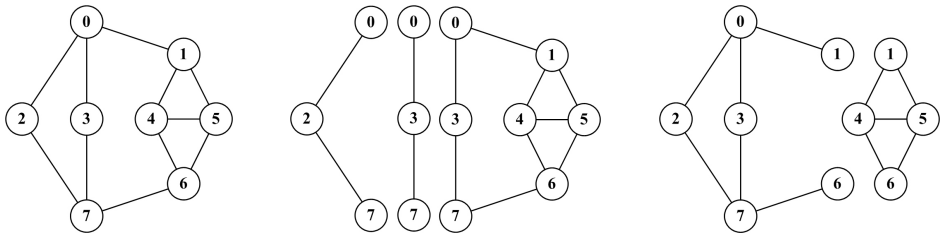


Figure 2.3: (a) Biconnected graph, (b) the split components with respect to split pair $\{0, 7\}$, and (c) split pair $\{1, 6\}$.

testing. Since then, SPQR-trees were utilized for solving many problems in graph theory.

Later on, we will show how SPQR-trees can be used to enumerate all combinatorial embeddings of a biconnected graph (Section 2.4.3). Unfortunately the number of embeddings is exponential in general. However, there are several problems, mostly concerning planar graphs, that can be solved in linear time using the SPQR data structure. An example is the problem of inserting an edge into a planar graph with the minimum number of crossings, where all crossings involve the new edge. Gutwenger, Mutzel, and Weiskircher showed in [23] that this problem can be solved in linear time.

2.4.1 Definition

Our definition of the SPQR-tree data structure is adopted from the one of Di Battista and Tamassia in [3]. A slightly modified, detailed introduction can be found in [43].

Before we can define the SPQR-tree we have to introduce a few more terms. Let G be a biconnected graph. A *split pair* of G is either a separation pair or a pair of adjacent vertices. A *split component* of a split pair $\{v, w\}$ is either an edge (v, w) or a maximal subgraph G' of G such that $\{v, w\}$ is not a split pair of G'. Let {s,t} be a split pair of G. A *maximal split pair* $\{v, w\}$ of G with respect to $\{s, t\}$ is such that, for any other split pair $\{v', w'\}$, vertices s, t, v, and w are in the same split component; for an example see Figure 2.3.

Let e be an edge of G between vertices s and t, called the *reference edge*. The SPQR-tree $\mathcal{T}$ of G with respect to e describes a recursive decomposition of G induced by its split pairs.

Definition 2.10 (SPQR-Tree). *Tree $\mathcal{T}$ is a rooted ordered tree whose nodes are of four types: S, P, Q, and R. Each node μ of $\mathcal{T}$ has an associated biconnected multigraph called the skeleton of μ and denoted by skeleton(μ). Tree $\mathcal{T}$ is recursively defined as follows:*

***Trivial Case**: If G consists of exactly two parallel edges between s and t, then $\mathcal{T}$ consists of a single Q-node whose skeleton is G itself.*

***Parallel Case**: If the split pair $\{s, t\}$ has at least three split components $G_1, ..., G_k$*

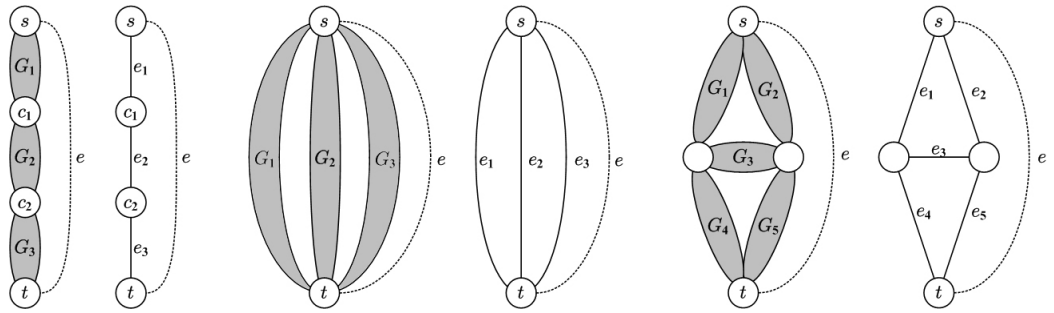


Figure 2.4: Pertinent graphs on the left and the related skeletons of (a) S-, (b) P-, and (c) R-nodes on the right with reference edge e.

($k \geq 3$), the root of $\mathcal{T}$ is a P-node μ. The graph skeleton(μ) consists of k parallel edges between s and t, denoted $e_1, ..., e_k$, with $e_1 = e$.

***Series Case**: Otherwise, the split pair $\{s, t\}$ has exactly two split components, one of them is the reference edge e, and we denote the other split component by G'. If G' has cutvertices $c_1, ..., c_{k-1}$ ($k \geq 2$) that partition G into its blocks $G_1, ..., G_k$, in this order from s to t, the root of $\mathcal{T}$ is an S-node μ. Graph skeleton(μ) is the cycle $e_0, e_1, ..., e_k$, where $e_0 = e, c_0 = s, c_k = t$ and e_i connects c_{i-1} with c_i ($i = 1, ..., k$).*

***Rigid Case**: If none of the above cases applies, let $\{s_1, t_1\}$,...,$\{s_k, t_k\}$ be the maximal split pairs of G with respect to $\{s, t\}$ ($k \geq 1$), and, for $i = 1, ..., k$, let G_i be the union of all the split components of $\{s_i, t_i\}$ but the one containing the reference edge e. The root of $\mathcal{T}$ is an R-node μ. Graph skeleton(μ) is obtained from G by replacing each subgraph G_i with the edge e_i between s_i and t_i.*

Except for the trivial case, μ has children $\mu_1, ..., \mu_k$ in this order, such that μ_i is the root of the SPQR-tree of the (multi) graph $G_i \cup e_i$ with respect to reference edge e_i ($i = 1, ..., k$).

The tree so obtained has a Q-node associated with each edge of G, except the reference edge e. We complete the SPQR-tree by adding another Q-node, representing the reference edge e, and making it the parent of μ so that it becomes the root.

We need to introduce some further notations. Let μ be an S-, P-, or R-node with the children $\mu_1, ..., \mu_k$ and related reference edges $e_i = (s_i, t_i)$, such that μ_i is the root of the SPQR-tree of $G_i \cup e_i$. The endpoints of edge e_i are called the *poles* of node μ_i. Edge e_i is said to be the *virtual edge* of node μ_i in the skeleton of μ and of node μ in the skeleton of μ_i. We call node μ the *pertinent node* of e_i in skeleton of μ_i, and μ_i the *pertinent node* of e_i in the skeleton of μ. The virtual edge of μ in the skeleton of μ_i is called the reference edge of μ_i.

Let e_ν be an edge in *skeleton*(μ) and ν the pertinent node of e_ν. Deleting edge (μ, ν) in $\mathcal{T}$ splits $\mathcal{T}$ into two connected components. Let $\mathcal{T}_\nu$ be the connected component containing ν. The *expansion graph* of e_ν, denoted with *expansion*(e_ν), is the graph induced by the edges that are represented by the Q-nodes in $\mathcal{T}_\nu$. We

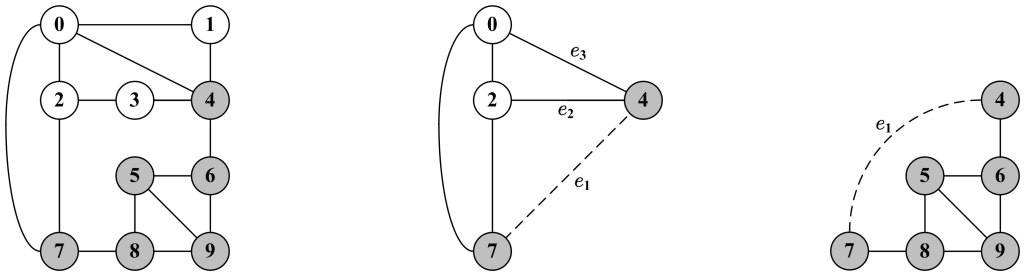


Figure 2.5: (a) Biconnected graph, (b) the skeleton μ of an R-node in the SPQR-tree with virtual edges e_1, e_2, and e_3, and (c) the graph $expansion^+(e_1)$.

further introduce the notation $expansion^+(e_\nu)$ for the graph $expansion(e_\nu) \cup e_\nu$. An example is given in Figure 2.5.

The *pertinent graph* of a tree node μ results from replacing all edges in $skeleton(\mu)$ by its expansion graph except for the reference edge of μ and is denoted with $pertinent(\mu)$. Hence, if e_ν is a skeleton edge and ν its pertinent node, then $expansion^+(e_\nu)$ equals $pertinent(\nu)$. For a vertex v in G, we call a node in $\mathcal{T}$ whose skeleton contains v an *allocation node* of v. Figure 2.4 gives examples for pertinent graphs and their related skeletons for the different node types.

2.4.2 Example Decomposition

In this section we perform an example decomposition of the biconnected graph G shown in Figure 2.6 (a). The resulting SPQR-tree $\mathcal{T}$, with the associated skeleton graphs of each tree-node, is presented in Figure 2.6 (b).

As reference edge for the first decomposition step we select edge $e = (0, 1)$. The graph G has two split components with respect to e: C_1 and C_2. Let C_1 be the split component that only consists of edge $(0, 1)$ and let C_2 be $G - (0,1)$. Because C_2 is biconnected, the rigid case holds. The maximal split pairs of G are $(0, 8)$, $(1, 8)$ and $(2, 8)$. Therefore the skeleton graph of the R-node consists of the vertex set $\{0, 1, 2, 8\}$ and edges $(0, 1)$, $(0, 2)$, $(0, 8)$, $(1, 8)$, and $(2, 8)$. The current R-node becomes the root of the SPQR-tree and the decomposition proceeds recursively with the subgraphs induced by the split pairs which are augmented with their reference edges. Let G_1 be the subgraph induced by $(0, 8)$, G_2 by $(1, 8)$ and G_3 by $(2, 8)$. Additionally, the three Q-nodes associated with the edges $(0, 1)$, $(0, 2)$, and $(1, 2)$ are created and inserted into the current SPQR-tree. For simplicity, the Q-nodes are omitted in Figure 2.6.

We continue the decomposition with G_1 and reference edge $(0, 8)$. In this case, the graph consists of exactly two split components, again one being the reference edge. This time, the other split component is not biconnected and has two cutvertices, namely 3 and 7. Hence, an S-node and two Q-nodes are created. The related skeleton of the S-node contains the cycle $\{(0, 3), (3, 7), (7, 8), (0, 8)\}$ and the two trivial cases occur for the edges $(0, 3)$ and $(7, 8)$. Furthermore there is one recursive step for the subgraph induced by the two cutvertices.

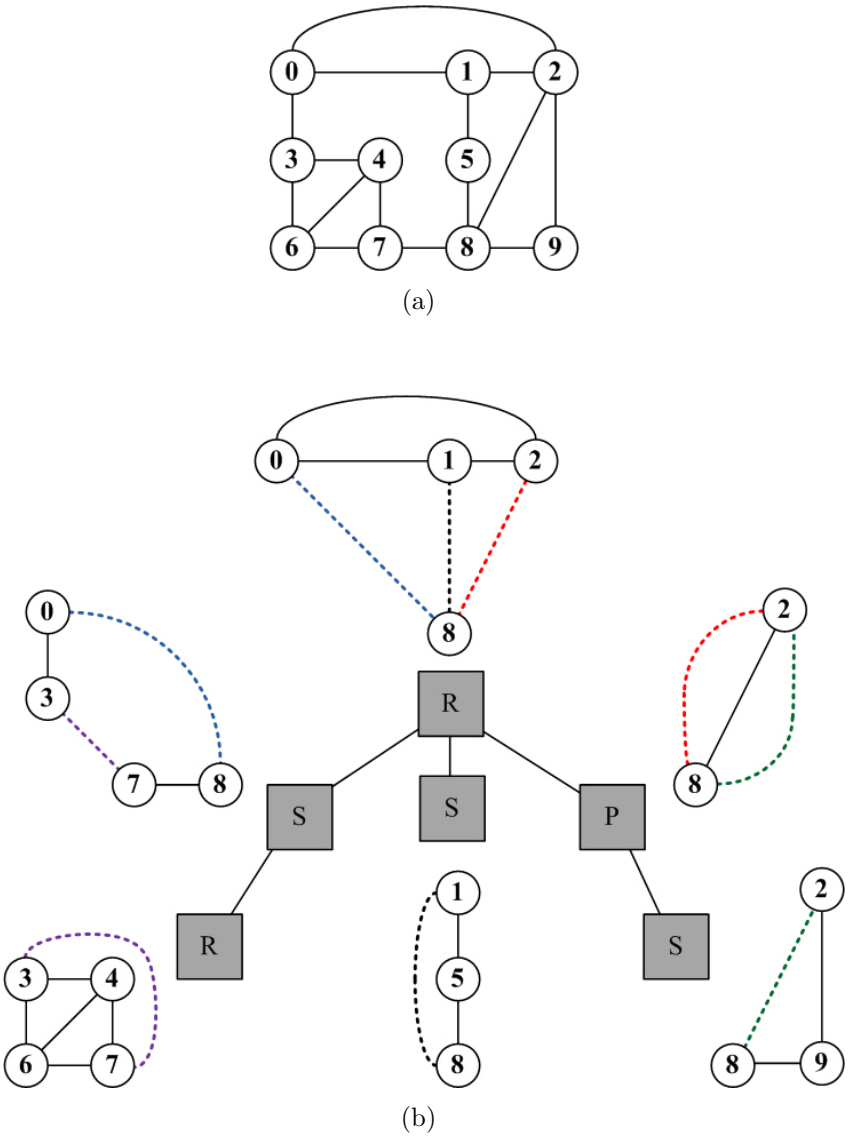


Figure 2.6: (a) Biconnected graph and (b) the related SPQR-tree rooted at the R-node. The Q-nodes are omitted for simplicity.

This subgraph, shown in the bottom left corner of Figure 2.6 (b), is triconnected and therefore has no more split pairs. The SPQR-tree is extended by one R-node and five Q-nodes. The recursive construction stops and jumps back to the unfinished decomposition in the root.

There the two subgraphs G_2 and G_3 still have to be proceeded. The first one consists of the vertices $1, 5, 8$, edges $(1, 5)$, $(5, 8)$, and reference edge $(1, 8)$. They describe a cycle and therefore one S-node is created with two adjacent nodes. These nodes are both Q-nodes because their blocks each consist of one single edge, namely $(1, 5)$ and $(5, 8)$, respectively.

The subgraph G_3 is induced by the split pair $(2, 8)$. It consists of three split components, the first is reference edge $(2, 8)$, the second one edge $(2, 8)$, and the third one is the chain $\{2, 9, 8\}$. Therefore the parallel case occurs and the SPQR-tree is extended by one P-node and one Q-node associated with edge $(2, 8)$. The skeleton graph simply consists of the three parallel edges between the poles 2 and 8.

The final decomposition step affects the subgraph of the cycle containing the reference edge $(2, 8)$ and edges $(2, 9)$ and $(9, 8)$. Again the series case occurs and the SPQR-tree is augmented by one S-node and two Q-nodes.

2.4.3 Properties of SPQR-Trees

The main feature of SPQR-trees is that they can be used to represent every combinatorial embedding of the related biconnected planar graph. Furthermore, the data structure provides a way to enumerate over all embeddings. The following theorem was taken from [43].

Theorem 2.5. *Let G be a biconnected planar graph and let $\mathcal{T}$ be its SPQR-tree. A combinatorial embedding Π of G uniquely defines a combinatorial embedding of the skeleton of each node in $\mathcal{T}$. On the other hand, fixing the combinatorial embedding of the skeleton of each node in $\mathcal{T}$ uniquely defines a combinatorial embedding of G.*

Consider a planar embedding of G. By replacing subgraphs in G with single edges each skeleton graph of $\mathcal{T}$ can be created. Hence, the embedding of G defines the embedding of each skeleton. On the other hand, Graph G can be obtained by merging the skeletons of $\mathcal{T}$ in the case they share the same reference edge until only one skeleton is left. This skeleton is then isomorphic to the original graph, that is there exists a bijective mapping of vertices from G to this skeleton such that the topology is the same. Therefore, an embedding of G can be constructed from the combinatorial embeddings of the skeletons.

For enumerating over all combinatorial embeddings, we have to choose embeddings for the skeletons of the nodes in the SPQR-tree. The skeletons of S- and Q-nodes are simple cycles, so they have only one embedding. P-nodes represent a set of multi edges between the two poles. Therefore the number of embeddings is the number of permutations of these edges, except the reference edge. For k edges this number equals $(k-1)!$. An R-node-skeleton, which is a triconnected graph, has exactly two embeddings. Thus, if the SPQR-tree of G has r R-nodes and k P-nodes $P_1, ..., P_k$, where the skeleton of P_i has p_i multi edges, then the total number of

combinatorial embeddings of G is

$$2^r \prod_{i=1}^{k} (p_i - 1)!.$$

Hence, the number of combinatorial embeddings of a graph can be exponential with respect to its number of vertices.

Finally, we take a look at the total size of the generated SPQR-tree including its skeleton graphs and the time complexity for computing a related SPQR-tree.

Let $G = (V, E)$ be a biconnected graph.

Lemma 2.6 ([3]). *The SPQR-tree $\mathcal{T}$ of G has $|E|$ Q-nodes and $\mathcal{O}(|V|)$ S-, P-, and R-nodes. Also, the total number of vertices of the skeleton graphs of $\mathcal{T}$ is $\mathcal{O}(|V|)$.*

The algorithm for constructing the SPQR-tree relies on the ideas of the algorithm for dividing a graph into its triconnected components by Hopcroft and Tarjan [24]. In [22], Gutwenger and Mutzel corrected some mistakes in this approach and presented the first linear time implementation[1].

Corollary 2.7. *The SPQR-tree of a biconnected graph $G = (V, E)$ can be computed in time $\mathcal{O}(|V| + |E|)$.*

Since the number of edges in a planar graph is at most $3|V| - 6$, the SPQR-tree of a planar graph can be computed in time $\mathcal{O}(|V|)$.

2.5 $\mathcal{NP}$-Completeness

In this section we give a very short introduction to *algorithms*, *algorithmic problems*, and complexity theory. All definitions are based on the work of Wegener [42]. Other introductions to the terms around *$\mathcal{NP}$-Completeness* can be found in [8] or [16].

An *algorithmic problem* is defined by the set of feasible inputs and a function that maps each feasible input to a non-empty output. If the problem is to find a solution with maximum quality it is an *optimization problem*. In the case, the output can only have two feasible values "yes" and "no" (or "1" and "0"), and the problem is to decide which one is the correct answer to a given question, then the problem is a so-called *decision problem*.

An *algorithm* is a sequence of instructions that transform the input into the output. If each next step of an algorithm is always well-defined then the algorithm is called *deterministic*. By contrast, the next step of a *randomized algorithm* additionally depends on the evaluation of a random event with two outcomes, each with probability $\frac{1}{2}$.

Definition 2.11 ($\mathcal{P}$). *An algorithmic problem belongs to the complexity class $\mathcal{P}$ if there exists an algorithm that solves the problem in polynomial time in the input size.*

[1] A C++-implementation of SPQR-trees can be found in the *OGDF (Open Graph Drawing Framework)*, a self-contained C++ class library for the automatic layout of diagrams. For further details visit the homepage on `www.ogdf.net`.

A *nondeterministic algorithm* also has the possibility to choose between two actions in each step, but there is no rule for the selection of the next action.

Definition 2.12 ($\mathcal{NP}$)**.** *A decision problem belongs to the complexity class $\mathcal{NP}$ (nondeterministic polynomial time), if there exists a nondeterministic algorithm with polynomial running time in the input size that*

1. *has at least one possible sequence of instructions that leads to the acceptance of a yes-instance and*

2. *that rejects every no-instance.*

Nondeterministic algorithms are theoretically important because they lead to the complexity class $\mathcal{NP}$. However, they are considered to be practically not feasible because they are considered being able to "guess" the correct sequence of instructions that lead to the acceptance of a yes-instance.

These are the two complexity classes involved in one of the most important open mathematical problems, namely the question whether $\mathcal{NP} = \mathcal{P}$ or $\mathcal{NP} \neq \mathcal{P}$.

Definition 2.13 (**polynomial-time reducible**)**.** *A decision problem A^{dec} is polynomial-time reducible to a decision problem B^{dec} if there exists a function f that maps each instance $\mathcal{I}$ of A^{dec} to an instance $f(\mathcal{I})$ of B^{dec} such that $\mathcal{I}$ is a yes-instance for A^{dec} if and only if $f(\mathcal{I})$ is a yes-instance for B^{dec}. Furthermore, the function f has to be computable in polynomial time.*

If a problem A is polynomial-time reducible to another problem B, then this is a statement about the complexity of problem B. Since problem A can be solved by one call of an algorithm for problem B, the second problem is not easier to solve than the first one.

Definition 2.14 ($\mathcal{NP}$-**hard**, $\mathcal{NP}$-**complete**)**.** *A decision problem A is $\mathcal{NP}$-hard, if every decision problem $B \in \mathcal{NP}$ is polynomial-time reducible to A. If an $\mathcal{NP}$-hard decision problem A also belongs to $\mathcal{NP}$ then it is $\mathcal{NP}$-complete.*

Mostly, the decision version of a problem is not harder than the optimization version. If the question of a decision problem is whether a solution exists with a value of at most or at least a parameter k, this can be solved by computing an optimum solution *opt* with the optimization algorithm, comparing the two values k and *opt*, and giving the correct answer.

Although the term $\mathcal{NP}$-hardness is only defined for decision problems, it is also used for optimization problems in the case its related decision problem is $\mathcal{NP}$-complete.

A very important result in the complexity theory is due to Cook [7], who discovered the first proof for the $\mathcal{NP}$-completeness of a problem, namely *SAT*. The *Satisfiable Problem* (*SAT*) is the problem of deciding whether a conjunction of disjunctions over a set of variables is satisfiable or not.

Theorem 2.8 (**Theorem of Cook**)**.** *SAT is $\mathcal{NP}$-complete.*

The first $\mathcal{NP}$-complete problem reduces the difficulty for the prove of the $\mathcal{NP}$-completeness of other problems. Since the polynomial-time reduction is transitive, it is sufficient to reduce a known $\mathcal{NP}$-complete problem A to a problem $B \in \mathcal{NP}$ to show the $\mathcal{NP}$-completeness of problem B.

Since many problems are known to be $\mathcal{NP}$-complete and no polynomial time algorithm is found for any of them, most experts belief that $\mathcal{NP} \neq \mathcal{P}$. If one of these problems can be solved in polynomial time, then $\mathcal{NP} = \mathcal{P}$ holds and there do exist polynomial time algorithms for all $\mathcal{NP}$-complete problems. If $\mathcal{NP} \neq \mathcal{P}$, no $\mathcal{NP}$-complete problem can be solved in polynomial time.

Hence, the proof of the $\mathcal{NP}$-completeness of a problem is a strong indicator that probably no polynomial time algorithm exists for this problem.

For $\mathcal{NP}$-hard optimization problems, optimum solutions can be computed in exponential time by iterating over all possible values of the variables. Another approach is to compute solutions in polynomial time, which are nearly optimal. These algorithms are called *approximation algorithms*.

Let Π be an optimization problem, A an algorithm for Π and let $OPT_\Pi(\mathcal{I})$ denote the optimum solution for an instance $\mathcal{I}$. We say that algorithm A has an *approximation ratio* of $\rho_A \geq 1$ if, for any input $\mathcal{I}$, the solution produced by the algorithm $A(\mathcal{I})$ is within a factor of ρ_A of the optimum solution $OPT_\Pi(\mathcal{I})$:

$$\max\left(\frac{A(\mathcal{I})}{OPT_\Pi(\mathcal{I})}, \frac{OPT_\Pi(\mathcal{I})}{A(\mathcal{I})}\right) \leq \rho_A$$

An algorithm with approximation ratio ρ is an *ρ-approximation algorithm*.

Chapter 3

(Planar) Augmentation Problems

After introducing the basic notations and data structures, we are now able to define the central augmentation problems studied in this thesis. In Section 3.1, we give a general survey and present the results for several augmentation problems. Some problems are known to be $\mathcal{NP}$-hard, whereas some other problems can be solved exactly in polynomial time. Further, we introduce the basic ideas for solving biconnectivity augmentation problems in Section 3.2 and determine the complexity of the *Planar Augmentation Problem* in Section 3.3. Afterwards, in Section 3.4, we consider two approximation algorithms with ratio 2 and $\frac{5}{3}$, respectively. However, we present a counter-example for the latter approach implying that its ratio is also only two. Finally, in the last section of this chapter, we discuss the subproblem of connecting a disconnected graph.

3.1 Problem Definitions

The problem of adding edges to a graph to satisfy a given connectivity condition under several constraints and the optimization of an objective function is called *Augmentation Problem*.

Although there are several augmentation problems concerning directed or even mixed graphs, e.g., [11, 13, 15, 19, 38], we focus on the cases where the input graphs are undirected.

The *General Augmentation Problem* is to add the minimum number of edges to an undirected graph such that the resulting graph is k-connected, for a fixed $k \in \mathbb{N}$. In [11], Eswaran and Tarjan gave a lower bound on the required number of edges for biconnectivity augmentation, c.f. Theorem 3.3, and proved that this bound is also sufficient. Hsu and Ramachandran [27] presented a linear time algorithm that achieves this bound, based on the ideas of Rosenthal and Goldner [37]. Due to Watanabe and Nakamura, the problem of triconnectivity augmentation can be solved in linear time, too, cf. [41].

For $k = 4$ only algorithms were known that work on already triconnected graphs, see [26], and augmentation of graphs to reach k-connectivity with $k \geq 5$ was for a long time an open problem. Recently, Jackson and Jordán [28] found a polynomial time algorithm for a fixed $k > 2$. This algorithm runs in time $\mathcal{O}(n^5) + \mathcal{O}(f(k)n^3)$,

where n is the size of the input graph and f is an exponential function.

The weighted version of the *General Augmentation Problem*, that is there are additionally edge-costs and the objective is to minimize the total costs of the inserted edges, is $\mathcal{NP}$-hard for all $k > 1$.

In this thesis we study unweighted biconnectivity augmentation problems with an additional requirement for planarity. The main problem was first introduced by Kant and Bodlaender in [31].

Definition 3.1 (PA)**.** *Let $G = (V, E)$ be a planar, connected, and undirected graph. The Planar Augmentation Problem (PA) is the problem of finding the smallest set of edges E' such that $G = (V, E \cup E')$ is planar and biconnected.*

In the same work, Kant and Bodlaender showed, that the *Planar Augmentation Problem* is $\mathcal{NP}$-hard, for the proof see Section 3.3. Furthermore, they presented a rather simple approximation algorithm with approximation ratio 2 and running time $\mathcal{O}(|V| \log |V|)$. We will present the general ideas of this algorithm in Section 3.4

They also considered a special case of PA with the precondition, that all cutvertices of the input graph belong to the same triconnected component. We refer to this problem as PA_{Tric}. By restricting PA to this types of graphs it becomes solvable in polynomial time, more precisely $\mathcal{O}(|V|^{2.5})$, compare [31] and Section 5.1.

Since the general case PA is $\mathcal{NP}$-hard, and special graphs which are valid for PA_{Tric} can be augmented in polynomial time, it is interesting to investigate another special case that seems to be more complex than PA_{Tric} but not as difficult as PA.

Definition 3.2 (PA_{Bic})**.** *The Planar Augmentation Problem for graphs with the constraint that all cutvertices belong to one biconnected component is called PA_{Bic}.*

PA_{Bic} is the central problem of this thesis and will be investigated in Chapter 5.

Another variation of the *Planar Augmentation Problem* arises when the embedding of the graph is fixed:

Definition 3.3 (PA_{Fix})**.** *Let $G = (V, E)$ be a planar, connected, and undirected graph and $\Pi(G)$ a combinatorial embedding of G. The Planar Augmentation Problem with fixed embedding (PA_{Fix}) is the problem of finding the smallest set of edges E' such that $G = (V, E \cup E')$ is planar and biconnected and $\Pi(G)$ is preserved.*

An exact algorithm for PA_{Fix} is presented in Chapter 4.

3.2 Basics

As described in Section 2.3, a BC-tree represents the biconnected structure of the underlying graph and therefore, it is an adequate data structure for finding solutions for the biconnectivity augmentation problems. Most of the known algorithms are iterative and repeat roughly three steps. They consider the BC-tree, insert a *profitable* edge into the graph, update the BC-tree, and continue until the graph is biconnected.

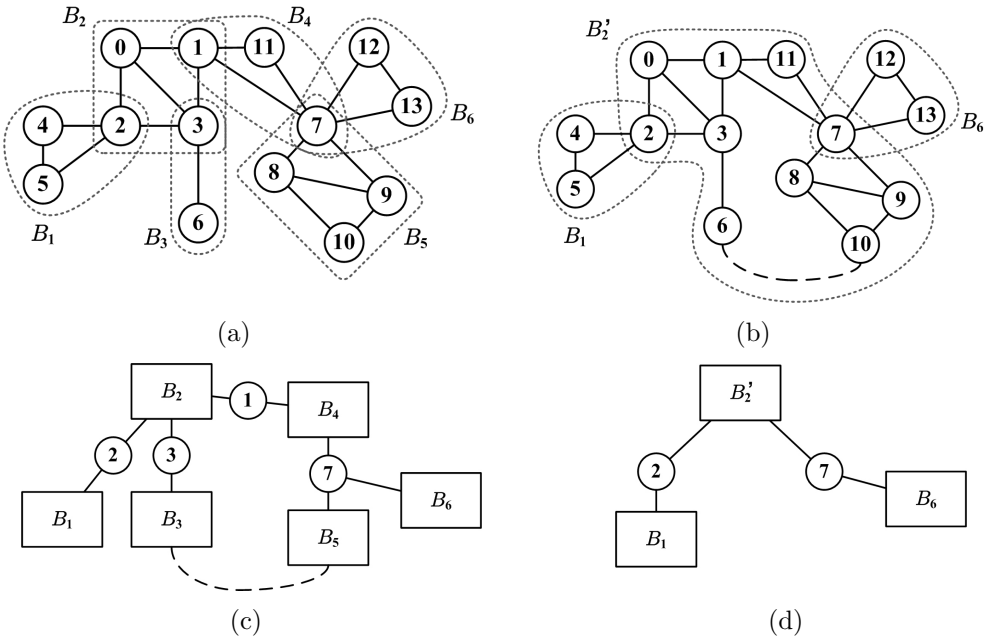


Figure 3.1: (a) Connected graph and its blocks, (b) the same graph with the updated biconnected components after adding edge $(6, 10)$, (c) the BC-tree of the original graph with the corresponding edge, and (d) the resulting BC-tree.

We will define later, what properties classify a new edge as a *profitable* one. First, we will discuss how the BC-tree is affected by the insertion of a new edge into the BC-tree and into the corresponding graph, respectively. A new edge between two vertices of the same biconnected component does not require any update. To decrease the number of blocks and therefore, reduce the size of the BC-tree, the new edge needs to induce a new cycle. The maximum cycles are obviously achieved by connecting two vertices that belong to biconnected components of leaves.

Furthermore, edges need to be added between simple vertices, since an edge with an cutvertex as endpoint makes only one incident block being part of the new cycle. Hence, the cycle would not be maximized. Although inner b-nodes can consist only of cutvertices, leaves always contain at least one simple vertex.

The following observation is adopted from [27].

Observation 3.1. *Let $G = (V, E)$ be a connected graph, $bc(G)$ its BC-tree, and b_1, b_2 two distinct leaves of $bc(G)$. Let C be the cycle formed by the unique path between b_1 and b_2 and the edge (b_1, b_2) in $bc(G)$. Let $G'/bc(G')$ be the graph/BC-tree obtained from G by adding an edge between simple vertices v and w which belong to the blocks represented by b_1 and b_2. The following relations hold between $bc(G)$ and $bc(G')$.*

1. *Nodes and edges of $bc(G)$ that are not on C remain the same in $bc(G')$.*
2. *All b-nodes on C are contracted to one b-node b'.*
3. *Any c-node on C with degree two is eliminated.*

4. *A c-node c^* on C with degree ≥ 3 remains in $bc(G')$ with edges incident to nodes not in the cycle. The c-node is also connected to the new block b'. It follows directly that the degree of c^* decreases by one.*

An example of a BC-tree and the effects of updates after inserting an edge is illustrated in Figure 3.1.

The following property specifies which nodes in the BC-tree are especially suitable for being connected.

Definition 3.4 (**Leaf-Connection Condition** [27])**.** *Let $G = (V, E)$ be a connected graph. Two distinct leaves b_1 and b_2 in $bc(G)$ satisfy the leaf-connection condition, short lcc, if and only if the path between b_1 and b_2 in $bc(G)$ contains either*

1. *two nodes of degree ≥ 3, or*
2. *one b-node of degree ≥ 4*

An edge (b_1, b_2) in $bc(G)$ is called *profitable*, if b_1 and b_2 satisfy the leaf-connection condition. The reason is given by the following lemma.

Lemma 3.2. *Let $G = (V, E)$ be a connected graph, $bc(G)$ its BC-tree and b_1 and b_2 two leaves in $bc(G)$ that satisfy the leaf-connection condition. Furthermore, let v and w be two simple vertices of the biconnected components of b_1 and b_2, respectively. The insertion of edge (v, w) in G decreases the number of leaves in $bc(G)$ by two.*

Proof. Assume that the first case of the leaf-connection condition holds, i.e. there are two nodes n_1 and n_2 with degree at least three on the path between b_1 and b_2. Let n_1' and n_2' be adjacent nodes of n_1 and n_2, respectively, that do not lie on the cycle. From Observation 3.1, it follows that b_1, b_2, and all other b-nodes on the cycle are contracted to one b-node b'. On the one hand, if n_i, $i \in \{1, 2\}$, is a b-node then n_i' has to be a c-node. The corresponding cutvertex remains a cutvertex for the new block b'. On the other hand, if n_i is a c-node then it is connected to a b-node n_i' which is not affected by the inserted edge. Therefore, n_i is still a cutvertex in the block of b'. Altogether, the new b-node has at least degree two and the number of leaves decreases by two.

Now, assume the second case holds with a b-node b^* with $\deg(b^*) \geq 4$. This b-node is obviously connected to at least four c-nodes. Two of them define the path to the newly connected leaves. The two other nodes are unchanged. Thus, b^* cannot become a leaf and the number of leaves also decreases by two.

□

As mentioned above, Eswaran and Tarjan found a lower bound for the *General Augmentation Problem* that is also sufficient for biconnecting a given graph, cf. [11].

Theorem 3.3 (**Lower bound on connected graphs**)**.** *Let $G = (V, E)$ be a connected graph. Let p be the number of leaves and d the maximum degree of a c-node in $bc(G)$. Then $\max\{d - 1, \lceil \frac{p}{2} \rceil\}$ edges are necessary and sufficient to make G biconnected.*

The idea of the theorem is as follows. Removing a c-node c^* splits the BC-tree in $\deg(c^*)$ subtrees and disconnects the graph into an equal number of connected components. To avoid this, at least $\deg(c^*)-1$ edges need to be inserted. Moreover, each new edge can obviously eliminate at most two leaves. Since a biconnected graph does not contain any leaves, at least $\lceil \frac{p}{2} \rceil$ new edges are necessary.

The original theorem gives a lower bound for disconnected graphs, too. We consider this theorem and the problem of connecting a disconnected planar graph in Section 3.5.

Definition 3.5 (**Massive c-node, balanced BC-tree**). *Let $bc(G)$ be a BC-tree and p the number of leaves in $bc(G)$. A c-node c^* is called massive if and only if $\deg(c^*) \geq \lceil \frac{p}{2} \rceil + 2$. A BC-tree is balanced if it does not contain any massive c-node. Otherwise, the BC-tree is unbalanced.*

It follows directly from the property of a massive c-node, that there can exist at most one in a BC-tree. Otherwise, a BC-tree with p leaves would have at least $\lceil \frac{p}{2} \rceil + 1 + \lceil \frac{p}{2} \rceil + 1$ leaves, what is a contradiction.

The existence of a massive c-node c^* implies that the lower bound is dominated by the term $d-1$, since $d-1 = \deg(c^*) - 1 \geq \lceil \frac{p}{2} \rceil + 1 > \lceil \frac{p}{2} \rceil$. In case of a balanced BC-tree, $d-1 = \deg(c) - 1 \leq \lceil \frac{p}{2} \rceil$ holds for the c-node c with maximum degree. Therefore, the exact number of required edges for biconnectivity augmentation depends on the existence of a massive c-node.

Corollary 3.4. *Let $G = (V, E)$ be a connected graph, $bc(G)$ the corresponding BC-tree and p the number of leaves in $bc(G)$.*

If the graph does not contain any massive c-node then $\lceil \frac{p}{2} \rceil$ edges are necessary and sufficient to biconnect the graph. Otherwise, if c^* is a massive c-node in G then this number equals $\deg(c^*) - 1$.

3.3 $\mathcal{NP}$-Completeness

Obviously, the bound of Theorem 3.3 is also a lower bound for the *Planar Augmentation Problem*. Unfortunately, $\max\{d-1, \lceil \frac{p}{2} \rceil\}$ is in general not sufficient, because planarity needs to be preserved. The exact number of required edges is probably not computable in polynomial time, due to the complexity of this problem.

Theorem 3.5 ([31]). *The Planar Augmentation Problem is $\mathcal{NP}$-hard.*

Proof. We consider the decision version of the optimization problem, denoted by PA^{dec}. The input is a planar, connected graph and a positive integer k. The question is whether the graph can be made biconnected by adding at most k edges. We show that PA^{dec} is $\mathcal{NP}$-complete.

The decision problem belongs to the complexity class $\mathcal{NP}$, since for any graph and an added edge set it is possible to verify in polynomial time whether the resulting graph is planar and biconnected, or not.

To prove $\mathcal{NP}$-completeness, we construct a polynomial-time reduction from a strong $\mathcal{NP}$-complete problem, namely *3-Partition*, to PA^{dec}.

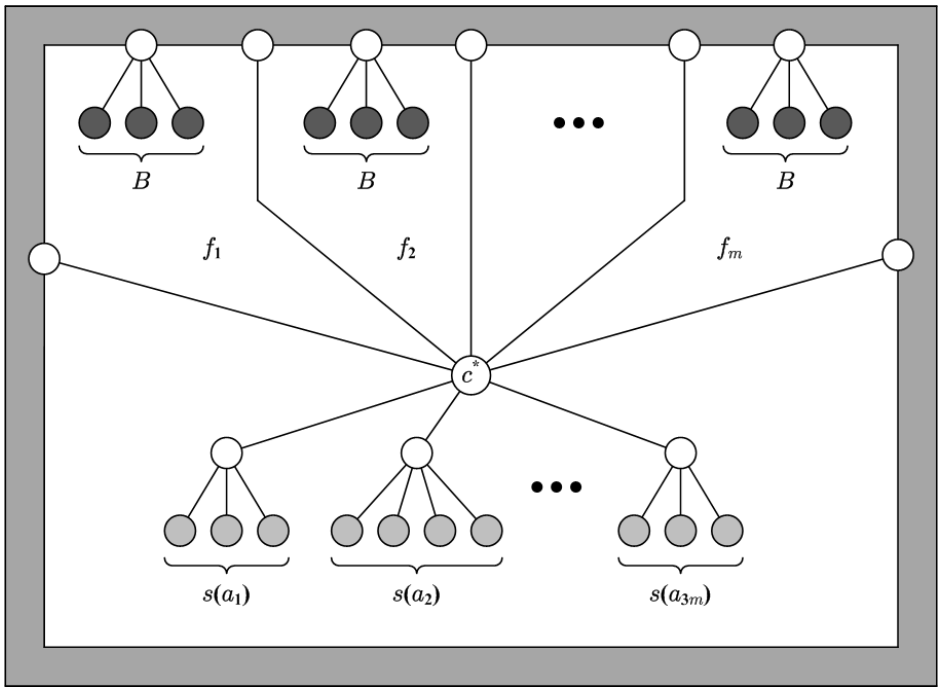


Figure 3.2: The constructed graph for the polynomial-time reduction from *3-Partition* to PA^{dec}.

Let $a_1, ..., a_{3m}$ be $3m$ elements and B a positive integer. Further, let $s(a_i) \in \mathbb{N}$ denote the size of each element a_i, $i = 1, ..., 3m$, with $\frac{B}{4} < s(a_i) < \frac{B}{2}$, and $\sum_{i=1}^{3m} s(a_i) = mB$. *3-Partition* is the problem of deciding whether the $3m$ elements can be partitioned into m disjoint subsets $S_j \subset \{a_1, ..., a_{3m}\}$ such that $\sum_{a_i \in S_j} s(a_i) = B$ holds for each $j = 1, ..., m$. From the definition of the element-sizes it follows directly that each subset needs to consist of exactly three elements.

For an instance of *3-Partition*, we construct a connected graph with $2mB$ leaves with the property, that it can be made biconnected with mB edges if and only if the original instance has a valid partition.

The graph has a triconnected structure with a central cutvertex c^* and m relevant faces. On the one hand, each element a_i is represented by a simple tree with $s(a_i)$ leaves. Each root is adjacent to the central vertex c^*. On the other hand, there are m identical subgraphs representing the m sets S_j. These subgraphs are also trees but they are located inside the m separated faces. Since they represent the sets S_j, each tree contains exactly B leaves. Figure 3.2 illustrates a scheme of the constructed graph. The shaded area represents a triconnected subgraph, the dark vertices at the top are the S_j-sets, the bright ones at the bottom correspond to the elements a_i.

Let $\mathcal{I}$ be an instance for *3-Partition* with the described variables. The constructed graph $G(\mathcal{I})$ consists of exactly $2mB$ leaves. It is important that the leaves corresponding to the S_j-sets are embedded into separate faces f_j and cannot be connected among each other. Moreover, all $s(a_i)$ leaves representing an element a_i

can only be embedded into one face f_j and therefore, they can only be connected to the leaves of one S_j-set.

If $\mathcal{I}$ is an acceptable instance for *3-Partition*, that is there exists a valid partition, we can construct a solution for the *Planar Augmentation Problem* by embedding each tree a_i into the face f_j and connecting the $a(s_i)$ leaves with the B-leaves, if a_i belongs to the set S_j in the partition. Since $\sum_{s_i \in S_j} s(a_i) = B$, for all $j = 1, ..., m$, each leaf can be connected to one leaf of another tree. Hence, mB edges are sufficient to biconnect $G(\mathcal{I})$.

Conversely, assume $G(\mathcal{I})$ has an augmenting set with exactly mB edges. The connection of two leaves from the same tree reduces the number of leaves only by one. Since the total number of leaves is $2mB$ each leaf has to be connected to one leaf of another tree. Therefore, the added edges reflect the assignment of the elements to the sets in the partition.

Altogether, $G(\mathcal{I})$ can be augmented with mB edges if and only if $\mathcal{I}$ can be partitioned into m subsets with the described constraints.

The graph can be constructed in polynomial time in the total size of the integers $s(a_i)$ and m. Since *3-Partition* is $\mathcal{NP}$-complete in the strong sense there does not exist a pseudo-polynomial algorithm, unless $\mathcal{NP} = \mathcal{P}$ holds. Hence, the decision problem of *PA* is $\mathcal{NP}$-complete and the theorem follows.

□

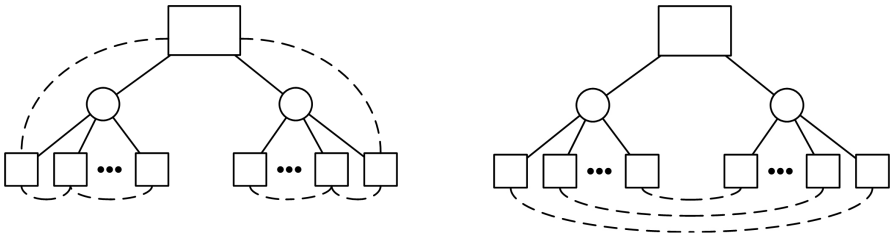
3.4 Approximation Algorithms

Like mentioned above, Kant and Bodlaender also described a 2-approximation algorithm for the *Planar Augmentation Problem* in [31]. Although there are several suggestions and ideas for new methods, two is the best known approximation ratio and the running time of this algorithm is only $\mathcal{O}(|V| \log |V|)$. The approach is straightforward and it is outlined in algorithm PA_2-Approximation.

Algorithm 1 PA_2-Approximation

Input: planar Graph $G = (V, E)$
Output: set of edges E' such that $G = (V, E \cup E')$ is planar and biconnected

```
1: compute the BC-tree bc(G)
2: while (number of b-nodes is ≥ 1) do
3:    c ← cutvertex that has only leaves as children in bc(G)
4:    if (c has more than one child) then
5:       connect all children of c such that a single leaf b arises
6:    else
7:       b ← single leaf attached to c
8:    end if
9:    connect b with the highest b-node in bc(G) with regard to planarity
10:   update bc(G)
11: end while
```



(a) (b)

Figure 3.3: (a) BC-tree with dashed edges representing the solution of PA_2-Approximation and (b) the optimum solution.

By Theorem 3.3, the size of an augmenting edge set is at least $\lceil \frac{p}{2} \rceil$. In PA_2-Approximation, every leaf gets at least one additional edge. Moreover, the algorithm may be forced to insert more edges due to planarity. But in an optimum solution planarity has to be preserved, too.

It seems, that the PA_2-Approximation-algorithm can be improved, because the connection of two leaves, which satisfy the leaf-connection condition, would decrease the number of leaves by two in each iteration. In this approach, the number of leaves decreases only by one with each inserted edge and the relation between leaves is completely disregarded. Figure 3.3 (a) illustrates a primitive BC-tree with $2k$ leaves and hence $2k$ inserted edges by PA_2-Approximation. An optimal augmenting edge set has cardinality k, cf. Figure 3.3 (b).

In [31], Kant and Bodlaender suggested another algorithm with ratio $\frac{3}{2}$. Fialko and Mutzel detected problems in this approach and presented a counter-example that yields approximation ratio two, compare [12]. In the same work, they introduced a new $\frac{5}{3}$-approximation algorithm. Unfortunately, this upper bound is incorrect. At the end of this section, we present a counter-example showing that this algorithm has only ratio 2.

Although the desired approximation ratio cannot be achieved, we will describe the algorithm and the corresponding ideas, because they will be adopted for the algorithms in Chapter 4 and Chapter 5. Furthermore, the algorithm has a good practical behaviour, since the computed solutions in the experiments are often optimum. In the other cases the solutions mostly contain only one or two more edges than in the optimum solution, cf. [12].

First we need to introduce some further notations and definitions, most of them are taken from [12].

As naming convention we call a leaf in the bc-tree a *pendant*, denote the set of pendants by P and its cardinality by p. A set $B \subseteq P$ is a *bundle of pendants* if the following three conditions hold:

1. Every pair of pendants $p_1, p_2 \in B$ can be connected without losing planarity.

2. Adding a new edge between two pendants $p_1, p_2 \in B$ leads to a new pendant.

3. The set B is maximal with respect to all sets satisfying the first two conditions.

Condition 2 and Lemma 3.2 guarantee that the path between two pendants of the same bundle contains exactly one bc-node with degree at least three. We call this bc-node the *parent* of the pendants belonging to the bundle and refer to the pendants as *children*. The parent of a bundle is either a b-node or a c-node. In the first case the b-node has exactly degree three, whereas in the latter case the c-node has degree at least three.

If the bundle contains exactly one pendant, say p_1, the parent c of p_1 is defined as the c-node satisfying the following three conditions.

1. All inner vertices on the path between p_1 and c have degree two.
2. Adding the edge (p_1, c) preserves a planar graph, where c is the only cutvertex to the new pendant.
3. The path from p_1 to c in the BC-tree is the longest among all c-nodes satisfying the two previous conditions.

Definition 3.6 (**(b/c)-Label**). *A bundle of pendants together with its parent is called label. If the parent is a c-node the label is also called c-label, otherwise it is called b-label.*

The size of a label l_1 is the number of pendants contained in the bundle and is denoted by $size(l_1)$.

The path from a pendant to the parent of the corresponding label contains only nodes with degree two. Therefore, these simple paths starting at parent v^* are called *v^*-chains*.

The mechanism of labels reflects the idea of profitable edges, since two pendants of different labels always satisfy the leaf-connection condition. This leads to the main ideas of the algorithm by Fialko and Mutzel: Select the label with maximum size, say l_1, find the largest label l_2 that is planar to l_1 and connect as many pendants as possible between l_1 and l_2. Two labels are planar if and only if their parents can be connected without losing planarity. In this case, $\min\{size(l_1), size(l_2)\}$ edges can be added between appropriate pendants of the two labels. An outline of the main procedure is presented by algorithm PA_Approximation.

If no planar label can be found (line 4), the algorithm works exactly like algorithm PA_2-Approximation by adding edges between the pendants of the same label and connecting the resulting pendant; cf. line 6.

The structure of a counter-example for the approximation quality of this algorithm is illustrated in Figure 3.4. The graph has two parallel structures which again consist of alternating serial and parallel structures. Furthermore, this graph has four triconnected subgraphs with two labels of size three attached on each side, and two subgraphs with only one label of size three. The triconnected subgraphs are represented by the shaded areas.

In general, there are $l+2$ triconnected subgraphs, l with two labels and two with one, each. Each of the l labels has size k, with $k, l \in \mathbb{N}$ and l being even. Figure 3.4 (a) illustrates a possible result of the first iteration. The algorithm might select one of the two central labels as l_1 and the other one as l_2. After adding k edges, no other

Algorithm 2 PA_Approximation

Input: planar Graph $G = (V, E)$

```
1: compute the BC-tree, pendants, and labels
2: while (number of labels ≥ 1) do
3:    l_1 ← label with maximum size
4:    l_2 ← largest label planar to l_1; nil if none is planar
5:    if (l_2 = nil) then
6:       connect all pendants of l_1 among each other and connect the arising
         ↪ pendant with the highest possible b-node in the bc-tree
7:    else
8:       connect all pendants of l_2 with pendants of l_1
9:    end if
10:   update the bc-tree and labels
11: end while
12: if (number of labels = 1) then
13:    connect the pendants of the remaining label
14: end if
```

two labels are planar to each other and each remaining pendant has to be connected by a single edge. Therefore, the total number of inserted edges is $(l + 1)k$. Figure 3.4 (b) presents an optimal solution. There, only $(\frac{l}{2} + 1)k$ edges are necessary to biconnect the graph. Hence, the ratio equals

$$\frac{(l+1)k}{(\frac{l}{2}+1)k} = \frac{l+1}{\frac{l}{2}+1}.$$

For sufficient large l, the ratio can be made to be as close to two as desired.

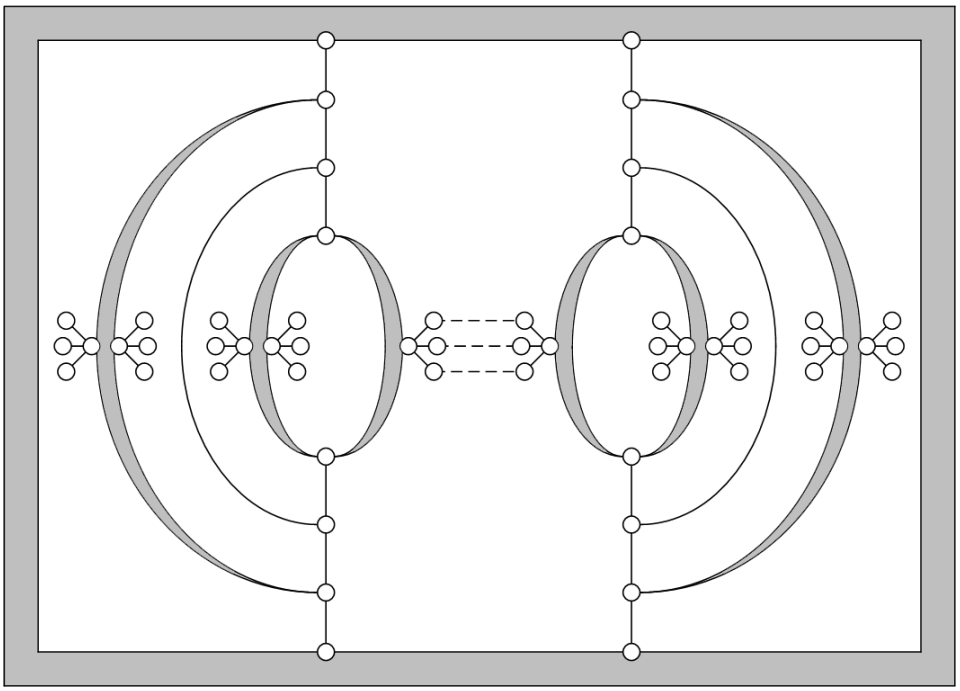
The major problem of this approach is the fact, that the connection of two pendants can fix the whole embedding in such a way, that all following profitable edges would destroy planarity.

3.5 Connectivity

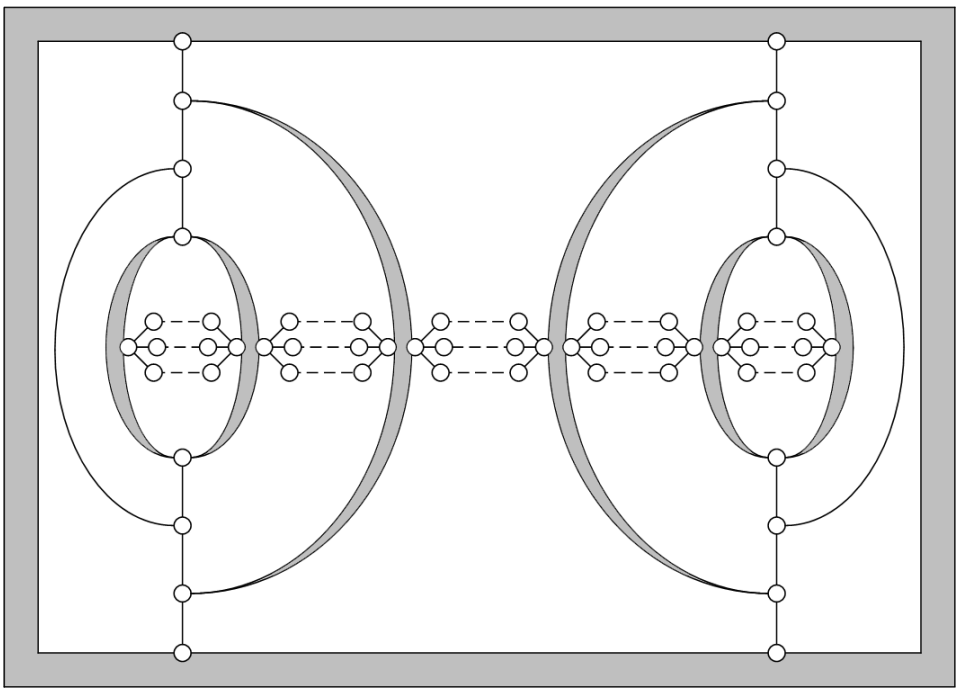
So far we considered only connected graphs. Like mentioned above, Theorem 3.3 is an adoption from the original lower bound by Eswaran and Tarjan [11]. In this section, we give the general lower bound and discuss how a disconnected graph can be made connected with a minimum number of edges. Furthermore, it is important that the added edges for connectivity do not affect the lower bound of the biconnectivity augmentation. It must be no difference if the graph is made biconnected at once or if the graph is made biconnected in two steps.

Theorem 3.6 (**General lower bound**). *Let G be an undirected graph with h connected components, let q be the number of isolated b-nodes, p the number of pendants and d the maximum degree of a c-node in $bc(G)$.*

Then $\max\{d + h - 2, \lceil \frac{p}{2} \rceil + q\}$ *edges are necessary and sufficient to biconnect G.*

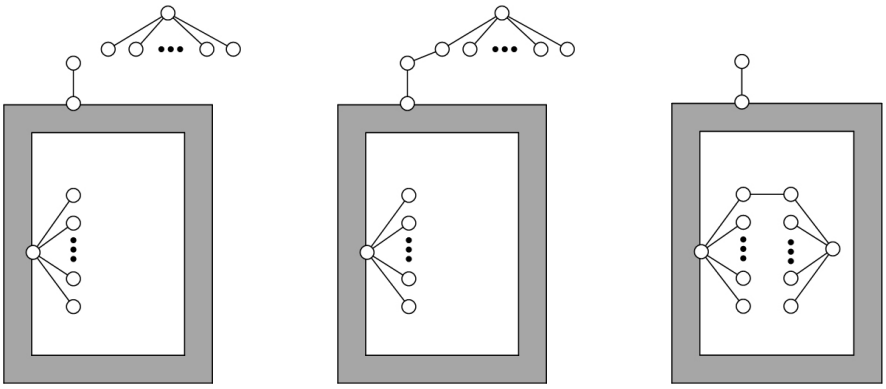


(a)



(b)

Figure 3.4: (a) A worst-case instance for the approximation algorithms with k inserted edges after the first iteration and (b) the optimal augmentation solution.



(a) (b) (c)

Figure 3.5: (a) Graph with two connected components and $2k+1$ pendants, (b) the situation after connecting the second connected component to the single pendant, and (c) the optimal connection.

Our definition of BC-trees also requires a connected graph. For disconnected graphs, the computation can be applied to each connected component separately. Obviously, the resulting set of BC-trees is a forest.

Let $G = (V, E)$ be a graph with h connected components $C_i, i = 1, ..., h$, and let $bc(C_i)$ be the corresponding BC-trees with a total number of p pendants and q isolated nodes. By inserting edges between components i and $i+1$ for $i = 1, ..., h-1$, G can be made connected with $h-1$ edges. The edges are added between simple vertices of the blocks corresponding to leaves or isolated nodes of each BC-tree. Two edges e_j and e_{j+1}, connecting components C_j with C_{j+1} and C_{j+1} with C_{j+2}, respectively, have an endpoint in common if and only if C_{j+1} is an isolated node.

Therefore, the resulting BC-tree has $p' := p + 2q - 2(h-1)$ pendants and the degrees of all inner nodes are unchanged. We need to ensure that the lower bound can still be achieved. After the connection, the resulting BC-tree has p' pendants and $h = 1$ and $q = 0$ holds. So far, we inserted exactly $h-1$ edges. Due to Theorem 3.6 (or 3.3), the complete biconnectivity augmentation then requires

$$\begin{aligned} &= h - 1 + \max\{d-1, \left\lceil \frac{p'}{2} \right\rceil\} \\ &= \max\{d+h-2, h-1+\left\lceil \frac{p+2q-2(h-1)}{2} \right\rceil\} \\ &= \max\{d+h-2, \left\lceil \frac{p}{2} \right\rceil + q\}. \end{aligned}$$

Hence, a disconnected graph can be made connected first without exceeding the lower bound of biconnectivity augmentation.

Unfortunately, this procedure cannot be applied to input-graphs of the *Planar Augmentation Problem*, if the approximation ratio shall be less than two. Figure 3.5 (a) illustrates a graph with two connected components, the above one with k and the lower one with $k+1$ pendants. The connection of the two wrong pendants

will lead to a solution with $2k$ edges (Figure 3.5 (b)), whereas the optimum is $k+2$ (Figure 3.5 (c)).

Because of the difficulty of making a planar and disconnected graph connected, without restricting the possible solutions for planar augmentation, we will consider all planar augmentation problems only for already connected graphs.

Chapter 4

Planar Augmentation with Fixed Embedding

In this chapter we present and analyze a new algorithm for the *Planar Augmentation Problem with fixed embedding*. First we describe the algorithm and its subprocedures (Section 4.1). After this, we prove that our approach is optimal (Section 4.2) and finally, we show that it has linear running time for all practical purposes and uses linear space (Section 4.3).

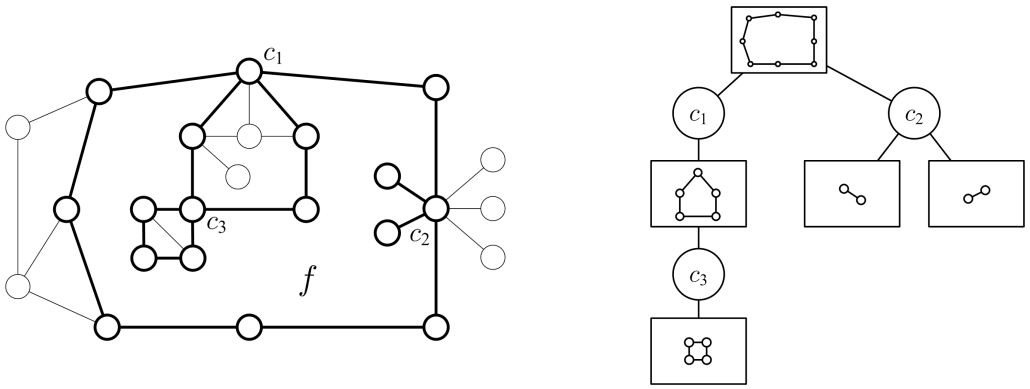
4.1 The Algorithm

The main procedure is outlined in PlanarAugmentationFix and the whole algorithm is split up into several smaller procedures named Update, HandlePendant, HandleRootDeg2, FindMatching and HandlePseudoLabel.

One basic idea of PlanarAugmentationFix is to consider each face separately. Since the embedding is fixed there are no edges allowed between inner vertices of different faces because that would destroy planarity. Therefore, each iteration of the algorithm (lines 2–21) biconnects a subgraph induced by another face. The induced subgraph of a face consists of the bounding vertices and edges of that face. Such a subgraph can be computed in linear time by a simple graph traversal that considers the boundary of the face in clockwise or counterclockwise direction. If the boundary is a simple circle, i.e. no vertex occurs more than once, then the induced subgraph is already biconnected. Figure 4.1 illustrates a planar graph with a face-induced subgraph and the corresponding BC-tree.

The augmentation of a face and its induced subgraph, respectively, is straightforward. First of all, the corresponding BC-tree is generated where it is important for later edge insertions that the BC-tree reflects the embedding of the graph. Then, the labels are computed by calls of procedure HandlePendant (line 8).

In every step the largest label is used to compute two appropriate pendants by calling procedure FindMatching. After every single augmentation, the BC-tree and all affected labels have to be updated (line 16). We continue until none or only one label is left. In the latter case all pendants of the remaining label need to be connected among each other (line 19).



(a) (b)

Figure 4.1: (a) A planar graph with a designated face f. The induced subgraph of f is emphasized by thick vertices and edges. (b) The corresponding BC-tree $bc(f)$.

Algorithm 3 PlanarAugmentationFix

Input: planar, connected graph G with fixed embedding $\Pi(G)$
Output: list of new edges E' such that $G = (V, E \cup E')$ is planar and biconnected

1: $E' \leftarrow \emptyset$
2: **for all** faces $f \in \Pi(G)$ **do**
3: construct the BC-tree $bc(f)$ induced by the face f, root it at ↪ a b-node with degree ≥ 2
4: **if** (number of pendants = 2) **then**
5: create a label with the two pendants
6: **else**
7: **for all** pendants p of $bc(f)$ **do**
8: HandlePendant(p)
9: **end for**
10: **end if**
11: **while** (number of Labels > 1) **do**
12: $l_1 \leftarrow$ label with maximum size
13: $(p_1, p_2) \leftarrow$ FindMatching(l_1)
14: $l_2 \leftarrow$ label of p_2
15: $E' \leftarrow E' \cup$ new edge between simple vertices of p_1 and p_2
16: Update()
17: **end while**
18: **if** (number of labels = 1) **then**
19: connect all p pendants of the remaining label with $p - 1$ new edges ↪ and insert them into E'; the edges are inserted between simple ↪ vertices of neighbouring pendants regarding the embedding
20: **end if**
21: **end for**
22: **return** E'

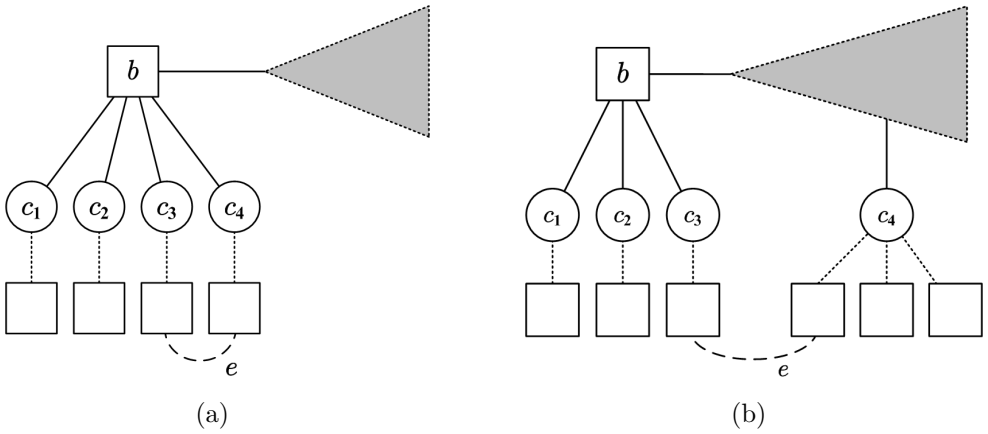


Figure 4.2: The two cases where a pseudo-label with parent b becomes a real b-label after inserting edge e.

The correct computation of the labels is an essential invariant. To provide the correct assignment of pendants to their labels and achieve the desired running time, we require an auxiliary type of labels, the *pseudo-labels*. A *pseudo-label* is a potential b-label. It consists of a b-node as parent and a set of c-labels with size one, whose parents are connected directly to the b-node.

Pseudo-labels are utilized during the computation of labels in HandlePendant. Further, they play an important role during the updates after each edge insertion. This is because a b-node with degree four or five, respectively, that lies on the insertion path can become a parent of a label. Moreover, a c-label with only one pendant, whose parent is not on the insertion path, can be affected indirectly and need to be merged with other labels into a b-label. We cannot inspect all labels in each iteration because of the aspired linear running time. Therefore, we handle those potential b-labels with pseudo-labels. Figure 4.2 illustrates the two cases when pseudo-labels are transformed into real labels during an update. After inserting edge e, the two c-labels with size one, whose parents c_1 and c_2, respectively, do not lie on the insertion path, become obsolete and need to be merged into one b-label.

The procedure HandlePendant preserves the correct assignment of pendants to their labels and pseudo-labels. To compute the corresponding label we traverse the BC-tree from the pendant upwards until we reach a node with degree at least three or the root with degree two. In the first case we check whether the current node is a c- or a b-node and if it is the parent of a label or pseudo-label. Then we either assign the pendant to this label or to a newly created one. In the second case the root with degree two is simply skipped and the traversal continues on the other side of the root. This procedure is outlined in HandleRootDeg2 where the traversal works in the same way, only top-down instead of bottom-up. We also stop at a node with degree at least three and distinguish between the cases of b- or c-nodes.

It is very important that the root is not a stop point for the traversal in case it has degree two. Otherwise, it may occur that the root breaks up two labels that actually belong together and therefore the algorithm would insert more edges than

Algorithm 4 HandlePendant

Input: pendant p of $bc(f)$

```
1: traverse in bc(f) from p to the root until the root or a
   ↪ bc-node with degree ≥ 3 is found, let bcn be the current node
2: if (bcn is a c-node) then
3:    add p to the label of bcn or
      ↪ create a new label with parent bcn and pendant b
4: else
5:    c* ← the last visited c-node
6:    if (bcn is the root and has degree = 2) then
7:       handleRootDeg2(p, c*)
8:    else
9:       create a new label with parent c* and pendant p
10:      if (bcn is a parent of a pseudo-label l') then
11:         add p to l'
12:         HandlePseudoLabel(l', p)
13:      else
14:         create a new pseudo-label with parent bcn and pendant p
15:      end if
16:   end if
17: end if
```

in the optimum solution.

Like mentioned above, the data structures need to be updated after each edge insertion. The basic steps are implemented in procedure Update where we check if the two involved labels have reached size one. In this case the remaining single pendant may need to be reassigned to another label (lines 7 and 10). Furthermore, we have to test the pseudo-labels whose parents are part of the insertion path, because they might become obsolete or need to be transformed into real labels (line 13).

One of the main ideas of the algorithm for biconnecting a graph with a fixed embedding is how edge-insertion works. Since the embedding is fixed the order of the adjacent edges of a vertex cannot be modified. Unlike in the *General* or *Planar Augmentation Problem* we cannot select two labels and connect two random pendants. The new edge divides the face and might isolate some pendants of one label. Therefore, each inserted edge has to satisfy the property, that all other pendants lie on the same side of the newly inserted edge. Hence, the pendants need to be neighbours in the BC-tree with regard to the embedding. When every edge satisfies this property pendants are not isolated and planarity will be preserved.

In our algorithm we select the label with maximum size and pick the first pendant in its list, see FindMatching. Then, we traverse the BC-tree in cyclic order to the next pendant. If this pendant belongs to the same label, we correct the order in the pendant-list of its label. Therefore, every path in the BC-tree between the same two pendants is not traversed twice. The traversal continues with this pendant and

stops when another label is found. Hence both returned pendants are neighbours and belong to different labels.

Algorithm 5 FindMatching

Input: the label l
Output: the two pendants p_1 and p_2 for the new edge

```
1: p1 ← first pendant in the list of l
2: p2 ← nil
3: while (p2 = nil) do
4:    traverse the BC-tree from p in cyclic order to the next pendant p'
5:    if (p' belongs to l) then
6:       delete p' in the list of l and reinsert p' in front of p
7:       p ← p'
8:    else
9:       p2 ← p'
10:   end if
11: end while
12: return (p1, p2)
```

Algorithm 6 Update

```
1: update bc(f) with new edge e
2: if (number of pendants = 2) then
3:    delete all labels and create a new label with the two pendants
4: else
5:    delete p1 and p2 from their labels and pseudo-labels
6:    if (size(l1) = 1) then
7:       HandlePendant(pendant of l1)
8:    end if
9:    if (size(l2) = 1) then
10:      HandlePendant(pendant of l2)
11:   end if
12:   for all (pseudo-labels l' with parent p' on the insertion path) do
13:      HandlePseudoLabel(l', p')
14:   end for
15: end if
```

Algorithm 7 HandleRootDeg2

Input: pendant p and the last c-node c^* on the path to the root

1: $bcn \leftarrow$ the c-node adjacent to the root with $bcn \neq c^*$
2: traverse $bc(f)$ from bcn downwards, until a bc-node with degree ≥ 3 is found, let bcn be the current node
3: **if** (bcn is a c-node) **then**
4: add p to the label of bcn or create a new label with parent bcn and pendant b
5: **else**
6: $c_2 \leftarrow$ the last visited c-node
7: create a new label with parent c_2 and pendant p
8: **if** (bcn is a parent of a pseudo-label l') **then**
9: add p to l'
10: HandlePseudoLabel(l', p)
11: **else**
12: create a new pseudo-label with parent bcn and pendant p
13: **end if**
14: **end if**

Algorithm 8 HandlePseudoLabel

Input: the pseudo-label l' and its parent p^*

1: **if** ($size(l') = 1$) and ($\deg(p^*) = 2$) **then**
2: $p \leftarrow$ pendant of l'
3: delete the c-label of p
4: delete l'
5: handlePendant(p)
6: **else**
7: **if** ($size(l') = 2$) and ($\deg(p^*) = 3$) **then**
8: delete the c-labels of the pendants of l'
9: transform l' into a label
10: **end if**
11: **end if**

4.2 Optimality

In this section we prove that the previously described algorithm always biconnects a planar graph with a given fixed embedding with the minimum number of edges. To achieve this we take a closer look at the algorithm and point out some properties of the algorithm and of BC-trees.

Like mentioned before, it is valid to consider each face separately. Therefore, the optimality of the algorithm depends on the optimum augmentation of each subgraph. Obviously, the lower bound of Theorem 3.3 is also a lower bound for the augmentation of each face-induced subgraph. We will show that the algorithm always achieves this lower bound, i.e. $\max\{d-1, \lceil \frac{p}{2} \rceil\}$, with d being the maximum degree of a c-node and p the number of pendants in the related BC-tree $bc(f)$.

First of all, we show that the newly inserted edges preserve planarity and that the embedding has not changed. A new edge is inserted between two simple vertices of two pendant-blocks. The pendants are neighbours in cyclic order in the BC-tree which represents the embedding of the graph. So all other pendants lie on the same side of this edge. Furthermore, the two newly connected vertices of the graph belong to the same face. Therefore, the embedding is preserved and planarity is not violated. Notice that this property also allows us to omit planarity tests for each edge insertion.

To achieve a minimum number of edges and the lower bound of $\max\{d-1, \lceil \frac{p}{2} \rceil\}$, we need to ensure that the leaf-connection condition (*lcc*) is always fulfilled, because only then the number of pendants decreases by two in every iteration. Instead of checking the *lcc* in every step explicitly the algorithm uses the mechanism of labels and pseudo-labels. Initially, the labels are computed correctly by the procedure HandlePendant. This is easy to see because the procedure just works like the definition of labels intended. Also the root as a b-node is handled correctly and can be a parent of a label, a pseudo-label, or will just be skipped during the traversal when it has degree two. A BC-tree with the root being a pendant is invalid and can be prevented easily.

During the algorithm labels change and are affected directly when a related pendant is connected. Furthermore, labels are affected indirectly if their parent is a part of the insertion path. Previously, we showed how the BC-tree is modified when an edge is inserted, c.f. Observation 3.1. Labels that are directly involved in the new edge, since one of their pendants is connected, are always considered by Update. They are deleted correctly if they reach minimum size and the remaining pendant gets a new label by HandlePendant. Consequently, all affected b- or c-labels with parents on the insertion path are then updated correctly. Pseudo-Labels are checked separately by HandlePseudoLabel and are transformed into labels, if necessary. In this case, the obsolete c-labels that are connected directly to the parent are also deleted. All other labels remain the same.

Now we can conclude that our algorithm terminates and that the resulting graph is biconnected. Since we insert edges between pendants of different labels, the *lcc* is always fulfilled and hence, the number of pendants decreases each time. After each iteration the BC-tree and the labels are updated correctly. The loop terminates

when there is at most one label left. In the case of one remaining label, its pendants are connected and the BC-tree finally contains one b-node. Therefore, the resulting graph is biconnected.

To prove optimality, we need to take a look at unbalanced BC-trees and their massive c-nodes. In our algorithm we always pick the largest label and connect one of its pendants with a matching pendant of another label. Since we want to decrease the degree of the massive c-node, we need the insertion path to include this c-node. Therefore, we first show that the massive c-node needs to be a parent of a label. Second, we guarantee that the algorithm automatically selects the label of the massive c-node because this label is always the maximum one. Both properties are gathered in the following lemma.

Lemma 4.1. *If a BC-tree is unbalanced, its massive c-node c^* has to be the parent of a label l_0. Furthermore, l_0 is the label with maximum size and no other label has the same size.*

Proof. Assume that the BC-tree contains p pendants. We prove both statements by contradiction.

If c^* is not the parent of a label, there cannot exist any c^*-chain and therefore, all subtrees connected to c^* would contain more than one pendant. From the definition of a massive c-node, c^* has degree $d := \deg(c^*) \geq \lceil \frac{p}{2} \rceil + 2$. Thus, there would be altogether $\geq 2(\lceil \frac{p}{2} \rceil + 2)$ pendants; however the BC-tree has only p. It follows that c^* has to be the parent of a label.

Now, let j be the number of pendants, or c^*-chains, respectively, of l_0. Then, there are $d - j$ subtrees connected to c^*, whereas each one contains at least two pendants. Assume there exists another c-node c_2 which is the parent of another label with at least the same size as l_0. Obviously, c_2 lies in one of the $d - j$ subtrees of c^*. Altogether the BC-tree would have

$$
\begin{aligned}
&\geq \underbrace{j}_{c^*} + \underbrace{j}_{c_2} + \underbrace{2(d-j-1)}_{\text{subtrees of } c^*} \\
&= 2d - 2 \\
&\geq 2(\left\lceil \frac{p}{2} \right\rceil + 2) - 2 \\
&\geq p + 2
\end{aligned}
$$

pendants. This is a contradiction and it follows that l_0 is the unique maximum label. □

We can also show in an analogue way that *critical* c-nodes—they have degree $\lceil \frac{p}{2} \rceil + 1$—are also always parents of labels. In general, the second statement from above is not true for critical nodes. However, we can show again by contradiction that no label is greater than the one of the critical node. We state this in the following lemma and skip the proof because it is similar to the one above.

Lemma 4.2. *If a BC-tree contains a critical c-node, this c-node always has to be the parent of a label. There is no other label with greater size.*

After these two lemmas we can finally prove the optimality of our approach for solving the *Planar Augmentation Problem* with fixed embedding.

Theorem 4.3. *Algorithm* PlanarAugmentationFix *solves* PA_{Fix} *with the minimum number of edges.*

Proof. As shown above, the algorithm always terminates, planarity is preserved, and the embedding remains unchanged. To complete the proof we now show that each face-induced subgraph is made biconnected by inserting the minimum number of edges, namely $\max\{d-1, \lceil \frac{p}{2} \rceil\}$ (cf. Theorem 3.3), with d being the maximum degree of a c-node and p the number of pendants in the BC-tree. To accomplish this, we consider two cases separately, which are based on the existence of a massive c-node:

1. $d \geq \lceil \frac{p}{2} \rceil + 2$
   $\Rightarrow$ the BC-tree is unbalanced and has a massive c-node c^*.

2. $d \leq \lceil \frac{p}{2} \rceil + 1$
   $\Rightarrow$ the BC-tree is balanced.

Case 1 – unbalanced graph:
If the first case the algorithm must not exceed the lower bound of $\deg(c^*) - 1$ new edges to biconnect the graph. If c^* is the massive c-node we have shown in Lemma 4.1 that c^* has to be the parent of the maximum label, say l_0. Therefore, the algorithm automatically selects l_0 to find an appropriate matching. The following induction on $\deg(c^*)$ proves that the bound $\deg(c^*) - 1$ is achieved:

- If $\deg(c^*) \in \{1, 2, 3\}$ there exists no valid unbalanced BC-tree:
  The cases $\deg(c^*) \in \{1, 2\}$ are obvious. If $\deg(c^*) = 3$, the number of pendants needs to be at least 3. But then $\lceil \frac{p}{2} \rceil + 2 \geq 4 > \deg(c^*)$ would hold.

- The base case is $\deg(c^*) = 4$:
  Since $\deg(c^*)$ needs to be $\geq \lceil \frac{p}{2} \rceil + 2$ it follows that $p = 4$. Hence, the BC-tree consists of 1 label l_0 with 4 pendants, and $3 = \deg(c^*) - 1$ edges are inserted to biconnect the graph.

- Inductive hypothesis:
  For all unbalanced BC-trees with a massive c-node c_2^* with $\deg(c_2^*) < \deg(c^*)$ the algorithm biconnects the graph with $\deg(c_2^*) - 1$ edges.

- Inductive step $\deg(c^*) \geq 5$:

  Case 1: $= 1$ label $\Rightarrow$ only c^*-chains
  $\Rightarrow$ $\deg(c^*) - 1$ edges are inserted
  Case 2: > 1 label $\Rightarrow$ a new edge between a pendant of l_0 and another label is inserted
  $\Rightarrow$ $\deg(c^*)$ decreases by 1
  $\Rightarrow$ c^* remains massive because $\lceil \frac{p}{2} \rceil$ decreases by 1, too
  $\Rightarrow$ it follows by the induction hypothesis that $((\deg(c^*) - 1) - 1) + 1$ $= \deg(c^*) - 1$ edges are inserted.

Case 2 – balanced graph:
In the second case the BC-tree is balanced and we have to show that we achieve the lower bound of $\lceil \frac{p}{2} \rceil$. The proof is based on induction too, this time on the number of pendants p:

- The base cases are $p = 2$ and $p = 3$:
  - $p = 2$: only 1 label $\Rightarrow 1 = \lceil \frac{p}{2} \rceil$ edges are inserted
  - $p = 3$: either 1 or 3 labels. In both cases $2 = \lceil \frac{p}{2} \rceil$ edges are inserted
- Inductive hypothesis:
  For all balanced BC-trees with $p' < p$ pendants the algorithm biconnects the graph with $\left\lceil \frac{p'}{2} \right\rceil$ edges.
- Inductive step $p > 3$:
  - $\Rightarrow$ As precondition we have a balanced BC-tree.
  - $\Rightarrow$ Since $p > 3$ holds, we can conclude that we have at least two labels.
  - $\Rightarrow$ The *lcc* is fulfilled for the new edge.
  - $\Rightarrow$ p decreases by 2.
  - $\Rightarrow$ If we can show that the BC-tree is still balanced after the edge-insertion the induction hypothesis will ensure that $\lceil \frac{p-2}{2} \rceil + 1 = \lceil \frac{p}{2} \rceil$ edges are inserted.

Therefore, we show by contradiction that no massive c-node arises during the algorithm.

Assume that c^* is the new massive c-node after the insertion of a new edge. Let p and p' denote the number of pendants before and after the insertion, respectively. Therefore, $p' = p - 2$ holds and from the definition of a massive c-node we have $\deg^{after}(c^*) \geq \left\lceil \frac{p'}{2} \right\rceil + 2 = \lceil \frac{p-2}{2} \rceil + 2$. After Observation 3.1, the degree of a c-node

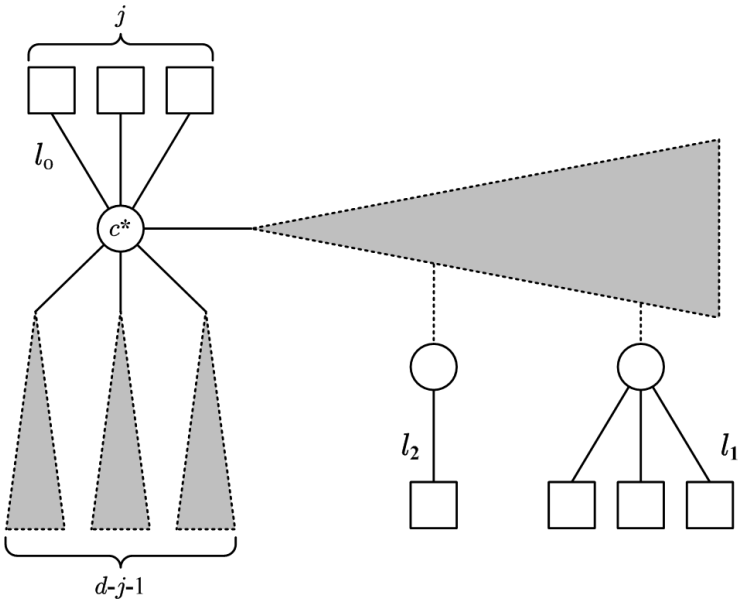


Figure 4.3: Situation before inserting an edge with a critical c-node c^*, its label l_0, and the two involved labels l_1 and l_2.

lying on the insertion path between the two connected pendants is either reduced by one or the c-node is contracted to a new b-node. Therefore, we can conclude that c^* is not in the insertion path, since otherwise $\deg^{before}(c^*) = \deg^{after}(c^*) + 1 \geq \left\lceil \frac{p'}{2} \right\rceil + 3 = \left\lceil \frac{p}{2} \right\rceil + 2$ and c^* would already be a massive c-node.

Consequently, $d := \deg^{before}(c^*) = \deg^{after}(c^*) = \left\lceil \frac{p'}{2} \right\rceil + 2 = \left\lceil \frac{p}{2} \right\rceil + 1$ holds and therefore, c^* has to be a critical c-node before edge-insertion. From Lemma 4.2 it follows that c^* is a parent of a label and that no other label is greater in size. Let l_0 denote this label with $size(l_0) =: j$. Thus, there are $d - j$ subtrees of c^* where each of them contains at least two pendants. There have to exist two other labels l_1 and l_2 in the BC-tree because l_0 is not involved in the new edge. One of these labels also need to contain at least j pendants because this label was selected by the algorithm; w.l.o.g. we assume that this is l_1, see Figure 4.3. Both of the labels l_1 and l_2 are part of the same subtree of c^* because otherwise c^* would be a part of the insertion path.

We can now sum up the number of pendants in the BC-tree before edge-insertion:

$$
\begin{aligned}
&\geq \underbrace{j}_{l_0} + \underbrace{j}_{l_1} + \underbrace{2(d-j-1)}_{\text{subtrees of } c^*} + \underbrace{1}_{l_2} \\
&= 2d - 1 \\
&\geq 2\left(\left\lceil \frac{p}{2} \right\rceil + 1\right) - 1 \\
&\geq p + 1
\end{aligned}
$$

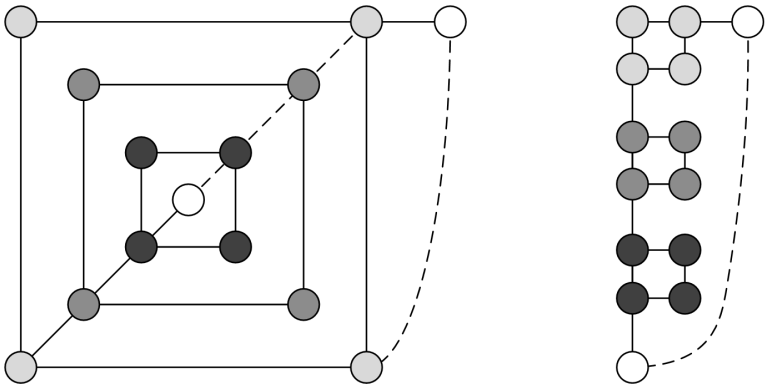


Figure 4.4: An example graph with two pendants and two embeddings. The fixed embedding in (a) induces an augmentation with four edges, whereas the optimal solution in (b) consists of one edge.

This is the desired contradiction. It follows that no c-node can become massive and the BC-tree remains balanced. Therefore, the lower bound of $\lceil \frac{p}{2} \rceil$ edges is inserted in the second case.

Altogether, the algorithm always achieves the lower bound of $\max\{d-1, \lceil \frac{p}{2} \rceil\}$ for each face-induced subgraph and the theorem follows.

□

Although PA_{Fix} can be solved optimally, this does not result automatically in a good approximation ratio for the general *Planar Augmentation Problem*. The wrong fixed embedding possibly requires k times the cardinality of the augmentation for the optimal embedded graph, for an arbitrary integer $k > 0$. Figure 4.4 illustrates an example. The vertices are colored for a better overview. The augmentation with the fixed embedding requires four edges, as shown in (a), whereas the optimal solution contains only one edge, as shown in (b). This example can be expanded to arbitrary k.

4.3 Running Time and Space

After proving that our approach biconnects a given graph with the minimum number of edges we determine now the required space and the running time.

Theorem 4.4. *Algorithm* PlanarAugmentationFix *requires* $\mathcal{O}(|V|+|E|)$ *space.*

Proof. The algorithm considers each face of the graph separately. Therefore, the current subgraph requires $\mathcal{O}(|V|+|E|)$ space. Furthermore, a BC-tree is also linearly bounded in the size of the underlying graph. Since all other data structures like label-lists depend on the BC-tree they are also linearly bounded.

□

For the proof of the running time, the *Ackermann function* $A_k(j)$ and its inverse is required (the definition is adopted from [8]):

$$A_k(j) := \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$$

The notation A^j denotes, that the function A is iteratively applied j times. The inverse of the Ackermann function is defined as

$$\alpha(n) := \min\{k : A_k(1) \geq n\}.$$

The function A_k is a very quickly growing function whereas $\alpha(n)$ grows very slowly:

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2047 \\ 4 & \text{for } 2048 \leq n \leq A_4(1) \end{cases}$$

Since $A_4(1) \gg 10^{80}$, which is the estimated number of atoms in the universe, $\alpha(n) \leq 4$ holds for all practical purposes. In literature, there are different definitions of the Ackermann function and its inverse. However, all of them grow really quickly and slowly, respectively.

Theorem 4.5. *Algorithm* PlanarAugmentationFix *has running time* $\mathcal{O}(|V| + |E| + \alpha(|V|)|V|)$.

Proof. In the algorithm each face is considered separately and—even though vertices and edges may belong to several faces—the sum over all induced subgraphs is linear in the input size. Therefore, we need to show that the augmentation of one face does not exceed the time bound, and hence we are able to apply the running time to the whole algorithm.

BC-trees can be constructed in linear time of the size of the underlying graph [39]. An update after inserting a new edge in the graph and in the BC-tree, respectively, takes amortized time $\mathcal{O}(\alpha(|V|))$. This can be achieved by using a union-find data structure as described in [44]. Since after an edge-insertion the path between the two newly adjacent pendants is part of a new cycle, and therefore it is contracted to one b-node, all paths in the BC-tree involved in FindMatching and during BC-tree-updates are traversed only a constant time.

We can apply the same argument to pendants and the procedure HandlePendant. This function can be called multiple times for each pendant and it does not seem that the total costs are linear. But we can improve this function by starting the traversal at the parent of the associated label, if existent. Therefore, we do not traverse an already visited path twice.

The last crucial part concerning the running time of the algorithm is the selection of the largest label and the correct sorting in each iteration. We can solve this problem by using buckets for sets of labels with the same size. Then, a decrease

or increase in the size of a label can be managed in constant time. Since a newly inserted edge affects only at most four labels we do not exceed the time limit.

All other minor operations also take constant time. The number of pendants and labels is bounded by $\mathcal{O}(|V|)$ and so is the number of iterations of the main loop.

Altogether the running time is $\mathcal{O}(|V_i| + |E_i|)$ for each subgraph $G_i = (V_i, E_i)$ induced by face f_i, except for the updates of the BC-tree. Since the maximal number of inserted edges is $\mathcal{O}(|V_i|)$, these operations take in total time $\mathcal{O}(\alpha(|V_i|)|V_i|)$.

□

Chapter 5

Planar Augmentation for Almost Biconnected Graphs

This chapter focuses on the *Planar Augmentation Problem* for graphs with the special property that all cutvertices belong to the same biconnected component. The problem was defined in Section 3.1 and is shortened by PA_{Bic}. Section 5.1 gives an introduction to this problem by describing the differences to the general problem. In Section 5.2, we determine the complexity of this problem and although the input graphs have a quite simple biconnected structure, we show that this special case is also $\mathcal{NP}$-hard. As a consequence of the constructed polynomial-time reduction, PA_{Bic} is even $\mathcal{NP}$-hard in case the SPQR-tree (without Q-nodes) has only height one. Afterwards, we will discuss a new approximation algorithm for this special case with approximation ratio $\frac{5}{3}$. We present the algorithm in Section 5.3, the analysis of the approximation quality in Section 5.4, and finally, the running time in Section 5.5.

5.1 Introduction

Before focusing on PA_{Bic}, we need to introduce another well-known combinatorial optimization problem on graphs, the *Maximum (Cardinality) Matching Problem*. A *matching* in a graph $G = (V, E)$ is an edge set $M \subseteq E$ such that each vertex has at most one incident edge in the set. A *maximum (cardinality) matching* is a matching with the maximum number of edges. By contrast, a *maximal matching* is maximal with respect to the edge set, i.e., no more edges can be added to the set. Both problems can be solved in polynomial time; an algorithm for the maximum cardinality matching with running time $\mathcal{O}(\sqrt{|V|}|E|)$ can be found in [36]. A maximal matching can be computed in linear time by a greedy approach.

The *Planar Augmentation Problem* can be interpreted as a computation of a matching between the pendants of different labels. In particular, this holds for the PA_{Bic} problem. A new edge between two pendants of different labels fulfills the leaf-connection condition and therefore, the number of pendants decreases by two. We call a pendant *matched* or *cheap*, respectively, if it is connected to a pendant of another label in the solution. All other pendants, i.e., pendants being eliminated by

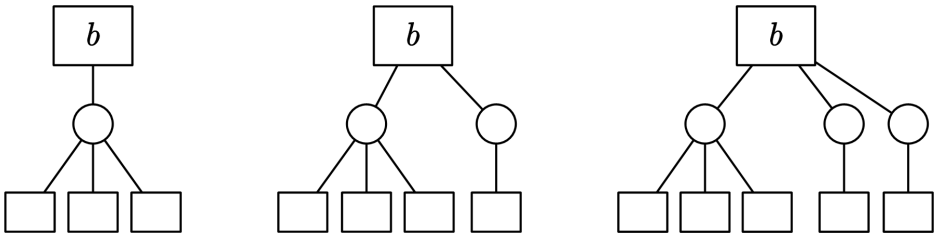


Figure 5.1: Three examples of BC-trees for which the two label definitions induce different pendant sets. The b-node representing the biconnected core is always the root b.

unprofitable edges, are *expensive*.

As mentioned in Section 3.1 and in [31], the *Planar Augmentation Problem* on graphs with one triconnected component containing all cutvertices (PA_{Tric}) can be solved in polynomial time. We summarize the basic ideas, since they are used in some parts of the new algorithm.

In case all cutvertices belong to one triconnected component, the embedding of the graph can be fixed arbitrarily. S-skeletons have only one embedding. An R-skeleton has exactly two embeddings which are mirror images of each other. A P-skeleton has many possible embeddings, but since all cutvertices are part of one component, the parents of the labels can only correspond to the poles of the P-skeleton. Therefore, fixing an embedding does not make a difference for augmentation.

The algorithm for PA_{Tric} generates an auxiliary graph H. Each pendant in the BC-tree of the original graph gets one representing vertex in H. An edge is inserted between two vertices of H, if and only if the corresponding pendants do not belong to the same label and the two label-parents are adjacent to the same face. The algorithm then computes a maximum cardinality matching M in H and embeds the pendants according to M into the faces. The graph with this fixed embedding can be augmented optimally with the algorithm presented in the previous chapter.

The fact, that the embedding of the triconnected component can be fixed without restricting the set of optimum solutions makes this augmentation problem easy.

An instance of PA_{Bic} also has a quite simple biconnected structure. Since all cutvertices belong to one biconnected component, called the *biconnected core* of the graph, the BC-tree contains the corresponding b-node and all other c-nodes are adjacent to it. All other b-nodes are leaves and are directly attached to the c-nodes.

The computation of an optimal solution for PA_{Bic} with a fixed embedding is simpler than in the general case. This problem can be solved in linear time because the updates of the BC-tree are not necessary. We only have to modify the original definition of labels slightly.

Two pendants are *siblings* if they are adjacent to the same c-node and neither of them is the b-node that represents the biconnected core. Obviously, the relation "sibling" between the pendants is an equivalence relation.

Definition 5.1 (**Label** (for PA_{Bic}))**.** *Consider the BC-tree of an instance of PA_{Bic}. The labels are the equivalence classes with respect to the sibling relation together with*

the parent node.

Therefore, each c-node becomes the parent of a label and b-labels do not exist anymore. From now on, we use this new definition of labels.

Figure 5.1 illustrates three examples for which the two definitions of labels induce different pendant sets. Assume that the b-node representing the biconnected core is always the root of the BC-tree. Obviously, in case the root has at least degree four both label sets are always identical. If the root is a pendant, cf. Figure 5.1 (a), then there exists at most one c-node. Thus, there is at most one label in both cases which varies only in this pendant. If the root has degree two, the two definitions only differ when the root splits up two labels, as seen in Figure (b). The same difference may occur in case of degree three. Then, the b-node representing the biconnected core might originally be a b-label, cf. Figure (c).

For the augmentation of an instance for PA_{Bic} with a fixed embedding, a similar approach to the original algorithm can be applied. By iterative selection of the largest label and connection of its rightmost pendant with the leftmost pendant of the right neighbouring label, the BC-tree can be augmented optimally. In case, there is one label left, say l_0, $size(l_0)$ edges are inserted—$size(l_0) - 1$ edges to connect the pendants of l_0 and the last one for the connection of the biconnected core with the remaining pendant. During the algorithm, the labels need not to be merged, only the size has to be updated.

The proof of the optimality of this approach is based on the same ideas as the proof of Lemma 4.3. In case of a massive c-node c^*, the related label is also the maximum one and it is always utilized for the next edge. Similar arguments ensure, that the number of added edges is $\deg(c^*) - 1$. In the other case, i.e. the BC-tree is balanced, the algorithm inserts an edge between two pendants that fulfill the leaf-connection-condition. Hence, the BC-tree remains balanced and the lower bound of $\lceil \frac{p}{2} \rceil$ can be achieved.

Furthermore, the number of cheap and expensive pendants inside a fixed face can be computed only on basis of the size of the largest label and the number of the remaining pendants.

Let f be a face of an instance for PA_{Bic} with fixed embedding and $bc(f)$ the corresponding BC-tree. Let l_0 be the largest label with c being its parent c-node and k the number of the remaining pendants. Further, let p denote the number of pendants in $bc(f)$, hence $p = size(l_0) + k$. Let x be the integer such that $x = size(l_0) - k$.

The degree of c equals $size(l_0) + 1$. Therefore,

$$\deg(c) = size(l_0) + 1 = \frac{1}{2}(size(l_0) + k + x) + 1 = \frac{1}{2}p + \frac{1}{2}x + 1.$$

- If $x \geq 4$, $\deg(c) \geq \frac{1}{2}p + 2 + 1 \geq \lceil \frac{p}{2} \rceil + 2 \Rightarrow c$ is massive.
- If $x = 3$, p is odd and $\deg(c) = \frac{1}{2}p + 2.5 = \lceil \frac{p}{2} \rceil + 2 \Rightarrow c$ is massive.
- If $x = 2$, p is even and hence, $\deg(c) = \frac{1}{2}p + 2 \Rightarrow c$ is massive.

- If $x = 1$, c is not massive, but since l_0 is the largest label, there cannot exist another massive c-node. For a balanced BC-tree, the lower bound of required edges is $\lceil \frac{p}{2} \rceil$. Here, $\lceil \frac{p}{2} \rceil = \left\lceil \frac{size(l_0)+k}{2} \right\rceil = \lceil \frac{2k+1}{2} \rceil = k + 1 = size(l_0)$.

- If $x \leq 0$, there is no massive c-node and hence, the number of required edges is $\lceil \frac{p}{2} \rceil$.

For an unbalanced graph, the number of required edges depends on the degree of the massive c-node. In case $size(l_0) > k$, the parent of l_0 is massive or its degree equals $\lceil \frac{p}{2} \rceil + 1$. Otherwise, if $size(l_0) \leq k$ holds, the underlying BC-tree is balanced.

Observation 5.1. *The number of required edges for augmenting a fixed face of an instance of PA_{Bic} depends on the size of the maximum label l_0 and the number of the remaining pendants k. In case $size(l_0) > k$, $size(l_0)$ edges are required and sufficient. Otherwise, if $size(l_0) \leq k$ holds, an optimal augmentation consists of $\left\lceil \frac{size(l_0)+k}{2} \right\rceil$ edges.*

5.2 $\mathcal{NP}$-Completeness

We already presented a proof for the $\mathcal{NP}$-hardness of the *Planar Augmentation Problem* in Section 3.3. Unfortunately, the constructed graph in the reduction is not feasible for PA_{Bic} since the cutvertices do not belong to one biconnected component. Therefore, the complexity result cannot be adopted to this special case of PA by the latter proof.

Hence, we construct a new polynomial-time reduction from another problem on planar graphs, namely the *Planar Vertex Cover Problem*, to PA_{Bic}.

For an undirected graph $G = (V, E)$ a subset of vertices $V_{vc} \subseteq V$ is called *vertex cover* if every edge has at least one endpoint in V_{vc}. The notation may be somewhat misleading, since not the vertices are being covered, but the edges are.

Definition 5.2 (**Vertex Cover Problem**). *Let $G = (V, E)$ be an undirected and connected graph. The Vertex Cover Problem (VC) is the problem of deciding whether G has a vertex cover with at most k vertices or not.*

The *Vertex Cover Problem* is well-studied. In fact, it is one of the famous 21 problems mentioned in [33], for which Karp proved the $\mathcal{NP}$-completeness. In [32], Karakostas introduced an approximation algorithm with ratio $2 - \Theta(1/\sqrt{\log |V|})$, which is the best known approximation quality. Moreover, Dinur and Safra showed that the optimization problem of VC cannot be approximated with a ratio less than 1.3606, unless $\mathcal{NP} = \mathcal{P}$; see [10].

Since we want to prove the $\mathcal{NP}$-completeness of PA_{Bic}, which requires planar graphs, the more interesting problem for us is the planar version of this problem.

Definition 5.3 (**Planar Vertex Cover Problem**). *If the input graphs of the Vertex Cover Problem are restricted to planar graphs then this problem is called Planar Vertex Cover Problem and it is denoted by VC_{planar}.*

In [18], Garey and Johnson investigated restricted cases of some $\mathcal{NP}$-complete problems, among them VC_{planar}, to which they refer as *Planar Node Cover*. They constructed a polynomial-time reduction from the general *Vertex Cover Problem* to this special case by replacing crossings in the graph with planar subgraphs to show the following theorem.

Theorem 5.2. *The Planar Vertex Cover Problem is $\mathcal{NP}$-complete.*

Garey and Johnson actually proved that the restricted *Vertex Cover Problem* on input graphs with maximum degree three is also $\mathcal{NP}$-complete. This implies, that the *Planar Vertex Cover Problem* with maximum degree six is $\mathcal{NP}$-complete, since the constructed graph in the reduction between the two problems does not contain a vertex that exceeds this bound. However, for our purposes this further restriction is not necessary.

Because of the $\mathcal{NP}$-completeness of VC_{planar} we are now able to determine the complexity of PA_{Bic}.

Theorem 5.3. *The Planar Augmentation Problem with the restriction that all cutvertices belong to the same biconnected component (PA_{Bic}) is $\mathcal{NP}$-hard.*

Proof. To verify the theorem, we consider the decision problem of PA_{Bic} and show that it is $\mathcal{NP}$-complete. The decision problem of PA_{Bic}, denoted by PA_{Bic}^{dec}, is to decide whether at most k edges are sufficient to biconnect a given graph without losing planarity.

Obviously, PA_{Bic}^{dec} belongs to the complexity class $\mathcal{NP}$. For a graph $G = (V, E)$ and a given set of edges E' it is easy to decide in polynomial time whether $G' = (V, E \cup E')$ is planar and biconnected, or not.

Assume $G = (V, E)$ is the input graph for the *Planar Vertex Cover Problem*. We construct an input graph $G' = (V', E')$ for PA_{Bic}^{dec} with the property that G has a vertex cover with size k if and only if G' can be made biconnected with $7.5m|E| - |V| + k$ edges, for an even integer m which will be specified later.

Each vertex $v \in V$ is represented in G' by a subgraph, a *vertex gadget*, which is planar and has a triconnected structure. A vertex gadget has two relevant faces f_v and f'_v which are separated by one triconnected subgraph, the *decision component*. Furthermore, the boundary is split up by $\deg(v)$ triconnected subgraphs, one for each incident edge, the so-called *edge connection components*. The gadgets of two adjacent vertices v, w are connected, inducing one relevant face f_{vw} which is bordered on two sides by the edge connection components. We refer to the face between two connected gadgets as the *edge component*. Figure 5.2 illustrates the triconnected structure for two vertex gadgets.

So far, the constructed graph is biconnected and planar. Therefore, the graph G' is expanded by some pendants. Let m be an even integer greater than $2|V|$, say $m := 4|V|$. Each edge component and edge connection component obtains one label with exactly m pendants. Inside the faces of the vertex gadgets there are $6\deg(v)m - 2$ pendants, depending on the degree of v. The decision component is extended by two labels, one on each side, with sizes $2\deg(v)m$ and $\deg(v)m$, respectively. Furthermore, there are two labels in face f'_v, one with $\deg(v)m$ and the

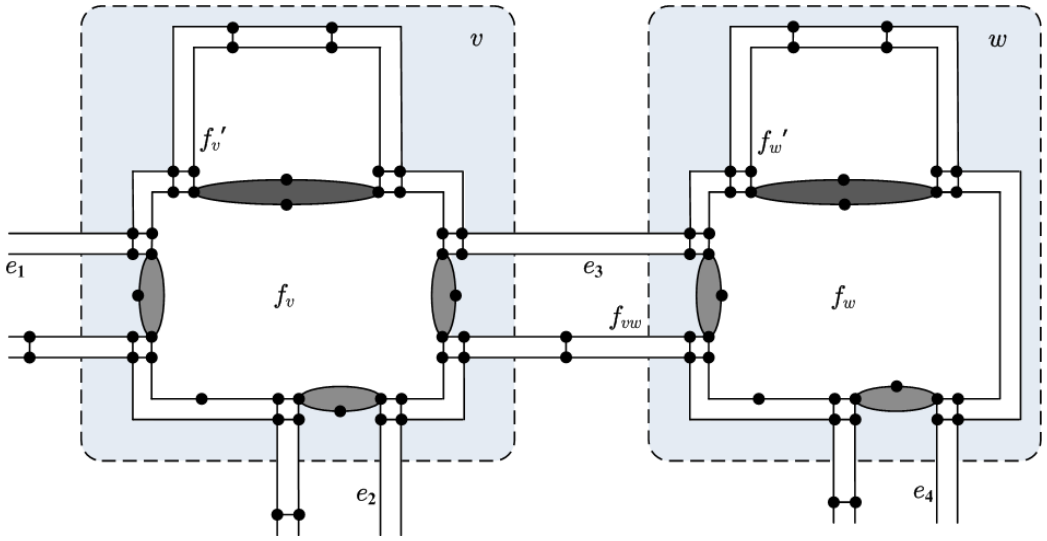


Figure 5.2: The triconnected structure of the constructed graph with two vertex gadgets for the adjacent vertices v and w and an edge $e_3 = (v, w)$. The three incident vertices of edges e_1, e_2, and e_4 are cut out. The shaded parts are triconnected components. The dark one at the top of each vertex gadget is the decision component, the lighter ones are the edge connection components.

other with $\deg(v)m - 2$ pendants. Finally, a label, again with $\deg(v)m$ pendants, is inserted into face f_v. The exact positions are illustrated in Figure 5.3. There, both vertex gadgets have the described labels and pendants attached, but the decision and the edge connection components are orientated contrary.

The number of pendants of one vertex gadget corresponding to vertex v is $7\deg(v)m - 2$, the number of added pendants for each edge is m. Hence, the total number of pendants is

$$\begin{aligned} &= \sum_{v \in V} (7\deg(v)m - 2) + \sum_{e \in E} m \\ &= 7m \sum_{v \in V} \deg(v) + m|E| - 2|V| \\ &= 14m|E| + m|E| - 2|V| \\ &= 15m|E| - 2|V|. \end{aligned}$$

Obviously, the pendants of the edge-labels can only be connected with pendants from the labels of the edge connection components of incident vertices. Moreover, the only possible matching partner for the pendants in face f_v' are those which are attached to the decision component.

The most crucial parts of the graph are the decision components. If one is orientated, such that the $2\deg(v)m$ pendants are embedded into face f_v', for a vertex v, then it will be equivalent of adding v to the vertex cover; compare the left vertex gadget in Figure 5.3. In this case, the pendants of the edge connection components can be embedded into the faces that represent the edges, because in f_v exist two

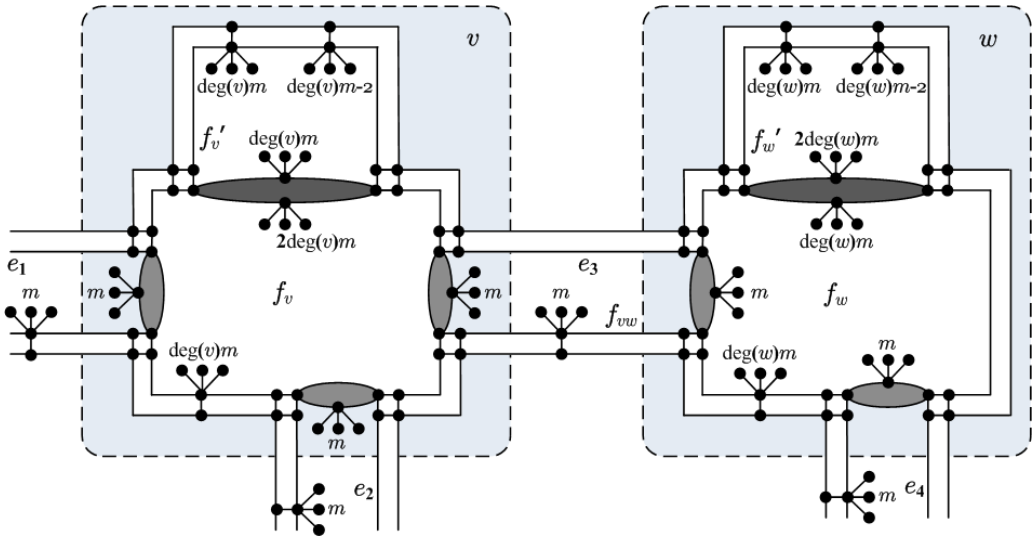


Figure 5.3: The constructed graph expanded by the pendants. The decision component and the edge connection components of the two vertices v and w are embedded contrarily, that is $v \in V_{vc}$ and $w \notin V_{vc}$.

labels with size $\deg(v)m$ which can be connected among each other. Notice that this decision induces two expensive pendants in face f_v'.

The contrary embedding of one decision component reflects the decision $v \notin V_{vc}$; compare the right vertex gadget in Figure 5.3. Then, every edge connection component has to be orientated such that the m pendants are embedded into face f_v, because otherwise there would exist m expensive pendants.

To verify the correctness of this reduction, we have to show that an instance $G = (V, E)$ with parameter k is a yes-instance for the *Planar Vertex Cover Problem* if and only if $G' = (V', E')$ is a yes-instance for PA^{dec}_{Bic} with parameter $7.5m|E|-|V|+k$.

"$\Rightarrow$": Let $\{G = (V, E), k\}$ be a valid instance for VC_{planar} and $V_{vc} \subseteq V$ be a vertex-cover of size k. We construct the graph G' as described before and fix the embedding such that $7.5m|E| - |V| + k$ new edges will be sufficient to biconnect G'. For every vertex $v \in V_{vc}$ we orientate the decision component of the corresponding vertex gadget such that the $2\deg(v)m$ pendants are embedded into face f_v'. The pendants of each edge connection component of v are embedded into the faces that represent the incident edges. For a vertex $w \notin V_{vc}$, the decision component is orientated the other way round and the m-labels of all edge connection components of w are inserted into f_v.

Since V_{vc} is a vertex cover, every edge has an incident vertex in the set. After the described embedding decisions, either two or three labels with size m are embedded into each edge component. Therefore, the m pendants of each edge component can be connected to at least one label of an edge connection component. Hence, all contained pendants are cheap ones. A vertex gadget of a vertex $v \in V_{vc}$ induces exactly two expensive pendants in face f_v' and none in f_v, whereas all pendants in a vertex gadget of a vertex $w \notin V_{vc}$ are cheap ones.

Altogether there are $2k$ expensive pendants, two for each $v \in V_{vc}$, what leads to the desired number of

$$\begin{aligned} &= \frac{1}{2}(15m|E| - 2|V| - 2k) + 2k \\ &= 7.5m|E| - |V| + k \end{aligned}$$

added edges for augmentation.

"$\Leftarrow$": Now, let G' be the constructed and augmented graph and let G be the underlying instance for the *Planar Vertex Cover Problem*. Assume that the augmentation of G' requires $7.5m|E| - |V| + k$ edges.

By the construction of the graph, the pendants of each label, except for the one attached to the decision component with $2\deg(v)m$ pendants, are either all cheap or all expensive. Furthermore, there are only two possibilities for expensive pendants. On the one hand, if the label with size $2\deg(v)m$ is embedded into f'_v, for a vertex v, there do arise two expensive pendants. On the other hand, if the decision component is embedded contrarily, all pendants in f'_v are cheap ones. Then, there can be m expensive pendants among the $2\deg(v)m$-label, in the case one edge connection component is embedded with the m-label into the edge-face, or the edge-pendants are expensive, if both incident edge connection components are embedded with their labels into the corresponding f_v-faces. In both cases the number of expensive pendants is a multiple of m.

The total number of pendants is $15m|E| - 2|V|$ and the augmentation requires $7.5m|E| - |V| + k$ edges. Let p_c denote the number of cheap and p_x the number of expensive pendants.

$$\begin{aligned} p_c + p_x &= 15m|E| - 2|V| \text{ and} \\ 0.5p_c + p_x &= 7.5m|E| - |V| + k \\ \Rightarrow 0.5p_c &= 7.5m|E| - |V| - k \\ p_c &= 15m|E| - 2|V| - 2k \end{aligned}$$

It follows that there are exactly $2k$ expensive pendants. Since $m = 4|V| > 2k$, all m-labels need to be cheap and all of the expensive pendants are part of the labels with size $2\deg(v)m$.

It follows, that there are k vertex components whose decision component is embedded such that the label with $2\deg(v)m$ pendants lies inside the face f'_v.

Now, we construct a vertex cover V_{vc} with size k as follows. A vertex $v \in V$ is inserted into the set V_{vc} if the corresponding vertex gadget contains two expensive pendants. Therefore, $|V_{vc}| = k$ and since no edge-label is expensive, each edge has an incident vertex in V_{vc}, and the set is a valid vertex cover.

To complete the proof, we need to verify that the size of the constructed graph is a polynomial in the size of the input graph and that the computation can be done in polynomial time. The total number of pendants is $15m|E| - 2|V|$. With

$m = 4|V|$, the number of pendants is bounded by $\mathcal{O}(|V||E|)$. The number of all other edges and vertices of the constructed graph G' is linearly bounded in the size of the input graph G. Altogether, G' can be computed in time $\mathcal{O}(|V||E|)$, it uses space $\mathcal{O}(|V||E|)$, and the theorem follows.

□

The previous reduction allows to restrict the *Planar Augmentation Problem* even more without losing the $\mathcal{NP}$-hardness. Consider the constructed graph G' and the SPQR-tree of the biconnected core. Each edge and decision component is a triconnected graph and the skeleton of the remaining graph is also triconnected. Therefore, the SPQR-tree of the biconnected core contains exclusively R-nodes. Furthermore, after omitting the Q-nodes, the SPQR-tree has only height one.

Definition 5.4. *PA_{Bic-2} is the restricted version of the Planar Augmentation Problem with the constraints that all cutvertices belong to one biconnected component and the SPQR-tree of the biconnected core has height one (Q-nodes are omitted).*

Corollary 5.4. *PA_{Bic-2} is $\mathcal{NP}$-hard, even if the SPQR-tree of the biconnected core contains only R-nodes (and Q-nodes).*

5.3 The Approximation Algorithm

The main advantage of $PA_{Bic(-2)}$ over the general *Planar Augmentation Problem* is the central biconnected structure of the input graphs which allows the use of the SPQR data structure. We introduced the SPQR-tree in Chapter 2.4 and we will utilize this data structure to develop an approximation algorithm for PA_{Bic-2} with ratio $\frac{5}{3}$. Like described in Section 2.4.3, the permutation of the parallel edges of a P-skeleton and the orientation of the R-nodes have to be specified to obtain a desired embedding. The general idea of the approach for PA_{Bic-2} is to build the SPQR-tree, traverse it bottom-up, and make decisions for the embedding of each skeleton with respect to the position and the size of the labels. Unlike the former approximation algorithms, edges are inserted only at the end of the algorithm after the embedding has been fixed. During the traversal, the major difficulty is the positioning of the pendants and labels to each other. For a good approximation ratio, it is important that the decisions do not affect too many pendants in a way that they cannot be connected profitably anymore.

For the following approximation algorithm, we modify the general definition of the SPQR-tree. First, the root of the SPQR-tree is changed from a Q-node to the R-, S-, or P-node with the largest degree. Second, all Q-nodes are omitted, that is they are merged with their parent nodes and therefore, each edge of a skeleton is either a virtual edge or a real edge.

Since the SPQR-tree only represents the triconnected structure of the biconnected core, the general skeleton graphs need to be expanded further. Consider a skeleton graph G_μ corresponding to a node μ of the SPQR-tree $\mathcal{T}$. There are two types of pendants. First of all, there are pendants whose parents are vertices in the current skeleton graph. We call these pendants *R-*, *S-*, and *P-pendants*, respectively,

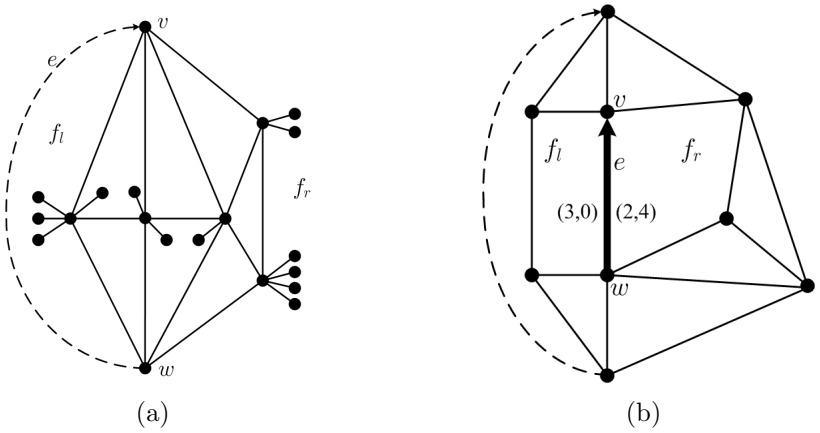


Figure 5.4: (a) A skeleton μ of an R-node with the reference edge e and a fixed embedding of the pendants. (b) The skeleton of the parent of μ with the virtual edge e and the corresponding *dp*-values.

depending on the current node type. The same notation is used for the corresponding *R-*, *S-*, and *P-labels*. Second of all, there might be pendants which are embedded into the external faces of expansion graphs of virtual edges.

Because of the optimality of the algorithm for fixed embedding it is possible to compute the exact number of cheap and expensive pendants inside a face. The information for external faces of skeleton graphs is incomplete, but the number of pendants which can be matched already with pendants from other labels and the number of pendants which require an additional pendant to become cheap can be computed analogously. We refer to the first set of pendants as *potential* and to the second set as *demanding* pendants and call the pair of demand and potential the *dp-value*. These values are added to both sides of each virtual edge of a skeleton. In case the edge is a real edge the *dp*-value is $(0, 0)$.

Figure 5.4 illustrates an example of an R-skeleton with reference edge e and the corresponding skeleton of the parent node. There are three demanding and zero potential pendants in f_l and two demanding and four potential pendants in face f_r.

Furthermore, the virtual and the reference edges are considered to be directed. The direction allows parent nodes to flip entire subgraphs by reversing the virtual edges. Let ν be a child of μ in the SPQR-tree and f_l and f_r the external faces of $skeleton(\nu)$ with directed reference edge e_μ. Furthermore, let e_ν be the directed virtual edge of ν in $skeleton(\mu)$. The right and the left face of e_ν correspond to the right and left face of e_μ. By flipping the direction of e_ν, the related skeleton and hence, the expansion graph, is mirrored.

Furthermore, the *dp*-value provides all required information for the parent. For example, the size of the largest label can be computed easily in case the demand is greater than one. Consider a side of a virtual edge e with a *dp*-value of (e_d, e_p) and $e_d \geq 2$. Since the potential is always a power of two, the largest label has $\frac{1}{2}e_p$ cheap pendants. Therefore, the size of the largest label equals $e_d + \frac{1}{2}e_p$.

The algorithm is a greedy approach based on the traversal of the SPQR-tree and

the separate consideration of each skeleton. The main procedure of the approximation algorithm for PA_{Bic-2} is outlined in algorithm PlanarAugmentationBic-2. Procedure PA-Bic-2-Recursive manages the recursive calls such that a node is considered not until all its children are processed. Furthermore, there are several sub-procedures concerning the different types of SPQR-nodes: Handle-R-Node, Handle-S-Node, and Handle-P-Node, respectively.

Algorithm 9 PlanarAugmentationBic-2

Input: valid graph $G = (V, E)$ for PA_{Bic-2}
Output: list of new edges E' such that $G = (V, E \cup E')$ is planar and biconnected

1: construct the BC-tree $bc(G)$ and compute the labels
2: construct the SPQR-tree $\mathcal{T}$ of the biconnected core of G; root $\mathcal{T}$ at
   ↪ the node with the maximum degree
3: $\mu_r \leftarrow$ root of $\mathcal{T}$
4: PA-Bic-2-Recursive(μ_r)
5: fix the current embedding $\Pi(G)$ of G
6: **return** PlanarAugmentationFix(G, $\Pi(G)$)

In procedure PlanarAugmentationBic-2, the BC-tree is computed for obtaining the pendants and labels. Afterwards, the biconnected core of the graph is decomposed into its triconnected components. The traversal based on the SPQR-tree is a simple bottom-up approach with case differentiation between the types of tree-nodes and calls of the corresponding procedures in PA-Bic-2-Recursive, cf. lines 5–7. The actual augmentation takes place in the algorithm PlanarAugmentationFix which is called after the traversal and the fixing of the embedding, see lines 5 and 6.

Algorithm 10 PA-Bic-2-Recursive

Input: Node μ of the SPQR-tree $\mathcal{T}$

1: **for all** children ν of μ **do**
2: PA-Bic-2-Recursive(ν)
3: **end for**
4: **switch**(μ)
5: **case** R-node: Handle-R-Node(μ)
6: **case** S-node: Handle-S-Node(μ)
7: **case** P-node: Handle-P-Node(μ)
8: **end switch**

5.3.1 R-Node

The algorithm for R-nodes works in a greedy way and has two objectives. On the one hand, we try to transform as many demanding pendants as possible into potential pendants. On the other hand, we need to limit the number of negatively affected

Algorithm 11 HANDLE-R-NODE

Input: R-Node μ of the SPQR-tree $\mathcal{T}$

1: **let** $S_\mu = (V_\mu, E_\mu)$ be the skeleton-graph of μ with *dp*-values
   ↪ for each edge and additional R-pendants
2: $E'_\mu \leftarrow \{e \in E_\mu |$ one *dp*-value of e is $\neq (0,0)\}$ ▷ set of free edges
3: sort E'_μ by *dp*-value in descending order
4: **while** $(E'_\mu \neq \emptyset)$ **do**
5: $e \leftarrow$ first edge $\in E'_\mu$ with maximum demand e_d and potential e_p
6: $f_x \leftarrow$ adjacent face of e with the maximum number of pendants
7: FIXEMBEDDING()
8: update E'_μ
9: **end while**
10: MATCH-R-PENDANTS()
11: **if** (μ is not the root of $\mathcal{T}$) **then**
12: add *dp*-values to the virtual edge representing μ in *parent*(μ)
13: **end if**

pendants, i.e. pendants that are cheap in an optimum solution and expensive in the algorithmic solution as consequence of the embedding decisions.

Like described in the previous section, there are two types of pendants, the R-pendants and the demanding and potential pendants, respectively. The pendants of an R-label can be embedded separately into any adjacent face of the parent vertex. By contrast, demanding and potential pendants can only be embedded into the two adjacent faces of the edge. Furthermore, potential pendants can already be connected profitably whereas demanding pendants still require a matching partner.

The algorithm proceeds as follows. In every iteration, the edge with the largest demand is selected (line 5). Then, both adjacent faces are considered and the maximum number of pendants that can be embedded into each face is computed. For this computation, the demanding pendants are prioritized, i.e., potential pendants do not count unless both demanding values of an edge are equal. We define that a *dp*-value (d_1, p_1) is greater than a *dp*-value (d_2, p_2) iff $d_1 > d_2$ or, if $d_1 \geq d_2$ and $p_1 > p_2$. Furthermore, adjacent R-pendants are added to the face until they would become expensive.

The number of added pendants for each R-label can be decided on the basis of a comparison of the label-size and the number of pendants inside the face. Let r_i be the size of the i-th adjacent label, let R denote the number of R-pendants, and let k be the number of the remaining pendants, i.e., the sum of the demanding and potential pendants. Furthermore, assume that the labels are sorted in descending order such that r_1 is the size of the largest R-label. For each label with size r_j and $j > 1$, all pendants are added to the face. For the largest label, the number of added pendants is r_1, if $R - r_1 + k > r_1$ and $R + k$ is even. In case $R - r_1 + k > r_1$ and $R + k$ is odd, then $r_1 - 1$ pendants are added. If $R - r_1 + k \leq r_1$ then $R - r_1 + k$ pendants of the largest label are embedded into that face. Therefore, all added R-pendants would become cheap and there would be at most one adjacent label with free pendants after the embedding decisions.

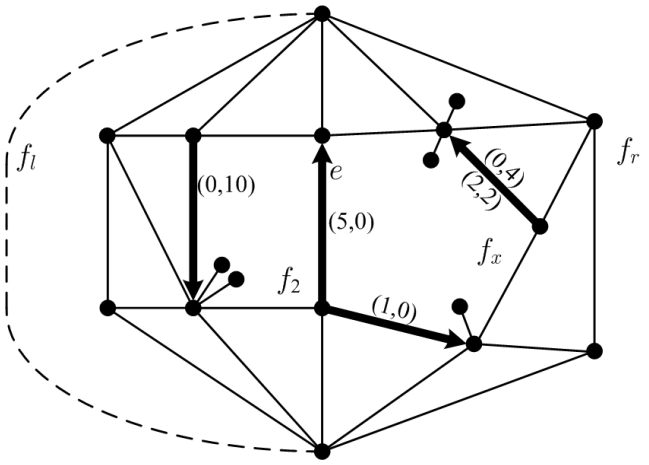


Figure 5.5: A skeleton of an R-node with corresponding *dp*-values, five R-pendants, and current edge e with demand 5. Since the number of demanding pendants in face f_x is 5 and in f_2 only 2, the edge is orientated such that the 5 demanding pendants are embedded into f_x. Notice that the R-label with size two is not added entirely to the face because otherwise, the total number of pendants would be odd.

After considering both adjacent faces the embedding is fixed by the procedure FixEmbedding which is not outlined. This procedure simply orientates e such that the demand e_d lies inside of face f_x which is the face with the prioritized set of pendants. Furthermore, all bordering edges of that face are fixed and deleted from the set of free edges E'_μ. Already fixed edges are naturally excepted. The involved R-labels are also being updated.

Figure 5.5 illustrates a typical situation concerning the current edge e with five demanding pendants and zero potential ones.

Algorithm 12 Match-R-Pendants

```
1: P ← set of free R-pendants in skeleton(μ)
2: generate a new graph H = (V_H, E_H) with |P| vertices, one for each
   ↪ pendant, add edges between pendants that are adjacent to the same
   ↪ face and do not belong to the same label
3: compute a maximum cardinality matching M in H
4: fix the embedding of the pendants according to M
5: if (μ is not the root of T) then
6:    if possible, embed free R-pendants into f_l or f_r (but no R-pendants
      ↪ with parents corresponding to one of the poles of skeleton(μ))
7: else
8:    embed free R-pendants arbitrarily
9: end if
```

After fixing the orientation of all edges and embedding the R-pendants, there might be remaining R-pendants. In particular, this is important for R-nodes that are leaves in the SPQR tree. Free R-pendants are processed like in the algorithm for PA_{Tric} by utilizing a maximum matching among the pendants of different labels

for the embedding. The procedure is outlined in algorithm Match-R-Pendants. In case there are still remaining R-pendants, they are embedded into the external faces f_l or f_r of μ, if possible. Since pole-pendants can be embedded arbitrarily in the parent node, they are excluded from this operation.

5.3.2 S-Node

Algorithm 13 Handle-S-Node

Input: S-Node μ of the SPQR-tree $\mathcal{T}$

1: **let** $S_\mu = (V_\mu, E_\mu)$ be the skeleton-graph of μ with the two faces f_l and f_r, ↪ *dp*-values for each edge, and additional S-Pendants
2: **for all** edges $e \in E_\mu$ **do**
3: **let** $(e_d, e_p), (e'_d, e'_p)$ be the *dp*-values of e with $(e_d, e_p) \geq (e'_d, e'_p)$
4: fix the embedding of e such that e_d lies in f_l
5: **end for**
6: embed S-pendants into the faces until they would remain demanding, start ↪ with f_l.
7: **if** (μ is not the root of $\mathcal{T}$) **then**
8: add *dp*-values to the virtual edge representing μ in $parent(\mu)$
9: **else**
10: embed free R-pendants arbitrarily
11: **end if**

The algorithm for orientating the edges in an S-skeleton and embedding the S-pendants is quite simple. Each edge is embedded such that the largest demand lies inside face f_l, cf. lines 2–4. The S-pendants are also fixed unless they would become expensive, compare line 6. All remaining S-pendants are unfixed and can be embedded in the parent node. Therefore, Handle-S-Node proceeds similar to Handle-R-Node.

Finally, the *dp*-values for the corresponding virtual edge in $parent(\mu)$ are computed. In case there exist free S-pendants, they count as demand on both sides. Furthermore, when the virtual edge is merged with another edge during Handle-P-Node, or an adjacent face is fixed in Handle-R-Node and there are still remaining S-pendants, they are embedded on the other side of the edge and the corresponding *dp*-value needs to be updated.

5.3.3 P-Node

The basic idea for P-skeletons is to consider the edges in decreasing order with respect to the *dp*-value. Therefore, the algorithm selects the two edges with the largest demanding values. Both edges are merged, that is they are fixed to be consecutive in the order of the edges of the P-skeleton. Furthermore, as many P-pendants as possible are added to that face. This procedure is outlined in P-Node-Merge.

Algorithm 14 HANDLE-P-NODE

Input: P-Node μ of the SPQR-tree $\mathcal{T}$

1: **let** $S_\mu = (V_\mu, E_\mu)$ be the skeleton-graph of μ with *dp*-values ↪ for each edge and additional P-pendants
2: $E'_\mu \leftarrow E_\mu$
3: **while** $(|E'_\mu| \neq 1)$ **do**
4: $e, e' \leftarrow$ the two edges in E'_μ with the largest and second largest ↪ *dp*-values (e_d, e_p) and (e'_d, e'_p)
5: P-NODE-MERGE()
6: **end while**
7: **if** (μ is not the root of $\mathcal{T}$) **then**
8: add *dp*-values to the virtual edge representing μ in $parent(\mu)$
9: **end if**

Algorithm 15 P-NODE-MERGE

1: $e^* \leftarrow$ merge e and e' such that e_d and e'_d are embedded into the new face
2: add free P-pendants to the new face until they would become expensive
3: if possible, embed remaining S-pendants into the external faces of the ↪ new component
4: compute new *dp*-values for both sides of e^*
5: $E'_\mu \leftarrow \{E'_\mu \backslash \{e, e'\}\} \cup \{e^*\}$

5.4 Approximation Ratio

Theorem 5.5. *Algorithm* PLANARAUGMENTATIONBIC-2 *solves* PA_{Bic-2} *with at most* $\frac{5}{3}$ *times the number of edges of an optimum solution.*

Proof. Let $G = (V, E)$ be an instance for PA_{Bic-2}, $bc(G)$ the corresponding BC-tree, and $\mathcal{P}$ the set of pendants in $bc(G)$. Let $\mathcal{T}$ be the SPQR-tree of the biconnected core of G. Furthermore, let $\mathcal{S}$ denote the solution computed by the algorithm and $\mathcal{S}_{opt}$ an optimum one.

The proof is based on an induction over the ordered sequence of embedding decisions of the algorithm.

Analogously to the algorithm, the analysis considers a current subset of pendants which are already embedded. During the algorithm there are complete and incomplete faces. A complete face lies either inside an R-skeleton and has already been fixed, or is a face that results from a merge step of P-NODE-MERGE, or is a fixed face of the skeleton that corresponds to the root node. The contained pendants can be classified, depending on whether they are cheap or expensive. In general, pendants that are embedded into incomplete faces cannot be classified entirely until the embedding is fixed.

The value of an augmentation is its number of added edges. Since every pendant is either connected by an profitable or an unprofitable edge, the evaluation can be changed to a simple accounting method. Each pendant becomes a cost unit and is charged with costs $\frac{1}{2}$ or 1, depending on whether it is cheap or expansive. Thus, the

costs of an added edge are partitioned among the incident pendants, and the sum of all pendants equals the number of required edges for augmentation.

Let c be the cost-vector of the constructed and c^* the one of the optimum solution. Furthermore, let $costs(\mathcal{P})$ and $costs^*(\mathcal{P})$, respectively, denote the sum of the cost-vector of the corresponding solution. If a pendant p is expensive in $\mathcal{S}$ and cheap in $\mathcal{S}_{opt}$ then we call p a *bad* pendant.

Each embedding decision causes some pendants to become cheap and some pendants are affected negatively, since they cannot be connected profitably anymore. Since the algorithm works on the triconnected structure of the graph, an upper bound of affected pendants can be estimated easily. Although the embedding decisions are the reason for the occurrence of bad pendants, we say that the pendants of the current face are responsible for the affected pendants since this number is always an upper bound. Therefore, each pendant is virtually charged with the number of bad pendants it causes.

The overall idea is to consider each decision, estimate an upper bound of negatively affected pendants, and show that the number of bad pendants can be compensated by the number of cheap ones. We will prove that x cheap pendants in $\mathcal{S}$ cause at most $2x$ bad pendants. All other pendants $P' \subseteq \mathcal{P}$ are expensive in $\mathcal{S}_{opt}$ and hence, the costs of the constructed solution cannot be worse than the costs of the optimum solution for this set. Thus, the costs of $\mathcal{S}$ are at most $\frac{1}{2}x + 2x + costs^*(P') = 2.5x + costs^*(P')$ and the one of $\mathcal{S}_{opt}$ are at least $\frac{1}{2}x + x + costs^*(P') = 1.5x + costs^*(P')$. Therefore the ratio is at most

$$\frac{2.5x + costs^*(P')}{1.5x + costs^*(P')} = 1 + \frac{x}{1.5x + costs^*(P')} \leq 1 + \frac{2}{3} = \frac{5}{3}.$$

The induction is based on the sequence of embedding decisions and the induction hypothesis is a set of statements:

- Potential pendants are already being considered.
- Demanding pendants have not caused any bad pendants.
- Bad pendants in the current face caused by previous decisions are already considered and can be compensated.

Base case:
The base case occurs in a node μ of the SPQR-tree with the property, that all label-parents in $pertinent(\mu)$ also belong to $skeleton(\mu)$. Then, all dp-values are zero and the algorithms for R-, S-, and P-nodes only have to deal with R-, S-, and P-skeletons, respectively.

Since the embedding decisions only concern the current skeleton graph, two pendants that belong to $G - pertinent(\mu)$ and which could have been connected previously can also be connected afterwards.

μ is an R-Node:
In the base case, the algorithm for R-nodes proceeds like the algorithm for PA_{Tric} by computing a maximum matching M between the R-pendants of different labels and

embedding the pendants according to this matching. In case the current R-node is also the root of $\mathcal{T}$, the augmentation based on the maximum matching equals an optimum one.

Therefore, assume that the current node is a leaf in the SPQR-tree. Then, unmatched pendants—but not free pole-pendants—are embedded into the external faces. Each matched pendant becomes either cheap, or it becomes potential if it lies in the external face of μ. By setting the property of being potential for some pendants they become virtually connected by profitable edges. Although the external face is unfinished, it is valid to assume that the potential pendants do not lose this property. An augmentation of a face can always be modified such that the solution is still optimal and all expensive pendants were demanding.

Consider two matched pendants which are embedded into an external face of μ. Afterwards, both pendants become potential. Assume that the computed matching is not optimal with respect to the augmentation of the entire graph and that one of the pendants would better be embedded inside the R-skeleton. Therefore, this pendant causes one other pendant to become bad. In the following embedding decisions, this potential pendant cannot be held responsible for another bad pendant. This has two reasons. First, potential pendants are less valuable than demanding pendants. Therefore, in each embedding decision, the side of an edge with the maximum number of demanding pendants is prioritized and embedded first. Second, each pendant can only be connected to one other pendant. Therefore, if one potential pendant already causes one bad pendant, i.e., it would be better connected to it, this pendant cannot be charged with another bad pendant in the future.

This observation is essential for the proof and it does not only hold for the base case in the R-node. It is also a valid observation in the inductive step.

In case the potential pendants do not cause bad pendants inside the current skeleton, this can still occur in the parent skeleton. But the same arguments like before will ensure that each potential pendant is responsible for at most one other bad pendant.

Altogether, a matching with size $|M|$ induces $2|M|$ cheap pendants and at most $2|M|$ negatively affected pendants.

μ is an S-Node:
The algorithm for S-nodes also computes a maximum matching on the pendants of the different labels by embedding pendants into the external face f_l until they would remain demanding. In case μ is also the root, the computed maximum matching induces an optimum solution. Otherwise, the remaining S-pendants are not fixed and can be reused in ancestor nodes. The arguments concerning the potential pendants are similar to the arguments for the R-node. Furthermore, the potential pendants cannot be separated anymore, i.e. they can only be embedded into one adjacent face of the parent skeleton. Therefore, the number of affected pendants is at most the number of new potential pendants.

μ is a P-Node:
In case of a P-skeleton, the algorithm fixes the permutation of the edges. Here, no child contains any pendants and therefore, the permutation is irrelevant. P-pendants

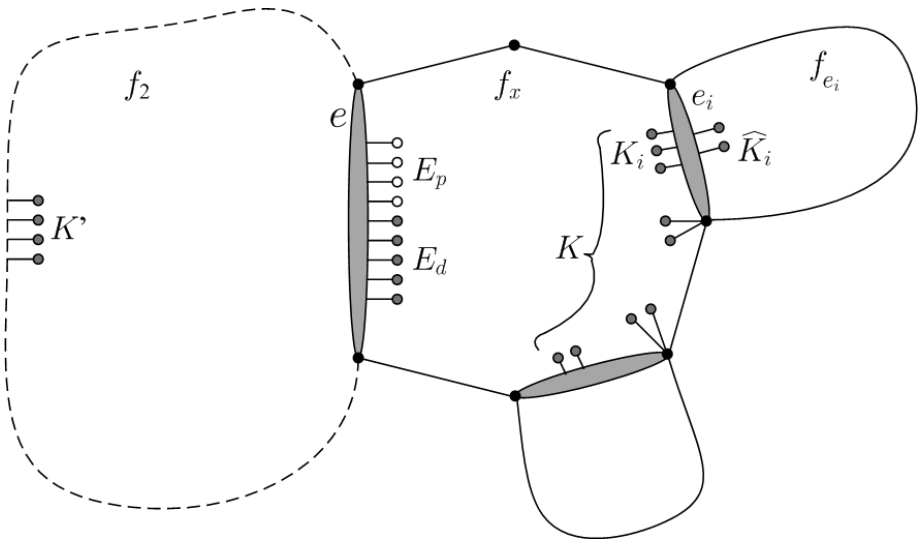


Figure 5.6: Sketch of the embedding situation concerning the virtual edge e with pendant set $E_d \cup E_p$. The grey-colored vertices are demanding vertices whereas the white ones are potential. Face f_x contains k and face f_2 k' pendants.

are matched during the first merge step and remaining pendants are unfixed. Therefore, the same arguments like before hold and the number of affected pendants equals at most the number of matched pendants.

If the current P-node is the root, the computed embedding depends on the maximum matching and is therefore optimal.

Inductive step:
Since the SPQR-tree has only height one, the inductive step deals with a skeleton that corresponds to the root of the tree. We can assume that the statements hold for the first i decisions, with $i \geq 2$. Again, let μ be the current node of the SPQR-tree.

Case 1*: μ is an R-Node*:
Let e be the current virtual edge with maximum demand e_d and potential e_p, and let E_d and E_p denote the corresponding pendant sets. In case the demand is zero all pendants embedded into the current face are potential pendants or R-pendants. The potential pendants can already be connected profitably and they are already considered. The R-pendants are embedded unless they would become expensive. This approach induces a maximal matching on the R-pendants and the same arguments like in the base case can be applied.

Now, assume that $e_d > 0$. Let $K = K_p \cup K_d$ be the set of possible pendants in f_x and f_2 the opposite face of e containing the set K' with k' pendants. K_p denotes the set of potential and K_d the one of demanding pendants. We will refer to the i-th virtual edge in cyclic order bordering f_x as e_i and the opposite face of e_i as f_{e_i}. Most of the used identifiers are illustrated in Figure 5.6.

The embedding decisions made by Handle-R-Node are not critical with respect to the embedding of the whole graph. Two pendants that belong to non-adjacent expansion graphs of the current face and that can be connected before each decision

are not affected. Since the number of demanding pendants in the adjacent faces of the current edge is known, it is possible to estimate an upper bound for the number of affected pendants.

First of all, we will compute an upper bound of negatively affected pendants in other faces caused by the bounding pendants of f_x. Consider edge e_i and its pendant set $K_i = K_{i_p} \cup K_{i_d}$ in f_x with potential pendants K_{i_p} and demanding pendants K_{i_d}. Let the opposite face of e_i be f_{e_i} and $\widehat{K}_i = \widehat{K}_{i_p} \cup \widehat{K}_{i_d}$ the opposite pendant set attached to e_i. The cardinality of each set is denoted by a small character with the corresponding index, e.g., $\widehat{k}_{i_d} = |\widehat{K}_{i_d}|$.

Since the edge e_i is embedded such that the maximum demand lies inside f_x, $k_{i_d} \geq \widehat{k}_{i_d}$ holds. Obviously, it is not possible to embed the sets K_i and $\widehat{K}_i$ into one face at the same time. Moreover, one pendant can only be connected to one other. Hence, there are at most k_{i_d} affected pendants located at expansion graphs adjacent to f_{e_i}, for all i.

In case edge e_i has already been fixed by a previous decision, the current embedding has no effect on that face. The same argument holds for the opposite face of e.

However, in the unfixed case, face f_2 contains at most $k' \leq k$ demanding pendants, excluding the pendants attached to e whose number is at most e_d. Like before, both sides of e are always separated. The same arguments like before ensure that at most $\min\{k_d, e_d\}$ pendants are affected in expansion graphs adjacent to f_2.

Moreover, potential pendants are already being considered. Therefore, there are at most $k_d + \min\{k_d, e_d\}$ new bad pendants located in other faces caused by the current decision.

Second of all, we will consider the current face f_x and the number of contained bad pendants. Since e_d is the maximum demand, $k_{i_d} \leq e_d$ holds for every i. The face can be augmented optimally such that all expensive pendants belong to the set E_d. In case $k_p + k_d > e_d$ there are exactly $((k_d + e_d) \mod 2)$ expansive pendants[1]. Therefore, this case induces at most one bad pendant in f_x caused by the current decision.

Otherwise, if $k_p + k_d \leq e_d$ holds, the number of expensive pendants equals $e_d - (k_p + k_d)$. Although the current face contains the maximum number of demanding pendants, k might not be the maximal possible number of pendants in f_x. Consider an edge e_i with the described pendant sets K_i and $\widehat{K}_i$. It is possible that k_i is small and $\widehat{k}_i > k_i$. The same argument holds for face f_2. It is possible that the number of pendants in f_2 is larger than in f_x. But since these pendants can only be potential, they have already been considered. Therefore, all expensive pendants among the set E_d are also expensive in $\mathcal{S}_{opt}$ or they are affected by previous decisions. Hence, in case $k_p + k_d \leq e_d$, there are no new bad pendants.

Altogether, there are

$$k_d + \begin{cases} k_p + k_d & \text{if } k_p + k_d < e_d \\ e_d & \text{if } k_p + k_d = e_d \\ e_d - ((k_d + e_d) \mod 2) & \text{if } k_p + k_d > e_d \end{cases}$$

[1] mod denotes the modulo-operation, that is $a \bmod b$ denotes the remainder after the division of a by b

new cheap pendants and at most

$$k_d + \begin{cases} k_d & \text{if } k_p + k_d < e_d \\ k_d & \text{if } k_p + k_d = e_d \\ e_d + ((k_d + e_d) \mod 2) & \text{if } k_p + k_d > e_d \end{cases}$$

affected pendants.

In case $k_p + k_d$ is less than or equal to e_d the ratio of cheap and bad pendants is at most 1. The third case, i.e., $k_p + k_d > e_d$, leads to the worst-case ratio of the algorithm:

$$\frac{k_d + e_d - ((k_d + e_d) \mod 2)}{k_d + e_d + ((k_d + e_d) \mod 2)} \leq \frac{k_d + e_d - 1}{k_d + e_d + 1}$$

Since $e_d \geq 1$ and therefore, $k_d \geq 2$ holds, the worst-case occurs if $e_d = 1$ and $k_d = 2$. Then, the number of bad pendants is two times the number of cheap pendants, namely four versus two.

Case 2: μ is an S-Node:
The algorithm for S-nodes simply orientates all edges such that the maximum demand is embedded into face f_l. Moreover, S-pendants are additionally fixed in both faces until they would remain demanding. Finally remaining S-pendants are embedded arbitrarily because they cannot be connected profitably anyway. Therefore, the algorithm works exactly like the one for the R-node and the analysis is similar.

Case 3: μ is an P-Node:
HANDLE-P-NODE always selects the two edges with the largest *dp*-values (e_d, e_p) and (e'_d, e'_p) and merges both edges into a new component. The two edges are orientated such that the largest demands are embedded into the inner face. Like in the two previous cases, the P-pendants are added to this face until they would become expensive. Then, the new component is considered as a virtual edge. Therefore, all other edges of μ can still be merged arbitrarily and the new edge can also be selected for following merge operations.

Since all other demanding values are at most e'_d the same arguments hold like for the R-node case.

□

The previous analysis of the approximation ratio is tight. Figure 5.7 illustrates a worst-case graph with six pendants where the algorithm computes an augmentation containing five edges. By contrast, three edges are already sufficient.

The corresponding SPQR-tree consists of four R-nodes, three of them, which are also the leaves, contain one pendant each, and the root contains the remaining three pendants. For each leaf, the algorithm embeds the single pendant into the adjacent external face of the skeleton and the corresponding virtual edge obtains one demanding pendant on one side. In the root skeleton one of these virtual edges is selected as edge with the maximum demand. Since the number of pendants in the inner face is three, each virtual edge is orientated such that the demanding pendant is inserted into this face. Therefore, only two pendants can be connected profitably,

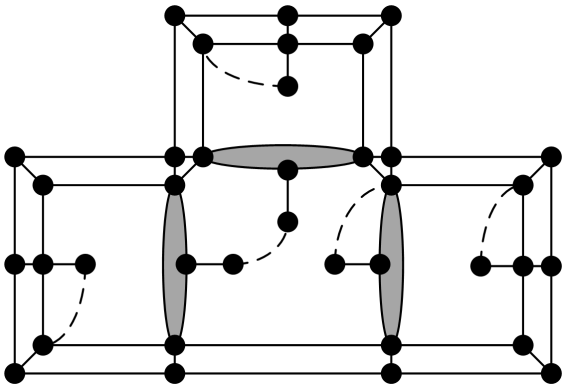


Figure 5.7: A worst-case example for PA_{Bic-2} with dashed edges representing the computed augmentation.

whereas four are expansive. By orientating all three virtual edges contrarily, the graph can be augmented with three edges.

5.5 Running Time

Theorem 5.6. *Algorithm* PlanarAugmentationBic-2 *runs in time* $\mathcal{O}(|V|^{2.5})$.

Proof. The SPQR-tree and the BC-tree can be constructed in linear time. Furthermore, each skeleton of the SPQR-tree is expanded by the related pendants. Since the height of the tree at most one, each pendant is represented by at most two instances.

The procedures Handle-P-Node and Handle-S-Node do not have any expensive operations that could exceed linear running time. Each sorting can be handled by a bucket sort routine in linear time. The computation of the *dp*-value can also be done in linear time in the number of adjacent labels.

In the R-skeleton, R-pendants are embedded by computing a maximum matching. The running time of a matching algorithm is $\mathcal{O}(\sqrt{n}m)$, if n is the number of vertices and m the number of edges in the underlying graph. Here, the graph is the auxiliary graph $H = (V_H, E_H)$. The vertices of H represent the R-pendants and the edges are inserted between pendants of different labels. Therefore, the cardinality of E_H can be $\Omega(|V_H|^2)$, leading to a running time of $\mathcal{O}(|V_H|^{2.5})$ for Match-R-Pendants.

The final augmentation takes time $\mathcal{O}(|V|+|E|)$ and hence, the total running time is dominated by the computation of the matching and is bounded by $\mathcal{O}(|V|^{2.5})$.

□

Chapter 6

Summary and Outlook

In this thesis we have dealt with the *Planar Augmentation Problem* and with several special cases of this problem. In Chapter 3, we considered known approximation algorithms for the general case, among them the approach of Kant and Bodlaender with ratio two [31] and the one of Fialko and Mutzel with ratio $\frac{5}{3}$ [12]. We constructed a counter-example for the latter algorithm showing that its approximation ratio is only two. Before, Fialko and Mutzel already detected problems in an approach of Kant and Bodlaender that should have ratio $\frac{3}{2}$ and actually achieves only ratio two. The problem of the previous approaches is the unpredictability of the effect of one added edge on the whole embedding of the graph. As shown by the worst-case example in Section 3.4, it is possible that one edge affects the embedding such that all other pendants cannot be connected profitably anymore because of planarity.

In case the instances are restricted to planar graphs with the constraint that all cutvertices belong to one triconnected component (PA_{Tric}), the problem becomes efficiently solvable, cf. Section 5.1 or [31]. Since the general problem is $\mathcal{NP}$-hard and PA_{Tric} is easily solvable, we investigated the *Planar Augmentation Problem* on graphs with one biconnected component containing all cutvertices (PA_{Bic}). Surprisingly, this special case is already $\mathcal{NP}$-hard. In Chapter 5, we presented a new polynomial-time reduction from the *Planar Vertex Cover Problem* to PA_{Bic}. As consequence, the problem remains $\mathcal{NP}$-hard even for the case, where the SPQR-tree (without Q-nodes) of the biconnected core has height one (PA_{Bic-2}).

Since the core of the input graph is biconnected, this restricted problem allows the direct use of the SPQR-tree. The advantages of this approach are due to the properties of the skeleton graphs, i.e.,the effects of one added edge—or in this case of one embedding decision—on the whole graph are bounded. Hence, we successfully developed an approximation algorithm with ratio $\frac{5}{3}$. The running time of this algorithm is $\mathcal{O}(|V|^{2.5})$.

Before, in Chapter 4, we considered another special case. By fixing the embedding of a planar graph (PA_{Fix}), its optimal augmentation becomes efficiently computable. We presented a new algorithm with running time $\mathcal{O}(|V|+|E|+\alpha(|V|)|V|)$. Thus, the running time is linearly bounded for all practical purposes. The approach relies on the idea of adding edges that fulfill the leaf-connection condition by utilizing

b- and c-labels introduced by Fialko and Mutzel in [12].

Through this thesis, we got a better insight into the complexity of the *Planar Augmentation Problem* and the complications concerning the development of new approximation algorithms. There are some open questions and unsolved problems, but also potential approaches for future work, e.g.:

- As described in Section 3.5, a disconnected graph cannot be connected easily without potentially destroying the optimum solution for the whole augmentation. This is an open problem which seems to be as complex as the main problem.
- The approximation algorithm for PA_{Bic-2} can possibly be improved to a $\frac{3}{2}$-approximation, since this is the ratio which is provided by the maximal matching on the pendants. Maybe there exists a completely different approach that leads to a better approximation ratio.
- Furthermore, the approximation of PA_{Bic} is now an open problem. Perhaps, the described algorithm can be be expanded to arbitrary heights of the related SPQR-tree without increasing the approximation ratio.
- The main objective remains the development of a new approximation algorithm for the general *Planar Augmentation Problem* with a ratio less than two. At first view, this seems to be feasible since the first approximation algorithm of Kant and Bodlaender with ratio two ignores profitable edges completely.

  The related question is whether the *Planar Augmentation Problem* is approximable with a ratio less than two, or not.

List of Figures

List of Algorithms

Bibliography

[1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing – Algorithms for the visualization of graphs*. Prentice Hall, 1999.

[2] G. Di Battista and R. Tamassia. Incremental planarity testing. In *30th Annual Symposium on Foundations of Computer Science*, pages 436–441. IEEE, 1989.

[3] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996.

[4] D. Bienstock and C. L. Monma. On the complexity of covering vertices by faces in a planar graph. *SIAM Journal on Computing*, 17(1):53–76, 1988.

[5] J. M. Boyer and W. J. Myrvold. On the cutting edge: simplified $O(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(2):241–273, 2004.

[6] M. Chimani, P. Mutzel, and J. M. Schmidt. Efficient extraction of multiple Kuratowski subdivisions. In *Lecture Notes in Computer Science*, volume 4875, pages 159–170. Springer Verlag, 2007.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM Symposium on Theory Of Computing*, pages 151–158. ACM, 1971.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Second edition, 2001.

[9] R. Diestel. *Graph Theory*. Springer-Verlag, Third edition, 2005.

[10] I. Dinur and S. Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162:439–485, 2005.

[11] K. Eswaran and R. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976.

[12] S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on discrete algorithms*, pages 260–269. Society for Industrial and Applied Mathematics, 1998.

[13] A. Frank. Augmenting graphs to meet edge-connectivity requirements. *SIAM Journal on Discretete Mathematics*, 5(1):25–53, 1992.

[14] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[15] G. Frederickson and J. Jájá. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10(2):270–283, 1981.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[17] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[18] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[19] D. Gusfield. Optimal mixed graph augmentation. *SIAM Journal on Computing*, 16(4):599–612, 1987.

[20] C. Gutwenger and P. Mutzel. Grid embedding of biconnected planar graphs. Extended Abstract, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1997.

[21] C. Gutwenger and P. Mutzel. Planar polyline drawings with good angular resolution. In *Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 1998.

[22] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer-Verlag, 2000.

[23] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005.

[24] J. Hopcroft and R. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[25] J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[26] T.-S. Hsu. On four-connecting a triconnected graph. *Journal of Algorithms*, 35(2):202–234, 2000.

[27] T.-S. Hsu and V. Ramachandran. On finding a smallest augmentation to biconnect a graph. *SIAM Journal on Computing*, 22(5):889–912, 1993.

[28] B. Jackson and T. Jordán. Independence free graphs and vertex connectivity augmentation. *Journal of Combinatorial Theory Series B*, 94(1):31–77, 2005.

[29] M. Jünger and P. Mutzel. *Graph Drawing Software*. Springer-Verlag, 2004.

[30] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.

[31] G. Kant and H. L. Bodlaender. Planar graph augmentation problems (extended abstract). In *Algorithms and Data Structures, 2nd Workshop WADS '91*, volume 519 of *Lecture Notes in Computer Science*, pages 286–298. Springer-Verlag, 1991.

[32] G. Karakostas. A better approximation ratio for the vertex cover problem. In *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 1043–1050. Springer-Verlag, 2005.

[33] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[34] M. Kaufmann and D. Wagner. *Drawing Graphs: Methods and Models*. Springer-Verlag, 2001.

[35] K. Kuratowski. Sur le problème des corbes gauches en topologie. In *Fundamenta Mathematicæ*, pages 271–283, 1930.

[36] S. Micali and V. V. Vazirani. An $\mathcal{O}(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science*, pages 17–27. IEEE, 1980.

[37] A. Rosenthal and A. Goldner. Smallest augmentations to biconnect a graph. *SIAM Journal on Computing*, 6(1):55–66, 1977.

[38] D. Soroker. Fast parallel strong orientation of mixed graphs and related augmentation problems. *Journal of Algorithms*, 9(2):205–288, 1988.

[39] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[40] W. T. Tutte. *Connectivity in graphs*, volume 15 of *Mathematical Expositions*. University of Toronto Press, 1966.

[41] T. Watanabe and A. Nakamura. A smallest augmentation to 3-connect a graph. *Discrete Applied Mathematics*, 28(2):183–186, 1990.

[42] I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer-Verlag, 2005.

[43] R. Weiskircher. *New applications of SPQR-trees in graph drawing*. PhD thesis, Universität des Saarlandes, 2002.

[44] J. Westbrook and R. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(5&6):433–464, 1992.