
**Application of SPQR-Trees
in the Planarization Approach
for Drawing Graphs**

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Carsten Gutwenger

Dortmund

2010

Tag der mündlichen Prüfung:
20.09.2010

Dekan:
Prof. Dr. Peter Buchholz

Gutachter:
Prof. Dr. Petra Mutzel, Technische Universität Dortmund
Prof. Dr. Peter Eades, University of Sydney

To my daughter Alexandra.

Contents

Abstract	vii
Zusammenfassung	viii
Acknowledgements	ix
1 Introduction	1
1.1 Organization of this Thesis	4
1.2 Corresponding Publications	5
1.3 Related Work	6
2 Preliminaries	7
2.1 Undirected Graphs	7
2.1.1 Paths and Cycles	8
2.1.2 Connectivity	8
2.1.3 Minimum Cuts	9
2.2 Directed Graphs	9
2.2.1 Rooted Trees	9
2.2.2 Depth-First Search and DFS-Trees	10
2.3 Planar Graphs and Drawings	11
2.3.1 Graph Embeddings	11
2.3.2 The Dual Graph	12
2.3.3 Non-planarity Measures	13
2.4 The Planarization Method	13
2.4.1 Planar Subgraphs	14
2.4.2 Edge Insertion	17
2.5 Artificial Graphs	19
3 Graph Decomposition	21
3.1 Blocks and BC-Trees	22
3.1.1 Finding Blocks	22
3.2 Triconnected Components	24
3.3 SPQR-Trees	31
3.4 Linear-Time Construction of SPQR-Trees	39
3.4.1 Split Components	40
3.4.2 Primal Idea	42
3.4.3 Computing SPQR-Trees	44
3.4.4 Finding Separation Pairs	46

3.4.5	Finding Split Components	58
3.4.6	Corrections on the Hopcroft and Tarjan Algorithm	61
3.4.7	Computational Experiments	63
4	Crossing Minimization	73
4.1	The One-Edge Insertion Problem	74
4.1.1	Traversing Costs	76
4.1.2	Biconnected Graphs	78
4.1.3	General Graphs	84
4.1.4	Near-Planar Graphs	86
4.2	Crossing Minimization Heuristics	88
4.2.1	Preprocessing	89
4.2.2	Planar Subgraphs	96
4.2.3	Edge Re-Insertion	97
4.3	Experimental Analysis	100
4.3.1	Results for the Rome Graphs	102
4.3.2	Results of the Artificial Graphs	122
4.3.3	Summary	124
5	Graph Embedding	127
5.1	Maximum External Face	131
5.1.1	Biconnected Graphs	132
5.1.2	Connected Graphs	138
5.2	Minimum Block-Nesting Depth	141
5.3	Minimum Block-Nesting Depth and Maximum External Face	146
5.4	Experimental Analysis	147
5.4.1	The Rome Graphs	148
5.4.2	The Block Graphs	149
5.4.3	Summary	153
6	Embedding Constraints	155
6.1	ec-Constraints and ec-Planarity	157
6.2	ec-Expansion	158
6.2.1	Construction of the ec-Expansion	158
6.2.2	ec-Expansion and ec-Planar Embeddings	159
6.3	ec-Planarity Testing	160
6.4	ec-Edge Insertion	163
6.4.1	ec-Edge Insertion Paths and ec-Traversing Costs	163
6.4.2	The Algorithm for Biconnected Graphs	164
6.4.3	Correctness and Optimality	168
6.4.4	Generalization to Connected Graphs	171
6.5	Summary and Discussion	173
7	Conclusion	175
	Bibliography	179

Abstract

The planarization method is the most successful practical approach for minimizing the number of crossings in a drawing of a graph and, when used as the first step of the topology-shape-metrics approach, one of the best methods to draw sparse graphs in general. In this thesis, we apply BC- and SPQR-trees, that is, data structures for decomposing a graph with respect to its bi- and triconnected components, for improving the planarization method in several aspects.

First of all, we present a corrected version of a linear-time algorithm for finding the triconnected components of a graph and apply it to SPQR-trees, thus giving the first correct linear-time implementation for constructing SPQR-trees. Then, we apply BC- and SPQR-trees for solving the one-edge insertion problem (OEIP) in linear time. The OEIP can be stated as follows: Given a planar graph G and two vertices v, w in G , find an embedding of G in which we can draw the edge (v, w) with a minimum number of crossings. Our algorithm thus solves a long-standing open problem in graph drawing. Applying the OEIP within the planarization approach already improves this crossing minimization method substantially. Additionally, we introduce further pre- and postprocessing methods and experimentally analyze their performance with a well-known benchmark set of graphs, as well as a new one containing graphs with known crossing numbers. Our study shows that the crossing minimization heuristic comes very close to the optimal results in most cases, or even finds the optimum.

Apart from being a tool for crossing minimization, the planarization method also provides a planar embedding which is used as input for a planar graph drawing algorithm. However, the choice of the planar embedding is essential for the quality of the resulting drawing, for example, with respect to number of bends or total edge length. We propose properties of good planar embeddings, namely a minimal block-nesting depth and an external face of maximum degree, and present linear-time algorithms for computing such planar embeddings. Experimental evaluations suggest that using such embeddings can improve the quality of the resulting layouts.

Finally, we extend the planarization method by incorporating embedding constraints, that is, constraints imposed on the order of incident edges around a vertex. Such constraints typically occur in practical applications; for example, we can compute planar embeddings respecting side constraints (each incident edge is assigned to one of the four sides of the vertex) or port constraints. We give linear-time algorithms for both planarity testing and edge insertion with embedding constraints, thus yielding the same asymptotic running times as in the unconstrained case.

Zusammenfassung

Die Planarisierungsmethode ist in der Praxis der erfolgreichste Ansatz zur Minimierung von Kreuzungen in Zeichnungen von Graphen. Sie wird insbesondere auch als erster Schritt im so genannten *Topology-Shapes-Metrics*-Verfahren verwendet, was damit eines der besten Verfahren zum Zeichnen von dünnen Graphen darstellt. In dieser Doktorarbeit benutzen wir BC- und SPQR-Bäume — Datenstrukturen zur Zerlegung eines Graphen in seine Zwei- und Dreizusammenhangskomponenten — zur Verbesserung der Planarisierungsmethode in mehrerlei Hinsicht.

Als Erstes präsentieren wir eine korrigierte Version eines Linearzeitalgorithmus, der die Dreizusammenhangskomponenten eines Graphen finden kann, und wenden diesen auf die Konstruktion von SPQR-Bäumen an. Die daraus resultierende Implementierung ist damit die erste korrekte Linearzeitimplementierung zur Konstruktion von SPQR-Bäumen. Danach benutzen wir BC- und SPQR-Bäume, um das Eine-Kante-Einfügeproblem in Linearzeit zu lösen. Dieses Problem lässt sich wie folgt formulieren: Gegeben sei ein planarer Graph G und zwei Knoten v und w aus G , finde eine Einbettung von G , in die wir die Kante (v, w) mit der kleinstmöglichen Anzahl an Kreuzungen einzeichnen können. Der hier vorgestellte Algorithmus löst damit ein seit langem bestehendes offenes Problem im Graphenzeichnen. Durch Anwendung dieses Algorithmus in der Planarisierungsmethode kann die Kreuzungsminimierung bereits erheblich verbessert werden. Zusätzlich stellen wir weitere Vor- und Nachbearbeitungsmethoden vor und analysieren diese experimentell mit Hilfe bekannter Benchmarkgraphen und einer neuen Benchmark-Bibliothek von Graphen mit bekannten Kreuzungszahlen. Unsere Studie zeigt, dass die daraus resultierenden Heuristiken zur Kreuzungsminimierung sehr nahe an das Optimum herankommen und es in vielen Fällen auch erreichen.

Neben der Minimierung von Kreuzungen an sich liefert die Planarisierungsmethode als Ergebnis auch eine planare Einbettung, die als Eingabe für planare Zeichenalgorithmen dient. Allerdings ist die richtige Wahl der planaren Einbettung entscheidend für die Qualität der resultierenden Zeichnung, zum Beispiel im Hinblick auf Anzahl der Knicke oder Kantenlängen. Daher schlagen wir zwei einfache Kriterien für planare Einbettungen vor, die zu guten Zeichnungen führen können: Eine minimale Verschachtelungstiefe der Blöcke des Graphen und eine Außenfläche maximalen Grades. Wir stellen für beide Kriterien optimale Linearzeitalgorithmen vor. Experimentelle Studien zeigen, dass diese planaren Einbettungen tatsächlich zu besseren Zeichnungen führen können.

Abschließend erweitern wir die Planarisierungsmethode um Einbettungseinschränkungen, welche die mögliche Reihenfolge der Kanten um einen Knoten beschränken. Solche Einschränkungen findet man häufig in praktischen Anwendungen, wie etwa bei der Modellierung von *Side Constraints* und *Port Constraints*. Wir präsentieren Linearzeitalgorithmen für das Testen auf Planarität und optimale Einfügen einer Kante unter Beachtung der Einbettungsbeschränkungen.

Acknowledgements

Many people supported and encouraged me during the time in which I did research for and worked on this thesis. I wish to express here my gratitude to all of them.

First of all, I wish to thank my advisor Petra Mutzel for introducing me to the field of graph drawing. It was her lecture on *Automatic Graph Drawing* at Saarland University that raised my interest in graph drawing and the beautiful theory behind it. She gave me the opportunity to work on AGD, our first C++ library for automatic graph drawing, and the right balance of scientific guidance and freedom to do my research. She also made it possible that I could attend numerous conferences and workshops all over the world, which I enjoyed a lot.

I also want to thank the people at the Chair XI for Algorithm Engineering at the Technische Universität Dortmund for the highly enjoyable atmosphere and fruitful discussions with my colleagues. In particular, I thank my roommate Karsten Klein and the other people of Petra Mutzel's research group, Markus Chimenti, Maria Kandyba, Wolfgang Paul, Hoi-Ming Wong, and Bernd Zey, as well as our secretary Gundel Jankord for taking so much administrative work from our shoulders. I also want to thank my further co-authors for sharing their knowledge with me. I am especially grateful to Sebastian Leipert, Michael Schulz, and René Weiskircher. I am particularly grateful to Michael Jünger who was leading the project group at caesar in Bonn, sharing my vision of bringing graph drawing software into real-world applications. And thank you for proof-reading parts of my thesis, Bernd and Karsten.

I want to thank Prof. Müller, Petra, Peter Eades, and Stefan Dissmann for their immediate willingness to serve on my defense commission. In particular, I want to thank Peter for making it possible to attend my defense.

Last but not least, I thank my parents for giving me the opportunity and the freedom to study computer science, and Tatjana for supporting me at all times.

Chapter 1

Introduction

Two things are infinite: the universe and human stupidity; and I'm not sure about the universe.

ALBERT EINSTEIN (1879 – 1955)

Reinhard Diestel starts the chapter on *Planar Graphs* in his famous book on *Graph Theory* [Diestel, 2005] with the following statement:

“When we draw a graph on a piece of paper, we naturally try to do this as transparently as possible. One obvious way to limit the mess created by all the lines is to avoid intersections. For example, we may ask if we can draw the graph in such a way that no two edges meet in a point other than a common end.”

Of course, not all graphs can be drawn without any such line intersections. We call the graphs that can be drawn in this way *planar graphs*. On the other hand, graphs in practice are often not planar, but—as stated above by Diestel—we can limit the mess in the drawing by reducing the number of line intersections. Hence, it is not surprising that one of the most important aesthetic criteria in graph drawing is to minimize the number of edge crossings in a drawing. Further important aesthetics include to avoid that vertices are drawn too close to each other and angles between edges are too small. The latter is automatically fulfilled by *orthogonal drawings*, that is, drawings in which edges are represented by lines consisting of only vertical and horizontal line segments. In such drawings, it is preferred to have a small number of bends. Moreover, edges should be short, that is the sum of the edge lengths or the maximal length of an edge should be minimized. Similarly, a small drawing area is desirable, thus leading to compact drawings.

Figure 1.1 shows two examples. On the left hand side, we see a straight-line drawing of the graph `4elt` drawn with a multilevel force-directed algorithm [Gronemann, 2009]. This graph has 15,606 vertices and 45,878 edges and is part of a benchmark set for graph partitioning¹, which is also frequently used for evaluating drawing algorithms for large graphs. It is a little bit surprising that this

¹See: <http://staffweb.cms.gre.ac.uk/~wc06/partition/>

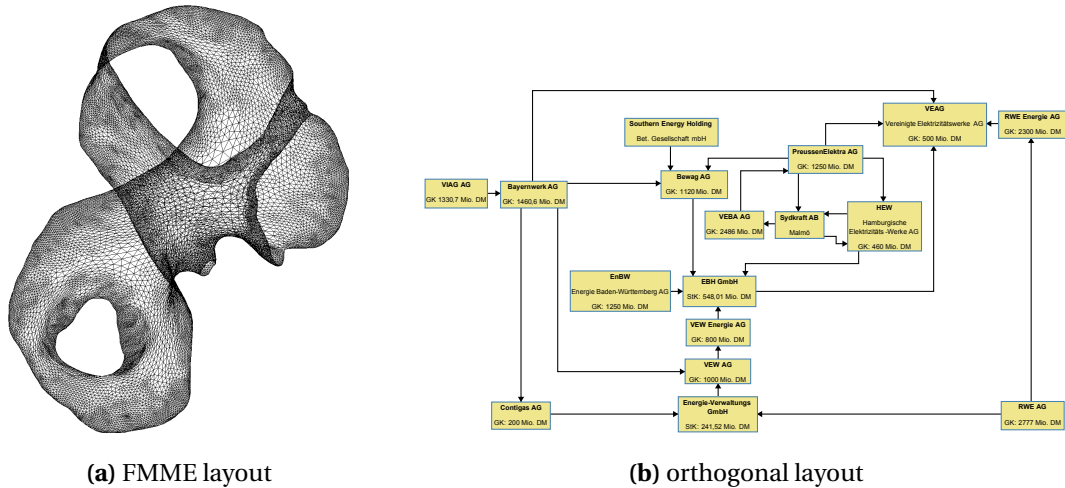


Figure 1.1: (a) shows the graph `4elt` drawn with the multilevel force-directed algorithm FMME; (b) shows interconnections in the electricity industry.

graph is actually planar. In Figure 1.1(b), we see a real-world graph displaying the interconnections between companies in the electricity industry. It originally appeared in a German newspaper and was drawn in such a confusing way that it was included in a book on the most confusing diagrams in the world [Henschel, 2003]. However, applying sophisticated layout algorithms reveals that the interconnections can be presented in a clear and concise way. On the other hand, we are not sure, whether the author of the newspaper article intentionally used a confusing diagram to support his point of view. A similar example is a diagram showing the financial integration of banks, funds, and companies; see Figure 1.2. It appeared in an online article on `wordpress.com`². The original version was drawn in an orthogonal layout style with a huge amount of edge crossings. The drawing shown here is the output of an orthogonal layout algorithm (we omitted isolated vertices), presenting the information in a much clearer way that allows to follow edges quite easily.

The drawings in Figure 1.1(b) and 1.2 are obtained by applying the *topology-shape-metrics approach*. This general approach for orthogonal graph layout was proposed by and Tamassia; see [Tamassia, 1987, Tamassia et al., 1988]. It consists of three steps:

- *Topology*: This step determines a planar embedding of the graph. If the input graph is not planar, it is planarized, that is, additional vertices with degree four are inserted that represent edge crossings in the final drawing. Hence, this step fixes the number of crossings in the drawing.
- *Shape*: This step—also called *orthogonalization phase*—determines the angles and the bends in the drawing, in particular, it fixes the number of bends in the drawing.

²<http://blogpoliteia.wordpress.com/2009/07/>; a PDF of this diagram can be found at: <http://blogpoliteia.files.wordpress.com/2009/03/world-gov3.pdf>

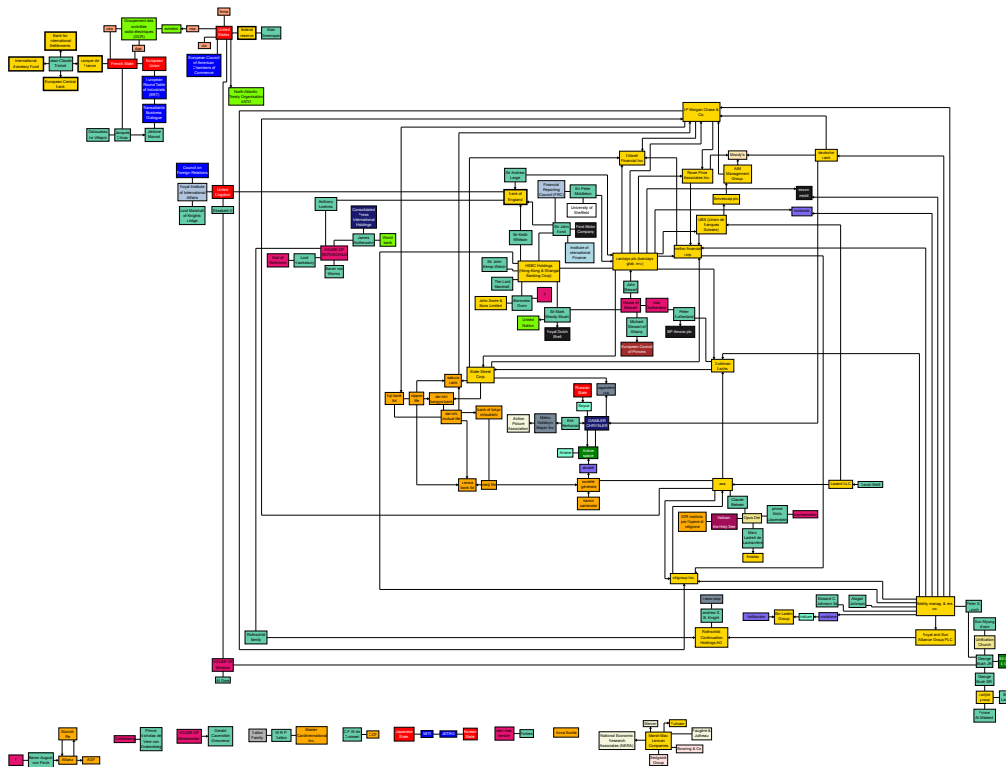


Figure 1.2: A graph showing the financial integration of banks, funds, and companies; source: <http://blogpoliteia.wordpress.com/2009/07/>

- *Metrics:* This step computes the final coordinates of the vertices and bend points. Since its main objective is to yield short edges and a small drawing area, it is also called *compaction phase*.

The distinct steps of this modular framework are each devoted to specific aesthetic criteria: The first step minimizes the number of edge crossings, the second step minimizes the number of bends, and finally the last step minimizes the edge lengths. The criteria are considered in decreasing level of importance, since an optimized criterion is never worsened by subsequent steps.

In this thesis, we focus on the first step of the topology-shape-metrics approach, realized by the *planarization method*. This method was introduced by Batini et al. [1984] and is the most successful method for minimizing the number of crossings in practice. We give a comprehensive introduction to this method in Section 2.4. We apply BC- and SPQR-trees, the decomposition of a graph into its bi- and triconnected components, for improving the planarization method in various respects: significantly improve the quality of the crossing reduction step, find planar embeddings that lead to better solutions in the topology-shape-metrics approach, and extend the applicability of the method in practice by supporting embedding constraints which impose certain restrictions on the incident edges around a vertex.

1.1 Organization of this Thesis

This thesis is organized as follows. After giving some basic definitions and results from graph theory and graph drawing in Chapter 2, Chapter 3 deals with decomposing a graph into its bi- and triconnected components. The focus lies on the triconnected components and their representation in form of SPQR-trees, which are essential for all new algorithms introduced in this thesis. At the end of this chapter, we give a linear time algorithm to compute the triconnected components of a graph and to construct the corresponding SPQR-tree. This algorithm builds upon Hopcroft and Tarjan's algorithm [Hopcroft and Tarjan, 1973a] and corrects some serious errors in this algorithm.

The next chapter presents state-of-the-art algorithms for heuristic crossing minimization. We solve the long-standing open problem of optimally inserting one edge into a planar graph by giving a linear time algorithm in Section 4.1 and describe several variations of the planarization approach for crossing minimization in Section 4.2. The chapter closes with an extensive experimental analysis of these crossing minimization heuristics, considering both practical benchmark instances as well as a new benchmark set of special graph classes with known crossing number.

Chapter 5 revisits the planar embedding step required in all graph drawing algorithms based on planarity. After minimizing the number of crossings, we obtain a planar graph representing the actual graph we want to draw in such a way that edge crossings are represented as vertices with degree four. Traditionally, just any graph embedding algorithm is applied to compute a planar embedding of this graph, which is then used to generate, for example, an orthogonal drawing. However, a planar graph might have an exponential number of possible embeddings, but the choice of the embedding can have a big influence on the quality of the resulting drawing. We propose a different approach of graph embedding by presenting efficient, linear time algorithms that optimize important properties of the embedding like the degree of the external face (Section 5.1) and the nesting depth of blocks (Section 5.2).

Chapter 6 extends the planarization approach by including embedding constraints, which impose additional restrictions on the order of edges around a vertex. We present a linear-time algorithm that tests if a given graph with a set of embedding constraints C has a planar embedding observing the constraints C ; in the positive case, this algorithm also computes such a planar embedding. Moreover, we generalize the edge insertion problem accordingly and give an optimal linear-time algorithm for inserting one edge, thus yielding a planarization method for graphs with embedding constraints.

Finally, Chapter 7 concludes by recapitulating the achievements presented in this thesis and their impact on current research, as well as with an outlook on future work.

1.2 Corresponding Publications

The results presented in this thesis have partially been published in conference proceedings and journals. This section lists these publications with references to the corresponding chapters or sections in this thesis.

- The algorithm for constructing SPQR-trees in linear time (see Section 3.4) appeared in the conference proceedings of *Graph Drawing 2000*; see [Gutwenger and Mutzel, 2001].
- The optimal edge insertion algorithm (see Section 4.1) was first published in the conference proceedings of *SODA 2001* [Gutwenger, Mutzel, and Weiskircher, 2001] and appeared then in the *Algorithmica* journal; see [Gutwenger, Mutzel, and Weiskircher, 2005].
- A first version of the experimental study of crossing minimization heuristics appeared in the conference proceedings of *Graph Drawing 2003*; see [Gutwenger and Mutzel, 2003a]. However, all the experiments of the study as presented in Section 4.3 have been redone: The set of crossing minimization heuristics has been extended, an additional benchmark set of graphs with known crossing numbers is introduced, and new findings have been obtained.
- The algorithms for computing planar embeddings with maximum external face and minimum block-nesting depth (see Chapter 5) appeared also in the conference proceedings of *Graph Drawing 2003*; see [Gutwenger and Mutzel, 2003b].
- The preprocessing method for crossing minimization, the *non-planar core reduction* (see Section 4.2.1), was first published in the conference proceedings of *Graph Drawing 2005* [Chimani and Gutwenger, 2005] and appeared later in the Special Issue on Graph Drawing 2005 in the journal *Discrete Mathematics* [Chimani and Gutwenger, 2009]. These articles also apply the preprocessing method to the skewness, thickness, and coarseness of a graph; in this thesis, only the essential results concerning crossing minimization are presented, since this preprocessing strategy is used in the experimental study on crossing minimization heuristics.
- The results on graph embedding and edge insertion with embedding constraints (see Chapter 6) was first published in the conference proceedings of *Graph Drawing 2006* [Gutwenger, Klein, and Mutzel, 2006] and an extended version appeared then in the *Journal of Graph Algorithms and Applications* (Special Issue on Selected Papers from GD 2006); see [Gutwenger, Klein, and Mutzel, 2008].

1.3 Related Work

Two further results, which are related to crossing minimization, are not described in detail here, but are used in the sections on crossing minimization heuristics (see Section 4.2 and 4.3).

- In [Buchheim, Chimani, Ebner, Gutwenger, Jünger, Klau, Mutzel, and Weiskircher, 2008] we present a branch-and-cut based approach for exact crossing minimization, which was published in the journal *Discrete Optimization*. An experimental study examining the performance and usefulness of the column generation scheme applied in this algorithm was presented at *WEA 2006* [Chimani, Gutwenger, and Mutzel, 2006] and was published in the *ACM Journal of Experimental Algorithmics* [Chimani, Gutwenger, and Mutzel, 2009a]. We will use the exact crossing numbers of some benchmark graphs to evaluate the quality of heuristic solutions.
- The second result relates to the minimum cut of planarizations and is used in the application of the non-planar core reduction to the crossing number (Section 4.2.1). It was presented at the 6th Czech-Slovak International Symposium on Combinatorics, Graph Theory, Algorithms and Applications in 2006 and appeared in *Electronic Notes in Discrete Mathematics* [Chimani, Gutwenger, and Mutzel, 2007].

Chapter 2

Preliminaries

You have to learn the rules of the game. And then you have to play better than anyone else.

ALBERT EINSTEIN (1879 – 1955)

This chapter introduces the required mathematical and graph theoretic background and provides notational conventions used throughout this thesis. The first two sections deal with basic definitions concerning graphs and trees and the third section discusses fundamental terms and results from graph drawing focusing on graph planarity and embeddings. Finally, the last section is devoted to special classes of graphs which are applied in the study of graphs with known crossing numbers. We use such graphs in computational studies on crossing minimization (see Section 4.3). The mathematical notation is mainly derived from the text books by Diestel [2005] and Jünger and Mutzel [2004]. We denote with \mathbb{R} the set of real numbers and with $\mathbb{N} = \{0, 1, 2, \dots\}$ the set of natural numbers including 0.

2.1 Undirected Graphs

A graph $G = (V, E)$ consists of a finite set V of *vertices* and a finite multi-set E of *edges*. Each edge $e \in E$ is an unordered pair (u, v) with $u, v \in V$. The vertices u and v are the *endvertices* of e , and e is *incident* to u and v . Two vertices $u, v \in V$ are *adjacent*, or *neighbors*, if $(u, v) \in E$. The *degree* of a vertex $v \in V$ is the number of edges incident to v and is denoted with $\deg(v)$. If all the vertices of G are pairwise adjacent, then G is *complete*. An edge (v, v) is called a *self-loop*. If an edge $(u, v) \in E$ occurs more than once in E , it is called a *parallel* or *multiple edge*. G is called *simple*, if it contains neither self-loops nor multiple edges.

We refer with $V(G)$ to the set of vertices of a graph G and with $E(G)$ to the set of its edges. For ease of notation, we shall not always distinguish strictly between a graph and its set of vertices and edges, that is, we shall also write $v \in G$ or $e \in G$, rather than $v \in V(G)$ or $e \in E(G)$, respectively.

The complete graph with n vertices is denoted as K_n and the graph (\emptyset, \emptyset) is called the *empty graph*. A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ (and G a

supergraph of G') if $V' \subseteq V$ and $E' \subseteq E$. In this case, we write $G' \subseteq G$ and say that G *contains* G' . We call G' a *spanning* subgraph of G if G' contains all vertices of G , that is, $V' = V$. For any set $F \subseteq E$ of edges, $V(F)$ denotes the set of all vertices incident to at least one edge in F .

If U is a set of vertices of $G = (V, E)$, we write $G - U$ for the graph $G' = (V \setminus U, E')$, where E' contains exactly the edges in E with both endvertices in $V \setminus U$. In other words, $G - U$ results from G by deleting all the vertices in U and their incident edges. If $U = \{v\}$ is a singleton, we simply write $G - v$ rather than $G - \{v\}$. For a subset $F \subseteq V \times V$, we write $G - F$ for the graph $(V, E \setminus F)$ and $G + F$ for the graph $(V, E \cup F)$; as for vertex sets, we use the abbreviations $G - e$ and $G + e$ if F is a singleton $\{e\}$. For two graphs $G = (V, E)$ and $G' = (V', E')$, we use the notations $G \cup G' := (V \cup V', E \cup E')$ and $G \cap G' := (V \cap V', E \cap E')$.

2.1.1 Paths and Cycles

A *path* $p : v \overset{*}{\Rightarrow} w$ in G is an alternating sequence of vertices and edges $v = v_0, e_1, v_1, \dots, e_k, v_k = w$ with $k \geq 0$ such that v_{i-1} and v_i are incident to e_i for $i = 1, \dots, k$. The vertices v_1, \dots, v_{k-1} are called *inner vertices* of p . A path is *simple* if all its vertices are distinct. The *length* of a path is the number of edges on the path. Two or more paths are *independent* if none of them contains an inner vertex of another. A graph H is a *subdivision* of G if H is obtained from G by replacing some of the edges of G with independent paths between their endvertices so that none of these paths has an inner vertex in G .

If $p : v \overset{*}{\Rightarrow} w$ is a simple path, then p plus the edge (w, v) is called a *cycle* and its *length* is the number of edges (equivalent: vertices) on the cycle. In particular, a self-loop forms a path of length one and two multiple edges a path of length two. For ease of notation, we omit the edges in the description of a path if they are evident by context. A graph that does not contain any cycle is called a *forest*.

2.1.2 Connectivity

A non-empty graph $G = (V, E)$ is *connected* if every pair of vertices in G is connected by a path, otherwise it is called *disconnected*. A maximal connected subgraph of G is called a *component* of G . If there exist three distinct vertices v, w, a of G such that a lies on every path $v \overset{*}{\Rightarrow} w$, then a is called a *cut vertex* or *articulation point*. Similarly, an edge $e = (v, w)$ is called a *bridge* if e lies on every path $v \overset{*}{\Rightarrow} w$. For an integer $k \geq 0$, $G = (V, E)$ is called *k-connected* if $|V| > k$ and $G - X$ is connected for every $X \subseteq V$ with $|X| < k$. Every non-empty graph is 0-connected, and the 1-connected graphs are exactly the connected graphs with at least two vertices. It is also common to write *biconnected* and *triconnected* instead of 2-connected and 3-connected, respectively. A connected forest is called a *tree*.

The following theorem is one of the cornerstones of graph theory.

Theorem 2.1 (Menger's theorem [Menger, 1927, Whitney, 1932b, Diestel, 2005]). *A graph is k -connected if and only if it contains k independent paths between any two vertices.*

2.1.3 Minimum Cuts

A *cut* in G is a partition (S, \bar{S}) of the vertices of G . The *capacity* $c(S, \bar{S})$ of the cut is the cardinality of the set $E(S, \bar{S})$ of all the edges connecting vertices in S with vertices in \bar{S} . For two vertices $s, t \in V$, we call (S, \bar{S}) an st -cut if s and t are in different sets of the cut. A *minimum st -cut* is an st -cut of minimum capacity. We denote the capacity of a minimum st -cut in G with $\text{mincut}_{s,t}(G)$.

2.2 Directed Graphs

Directed graphs are defined similarly to graphs with the exception that the edges are ordered pairs of vertices. To avoid confusion, we refer to the vertices of a directed graph as nodes and to its edges as arcs. More formally, a *directed graph* (*digraph*) $G = (V, A)$ consists of a finite set V of *nodes* and a finite multi-set A of *arcs* with $A \subseteq V \times V^1$. If $a = (v, w)$ is an arc, v is called the *source node* and w the *target node* of a . Furthermore, a is an *incoming arc* of w and an *outgoing arc* of v . The number of incoming arcs of a node v is called the *in-degree* of v (denoted with $\text{indeg}(v)$) and the number of outgoing arcs the *out-degree* of v (denoted with $\text{outdeg}(v)$).

The *underlying undirected graph* of a digraph G is simply the graph that results from G by considering each arc in G as an unordered pair of vertices. Thus, concepts like path, cycle, subgraph, or connected component naturally carry over to digraphs. In addition, for a *directed path* or *cycle* we demand that the nodes in the path or cycle correspond to the direction of the arcs, for example, if we have a directed path $v_0, e_1, v_1, \dots, e_k, v_k$, then $e_i = (v_{i-1}, v_i)$ for $i = 1, \dots, k$. An *acyclic* digraph is a digraph with no directed cycle. A *source* is a node with no incoming arcs and a *sink* is a node with no outgoing arcs.

2.2.1 Rooted Trees

A *rooted tree* T is a digraph with a single source whose underlying undirected graph is a tree. The unique source r of T is called its *root* and all sinks are called *leaves*. An arc in T from v to w is denoted with $v \rightarrow w$. If there is a directed path from v to w , we write $v \xrightarrow{*} w$. If $v \rightarrow w$, v is the *parent* of w and w a *child* of v . If $v \xrightarrow{*} w$, then v is an *ancestor* of w and w a *descendant* of v . Every node is an ancestor and a descendant of itself. The *depth* of a node v is the length of the (unique) path $r \xrightarrow{*} v$ and the *depth* of the tree is the length of the longest path from r to a leaf.

¹Here, we consider $V \times V$ as a multi-set of ordered pairs (v, w) with $v, w \in V$.

```

Input: undirected graph  $G = (V, E)$ , start vertex  $s$ 
Output: edges of  $G$  are marked as T- or B-arc

1:  $nextNum := 0$ 
2: for each  $v \in V$  do
3:    $number[v] := 0$ 
4: end for
5: DFS-VISIT( $s, nil$ )

6: function DFS-VISIT(vertex  $v$ , edge  $e_p$ )
7:    $nextNum := nextNum + 1$ 
8:    $number[v] := nextNum$ 
9:   for all edges  $e = (v, w)$  incident to  $v$  do
10:    if  $number[w] = 0$  then
11:      mark  $e$  as T-arc  $v \rightarrow w$ 
12:      DFS-VISIT( $w, e$ )
13:    else if  $e \neq e_p$  and  $number[w] < number[v]$  then
14:      mark  $e$  as B-arc  $v \hookrightarrow w$ 
15:    end if
16:  end for
17: end function

```

Listing 2.1: Depth-first search (DFS).

2.2.2 Depth-First Search and DFS-Trees

Let G be a connected undirected graph without self-loops. A *depth-first search* (DFS) traversal of G assigns each edge a direction and partitions the edges of G into two classes. We obtain *T-arcs* (tree-arcs, denoted with $v \rightarrow w$) and *B-arcs* (backward-arcs, denoted with $v \hookrightarrow w$). The T-arcs form a rooted tree (*DFS-tree*) that spans G , and for each B-arc $v \hookrightarrow w$, there exists a (directed) path $w \xrightarrow{*} v$ of T-arcs leading from w back to v . Listing 2.1 shows an $\mathcal{O}(|V| + |E|)$ time algorithm for a DFS traversal of a graph $G = (V, E)$ starting at vertex s . The recursive procedure DFS-VISIT also assigns a number to each vertex indicating the order in which vertices are visited by DFS. Figure 2.1 gives an example for a graph processed by DFS. The following table summarizes the variables used in Listing 2.1.

variable	purpose
s	Root vertex of DFS-tree.
$nextNum$	The next DFS number.
$number[v]$	DFS number of vertex v .

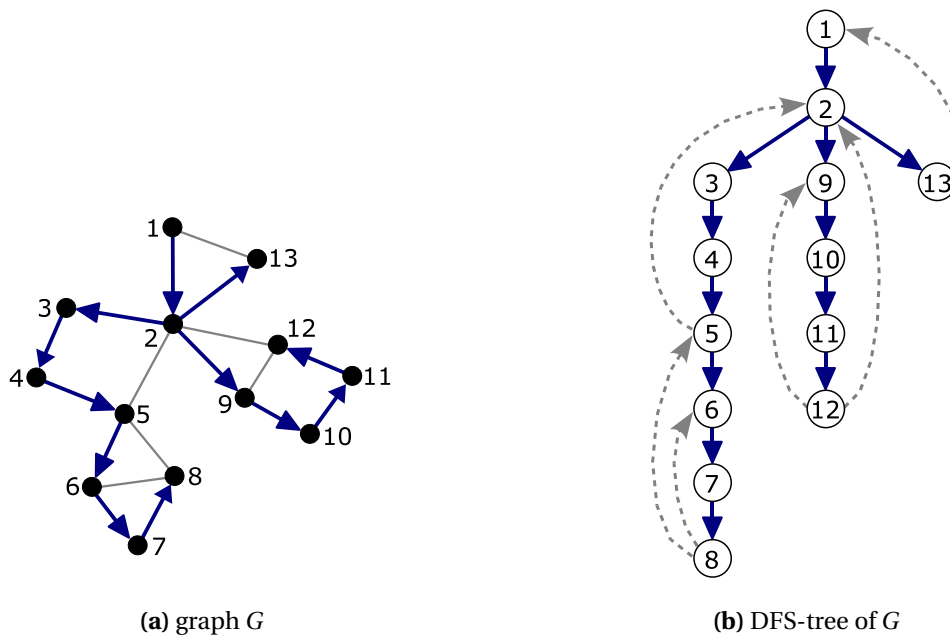


Figure 2.1: A graph processed by DFS: (a) shows the graph with vertices labeled by their DFS number, (b) shows the resulting DFS-tree (B-arcs are drawn as dashed lines).

2.3 Planar Graphs and Drawings

In a *drawing* of a graph $G = (V, E)$, each vertex $v \in V$ is mapped to a distinct point p_v in the plane and each edge $(u, v) \in E$ is mapped to a closed simple curve that connects the points p_u and p_v and does not pass through the image of any other vertex. If two curves share an interior point p , we say that they *cross* at p . We call a drawing of G without any edge crossing a *planar drawing* and a graph that can be drawn without edge crossings a *planar graph*. In the following, we assume that G is connected.

2.3.1 Graph Embeddings

A planar drawing of a graph divides the plane into topologically connected regions called *faces* that are bounded by the curves corresponding to the edges. Exactly one of the faces is unbounded and is called the *external face*. A face is uniquely described by the sequence of its boundary edges. The *degree* $\deg(f)$ of a face f is defined as the number of its boundary edges, where each edge with both sides on the boundary of f is counted twice. The set of vertices on f is denoted with $V(f)$. Two faces are *adjacent* if their boundaries share a common edge.

We say that planar drawings with the same set of faces realize the same combinatorial embedding. A *combinatorial embedding* Γ essentially fixes the topol-

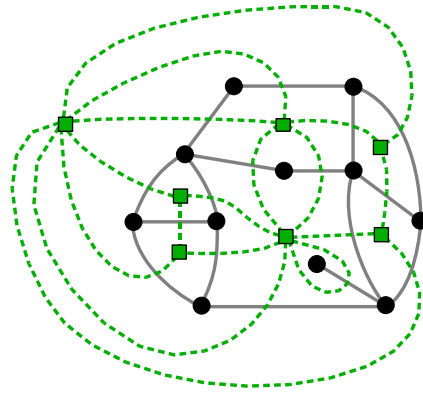


Figure 2.2: The *dual graph* is constructed by placing a dual vertex within every face and connecting the dual vertices of adjacent faces. Dual vertices are drawn as squares and dual edges as dashed lines.

ogy of the graph and is defined as a cyclic, clockwise ordered list for each vertex $v \in V$ that contains the edges incident to v . This cyclic ordering uniquely defines the set of faces in any drawing that realizes Γ . When, in addition to the combinatorial embedding Γ , the external face f_0 is fixed, then (Γ, f_0) is called a *planar embedding* of G .

A famous result by Euler for polytopes relates the number of vertices, edges, and faces in any combinatorial embedding of a connected planar graph:

Theorem 2.2 (Euler's formula 1758). *Let Γ be a combinatorial embedding of a connected planar graph $G = (V, E)$ and let F be the set of faces in Γ . Then $|V| - |E| + |F| = 2$.*

This equation implies a well-known upper bound for the number of edges in a simple planar graph with at least 3 vertices. A nice proof of the following corollary can be found in [Diestel, 2005, Corollary 4.2.10].

Corollary 2.1. *Let $G = (V, E)$ be a simple planar graph with at least 3 vertices. Then $|E| \leq 3|V| - 6$.*

Given a planar graph, a combinatorial embedding can be computed in linear time [Chiba et al., 1985, Mehlhorn and Mutzel, 1996, Boyer and Myrvold, 2004] and any face can be chosen to be the external face. However, a planar graph can have an exponential number of combinatorial embeddings in general. Throughout this thesis, we will frequently use the name *embedding* for planar or combinatorial embedding.

2.3.2 The Dual Graph

The dual graph expresses the adjacency relationships between the faces in an embedding. Given an embedding Γ of a planar graph $G = (V, E)$ with face set F , the *dual graph* $\Gamma^* = (V^*, E^*)$ is constructed as follows: $V^* = F$ and E^* contains an

edge (f_1, f_2) for each $e \in E$ such that e is on the boundary of both f_1 and f_2 . If e is only on the boundary of a single face f (which means that e is a bridge), the dual graph contains a self-loop (f, f) . G and Γ^* are dual in the following sense: The order of the edges on the face boundaries of Γ implies a natural embedding Π of Γ^* . With respect to Π , G is in turn the dual graph of Γ^* . An example for the construction of the dual graph is shown in Figure 2.2. It illustrates that the dual graph may contain multiple edges as well as self-loops.

2.3.3 Non-planarity Measures

If a graph G is not planar, the following question arises naturally: How far away is G from planarity? For that reason, various measures for non-planarity have been proposed; see also Liebers [2001] for a survey. The most prominent measure is the *crossing number* $cr(G)$ which is the minimal number of crossings in any drawing of G . The *crossing number problem* is the problem of finding the crossing number for a given graph G .

The *skewness* of G is the minimum number of edges that have to be removed from G for obtaining a planar graph. This is computationally equivalent to the *maximum planar subgraph* problem that asks for a planar subgraph of G with the maximum number of edges. The *thickness* of G is the minimum number of planar subgraphs of G whose union is G . On the other hand, the *coarseness* is the maximum number of edge-disjoint non-planar subgraphs of G .

2.4 The Planarization Method

The most prominent and practically successful method used for minimizing the number of crossings in a drawing is the *planarization method*. This approach was introduced by Batini et al. [1984] and can be viewed as a general framework which addresses the crossing minimization problem with a two step strategy. Each step aims at solving a particular optimization problem for which various solution methods are possible. Let $G = (V, E)$ be the graph for which we want to find a crossing minimal drawing. Then, the two steps to be executed are:

1. Compute a planar subgraph of G that contains as many edges as possible.
2. Reinsert the edges not contained in the planar subgraph with as few crossings as possible. Each crossing that occurs when inserting an edge is replaced by a new vertex of degree four called a *crossing vertex*.

The insertion of crossing vertices during the edge reinsertion step ensures that the (modified) graph remains planar. Finally, we end up with a planar graph G_p consisting of representatives of the vertices of G plus some crossing vertices. Each edge e of G is mapped to a path in G_p that connects the two representatives of the end vertices of e via zero or more crossing vertices; vice versa, each edge

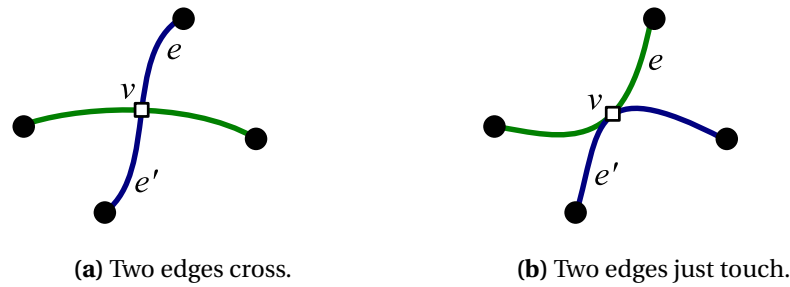


Figure 2.3: The two possible configurations for a crossing vertex v in an embedding of the planarized representation.

of G_p corresponds to a unique edge of G . Hence, we call G_p a *planarized representation* of G and the number of crossing vertices in G_p the *number of crossings* in G_p .

Since G_p is a planar graph, we can create a planar drawing \mathcal{D}_p of G_p and interpret it as a drawing \mathcal{D} of G . For each crossing vertex v of G_p , two configurations are possible; see Figure 2.3. Either the two corresponding edges in G , say e and e' , cross each other in \mathcal{D}_p , then v in fact represents a crossing between e and e' in \mathcal{D} ; or e and e' just touch and there is no crossing required in \mathcal{D} .

Figure 2.4 illustrates the different stages of the planarization approach with an example. In this case, the planar subgraph contains all but one edge (edge (2,5) is missing) and the final drawing of G has only a single crossing.

The two optimization problems we have to solve in the planarization approach are the *maximum planar subgraph* (MPSP) and the *edge insertion problem* (EIP). Both problems are NP-hard and are usually solved with heuristic approaches. We will see later that even an optimal solution of MPSP in the first step and of EIP in the second step does not yield a crossing minimal solution in general; in fact the solution may be arbitrarily bad. In Section 4.1.4, we give an example of a family of graphs G'_m , for which a maximum planar subgraph contains all but one edge and an optimal solution of EIP results in m crossings. On the other hand, a crossing minimal drawing of G'_m has only two crossings. However, this is merely a pathological example and experimental analysis shows that the planarization approaches typically performs excellent; see Section 4.3.

2.4.1 Planar Subgraphs

In many practical applications, we expect that a graph can be made planar by removing only a few edges. Therefore, it is reasonable to use a planar subgraph with as many edges as possible as a starting point for crossing minimization. The corresponding optimization problem is stated as follows:

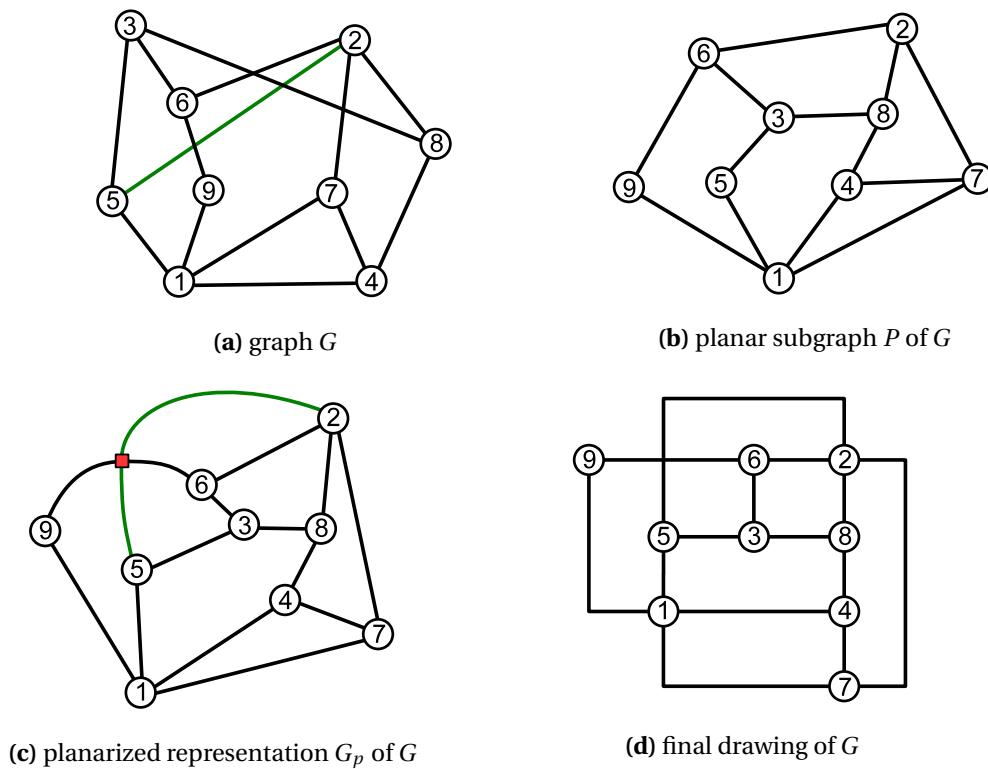


Figure 2.4: A sample application of the planarization method.

MAXIMUM PLANAR SUBGRAPH PROBLEM (MPSP)	
Instance:	a graph $G = (V, E)$
Solution:	a subset $E' \subseteq E$ such that $G' = (V, E')$ is planar
Maximize:	the size of the E'

Garey and Johnson [1979] have shown that MPSP is NP-hard. In [Jünger and Mutzel, 1996], Jünger and Mutzel presented a branch-and-cut algorithm for solving MPSP to optimality. Computational results show that the algorithm is able to provide a provably optimal solution pretty fast if the number of edges to be deleted is small. However, the method is quite complicated to understand and to implement. Moreover, if the number of deleted edges exceeds 10, the algorithm usually needs far too long to be acceptable for practical computation.

Since finding a maximum planar subgraph is hard, the problem of finding just a maximal planar subgraph has received much attention. A *maximal planar subgraph* of $G = (V, E)$ is a planar subgraph $P = (V, E \setminus F)$ of G such that adding any edge of F to P destroys the planarity, i.e., $P \cup e$ is not planar for every $e \in F$. A widely used standard heuristic for finding a maximal planar subgraph is to start with a spanning tree of G , and to iteratively try to add the remaining edges one by one; see Listing 2.2. In every step, a planarity testing algorithm is called for the obtained graph. If the addition of an edge would lead to a non-planar graph, then the edge is disregarded; otherwise, the edge is added permanently to the

```

1: procedure MAXIMALPLANARSUBGRAPH(Graph  $G = (V, E)$ )
2:   let  $P = (V, E_P)$  be a spanning tree of  $G$ 
3:    $F := E \setminus E_P$ 
4:   for all  $e \in F$  do
5:     if  $P \cup e$  is planar then
6:        $P := P \cup e$ 
7:     end if
8:   end for
9: end procedure

```

Listing 2.2: A simple algorithm for computing a maximal planar subgraph P .

planar graph obtained so far. After $|F|$ planarity tests, we obtain a maximal planar subgraph P of G . Since planarity can be tested in linear time [Hopcroft and Tarjan, 1974, Booth and Lueker, 1976, Boyer and Myrvold, 2004], the running time of the procedure is $\mathcal{O}((1 + |F|)(|V| + |E|))$.

This incremental approach can be made more efficient by using incremental planarity testing algorithms. Di Battista and Tamassia [1996a] presented an algorithm that tests in $\mathcal{O}(\log|V|)$ time if an edge can be added while preserving planarity, and that performs the required updates of the data structure when adding an edge in $\mathcal{O}(\log|V|)$ amortized time. The running time for incremental planarity testing has been improved by La Poutré [1994] to $\mathcal{O}(\alpha(|E|, |V|))$ amortized time per query and update operation. This yields an almost linear time algorithm for the maximal planar subgraph problem that runs in $\mathcal{O}(|V| + |E| \cdot \alpha(|E|, |V|))$ time. Here, $\alpha(x, y)$ denotes the inverse Ackermann function, which means that $\alpha(x, y)$ is a function that grows extremely slowly. A linear time algorithm for finding a maximal planar subgraph is given by Djidjev [1995]. This algorithm uses the decomposition of the graph into BC- and SPQR-trees and applies a fast data structure for on-line planarity testing in triconnected graphs. BC- and SPQR-trees are discussed in detail in Chapter 3.

Jayakumar et al. [1989] [see also Jünger et al., 1998] proposed a method for computing a planar subgraph that is based on PQ-trees. The PQ-tree data structure has been developed by Booth and Lueker [1976] for solving the problem of finding permissible permutations of a set U . The permissible permutations are those in which certain subsets $S \subseteq U$ occur as consecutive subsequences. Drawbacks of this planar subgraph algorithm are that it cannot guarantee to find a maximal planar subgraph, and that the theoretical worst case running time is $\mathcal{O}(|V|^2)$. However, in practice it is usually very fast and the quality of the results can be improved by introducing random events and calling the algorithm several times. The algorithm starts by computing an st -numbering of G which determines the order in which the vertices are processed. A simple but useful randomization is to choose a random edge (s, t) for each run. We will use this algorithm in the experimental study presented in Section 4.3.

The trivial approach for finding a planar subgraph consists of computing a spanning tree. If G is a graph with n vertices and c components, then this ap-

proach has an approximation factor of $\frac{n-c}{3|V|-6c} > \frac{1}{3}$ for MPSP, since a spanning tree of G contains $|V| - c$ edges and a planar graph with c components has at most $3|V| - 6c$ edges by Euler's formula. Surprisingly, we cannot guarantee a better approximation factor than that of the spanning tree approach if we also demand that the computed subgraph must be maximal planar; see [Dyer et al., 1985]. The currently best approximation factor of $4/9$ is achieved by the algorithm by Călinescu et al. [1998] which runs in $\mathcal{O}(|E|^{3/2}|V|\log^6|V|)$ time.

2.4.2 Edge Insertion

The planar subgraph P computed in the first step of the planarization approach is a good starting point for finding a planarized representation G_p of G with few crossings. In practice, we expect that only a small number of edges has to be inserted into P in order to obtain G_p . However, the edge insertion step fixes the crossings in the final drawing, and the choice of the edge insertion technique may have a significant impact on the quality of the final solution. Formally, the edge insertion problem is defined as follows:

EDGE INSERTION PROBLEM (EIP)	
Instance:	a graph $G = (V, E)$ and a set $E' \subseteq V \times V$
Solution:	a planarized representation G_p of $G' = (V, E \cup E')$ in which no two edges of E cross
Minimize:	the number of crossings in G_p

Mutzel and Ziegler [1999] [see also Ziegler, 2000] have shown that even a restricted variant of the edge insertion problem is NP-hard: The *constrained crossing minimization problem (CCMP)* asks for the minimum number of crossings required for inserting a set of edges into a fixed embedding of a planar graph. In contrast to CCMP, the general edge insertion problem leaves the freedom to choose a suitable embedding of the planar subgraph. Ziegler and Mutzel also gave a branch-and-cut algorithm to solve CCMP. However, experiments showed that it can only solve instances to provable optimality if there are less than 10 edges to be inserted.

The standard method for heuristically solving EIP fixes an embedding of the planar subgraph and processes the edges to be inserted one by one; compare Listing 2.3.

Fixed Embedding. Suppose, we want to insert edge $e = (v, w)$ into the planar graph G_p . Let Π be a fixed embedding of G_p . We construct the *extended dual graph* G^* of Π with respect to e as follows. The vertices of G^* are the faces of Π plus two new vertices v^* and w^* representing v and w . For each edge e' in G_p , we have an edge in G^* connecting the two faces separated by e' (if e' is a bridge, we have a self-loop in G_p). Additionally, we have an edge (v^*, f_v) for each face f_v adjacent to v , and (w^*, f_w) for each face f_w adjacent to w .

```

1: procedure FIXEDEMBEDDINGINSERTER(Graph  $P = (V, E)$ ,  $E' \subseteq V \times V$ )
2:    $G_p := P$ 
3:   let  $\Pi$  be an arbitrary embedding of  $G_p$ 
4:   for each  $(u, v) \in E'$  do
5:     Compute extended dual  $G^*$  of  $\Pi$  with respect to  $(u, v)$ 
6:     let  $e_u, e_1^*, \dots, e_\ell^*, e_v$  be a shortest path in  $G^*$  from  $u^*$  to  $v^*$ 
7:     Insert edge  $(u, v)$  into  $G_p$  and  $\Pi$  such that it crosses  $e_1, \dots, e_\ell$ 
8:   end for
9: end procedure

```

Listing 2.3: Standard procedure for inserting a set of edges E' into a fixed embedding of a planar graph P ; the result is a planarized representation G_p .

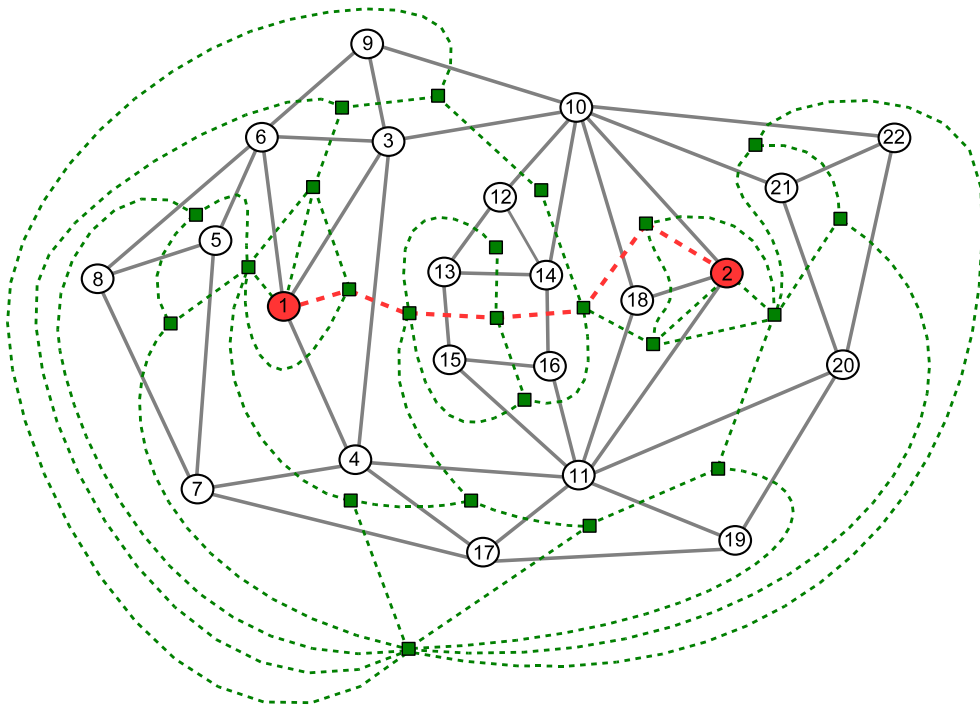


Figure 2.5: Edge insertion with fixed embedding by finding a shortest path in the extended dual graph.

We observe that inserting e into Π corresponds to finding an (undirected) path from v^* to w^* in G^* . If such a path has length ℓ , then we can insert e with $\ell - 2$ crossings, since the first and the last edge on this path do not produce a crossing. Therefore, in order to insert e into Π with the minimum number of crossings, we have to find a shortest path from v^* to w^* in G^* . This is possible in linear time using a simple breadth-first search traversal starting at v^* . Figure 2.5 shows a non-trivial example. Here, we want to connect the vertices 1 and 2. The dashed vertices and edges belong to the extended dual graph. The optimal solution highlighted in bold crosses four edges.

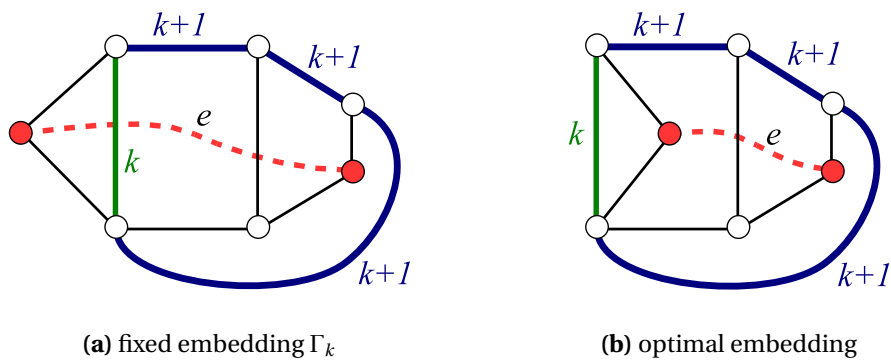


Figure 2.6: A family of graphs G_k and embeddings Γ_k for which the insertion of an edge e requires k crossings more than the optimal solution.

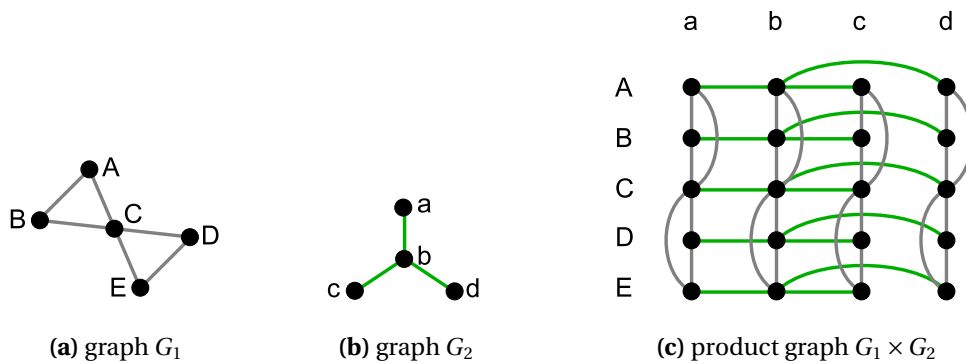


Figure 2.7: The construction of the Cartesian product of two graphs: (c) shows the Cartesian product $G_1 \times G_2$ of the graphs G_1 and G_2 shown in (a) and (b).

Though we can easily find a crossing minimal solution if the embedding of G_p is fixed, the drawback of this method is that fixing an unfavorable embedding may result in an arbitrarily bad solution. Figure 2.6(a) gives an example of such a family of graphs G_k with embeddings Γ_k . The blue fat lines in this figure denote bundles of $k + 1$ parallel edges, and the green fat line a bundle of k parallel edges. Hence, inserting edge e into the given embedding requires at least $k + 1$ crossings. On the other hand, it is possible to insert e with only one crossing by changing the embedding; see Figure 2.6(b). It is easy to see that this example can also be adapted to the case of simple graphs by splitting all the edges in each bundle.

2.5 Artificial Graphs

The product of graphs is a useful tool for constructing artificial graphs and it is intensively used in the field of known crossing numbers for special graph families. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. The (*Cartesian*) *product* of

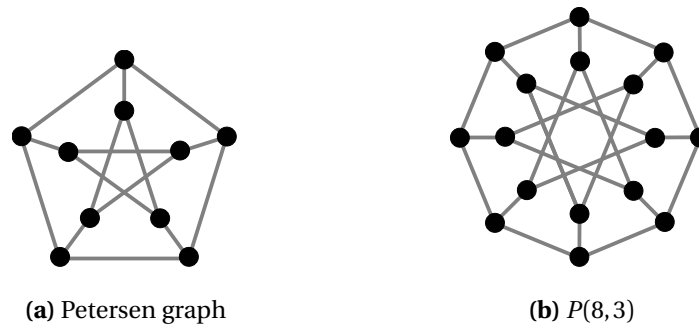


Figure 2.8: Examples for generalized Petersen graphs: (a) shows a drawing of the original Petersen graph and (b) a drawing of $P(8,3)$.

G_1 and G_2 is the graph $G_1 \times G_2 = (V_1 \times V_2, E)$, where

$$E = \{((u, v), (u, v')) \mid u \in V_1, (v, v') \in E_2\} \cup \{((u, v), (u', v)) \mid v \in V_2, (u, u') \in E_1\}.$$

Figure 2.7 illustrates the definition of the product graph with an example. The vertices of $G_1 \times G_2$ are arranged as a 5×4 matrix (corresponding to the 5 vertices of G_1 and the 4 vertices of G_2) such that each row represents G_1 and each column G_2 .

The *Petersen graph* (see Figure 2.8) occupies an important position in the development of several areas of modern graph theory, because it often appears as a counter-example to important conjectures. There is even a book by Holton and Sheehan [1993] completely devoted to this topic.

The Petersen graph has been generalized to a whole family of graphs which is referred to as generalized Petersen graphs. Let ℓ be a positive integer. The *generalized Petersen graph* $P(m, \ell)$ is obtained as follows. Start with an m -cycle $C = (v_0, \dots, v_{m-1})$ and m additional vertices w_0, \dots, w_{m-1} not in C . Then insert the edges (v_i, w_i) and $(w_i, w_{i+\ell})$ for each $i = 0, \dots, m$, where indices are read modulo m . Figure 2.8(b) shows a drawing of the graph $P(8, 3)$. Using this notation, $P(5, 2)$ yields the original Petersen graph shown in Figure 2.8(a).

Chapter 3

Graph Decomposition

Everything should be made as simple as possible, but not one bit simpler.

ALBERT EINSTEIN (1879 – 1955)

The connectivity structure of a graph has been well studied in graph theory. One of the key questions was how to decompose a graph into non-decomposable constituents according to its connectivity. Many fundamental results have already been contributed in the 1930s by Kuratowski and Whyburn [1930], Whitney [1932a,b], and MacLaine [1937]. Though the theory is applicable to general graphs, these results show in particular that the connectivity structure of a graph has important consequences for planar graphs. In fact, it is the key for enumerating all possible embeddings of a planar graph. The algorithmic complexity for the 2- and 3-connected case has been explored in the early 70s, when Hopcroft and Tarjan published optimal algorithms for finding the biconnected [Tarjan, 1972] and triconnected components [Hopcroft and Tarjan, 1973a] of a graph.

In Sections 3.1 through 3.3, we introduce the graph decomposition theory for general graphs. However, our main focus is on planar graphs and their embeddings. In particular, we present data structures that efficiently represent all embeddings of a planar graph, namely the BC-tree and the SPQR-tree, and give algorithms with optimal time and space complexity for their construction. These data structures will play a key role in the algorithms presented in Chapter 4 and 5.

Our main contributions are a thorough introduction into the theory of triconnected components showing the relationships between the different approaches by MacLaine [1937] / Tutte [1966] and the SPQR-tree data structure introduced by Di Battista and Tamassia [1989] (see Section 3.2 through 3.3), as well as non-trivial corrections for the linear-time triconnectivity decomposition algorithm by Hopcroft and Tarjan [1973a] (see Section 3.4).

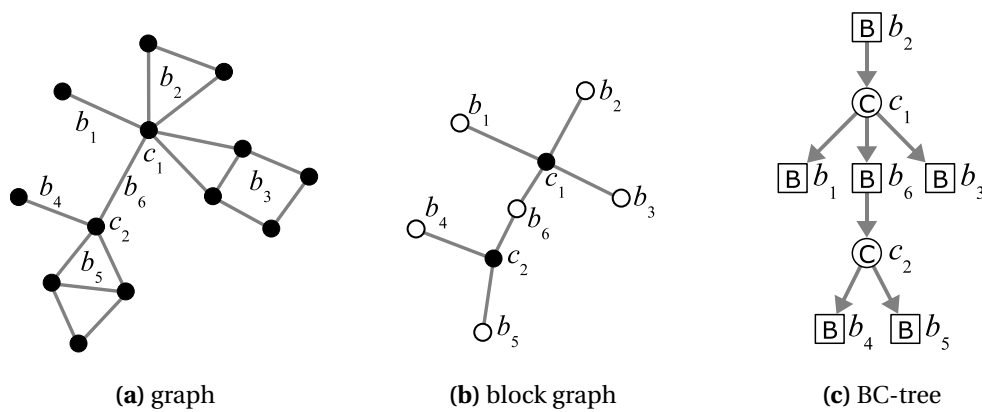


Figure 3.1: A graph, its block graph, and the BC-tree that results from rooting the block graph at block b_2 .

3.1 Blocks and BC-Trees

Let $G = (V, E)$ be a graph. A *block* of G is a maximal connected subgraph that does not contain a cut vertex. That means, every block of G is either a maximal biconnected subgraph, a bridge, or an isolated vertex. Vice versa, every such subgraph is a block of G . Different blocks may share at most a single vertex, which is then a cut vertex of G , and the blocks of G partition the edge set E .

The blocks describe a coarse structure of G , which consists not only of the blocks themselves but also their intersection at cut vertices. Let C denote the set of cut vertices of G and B the set of its blocks. Then, we have a natural bipartite graph on $C \cup B$ formed by the edges (c, b) with c is contained in block b . We call this graph the *block graph* of G . Obviously, the block graph is a forest and unique.

Assume now that G is connected. In this case, the block graph of G is a tree. We root this tree at an arbitrary block $b \in B$ and call the resulting rooted tree a *BC-tree* of G . We distinguish between two kinds of nodes: *B-nodes* representing blocks of G and *C-nodes* representing cut vertices of G . Figure 3.1 shows an example of a connected graph and its BC-tree. BC-trees will play an important role in the algorithms presented in the following chapters.

An important result by Whitney relates the block structure of a graph with the property of being planar.

Theorem 3.1 (Whitney 1932a). *A graph G is planar if and only if each block of G is planar.*

3.1.1 Finding Blocks

Hopcroft and Tarjan [1973b], Tarjan [1972] presented a linear time and space algorithm for finding the blocks of a graph, where they exploit some properties of the DFS-tree of a graph. Consider the DFS-tree of a connected graph G and let

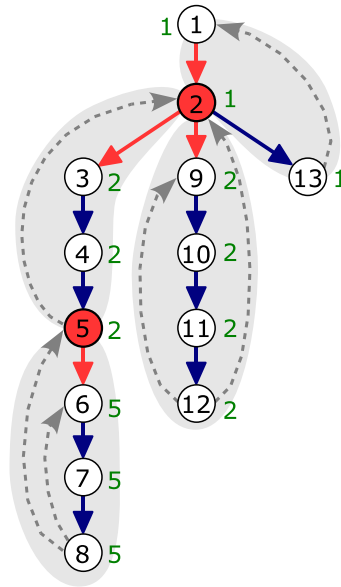


Figure 3.2: DFS-tree of the graph shown in Figure 2.1 with annotated *lowpt*-values; the two cut vertices and four blocks are highlighted.

$number[v]$ denote the DFS-number of a vertex $v \in G$. For a vertex v of G , we define the value $lowpt(v)$ to be the lowest DFS-number of a vertex reachable by traversing zero or more T-arcs followed by one B-arc, or the DFS-number of v if no such vertex exists:

$$lowpt(v) = \min \left(\{number[v]\} \cup \{number[w] \mid v \xrightarrow{*} w\} \right)$$

The following lemma gives us an easy to test condition for cut vertices in G using the *lowpt* values of vertices.

Lemma 3.1 (Lemma 5 in Tarjan 1972). *Let c, v, w be distinct vertices of G such that $c \rightarrow v$ and w is not a descendant of v . If $lowpt[v] \geq number[c]$, then c is a cut vertex of G and removing c disconnects v and w . Conversely, if c is a cut vertex of G , then there exist vertices v and w satisfying the properties above.*

We can find the blocks of G by a bottom-up traversal of the DFS-tree. Suppose that c and v are vertices as defined in Lemma 3.1, and that there is no cut vertex c' with $c \rightarrow c'$. Then, the block containing edge (c, v) is induced by the vertices $\{c\} \cup \{u \mid v \xrightarrow{*} u\}$. Moreover, the vertices u with $v \xrightarrow{*} u$ will not appear in any further block. If we cut the tree at $c \rightarrow v$, we can climb up the tree until the root and iteratively use this condition to identify the blocks.

As an example, consider Figure 3.2. The vertices are labeled with their DFS-numbers and annotated with their *lowpt*-values. First, we find the block containing $5 \rightarrow 6$, which is induced by the vertices 5, 6, 7, and 8. Then we cut the tree at $5 \rightarrow 6$ and find the block containing $2 \rightarrow 3$ (induced by 2, 3, 4, and 5), and so on. In the following, we describe an algorithm that implements this idea using a stack

of vertices, so that the vertices in the next block are always the topmost vertices on the stack.

The algorithm in Listing 3.1 computes the blocks of a (not necessarily connected) graph G without self-loops. It efficiently computes the *lowpt*-values using the following formula:

$$\text{lowpt}[v] := \min \left(\{ \text{number}[v] \} \cup \{ \text{lowpt}[w] \mid v \rightarrow w \} \cup \{ \text{number}[w] \mid v \leftrightarrow w \} \right)$$

The implementation performs this computation at line 8 (initialization), line 13 (T-arc), and line 15 (B-arc). The condition of Lemma 3.1 is tested in line 18. At this stage, we need the set of vertices that induce the corresponding block. For this purpose, we maintain a stack of vertices *calledVertices*, on which we put the vertices in the order they are processed by DFS. Hence, the topmost vertices on this stack are the vertices we have to consider. This is done in the **repeat**-loop starting at line 19. Then, these vertices are removed from the stack which corresponds to pruning the subtree.

The following table summarizes the variables used in Algorithm 3.1.

variable	purpose
$B[i]$	Output of the algorithm. Stores the set of edges in the i -th block.
<i>nextNum</i>	The next DFS number.
<i>nextComp</i>	The next block number.
$\text{number}[v]$	DFS number of vertex v .
$\text{lowpt}[v]$	The <i>lowpt</i> value of vertex v as defined above.
<i>calledVertices</i>	A stack containing vertices of G by decreasing DFS number (top element has highest number).

3.2 Triconnected Components

MacLaine [1937] was the first to publish a paper on the triconnectivity structure of a graph. His goal was to decompose a biconnected graph into certain maximal triply¹ connected subgraphs called *atoms* as an analog to the decomposition of a 1-connected graph into its blocks. Apart from his main goal—the identification of the intrinsic triply connected subgraphs—his theory also revealed two additional types of structures making up a biconnected graph: parallel and serial structures². MacLaine also applied his theory to planar biconnected graphs

¹MacLaine's definition of triply connected is not equivalent to triconnectivity as defined in Chapter 2. He calls a graph triply connected if it is a subdivision of a triconnected graph.

²Parallel structures are called *branch graphs* in his theory, whereas serial structures occur only in form of paths in branch graphs and atoms.

Input: Graph $G = (V, E)$ without self-loops
Output: edges of blocks of G in $B[0], \dots, B[\text{nextComp} - 1]$

```

1: var Stack calledVertices
2: nextNum := nextComp := 0
3: for each  $v \in V$  do number[ $v$ ] := 0
4: for each  $v \in V$  do
5:   if number[ $v$ ] = 0 then DFS-BC( $v$ , nil)
6: end for

7: procedure DFS-BC(vertex  $v$ , vertex parent)
8:   number[ $v$ ] := lowpt[ $v$ ] := nextNum := nextNum + 1
9:   calledVertices.push( $v$ )
10:  for all edges  $e = (v, w)$  incident to  $v$  do
11:    if number[ $w$ ] = 0 then
12:      DFS-BC( $w$ ,  $v$ )
13:      lowpt[ $v$ ] := min(lowpt[ $v$ ], lowpt[ $w$ ])
14:    else if number[ $w$ ] < number[ $v$ ] then
15:      lowpt[ $v$ ] := min(lowpt[ $v$ ], number[ $w$ ])
16:    end if
17:  end for
18:  if parent  $\neq$  nil and lowpt[ $v$ ]  $\geq$  number[parent] then
19:    repeat
20:       $w :=$  calledVertices.pop()
21:      for all edges  $e = (w, u)$  incident to  $w$  do
22:        if number[ $w$ ] > number[ $u$ ] then
23:          add  $e$  to block  $B[\text{nextComp}]$ 
24:        end if
25:      end for
26:    until  $v = w$ 
27:    nextComp := nextComp + 1
28:  end if
29: end procedure

```

Listing 3.1: Finding the blocks of a graph.

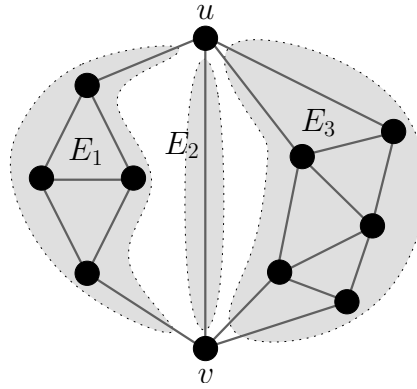


Figure 3.3: The split classes E_1, E_2, E_3 of a split pair $\{u, v\}$.

and studied the relationship between the triconnectivity structure and combinatorial embeddings. In fact, he was the first to give a formula for the number of combinatorial embeddings that a biconnected graph admits.

MacLane's theory of triconnected components was refined by Tutte [1966]. Tutte introduced a further level of abstraction: the concept of *virtual edges*. A virtual edge is a placeholder for a subgraph that is only attached to the rest of the graph at the two endpoints of the virtual edge. In MacLane's theory, a virtual edge is represented by a path in the respective subgraph that connects the two endpoints of the virtual edge. This additional mean of abstraction gives two important benefits: On the one hand the resulting triconnected components are significantly more compact, which turns out to be favorable for data structures and efficient algorithms, and on the other hand the representation is unique, whereas MacLane's theory allows to chose an arbitrary path in the subgraph represented by a virtual edge.

The theory of triconnected components presented in this section is based on Tutte's theory. Let $G = (V, E)$ be a not necessarily simple graph without cut vertices and self-loops. A *separation pair* of G is a pair of vertices $\{u, v\} \in V$ whose removal disconnects G , that is, $G - \{u, v\}$ is not connected. We call $\{u, v\}$ a *split pair* if $\{u, v\}$ is a separation pair or a pair of adjacent vertices.

Consider a split pair $\{u, v\}$. We can partition the edges of G into E_1, \dots, E_k such that two edges belong to the same set E_i if and only if they lie on a path not containing any vertex of $\{u, v\}$ except as an endpoint. We call the sets E_1, \dots, E_k the *split classes* of the split pair $\{u, v\}$ and the graphs induced by the split classes the *split class graphs*. Each split class graph G_i is either

- a graph consisting of the vertices $\{u, v\}$ joined by a single edge, or
- a maximal connected subgraph $G' \subseteq G$ such that neither (u, v) is an edge of G' , nor $\{u, v\}$ is a separation pair of G' .

Moreover, all these graphs have exactly u and v in common, that is, $G_i \cap G_j = (\{u, v\}, \emptyset)$ for $i \neq j$. Figure 3.3 shows an example for the split classes of a split pair $\{u, v\}$.

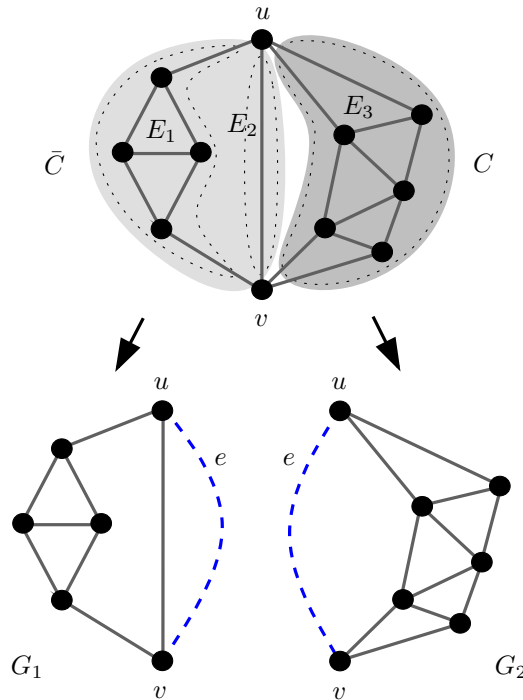


Figure 3.4: A Tutte split applied to the graph shown in Figure 3.3, where the edges are partitioned into $C := E_3$ and $\bar{C} := E_1 \cup E_2$.

It is a natural idea to break up the graph at split pairs, until the remaining pieces are of a basic, unbreakable structure. For this decomposition process, we define the following split operation that breaks up a graph into two graphs.

Definition 3.1. Let $\{u, v\}$ be a split pair with split classes E_1, \dots, E_k , and let $C = E_1 \cup \dots \cup E_j$ and $\bar{C} = E \setminus C$ be such that $|C| \geq 2$ and $|\bar{C}| \geq 2$. A *split operation* $s(u, v, \ell)$ applied to G replaces G by two new graphs $G_1 := (V(C), C \cup e)$ and $G_2 := (V(\bar{C}), \bar{C} \cup e)$, where $e := (u, v, \ell)$ is a new edge with label ℓ . We call e a *virtual edge* and the two created graphs G_1 and G_2 *split graphs*. If, in addition, C is a single split class, say E_a , and the subgraph induced by C or \bar{C} contains no cut vertex, then we call the split operation a *Tutte split*.

The label ℓ assigned to the virtual edges is important. Its purpose is to identify the particular split operation and to distinguish e from edges in G . Suppose now we start with graph $G = (V, E)$ and perform a number of split operations. It is evident that we can only perform a finite number of split operations, since a split operation results in two split graphs each having less edges than the graph that has been split. The resulting split graphs contain virtual edges as well as edges of G .

Observation 3.1. *Each edge of G is contained in exactly one, and each virtual edge in exactly two split graphs.*

Proof. We prove the lemma by induction on the number of split operations k . If $k = 0$, there is only G and the assumption holds obviously.

Let the assumption be true for $k - 1 \geq 0$, and let G_1, \dots, G_k be the split graphs after $k - 1$ split operations. Suppose G_i is split next into split graphs G' and G'' . Then, each edge of G that is also contained in G_i is contained either in G' or in G'' , and in no other graph G_j with $j \neq i$. Each virtual edge contained in G_i is contained in either G' or G'' , and in exactly one more graph $G_j \neq G_i$. The newly created virtual edge is contained in both G' and G'' and in no other graph. Finally, the virtual and non-virtual edges in all graphs G_j with $j \neq i$ are untouched. Hence, the assumption holds for k , too. \square

Observation 3.2. *Each split graph contains no cut vertex and at least three edges.*

Proof. Let $\{u, v\}$ be the split pair and C and $\bar{C} := E \setminus C$ the respective edge sets. Since $|C|, |\bar{C}| \geq 2$ and each split graph contains an additional edge (u, v) , each split graph clearly contains at least three edges.

We denote with G' any of the two split graphs and show that it contains no cut vertex. Let p, q be two arbitrary vertices of G' . Since G contains no cut vertex and at least four edges, p and q lie on a cycle in G (this follows from Menger's theorem, Theorem 2.1). If we replace every part of the cycle that is not in G' with the virtual edge (u, v) , we get a cycle in G' . Hence, G' contains no cut vertex. \square

Lemma 3.2 (see also [Hopcroft and Tarjan, 1973a]). *The total number of edges in all split graphs is bounded by $3|E| - 6$.*

Proof. We use induction on the number of edges in G . If G has $m = 3$ edges, then G cannot be split, and therefore the total number of edges in all split graphs is $3 \leq 3m - 6$.

Suppose G has $m > 3$ edges and the lemma is true for graphs with at most $m - 1$ edges. If G cannot be split, then the lemma follows immediately. Otherwise, suppose that G is split into G' with $k + 1$ edges and G'' with $m - k + 1$ edges for some k with $2 \leq k \leq m - 2$. By induction, the total number of edges in all split graphs of G' is at most $3(k + 1) - 6 = 3k - 3$, and in G'' at most $3(m - k + 1) - 6 = 3m - 3k - 3$. Therefore, the total number of edges in all split graphs of G is at most

$$(3k - 3) + (3m - 3k - 3) = 3m - 6. \quad \square$$

We are now ready to define the triconnected components of a graph.

Definition 3.2. Let G be a biconnected graph without self-loops. The *triconnected components* of G are the split graphs obtained by successively applying a Tutte split until no more Tutte split is possible.

Figure 3.5 shows a biconnected graph and Figure 3.6 its triconnected components. We can classify the resulting split graphs by their structure.

Theorem 3.2. *Every triconnected component is of one of the following three fundamental types:*

- (a) *a simple, triconnected graph;*

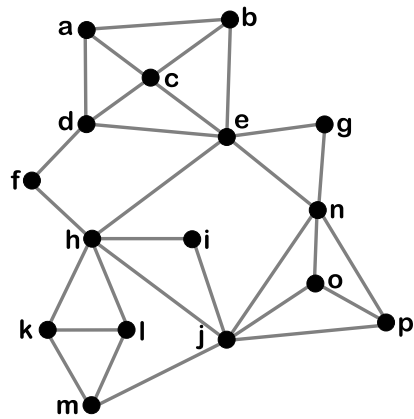


Figure 3.5: A biconnected graph that serves as running example; vertices are labeled a, \dots, p .

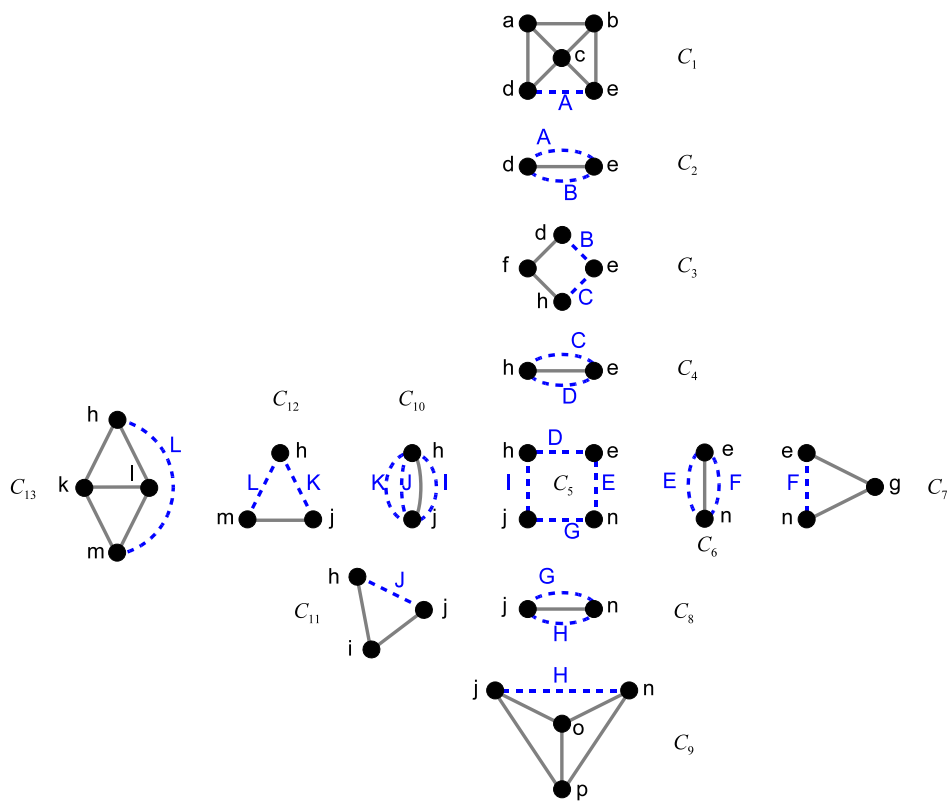


Figure 3.6: The triconnected components of the example graph shown in Figure 3.5; virtual edges are labeled A, \dots, K .

(b) a cycle with at least three edges (also called a polygon); or

(c) a pair of two vertices with at least three parallel edges (also called a bond).

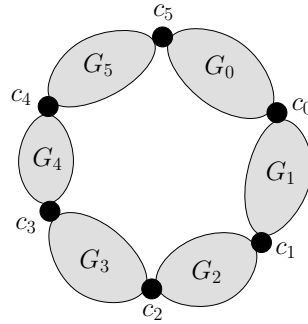
Moreover, every such graph does not admit a Tutte split.

Proof. We first show that, if $G = (V, E)$ is such a graph, no Tutte split is applicable. If G is a bond, then every split class (with respect to the two only vertices in G) consists of a single edge. If G is a cycle, then every split class with more than one edge is a path with two or more edges, and hence the induced graph contains a cut vertex. Let finally G be simple and triconnected. Then, G contains no separation pair, and therefore the split classes of any pair of vertices u, v are either a single edge or the set E' containing all edges of G except edges between u and v . Since G is also simple, there is only one split class $\{(u, v)\}$, and hence no Tutte split can be applied because $E \setminus E'$ contains only one edge.

Suppose now G is of none of the types above. We have to show that a Tutte split is applicable. According to Observation 3.2, G contains no cut vertex and at least three edges. First, we assume that G contains a multiple edge (u, v) . Since G is not a bond, there must be a split class C of (u, v) with $|C| \geq 2$. We denote with $\bar{C} := E \setminus C$ the remaining edges. Obviously, $(u, v) \in \bar{C}$ and $|\bar{C}| \geq 2$. Moreover, the graph induced by \bar{C} contains no cut vertex since $(u, v) \in \bar{C}$. Hence, we can apply a Tutte split at (u, v) .

Second, we consider the case where G is simple. Since G is not triconnected, G must contain separation pairs. Again, we distinguish two cases.

Case 1: Suppose G has the following structure:



Let c_0, \dots, c_{k-1} be $k \geq 3$ vertices, and let G_0, \dots, G_{k-1} be k subgraphs of G such that, for $0 \leq i \leq k-1$, G_i contains no cut vertex, $G_i \cap G_{i+1} = (\{c_i\}, \emptyset)$ and $G_i \cap G_j = \emptyset$ for $j \notin \{i-1, i, i+1\}$, where indices are read modulo k . Since G is not a polygon, there must be an i_0 such that G_{i_0} contains at least two edges, and hence a Tutte split at (c_{i_0}, c_{i_0+1}) is possible.

Case 2: G is not of that structure. Let $\{u, v\}$ be a separation pair and let E_j be a split class with $|E_j| \geq 2$. If both the graph induced by E_j and the graph induced by $E \setminus E_j$ would contain a cut vertex, then G would be of the structure in case 1. Hence, we have a Tutte split at $\{u, v\}$ with split class E_j .

□

In general, we have many choices how to carry out Tutte splits on a graph. However, MacLane's and Tutte's theories show that the resulting triconnected components we obtain are always the same. We give the following result without a proof:

Theorem 3.3 (MacLaine 1937, Tutte 1966). *The triconnected components of G are unique.*

Lemma 3.2 directly implies an upper bound for the size of the triconnected components:

Corollary 3.1. *Let $G = (V, E)$ be a biconnected graph. The total number of edges in all triconnected components of G is at most $3|E| - 6$.*

In particular, this property allows to develop a data structure for the representation of the triconnected components of a graph G whose size is only linear in the size of G . We present such a data structure in the following section.

The virtual edges establish a relationship between the triconnected components just as the cut vertices do for the blocks. By Observation 3.1, we know that a virtual edge is contained in exactly two triconnected components. This defines a graph $T(G)$ as follows. The vertices of $T(G)$ are the triconnected components of G , and there is an edge between two triconnected components C and C' if there is a virtual edge $e = (u, v, \ell)$ contained in C and C' .

Theorem 3.4 (Tutte 1966). *$T(G)$ is a tree.*

Proof. We use induction on the number of triconnected components k . For $k = 1$, there is only a single triconnected component and $T(G)$ consists of a single vertex.

Suppose now, that G has k triconnected components, and that $T(G')$ is a tree if G' has at most $k - 1 \geq 1$ triconnected components. Since $k \geq 2$, we can perform a Tutte split $s(u, v, \ell)$ on G yielding two graphs G_1 and G_2 , and the triconnected components of G are the union of the disjoint triconnected components C_1, \dots, C_n of G_1 and $C'_1, \dots, C'_{n'}$ of G_2 . Obviously, both G_1 and G_2 have at most $k - 1$ triconnected components, so $T(G_1)$ and $T(G_2)$ are trees by assumption. The virtual edge (u, v, ℓ) corresponding to the Tutte split is contained in C_i and C'_j for some i and j , so $T(G)$ results from $T(G_1)$ and $T(G_2)$ by joining the two trees with edge (C_i, C'_j) , which results again in a tree. \square

Figure 3.7 shows the tree $T(G)$ of our running example. Tree vertices are labeled C_1, \dots, C_{12} and tree edges are labeled A, \dots, K just as their corresponding virtual edges in Figure 3.6.

3.3 SPQR-Trees

The SPQR-tree data structure has been introduced by Di Battista and Tamassia [1989]. In [Di Battista and Tamassia, 1989, 1996a], the authors use SPQR-trees for the representation of the set of all embeddings of a planar biconnected

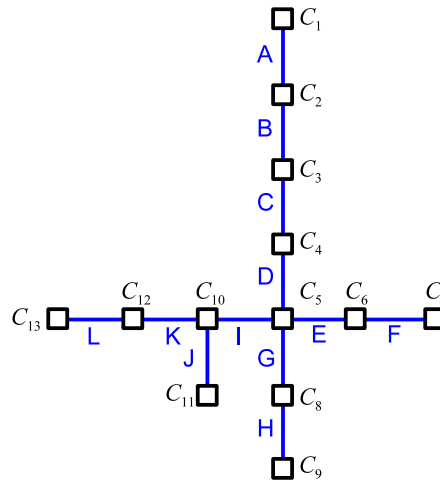


Figure 3.7: The tree $T(G)$ of our running example resulting from the triconnected components depicted in Figure 3.6.

graph, and they equip the data structure with efficient dynamic update operations. Since then, SPQR-trees evolved to an important data structure in the field of graph algorithms—not only for using their dynamic update capabilities but also for solely representing all embeddings of a graph. Many linear time algorithms that work for triconnected graphs only can be extended to work for bi-connected graphs using SPQR-trees [for example, Bertolazzi et al., 1998, Kant, 1996]. Often it is essential to represent the set of all embeddings of a planar graph, for example, in order to optimize a specific criteria over all planar embeddings [Gutwenger et al., 2005, Mutzel and Weiskircher, 1999, Bertolazzi et al., 2000, Bienstock and Monma, 1990], or for testing cluster planarity [Lengauer, 1989, Dahlhaus, 1998, Gutwenger et al., 2003, 2002]. The dynamic (or incremental) update features of SPQR-trees are exploited in a variety of on-line graph algorithms dealing with triconnectivity, transitive closure, minimum spanning tree, and planarity testing [Di Battista and Tamassia, 1990]. In this thesis, we restrict us to the static case. The algorithms presented in Chapter 4 and 5 will intensively utilize the static SPQR-tree data structure.

SPQR-trees are closely related to the triconnected components of a graph. We will see that they are merely an enriched version of the triconnected components, and that it is easy to construct the SPQR-tree data structure once we have computed the triconnected components. SPQR-trees were originally defined in [Di Battista and Tamassia, 1989] for planar graphs only, but have been generalized in [Di Battista and Tamassia, 1996b] to general graphs. Our definition is equivalent to the general definition.

Let G be a connected graph without cut vertices and self-loops, and let $\{s, t\}$ be a split pair of G . The *pertinent graph* of a split pair $\{u, v\}$ with respect to an edge $e \in E$ is the union of the split class graphs of $\{u, v\}$ but the one containing e . We say a split pair $\{u, v\}$ is *dominated* by another split pair $\{x, y\}$ with respect

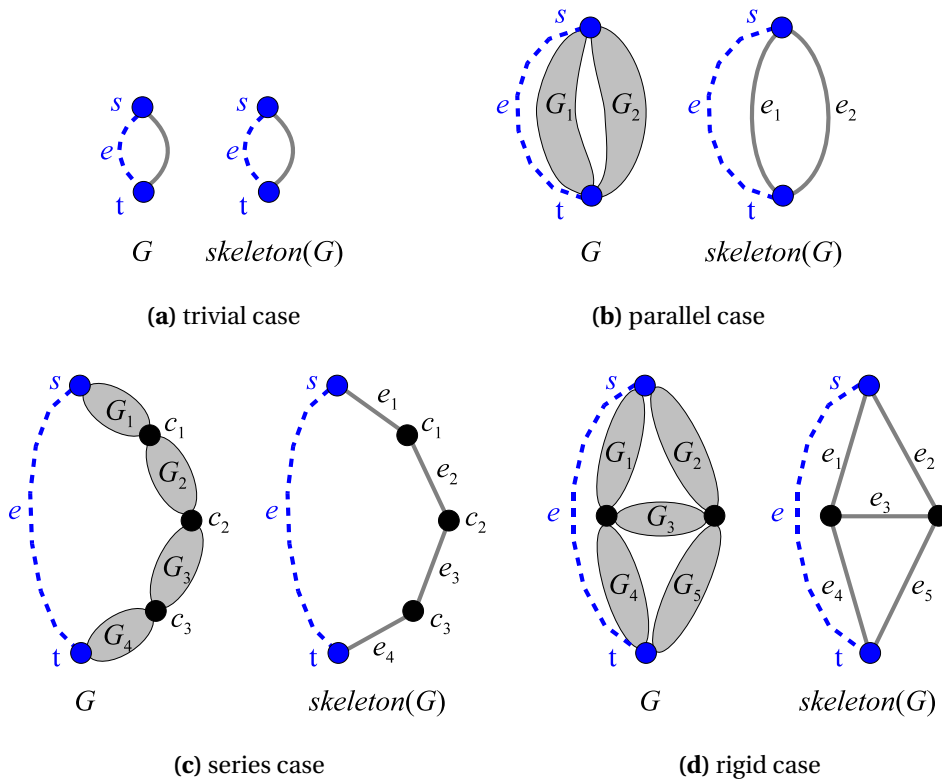


Figure 3.8: The different cases in the definition of SPQR-trees.

to an edge e if the pertinent graph of $\{u, v\}$ with respect to e is a proper subgraph of the pertinent graph of $\{x, y\}$ with respect to e .

Observation 3.3. *The dominance relation between split pairs with respect to an edge e is irreflexive, antisymmetric, and transitive.*

Proof. The observation follows directly from the fact that the proper subgraph relation is irreflexive, antisymmetric, and transitive. \square

This property of the dominance relation allows us to define maximal split pairs. A *maximal split pair* $\{u, v\}$ of G with respect to $\{s, t\}$ is a split pair of G that is not dominated by any other split pair of G with respect to (s, t) .

In order to define the SPQR-tree data structure, we first define an auxiliary construction called *pre-SPQR-tree* which allows an easy recursive definition.

Definition 3.3 (Pre-SPQR-tree). Let $e = (s, t)$ be an edge of G , called the *reference edge*. The *pre-SPQR-tree* \mathcal{T} of G with respect to e 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 graph called the *skeleton* of μ . We denote with $G' := G - e$ the graph resulting from removing edge e in G . Tree \mathcal{T} is recursively defined as follows (compare also Figure 3.8):

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 $\{s, t\}$ is a split pair in G' with $k \geq 2$ split class graphs G_1, \dots, G_k , the root of \mathcal{T} is a P-node μ , whose skeleton consists of $k + 1$ parallel edges e and e_1, \dots, e_k between s and t .

Series Case: Otherwise, the split pair $\{s, t\}$ has exactly two split class graphs, one of them is e , and the other one is G' . Suppose that G' contains cut vertices. Since $G' \cup (s, t)$ contains no cut vertex, the block graph of G' is a chain $G_1, c_1, G_2, \dots, c_{k-1}, G_k$ with $s \in G_1$, $t \in G_k$, and $k \geq 2$. In this case, the root of \mathcal{T} is an S-node μ , whose skeleton is the cycle e_0, e_1, \dots, e_k , where $e_0 = e$, $c_0 = s$, $c_k = t$, and $e_i = (c_{i-1}, c_i)$ for $i = 1, \dots, k$.

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

Except for the trivial case, μ has children μ_1, \dots, μ_k , such that μ_i is the root of the pre-SPQR-tree of $G_i \cup e_i$ with reference edge e_i ($i = 1, \dots, k$). The endpoints of edge e_i are called the *poles* of node 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 skeleton of μ .

The skeleton graphs of a pre-SPQR-tree of G contain two types of edges: edges of the graph G and newly created edges. We call the edges that are newly created during the construction *virtual edges* and the edges originating from G *real edges*. Using the auxiliary definition of pre-SPQR-tree, it is easy to define the SPQR-tree.

Definition 3.4 (SPQR-tree). The *SPQR-tree* \mathcal{T} of G with respect to edge (s, t) is obtained from the pre-SPQR-tree \mathcal{T}' of G as follows. The root of \mathcal{T} is a new Q-node whose skeleton consists of the real edge (s, t) and a virtual edge between s and t and whose single child is the root of \mathcal{T}' .

Let e be an edge in skeleton of μ and v the pertinent node of e . Deleting edge (μ, v) in \mathcal{T} splits \mathcal{T} into two connected components. Let \mathcal{T}_v be the connected component containing v . The *expansion graph* of e (denoted with $\text{expansion}(e)$) is the graph induced by the edges that are represented by the Q-nodes in \mathcal{T}_v . We further introduce the notation $\text{expansion}^+(e)$ for the graph $\text{expansion}(e) + e$.

Replacing a skeleton edge e by its expansion graph is called *expanding e* . The *pertinent graph* of a tree node μ results from expanding all edges in $\text{skeleton}(\mu)$ except for the reference edge of μ and is denoted with $\text{pertinent}(\mu)$. Hence, if e is a skeleton edge and v its pertinent node, then $\text{expansion}^+(e)$ equals $\text{pertinent}(v)$.

Let v be a vertex of G . The *allocation nodes* of v are the nodes of \mathcal{T} whose skeleton contains v . The least common ancestor μ of the allocation nodes of v is itself an allocation node of v and is called the *proper allocation node* of v .

Observation 3.4. *Two S-nodes cannot be adjacent in \mathcal{T} . Two P-nodes cannot be adjacent in \mathcal{T} .*

Proof. Let μ be a P-node with poles s, t and let G_1, \dots, G_k be the split components of the split pair $\{s, t\}$ according to the definition of pre-SPQR-tree (parallel case). If μ has a child which is also a P-node, then there is a split component G_i which is not a single edge and for which $\{s, t\}$ is a split pair. This contradicts the definition of split component.

Let μ be an S-node with poles $\{s, t\}$ and let G_1, \dots, G_k be the blocks of G' according to the definition of pre-SPQR-tree (series case). Since for each $1 \leq i \leq k$, the graph G_i without the edge $e = (s, t)$ is already biconnected, no child of μ can be an S-node. \square

The following theorem establishes the elementary relationship between SPQR-trees and triconnected components. It shows in particular, how to build up the SPQR-tree data structure from the triconnected components.

Theorem 3.5. *Let G be a biconnected graph and \mathcal{T} an SPQR-tree of G . Then, the following statements are true.*

- (a) *The Q-nodes of \mathcal{T} are in one-to-one correspondence to the edges of G .*
- (b) *The skeleton graphs of the S-, P-, and R-nodes of \mathcal{T} are the triconnected components of G . P-nodes correspond to bonds, S-nodes to polygons, and R-nodes to triconnected simple graphs.*
- (c) *The graph obtained from \mathcal{T} by removing the Q-nodes is the tree $T(G)$ formed by the triconnected components (as defined in Section 3.2).*

Proof. Let e_r be the reference edge of \mathcal{T} .

- (a) We consider the construction of the pre-SPQR-tree \mathcal{T}' with respect to e_r . Each non-trivial case of the decomposition, say for decomposing graph G_α with reference edge e_α , creates a skeleton that contains only e_α plus $k \geq 2$ virtual edges, but not any real edge. The definition is recursively applied to graphs G_1, \dots, G_k that partition the real edges $\neq e_r$ in G_α . It follows for the decomposition of G , that each real edge $\neq e_r$ is contained in the skeleton of a Q-node. On the other hand, each Q-node contains exactly one virtual edge (its reference edge) and one real edge. Hence, we have a one-to-one correspondence between real edges $\neq e_r$ and Q-nodes in \mathcal{T}' . The Q-node that is added to \mathcal{T}' for obtaining \mathcal{T} is the counterpart for the reference edge e_r of \mathcal{T} which completes the proof.
- (b) We consider the SPQR-tree without Q-nodes, that is, we omit splitting off single edges in the construction of the pre-SPQR-tree and leave these edges in the skeleton graph instead. We have to show, that the resulting skeleton graphs are the triconnected components of G .

In the parallel, series, and rigid case in the definition of the pre-SPQR-tree, subgraphs G_1, \dots, G_k are considered. Assume w.l.o.g. that G_1, \dots, G_ℓ each contain more than one edge, and that $G_{\ell+1}, \dots, G_k$ each contain exactly one edge. In each of the three cases, the recursive decomposition step can be realized by performing ℓ split operations, each splitting off one G_i , $1 \leq i \leq \ell$, and introducing a new virtual edge e_i in the skeleton of node μ and the skeleton of a child μ_i of μ . We have to show that these split operations are Tutte splits.

We denote with E_i the edges in G_i and consider a split operation for some E_i with $1 \leq i \leq \ell$. Let E' be the remaining edges not in E_i (these are the edges in E_j with $j \neq i$ or a virtual edge that resulted from splitting of E_j), and let H be the graph induced by $E_j \cup E'$. By construction, we have $|E_i| \geq 2$, $|E'| \geq 2$, and we split off a single split class. It remains to show that the graph G_i induced by E_i or the graph G' induced by E' contains no cut vertex.

In the parallel case, E' consists of at least two split classes, and hence G' contains no cut vertex. In the series case, each graph G_i contains no cut vertex by definition. In the rigid case, G_i and G' cannot both contain a cut vertex, since then the series case would apply.

It follows that the skeletons of S-, P-, and R-nodes are obtained by a sequence of Tutte splits. By construction, the S-nodes are cycles with at least three edges and the P-nodes are bonds with at least three parallel edges. So consider the skeleton S of an R-node that is created by splitting a graph H with reference edge (s, t) . It is easy to see that S has at least four vertices, otherwise a different case would apply. We first observe that S contains no multiple edges, since such edges are either replaced by a single edge or not both of their endpoints are contained in S . Hence, it is sufficient to show that S contains no separation pair. Assume that $\{x, y\}$ is a separation pair in S . By construction, $\{x, y\}$ is also a separation pair in H . If $\{x, y\}$ is maximal with respect to (s, t) , then its pertinent graph would be replaced by a single edge (x, y) and $\{x, y\}$ is not a separation pair in S . If $\{x, y\}$ is not maximal, then there must be maximal split pair $\{u, v\}$ with respect to (s, t) in H such that its pertinent graph contains the pertinent graph of $\{x, y\}$. But this implies that not both x and y can be contained in S . It follows that S contains no separation pair.

- (c) According to the definition of $T(G)$, its vertices are the triconnected components of G and two triconnected components are connected by an edge in $T(G)$ if both contain a virtual edge created by the same Tutte split. On the other hand, the edges of \mathcal{T} correspond to the parent-child relationship in the tree.

We have already shown in (b) that each step of the recursive definition of \mathcal{T} corresponds to several Tutte splits. It is easy to see that the created parent-child relationships correspond to the edges of $T(G)$. \square

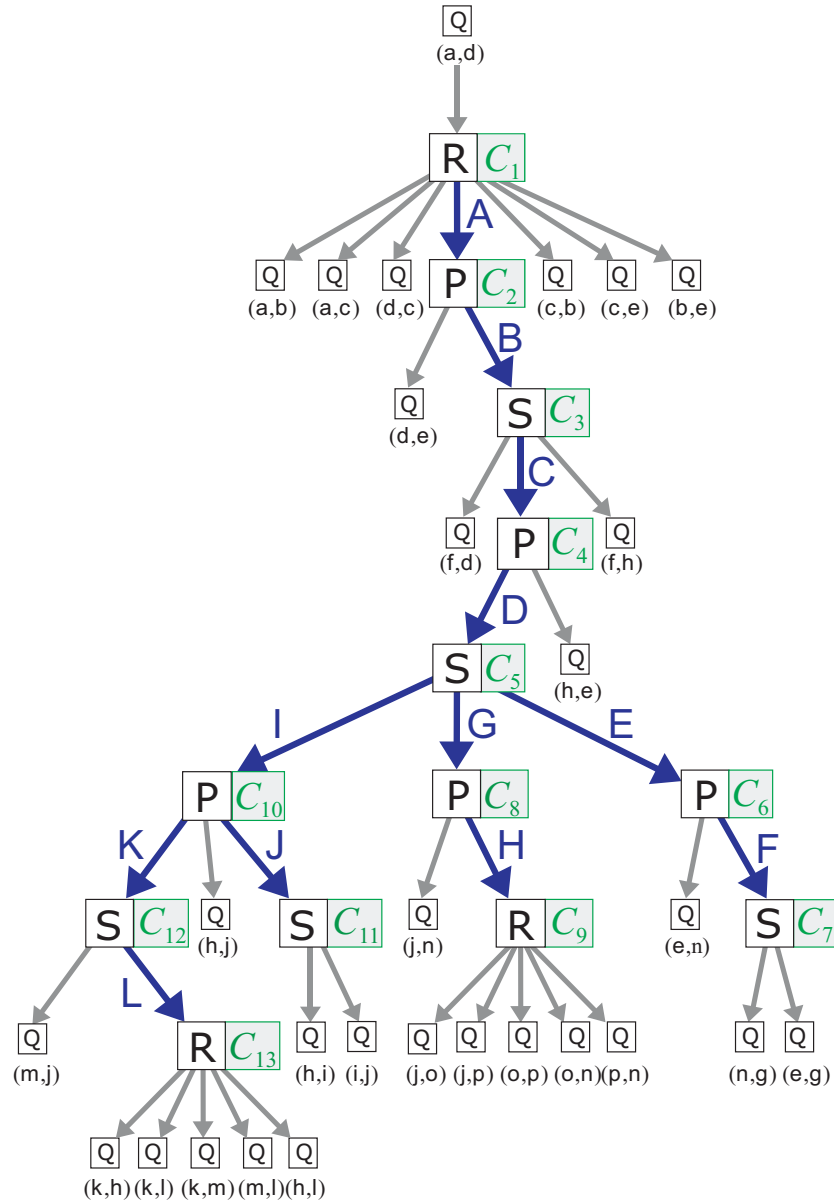


Figure 3.9: The SPQR-tree \mathcal{T} of our running example (graph G in Figure 3.5). The Q-nodes are labeled with their corresponding edges of G ; T-arcs corresponding to virtual edges are labeled A, \dots, K as in Figure 3.6; the skeleton graphs of S-, P-, and R-nodes are the triconnected components C_1, \dots, C_{12} shown in Figure 3.6.

Figure 3.9 shows the SPQR-tree \mathcal{T} of our example graph (Figure 3.5) with reference edge (a, d). The relationship to the triconnected components shown in Figure 3.6 are illustrated by annotating the non-Q-nodes with the corresponding triconnected components and highlighting the arcs corresponding to the edges of $T(G)$.

Corollary 3.2. *Let \mathcal{T} be the SPQR-tree of G with respect to e_r . For any edge e of G , we obtain the SPQR-tree of G with respect to e by rooting \mathcal{T} at the Q-node corresponding to e .*

Proof. This follows directly from Theorem 3.5 and the uniqueness of the triconnected components. \square

Lemma 3.3. *Let \mathcal{T} be an SPQR-tree of $G = (V, E)$. Denote with n_S , n_R , n_P , and n_Q the number of S-, R-, P-, and Q-nodes. Then, the following statements are true:*

- (a) $n_Q = |E|$
- (b) $n_S + n_R \leq |V|$
- (c) $n_P \leq |E| - 2$
- (d) $n_P \leq |V|$ if G is simple.
- (e) The total number of skeleton edges in \mathcal{T} is $3n_Q + 2(n_S + n_R + n_P) - 2$.

Proof. (a) follows directly from the one-to-one correspondence between Q-nodes and edges (Theorem 3.5(a)). Since, for each S- and R-node μ , there is a vertex $v \in G$ with proper allocation node μ and a vertex has a unique proper allocation node, we have $n_S + n_R \leq |V|$ and (b) holds.

In order to proof (c), we show for each proper subtree \mathcal{S} of \mathcal{T} that we can assign a Q-node of \mathcal{S} to each P-node in \mathcal{S} , so that no Q-node is assigned to several P-nodes and at least one Q-node is unassigned. We use induction on the height of \mathcal{S} . Let μ be the root of \mathcal{S} .

- If μ is a Q-node, then μ itself is the unassigned node of \mathcal{S} .
- If μ is an S- or R-node, then we chose an unassigned node of a subtree rooted at a child of μ to be the unassigned node of \mathcal{S} .
- If μ is a P-node, then μ has at least two children and there are two unassigned nodes v_1 and v_2 in the respective subtrees. Assign v_1 to μ and let v_2 be the unassigned node of \mathcal{S} .

It follows that there are at most $n_Q - 2$ P-nodes in \mathcal{T} (one Q-node is unassigned and one is the root of \mathcal{T}) and, with $n_Q = |E|$, (c) holds.³

³Di Battista and Tamassia [1996b] claimed that $n_P \leq |V| + 1$ in general (Proof of Lemma 1 in [Di Battista and Tamassia, 1996b]). Their argument was that the parent of a P-node is either an S- or an R-node, and hence there are at most $n_S + n_R + 1$ P-nodes. This argument is wrong since two P-nodes may share a common parent. It is also easy to construct a graph with more than $|V| + 1$ P-nodes if multiple edges are allowed.

Now assume that G is simple and consider a P-node μ . We know that μ has two or more children and none of them is a P-node by Observation 3.4. Since G is simple, at most one child of μ is a Q-node. Hence, μ has a child that is an S- or an R-node. Since no two nodes have a common child, it follows that there are at most $|V|$ P-nodes and (d) holds.

By construction, \mathcal{T} has two skeleton edges for every tree edge plus $|E|$ additional edges (the real edges in the Q-nodes). Since a tree with n nodes has $n - 1$ arcs, we have

$$|E| + 2(n_Q + n_S + n_R + n_P - 1) = 3n_Q + 2(n_S + n_R + n_P) - 2$$

skeleton edges in total, which completes the proof. \square

Corollary 3.3. *Let \mathcal{T} be an SPQR-tree of $G = (V, E)$. The total number of edges in all skeleton graphs of \mathcal{T} is bounded by $5|E| - 6$. If G is simple, then the total number of skeleton edges is also bounded by $3|E| + 4|V| - 2$.*

Proof. We first consider the general case. By Lemma 3.3(a), we have $|E|$ Q-nodes and each Q-node skeleton has two edges. By Theorem 3.5(b) and Corollary 3.1, the number of edges in the skeletons of all S-, P-, and R-nodes is at most $3|E| - 6$. Summing up yields a bound of

$$2|E| + 3|E| - 6 = 5|E| - 6.$$

Now assume that G is simple. By Lemma 3.3, the number of skeleton edges in this case is

$$3n_Q + 2(\underbrace{n_S + n_R}_{\leq |V|} + \underbrace{n_P}_{\leq |V|}) - 2 \leq 3|E| + 4|V| - 2. \quad \square$$

3.4 Linear-Time Construction of SPQR-Trees

In the theoretical papers on SPQR-trees [for example, Di Battista and Tamassia, 1989, 1996b,a], the authors always suggest to construct the data structure in linear time “using a variation of the algorithm of [Hopcroft and Tarjan, 1973a] for finding the triconnected components of a graph...” [Di Battista and Tamassia, 1996a]. However, apart from the implementation reported in [Hopcroft and Tarjan, 1973a], no linear time implementation of the Hopcroft/Tarjan algorithm was known at that time, let alone a linear time implementation of SPQR-trees. To our knowledge, the only correct implementation of SPQR-trees was part of GDTToolkit [see also GDTToolkit], where SPQR-trees are used in connection with a branch-and-bound algorithm to compute an orthogonal drawing of a biconnected planar graph with the minimum number of bends, but this implementation does not run in linear time [Didimo, 1999]. The reason for the lack of a linear time implementation of the Hopcroft/Tarjan algorithm is twofold: On the one hand the algorithm is quite complicated and hard to understand, on the other hand the presentation contains several non-trivial errors.

In this section, we present a linear time implementation of the data structure SPQR-tree based on the algorithm by Hopcroft and Tarjan [1973a] for decomposing a graph into its triconnected components. We identify the incorrect parts and develop a correct algorithm for triconnectivity decomposition by correcting and replacing the faulty parts in [Hopcroft and Tarjan, 1973a]. Section 3.4.6 summarizes the necessary corrections. Finally, we apply it to the computation of SPQR-trees. The implementation presented here is publicly available in the OGDF (Open Graph Drawing Framework) library [Chimani et al., 2010].

We remark that also further algorithms for triconnectivity decomposition exist. Miller and Ramachandran [1992] give a linear time algorithm based on open ear decomposition which can be parallelized. The processor-time product of a parallel implementation on a CRCW PRAM is $\mathcal{O}((|V| + |E|)\log^2 |V|)$ for a graph $G = (V, E)$. Fussell et al. [1993] present an improved, almost work-efficient parallel algorithm with a processor-time product of $\mathcal{O}((|V| + |E|)\log \log |V|)$. However, we are not aware of any public implementation of these algorithms.

3.4.1 Split Components

The triconnected components are obtained by successively applying Tutte splits until no more Tutte split is possible. From an algorithmic point of view, a Tutte split has the drawback that we need to check the additional preconditions (see also Definition 3.1)

- C is a single split class; and
- not both C and \bar{C} may contain a cut vertex.

It would be favorable if we just could perform any split operation. In this case, it is basically sufficient to identify separation pairs and parallel edges. It turns out, that it is easy to recreate the triconnected components from the split graphs obtained from successively applying any split operation until no more such operation is possible. The algorithm presented in this section follows this approach.

Definition 3.5. The *split components* of a biconnected graph G are the split graphs obtained from G by successively applying a split operation until no more split operations are possible.

Each split component is either

- a set of three multiple edges (*triple bond*);
- a cycle of length three (*triangle*); or
- a triconnected simple graph.

It is easy to see that the split components are not necessarily unique. Imagine a cycle $c := v_1, v_2, v_3, v_4$ of length four. We can either split c at the separation pair $\{v_1, v_3\}$ or $\{v_2, v_4\}$, giving us two different variations of the split components.

In order to reconstruct the triconnected components from some split components, we use the merge operation, which is the inverse of a split operation.

Definition 3.6. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two split graphs both containing a virtual edge $e = (u, v, \ell)$ with the same label ℓ . A *merge operation* replaces the two graphs G_1 and G_2 with the graph $G' := (V_1 \cup V_2, (E_1 \cup E_2) \setminus \{e\})$. We also say that we *merge* G_1 and G_2 .

The *triconnected components* of G are obtained from its split components by merging the triple bonds into maximal sets of multiple edges (*bonds*) and the triangles into maximal simple cycles (*polygons*):⁴

Theorem 3.6. *Let G be a biconnected graph, and let G_1, \dots, G_k be the graphs obtained from the split components of G by successively merging two bonds or two polygons until no more such merge operation is possible. Then, G_1, \dots, G_k are the triconnected components of G .*

Proof. Analogously to the tree $T(G)$ defined in Section 3.2, which represents the relationships between the triconnected components via corresponding virtual edges, we can define a tree $T'(G)$ for the split components. Each edge in this tree corresponds to a split operation. We further observe that the order of these split operations is not important for the final decomposition we obtain.

A merge operation is simply the inverse of a split operation, that is, a merge simply contracts an edge in the tree $T'(G)$ and merges the split components that are endpoints of this tree edge. First, consider a merge operation that merges two bonds and denote with C and \bar{C} the partitioning of edges in the corresponding split operation. Then, neither C nor \bar{C} consists of a single split class and thus this split operation violates the first condition of a Tutte split.

Second, we consider a merge operation that merges two polygons. Let again C and \bar{C} be the partitioning of edges in the corresponding split operation. Then, both $G[C]$ as well as $G[\bar{C}]$ contain a cut vertex, and thus this split operation violates the second condition of a Tutte split.

Therefore, we only undo split operations that are not Tutte splits. Let $T''(G)$ be the resulting tree after merging all polygons and all bonds. We show that all edges in $T''(G)$ correspond to Tutte splits. Notice that the split graphs now are either bonds, polygons, or simple triconnected graphs. Consider again an edge connecting two split graphs G_1 and G_2 corresponding to a split operation with edge partitioning C and \bar{C} . The first condition of a Tutte split demands that not both C and \bar{C} consist of more than one split class. This is only possible if both G_1 and G_2 are bonds; however, this cannot be the case, since we have already merged all bonds adjacent in the tree. The second condition demands that not both $G[C]$ and $G[\bar{C}]$ contain a cut vertex. This is only possible if both G_1 and G_2 are polygons; however, again, we would have merged G_1 and G_2 .

Hence, the tree $T''(G)$ represents a decomposition only obtained from applying Tutte splits and none of the split graphs allows to apply further Tutte splits. Since the triconnected components are unique by Theorem 3.3, $T(G) = T''(G)$ and the theorem follows. \square

⁴In [Hopcroft and Tarjan, 1973a], this statement is actually used for defining the triconnected components of a graph. The authors claim that this definition is equivalent to Tutte's definition [Tutte, 1966]. However, no proof for that claim is given in the paper.

The merging of bonds and polygons simply undoes previously applied split operations that were not Tutte splits. Hence, we just need as many merge operations as unnecessary split operations were performed.

3.4.2 Primal Idea

The hard part of the algorithm is to find all separation pairs of G in order to compute the split components of G . In this subsection, we describe the idea behind the respective procedure in the algorithm; the algorithm itself is then presented in detail in the remaining part of this section.

Hopcroft and Tarjan adapted an idea by Auslander and Parter [1961], Goldstein [1963], which gives rise to a linear time planarity testing algorithm [Hopcroft and Tarjan, 1974]. Suppose c is a cycle in G . A subgraph $S \subseteq G$ is called a *segment* relative to c if either S consists of a single edge $e = (u, v)$ with $e \notin c$ and $u, v \in c$, or S is a connected component C_S of $G \setminus c$ plus all edges connecting C_S with c . The following lemma gives us a sufficient condition for developing an efficient algorithm that finds all separation pairs in G .

Lemma 3.4 (Lemma 4 in [Hopcroft and Tarjan, 1973a]). *Let S_1, \dots, S_n be the segments relative to the cycle c . Let $\{a, b\}$ be a separation pair of G such that (a, b) is not a multiple edge. Then the following three statements hold:*

- (a) *a and b both lie on c , or a and b lie in the same segment.⁵*
- (b) *Suppose a and b both lie on c . Let p_1 and p_2 be the two paths comprising c which join a and b . Then*

Type-1 case: *some segment S_i with at least two edges has only a and b in common with c , and some vertex v does not lie in S_i (compare Figure 3.10); or*

Type-2 case: *no segment contains a vertex $v \neq a, b$ in p_1 and a vertex $w \neq a, b$ in p_2 , and p_1 and p_2 each contain a vertex besides a and b (compare Figure 3.11).*

- (c) *Conversely, every pair $\{a, b\}$ satisfying one of the two cases above is a separation pair.*

Proof. (a) The segments S_1, \dots, S_n and the cycle c partition the edges of G into $n+1$ sets. We have to show that deleting two vertices a, b that lie in different segments and do not both lie on c does not disconnect G . Let $a \in S_i$ and $b \in S_j$ with $i \neq j$, and suppose w.l.o.g. that b does not lie on c .

For each segment S_k , we can find a subpath p_k of c , such that $S_k \cup p_k$ is biconnected and $a \notin p_k$ if $a \notin S_k$. Therefore, for each vertex $v \in S_k \setminus (c \cup$

⁵This statement is written as “Either ... or” in [Hopcroft and Tarjan, 1973a], but it is easy to see that a and b can belong to both the cycle c and a segment S_i ; in fact this holds for every type-1 separation pair.

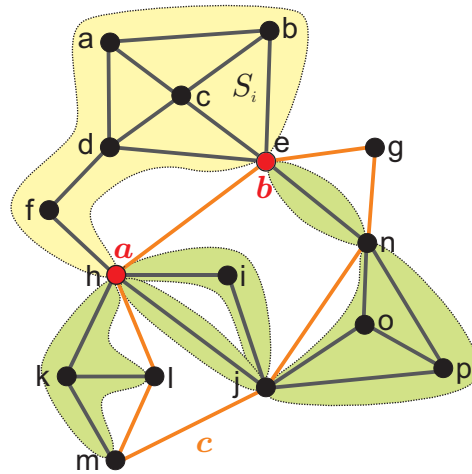


Figure 3.10: Type-1 separation pair $\{a, b\}$ relative to the cycle c . Removing a and b separates the segment S_i .

$\{a, b\}$) there is a path p_v connecting v with c that uses only edges in S_k and contains neither a nor b . Such a path exists because $S_k \cup p_k$ is biconnected and not both a and b lie in $S_k \cup p_k$.

If we delete a and b , the cycle c is not disconnected, and each remaining vertex not on c is still connected to the remaining vertices on c .

- (b) Suppose $\{a, b\}$ is neither a type-1 nor a type-2 separation pair. We show that there is only one separation class with respect to $\{a, b\}$ that contains more than one edge.

Since $\{a, b\}$ is not a type-1 pair, there are at most two separation classes, say E_1 and E_2 , with respect to $\{a, b\}$ which contain more than one edge. E_1 contains p_1 and all segments connected to a vertex on p_1 different from a and b , and E_2 contains p_2 and all segments connected to a vertex on p_2 different from a and b .

If both E_1 and E_2 contain more than one edge, then neither p_1 nor p_2 consists of a single edge. But in this case, there is a segment connected to a vertex $v \neq a, b$ on p_1 and a vertex $w \neq a, b$ on p_2 , since $\{a, b\}$ is not a type-2 pair. Then, it follows that $E_1 = E_2$.

- (c) follows immediately. □

Figure 3.10 shows a type-1 separation pair in the example graph from Figure 3.5. Removing a and b splits off segment S_i . On the other hand, Figure 3.11 shows an example for a type-2 pair. Removing a and b separates the upper from the lower part of the dotted line through a and b . It is also possible, that $\{a, b\}$ is both a type-1 and a type-2 pair, for example, $\{h, j\}$ in our example graph is a type-1 as well as a type-2 separation pair with respect to the cycle shown in Figure 3.10.

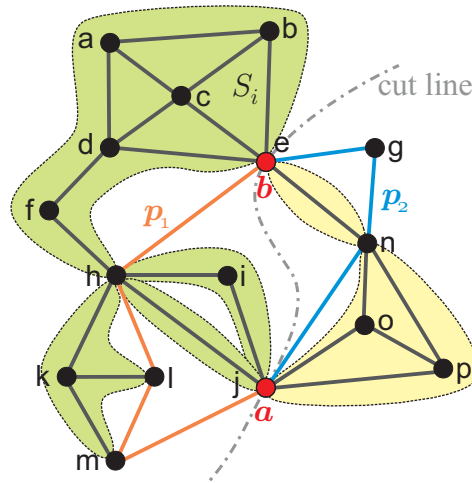


Figure 3.11: Type-2 separation pair $\{a, b\}$ relative to the cycle c . Removing a and b disconnects the graph at the shown cut line.

The algorithm will find a cycle c , test each segment for separation pairs by a recursive call, and test c itself for separation pairs by checking the criteria in Lemma 3.4. The finding of cycles is realized using a depth first search traversal on G satisfying certain conditions, which we discuss later. This traversal gives us a DFS-tree of G . Each cycle c consists of a sequence of T-arcs followed by one B-arc. The segments relative to c are either a single B-arc, or a T-arc $v \rightarrow w$ with $v \in c$ and $w \notin c$ plus the subtree T_w rooted at w plus all B-arcs leaving T_w .

Figure 3.12 illustrates the situation. It shows the graph G from Figure 3.5 and its DFS-tree. The cycle c we consider is formed by T-arcs leading from the root h to vertex e plus the B-arc $e \leftrightarrow h$. There are six segments S_1, \dots, S_6 relative to c . For example, S_2 consists only of a single B-arc $j \leftrightarrow h$, and S_6 consists of the T-arc $e \rightarrow c$ plus the subtree rooted at c plus all B-arcs leaving this subtree.

3.4.3 Computing SPQR-Trees

Let $G = (V, E)$ be a biconnected graph without self-loops and e_r an edge of G . The SPQR-tree \mathcal{T} of G with reference edge e_r is computed as follows:

1. Replace each bundle of multiple edges by a virtual edge thus creating the split graphs G_1, \dots, G_p . Call the resulting simple graph G' .
2. Compute the split components G_{p+1}, \dots, G_k of G' .
3. Starting with the split graphs G_1, \dots, G_k , successively merge two bonds or two polygons sharing an edge with the same label until no more such merge operation is possible. This results in the triconnected components of G .
4. Construct the undirected SPQR-tree \mathcal{T} (without Q-nodes) whose skeleton graphs are the triconnected components computed in the previous step.

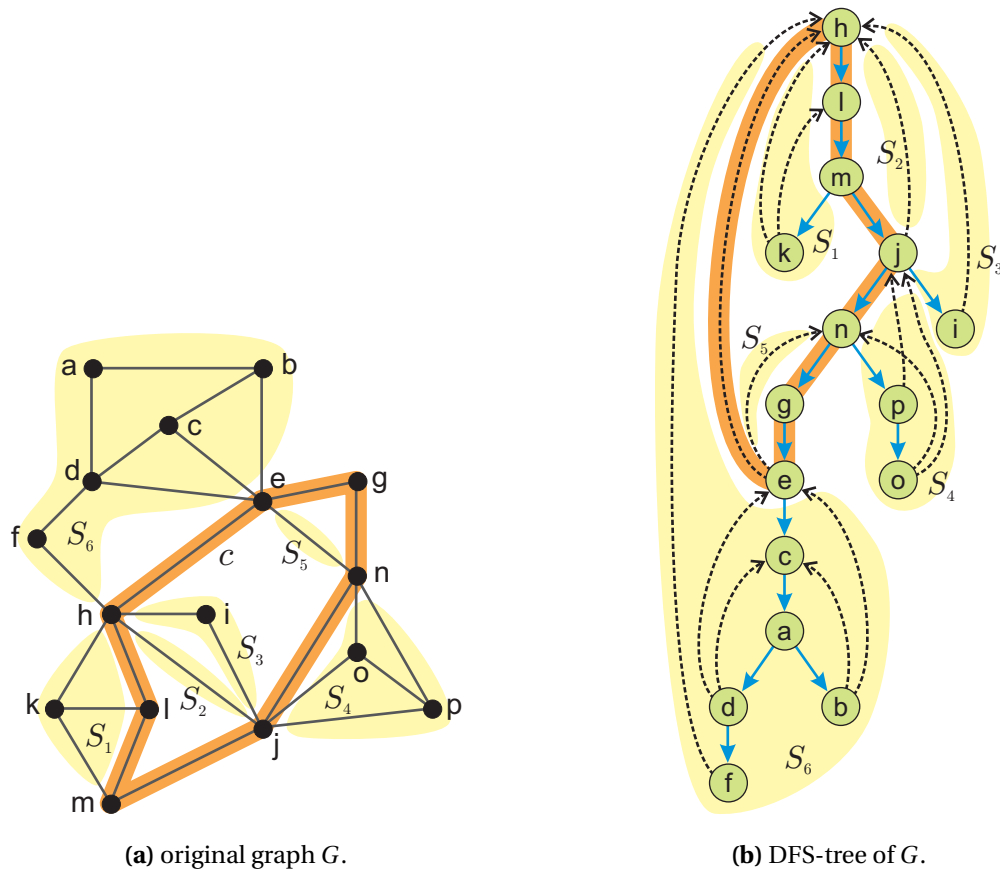


Figure 3.12: A cycle c in the graph G and the segments S_1, \dots, S_6 relative to c . (a) shows G as drawn in Figure 3.5, where the cycle c is highlighted; (b) shows the computed DFS-tree of G .

```

1: Sort the edges of  $G$  such that all multiple edges come after each other.
2: for each maximal bundle of multiple edges  $e_1, \dots, e_t$  with  $t \geq 2$  do
3:   let  $e_1, \dots, e_t$  be edges incident to  $v$  and  $w$ 
4:   Replace  $e_1, \dots, e_t$  by a new edge  $e' = (v, w, \ell)$ , where  $\ell$  is a new label.
5:   Create a new component  $C = \{e_1, \dots, e_t, e'\}$ .
6: end for

```

Listing 3.2: Split off multiple edges.

5. For each edge of G , add a Q-node to \mathcal{T} and root \mathcal{T} at the Q-node representing the reference edge e_r .

In the first step, bundles of multiple edges are replaced by a new virtual edge as shown in Listing 3.2. This creates a set of bonds C_1, \dots, C_p and results in a simple graph G' . The required sorting of the edges in line 1 can be done in $\mathcal{O}(|E|)$ time with two applications of bucket sort. We first sort the edges according to the endpoint with lower index and thereafter, we perform a stable sort according to the endpoint with higher index. Here, we assume that vertices have unique indices in the range $1, \dots, |V|$. The **for**-loop iterates over all edges, thus the algorithm in Listing 3.2 has running time $\mathcal{O}(|E|)$.

The second step finds the split components C_{p+1}, \dots, C_k of G' . The procedure is discussed in detail in the next subsections. The third step creates the triconnected components of G by partially reassembling the graphs C_1, \dots, C_k . As long as two bonds or two polygons C_i and C_j containing a virtual edge with the same label exist, C_i and C_j are merged. This is shown in Listing 3.3. The function $type(C)$ returns the type (bond, polygon, or triconnected simple graph) of a component C . Removed components are marked as empty. The **for all**-loop steps over *all* edges in C_i , that is, including those added to C_i during the loop. The test in line 4 can be done in constant time by precomputing for each virtual edge e the two components to which e belongs. We represent the edges in a component C_i by a list of edges, which allows to implement the set operations in line 5–6 in constant time. According to Lemma 3.2, the total number of edges in all components is $\mathcal{O}(|E|)$, so the algorithm in Listing 3.3 can also be implemented with a running time of $\mathcal{O}(|E|)$.

The preceding steps give enough information to build the SPQR-tree \mathcal{T} of G . Applying Theorem 3.5, it is easy to construct the unrooted version of \mathcal{T} . Since we omit Q-nodes in our representation, we root \mathcal{T} at the node whose skeleton contains the reference edge e_r . During the construction, we also create cross links between each tree edge $\mu \rightarrow v$ in \mathcal{T} and the two corresponding virtual edges in the skeleton of μ and the skeleton of v .

3.4.4 Finding Separation Pairs

Suppose we have a DFS-tree T for the simple, biconnected graph $G' = (V, E')$, and the vertices of G' are numbered $1, \dots, |V|$ with their DFS-numbers. For ease of notation, we identify the vertices with their numbers. We introduce the following

```

1: for  $i := 1$  to  $k$  do
2:   if  $C_i \neq \emptyset$  and  $C_i$  is a bond or a polygon then
3:     for all  $e = (u, v, \ell) \in C_i$  do
4:       if ex.  $j \neq i$  with  $e \in C_j$  and  $\text{type}(C_i) = \text{type}(C_j)$  then
5:          $C_i := (C_i \cup C_j) \setminus \{e\}$ 
6:          $C_j := \emptyset$ 
7:       end if
8:     end for
9:   end if
10: end for

```

Listing 3.3: Reassembling the triconnected components.

notation:

$$\text{lowpt1}(v) := \min(\{v\} \cup \{w \mid v \xrightarrow{*} \hookrightarrow w\})$$

$$\text{lowpt2}(v) := \min(\{v\} \cup (\{w \mid v \xrightarrow{*} \hookrightarrow w\} \setminus \{\text{lowpt1}(v)\}))$$

That is, $\text{lowpt1}(v)$ is the *lowest* numbered vertex reachable by traversing zero or more T-arcs followed by one B-arc of P (or v if no such vertex exists), and $\text{lowpt2}(v)$ is the *second lowest* numbered vertex reachable this way (or v if no such vertex exists). Hence, the lowpt1 -value is the lowpt -value introduced in Section 3.1.1. Similar to the lowpt -values, both the lowpt1 - and lowpt2 -values can be computed simultaneously with a single DFS-traversal as shown in Listing 3.4. The following table summarizes the important variables.

variable	purpose
s	The start vertex of the DFS-traversal; will be the root of the DFS-tree.
nextNum	The next DFS number.
$\text{number}[v]$	The DFS number of vertex v .
$\text{lowpt1}[v]$	The lowpt1 value of vertex v .
$\text{lowpt2}[v]$	The lowpt2 value of vertex v .
$\text{nd}[v]$	The number of vertices in the subtree rooted at v .

We denote with $\text{Adj}(v)$ the ordered (non-cyclic) adjacency list of a vertex v , and with $D(v) := \{w \mid v \xrightarrow{*} w\}$ the set of descendants of v in the DFS-tree. We seek for a numbering of the vertices and ordering of the edges in the adjacency lists satisfying the following three properties:

- (P1) The root of T is numbered with 1.
- (P2) If $v \in V$ and w_1, \dots, w_n are the children of v in T according to the ordering in $\text{Adj}(v)$, then $w_i = w + |D(w_{i+1}) \cup \dots \cup D(w_n)| + 1$.

Input: biconnected simple Graph $G' = (V, E')$, start vertex $s \in V$

```

1:  $nextNum := 0$ 
2: DFS1( $s, nil$ )

3: procedure DFS1( $vertex\ v, vertex\ parent$ )
4:    $number[v] := nextNum := nextNum + 1$ 
5:    $lowpt1[v] := lowpt2[v] := number[v]$ 
6:    $nd[v] := 1$ 
7:   for all edges  $e = (v, w) \in E'$  do
8:     if  $e$  is already marked then continue
9:     if  $number[w] = 0$  then
10:      Mark  $e$  as T-arc  $v \rightarrow w$ 
11:      DFS1( $w, v$ )
12:      if  $lowpt1[w] < lowpt1[v]$  then
13:         $lowpt2[v] := \min\{lowpt1[v], lowpt2[w]\}$ 
14:         $lowpt1[v] := lowpt1[w]$ 
15:      else if  $lowpt1[w] = lowpt1[v]$  then
16:         $lowpt2[v] := \min\{lowpt2[v], lowpt2[w]\}$ 
17:      else
18:         $lowpt2[v] := \min\{lowpt2[v], lowpt1[w]\}$ 
19:      end if
20:       $nd[v] := nd[v] + nd[w]$ 
21:    else
22:      Mark  $e$  as B-arc  $v \leftrightarrow w$ 
23:      if  $number[w] < lowpt1[v]$  then
24:         $lowpt2[v] := lowpt1[v]$ 
25:         $lowpt1[v] := number[w]$ 
26:      else if  $number[w] > lowpt1[v]$  then
27:         $lowpt2[v] := \min\{lowpt2[v], number[w]\}$ 
28:      end if
29:    end if
30:  end for
31: end procedure

```

Listing 3.4: Computing the $lowpt1$ - and $lowpt2$ -values.

(P3) The edges in $Adj(v)$ are in ascending order according to $lowpt1(w)$ if the corresponding edge is $v \rightarrow w$, or w if it is $v \hookrightarrow w$.

Let w_1, \dots, w_j be the children of v with the same $lowpt1$ -value u in the order given by $Adj(v)$. Then there exists an i' such that $lowpt2(w_k) < v$ for $i \leq k \leq i'$, and $lowpt2(w_k) \geq v$ for $i' < j \leq j$. If $v \hookrightarrow u \in E'$, then $v \hookrightarrow u$ comes in $Adj(v)$ between $v \rightarrow w_{i'}$ and $v \rightarrow w_{i'+1}$.

Notice that the $lowpt1$ - and $lowpt2$ -values refer to the same vertex for any numbering of the vertices that satisfies $v < w$ for each T-arc $v \rightarrow w$. Since the $lowpt1$ - and $lowpt2$ -values of the children w_1, \dots, w_k of v only refer to vertices on the tree path from the root to v plus the respective child, the required sorting in (P3) is independent of the numbering of the vertices, as long as the numbering satisfies the condition above. Therefore, we can compute the sorting of the adjacency lists with the original DFS-numbering. For each edge $e \in E'$, we define a value $\phi(e)$:

$$\phi(e) := \begin{cases} 3lowpt1(w) & \text{if } e = v \rightarrow w \text{ and } lowpt2(w) < v \\ 3w + 1 & \text{if } e = v \hookrightarrow w \\ 3lowpt1(w) + 2 & \text{if } e = v \rightarrow w \text{ and } lowpt2(w) \geq v \end{cases}$$

The required sorting can then be obtained by sorting the edges with bucket sort in ascending order according to their ϕ -values. It is then easy to compute the new numbering. We will compute it later in Listing 3.5 together with additional information.

Remark 3.1. Unlike [Hopcroft and Tarjan, 1973a], we demand that a B-arc $v \hookrightarrow w$, if contained in E' , must come between $v \rightarrow w_{i'}$ and $v \rightarrow w_{i'+1}$ in $Adj(v)$. Using only the function ϕ as defined in [Hopcroft and Tarjan, 1973a] and the procedure PATHSEARCH as suggested in [Hopcroft and Tarjan, 1973a] will not recognize all multiple edges and thus not correctly compute the split components of G' .

Figure 3.13 illustrates property (P3). Let w_1, \dots, w_n be the children of v in the DFS-tree T with $lowpt1(w_i) = u$ for each $1 \leq i \leq n$ in the order required by (P3). Suppose we remove vertices u and v . Then, each vertex in $D(w_i)$ with $1 \leq i \leq i_0$ (i_0 defined as in (P3)) is still connected with a vertex on the tree path from u to v , since $u < lowpt2(w_i) < v$. On the other hand, each vertex in $D(w_j)$ with $i_0 < j \leq n$ is no longer connected to the rest of the graph, since $lowpt2(w_j) \geq v$. In this case, $\{u, v\}$ is a separation pair (more exactly a type-1 separation pair) that splits off the vertices in $D(w_j)$.

Suppose we perform a depth-first-search on G' using the ordering of the edges in the adjacency lists. This divides G' into a set of paths consisting of zero or more T-arcs followed by one B-arc. The first path starts at vertex 1 and a path ends, when the first B-arc on the path is reached (see Figure 3.14). Each path ends at the lowest possible vertex and has only its initial and terminal vertex in common with previously traversed paths. From each such path $p : v \xrightarrow{*} w$, we can form a

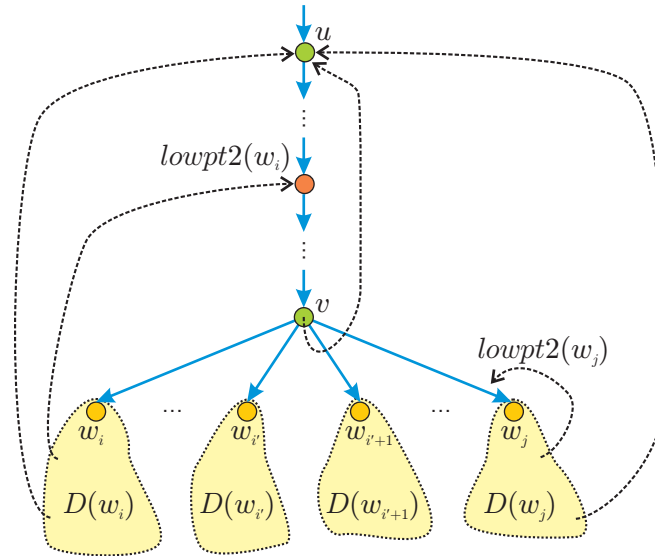


Figure 3.13: The order of the children w_i, \dots, w_j of v with the same $lowpt1$ -value u .

cycle by adding the tree path $w \xrightarrow{*} v$ to p (compare [Hopcroft and Tarjan, 1973a, 1974]). These are the cycles described in subsection 3.4.2.

Figure 3.14 shows a DFS-tree with a numbering that satisfies (P1)-(P3). The edges are labeled according to the generated paths. The adjacency lists are ordered as follows (blue numbers refer to T-arcs and red numbers to B-arcs):

1: 2	9: 10
2: 3	10: 1, 11, 6
3: 16, 4	11: 12
4: 1, 6, 5	12: 14, 13
5: 1	13: 10, 11
6: 9, 7	14: 15, 10, 11
7: 8, 4	15: 1
8: 4, 6	16: 1, 2

The generated paths are:

A: $1 \rightarrow 2 \rightarrow 3 \rightarrow 16 \hookrightarrow 1$	H: $12 \rightarrow 13 \hookrightarrow 10$
B: $16 \hookrightarrow 2$	I: $13 \hookrightarrow 11$
C: $3 \rightarrow 4 \hookrightarrow 1$	J: $10 \hookrightarrow 4$
D: $4 \rightarrow 6 \rightarrow 9 \rightarrow 10 \hookrightarrow 1$	K: $6 \rightarrow 7 \rightarrow 8 \hookrightarrow 4$
E: $10 \rightarrow 11 \rightarrow 12 \rightarrow 14 \rightarrow 15 \hookrightarrow 1$	L: $8 \hookrightarrow 6$
F: $14 \hookrightarrow 10$	M: $7 \hookrightarrow 4$
G: $14 \hookrightarrow 11$	N: $4 \rightarrow 5 \hookrightarrow 1$

For example, the cycle considered in Figure 3.12 is composed from the paths A, C, and D.

Procedure DFS2 in Listing 3.5 shows how to compute a numbering satisfying (P1)-(P3). It assumes that the adjacency lists are already in the correct order. It

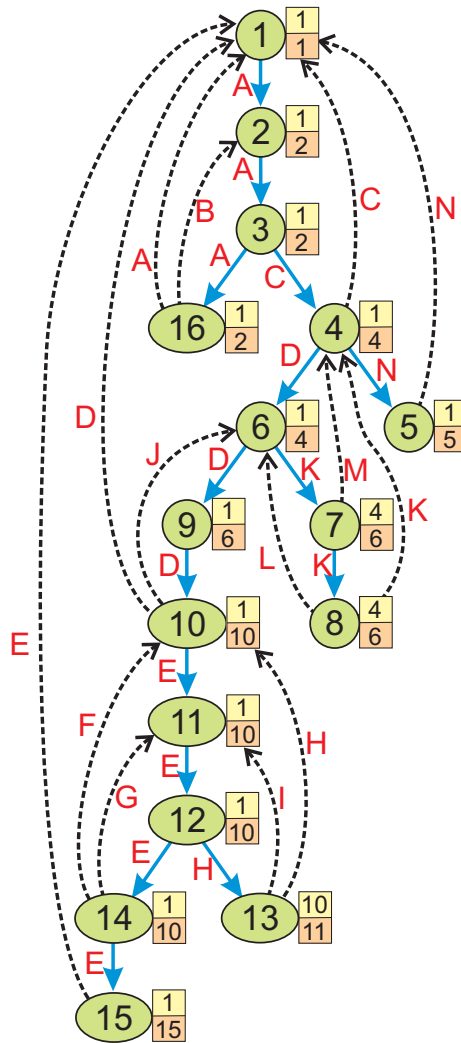


Figure 3.14: DFS-tree with numbered vertices and generated paths A, \dots, L . The vertices are also annotated with their *lowpt1*-values (top) and *lowpt2*-values (bottom).

also adjusts the values of the *lowpt1*- and *lowpt2*-values to the new numbering. The following table gives a description of the important variables.

variable	purpose
s	The root of the DFS-tree.
$A(v)$	The ordered adjacency list of vertex v .
$number[v]$	The DFS-number of vertex v .
$nd[v]$	The number of vertices in the subtree rooted at v .
$lowpt1[v]$	The <i>lowpt1</i> value of vertex v .
$lowpt2[v]$	The <i>lowpt2</i> value of vertex v .
$newnum[v]$	The new number of vertex v satisfying (P1)–(P3).
$startsPath[e]$	Set to true iff e starts a path.
$highpt[v]$	The list of B-arcs $v_i \leftrightarrow v$ ending at v in the order they are visited during the path search.
$old2new[i]$	Translates DFS-number to new number.

The following Lemma shows a general property of DFS-trees for biconnected graphs:

Lemma 3.5 (Hopcroft and Tarjan 1973a, Tarjan 1972). *Let G be a biconnected graph and T an arbitrary DFS-tree of G . Then, vertex 1 has only one child, and for each T -arc $v \rightarrow w$ in T holds*

$$lowpt1(w) \begin{cases} < v & \text{if } v \neq 1 \\ = 1 & \text{if } v = 1. \end{cases}$$

Proof. If vertex 1 would have more than one child, then 1 would obviously be a cut vertex, what contradicts the biconnectivity of G .

Let $v \neq 1$ and suppose that $lowpt1(w) \geq v$. If we remove v , then the vertices in $D(w)$ are no longer connected to the rest of the graph, which contains at least vertex 1, since $v \neq 1$. But that would mean, that v is a cut vertex, and we have again a contradiction.

Let now $v = 1$ and suppose that $lowpt1(w) > 1$. Then, removing w would separate the vertices in $D(w) \setminus \{w\}$ from v . There is a vertex $\neq w$ in $D(w)$, because G contains at least three vertices and w is the only child of vertex $v = 1$. Thus, w would be a cut vertex. \square

Definition 3.7. Let u_0 and u_n be two vertices. We say that u_n is a *first descendant* of u_0 if $u_0 \rightarrow \dots \rightarrow u_n$ and for each $0 \leq i < n$, the tree arc $u_i \rightarrow u_{i+1}$ is the first edge in $Adj(u_i)$.

In the sequel, we consider a DFS-tree T satisfying the conditions (P1)–(P3). The following Lemma derives some important properties of descendants in T , which we will use below in the proof of Lemma 3.7.

Input: biconnected, simple Graph $G' = (V, E')$, start vertex $s \in V$,
ordered adjacency lists $A(v)$ for each $v \in V$

```

1: procedure DFS2(vertex  $s$ )
2:    $numCount := |V|$ 
3:    $newPath := \mathbf{true}$ 
4:   for each  $e \in E'$  do  $startsPath[e] := \mathbf{false}$ 
5:   PATHFINDER( $s$ )
6:   for all  $v \in V$  do
7:      $old2new[number[v]] := newnum[v]$ 
8:   end for
9:   for all  $v \in V$  do
10:     $lowpt1[v] := old2new[lowpt1[v]]$ 
11:     $lowpt2[v] := old2new[lowpt2[v]]$ 
12:  end for
13: end procedure

14: procedure PATHFINDER(vertex  $v$ )
15:    $newNum[v] := numCount - nd[v] + 1$ 
16:   let  $A(v) = e_1, \dots, e_k$ 
17:   for  $i := 1$  to  $k$  do
18:     let  $e_i = (v, w)$ 
19:     if  $newPath$  then
20:        $newPath := \mathbf{false}; startsPath[e] := \mathbf{true}$ 
21:     end if
22:     if  $e$  is a T-arc then
23:       PATHFINDER( $w$ )
24:        $numCount := numCount - 1$ 
25:     else
26:        $highpt[w].PUSHBACK(newNum[v])$ 
27:        $newPath := \mathbf{true}$ 
28:     end if
29:   end for
30: end procedure

```

Listing 3.5: Computing the new numbering and generated paths.

Lemma 3.6 (Lemma 11 and 12 in [Hopcroft and Tarjan, 1973a]). *Let G be a biconnected graph and T a DFS-tree of G satisfying the properties (P1)-(P3).*

- (a) *Let v be a vertex in G . Then $D(v) = \{x \mid v \leq x < v + |D(v)|\}$. If w is a first descendant of v , then $D(v) \setminus D(w) = \{x \mid v \leq x < w\}$.*
- (b) *Let $\{a, b\}$ be a separation pair in G with $a < b$. Then $a \xrightarrow{*} b$ in T holds.*

Proof.

- (a) We use induction on the number of descendants of v . If $|D(v)| = 1$, then $D(v) = \{v\} = \{x \mid v \leq x < v + 1\}$.

Let now $|D(v)| > 1$ and let w_1, \dots, w_n be the children of v . We have

$$\begin{aligned}
 D(v) &= \{v\} \cup \bigcup_{i=1}^n D(w_i) \\
 &\stackrel{\text{ind. hyp.}}{=} \{v\} \cup \bigcup_{i=1}^n \{x \mid w_i \leq x < w_i + |D(w_i)|\} \\
 &\stackrel{\text{def. of } w_i}{=} \{v\} \cup \bigcup_{i=1}^n \{x \mid v + |D(w_{i+1}) \cup \dots \cup D(w_n)| + 1 \leq x < \\
 &\qquad\qquad\qquad v + |D(w_i) \cup \dots \cup D(w_n)| + 1\} \\
 &= \{v\} \cup \{x \mid v + 1 \leq x < v + |D(w_1) \cup \dots \cup D(w_n)| + 1\} \\
 &= \{x \mid v \leq x < v + |D(v)|\}
 \end{aligned}$$

what establishes the induction step.

Let w_1, \dots, w_n be the children of v in the order given by $Adj(v)$. For the descendants of the first child w_1 of v we can show

$$\begin{aligned}
 D(w_1) &= \{x \mid w_1 \leq x < w_1 + |D(w_1)|\} \\
 &\stackrel{\text{def. of } w_1}{=} \{x \mid w_1 \leq x < v + |D(w_2) \cup \dots \cup D(w_n)| + 1 + |D(w_1)|\} \\
 &= \{x \mid w_1 \leq x < v + |D(v)|\}
 \end{aligned}$$

Inductively follows for a first descendant w of v , that $D(w) = \{x \mid w \leq x < v + |D(v)|\}$. Therefore, we have

$$\begin{aligned}
 D(v) \setminus D(w) &= \{x \mid v \leq x < v + |D(v)|\} \setminus \{x \mid w \leq x < v + |D(v)|\} \\
 &= \{x \mid v \leq x < w\}
 \end{aligned}$$

- (b) [Hopcroft and Tarjan, 1973a] Suppose that b is not a descendant of a and let E_1, \dots, E_k be the separation classes with respect to $\{a, b\}$. Since $a < b$, a cannot be a descendant of b and thus $D(a)$ and $D(b)$ are disjoint. Let $S = V \setminus (D(a) \cup D(b))$. $E(S)$ must be contained in some separation class, say E_1 , since the vertices in S define a subtree containing neither a nor b . Consider any child c of a . $E(D(c))$ must be contained in some separation class. But since G is biconnected and $a \neq 1$ (otherwise b would be a descendant

of a), $\text{lowpt1}(c) < a$ by Lemma 3.5, and hence $E(D(c)) \subseteq E_1$. A similar argumentation shows that $E(D(c')) \subseteq E_1$ for any descendant c' of b . But this means that there is only one separation class $\neq \{(a, b)\}$ and $\{a, b\}$ is not a separation pair. \square

We can now reformulate Lemma 3.4 using a DFS-tree T satisfying (P1)-(P3), such that it gives us three easy-to-check conditions for separation pairs.

Lemma 3.7 (Lemma 13 in [Hopcroft and Tarjan, 1973a]). *Let $G = (V, E)$ be a bi-connected graph and a, b be two vertices in G with $a < b$. Then $\{a, b\}$ is a separation pair if and only if one of the following conditions holds.*

Type-1 Case: *There are distinct vertices $r \neq a, b$ and $s \neq a, b$ such that $b \rightarrow r$, $\text{lowpt1}(r) = a$, $\text{lowpt2}(r) \geq b$, and s is not a descendant of r ; compare Figure 3.15(a).*

Type-2 Case: *There is a vertex $r \neq b$ such that $a \rightarrow r \xrightarrow{*} b$, b is a first descendant of r , $a \neq 1$, every B-arc $x \hookrightarrow y$ with $r \leq x < b$ has $a \leq y$, and every B-arc $x \hookrightarrow y$ with $a < y < b$ and $b \rightarrow w \xrightarrow{*} x$ has $\text{lowpt1}(w) \geq a$; compare Figure 3.15(b).*

Multiple Edge Case: *(a, b) is a multiple edge of G and G contains at least four edges.*

Proof. (If) Suppose $\{a, b\}$ satisfies one of the three cases. If it satisfies the multiple edge case, $\{a, b\}$ is certainly a separation pair. Suppose $\{a, b\}$ satisfies the type-1 case (see also Figure 3.16(a)). Denote with $D(r)$ the descendants of r . Each B-arc starting at a vertex in $D(r)$ ends at a vertex in $D(r) \cup \{a, b\}$, because $\text{lowpt1}(r) = a$ and $\text{lowpt2}(r) \geq b$. Therefore, removing a and b separates the vertices in $D(r)$ from the rest of the graph, which contains at least vertex s .

Suppose now that $\{a, b\}$ satisfies the type-2 case (see also Figure 3.16(b)). Let $S = D(r) \setminus D(b)$. Since b is a first descendant of r , $S = \{x \mid r \leq x < b\}$ by Lemma 3.6(i). Let b_1, \dots, b_n be the children of b in the order given by $\text{Adj}(b)$. Since, by property (P3), the children in $\text{Adj}(b)$ are ordered according to ascending lowpt1 -values, there exists an i_0 , such that $\text{lowpt1}(b_i) < a$ for $i < i_0$, and $\text{lowpt1}(b_i) \geq a$ for $i \geq i_0$. Let $S^* = S \cup \bigcup_{i \geq i_0} D(b_i)$. Consider a B-arc $x \hookrightarrow y$ with $x \in S^*$. If $x \in S$, then $r \leq x < b$ and thus $a \leq y < b$ by assumption, that is, $y \in S \cup \{a, b\}$. Otherwise, $x \in D(b_i)$ for some $i \geq i_0$, and therefore $y \in D(b_i) \cup S \cup \{a, b\}$, since $\text{lowpt1}(b_i) \geq a$. This proves that a B-arc starting in S^* ends in $S^* \cup \{a, b\}$. Consider now a B-arc $x \hookrightarrow y$ with $y \in S^*$. If $y \in \bigcup_{i \geq i_0} D(b_i)$, then obviously $x \in \bigcup_{i \geq i_0} D(b_i)$. Otherwise, $a < y < b$ holds. If $x \notin S \cup \{b\}$, then $b \rightarrow w \xrightarrow{*} x$ and $w = b_i$ for some $i \geq i_0$ by assumption, that is, $x \in D(b_i)$. This proves that a B-arc ending in S^* must start in $S^* \cup \{b\}$. Therefore, removing a and b separates the component S^* , which contains at least vertex r , from the rest of the graph, which contains at least vertex 1.

(Only-If) Let $\{a, b\}$ be a separation pair with $a < b$. If $\{a, b\}$ is a multiple edge in G , then $\{a, b\}$ must satisfy the multiple edge case. Thus suppose that $\{a, b\}$ is

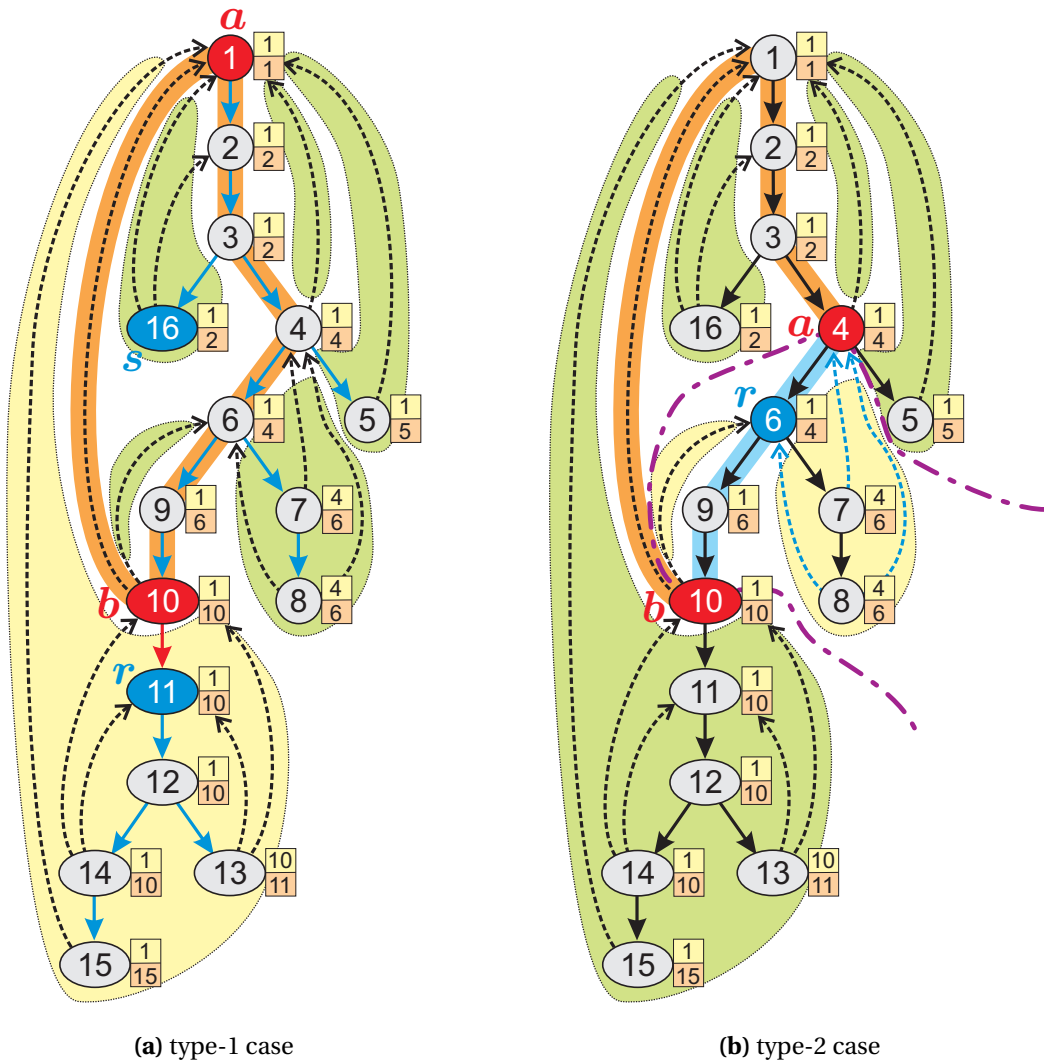


Figure 3.15: Examples for type-1 and type-2 separation pairs within the DFS-tree. (a) the type-1 pair from Figure 3.10; (b) the type-2 pair from Figure 3.11.

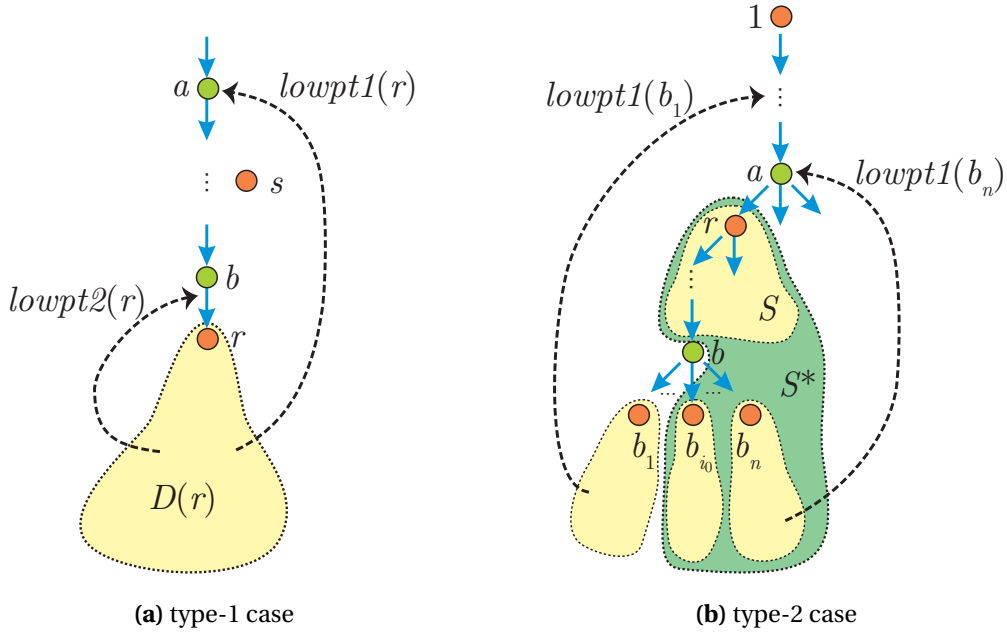


Figure 3.16: Proof of Lemma 3.7: (a) shows the type-1 case with subtree $D(r)$; (b) illustrates the type-2 case with the sets S and S^* .

not a multiple edge. By Lemma 3.6(ii), $a \xrightarrow{*} b$ holds (compare Figure 3.17(a)). Let E_1, \dots, E_k be the separation classes with respect to $\{a, b\}$, and let v be the child of a , such that $a \rightarrow v \xrightarrow{*} b$, $S = D(v) \setminus D(b)$, and $X = V \setminus D(a)$. We have $S = \emptyset$ if $v = b$, and $X = \emptyset$ if $a = 1$. $E(S)$ and $E(X)$ are each contained in a separation class, say $E(S) \subseteq E_1$ and $E(X) \subseteq E_2$.

If a has a child a_s different from v , then we have $a \neq 1$ and $\text{lowpt1}(a_s) < a$ by Lemma 3.5. Therefore $E(D(a_s)) \subseteq E_2$. If we denote with Y the set $X \cup \bigcup_{\substack{a \rightarrow a_s \\ a_s \neq v}} D(a_s)$, then $E(Y) \subseteq E_2$. Let b_1, \dots, b_n be the children of b in the order they occur in $\text{Adj}(b)$. We have that each separation class must be a union of the sets $E(S)$, $E(Y)$, $\{(a, b)\}$, $E(D(b_1)), \dots, E(D(b_n))$.

We consider now the case, that there exist some i and j , such that $E_j = E(D(b_i))$. We claim that, in this case, $\{a, b\}$ satisfies the type-1 case with $r = b_i$. We have $\text{lowpt1}(b_i) = a$ and $\text{lowpt2}(b_i) \geq b$, because $E(D(b_i))$ and $E(S) \cup E(Y)$ are disjoint and G is biconnected. Since $\{a, b\}$ is a separation pair, there must be a separation class other than E_j and $\{(a, b)\}$, what means that there is a vertex $s \notin \{a, b\} \cup D(b_i)$.

Otherwise, no $E(D(b_i))$ is a separation class on its own, and therefore each $E(D(b_i))$ must be contained either in E_1 or E_2 . The ordering of the children of b implies that there is an i_0 , such that $\text{lowpt1}(b_i) < a$ for $i < i_0$ and $\text{lowpt1}(b_i) \geq a$ for $i \geq i_0$. Thus, since G is biconnected, the separation classes with respect to $\{a, b\}$ are $E_1 = E(S) \cup \bigcup_{i \geq i_0} E(D(b_i))$, $E_2 = E(Y) \cup \bigcup_{i < i_0} E(D(b_i))$, and additionally $E_3 = \{(a, b)\}$ if $(a, b) \in E$ (compare Figure 3.17(b)). Since $\{a, b\}$ is a separation pair, neither E_1 nor E_2 are empty. We claim that $\{a, b\}$ satisfies the type-2 case

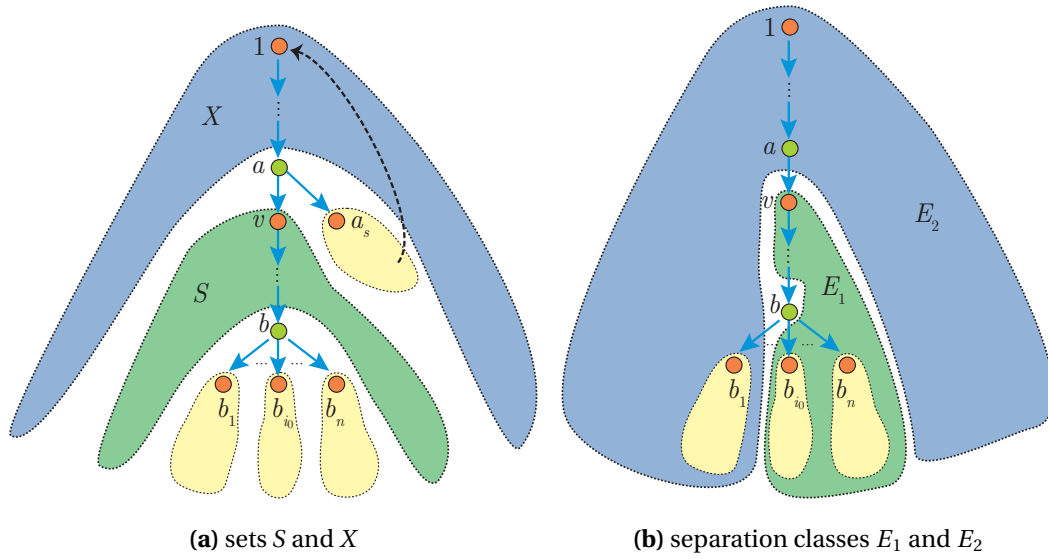


Figure 3.17: Proof of Lemma 3.7: Illustration of (a) the sets S and X and (b) the separation classes E_1 and E_2 .

with $r = v$. We have $v \neq b$ since otherwise, each $E(D(b_i))$ with $i \geq i_0$ would be a separation class on its own, or E_1 would be empty if $i_0 > n$. Moreover, $a \neq 1$, since otherwise Y and thus E_2 would be empty. Let $x \hookrightarrow y$ be a B-arc with $v \leq x < b$. Then $x \in S$, $(x, y) \in E_1$, and therefore $a \leq y$. Let now $x \hookrightarrow y$ be a B-arc with $a < y < b$ and $b \rightarrow b_i \rightarrow^* x$. Then, $y \in S$, $(x, y) \in E_1$, and therefore $i \geq i_0$, what implies that $\text{lowpt1}(b_i) \geq a$. Finally, we have to show that b is a first descendant of v . We have $\text{lowpt1}(v) < a$ by Lemma 3.5, since $a \neq 1$. This implies that there is a B-arc $x \hookrightarrow y \in E_2$ with $x \in D(v)$ and $y < a$, such that x is a first descendant of v , because the children in the adjacency lists are ordered according to ascending lowpt1 -values. Since $E_1 \neq E_2$, it follows that $x \in \{b\} \cup \bigcup_{i < i_0} D(b_i)$, and thus b is a first descendant of v . \square

Consider the DFS-tree from Figure 3.14. We have the following separation pairs:

type-1 pairs: (1, 4), (1, 5), (4, 5), (1, 8), (1, 3)
type-2 pairs: (4, 8), (8, 12)

3.4.5 Finding Split Components

During the algorithm, we maintain a graph G_c and a DFS-tree P_c of G_c . We denote with $\text{deg}(v)$ the degree of v in G_c , with $v \rightarrow w$ a T-arc in P_c , with $v \hookrightarrow w$ a B-arc in P_c , and with $\text{parent}(v)$ the parent of v in P_c . The (already computed) value $\text{nd}[v]$ is the number of descendants of v in P_c . Each time we identify a split component C we split it off, and the graph G_c along with its DFS-tree P_c is updated. We use the following update functions:

$C := \text{NEWCOMPONENT}(e_1, \dots, e_\ell)$: creates a new component $C = \{e_1, \dots, e_\ell\}$ and removes e_1, \dots, e_ℓ from G_c .

$C := C \cup \{e_1, \dots, e_\ell\}$: the edges e_1, \dots, e_ℓ are added to C and removed from G_c .

$e' := \text{NEWVIRTUALEGE}(v, w, C)$: creates a new virtual edge $e' = (v, w)$ and adds it to component C and G_c .

$\text{MAKETREEEDGE}(e, v \rightarrow w)$: makes edge $e = (v, w)$ a new tree edge in P_c .

Moreover, we define the access functions

$$\begin{aligned} \text{firstChild}(v) &= \text{first child of } v \text{ in } P_c \text{ according to } \text{Adj}(v). \\ \text{high}(w) &= \begin{cases} 0 & \text{if } \text{highpt}[w] \text{ is empty} \\ \text{highpt}[w].\text{FRONT}() & \text{otherwise} \end{cases} \end{aligned}$$

and we use two stacks for which the usual operations PUSH, POP, and TOP are defined:

EStack contains already visited edges that are not yet assigned to a split component.

TStack contains triples (h, a, b) (or a special end-of-stack marker *eos*), such that $\{a, b\}$ is a potential type-2 separation pair, and h is the highest numbered vertex in the component that would be split off.

The algorithm starts by calling the recursive procedure `PATHSEARCH` for vertex 1 which is the root vertex of P_c . When returning from the call, the edges belonging to the last split component are on *EStack*.

```

1: TStack.PUSH(eos)
2: PATHSEARCH(1)
3: let  $e_1, \dots, e_\ell$  be the edges on EStack
4:  $C := \text{NEWCOMPONENT}(e_1, \dots, e_\ell)$ 

```

Listing 3.6: Finding the split components.

The procedure `PATHSEARCH` is shown in Listing 3.7 and 3.8. The testing for separation pairs applies Lemma 3.7 and is depicted separately in Listing 3.9 for type-2 and in Listing 3.10 for type-1 separation pairs⁶. In order to achieve linear running time, we set up the following data structures:

- The DFS-tree P_c is represented by the arrays $\text{parent}[v]$, $\text{treeArc}[v]$ (the T-arc entering v), and $\text{type}[e]$ (which is T-arc or B-arc).
- The values of $\text{lowpt1}[v]$, $\text{lowpt2}[v]$, $\text{nd}[v]$, and $\text{startsPath}[e]$ are precomputed (see Listing 3.4 und 3.5). It is not necessary to update them.

⁶The algorithm will not find *all* separation pairs, but only the separation pairs needed for dividing the graph into its split components

```

1: procedure PATHSEARCH(vertex v)
2:   outv := |Adj(v)|
3:   for all  $e \in \text{Adj}(v)$  do
4:     if  $e = v \rightarrow w$  then ▷ e is a T-arc
5:       if startsPath[e] then
6:         Pop all (h, a, b) with  $a > \text{lowpt1}[w]$  from TStack.
7:         if no triples deleted then
8:           TStack.PUSH( $w + nd[w] - 1, \text{lowpt1}[w], v$ )
9:         else
10:           $y := \max\{h \mid (h, a, b) \text{ deleted from } TStack\}$ 
11:          let (h, a, b) be last triple deleted.
12:          TStack.PUSH(y, lowpt1[w], b)
13:        end if
14:        TStack.PUSH(eos)
15:      end if
16:      PATHSEARCH(w)
17:      EStack.PUSH( $v \rightarrow w$ )
18:      Check for type-2 pairs.
19:      Check for a type-1 pair.
20:      if startsPath[e] then
21:        Remove all triples on TStack down to and including eos.
22:      end if
23:      while (h, a, b) on TStack has  $b \neq v$  and  $\text{high}(v) > h$  do
24:        TStack.POP()
25:      end while
26:    else
▷ continued on next page...

```

Listing 3.7: Procedure PATHSEARCH (part 1).


```

27:         let  $e = v \hookrightarrow w$  ▷  $e$  is a B-arc
28:         if  $startsPath[e]$  then
29:             Pop all  $(h, a, b)$  with  $a > w$  from  $TStack$ .
30:             if no triples deleted then
31:                  $TStack.PUSH(v, w, v)$ 
32:             else
33:                  $y := \max\{h \mid (h, a, b) \text{ deleted from } TStack\}$ 
34:                 let  $(h, a, b)$  be last triple deleted.
35:                  $TStack.PUSH(y, w, b)$ 
36:             end if
37:         end if
38:          $EStack.PUSH(e)$ 
39:     end if
40:      $outv := outv - 1$ 
41: end for
42: end procedure

```

Listing 3.8: Procedure PATHSEARCH (part 2).

- The array $degree[v]$ contains the degree of a vertex $v \in G_c$. It is updated each time an edge is added to or removed from G_c .
- In order to compute $firstChild(v)$, we update the adjacency lists each time an edge is added to or removed from G_c .
- In order to compute $high(v)$, we precompute the list of the source nodes of the B-arcs $v_i \hookrightarrow w$ ending at w in the order they are visited. When a B-arc is removed from or added to G_c , the respective list is updated.

3.4.6 Corrections on the Hopcroft and Tarjan Algorithm

Procedure SPLIT in [Hopcroft and Tarjan, 1973a] does not correctly split a graph into its split components. We summarize the important changes we have made in our algorithm:

- The sorting function ϕ had to be modified as described in subsection 3.4.4 in order to identify all multiple edges.
- The creation of the last split component (see line 4 in Listing 3.6) was missing.
- The condition in line 23 in Listing 3.7 was changed. The original condition could remove triples from $TStack$ corresponding to real type-2 separation pairs. Such a separation pair could not be recognized by the original SPLIT procedure.

```

1: while  $v \neq 1$  and ((( $h, a, b$ ) on  $TStack$  has  $a = v$ ) or ( $\text{deg}(w) = 2$  and
    $\text{firstChild}(w) > w$ )) do
2:   if  $a = v$  and  $\text{parent}(b) = a$  then
3:      $TStack.POP()$ 
4:   else
5:      $e_{ab} := nil$ 
6:     if  $\text{deg}(w) = 2$  and  $\text{firstChild}(w) > w$  then
7:        $C := \text{NEWCOMPONENT}()$ 
8:       Remove top edges  $(v, w)$  and  $(w, x)$  from  $EStack$  and add to  $C$ .
9:        $e' := \text{NEWVIRTUALEDGE}(v, x, C)$ 
10:      if  $EStack.TOP() = (x, v)$  then
11:         $e_{ab} := EStack.POP()$ 
12:         $\text{DELHIGH}(e_{ab})$ 
13:      end if
14:    else
15:       $(h, a, b) := TStack.POP()$ 
16:       $C := \text{NEWCOMPONENT}()$ 
17:      while  $(x, y)$  on  $EStack$  has  $a \leq x \leq h$  and  $a \leq y \leq h$  do
18:        if  $(x, y) = (a, b)$  then
19:           $e_{ab} := EStack.POP()$ 
20:           $\text{DELHIGH}(e_{ab})$ 
21:        else
22:           $e_{xy} := EStack.POP()$ 
23:           $\text{DELHIGH}(e_{xy})$ 
24:           $C := C \cup \{e_{xy}\}$ 
25:        end if
26:      end while
27:       $e' := \text{NEWVIRTUALEDGE}(a, b, C)$ 
28:    end if
29:    if  $e_{ab} \neq nil$  then
30:       $C := \text{NEWCOMPONENT}(e_{ab}, e')$ 
31:       $e' := \text{NEWVIRTUALEDGE}(v, b, C)$ 
32:    end if
33:     $EStack.PUSH(e')$ 
34:     $\text{MAKETREEEDGE}(e', v \rightarrow b)$ 
35:     $w := b$ 
36:  end if
37: end while

```

Listing 3.9: Check for type-2 pairs.

```

1: if  $lowpt2[w] \geq v$  and  $lowpt1[w] < v$  and ( $parent(v) \neq 1$  or  $outv \geq 2$ ) then
2:    $C := \text{NEWCOMPONENT}()$ 
3:   while  $(x, y)$  on  $EStack$  has  $w \leq x < w + nd[w]$  or  $w \leq y < w + nd[w]$  do
4:      $e_{xy} := EStack.POP()$ 
5:      $\text{DELHIGH}(e_{xy})$ 
6:      $C := C \cup \{e_{xy}\}$ 
7:   end while
8:    $e' := \text{NEWVIRTUALEDGE}(v, lowpt1[w], C)$ 
9:   if  $EStack.TOP() = (v, lowpt1[w])$  then
10:     $C := \text{NEWCOMPONENT}(EStack.POP(), e')$ 
11:     $e' := \text{NEWVIRTUALEDGE}(v, lowpt1[w], C)$ 
12:   end if
13:   if  $lowpt1[w] \neq parent(v)$  then
14:     $EStack.PUSH(e')$ 
15:   else
16:     $C := \text{NEWCOMPONENT}(e', lowpt1[w] \rightarrow v)$ 
17:     $e' := \text{NEWVIRTUALEDGE}(lowpt1[w], v, C)$ 
18:     $\text{MAKETREEEDGE}(e', lowpt1[w] \rightarrow v)$ 
19:   end if
20: end if

```

Listing 3.10: Check for a type-1 pair.

- The condition in line 1 in Listing 3.10 was changed. The original condition could incorrectly identify separation pairs after the graph had been modified.
- The updates for $firstChild(v)$ (which is $A1(v)$ in [Hopcroft and Tarjan, 1973a]) and $DEGREE(v)$ were not sufficient.
- $high(w)$ (which is $HIGHPT(w)$ in [Hopcroft and Tarjan, 1973a]) was not updated, which is not correct. It is necessary to update $HIGHPT$ dynamically, when G_c is modified. We replaced $HIGHPT(w)$ by a list of B-arcs ending at w , which is updated as G_c changes.

3.4.7 Computational Experiments

The implementation of the triconnectivity decomposition algorithm and the data structures for the representation of SPQR-trees build upon the Open Graph Drawing Framework (OGDF) [Chimani et al., 2010] and are thereby publicly available. Graphs are represented by the OGDF data type *Graph*. There are two main classes for the representation of a static SPQR-tree:

- The class `StaticSPQRTree` implements SPQR-trees for general (not necessarily planar) biconnected graphs. The skeleton of a tree node is represented by an instance of the class `Skeleton`, which allows to access the

skeleton graph as a usual OGDF graph. There is also a method for computing the pertinent graph of a tree node, which is then represented by the class `PertinentGraph`.

- A specialized class `StaticPlanarSPQRTree` is derived from `StaticSPQRTree` and supports modifying and updating the combinatorial embedding of a *planar* graph. The graph is embedded immediately after construction, and the following basic update operations allow to modify the embedding:
 - Mirror the embedding of the skeleton of an R-node.
 - Permute the parallel edges in the skeleton of a P-node.

Using these methods, it is also possible to compute a random embedding of the graph.

Furthermore, OGDF implements dynamic SPQR-trees [Di Battista and Tamassia, 1996a,b]. These are represented by the classes `DynamicSPQRTree` and `DynamicPlanarSPQRTree`; their implementation has been contributed by Jan Papenfuß. In our tests, we always give the runtime for the creation of a `StaticSPQRTree`, even for planar graphs.

Test Suite. We tested our implementation with generated planar and non-planar biconnected graphs, as well as the benchmark graphs from the Rome library, graphs collected by Stefan Hachul, and road networks:

- **Randomly generated graphs:** We generated planar and non-planar biconnected graphs with up to 100,000 edges; each graph had a density (number of nodes divided by number of edges) of 2.
 - A planar biconnected graph with n vertices and m edges is generated by starting with a triangle and performing $n - 3$ randomly chosen split-edge and $m - n$ split face operations. We used OGDF's graph generator `planarBiconnectedGraph()`.
 - A general biconnected graph with n vertices and m edges is generated by starting with a triangle and performing $n - 3$ randomly chosen split-edge and $m - n$ add edge operations. We used OGDF's graph generator `randomBiconnectedGraph()`.
- **Rome graphs:** The Rome graphs are a collection of 11,528 planar and non-planar graphs⁷ ranging from 10 to 100 vertices. The graphs are quite sparse, with an average density of 1.35. Figure 3.18 shows the distribution of the graphs in the library, as well as the average densities for each group of graphs with the same number of vertices. The Rome graphs have been introduced in an experimental study by Di Battista et al. [1997] for comparing graph drawing algorithms with respect to several aesthetic criteria. Since

⁷originally, it were 11,582 graphs, but some of the files are corrupted

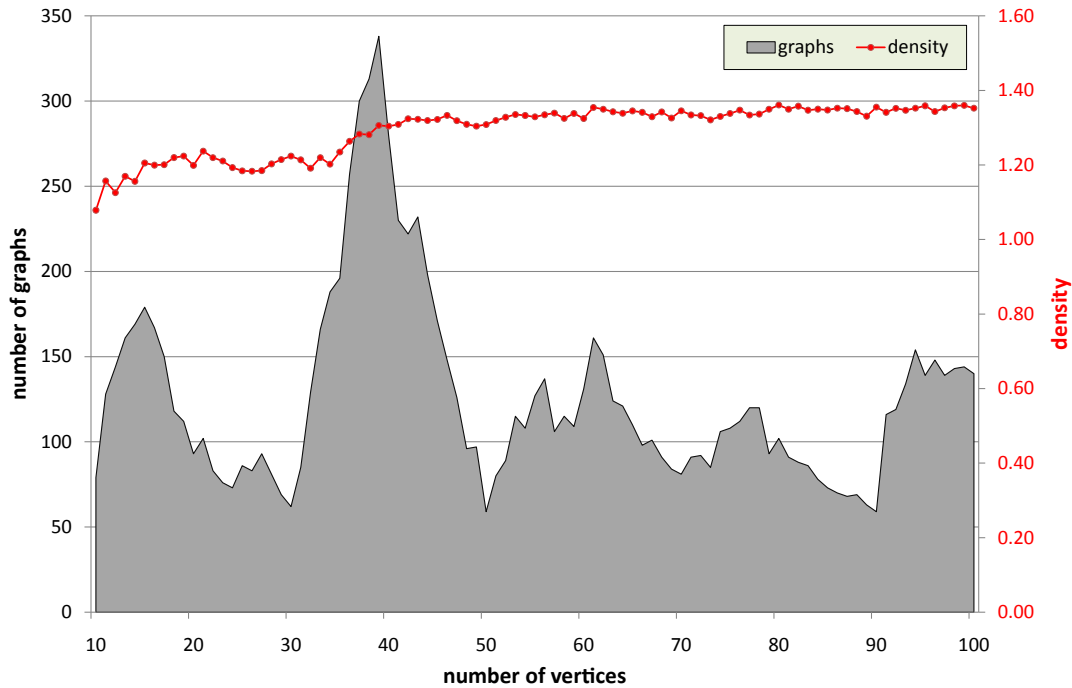


Figure 3.18: Distribution of graphs and average densities for the Rome graphs.

these graphs are not necessarily biconnected, we constructed the SPQR-tree for the largest block in each graph.

- Hachul library:** Stefan Hachul collected several graphs that have been used for evaluating the performance of force-directed graph drawing algorithms [see Hachul and Jünger, 2007]. Therefore, this benchmark set contains also very large graphs with up to 3.3 million edges. The graphs originate from various application fields and include some artificially constructed graphs as well. For the graphs in this library that are not biconnected we used again their largest blocks in our experiments. We selected 24 triconnected and 36 not-triconnected graphs, each having a largest block of at least 1,000 edges.
- DIMACS library:** The graphs in this library represent road networks of the 50 U.S. states and Washington D.C. They have been made publicly available during the *9th DIMACS Implementation Challenge* on shortest paths [Demetrescu et al., 2009]. All these graphs are planar and unconnected, therefore we also used the largest block of each graph in our experiments.

System Configuration. We performed the tests on a Windows PC with the following specifications:

<i>Operating system</i>	Windows 7 Professional (64-bit)
<i>Compiler</i>	Microsoft Visual C++ 9.0
<i>Chipset</i>	Intel P55
<i>CPU</i>	Intel Core i7 860 processor 2.8 GHz, 8 MB L3-Cache, 4*256 KB L2-Cache
<i>Memory</i>	8 GB RAM, 1333 MHz DDR3

Though this system provides 4 processor cores (8 hardware-threads through *Hyper Threading*), all our test programs utilize only a single processor core. We used Windows' high-resolution performance counter (Windows API function `QueryPerformanceCounter()`) to measure time. This counter has a resolution of 2,742,919 counts per second on our test system, which corresponds to a tick interval of 365 nanoseconds.

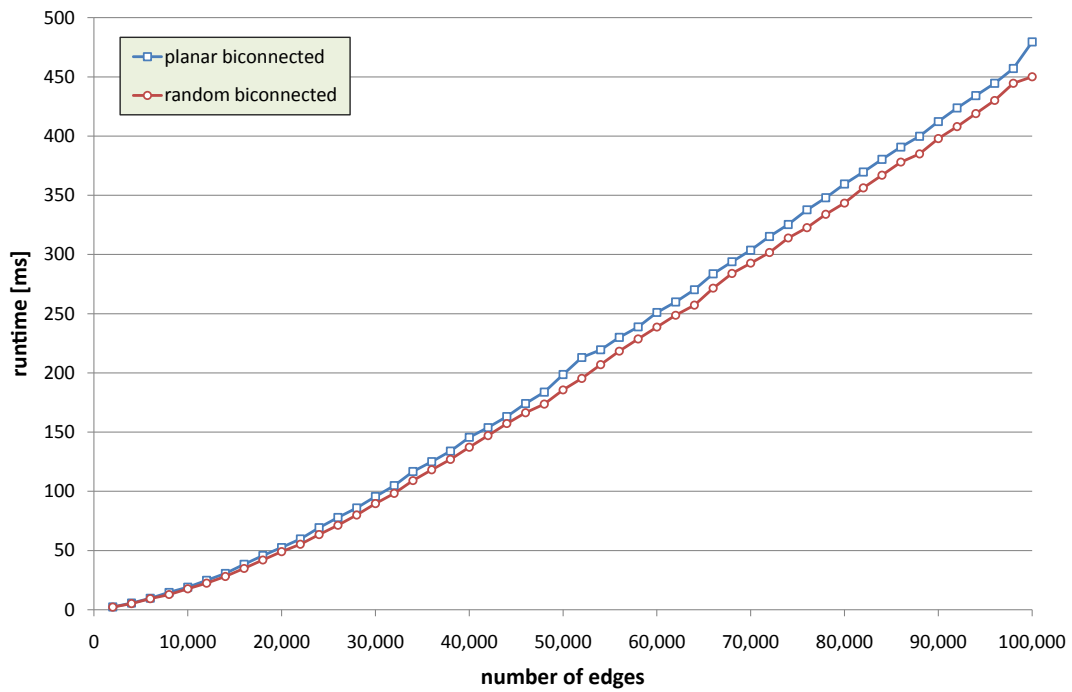
Test Results. The computed SPQR-trees are automatically checked with several consistency and plausibility tests when running in debug mode. The runtimes reported here are obtained by running the test programs in release mode, that is, extra checks are omitted and the code is optimized by the compiler.

The average running times for the randomly generated graphs are shown in Figure 3.19(a). The x-axis shows the number of edges and the y-axis the running time in milliseconds. Each data point corresponds to the average running time over 10 graphs. We observe that the running times for planar and non-planar graphs are very similar. Even instances with 100,000 edges can be solved in less than half a second.

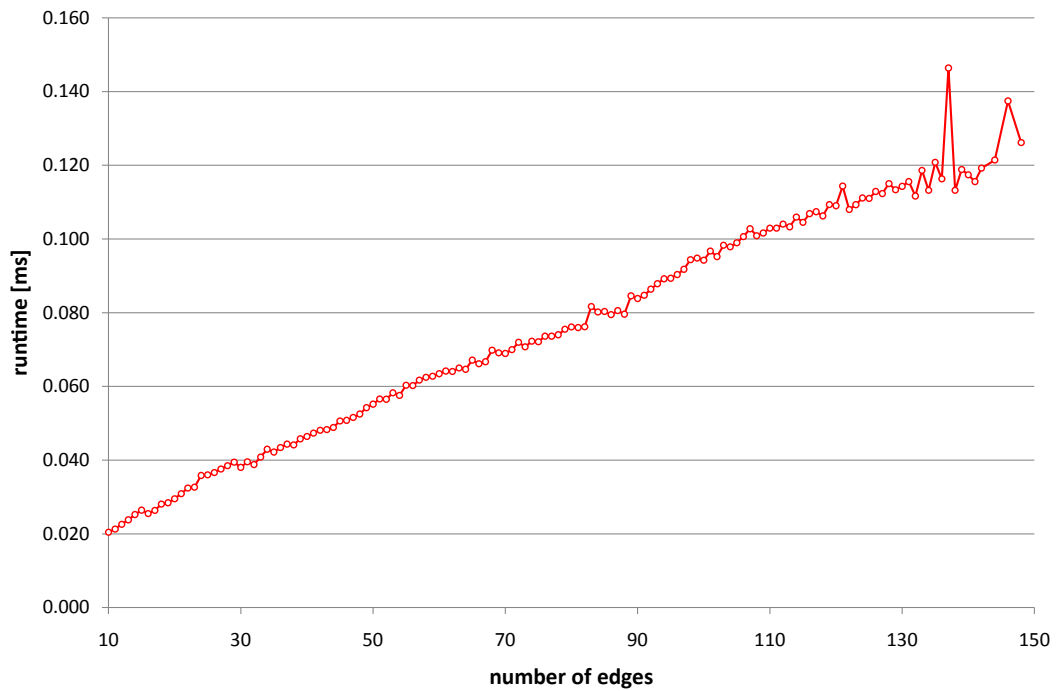
As a historical side note, we remark that Hopcroft and Tarjan [1973a] reported that their implementation was able to solve graphs with 1000 edges in less than 10 seconds on an IBM 360/65 mainframe in the early 70s. For comparison, our implementation requires an average runtime of 0.57 milliseconds on a standard PC for computing the triconnected components of graphs with 500 vertices and 1000 edges, which is a speed-up of about 17,500 over Hopcroft and Tarjan's results. Clearly, this is mainly a result of much faster computer systems. On the other hand, it is even unlikely that the Hopcroft/Tarjan implementation was correct.

The results for the Rome graphs are depicted in Figure 3.19(b). The x-axis shows the number of edges and the y-axis the running time in milliseconds. Each data point is the average running time over all blocks with the same number of edges. For the largest blocks, the average running time was about 120–140 microseconds. The longest running time was 245 microseconds required by a block with 82 vertices and 121 edges.

For the graphs from the Hachul and DIMACS library, we present the results in Figure 3.20 and Tables 3.1 through 3.4. Since these graphs have quite different structures, we show the data point for each instance in the diagrams and display further statistical information (number of vertices and edges; number of differ-

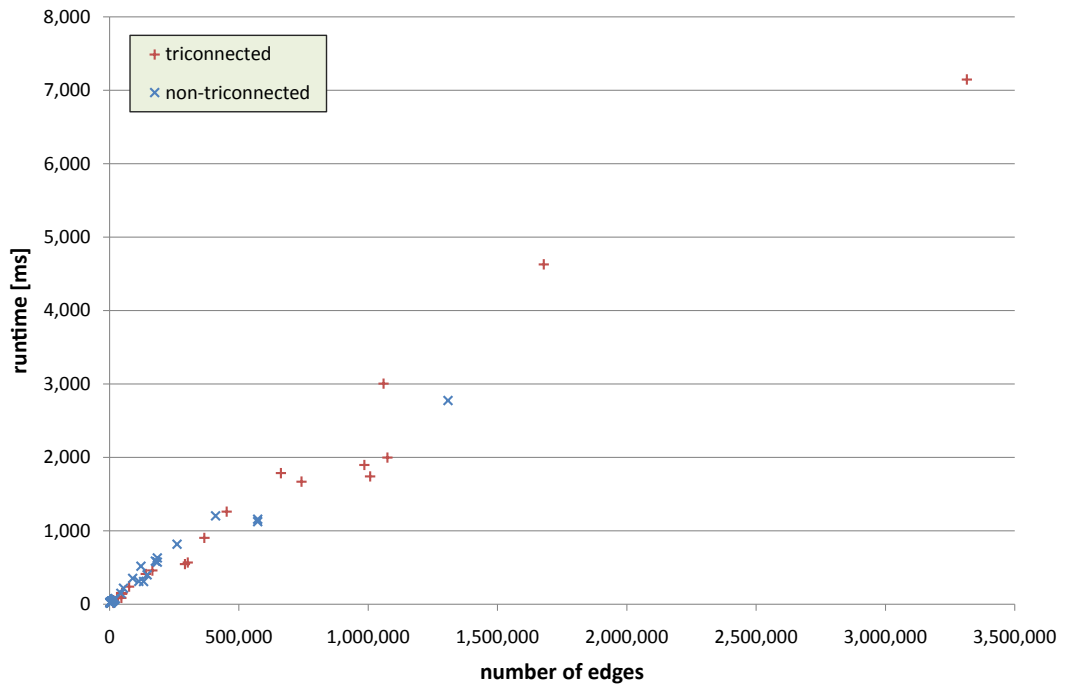


(a) generated graphs

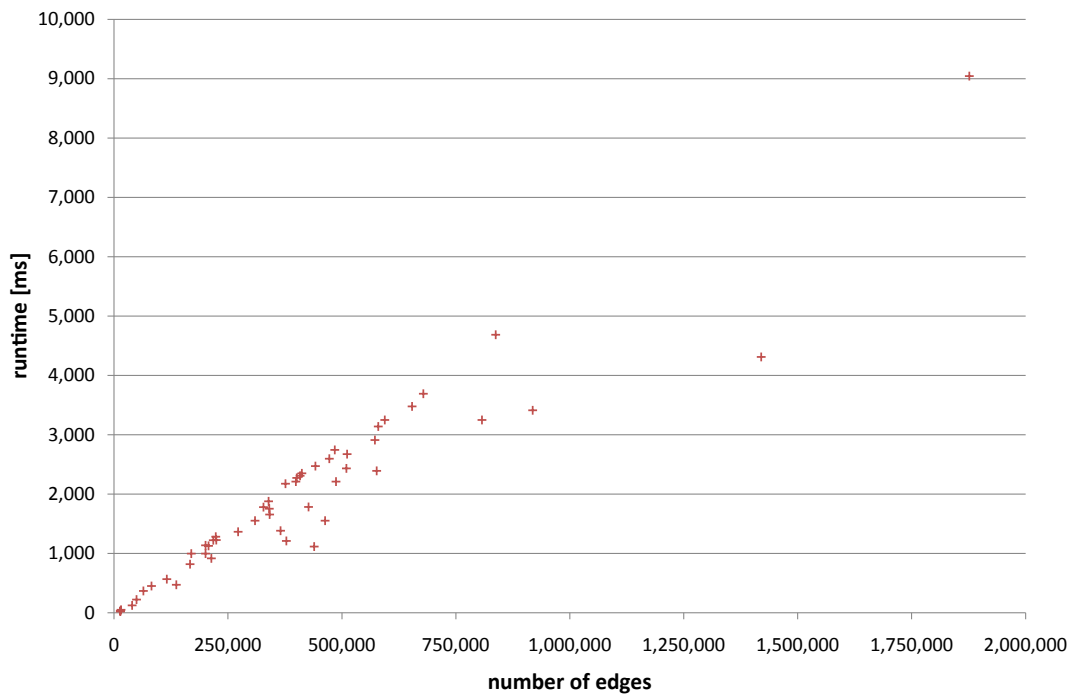


(b) Rome graphs

Figure 3.19: Average running times for the construction of SPQR-trees.



(a) Hachul library



(b) DIMACS library.

Figure 3.20: Running times for the construction of SPQR-trees.

graph	$ V_B $	$ E_B $	time [ms]
144	144,649	1,074,393	1,998
3elt	4,720	13,722	23
4elt	15,606	45,878	86
598a	110,971	741,934	1,670
auto	448,695	3,314,611	7,148
bcsstk29	13,830	302,424	568
bcsstk30	28,924	1,007,284	1,741
bcsstk32	44,607	985,044	1,897
bcsstk33	8,738	291,583	547
brack2	62,631	366,559	903
crack	10,240	30,380	84
cti	16,840	48,232	143
data	2,851	15,093	47
fe_4elt2	11,143	32,818	96
fe_rotor	99,617	662,431	1,786
fe_sphere	16,386	49,152	151
fe_tooth	78,136	452,591	1,261
m14b	214,765	1,679,018	4,630
spider_C	2,000	14,000	52
spider_D	20,000	140,000	412
vibrobox	12,328	165,250	459
wave	156,317	1,059,331	3,004
whitaker3	9,800	28,989	98
wing_nodal	10,937	75,488	240

Table 3.1: Runtimes for SPQR-tree construction: largest block $B = (V_B, E_B)$ of each graph from the Hachul-library (*triconnected blocks*).

ent types of nodes in the constructed SPQR-tree) in the corresponding tables. Our implementation runs stable even for the largest graphs in these benchmark sets, which are auto in the Hachul library (a triconnected graph with 448,695 vertices and 3,314,611 edges; 7.148 seconds runtime) and the state of Texas TX (1,394,230 vertices and 1,876,613 edges; 9.042 seconds; 370,812 nodes in the constructed SPQR-tree).

graph	$ V_B $	$ E_B $	time	n_S	n_P	n_R	total
add20	2,018	7,076	21	1210	1143	1	2354
add32	985	2,144	15	754	724	52	1530
all	1,174	4,432	17	399	645	2	1046
bcsstk31	35,539	572,866	1,126	5	12	17	34
bcsstk31_connected	35,539	572,866	1,158	5	12	17	34
cs4	22,499	43,858	149	1	0	1	2
cylinder_rnd_032_032	984	1,865	23	14	0	1	15
cylinder_rnd_100_100	9,493	17,937	63	130	0	1	131
cylinder_rnd_320_320	97,307	184,769	574	1376	5	1	1382
dg_274	935	4,958	32	51	1386	2	1439
dg_3578	2,112	6,223	34	528	96	6	630
dg_3691	572	2,025	24	143	30	8	181
dg_4765	3,040	10,427	44	736	282	16	1034
dg_4891	6,902	22,021	77	1164	2479	1	3644
dg_4941	659	2,126	25	165	41	1	207
fe_body	30,555	113,398	313	99	125	28	252
fe_ocean	143,398	409,554	1,205	451	13	1	465
fe_pwt	36,409	144,740	401	64	0	1	65
finan512	74,752	261,120	819	4608	3072	1	7681
flower_001	210	3,057	25	6	12	7	25
flower_005	930	13,521	44	6	36	31	73
flower_050	9,030	131,241	310	6	306	301	613
flower_500	90,030	1,308,441	2,775	6	3006	3001	6013
grid_rnd_032	984	1,833	24	19	0	1	20
grid_rnd_100	9,492	17,844	65	154	1	1	156
grid_rnd_320	97,302	184,475	631	1450	8	1	1459
memplus	17,575	53,973	217	9799	6619	22	16440
rand_uncon_D_comp_100	959	1,277	24	298	0	1	299
sierpinski_06	1,095	2,187	25	4	3	15	22
sierpinski_08	9,843	19,683	72	4	3	21	28
sierpinski_10	88,575	177,147	587	4	3	27	34
spider_B	200	1,400	25	0	600	1	601
t60k	60,005	89,440	352	936	0	1	937
ug_380	1,078	3,205	27	50	0	1	51
uk	4,508	6,448	40	396	41	7	444
wing	62,032	121,544	517	82	4	1	87

Table 3.2: Runtimes for SPQR-tree construction: largest block $B = (V_B, E_B)$ of each graph from the Hachul-library (*non-triconnected blocks*). The table shows the runtime in milliseconds, the number of S-, P-, and R-nodes (n_S , n_P , n_R , respectively), and the total number of nodes $n_S + n_P + n_R$.

graph	$ V_B $	$ E_B $	time	n_S	n_P	n_R	total
AK	11,214	13,848	21	421	130	2,499	3,050
AL	351,260	439,262	1,115	11,972	1,743	84,051	97,766
AR	303,952	378,088	1,212	10,265	1,390	70,439	82,094
AZ	351,079	463,104	1,551	11,644	2,549	89,122	103,315
CA	1,063,817	1,419,949	4,311	32,451	6,063	259,421	297,935
CO	279,447	365,471	1,383	11,722	1,709	66,865	80,296
CT	104,747	136,851	470	3,553	576	26,946	31,075
DC	8,544	13,818	31	92	13	1,088	1,193
DE	30,149	39,898	126	1,014	245	7,256	8,515
FL	667,932	918,646	3,412	25,237	5,784	146,969	177,990
GA	455,616	576,167	2,390	15,721	2,048	113,405	131,174
HI	11,664	15,527	50	497	145	2,750	3,392
IA	317,299	426,927	1,783	11,784	617	89,262	101,663
ID	169,035	213,677	917	5,648	917	42,512	49,077
IL	596,024	807,116	3,249	20,562	2,094	147,012	169,668
IN	361,278	487,161	2,210	13,891	1,355	94,200	109,446
KS	380,078	509,945	2,434	15,527	840	107,873	124,240
KY	256,284	309,454	1,552	8,198	1,170	57,260	66,628
LA	260,707	341,334	1,654	8,024	1,339	59,798	69,161
MA	202,517	272,283	1,364	6,416	1,255	49,468	57,139
MD	126,760	166,844	820	4,610	865	29,930	35,405
ME	98,156	115,705	567	2,975	507	20,211	23,693
MI	420,949	572,354	2,911	17,633	2,132	104,651	124,416
MN	391,073	511,203	2,672	14,919	1,168	102,809	118,896
MO	455,411	579,700	3,139	14,880	1,571	108,404	124,855
MS	273,144	339,252	1,877	9,691	1,101	68,028	78,820
MT	182,245	224,335	1,224	6,031	985	41,868	48,884
NC	483,487	593,978	3,249	15,603	2,402	113,622	131,627
ND	157,338	207,521	1,128	6,245	310	44,462	51,017
NE	245,622	327,843	1,781	9,794	473	66,840	77,107
NH	67,870	82,132	450	2,284	405	15,661	18,350
NJ	241,233	340,757	1,755	7,670	1,526	53,192	62,388
NM	304,021	398,842	2,212	10,989	1,854	74,124	86,967
NV	170,833	217,723	1,221	5,374	1,202	43,161	49,737

Table 3.3: Runtimes for SPQR-tree construction: largest block $B = (V_B, E_B)$ of each graph from the DIMACS-library. The table shows the runtime in milliseconds, the number of S-, P-, and R-nodes (n_S , n_P , n_R , respectively), and the total number of nodes $n_S + n_P + n_R$.

graph	$ V_B $	$ E_B $	time	n_S	n_P	n_R	total
NY	510,863	678,781	3,691	14,197	2,309	113,983	130,489
OH	493,985	653,793	3,478	15,109	1,729	123,673	140,511
OK	367,860	484,188	2,744	13,520	1,299	92,295	107,114
OR	325,223	412,113	2,350	10,194	1,771	77,875	89,840
PA	631,782	837,329	4,685	17,043	2,844	150,402	170,289
RI	35,325	49,524	224	966	230	7,669	8,865
SC	294,444	376,170	2,174	10,970	1,701	75,944	88,615
SD	154,111	200,658	1,137	6,589	402	41,511	48,502
TN	356,708	441,995	2,470	11,752	1,672	86,253	99,677
TX	1,394,230	1,876,613	9,042	42,956	8,166	319,690	370,812
UT	156,652	200,714	997	4,721	961	40,471	46,153
VA	325,666	400,954	2,273	12,585	1,882	69,454	83,921
VT	55,639	64,405	369	1,549	242	12,018	13,809
WA	317,336	408,247	2,309	9,630	1,692	75,702	87,024
WI	361,410	472,431	2,595	12,703	1,218	91,057	104,978
WV	144,023	169,350	996	4,486	842	27,549	32,877
WY	173,108	223,310	1,282	5,376	954	45,771	52,101

Table 3.4: Runtimes for SPQR-tree construction: largest block $B = (V_B, E_B)$ of each graph from the DIMACS-library (*continued*). The table shows the runtime in milliseconds, the number of S-, P-, and R-nodes (n_S , n_P , n_R , respectively), and the total number of nodes $n_S + n_P + n_R$.

Chapter 4

Crossing Minimization

The wireless telegraph is not difficult to understand. The ordinary telegraph is like a very long cat. You pull the tail in New York, and it meows in Los Angeles. The wireless is the same, only without the cat.

ALBERT EINSTEIN (1879 – 1955)

Minimizing the number of crossings in a drawing of a graph is among the most challenging problems in graph theory and graph drawing. Although, there is a vast amount of literature on the problem (see Vrt'o [2009] for a comprehensive overview), so far practically efficient exact algorithms for crossing minimization are only known for graphs with relatively small crossing numbers. The currently best such algorithms, which are based on branch-and-cut with column generation, have been presented by Buchheim et al. [2008] and Chimani et al. [2008]; see also [Chimani, 2008] for a detailed description.

The *crossing number problem* is the problem of finding the crossing number for a given graph G (see Section 2.3 for basic definitions). Since we are not only interested in the required number of crossings, but also in a suitable representation of these crossings in a drawing of G , we want to construct a planarized representation G_p of G , which contains a vertex of degree four for each crossing (see also Section 2.4). We refer to the corresponding optimization problem as the *crossing minimization problem*:

CROSSING MINIMIZATION PROBLEM (CMP)	
Instance:	a graph G
Solution:	a planarized representation G_p of G
Minimize:	the number of crossings in G_p

The crossing number problem is even older than the area of automatic graph drawing itself. It goes back to Paul Turán, who proposed the problem in his “Notes of Welcome” in the first issue of the *Journal of Graph Theory* [Turán, 1977]. While working in a labor camp during the Second World War, he noticed that

crossings of the rails between kilns and storage yards caused the trucks to jump the rails, thus making the work even harder. Minimizing these crossings corresponds to the crossing minimization problem for a complete bipartite graph.

It is well known that the general crossing minimization problem is NP-hard [Garey and Johnson, 1983]. More precisely, it was shown that the following problem is NP-complete:

“Given a graph G and a non-negative integer K , decide whether there is a drawing of G with at most K edge crossings.”

However, for a fixed K , we can obtain a polynomial time algorithm by examining all possible configurations with up to K crossings. Clearly, this algorithm is not appropriate in practical applications for larger values of K . Recently, Grohe [2004] could show that this problem can be solved in time $\mathcal{O}(|V|^2)$. Even though the exponent is independent of K , the constant factor of his algorithm grows doubly exponentially in K . Therefore, this method is also of no relevance in practice.

The search for approximation algorithms did not lead to significant results either. While there is no known polynomial time approximation algorithm with any type of quality guarantee for the general problem, Bhatt and Leighton could derive an algorithm for graphs with *bounded degree* that approximates the number of crossings *plus the number of nodes* in polynomial time [Bhatt and Leighton, 1984]. Due to the complexity of the crossing minimization problem, many restricted versions have been considered in the literature. However, in most cases, for example, for bipartite, linear, and circular drawings, the problem remains NP-hard [Eades and Wormald, 1994, Masuda et al., 1990, 1987].

4.1 The One-Edge Insertion Problem

Currently, the best known approach for solving CMP heuristically is the planarization method as described in Section 2.4. One point of criticism on the planarization method was that when choosing a “bad” embedding in the edge re-insertion phase, the number of crossings may get much higher than necessary [Holton and Sheehan, 1993]. Hence, the question arose if there is a polynomial time algorithm for inserting an edge into the planar subgraph P so that the number of crossings is minimized. In this case, the task is to optimize over the set of all possible combinatorial embeddings of P .

While it is possible to compute an arbitrary combinatorial embedding for a planar graph in linear time [Mehlhorn and Mutzel, 1996, Chiba et al., 1985], it is often hard to optimize over the set of all possible combinatorial embeddings. For example, the problem of bend minimization can be solved in polynomial time for a fixed combinatorial embedding [Tamassia, 1987], while it is NP-hard over the set of all combinatorial embeddings [Garg and Tamassia, 2002]. When a linear function of polynomial size is defined on the cycles of a graph, it is NP-hard to

find the embedding that maximizes the value of the cycles that are face cycles in the embedding [Mutzel and Weiskircher, 2000, 1999].

Figure 4.1 shows a simple case in which the choice of the combinatorial embedding of the planar subgraph has an impact on the number of crossings produced when inserting the dashed edge. If we choose the embedding in Figure 4.1(a) for the planar subgraph (without the dashed edge), we get two crossings, whereas the minimal number of crossings over the set of all combinatorial embeddings is one; see Figure 4.1(b).

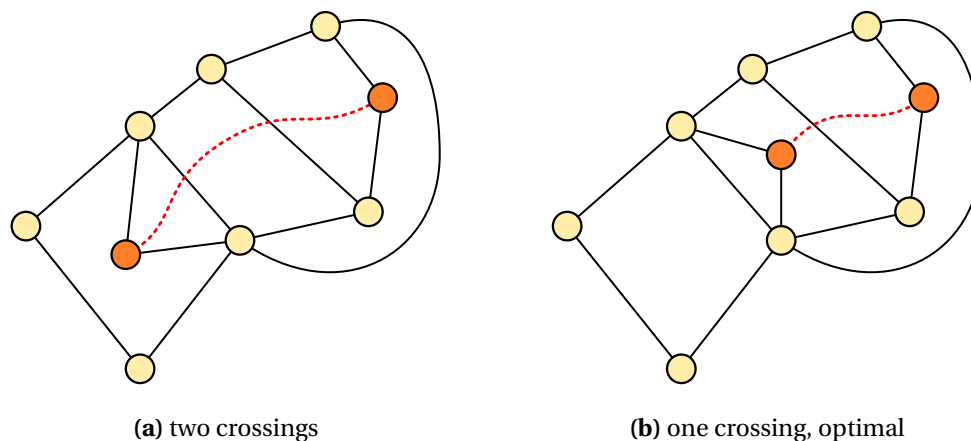


Figure 4.1: The number of crossings required when inserting an edge depends on the chosen embedding.

Formally, we define the *one-edge insertion problem* as follows: Given a planar graph $G = (V, E)$ and a pair of vertices (v_1, v_2) in G , find an embedding Π of G such that we can insert the edge $e = (v_1, v_2)$ into Π with the minimum possible number of crossings among all embeddings of G .

ONE-EDGE INSERTION PROBLEM (OEIP)	
Instance:	a planar graph $G = (V, E)$ and two vertices $v_1, v_2 \in V$ with $v_1 \neq v_2$
Solution:	an embedding Π of G
Minimize:	the number of crossings required to insert edge (v_1, v_2) into Π

This section shows that the OEIP can be solved in polynomial time, thus solving a long standing open problem in graph drawing. We present a conceptually simple linear time algorithm based on SPQR-trees which is able to solve the OEIP to optimality. Note that an optimal solution of the OEIP does not necessarily lead to a drawing of the graph $G' = (V, E \cup \{e\})$ with the minimum number of crossings. This is due to the fact that there may not always be a drawing of G' with a minimum number of crossings that induces a crossing-free drawing of $G = (V, E)$.

The rest of this section is organized as follows. After introducing the concept of traversing costs, we first present an algorithm for solving the OEIP for biconnected graphs. Then, we generalize this algorithm to arbitrary graphs. Finally, we address the difference and the connection between finding a drawing with the minimum number of crossings and the OEIP discussed in this section, in particular, the approximation of the crossing numbers of near-planar graphs with bounded degree.

4.1.1 Traversing Costs

Traversing costs of skeleton edges are a fundamental concept used in the one-edge insertion algorithm. First, we give a formal definition of the term *edge insertion path*. Let $G = (V, E)$ be a graph with embedding Π . Recall that we denote with Π^* the dual graph of G with respect to Π . In the following, we also use the notations e^* for the dual edge of an edge $e \in E$ and f^* for the dual vertex of a face $f \in \Pi$. An edge insertion path is basically associated with a path in the dual graph and determines the edges that are crossed when inserting an edge into a given embedding.

Definition 4.1 (Edge insertion path). Let $G = (V, E)$ be a connected planar graph and Π an embedding of G . Let v_1 and v_2 be two non-adjacent vertices in G . Then e_1, \dots, e_k is an *edge insertion path* for v_1 and v_2 in G with respect to Π if either $k = 0$ and v_1 and v_2 are contained in a common face in Π or the following conditions are all satisfied:

- (a) $e_1, \dots, e_k \in E$.
- (b) There is a face in Π with e_1 and v_1 on its boundary.
- (c) There is a face in Π with e_k and v_2 on its boundary.
- (d) e_1^*, \dots, e_k^* is a path in Π^* .

If $p = e_1, \dots, e_k$ is an edge insertion path for v_1 and v_2 with respect to Π , then it is possible to insert the edge (v_1, v_2) into Π with k crossings, where the i -th crossing involves edge (v_1, v_2) and edge e_i for $1 \leq i \leq k$. The length of p , denoted by $|p|$, is k . We call p an *optimal edge insertion path* for v_1 and v_2 in G , if there is no shorter edge insertion path for v_1 and v_2 in G with respect to any embedding of G .

Figure 4.2 shows three different edge insertion paths for v_1 and v_2 with respect to the embedding realized by the drawing. The three paths are the empty path, the path e_1, e_2, e_3 , and the path e_4, e_5, e_6 . In this case, the empty path is obviously the optimal edge insertion path for v_1 and v_2 .

Consider now a biconnected graph G and its SPQR-tree \mathcal{T} . The *traversing costs* $c(e)$ of a skeleton edge e in \mathcal{T} are defined as follows. Consider an arbitrary embedding Π of the graph $\text{expansion}^+(e)$ and its dual graph Π^* . Let f_1 and f_2 be the two faces in Π that are separated by e , and let f_1^* and f_2^* be the corresponding

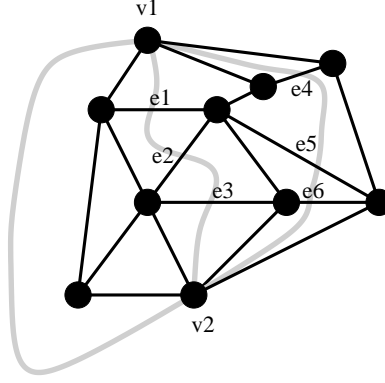


Figure 4.2: Three different edge insertion paths for v_1 and v_2

vertices in the dual graph. We denote with $P(\Pi^*, e)$ the shortest path in Π^* that connects f_1^* and f_2^* and does not use edge e^* . Lemma 4.1 below shows that the length of this path is independent of the embedding Π chosen for $\text{expansion}^+(e)$. Thus we define the *traversing costs* $c(e)$ simply as

$$c(e) = \text{length of the path } P(\Pi^*, e) \text{ for any embedding } \Pi \text{ of } \text{expansion}^+(e).$$

We also call the corresponding list of primal edges a *traversing path* for e .

Lemma 4.1. *Let μ be a node in \mathcal{T} and e be an edge in $\text{skeleton}(\mu)$. Then, the length of the path $P(\Pi^*, e)$ is independent of the embedding Π of $\text{expansion}^+(e)$.*

Proof. The expansion graph of e is defined using a subtree of the SPQR-tree \mathcal{T} . Let v be the pertinent node of e . We denote with \mathcal{T}_e the subtree of \mathcal{T} which is the connected component containing v of the graph $\mathcal{T} - (\mu, v)$. The root of \mathcal{T}_e is the node v , which is not necessarily a Q-node. We prove the lemma by induction over the height of \mathcal{T}_e .

If the height of \mathcal{T}_e is 1, then v is a Q-node and $\text{expansion}^+(e)$ is a circle of two edges. Thus, $\text{expansion}^+(e)$ has only a single embedding Π and the length of the path $P(\Pi^*, e)$ is simply 1.

Assume now that the height of \mathcal{T}_e is $k > 1$ and that the lemma holds for all skeleton edges \tilde{e} for which the height of $\mathcal{T}_{\tilde{e}}$ is less than k . Since $k > 1$, the root v of \mathcal{T}_e is either an S-, P-, or R-node. We denote with e' the virtual edge of μ in $\text{skeleton}(v)$. An embedding Π of $\text{expansion}^+(e)$ induces an embedding Π_v of $\text{skeleton}(v)$ and an embedding Π_h of $\text{expansion}^+(h)$ for each edge $h \neq e'$ in $\text{skeleton}(v)$. Since the height of \mathcal{T}_h is less than k , $c_h = P(\Pi_h^*, h)$ is independent of Π_h . We consider the three possible types of node v :

S-node: The skeleton of v is a circle e', e_1, \dots, e_ℓ with $\ell \geq 2$. The length of the path $P(\Pi^*, e)$ is $\min_{i=1}^{\ell} P(\Pi_{e_i}^*, e_i) = \min_{i=1}^{\ell} c_{e_i}$ which is independent of Π .

P-node: The skeleton of v consists of $\ell + 1$ parallel edges e', e_1, \dots, e_ℓ with $\ell \geq 2$ and the length of the path $P(\Pi^*, e)$ is $\sum_{i=1}^{\ell} P(\Pi_{e_i}^*, e_i) = \sum_{i=1}^{\ell} c_{e_i}$ which is also independent of Π .

R-node: The skeleton of v is a triconnected planar graph $S = (V_S, E_S)$. The length of the path $P(\Pi^*, e)$ is the length of a shortest path in Π_v^* connecting the two faces separated by e' without using the dual edge of e' , where each edge $h \in E_S \setminus \{e'\}$ has cost c_h . Since a triconnected planar graph has only two embeddings, which are mirror-images of each other, the length of this path is independent of the embedding Π_v and thus independent of Π .

□

According to Lemma 4.1, the traversing costs of a skeleton edge e can be computed by finding a shortest path in the dual graph of an arbitrary embedding of $\text{expansion}^+(e)$. This can be done in time $\mathcal{O}(|\text{expansion}^+(e)|)$ using a breadth first search approach.

4.1.2 Biconnected Graphs

In this section, we present an algorithm for optimally inserting an edge into a biconnected planar graph. We use the notion of the extended dual graph as defined in Section 2.4.2. We further say a skeleton edge e represents a vertex v of G if v is contained in $\text{expansion}(e)$ and v is not an endpoint of e , and we introduce the following notation for list concatenation. If $L_1 = a_1, \dots, a_k$ and $L_2 = b_1, \dots, b_\ell$ are two lists, we denote with $L_1 + L_2$ the list $a_1, \dots, a_k, b_1, \dots, b_\ell$. The algorithm OPTIMALBLOCKINSERTER for computing an optimal edge insertion path for a biconnected planar graph G and two non-adjacent vertices v_1 and v_2 of G is shown in Listing 4.1. We remark that it is not necessary to actually construct the graph G_i (defined in line 21) if μ_i is not an R-node. It is included in the presentation of the algorithm, since we refer to G_i in the correctness proofs (proofs of Lemma 4.2 and 4.3).

In order to prove the correctness of Algorithm OPTIMALBLOCKINSERTER, we first show that the path computed by the algorithm is indeed an edge insertion path with respect to some embedding.

Lemma 4.2. *Let $p_1 + \dots + p_k$ be the path computed by OPTIMALBLOCKINSERTER. Then, there exists an embedding Π of G such that $p_1 + \dots + p_k$ is an edge insertion path for v_1 and v_2 in G with respect to Π .*

Proof. Consider the path $\Lambda = \mu_1, \dots, \mu_k$ computed by the algorithm. By construction of Λ , the skeleton of μ_1 contains v_1 , the skeleton of μ_k contains v_2 (note that $k = 1$ is possible), and for each $j = 2, \dots, k - 1$, the skeleton of μ_j contains neither v_1 nor v_2 . Moreover, Λ does not contain a Q-node.

First, we prove the lemma for the case that Λ consists of a single node μ_1 . In this case, the skeleton of μ_1 contains both v_1 and v_2 . There are three possible cases for the type of μ_1 :

- (a) μ_1 **is an S-node:** Then v_1 and v_2 form a separation pair of G , see Figure 4.3(a). Let Π_1 be any embedding of G . Since $\{v_1, v_2\}$ is a separation pair, v_1 and v_2 lie in a common face of Π_1 . Thus, the empty path returned by the algorithm is an edge insertion path for v_1 and v_2 in G with respect to Π_1 .

```

1: function OPTIMALBLOCKINSERTER(graph  $G$ , vertex  $v_1$ , vertex  $v_2$ )
2:   Compute the SPQR-tree  $\mathcal{T}$  of  $G$ .
3:   Find the shortest path  $\Lambda = \mu_1, \dots, \mu_k$  in  $\mathcal{T}$  between an allocation node  $\mu_1$ 
4:   of  $v_1$  and  $\mu_k$  of  $v_2$ .
5:   for  $i = 1, \dots, k$  do
6:      $S_i := \text{skeleton}(\mu_i)$ 
7:     if  $v_1$  is in  $S_i$  then
8:        $x_i^1 := v_1$ 
9:     else
10:      Split the edge representing  $v_1$  in  $S_i$  by inserting a new vertex  $y_i^1$ .
11:      Mark the two edges produced by the split.
12:       $x_i^1 := y_i^1$ 
13:    end if
14:    if  $v_2$  is in  $S_i$  then
15:       $x_i^2 := v_2$ 
16:    else
17:      Split the edge representing  $v_2$  in  $S_i$  by inserting a new vertex  $y_i^2$ .
18:      Mark the two edges produced by the split.
19:       $x_i^2 := y_i^2$ 
20:    end if
21:    let  $G_i$  be the graph obtained from  $S_i$  by replacing each unmarked edge
22:    with its expansion graph.
23:    if  $\mu_i$  is not an R-node then
24:      set  $p_i$  to the empty path.
25:    else
26:      Compute an arbitrary embedding  $\Pi_i$  of  $G_i$ .
27:      let  $A_i$  be the extended dual graph of  $\Pi_i$  with respect to  $(x_i^1, x_i^2)$ .
28:      Compute the shortest path  $e_0^*, \dots, e_{\ell+1}^*$  in  $A_i$  between  $x_i^{1*}$  and  $x_i^{2*}$ .
29:       $p_i := e_1, \dots, e_\ell$ , where  $e_j$  is the primal edge of  $e_j^*$ .
30:    end if
31:  end for
32:  return  $p_1 + \dots + p_k$ 
33: end function

```

Listing 4.1: Computes an optimal edge insertion path for a pair of non-adjacent vertices v_1, v_2 in a biconnected planar graph G .

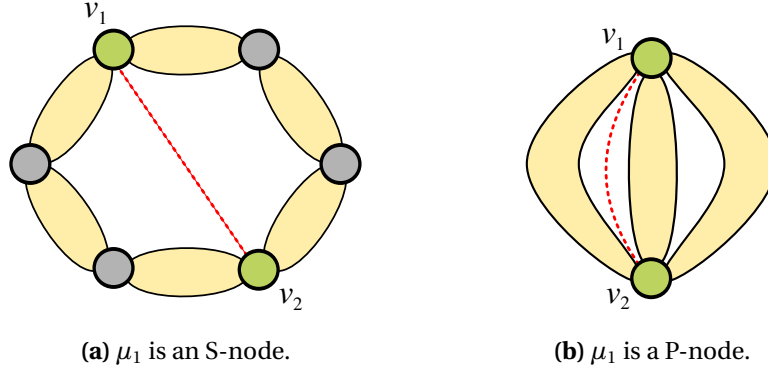


Figure 4.3: Path Λ consists of a single node μ_1 .

- (b) μ_1 **is a P-node:** Again, $\{v_1, v_2\}$ is a separation pair of G and similar arguing as for the first case holds, see Figure 4.3b.
- (c) μ_1 **is an R-node:** In this case, the graph G_1 constructed by the algorithm is the original graph G , since all skeleton edges are expanded, and Π_1 computed by the algorithm is an embedding of G . Thus, the algorithm computes an edge insertion path in G for v_1 and v_2 with respect to embedding Π_1 of G .

Assume now that $k > 1$. We define graphs H_1, \dots, H_k as follows. H_i is obtained from $skeleton(\mu_i)$ by replacing all skeleton edges not representing vertex v_2 by their expansion graphs, and, if $i < k$, splitting the skeleton edge that represents vertex v_2 and thereby introducing a new vertex r_i . The skeleton of μ_k contains vertex v_2 itself and we denote with r_k this vertex in $skeleton(\mu_k)$. We show by induction over i that there is an embedding Γ_i of H_i such that $p_1 + \dots + p_i$ is an edge insertion path for v_1 and r_i in H_i with respect to Γ_i . The embeddings $\Gamma_1, \dots, \Gamma_k$ are iteratively constructed during the proof.

$i = 1$: Consider the different types for node μ_1 :

- (a) μ_1 **is a P-node:** This case does not apply, since μ_2 is not an allocation node of v_1 .
- (b) μ_1 **is an S-node:** In this case, $\{v_1, r_1\}$ is a separation pair in H_1 and v_1 and r_1 lie in a common face in any embedding of H_1 ; see Figure 4.4(a). Thus, Γ_1 is set to an arbitrary embedding of H_1 and the empty path p_1 computed by the algorithm is an edge insertion path for v_1 and r_1 in H_1 with respect to Γ_1 .
- (c) μ_1 **is an R-node:** The graph G_1 constructed by the algorithm is the graph H_1 if r_1 is identified with vertex y_1^2 in the algorithm; compare Figure 4.4(b). Hence, Π_1 is also an embedding of H_1 and we define $\Gamma_1 := \Pi_1$. Since p_1 is an edge insertion path for v_1 and y_1^2 in G_1 with respect to Π_1 by construction, p_1 is also an edge insertion path for v_1 and r_1 in H_1 with respect to Γ_1 .

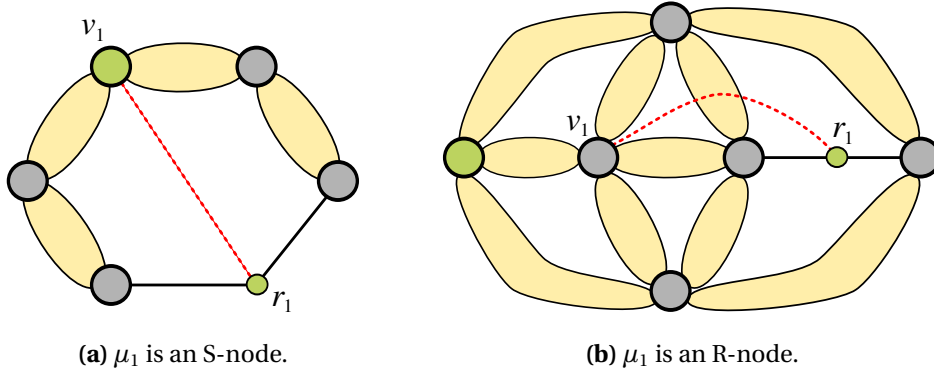


Figure 4.4: The different node types for the case $i = 1$.

$i > 1$: Assume now that $\Gamma_1, \dots, \Gamma_{i-1}$ are already constructed and $p_1 + \dots + p_{i-1}$ is an edge insertion path for v_1 and r_{i-1} in H_{i-1} with respect to Γ_{i-1} .

The graph G_i constructed in the algorithm contains a vertex x_i^1 adjacent to exactly two vertices, say a and b , and H_{i-1} contains vertex r_{i-1} adjacent to exactly two vertices, say a' and b' . By construction, both a and a' , as well as b and b' represent the same vertex of G and the graph H_i is obtained from G_i and H_{i-1} by identifying a and a' , b and b' , and removing the vertices x_i^1 and r_{i-1} (including their incident edges).

An embedding of H_i can be determined in the following way. Choose one of the two faces containing r_{i-1} as external face of Γ_{i-1} . This leads to an embedding in which either the last edge of $p_1 + \dots + p_{i-1}$ and r_{i-1} lie in a common face, or $p_1 + \dots + p_{i-1}$ is empty and v_1 and r_{i-1} lie in a common face. Then, determine an embedding of G_i , insert Γ_{i-1} into the embedding of G_i , and remove the vertices r_{i-1} and x_i^1 . It is also possible to mirror the embedding Γ_{i-1} before inserting, since $p_1 + \dots + p_{i-1}$ is still an edge insertion path in H_{i-1} with respect to the mirror embedding of Γ_{i-1} .

We distinguish the possible cases for the type of node μ_i :

- (a) **μ_i is an S-node:** There is just one embedding of G_i and inserting H_{i-1} into this embedding as described above leads to an embedding Γ_i of H_i such that $p_1 + \dots + p_{i-1} = p_1 + \dots + p_i$ is an edge insertion path in H_i with respect to Γ_i ; compare Figure 4.5(a).
- (b) **μ_i is a P-node:** Let Π_i be an embedding of G_i such that x_i^1 and x_i^2 lie in a common face. Obtain Γ_i by inserting Γ_{i-1} into Π_i in such a way that r_{i-1} and x_i^2 lie in a common face; compare Figure 4.5(b). This can be achieved by mirroring Γ_{i-1} if necessary. Then, $p_1 + \dots + p_{i-1} = p_1 + \dots + p_i$ is an edge insertion path in H_i with respect to Γ_i .
- (c) **μ_i is an R-node:** Let Π_i be the embedding computed by the algorithm. The list $p_i = e_1, \dots, e_\ell$ is an edge insertion path for x_i^1 and x_i^2 in G_i with respect to Π_i . We obtain the embedding Γ_i by inserting Γ_{i-1} into Π_i

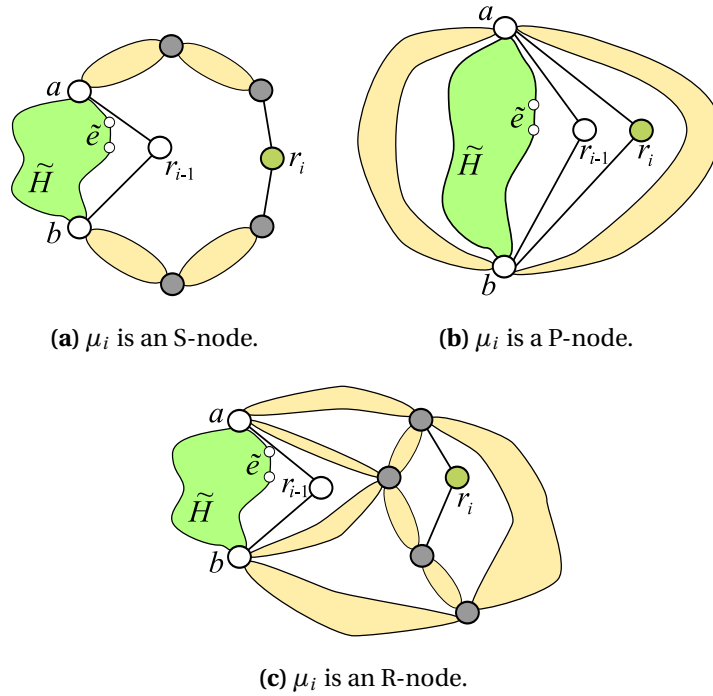


Figure 4.5: The different types for node μ_i . \tilde{H} denotes the graph $H_{i-1} - r_{i-1}$ and \tilde{e} denotes the last edge in $p_1 + \dots + p_{i-1}$.

in such a way that r_{i-1} and e_1 (or r_{i-1} and r_i if p_i is empty) lie in a common face; compare Figure 4.5(c). This is possible by mirroring Γ_i if necessary, since x_i^1 and e_1 (or x_i^1 and x_i^2 if p_i is empty) lie in a common face in Π_i .

Since $r_k = v_2$ and $H_k = G$, it follows that $p_1 + \dots + p_k$ is an edge insertion path for v_1 and v_2 in G with respect to $\Pi := \Gamma_k$ and the lemma holds. \square

Algorithm OPTIMALBLOCKINSERTER computes only an edge insertion path $p = e_1, \dots, e_\ell$ for the vertices v_1 and v_2 in G , but not the corresponding embedding of G . However, there is a simple way for finding an embedding Π such that p is an edge insertion path for v_1 and v_2 in G with respect to Π : First, construct a graph G' by splitting each edge e_i in p introducing a new vertex w_i and insert new edges forming a path $v_1, w_1, \dots, w_\ell, v_2$. Since p is an edge insertion path, the graph G' is planar and an embedding Π' for G' can be computed in linear time [see, for example, Hopcroft and Tarjan, 1974, Mehlhorn and Mutzel, 1996]. Replacing all split edges in Π' by original edges (thus removing the vertices w_1, \dots, w_ℓ and their adjacent edges again) results in an embedding Π for G such that p is an edge insertion path for v_1 and v_2 in G with respect to Π .

In order to prove the optimality of the edge insertion path p for v_1 and v_2 computed by the algorithm, we show that any edge insertion path for v_1 and v_2 is at least as long as p . Clearly, it is sufficient to consider a shortest edge insertion path for an arbitrary, fixed embedding.

Lemma 4.3. *Let Π' be an arbitrary embedding of G and let p' be a shortest edge insertion path for v_1 and v_2 in G with respect to Π' . Then, $|p'| \geq |p|$ holds.*

Proof. If the path $\Lambda = \mu_1, \dots, \mu_k$ computed by the algorithm contains no R-node, p is empty and $|p'| \geq |p| = 0$ obviously holds. Assume now that Λ contains at least one R-node.

Let μ_i be an R-node in Λ . Denote with S_i the modified skeleton of μ_i constructed in the algorithm that contains the vertices x_i^1 and x_i^2 as representatives of v_1 and v_2 , respectively. Let $G_i = (V_i, E_i \cup M_i)$ be the graph constructed in the algorithm such that E_i is the set of edges that results from expanding the unmarked edges and M_i denotes the set of marked edges in S_i . Since p' is a shortest edge insertion path for the embedding Π' , the edges in p' that are also contained in E_i form a subsequence $p'_i = e'_1, \dots, e'_{\ell'_i}$ of p' , and p'_i is an edge insertion path for x_i^1 and x_i^2 in G_i with respect to the embedding of G_i induced by Π' . We will show that $|p'_i| \geq |p_i|$, where p_i is the subsequence of p computed by the algorithm.

For each unmarked edge e in S_i , set the costs of e to the traversing costs $c(e)$ of e and define the length of an edge insertion path to be the sum of the costs of the edges in the path. All marked edges are incident to either x_i^1 or x_i^2 and will not appear in an edge insertion path we consider. Each edge insertion path for x_i^1 and x_i^2 in G_i induces an edge insertion path for x_i^1 and x_i^2 in S_i which contains all the skeleton edges whose expansion graphs are crossed in G_i .

Let \tilde{p}_i be the edge insertion path in S_i induced by p_i , and let \tilde{p}'_i be the edge insertion path in S_i induced by p'_i . Then

$$|p'_i| \geq |\tilde{p}'_i| = \sum_{e \in \tilde{p}'_i} c(e),$$

since crossing an expansion graph yields at least $c(e)$ crossings; see Lemma 4.1. Since p_i is a shortest edge insertion path for x_i^1 and x_i^2 in G_i with respect to Π_i , \tilde{p}_i is a shortest edge insertion path for x_i^1 and x_i^2 in S_i , which implies that $|\tilde{p}'_i| \geq |\tilde{p}_i| = |p_i|$ and thus $|p'_i| \geq |p_i|$.

Let $I = \{i \mid \mu_i \text{ is an R-node}\}$. Since all E_i , $i \in I$, are pairwise disjoint, it follows that

$$|p'| \geq \sum_{i \in I} |p'_i| \geq \sum_{i \in I} |p_i| = |p|.$$

□

Lemma 4.2 and Lemma 4.3 show that Algorithm OPTIMALBLOCKINSERTER computes an optimal edge insertion path for v_1 and v_2 in $G = (V, E)$. It remains to prove that its running time is linear in the size of G .

The SPQR-tree \mathcal{T} of G can be computed in time $\mathcal{O}(|V| + |E|)$; see Section 3.4. A path between two arbitrary allocation nodes of v_1 and v_2 can be found by inspecting each skeleton graph and using depth first search in the tree \mathcal{T} . The shortest path $\Lambda = \mu_1, \dots, \mu_k$ is obtained from this path by removing nodes from the start and the end of the path until it contains exactly one allocation node of v_1 and one allocation node of v_2 . Hence, finding path Λ takes time $\mathcal{O}(|V| + |E|)$, since the size of \mathcal{T} including all skeleton graphs is linear in the size of G .

The construction of the modified skeleton S_i , which results from splitting at most two edges in $skeleton(\mu_i)$, takes time linear in the size of $skeleton(\mu_i)$. Since the total size of all skeleton graphs in \mathcal{T} is linear in the size of G , the total time for constructing S_1, \dots, S_k is $\mathcal{O}(|V| + |E|)$.

Finally, consider graph $G_i = (V_i, E_i \cup M_i)$ for $1 \leq i \leq k$, where E_i is the set of edges that results from expanding the unmarked edges in S_i and M_i is the set of marked edges in S_i . Since $|M_i| \leq 4$ (and $|V_i| \leq |E_i|$), an arbitrary embedding of G_i is computed in time $\mathcal{O}(|E_i|)$ (see, for example, [Mehlhorn and Mutzel, 1996]), and the size of the extended dual graph A_i is $\mathcal{O}(|E_i|)$. Hence, a shortest path between x_i^1 and x_i^2 in A_i can be found in time $\mathcal{O}(|E_i|)$ using breadth first search. Since all the sets E_i are pairwise disjoint, the total time for constructing G_1, \dots, G_k and for finding p_1, \dots, p_k is $\sum_{i=1}^k \mathcal{O}(|E_i|) = \mathcal{O}(|E|)$. Thus, the following theorem holds:

Theorem 4.1. *Let $G = (V, E)$ be a biconnected planar graph and let v_1 and v_2 be two non-adjacent vertices in V . Then, Algorithm OPTIMALBLOCKINSERTER computes an optimal edge insertion path for v_1 and v_2 in G in time $\mathcal{O}(|V| + |E|)$.*

4.1.3 General Graphs

Let G be a connected, planar graph and \mathcal{B} its BC-tree. We say the *representative* of a vertex $v \in G$ in a block B is either v itself if $v \in B$, or the first cut-vertex c on a path in \mathcal{B} from B to a block B' containing v .

The Algorithm OPTIMALINSERTER for computing an optimal edge insertion path for a connected planar graph G and two non-adjacent vertices v_1 and v_2 is given in Listing 4.2. The algorithm constructs the BC-tree \mathcal{B} of G and considers only the blocks on the shortest path in \mathcal{B} connecting a representative B_1 of v_1 with a representative B_k of v_2 . For each block B_i , an optimal edge insertion path p_i for the representatives of v_1 and v_2 in B_i is computed using Algorithm OPTIMALBLOCKINSERTER (see Listing 4.1), and these paths are then concatenated. The following lemma shows that the resulting path $p_1 + \dots + p_k$ is indeed an optimal edge insertion path for v_1 and v_2 in G .

```

1: function OPTIMALINSERTER(graph  $G$ , vertex  $v_1$ , vertex  $v_2$ )
2:   Compute the BC-tree  $\mathcal{B}$  of  $G$ .
3:   Find the shortest path  $B_1, c_1, \dots, B_{k-1}, c_{k-1}, B_k$  in  $\mathcal{B}$  between a represen-
4:   tative  $B_1$  of  $v_1$  and  $B_k$  of  $v_2$ .
5:   for  $i = 1, \dots, k$  do
6:     let  $x_i$  and  $y_i$  be the representatives of  $v_1$  and  $v_2$  in  $B_i$ .
7:      $p_i :=$  OPTIMALBLOCKINSERTER( $B_i, x_i, y_i$ )
8:   end for
9:   return  $p_1 + \dots + p_k$ 
10: end function

```

Listing 4.2: Computes an optimal edge insertion path for a pair of non-adjacent vertices v_1, v_2 in a connected graph G .

Lemma 4.4. *Let $p_1 + \dots + p_k$ be the path computed by Algorithm OPTIMALINSERTER. Then, there exists an embedding Π of G such that $p_1 + \dots + p_k$ is an optimal edge insertion path for v_1 and v_2 in G with respect to Π .*

Proof. Let H_i be the union of the blocks B_1 to B_i . We show by induction that there is an embedding Γ_i of H_i such that $\Lambda_i := p_1 + \dots + p_i$ is an optimal edge insertion path in H_i for v_1 and y_i .

$i = 1$: In this case, H_1 equals B_1 and by Theorem 4.1, there is an embedding Γ_1 such that $\Lambda_1 = p_1$ is an optimal edge insertion path for $x_1 = v_1$ and y_1 in H_1 with respect to Γ_1 .

$i > 1$: Assume now that $\Gamma_1, \dots, \Gamma_{i-1}$ are already constructed such that Λ_{i-1} is an optimal edge insertion path for v_1 and y_{i-1} in H_{i-1} with respect to Γ_{i-1} .

By Theorem 4.1, there exists an embedding Π_i of B_i such that p_i as constructed in the algorithm is an optimal edge insertion path for x_i and y_i in B_i with respect to Π_i . Since y_{i-1} and x_i denote the same vertex in G , the embedding Γ_i of H_i can be constructed as follows. Since Λ_{i-1} is an edge insertion path for v_1 and y_{i-1} , there is face $f \in \Gamma_{i-1}$ that contains y_{i-1} and either v_1 if Λ_{i-1} is empty, or the last edge in Λ_{i-1} . Analogously, there is a face $f' \in \Pi_i$ that contains x_i and either y_i if p_i is empty, or the first edge in p_i . The embedding Γ_i is constructed by choosing f as external face of Γ_{i-1} and placing this planar embedding of H_{i-1} into face f' of Π_i . This is possible, since $B_1 \cup \dots \cup B_{i-1}$ and B_i have only the vertex $y_{i-1} = x_i$ in common. Thus, $\Lambda_i = p_1 + \dots + p_i$ is an edge insertion path for v_1 and y_i in H_i with respect to Γ_i .

It remains to show the optimality of Λ_i . Let \hat{p} be an arbitrary edge insertion path for v_1 and y_i in H_{i-1} with respect to some embedding $\hat{\Gamma}$. Obviously, \hat{p} can be partitioned into two subpaths \hat{p}_A and \hat{p}_B such that \hat{p}_B contains only the edges in B_i . Then, \hat{p}_B is an edge insertion path for x_i and y_i in B_i with respect to the embedding of B_i induced by $\hat{\Gamma}$, and \hat{p}_A is an edge insertion path for v_1 and y_{i-1} in H_{i-1} with respect to the embedding of H_{i-1} induced by $\hat{\Gamma}$. Since $|p_i| \leq |\hat{p}_B|$ by Theorem 4.1 and $|\Lambda_{i-1}| \leq |\hat{p}_A|$ by induction hypothesis, it follows $|\Lambda_i| \leq |\hat{p}|$.

Since a block shares only a single vertex with the rest of the graph, it is easy to see that Λ_k is still an edge insertion path for v_1 and $y_k = v_2$ in G with respect to an embedding Π that results from inserting the remaining blocks not contained in B_1, \dots, B_k arbitrarily into Γ_k .

The optimality of Λ_k in G can be shown using a similar argument as in the induction step. Let \hat{p} be an arbitrary edge insertion path for v_1 and v_2 in G . The subpath \hat{p}_B of \hat{p} containing only the edges in H_k is an edge insertion path for v_1 and y_k in H_k . Thus, $|\hat{p}| \geq |\hat{p}_B| \geq |\Lambda_k|$. \square

The BC-tree of $G = (V, E)$ can be computed in time $\mathcal{O}(|V| + |E|)$ by finding the blocks of G ; see 3.1.1. Since all blocks are pairwise edge-disjoint, the size of \mathcal{B} is

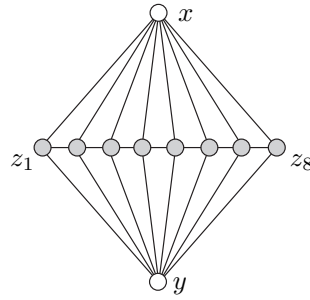


Figure 4.6: A wall with width 8.

$\mathcal{O}(|V|+|E|)$ and the path from B_1 to B_2 in \mathcal{B} can be found in time $\mathcal{O}(|V|+|E|)$ using, for example, depth first search. Algorithm `OPTIMALBLOCKINSERTER` is called for each block $B_i = (V_i, E_i)$ which takes time $\mathcal{O}(|V_i| + |E_i|)$ according to Theorem 4.1. Since all blocks are pairwise edge-disjoint, Algorithm `OPTIMALINSERTER` takes time $\mathcal{O}(|V|+|E|)$.

The algorithm presented in this section can easily be generalized to arbitrary planar graphs. If v_1 and v_2 belong to the same connected component, simply apply Algorithm `OPTIMALINSERTER`. Otherwise, the graph $G \cup \{(v_1, v_2)\}$ is obviously planar and the empty path is the optimal edge insertion path. Hence, we get the following result.

Theorem 4.2. *Let $G = (V, E)$ be a planar graph and let v_1 and v_2 be two non-adjacent vertices in V . Then, there exists an algorithm that computes an optimal edge insertion path for v_1 and v_2 in G in time $\mathcal{O}(|V|+|E|)$.*

4.1.4 Near-Planar Graphs

A non-planar graph G that can be made planar by removing a single edge, that is, there is an edge $e = (v_1, v_2) \in G$ such that $G - e$ is planar, is called a *near-planar* graph. Considering such a near-planar graph G , the one-edge insertion problem can also be stated as follows: Given a near-planar graph G and an edge $e = (v_1, v_2) \in G$ with $G' := G - e$ is planar, find a drawing of G that has the minimum number of crossings among all drawings of G in which every crossing involves the edge e .

We show in this section, that such a drawing of G is not necessarily crossing minimal. In particular, we give a class of graphs G'_m such that a solution of the OEIP for G'_m and (v_1, v_2) results in a drawing with m crossings, whereas a crossing minimal drawing of $G_m = G'_m + (v_1, v_2)$ has only two crossings.

First, we define a wall graph as follows. A *wall* with width k consists of the vertices x, y, z_1, \dots, z_k , the edges (z_i, z_{i+1}) for $1 \leq i < k$, and the edges (x, z_i) and (y, z_i) for $1 \leq i \leq k$; see Figure 4.6. The vertices x and y are called the *poles* of the wall. A wall with width ≥ 3 is a triconnected planar graph.

For an even number $m \geq 2$, the graph G'_m is constructed in the following way; compare Figure 4.7. We start with a ring of walls W_1, \dots, W_6 with width $m + 1$,

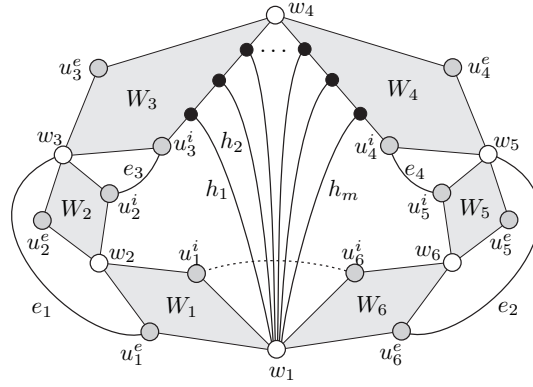


Figure 4.7: The graph G'_m ; each shaded region represents a wall with width $m + 1$. The dashed edge (u_1^i, u_6^i) is the edge to be inserted.

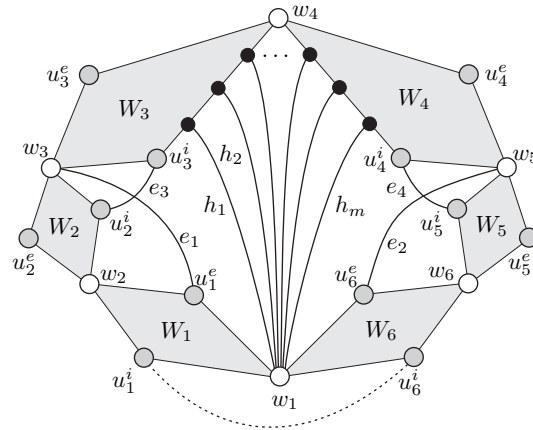


Figure 4.8: A drawing of the graph G_m with only two crossings.

where the poles of adjacent walls in the ring are identified. We denote the pole vertices with w_1, \dots, w_6 such that the poles of W_1 are w_1 and w_2 , and so forth. For each wall W_j , the other two vertices on the boundary are denoted with u_j^i and u_j^e , where u_j^i is inside the ring and u_j^e is on the external face; see Figure 4.7. Moreover, the edges $e_1 = (u_1^e, w_3)$, $e_2 = (u_6^e, w_5)$, $e_3 = (u_2^i, u_3^i)$, $e_4 = (u_5^i, u_4^i)$ are added, $m/2$ vertices are inserted by splitting edge (u_3^i, w_4) , $m/2$ vertices are inserted by splitting edge (w_4, u_4^i) , and every created split vertex is connected with vertex w_1 by an edge h_j , $1 \leq j \leq m$. The two vertices to be connected are $v_1 := u_1^i$ and $v_2 := u_6^i$, that is, $G_m = G'_m \cup \{(u_1^i, u_6^i)\}$.

By construction, G'_m is triconnected and planar. In particular, G'_m has only two embeddings which are mirror-images of each other. It is easy to see that an optimal edge insertion path for v_1 and v_2 has length m (by crossing the edges h_1, \dots, h_m), since passing through a wall would require at least $m + 1$ crossings. On the other hand, there is a drawing of G_m with only 2 crossings as shown in Figure 4.8. Here, only the two crossings e_1 with e_3 and e_2 with e_4 occur, independent of the choice of m .

In spite of this negative result, Hliněný and Salazar [2006] have shown that an optimal solution to the one-edge insertion problem approximates the crossing number of a near-planar graph $G = G' + (v_1, v_2)$ by a factor of its maximum degree $\Delta(G)$, that is, the number of crossings in an optimal solution of OEIP for G' and (v_1, v_2) is at most $\Delta(G) \cdot \text{cr}(G)$. This yields a constant factor approximation for near-planar graphs with bounded degree. The approximation factor could recently be improved to $\Delta(G)/2$ by Cabello and Mohar [2009a,b]. Hence, we conclude this section with the following theorem:

Theorem 4.3. *Let $G = G' + (v_1, v_2)$ be a near-planar graph with bounded degree Δ . Then the optimal one-edge insertion algorithm computes a $\frac{\Delta}{2}$ -approximation of the crossing number of G in linear time.*

More precisely, if $\text{oei}(G', v_1, v_2)$ is the number of crossings in the solution of OEIP for G' and (v_1, v_2) , then

$$\text{oei}(G', v_1, v_2) \leq \frac{\Delta}{2} \cdot \text{cr}(G).$$

This approximation result appears to be even stronger in the light of a very recent result by Cabello and Mohar [2010], where they could show that the crossing number problem for near-planar graphs is already NP-hard. Previously, NP-hardness of crossing number for near-planar graphs was only known for weighted graphs [Cabello and Mohar, 2009b].

4.2 Crossing Minimization Heuristics

[Di Battista et al., 1997] have conducted an extensive experimental study, comparing four general-purpose graph drawing algorithms for producing orthogonal grid drawings with respect to aesthetic criteria like number of crossings, number of bends, edge lengths, and drawing area. Two of these algorithms were based on the topology-shape-metrics approach applying graph planarization for minimizing crossings, and the others were incremental algorithms focusing on a small area and a small number of bends. They also introduced a new benchmark set of graphs which is now widely used in graph drawing and is commonly referred to as the *Rome graphs*. The study showed that the approaches based on the topology-shape-metrics approach were the clear winners; especially with respect to number of crossings, they outperformed the other algorithms by far. For example, the other two algorithms required 8 and 4 times more crossings, respectively, for the largest graphs.

In this section, we introduce crossing minimization heuristics based on the planarization method. Besides the standard heuristic used in the study by [Di Battista et al., 1997] (edge re-insertion with fixed embedding; see Section 4.2.3), we present very effective pre- and postprocessing methods, apply the optimal algorithm for the OEIP, and use a randomized permutation scheme for edge re-insertion. In the next section, we show the results of a comprehensive experimental evaluation of these heuristics, where we also use the Rome graphs

as benchmark set, as well as a new benchmark set of constructed graphs with known crossing numbers.

4.2.1 Preprocessing

It is well known that the crossing number of a graph is equal to the sum of the crossing numbers of its blocks, that is, we can process the blocks separately and thus perform the actual crossing minimization step for biconnected graphs only. If B_1, \dots, B_k are the blocks of G , then

$$\text{cr}(G) = \sum_{i=1}^k \text{cr}(B_i) .$$

The idea of our preprocessing is to reduce these biconnected graphs further to a smaller core graph with the same crossing number. We introduced such a method, called the *non-planar core reduction*, in Chimani and Gutwenger [2009]. This reduction method is not only applicable to the crossing number problem, but also to the skewness, thickness, and coarseness of a graph. The main motivation for such a preprocessing comes from exact algorithms for solving the crossing number problem, for example using branch-and-cut. In this case, reducing the number of edges in the graph allows to reduce the number of (potential) variables significantly, giving hope to be able to solve larger instances. In our experimental study, we will evaluate if such a preprocessing is worthwhile for crossing minimization heuristics as well.

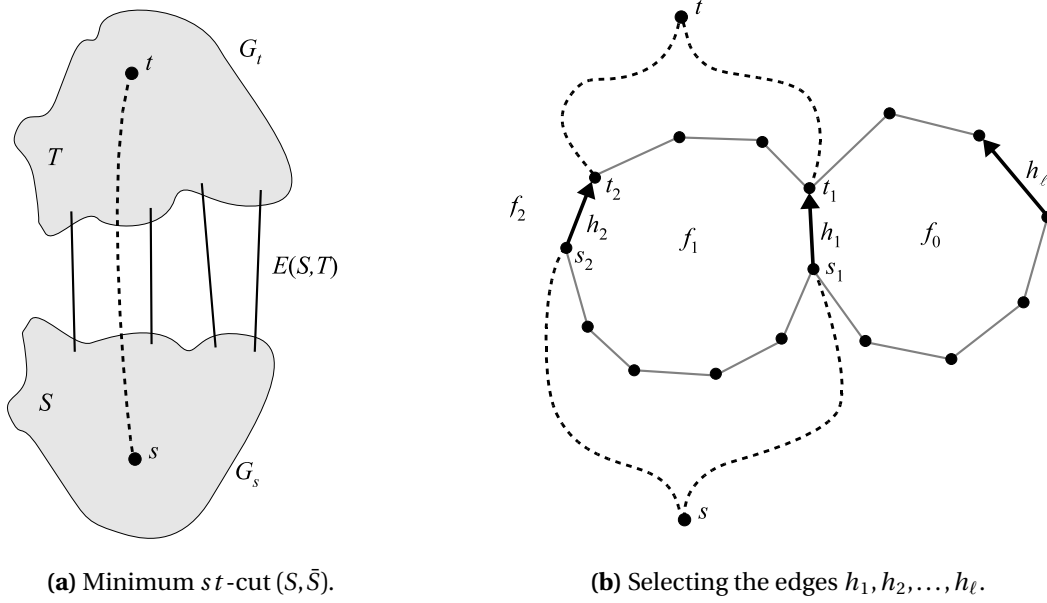
A basic observation motivating the non-planar-core reduction is the relationship between a traversing path (as defined in Section 4.1.1) and minimum st -cuts. It is easy to see that a traversing path for an edge (s, t) defines an st -cut in the following sense:

Lemma 4.5. *Let $G + (s, t)$ be a biconnected and planar graph, and let Γ be an embedding of G . If e_1, \dots, e_k is a traversing path of Γ with respect to (s, t) , then there exists an st -cut (S, \bar{S}) in G with $E(S, \bar{S}) = \{e_1, \dots, e_k\}$.*

Proof. By the definition of a traversing path, we can draw a Jordan curve in a drawing realizing Γ that crosses exactly the edges e_1, \dots, e_k and divides the plane into two regions: one region R_s containing s and one region R_t containing t . Let S be the set of vertices in R_s and \bar{S} be the set of vertices in R_t . Then, every edge in $E' = \{e_1, \dots, e_k\}$ connects a vertex in S with a vertex in \bar{S} and there is no edge in $E \setminus E'$ that connects a vertex in S with a vertex in \bar{S} . Hence, $E' = E(S, \bar{S})$ and (S, \bar{S}) is an st -cut. \square

The following theorem shows that this st -cut is even a minimum st -cut.

Theorem 4.4. *Let $G = (V, E)$ be a graph with $s, t \in V$ such that $G + (s, t)$ is biconnected and planar. Then, the traversing costs of G with respect to (s, t) are equal to $\text{mincut}_{s,t}(G)$.*

(a) Minimum st -cut (S, \bar{S}) .(b) Selecting the edges h_1, h_2, \dots, h_ℓ .**Figure 4.9:** Proof of Theorem 4.4.

Proof. Let λ be the capacity of a minimum st -cut in G and κ the traversing costs of G with respect to (s, t) . By Lemma 4.5, we have $\lambda \leq \kappa$. We have to show that $\kappa \leq \lambda$. Let Γ be an arbitrary embedding of $G' = G + (s, t)$ and let Γ^* be the corresponding dual graph. Furthermore, let (S, \bar{S}) be a minimum cut with $s \in S$ and $t \in \bar{S}$. Since G is connected and the cut (S, \bar{S}) is also minimal, removing the edges $E(S, \bar{S})$ splits G into two connected graphs $G_s = (S, E_s)$ and $G_t = (\bar{S}, E_t)$; see Figure 4.9(a). We can write the edges in $E(S, \bar{S})$ as $e_1 = (s_1, t_1), \dots, e_\lambda = (s_\lambda, t_\lambda)$ such that $s_i \in S$ and $t_i \in \bar{S}$ for $1 \leq i \leq \lambda$. Moreover, there is a path from s to s_i in G_s and from t_i to t in G_t for every $1 \leq i \leq \lambda$.

We show in the following that the dual edges of (possibly a subset of) e_1, \dots, e_λ , and (s, t) form a cycle $h_1^*, f_1, h_2^*, \dots, h_\ell^*, f_\ell$ in Γ^* , where h_i^* denotes the dual of the edge h_i . Obviously, (s, t) is one of the edges h_i . This implies that removing the edges h_1, \dots, h_ℓ splits G into two parts which must be G_s and G_t . Then, it follows that $\kappa \leq \ell - 1 \leq \lambda$ and the theorem holds.

We start our construction with $h_1 = (s_1, t_1)$. Let f_0 be the face right of h_1 and f_1 the face left of h_1 ; compare Figure 4.9(b). Since G' is biconnected, h_1 is not a bridge and hence $f_0 \neq f_1$. Since f_1 is also a cycle in G' and the cut separates s_1 and t_1 , there must be another edge $h_2 = (s_2, t_2)$ in $E(S, \bar{S})$ that is on f_1 . Let f_2 be the face right of h_2 . We distinguish two cases. If f_2 is one of the faces f_0 and f_1 , then we have found a cycle in Γ^* and we are done. Otherwise, there must be an edge $h_3 \in E(S, \bar{S})$ with $h_3 \neq h_2$ and h_3 is on f_2 , since f_2 is a cycle. We can continue this construction until we end up with an edge h_ℓ such that the left face of h_ℓ is one of the faces $f_0, \dots, f_{\ell-1}$. The construction will terminate, since $E(S, \bar{S})$ is an st -cut. \square

Definition of the Non-planar Core

We consider now a biconnected and non-planar graph $G = (V, E)$. We define a planar 2-component in G as follows:

Definition 4.2. Let $s, t \in V$ be two distinct vertices. We call an edge induced subgraph $C = G[E_C]$ a *planar 2-component* of G with *contact points* s and t if $C + (s, t)$ is planar and $V(C) \cap V' = \{s, t\}$, where $V' = V(G[E \setminus E_C])$ denotes the vertex set of the graph induced by the edges not contained in C . For brevity, we also call C a *planar st -component*. Moreover, a single edge $e = (s, t)$ is a *trivial planar st -component*.

A planar st -component C is basically a subgraph that is only connected at its contact points with the rest of the graph. The biconnectivity of G implies that $C + (s, t)$ is biconnected. For the definition of the non-planar core, we are only interested in maximal planar 2-components:

Definition 4.3. Let C be a non-trivial planar 2-component of G . We call C a *maximal planar 2-component* of G , if and only if there is no planar 2-component C^* of G with $C \subset C^*$.

An important property of maximal planar 2-components is that they do not overlap each other (except for their contact points):

Lemma 4.6. *All maximal planar 2-components of G are pairwise vertex and edge disjoint, except for their contact points.*

Proof. Consider two distinct maximal planar 2-components C_1 and C_2 with contact points s_1, t_1 and s_2, t_2 , respectively. Let

$$\bar{V} = (V(C_1) \cap V(C_2)) \setminus \{s_1, t_1, s_2, t_2\}$$

be the common vertices between the two components, disregarding their contact points, and assume that $\bar{V} \neq \emptyset$.

Since C_1 and C_2 are maximal, $E_1 = E(C_1) \setminus E(C_2)$ and $E_2 = E(C_2) \setminus E(C_1)$ are non-empty sets, that is, C_2 contains edges not in C_1 and vice versa. Since C_2 is a planar 2-component, only its contact points are incident to the rest of the graph. Hence E_1 cannot contain edges incident to \bar{V} . Analogously, E_2 cannot contain edges incident to \bar{V} .

Hence, the edges in C_1 and C_2 incident to s_1, t_1 and s_2, t_2 , respectively, are in $E(C_1) \cap E(C_2)$. Thus C_1 contains the contact points of C_2 and vice versa. Since they are both planar 2-components, they are only connected to the rest of the graph via two contact points, hence the pairs of their respective contact points have to be identical. But then, the union $C_1 \cup C_2$ would also be a planar 2-component, which contradicts the maximality of C_1 and C_2 .

If C_1 and C_2 are vertex disjoint, except for their contact points, they are also edge disjoint, except for edges (s_1, t_1) with $s_1 = s_2$ and $t_1 = t_2$. Then, $C_1 \cup C_2$ would be a planar 2-component, which again contradicts the maximality of C_1 and C_2 . \square

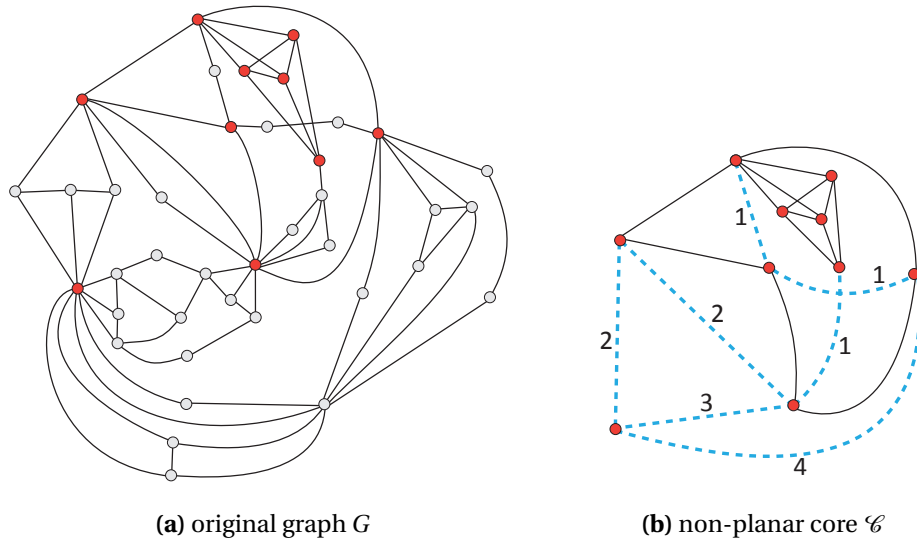


Figure 4.10: Example for the non-planar core reduction of a graph.

The main idea of the non-planar core reduction is to replace each maximal planar st -component by a single edge (s, t) . Obviously, routing an edge through a planar 2-component in a drawing may produce more crossings than just crossing a single edge. Therefore, we assign edge weights to the edges representing a maximal planar 2-component, reflecting the number of crossings that will occur when crossings through this 2-component. Accordingly, we define the non-planar core as follows:

Definition 4.4 (Non-planar core). The *non-planar core* (\mathcal{C}, w) of G is a graph \mathcal{C} with a weight function $w : E \rightarrow \mathbb{N}$ such that \mathcal{C} is a copy of G in which each maximal planar st -component C of G is substituted by a *virtual edge* $e_C = (s, t)$ with weight $w(e_C) = \text{mincut}_{s,t}(C)$, and each non-virtual edge e has weight $w(e) = 1$.

Figure 4.10 shows an example of a graph G and its non-planar core \mathcal{C} . By Lemma 4.6 above we have that the non-planar core is well-defined and unique.

Construction of the Non-planar Core

It turns out that the non-planar core of a graph G can easily be constructed using its SPQR-tree \mathcal{T} . Clearly, only skeletons of R-nodes can be non-planar. If we determine which R-nodes have a non-planar skeleton, it is easy to find planar st -components of maximal size. Let \mathcal{S} be a copy of \mathcal{T} and denote with $Q[\mathcal{S}]$ the tree obtained from \mathcal{S} by removing all the Q-nodes.

Assume $Q[\mathcal{S}]$ contains a leaf α with planar skeleton. Let μ be its adjacent tree node in $Q[\mathcal{S}]$ and let $e_\alpha = (s, t)$ be the edge in skeleton of μ whose pertinent node is α . Then, we replace α (and all adjacent Q-nodes) by a single Q-node representing a virtual edge (s, t) , thereby making this Q-node the new pertinent node of e_α .

We perform these replacements as long as $Q[\mathcal{S}]$ contains a leaf α with planar skeleton. The resulting tree \mathcal{S} contains Q-nodes representing virtual edges which will be the virtual edges in the non-planar core. However, we need two further adjustments for fulfilling the maximality condition of replaced planar 2-components: First, we inspect P-nodes in the resulting tree. If such a node is adjacent to more than one Q-node in \mathcal{S} , then we can merge all these edges to a single edge, and all corresponding Q-nodes to a single Q-node. Moreover, we have to consider S-nodes. If the skeleton of an S-node contains a path p such that the pertinent nodes of the edges in p are all Q-nodes, we can also replace p by a single edge (and thus replace the corresponding Q-nodes by a single Q-node). Hence, we do this replacement for all maximal paths in skeletons of S-nodes.

Finally, we obtain the non-planar core of G from \mathcal{S} by merging the skeleton graphs of its tree nodes. Since we created Q-nodes in \mathcal{S} that represent virtual edges, the core graph contains virtual edges as well. For these edges, we still have to compute their weights (all other edges have simply weight one), which are simply their traversing costs and thus can be computed efficiently. Hence, we can compute the non-planar core in linear time:

Theorem 4.5. *Let $G = (V, E)$ be a biconnected graph. Then, the non-planar core of G and the corresponding edge weights can be computed in $\mathcal{O}(|V| + |E|)$ time.*

Proof. See [Chimani and Gutwenger, 2009] for further details. \square

Figure 4.11 illustrates this reduction strategy for the original graph shown in Figure 4.10. The SPQR-tree \mathcal{T} of G has two R-nodes R_0 and R_3 with a non-planar skeleton. The resulting reduced tree \mathcal{S} consists of these two R-nodes, an S-node, 5+9 Q-nodes representing real edges, and 6+1 Q-nodes representing virtual edges (displayed in blue).

Application to the Crossing Number

Since the non-planar core is a weighted graph, we need to generalize the crossing number problem to weighted graphs. Therefore, we count a crossing between two edges with weights w_1 and w_2 as $w_1 \cdot w_2$ many crossings. This is equivalent to replacing an edge with weight w by a bundle of w parallel edges. Accordingly, we define the *crossing weight* of a drawing as the sum of these weighted crossings.

To prove the equivalence between the crossing number of a graph and its non-planar core, we first need the following lemma. It allows us to restrict the crossings in which the edges of a planar 2-component may be involved so that we can still obtain a crossing minimal drawing of G . A similar result has been reported by Širáň [1984]. However, as pointed out in [Chimani et al., 2007], the proof given by Širáň is not correct.

Lemma 4.7. *Let $C = (V_C, E_C)$ be a planar st -component of $G = (V, E)$. Then, there exists a crossing minimal drawing \mathcal{D}^* of G such that the induced drawing \mathcal{D}_C^* of C has the following properties:*

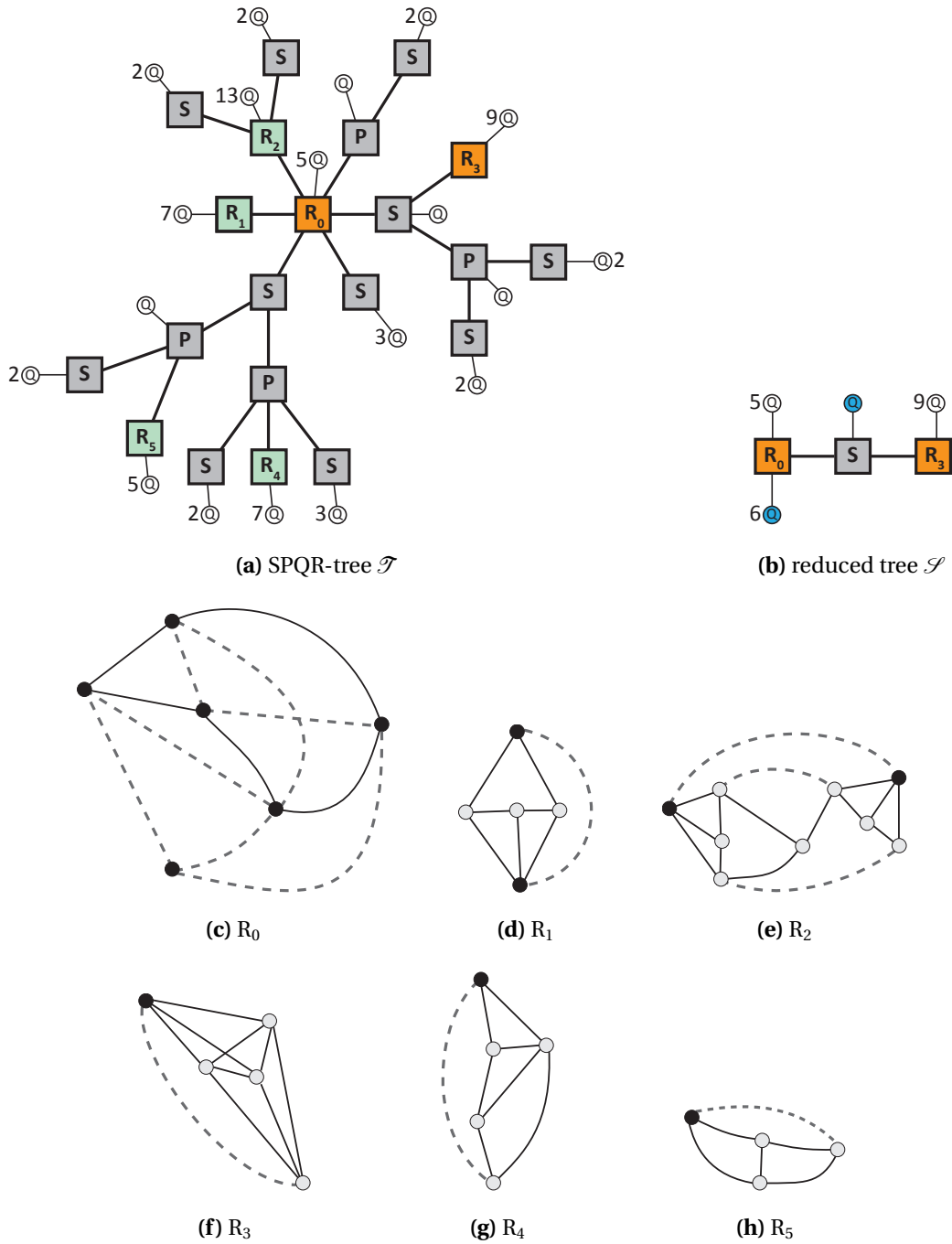


Figure 4.11: SPQR-tree \mathcal{T} of example graph (a) and resulting reduced tree \mathcal{S} ; R-nodes with non-planar skeletons are displayed in orange and Q-nodes in \mathcal{S} that represent virtual edges in blue; multiple Q-nodes adjacent to a common tree node are drawn as one Q-node with given multiplicity. The skeleton graphs of the R-nodes in \mathcal{T} are shown in (c) through (h).

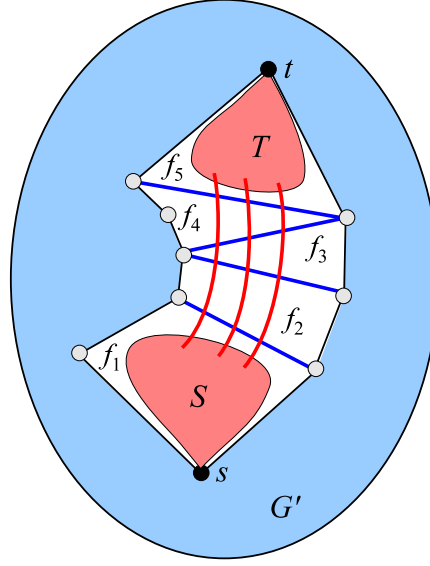


Figure 4.12: Final drawing \mathcal{D}^* of G ; here, $p = f_1, f_2, f_3, f_4, f_5$ is the shortest path in Γ_p^* .

- (a) \mathcal{D}_C^* contains no crossings;
- (b) s and t lie in a common face f_{st} of \mathcal{D}_C^* ;
- (c) all vertices in $V \setminus V_C$ are drawn in the region of \mathcal{D}^* defined by f_{st} ; and
- (d) there is a set $E_s \subseteq E_C$ with $|E_s| = \text{mincut}_{s,t}(C)$ such that any edge $e \in E \setminus E_C$ may only cross through all edges of E_s , or through none of E_C .

Proof. Let $G' = G[E \setminus E_C]$ be the graph that results from cutting C out of G . Let \mathcal{D} be an arbitrary, crossing minimal drawing of G , and let \mathcal{D}_C (respectively \mathcal{D}') be the induced drawing of C (respectively G'). We denote with P the planarized representation of G' induced by \mathcal{D}' , that is, the planar graph obtained from \mathcal{D}' by replacing edge crossings with dummy vertices. Let Γ_p be the corresponding embedding of P and Γ_p^* the dual graph of Γ_p .

Let $p = f_1, \dots, f_{k+1}$ be a shortest path in Γ_p^* that connects an adjacent face of s with an adjacent face of t . There are $\lambda = \text{mincut}_{s,t}(C)$ edge disjoint paths from s to t in C . Each of these λ paths crosses at least k edges of G' in the drawing \mathcal{D} . Hence, there are at least $\lambda \cdot k$ crossings between edges in C and edges in G' . We denote with E_p the set of primal edges of the edges on the path p . Let \mathcal{D}_C^* be a planar drawing of C in which s and t lie in the same face f_{st} , and let E_s be the edges in a traversing path in \mathcal{D}_C^* with respect to s and t . By Theorem 4.4, there is a minimum st -cut (S, \bar{S}) with $E(S, \bar{S}) = E_s$, and thus $|E_s| = \lambda$. We can combine \mathcal{D}' and \mathcal{D}_C^* by placing the drawing of $C[S]$ in face f_1 and the drawing of $C[\bar{S}]$ in f_{k+1} , such that all the edges in E_p cross all the edges in E_s ; see Figure 4.12. It is easy to verify that the conditions (a)–(d) hold for the resulting drawing \mathcal{D}^* . \square

The following theorem finally shows that it is sufficient to compute the crossing number of the non-planar core.

Theorem 4.6. *Let G be a biconnected graph and let (\mathcal{C}, w) be its non-planar core. Then,*

$$\text{cr}(G) = \text{cr}(\mathcal{C}, w).$$

Proof. “ \leq ” Let $\mathcal{D}_{\mathcal{C}}$ be a drawing of \mathcal{C} with minimum crossing weight. For each virtual edge $e = (s, t) \in \mathcal{C}$, we replace e by a planar drawing \mathcal{D}_e of the corresponding planar st -component so that all edges crossing e in $\mathcal{D}_{\mathcal{C}}$ cross the edges in a traversing path in \mathcal{D}_e with respect to (s, t) . Since $w(e)$ is equal to the traversing costs of \mathcal{D}_e with respect to (s, t) by definition, replacing all virtual edges in this way leads to a drawing of G with $\text{cr}(\mathcal{C}, w)$ crossings, and hence $\text{cr}(G) \leq \text{cr}(\mathcal{C}, w)$.

“ \geq ” On the other hand, let \mathcal{D} be a crossing minimal drawing of G . For each virtual edge $e = (s, t) \in \mathcal{C}$, we modify \mathcal{D} in the following way. Let C be the planar st -component corresponding to e and let G' be the rest of the graph. By Lemma 4.7, we obtain another crossing minimal drawing of G if we replace the drawing of C with a planar drawing \mathcal{D}_C of C such that all edges of G' that cross edges in C will cross the edges in $E(S, \bar{S})$, where (S, \bar{S}) is a minimum st -cut in C . If we replace \mathcal{D}_C with an edge $e = (s, t)$ with weight $w(e) = |E(S, \bar{S})| = \text{mincut}_{s,t}(C)$, we obtain a drawing with the same crossing weight.

By replacing all virtual edges in that way, we obtain a drawing of \mathcal{C} whose crossing weight is the crossing number of G . It follows that $\text{cr}(G) \geq \text{cr}(\mathcal{C}, w)$, and hence the theorem holds. □

4.2.2 Planar Subgraphs

For the first step of the planarization method, we need to find a feasible solution to the *maximum planar subgraph problem*, which has been shown to be NP-hard [Liu and Geldmacher, 1979]. If the number of edges to be deleted is small, the exact branch-and-cut algorithm suggested in [Jünger and Mutzel, 1996] is able to provide a provably optimal solution quite fast. However, the method is quite complicated to understand and to implement. Moreover, if the number of deleted edges exceeds 10, the algorithm usually needs far too long to be acceptable for practical computation. Since we are interested in approaches for practitioners, we did not include this exact method in our studies. Interested readers are referred to the study by Ziegler [2000] concerning the number of deleted edges in the Rome library benchmark set.

A widely used standard heuristic for finding a maximal planar subgraph is to start with the empty graph, and to iteratively try to add the edges one by one. In every step, a planarity testing algorithm is called for the obtained graph. If the addition of an edge would lead to a non-planar graph, then the edge is disregarded; otherwise, the edge is added permanently to the planar graph obtained so far. After $|E|$ iterations (planarity tests), we have obtained a maximal planar subgraph P

of G , that is, a subgraph of G which will get non-planar as soon as any of the edges in $G - P$ will be added. We will denote this method as `MAXIMAL`. The standard (and also our) implementation needs a running time of $\mathcal{O}(|E| \cdot |V|)$. Theoretically, this can be improved to nearly linear running time using incremental planarity testing algorithms [for example, Di Battista and Tamassia, 1996a, La Poutré, 1994], which apply dynamic updates of SPQR-trees (or a suitable representation of the triconnected components of the graph).

An alternative to this method is to use the planar subgraph algorithm based on PQ-trees as suggested in [Jayakumar et al., 1989, Jünger et al., 1998]. Observe, that this method cannot guarantee to derive a maximal planar subgraph. The theoretical worst case running time is $\mathcal{O}(|V|^2)$, but in practice it is usually much faster.

The quality of the results can be improved by introducing random events and calling the algorithm several times. The PQ-tree based algorithm starts by computing an st -numbering of G . Our random event was simply to choose a random edge $(s, t) \in E$. We studied the effects of up to 100 calls. We denote these methods as PQ1, PQ10, PQ25, PQ50, and PQ100 for 1, 10, 25, 50, and 100 iterations.

4.2.3 Edge Re-Insertion

The edge re-insertion step is also an NP-hard optimization problem [Ziegler, 2000]. Listing 4.3 shows the general framework for the edge insertion heuristics used in this study. The essential components and possible variations are discussed in the following.

Fixed Embedding. The standard algorithm used in practice re-inserts the edges e_1, e_2, \dots, e_k iteratively starting with a fixed planar embedding Π of P , see Section 2.4.2. The theoretical worst case running time of our implementation for inserting k edges is

$$\mathcal{O}\left(\sum_{i=1}^k (|V| + \sum_{j=1}^{i-1} c_j)\right) = \mathcal{O}(k(|V| + |C|)),$$

where c_j is the number of crossings introduced in step j and $C = c_1 + \dots + c_k$ is the number of crossings in the final drawing. In practice, the implementation performs usually much better, since the dual graph is updated incrementally, involving only those regions in which changes actually occurred. We denote this re-insertion method as `FIX`.

Variable Embedding. When fixing the embedding, the quality of the resulting drawing highly depends on the chosen embedding for P . The algorithm presented in the previous section can solve the OEIP to optimality and thus chooses an optimal embedding for inserting each edge. Our implementation has the

```

Input: planar subgraph  $P = (V, E_p)$  of  $G = (V, E)$ 
Output: planarized representation  $G_p^*$  of  $G$ 
1: Let  $E \setminus E_p = \{e_1, \dots, e_k\}$ 
2:  $best := \infty$ 
3:  $\triangleright$  Randomized permutations
4: for  $i := 1$  to  $nPermutations$  do
5:   Let  $\sigma$  be a randomly chosen permutation of  $\{1, \dots, k\}$ 
6:    $G_p := P$ 
7:    $\triangleright$  Edge re-insertion
8:   for  $j := 1$  to  $k$  do
9:     Insert edge  $e_{\sigma(j)}$  into  $G_p$ 
10:  end for
11:   $\triangleright$  Postprocessing
12:  Determine a set  $R \subseteq E$  of edges for which postprocessing shall be applied
13:  repeat
14:    for all  $e \in R$  do
15:      Remove edge  $e$  from  $G_p$ 
16:      Insert edge  $e$  into  $G_p$ 
17:    end for
18:  until number of crossings in  $G_p$  has not decreased
19:   $current :=$  number of crossings in  $G_p$ 
20:  if  $current < best$  then
21:     $G_p^* := G_p; best := current$ 
22:  end if
23: end for

```

Listing 4.3: The edge insertion step with postprocessing and permutations.

same worst-case running time as the variant `FIX`. However, we did not implement an incremental update procedure for the variable embedding setting. We denote this re-insertion method as `VAR`.

Constrained Crossing Minimization. Obviously, re-insertion of all edges at the same time will improve the solution. However, no practically efficient algorithm is known. The *constrained crossing minimization problem* asks for the minimum number of crossings required for inserting a set of edges F into a fixed embedding. The problem has been investigated in [Mutzel and Ziegler, 1999, Ziegler, 2000]. Experiments show that it can only be solved to provable optimality if there are less than 10 edges to be re-inserted—and even then, the running time is relatively high. Therefore, we did not include this method into our experiments.

Postprocessing Strategies. After all edges have been inserted, a simple post-processing technique tries to improve the current solution. It determines a set of edges R which have one or more crossings and repeatedly tries to find a better insertion path for each of them by removing an edge from G_p and inserting it again. Here, removing an edge refers to removing the edges of the path in the planarized representation that represents an edge of the original graph and undoing the split operations on other (crossed) edges; hence, this remove operation is simply the reverse operation of edge insertion.

The variant `INS` involves exactly those edges that have been deleted in the planar subgraph step, whereas the variants `ALL` and `MOST` involve the whole set of edges E in the original graph G . An iteration processes either the whole set (in variant `INS` and `ALL`) or $x\%$ of these edges (variant `MOST` $x\%$) iteratively one after the other. The procedure stops only if no improvement has been made within one iteration. The variant `MOST` $x\%$ considers the edges with the most crossings. After each iteration, we sort the edges in descending order according to the number of crossings they are involved in. Then, only the first $x\%$ edges of this list are taken for re-insertion. For the sake of completeness, the variant without any post-processing is called `NONE`.

An alternative approach combines the edge insertion with the postprocessing. Instead of performing the remove-reinsert strategy after all edges have been inserted, we can perform this strategy after each edge insertion. The idea behind this variation is to keep the number of crossings low as early as possible. For this variant, we always apply the remove-reinsert strategy to all original edges that are already contained in the graph. This alternative approach is denoted with `INC`. We remark, that this variant is not covered by the framework shown in Listing 4.3; in this case the postprocessing is performed within the `for`-loop for edge re-insertion, directly after edge $e_{\sigma(j)}$ was inserted.

Permutations. After a whole edge insertion step with chosen strategies for one-edge insertion and postprocessing, we get a certain number of crossings. Our permutation variant does nothing else but repeating the whole edge re-insertion

process in a randomized way and keeping the best result. The random effect exists in choosing a different ordering of the edges in $E \setminus E_p$ for the initial reinsertion step. The notation $\text{PERM}i$ gives the number of these repetition rounds. The parameter $n\text{Permutations}$ in the algorithm determines the number of permutation rounds.

4.3 Experimental Analysis

All algorithms have been implemented in the OGDF library [Chimani et al., 2010]. The crossing minimization framework is represented by the class `SubgraphPlanarizer`, which is a `CrossingMinimizationModule` in OGDF. The permutation scheme is integrated into the subgraph planarizer; particular implementations for the computation of the planar subgraph and for one-edge insertion can be set as module options, OGDF's mechanism to dynamically set and exchange implementations of modules with a specific functionality. We use the PQ-based planar subgraph algorithm (PQ1, PQ10, ...) realized by OGDF's `FastPlanarSubgraph` and `MaximalPlanarSubgraphSimple` for the simple implementation of a maximal planar subgraph (MAXIMAL). The one-edge insertion algorithms are realized by the classes `FixedEmbeddingInserter` (FIX) and `VariableEmbeddingInserter` (VAR); these classes also implement the postprocessing procedures. The class `NonPlanarCore` computes and represents the non-planar core preprocessing method. However, it is not yet fully integrated into the planarization method, and we simply first decompose the graph into its blocks before performing the crossing minimization if preprocessing is used.

Test Suite. For our experimental study, we use two benchmark sets of graphs: The *Rome graphs* (as introduced in Section 3.4.7) have already been used in the experimental study by Di Battista et al. [1997] for comparing graph drawing algorithms.

The *artificial graphs* are a new collection of non-planar graphs, consisting of special graph classes with known crossing numbers. This allows us to compare the quality of the heuristic solutions with the actual crossing numbers. It contains 1946 instances with up to 250 nodes, for which the number of edges and the number of crossings is linearly bounded by the number of vertices

In the following, we denote with C_n the cycle with n edges and with P_n the path with n edges. We distinguish four classes of graphs:

$C_m \times C_n$. For $3 \leq m \leq 7$ and $n \geq m$,

$$\text{cr}(C_m \times C_n) = (m - 2) * n \quad [\text{Adamsson and Richter, 2004}]$$

The benchmark set contains 251 of these graphs with $nm \leq 250$.

$G_i \times P_n$. Various results have been published for the crossing numbers of products of 5-vertex graphs G_i with paths P_n . Table 4.1 shows the results we have used. The benchmark set contains 893 graphs with $3 \leq n \leq 49$.




















i	G_i	$\text{cr}(G_i \times P_n)$	i	G_i	$\text{cr}(G_i \times P_n)$
2		$2(n-1)$ [Klešč, 1991]	13		$n-1$ [Klešč, 2001]
3		$n-1$ [Klešč, 2001]	14		$2(n-1)$ [Klešč, 2001]
4		$n-1$ [Klešč, 2001]	15		$3n-1$ [Klešč, 1995]
5		$n-1$ [Klešč, 2001]	16		$3n-1$ [Klešč, 1999a]
6		$2(n-1)$ [Klešč, 2001]	17		$2n$ [Klešč, 2001]
7		$n-1$ [Klešč, 2001]	18		$3n-1$ [Klešč, 1995]
9		$2(n-1)$ [Klešč, 2001]	19		$3n-1$ [Klešč, 2001]
10		$2n$ [Klešč, 1996]	20		$4n$ [Klešč, 2001]
11		$2(n-1)$ [Klešč, 2001]	21		$6n$ [Klešč, 1999b]
12		$2(n-1)$ [Klešč, 1995]			

Table 4.1: The crossing numbers of products of 5-vertex graphs with paths used in our benchmark set.

$\mathbf{G}_i \times \mathbf{C}_n$. Table 4.2 shows the crossing numbers of some products of 5-vertex graphs G_i with cycles C_n . The benchmark set contains 624 graphs with $3 \leq n \leq 50$.

$\mathbf{P}(\mathbf{m}, 2), \mathbf{P}(\mathbf{m}, 3)$. This class of graphs contains generalized Petersen graphs. Exoo et al. [1981] have shown that $P(2k, 2)$ is planar and

$$\begin{aligned} \text{cr}(P(5, 2)) &= 2 \\ \text{cr}(P(2k + 1, 2)) &= 3 \quad \text{for } k \geq 3 \end{aligned}$$

Richter and Salazar [2002] have shown that

$$\begin{aligned} \text{cr}(P(9, 3)) &= 2 \\ \text{cr}(P(3k, 3)) &= k \quad \text{for } k \geq 4 \\ \text{cr}(P(3k + 1, 3)) &= k + 3 \quad \text{for } k \geq 3 \\ \text{cr}(P(3k + 2, 3)) &= k + 2 \quad \text{for } k \geq 3 \end{aligned}$$

Our benchmark set contains the 61 graphs $P(2k + 1, 2)$ with $2 \leq k \leq 62$ and the 117 graphs $P(m, 3)$ with $9 \leq m \leq 125$.

System Configuration. We performed all test on a Linux SMP server machine with the following configuration:

<i>Operating system</i>	Debian Linux 2.6.16
<i>Compiler</i>	g++ 3.4.4
<i>CPU</i>	4 AMD Opteron 850 processors 2.4 GHz, 1 MB L2 Cache
<i>Memory</i>	32 GB RAM

Though this server machines has four CPUs, our implementations are single-threaded, that is, each test run was assigned to a single CPU.

4.3.1 Results for the Rome Graphs

Planar Subgraphs. As a preliminary experiment, we analyzed the effect of the different choices for the computation of a planar subgraph. We compared the maximal planar subgraph with the results of the PQ-based algorithm with 1, 10, 25, 50, and 100 iterations.

Figure 4.13(a) compares the size of the computed planar subgraphs in terms of number of deleted edges (we do not include the results of PQ50, since they were very close to PQ-100). We used the Rome graphs, grouped by their number of vertices, and give the average value for each group. The results of the PQ-tree based heuristics improve significantly as the number of iterations increases. While the number of deleted edges was about 18 on average for graphs














i	G_i	$cr(G_i \times C_n)$	i	G_i	$cr(G_i \times C_n)$
2		$\begin{cases} 2 & n=3 \\ 4 & n=4 \\ 8 & n=5 \\ 2n & n \geq 6 \end{cases}$ [Klešč, 1991]	9		$2n$ [Klešč, 2005]
3		$\begin{cases} 1 & n=3 \\ 2 & n=4 \\ 4 & n=5 \\ 2n & n \geq 6 \end{cases}$ [Klešč, 2005]	11		$\begin{cases} 7 & n=3 \\ 3n & n \geq 4 \end{cases}$ [Klešč and Kocúrová, 2007]
4		n [Klešč, 2005]	12		$2n$ [Klešč, 2005]
5		n [Klešč, 2005]	13		$\begin{cases} 7 & n=3 \\ 3n & n \geq 4 \end{cases}$ [Klešč, 2005]
6		$\begin{cases} 4 & n=3 \\ 6 & n=4 \\ 9 & n=5 \\ 2n & n \geq 6 \end{cases}$ [Klešč, 2005]	14		$3n$ [Klešč, 2005]
7		$\begin{cases} 4 & n=3 \\ 2n & n \geq 4 \end{cases}$ [Klešč, 2005]	16		$\begin{cases} 3n & n \text{ even} \\ 3n+1 & n \text{ odd} \end{cases}$ [Klešč, 1999a]
8		$\begin{cases} 5 & n=3 \\ 10 & n=4 \\ 3n & n \geq 5 \end{cases}$ [Ringelsen and Beineke, 1978] [Beineke and Ringelsen, 1980] [Klešč et al., 1996]			

Table 4.2: The crossing numbers of products of 5-vertex graphs with cycles used in our benchmark set.

with 100 vertices for PQ1, it was only about 15.9 for PQ10, and went down to 14.7 for PQ100. We also observe that 10 iterations or more are sufficient to outperform the MAXIMAL algorithm, which yields on average about 16.6 deleted edges for graphs with 100 vertices.

Obviously, it really pays off to run the PQ-tree based planar subgraph algorithm many times. However, for example, the results for 50 and 100 iterations are already very close to each other. This effect also comes from the random effect chosen for our implementation; by choosing other randomization techniques, this effect may occur at a higher number of iterations.

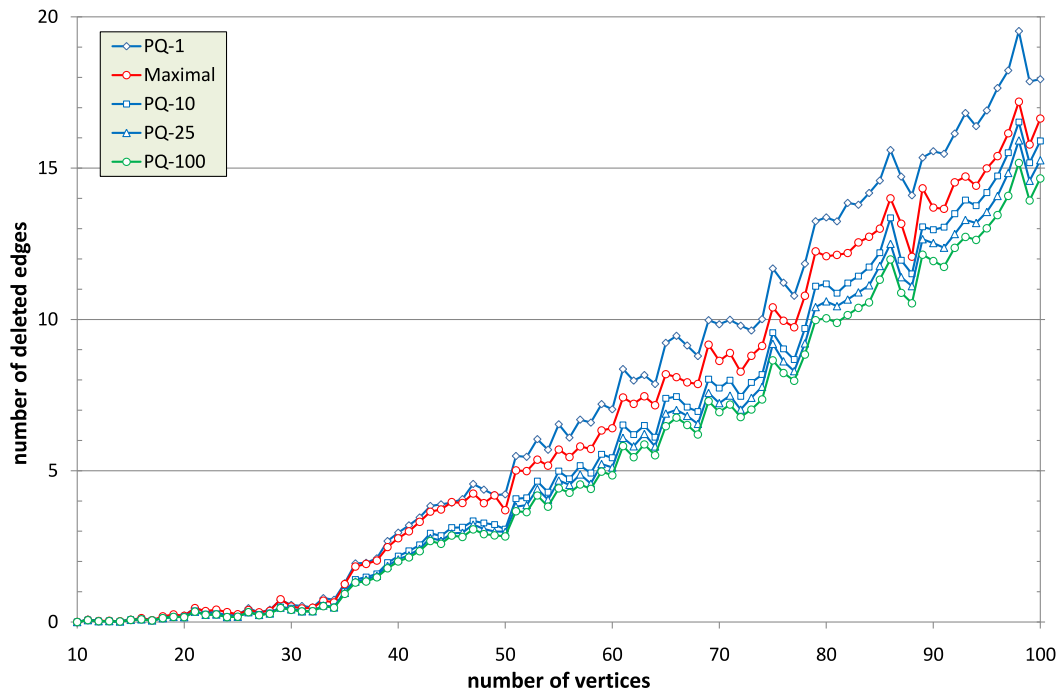
Figure 4.13(b) shows the running times of our implementations. The PQ variant requires only 0.4 milliseconds on average for graphs with 100 vertices when using just a single iteration, increasing to about 32 milliseconds for 100 iterations. In comparison, the MAXIMAL variant requires about 17 milliseconds. Considering the quality of the planar subgraphs and the fact, that the running times are quite low compared to the expected running times for the whole crossing minimization step, PQ100 seems to be a good choice for our further experiments.

To verify this assumption, we ran crossing minimization experiments with the FIX and VAR variant, without any pre- or postprocessing; see Figure 4.14. To our surprise, the MAXIMAL algorithm yields about the same number of crossings than PQ1. On the other hand, the better planar subgraphs obtained for higher number of iterations with the PQ-based algorithm also lead to less crossings. Due to the better one-edge insertion algorithm, the effect is a little bit smaller for the VAR variant. It is interesting to see that the maximality property of the planar subgraph does not seem to have any positive influence on the crossing minimization.

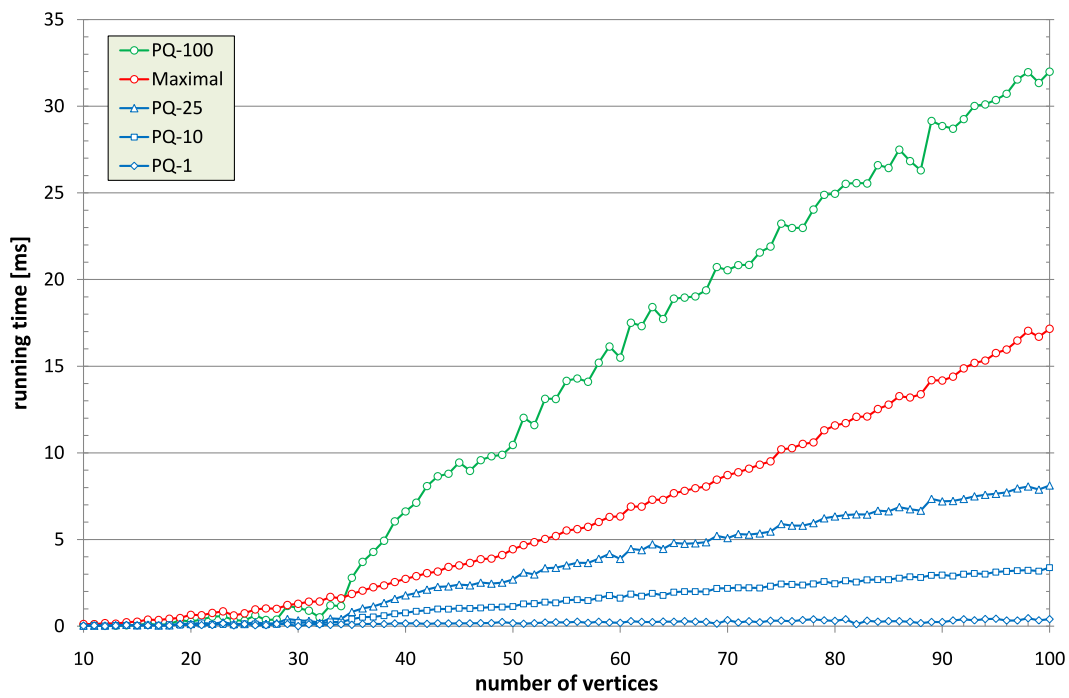
Since a better planar subgraph also means that we have to insert a fewer number of edges, we also analyzed if the better planar subgraph speeds up the edge insertion phase. We compared the overall runtime of the crossing minimization, applying edge insertion with postprocessing but without permutations, for PQ1 and PQ100. While the FIX strategies were always faster with PQ1 even when applying the ALL and INC postprocessing variant (this might change when using many permutations), we achieved a good speedup for the VAR strategy. Figure 4.15 shows the speedup (runtime with PQ1 divided by runtime with PQ100) for the postprocessing variants MOST10, MOST25, ALL, and INC. We get a speedup greater than one when using postprocessing for 25% of the edges or more (at least for the larger graphs). For the INC variant, we get a speedup factor of almost two and for the ALL variant even a factor of up to three.

In the rest of this study, we will mainly use the PQ100 variant for computing planar subgraphs.

FIX vs. VAR. Figure 4.16 compares the two one-edge insertion variants FIX and VAR without pre- or postprocessing. In Figure 4.16(a), we show the average number of crossings for PQ1 and PQ100 from Figure 4.14 in a single diagram. We observe that the influence of the planar subgraph is quite big, yielding better results for PQ100-FIX than PQ1-VAR. However, using the same planar subgraph we

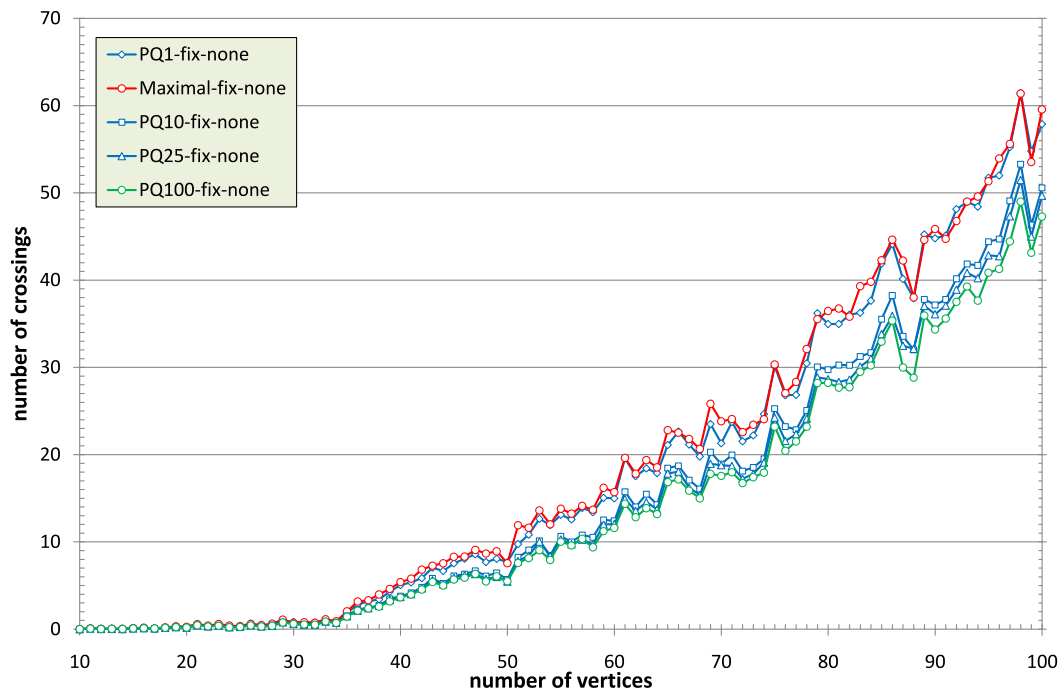


(a) number of deleted edges

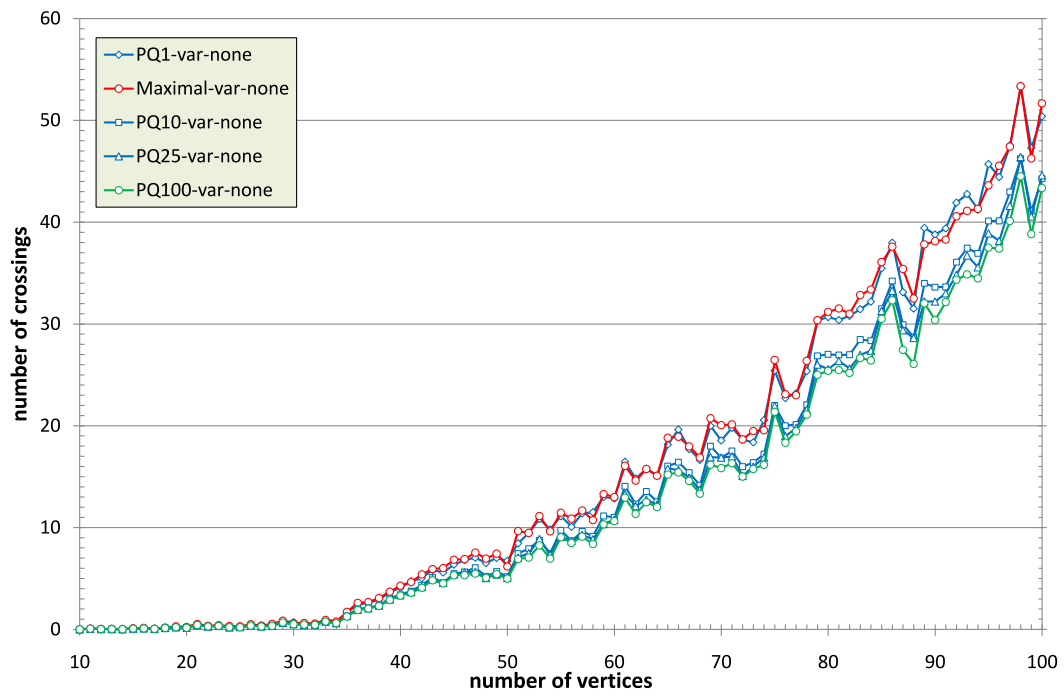


(b) running times in milliseconds

Figure 4.13: Comparison of planar subgraph heuristics (Rome graphs).



(a) fixed embedding



(b) variable embedding

Figure 4.14: Effect of the planar subgraph on the number of crossings (Rome graphs).

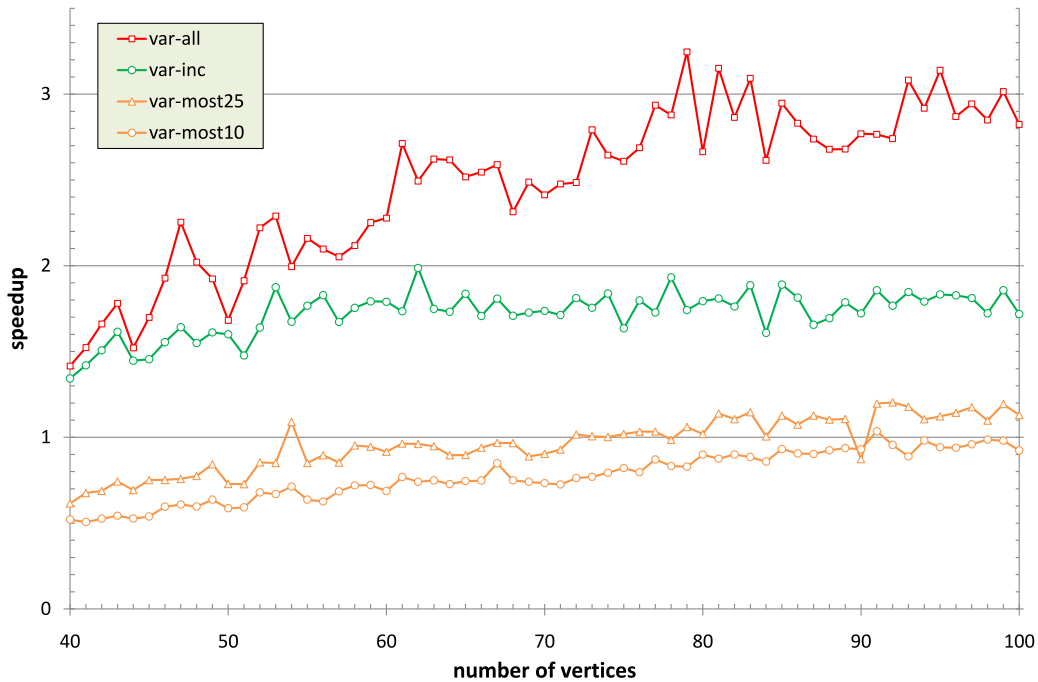


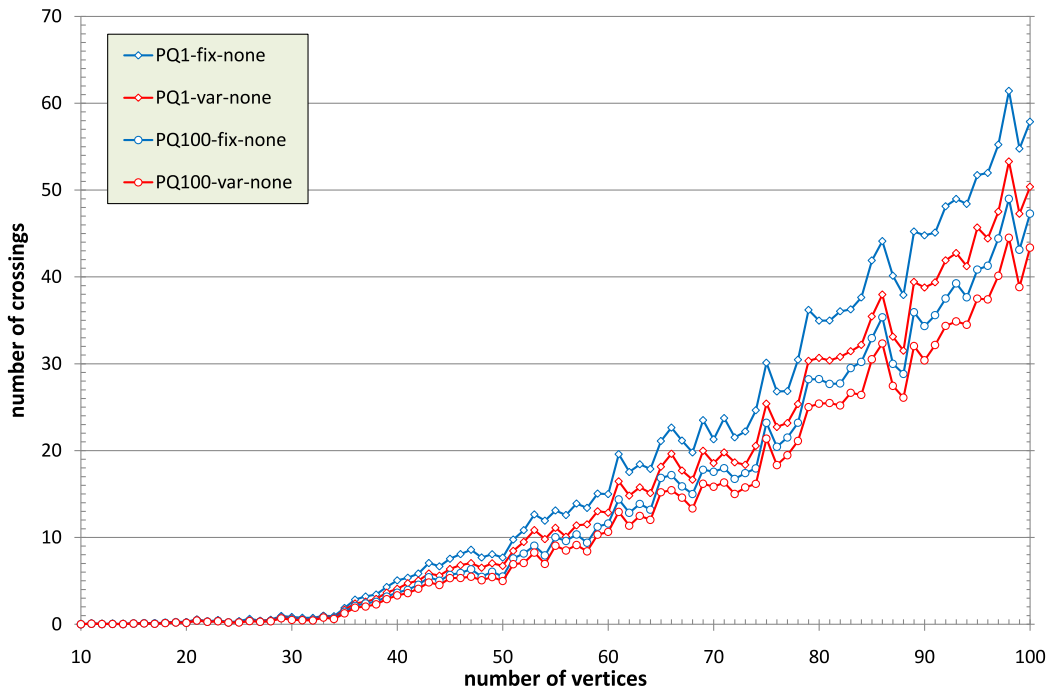
Figure 4.15: Speedup of PQ100 vs. PQ1 for VAR strategy (Rome graphs).

get significantly better results when using the VAR variant. This improvement is shown Figure 4.16(b), which gives the relative improvement of VAR in percent (average number of crossings with $\text{FIX} = 100\%$). VAR improves on average by roughly 9.5% when using PQ100, and even 14% when using the worse planar subgraphs obtained by PQ1.

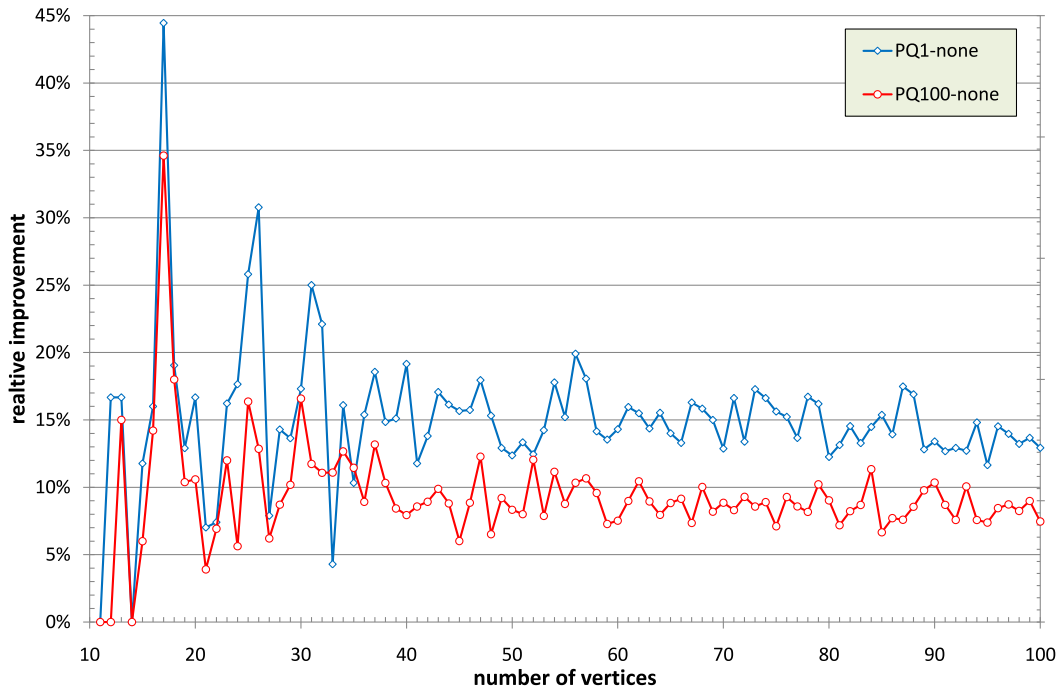
For the instances with 100 vertices, we get 57.86 crossings on average with PQ1-FIX. This is even better than the numbers reported in the study by Di Battista et al. [Di Battista et al., 1997]. The difference between their and our implementation is in the algorithm for computing the planar subgraph. They used a heuristic based on the Hopcroft and Tarjan planarity test [Nardelli and Talamo, 1984]. In the sequel, we call PQ1-FIX-NONE-PERM1 the *standard method* and report on the improvement over this method.

By computing a better planar subgraph the number of crossings reduces from 57.86 for the standard method to 47.28 (about 18.3% improvement). This number can further be reduced to 43.36 by choosing the best embedding for each inserted edge (variant PQ100-VAR), resulting in an improvement of about 25% compared to the standard method.

Postprocessing. Figure 4.17 shows the effect of the postprocessing variants for the FIX and VAR strategy compared to the standard method. The relative improvement compared to no postprocessing at all (PQ100-FIX-NONE or PQ100-VAR-NONE) is shown in Figure 4.18. Since the smaller graphs in the Rome library contain only very few non-planar graphs, we restrict us to present the results for



(a) average number of crossings



(b) average relative improvement

Figure 4.16: FIX vs. VAR (Rome graphs).

graphs with 40 vertices and more in the diagrams for improving readability and clarity; especially when showing relative improvements, the plots look otherwise unnecessarily confusing for the smallest graphs.

We observe that taking the inserted edges as candidates for postprocessing is already much better than the `NONE` strategy. However, the results can be improved a lot more by taking the whole set of edges into account. We observe that already considering 25% of the edges with the most crossings gives similar results to the variant `ALL` which considers all edges. `ALL` and `MOST25` improve the number of crossings up to about 40% (about 22 crossings at 100 vertices) for `FIX` compared to the standard method, and about 42% (about 23.75 crossings at 100 vertices) for `VAR`. Comparing the various strategies with `NONE`, we see that the improvement is a bit more for `FIX`. Moreover, `MOST25` even outperforms `ALL` for `VAR`. The reason for this could be the implementation of `MOST25`, which sorts the edges first by descending number of crossings, whereas `ALL` simply uses the given order. The clear winner is the `INC` variant, showing that the higher effort really pays off and can improve the number of crossings substantially.

Figure 4.19 shows the running times of the postprocessing variants. The pictures look a bit different for `FIX` and `VAR`. For `FIX`, `INS` and `MOST10` have about the same running time, as well as `ALL` and `MOST25`. The `INC` variant requires—as could be expected—more effort, but needs still only about double the time as for the simplest variants. Altogether, all algorithms are fast, with average running times below 0.07 seconds per graph.

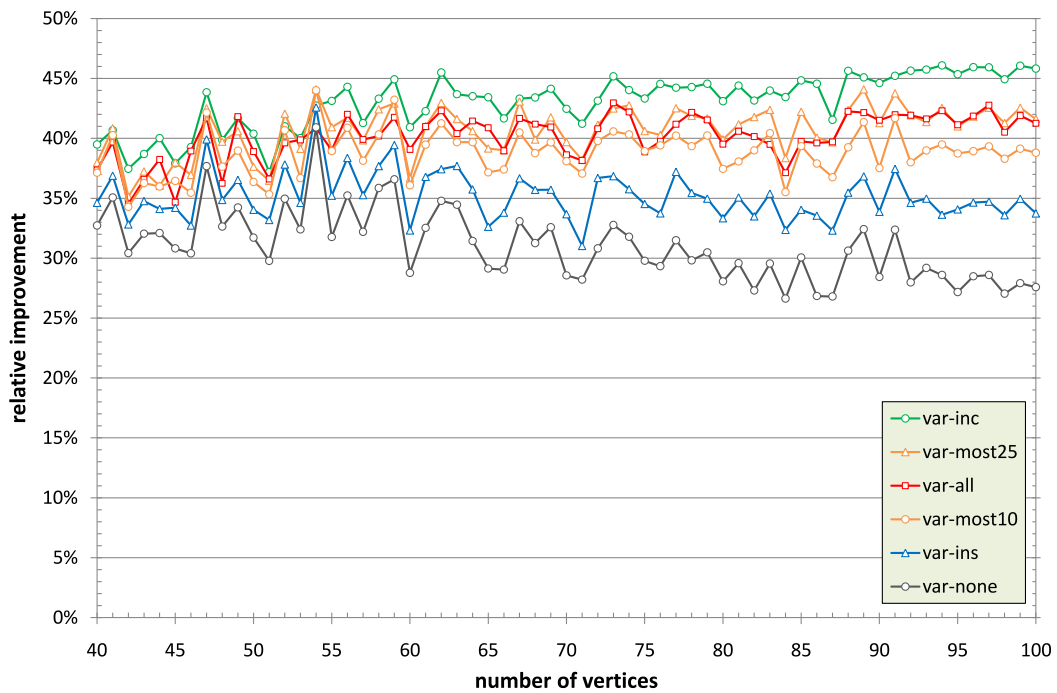
For the `VAR` strategy, the differences are much higher, especially the `INC` variant is much slower compared to the other postprocessing options. The reason is that we do not have an incremental update procedure implemented for `VAR`. Such a procedure would be much more complicated than for the fixed embedding case, requiring incremental updates for BC- and SPQR-trees. In particular, we would need updates for removing edges, which is, for example, not covered by the online planarity testing algorithm by Di Battista and Tamassia [Di Battista and Tamassia, 1996a]. However, even only practically efficient updates could improve the running times significantly. With running times below 0.3 seconds on average, the `MOST10`, `MOST25` and `ALL` variants are quite fast, and even `INC` with running times below 0.7 seconds is still fast enough for practical applications requiring immediate response.

Preprocessing. The purpose of applying the non-planar core reduction as preprocessing is to reduce the size of the input graph. We found that the structure of the Rome graphs is such that a non-planar graph always has only a single non-planar block. Furthermore, after decomposing this block into its triconnected components, the non-planar core is simply the skeleton of an R-node of its SPQR-tree. However, this seems to be merely a special property of the Rome graphs.

Figure 4.20 shows the number of edges in the non-planar core in percent of the number of edges in the original graph. We give the average value as well as the minimum and maximum for each graph group with the same number of vertices. A minimum of 0% means that the respective graph group contains at least one



(a) fixed embedding

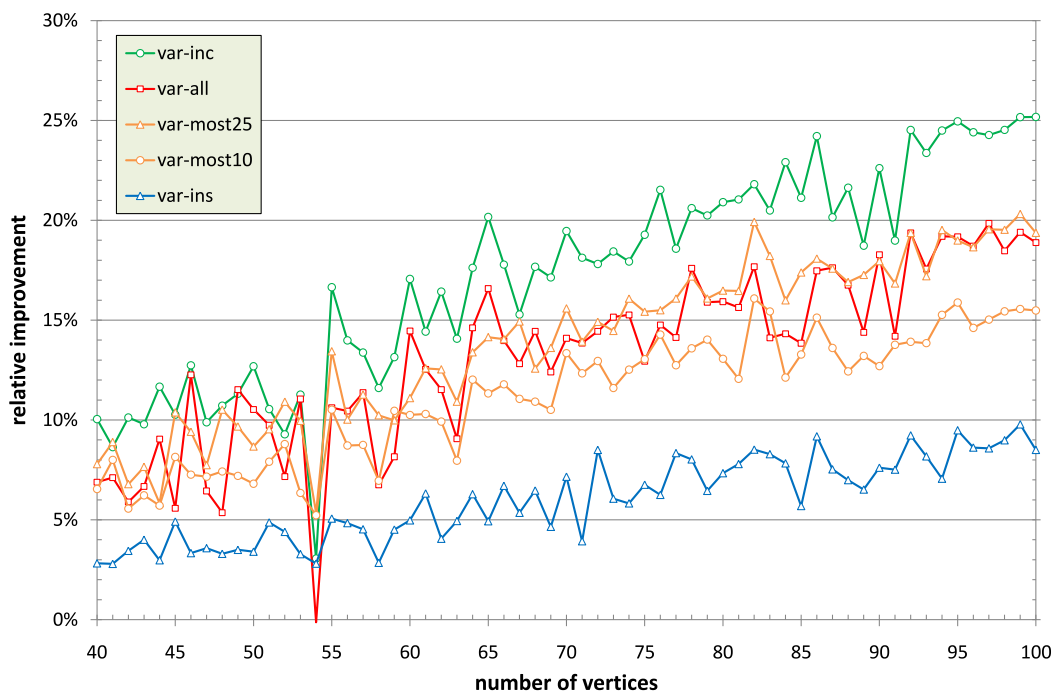


(b) variable embedding

Figure 4.17: Relative improvement of postprocessing strategies vs. standard method.

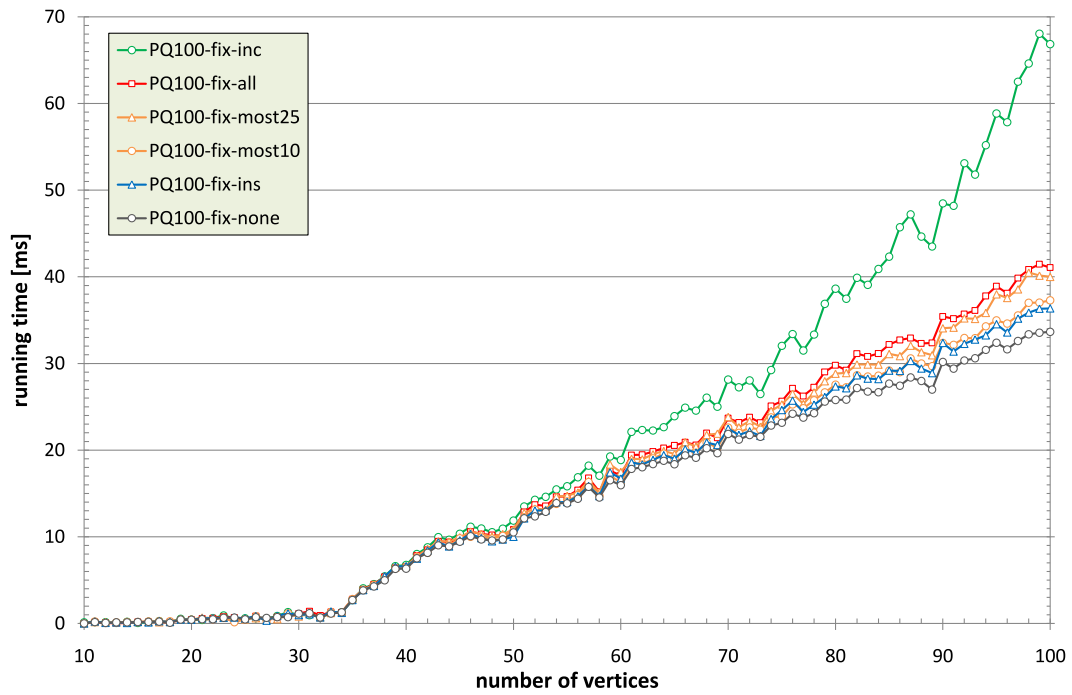


(a) fixed embedding

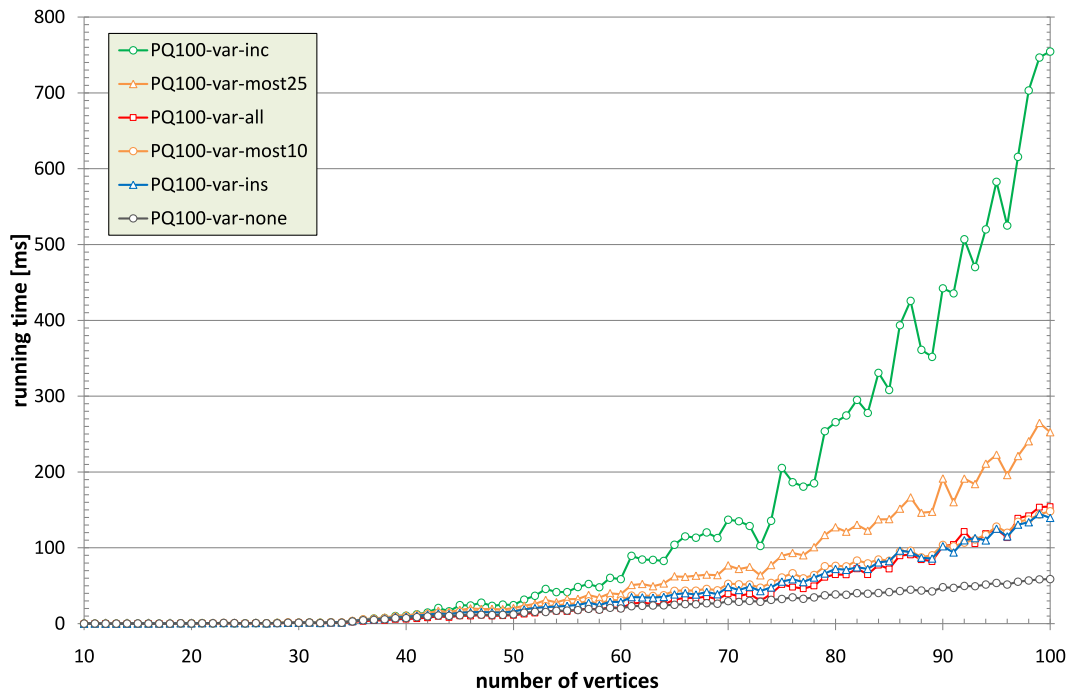


(b) variable embedding

Figure 4.18: Relative improvement of postprocessing strategies vs. PQ100-NONE.



(a) fixed embedding



(b) variable embedding

Figure 4.19: Running times of postprocessing strategies (Rome graphs).

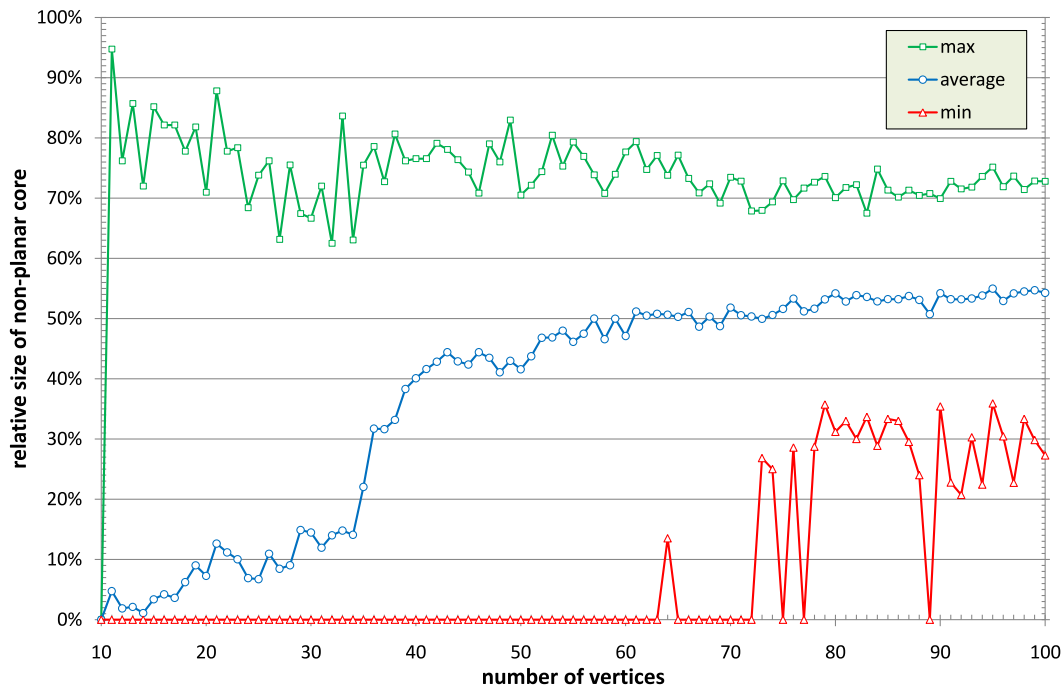


Figure 4.20: Relative average size of the core (Rome graphs).

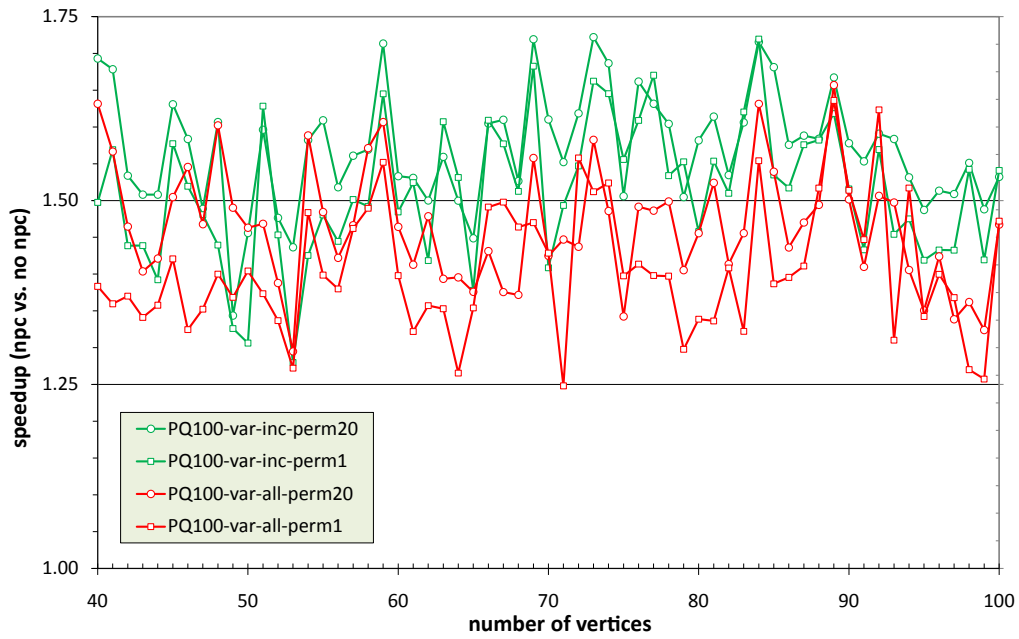
planar graph. On average, the non-planar core contains only half as many edges as the original graph, at least for the larger graphs.

Using the non-planar core reduction as preprocessing requires some implementation changes: Firstly, we need to decompose the graph into its blocks and handle each block separately; secondly, we need to construct the SPQR-tree for each block in order to obtain the non-planar core. And finally, we need to adjust the edge insertion algorithms so that they can handle graphs with positive integer weights on the edges.

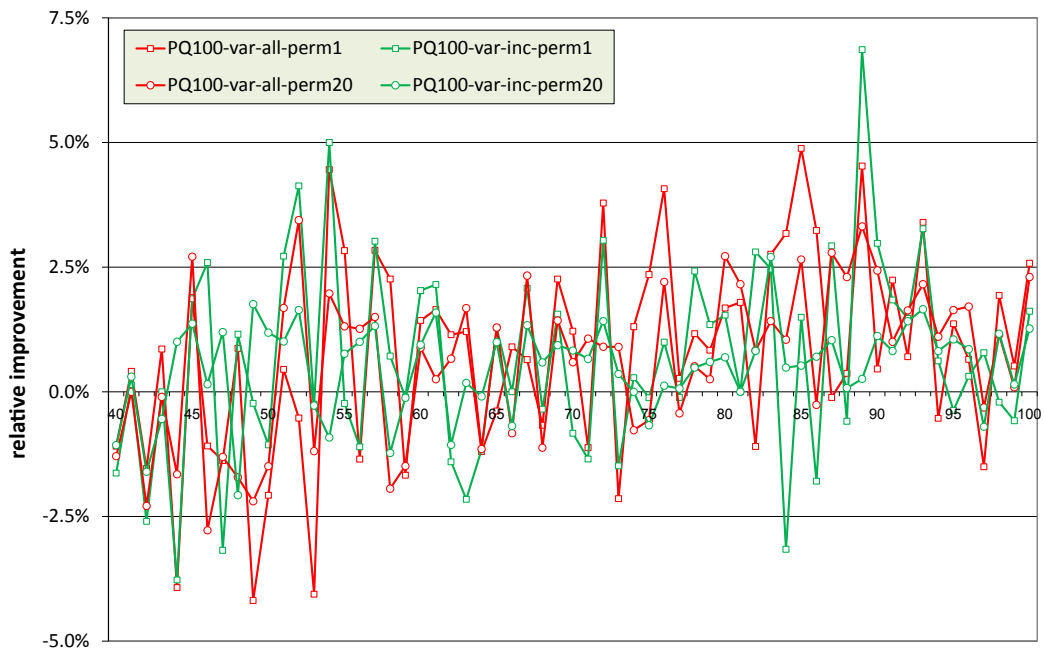
It turns out that these changes increase the running time of the heuristics with fixed embedding a lot, raising to almost the runtime of the heuristics based on variable embedding. Though the number of crossings obtained with preprocessing improves by about 4–5% on average, preprocessing is not a useful option for fixed embedding due to long running times.

On the other hand, preprocessing works very effective for the variable case, improving the running time by roughly a factor of 1.5. Figure 4.21(a) shows the speedup with the ALL and INC postprocessing options and 1 and 20 permutations, respectively. The resulting numbers of crossings get also a slight improvement (see Figure 4.21(b)), but by far not as much as in the fixed embedding case.

Permutations. Figure 4.22 shows the effect of the permutation variants for FIX and VAR. As expected, the results improve with higher permutation numbers. However, we observe that the postprocessing effects are stronger than the PERM20 effects: the FIX-ALL-PERM1 heuristic leads to much better results than the FIX-NONE-PERM20; the same holds for the VAR variants. Even FIX-MOST10-PERM1 and



(a) average speedup



(b) average improvement for number of crossings

Figure 4.21: Effect of preprocessing on VAR strategy (Rome graphs).

VAR-MOST10-PERM1 perform significantly better than FIX-NONE-PERM20 and VAR-NONE-PERM20, respectively. Obviously, the running times increase a lot, usually by the factor of performed permutations. Therefore, it is reasonable to apply more permutations only in conjunction with postprocessing methods.

An interesting question that comes up is how much improvement we can expect from permutations at all. The results shown in Figure 4.22 suggest that a few permutations have a big effect, but higher numbers of permutations show only small improvements. Therefore, we studied the effect of up to 200 permutations for the graphs with 40, 60, 80, and 100 vertices; see Figures 4.23 and 4.24 showing real crossing numbers this time. We considered the edge insertion variants fixed and variable with postprocessing (ALL and INC) and optional preprocessing. The picture looks in all cases very similar: the curves for each heuristic run more or less in parallel, showing that the effect of permutations is very similar for each heuristic. All curves seem to approach a minimum value, but constantly decrease until 200 permutations.

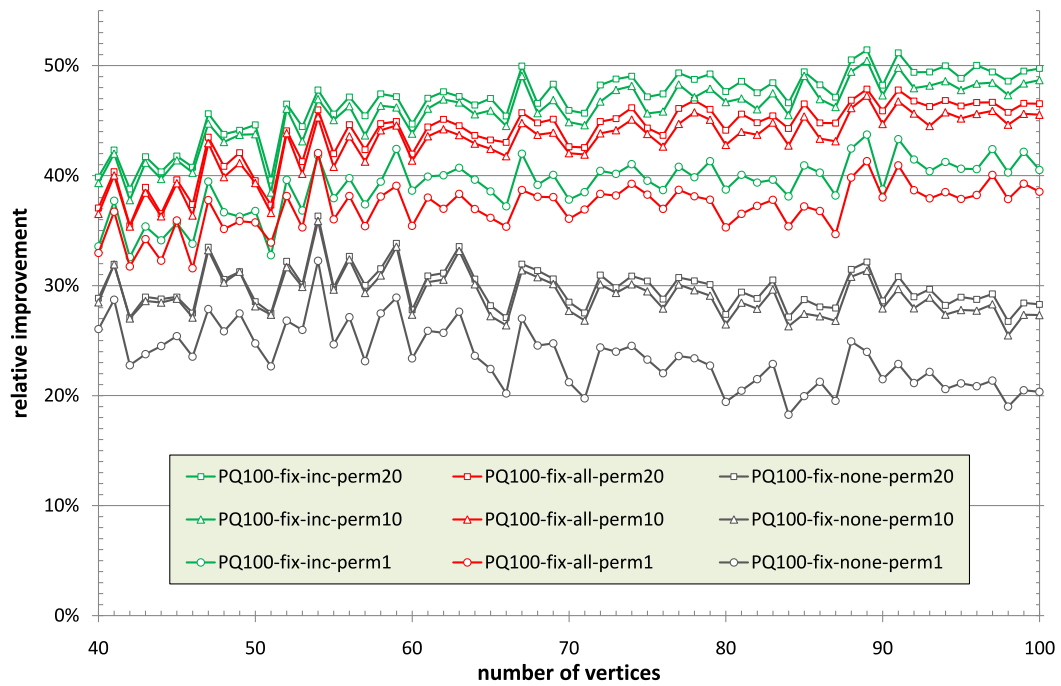
In order to study this effect in more detail, we increased the number of permutations to 500 for the graphs with 100 vertices; see Figure 4.25. The number of crossings still decreases slightly, for example, for the VAR-INC heuristic, we have 26.71 crossings at 20 permutations, 25.74 at 200 permutations, and 25.51 at 500 permutations on average. However, we do not know if this number is already close to the optimum value; we will study this in the following, where we first compare our results with the results of an exact method and then apply our crossing minimization heuristics to graphs with known crossing numbers.

Comparison with Exact Results. Obviously, the best way to evaluate the quality of heuristics is to compare their results with the optimal solutions. Though not all crossing numbers of the Rome graphs are known, many optimal or near-optimal solutions have been reported by Chimani [2008], obtained using exact branch-and-cut methods. We compare with some selected results for large graphs.¹ In particular, we consider the following four sets of graphs:

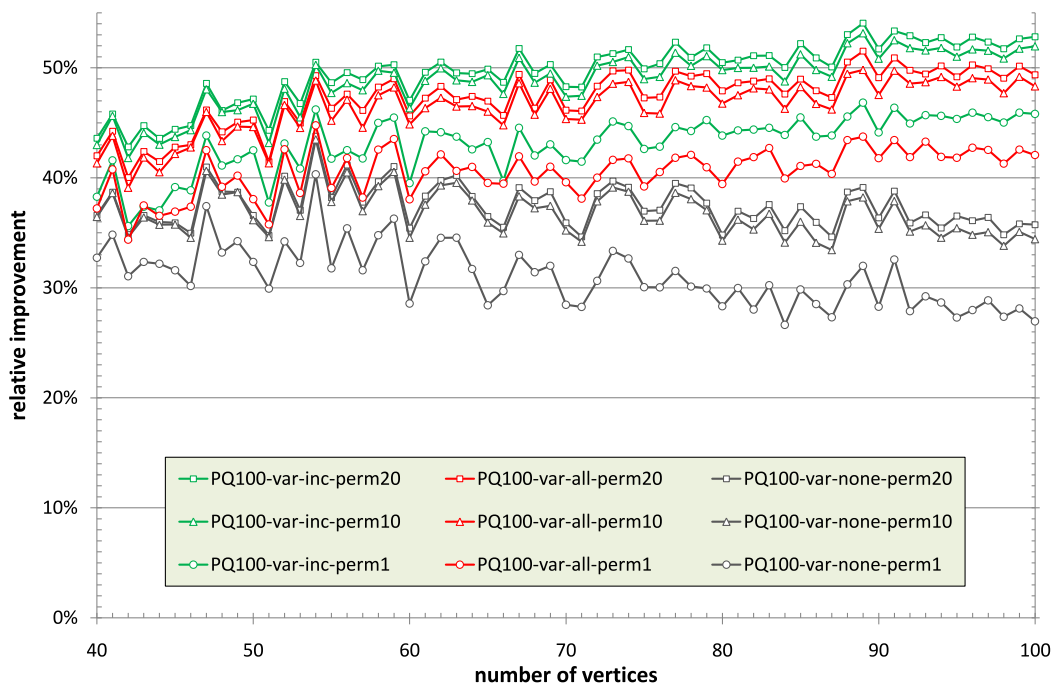
- *Graphs with 60 vertices:* All but one of the 131 graphs could be solved to optimality.
- *Graphs with 70 vertices:* All but four of the 81 graphs could be solved to optimality.
- *Graphs with 73 vertices:* This is the vertex set with the largest graphs that could completely be solved to optimality; it contains 85 graphs.
- *Graphs with 100 vertices:* From the 141 largest graphs, 76 (54.29%) could be solved to optimality.

For graphs that could not be solved to optimality, we used the best upper bound obtained by the exact method. We compared with the best results we have for

¹We thank Markus Chimani for providing the data.

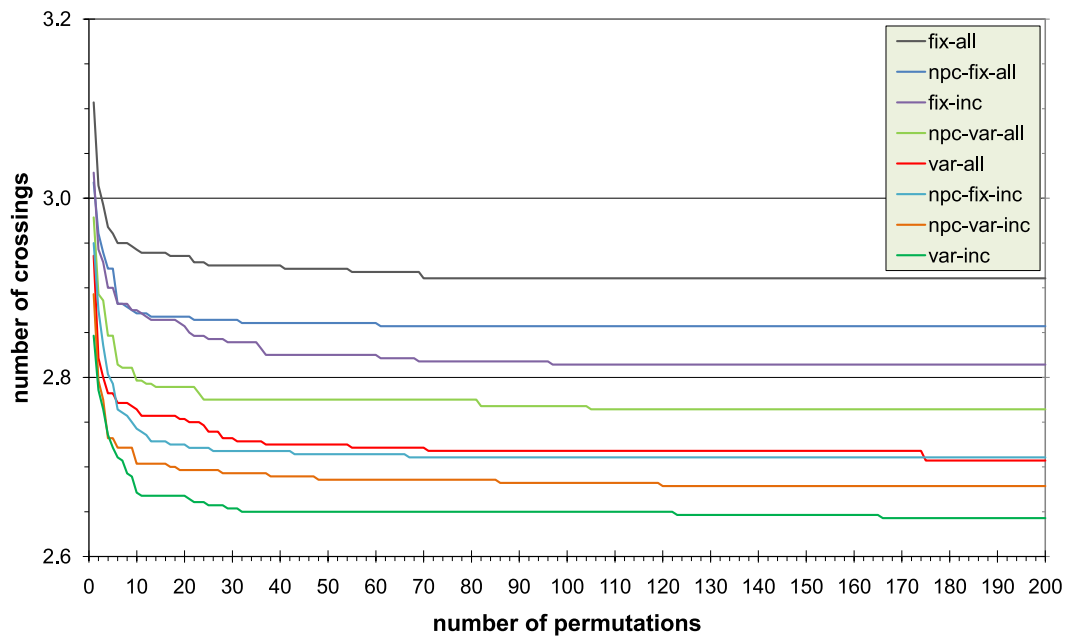


(a) fixed embedding

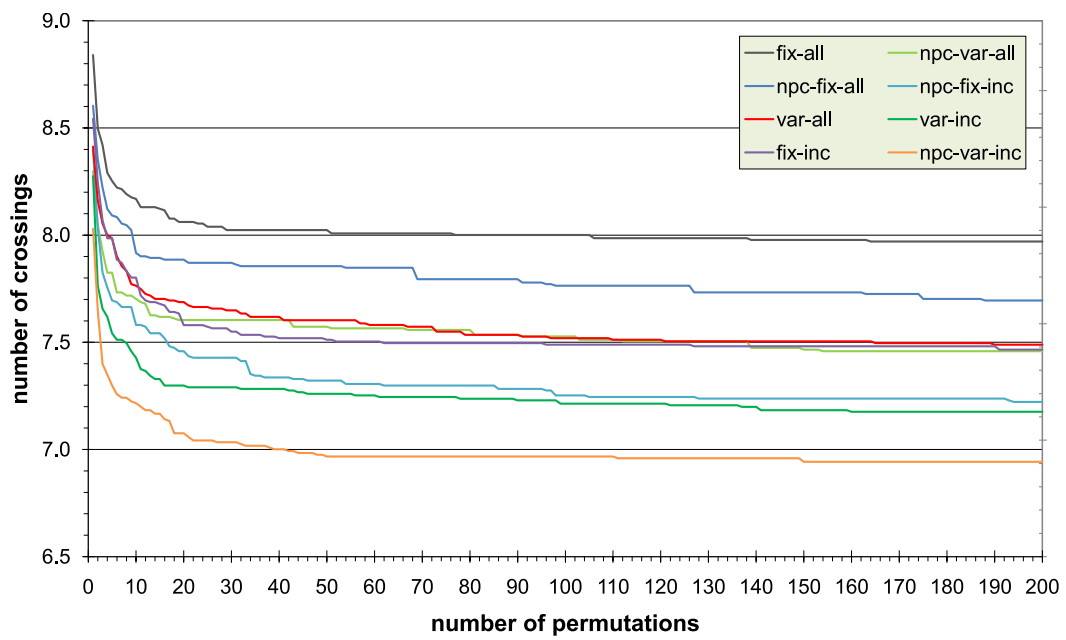


(b) variable embedding

Figure 4.22: Relative improvement of permutations vs. standard method (Rome graphs).

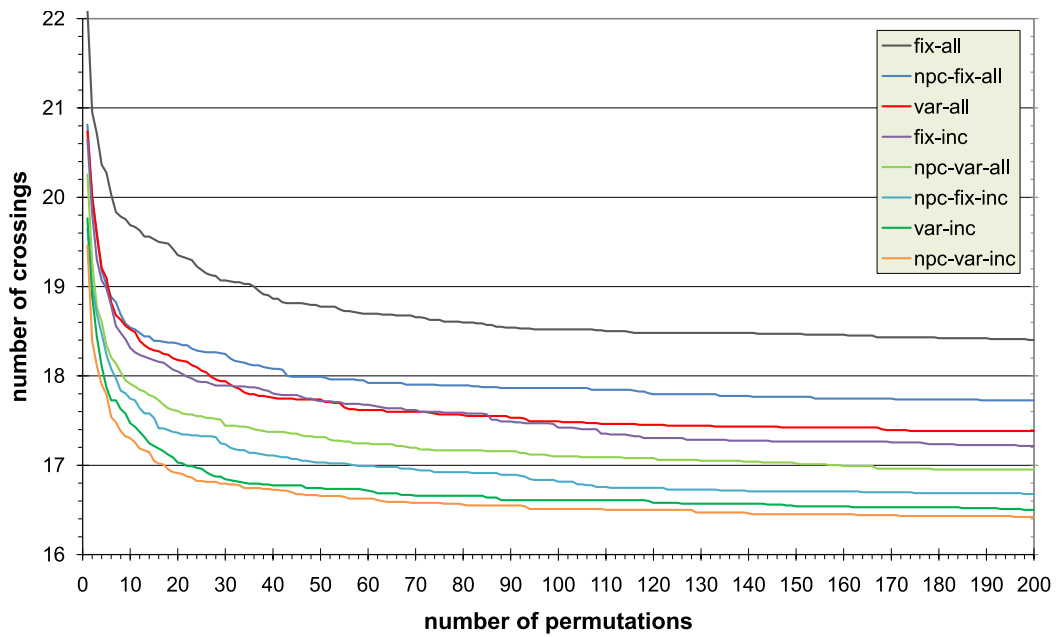


(a) 40 vertices

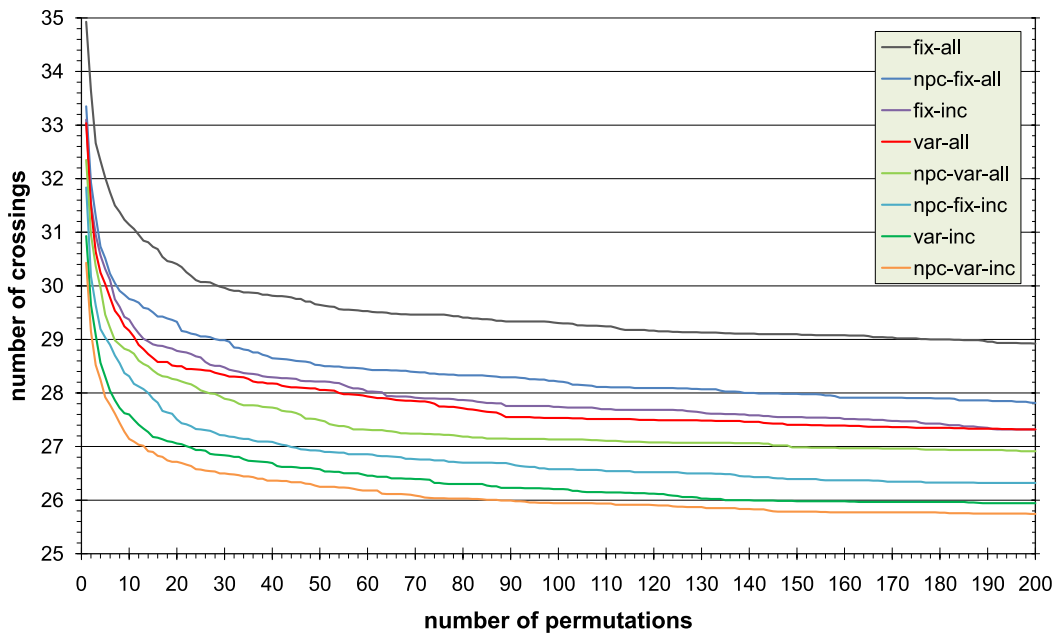


(b) 60 vertices

Figure 4.23: Effect of increasing number of permutations (Rome graphs).



(a) 80 vertices



(b) 100 vertices

Figure 4.24: Effect of increasing number of permutations (Rome graphs).

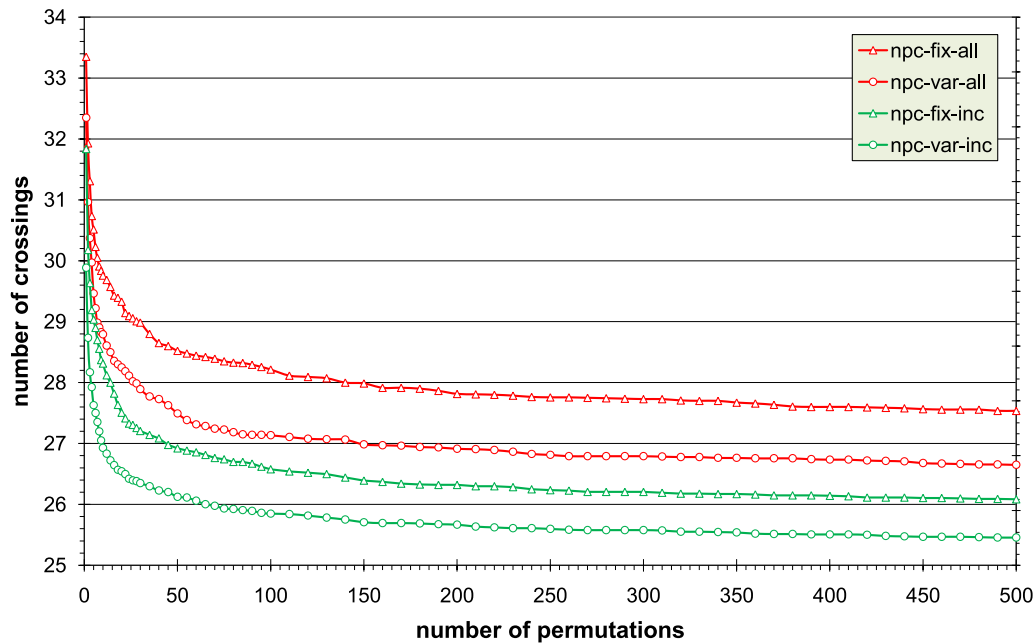


Figure 4.25: The effect of up to 500 permutations for Rome graphs with 100 vertices.

each graph group, that is, preprocessing with variable embedding and the post-processing variants ALL and INC, using 200 and 500 permutations.

Figure 4.26 shows the average number of crossings obtained by the various methods. We can see that the heuristic solutions are already very close to the exact results: For 60 vertices, the best INC variant requires 6.992 crossings on average, only 0.069 (0.99%) crossings more than the exact method. For 70 vertices, the results are almost as good with the INC variant requiring 0.198 (1.76%) crossings more than the exact method; however, here the exact method could not solve as many graphs as for 60 vertices. Unlike the graphs with 70 vertices, the exact method could solve all instances at 73 vertices. Here, the INC variant yields only 0.106 (1.14%) more crossings than the optimal solution. Hence, for the graphs where we know (almost) all crossing numbers, the best heuristics are only 1–2% away from the optimal values.

Since the exact method could only solve about half of the graphs with 100 vertices, two of the heuristics are even better than the results obtained with the exact method. The INC variant with 500 permutations is 0.593 (2.27%) crossings better than the exact method. In order to get a better idea of the optimal values, we combined the best heuristic and exact results, which gives 6.90 crossings for 60 vertices, 11.22 for 70 vertices, and 25.27 for 100 vertices on average.

The numbers above show that the heuristic results are very good on average, but this does not state that there are not some bad solutions. Therefore, we grouped the graphs by the difference of the number of crossings for the best heuristic solution minus the exact solution. Figure 4.27 shows the distribution of the graphs. The worst value was three crossings more, which occurred for only

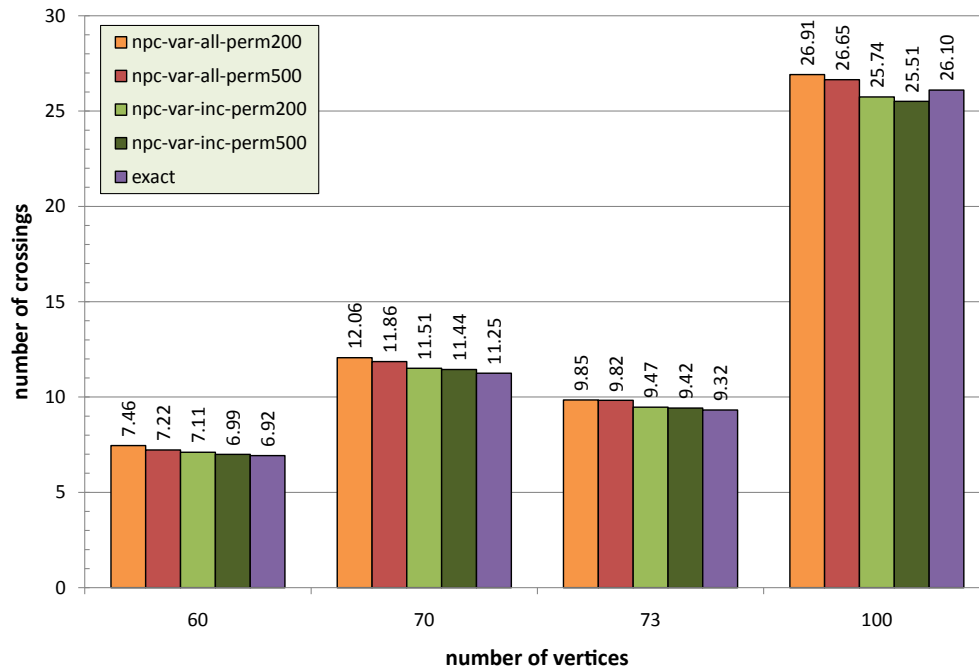


Figure 4.26: Average number of crossings of heuristics compared with exact method.

0.71% of the graphs with 100 vertices; all other solutions are at most two crossing worse than the exact solutions. On the other hand, we observe that some heuristic results are significantly better than the exact results: for two graphs with 100 vertices we achieved even 8 crossings less, and for two other graphs with 100 vertices 6 crossings less; we could even save one crossing for a graph with 70 vertices and 3 for two graphs with 60 vertices.

As a side note, we discuss shortly the relevance of the heuristic methods for the exact approach. Though our heuristic is used as primal heuristic in the exact method, some of our results are significantly better; the reason is mainly the increased number of permutations in our study. Since the primal heuristic is important for the exact method—the better the primal upper bound the easier it is for the branch-and-cut approach to prove or achieve optimality—, our new results could also have a positive impact on the exact method, leading to optimal solutions for even more graphs.

Summary. We have compiled the best heuristics in our study ranked by their performance on the Rome graphs with 100 vertices. Table 4.3 contains the average number of crossings and average running times. It is sorted by increasing number of crossings and decreasing running time; heuristics with higher number of crossings *and* higher running time have been omitted. We observe a few clear trends: The INC postprocessing variant is highly important for crossing reduction. As expected, the VAR strategy is required for achieving the best results and the FIX strategy with postprocessing is useful for obtaining good results with

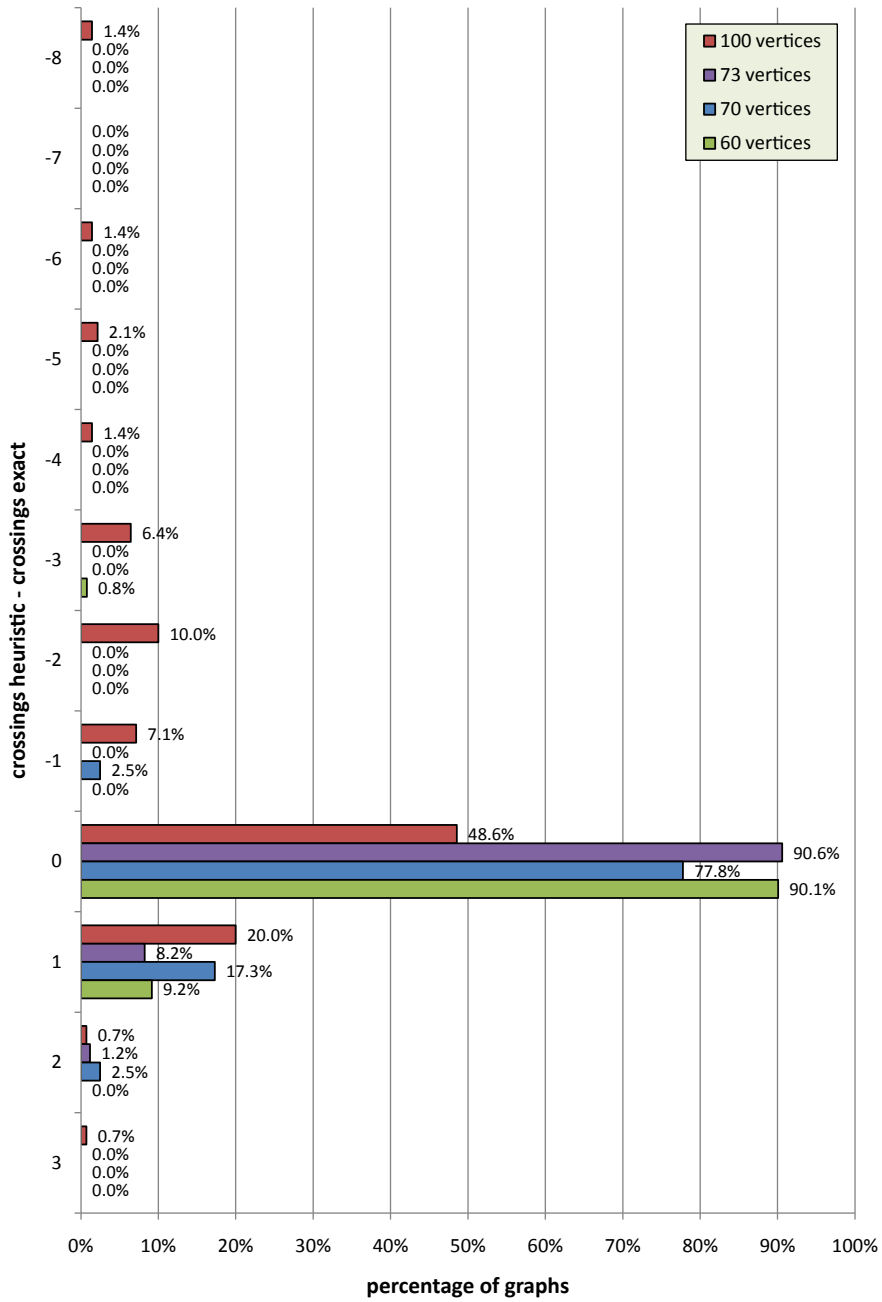


Figure 4.27: Distribution of graphs grouped by difference between best heuristic solutions and exact results.

rank	crossings	time [s]	EI	PRE	POST	PERM
1	25.51		VAR	NPC	INC	500
2	25.74		VAR	NPC	INC	200
3	25,94		VAR		INC	200
4	26.09		FIX	NPC	INC	500
5	26.32		FIX	NPC	INC	200
6	26.65		VAR	NPC	ALL	500
7	26.71	9.387	VAR	NPC	INC	20
8	27.14	4.681	VAR	NPC	INC	10
9	28.49	1.857	VAR	NPC	ALL	20
10	28.69	0.727	FIX		INC	20
11	30.43	0.490	VAR	NPC	INC	1
12	30.52	0.221	FIX		ALL	20
13	32.66	0.105	VAR	NPC	ALL	1
14	33.33	0.098	FIX	NPC	ALL	1
15	33.96	0.067	FIX		INC	1
16	35.09	0.041	FIX		ALL	1
17	35.79	0.040	FIX		MOST25	1
18	38.38	0.037	FIX		MOST10	1
19	41.61	0.036	FIX		INS	1
20	45.47	0.034	FIX		NONE	1

Table 4.3: The ranking list of the *best* heuristics.

short running time.

We discuss the findings of our study in more detail in Section 4.3.3 at the end of this chapter, giving hints for effective usage of crossing minimization heuristics.

4.3.2 Results of the Artificial Graphs

In order to compare the performance of the heuristics with the optimal solutions, we used our new benchmark set of artificial graphs with known crossing numbers. Since we are mainly interested in the quality of our heuristic solutions, we focus on the best heuristics and give the results of the standard heuristic FIX-NONE-PERM1 for comparison. We used the FIX and VAR strategy with the best postprocessing variants ALL and INC and up to 30 permutations; we do not use preprocessing here, since the structure of these artificial graphs is such that preprocessing is very ineffective.

Figure 4.28 shows the average number of crossings achieved by the heuris-

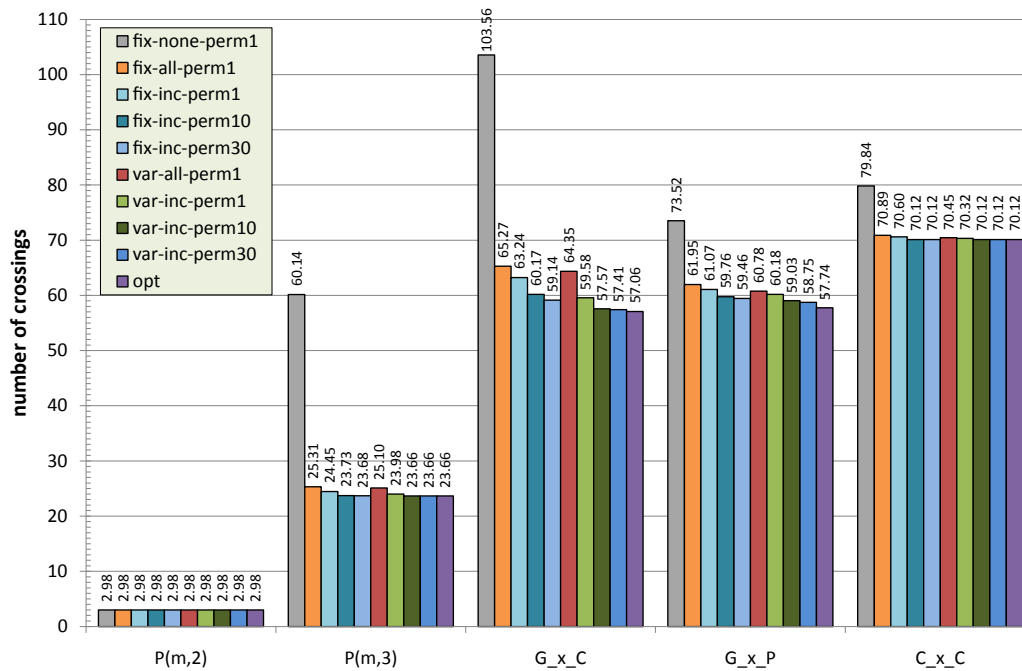


Figure 4.28: Average number of crossings for graphs with known crossing numbers.

tics as well as the optimal solutions, grouped by the different types of graphs in our benchmark set. The graphs $P(m,2)$ turn out to be easy and are already solved optimally by the simple heuristic FIX-NONE-PERM1. Unlike these graphs, the $P(m,3)$ graphs are more difficult, showing a drastic improvement using more advanced heuristics. However, the optimal solutions are only achieved by VAR-INC-PERM10 and VAR-INC-PERM30. The graphs $G_i \times C_n$ could not completely be solved by the heuristics, but the best heuristics are close to the optimum; VAR-INC-PERM30 achieved 57.41 crossings on average compared to the optimal value of 57.06. For the $G_i \times P_n$ graphs, the improvement compared to the simple heuristic is significantly smaller and the gap to the optimum is a bit larger, but still close, with 58.75 crossings for VAR-INC-PERM10 compared to the optimum of 57.74. The graphs $C_m \times C_n$ are easier to solve for the heuristics: All advanced heuristics are very close, and the VAR-INC and FIX-INC variants even achieve the optimum when applying at least 10 permutations.

Besides the average number of crossings, we are also interested in the question, for how many graphs the heuristic solutions are far away from the optimum. Figure 4.29 shows the distribution of graphs according to the difference to the optimum for the best heuristic VAR-INC-PERM30, grouped by the different types of graphs. The $P(m,2)$, $P(m,3)$, and $C_m \times C_n$ graphs could all be solved optimally. From the $G_i \times C_n$ graphs, 92.6% could be solved optimally and the largest difference that occurred was 12 for a single graph (graph $G_{16} \times C_{46}$ with 150 crossings). The $G_i \times P_n$ graphs are harder to solve, since only 73.8% could be solved optimally, but 94.1% required at most two crossing more. The largest difference of 28 was achieved by graph $G_{21} \times P_{48}$ with 316 crossings. Since the $G_{21} \times P_n$ graphs have a quite large crossing number of $6n$, this is not surprising and still a good re-

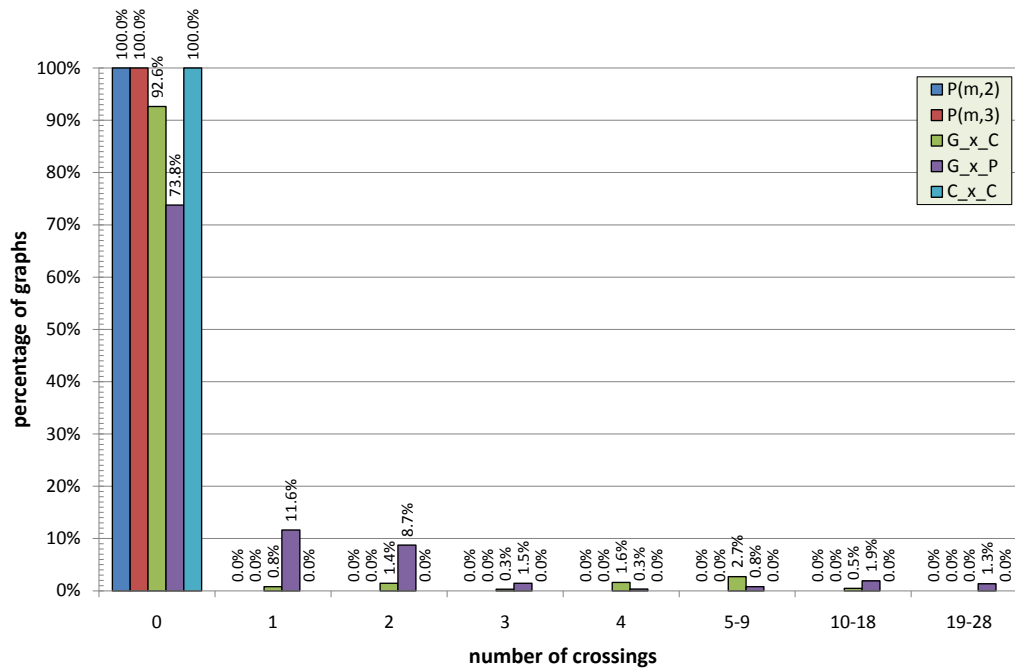


Figure 4.29: Distribution of graphs grouped by difference between heuristic solution of VAR-INC-PERM30 and the exact crossing number.

sult. Altogether, we found that there are no big outliers and the heuristic achieves reliably good results.

4.3.3 Summary

We have conducted an extensive experimental study on the crossing minimization problem for two benchmark sets of graphs. Our main conclusions are summarized as follows:

- Postprocessing always helps. It is recommended not to restrict the postprocessing procedure to the inserted edges. Already re-inserting 25% of all the edges helps a lot.
- Permutations and random effects help, but not as much as postprocessing; a small number of permutations, for example, 5–10, is already sufficient to give good improvements without increasing the running time too much.
- The higher running time of the incremental postprocessing variant is justified by the improvements achieved.
- It is important to start with a good planar subgraph. A better subgraph does not only improve the number of crossings achieved by the heuristics, but also improves the running times.
- The edge re-insertion with variable embedding is still worth doing, even if postprocessing is applied.

- (f) Preprocessing improves the running times if edge insertion with variable embedding is used; for the fixed embedding case preprocessing is not recommended, since the running times increase too much.

Chapter 5

Graph Embedding

*Everything that can be counted does not necessarily count;
everything that counts cannot necessarily be counted.*

ALBERT EINSTEIN (1879 – 1955)

In the previous chapter, we discussed how to obtain a drawing with a small number of edge crossings. The outcome of the crossing minimization procedure is a planarized representation of the graph we want to draw, and thus a planar graph allowing us to apply planar graph drawing algorithms. Usually, planar orthogonal drawing algorithms are used as proposed by the topology-shape-metrics approach, rendering each crossing vertex in the planarized representation as a nice crossing of a horizontal and a vertical edge segment. Such orthogonal drawing algorithms, in general, need as input a planar embedding of the graph. For example, the bend minimization algorithm by Tamassia [1987] allows to produce a bend minimal drawing of a four-planar graph (that is, a planar graph with maximum vertex degree four) for a given planar embedding. On the other hand, minimizing the number of bends over the set of all possible planar embeddings is already an NP-hard optimization problem, showing that the choice of the embedding is essential for the quality of the resulting drawing. Of course, generalizations of Tamassia's algorithm to planar graphs of higher degree, for example, the well known Kandinsky algorithm [Fößmeier and Kaufmann, 1996], suffer from the same problem. Examples that demonstrate the impact of the choice of the planar embedding are shown in Figure 5.1 and 5.2; these examples are discussed in more detail below.

There are two possible solutions to this problem. We could either try to solve the general bend minimization problem, or we try to find good additional criteria for the computation of the planar embedding that could lead to better orthogonal drawings. Though there are exact approaches to the general bend minimization problem for four-planar graphs by Bertolazzi et al. [2000] and Mutzel and Weiskircher [2002], these approaches are not suited for practical applications due to the complexity of the problem; for example, all the exact algorithms only work well for graphs with up to 80 edges. A better and more practical alternative seems to be developing efficient algorithms for computing planar em-

beddings optimizing certain criteria. Several authors have studied the problem of computing planar embeddings which are optimal with respect to some distance measures. Bienstock and Monma [1990] have suggested polynomial time algorithms for computing planar embeddings minimizing various distance measures:

- Two faces are adjacent if they share a common vertex. Minimize the *radius*, that is the maximal distance between a face and the external face.
- Two vertices are adjacent if they are the endpoints of an edge. Minimize the *width* (or *gauge*) of the embedding, that is, the maximal distance between a vertex and the external face cycle.
- Two vertices are adjacent if they are contained in a common face and the external face is adjacent to all vertices contained in it. Minimize the *outer-planarity*, that is, the maximal distance from a vertex to the external face.
- Two faces are adjacent if they share an edge. Minimize the *depth* of the embedding, that is, the maximal distance between a face and the external face.

In particular, they presented a $\mathcal{O}(n^5 \log n)$ algorithm for minimizing the depth. They also showed that it is NP-complete to test whether a planar graph has an embedding with dual diameter bounded above by an input number. Recently, Angelini et al. [2010b] improved the runtime for minimizing the depth to $\mathcal{O}(n^4)$. However, these asymptotic runtime bounds indicate that these approaches are not useful in practise. In particular, we are interested in algorithms with the same time complexity as computing a planar embedding, that is, linear running time.

In this chapter, we propose to study two distance measures, namely the block-nesting depth (formally defined below) and the degree of the external face; both measures can also be combined (see Section 5.3). We give linear time algorithms based on the SPQR-tree data structure for computing planar embeddings optimizing these distance measures. Experience in the graph drawing community has already shown that planar embeddings with unnecessarily high nesting of blocks or small external faces lead to less readable drawings, which can also be measured in terms of bends, edge lengths, and drawing area [Batini et al., 1986, 1984, Didimo and Liotta, 1998, Pizzonia and Tamassia, 2000, Weiskircher, 2002].

The first distance measure is the block-nesting depth of a planar embedding introduced by Pizzonia and Tamassia [2000]¹. They have suggested a linear-time algorithm for a restricted version of computing a planar embedding with minimum block-nesting depth in which the embeddings of the blocks are given and fixed. Unlike their algorithm, our new algorithm finds the planar embedding with minimum block-nesting depth over the set of all planar embeddings without any restrictions. Since—also in practice—we do not expect that blocks must

¹In their paper, the block-nesting depth is simply called depth; however, we prefer the more meaningful term block-nesting depth for clarity.

have a fixed embedding, the freedom of a variable embedding for each block gives us a much better chance to avoid unfavorable nestings of blocks.

The *block-nesting depth* of a planar embedding (Γ, f) is a measure of the topological nesting of the blocks of G in (Γ, f) . For the formal definition, we need the notion of the *extended dual BC-tree*.

Definition 5.1. Let (Γ, f) be a planar embedding of a graph G and denote with Γ^* the dual graph of Γ in which all self loops have been split, that is, replaced by a path of two edges. The *extended dual BC-tree* of (Γ, f) is a rooted tree; we distinguish two cases for its definition:

- (a) If f is a cut-vertex in Γ^* , then the extended dual BC-tree is defined as the BC-tree \mathcal{B} of Γ^* rooted at the C-node of \mathcal{B} associated with f .
- (b) Otherwise, all edges on the external face f belong to the same block B of G and the extended dual BC-tree is defined as the BC-tree \mathcal{B} of Γ^* , rooted at a new node r (representing f) connected to the B-node of \mathcal{B} associated with the block of Γ^* containing the dual edges of B .

The splitting of self-loops in the definition above is required for handling blocks consisting of a single edge correctly; such bridges would not be represented in the BC-tree of the dual graph since they contain no faces; however, they are relevant when considering the nesting of blocks.

Now we are ready to introduce the formal definition of the block-nesting depth. Given a planar embedding (Γ, f) of a connected and planar graph G , the *block-nesting depth* of (Γ, f) is the height of the extended dual BC-tree of (Γ, f) .

Figure 5.1 shows an example with two planar embeddings that have different block nesting depth; their corresponding extended dual BC-trees are shown below. It has already been provided by Pizzonia and Tamassia [2000] for justifying their approach for minimizing the block-nesting depth. Each of the two drawings has bends and area optimized for its embedding. The planar embedding in Figure 5.1(a) has a block-nesting depth of five, while the one in Figure 5.1(b) only has one. Obviously, the drawing in Figure 5.1(b) is easier to read and to understand than the one in Figure 5.1(a), displaying the underlying structure of the graph much cleaner.

Figure 5.2 shows two drawings of the same graph realizing different planar embeddings. Again, both drawings have the minimal number of bends with respect to their planar embeddings. The drawing in Figure 5.2(b) looks much more compact than the drawing in Figure 5.2(a). We observe that the graph is biconnected and hence the block-nesting depth of any planar embedding is simply one. On the other hand, the external face in Figure 5.2(b) is bordered by nine edges and is much larger than in Figure 5.2(a) with only three edges.

Also, it is evident that the two embeddings in Figure 5.1 differ in the number of edges contained in the external face: The embedding in Figure 5.1(a) has three edges bordering the external face, while the better drawing shown in Figure 5.1(b) has 15. This goes along with our observation that a higher number of edges on the external face leads to improved layout quality.

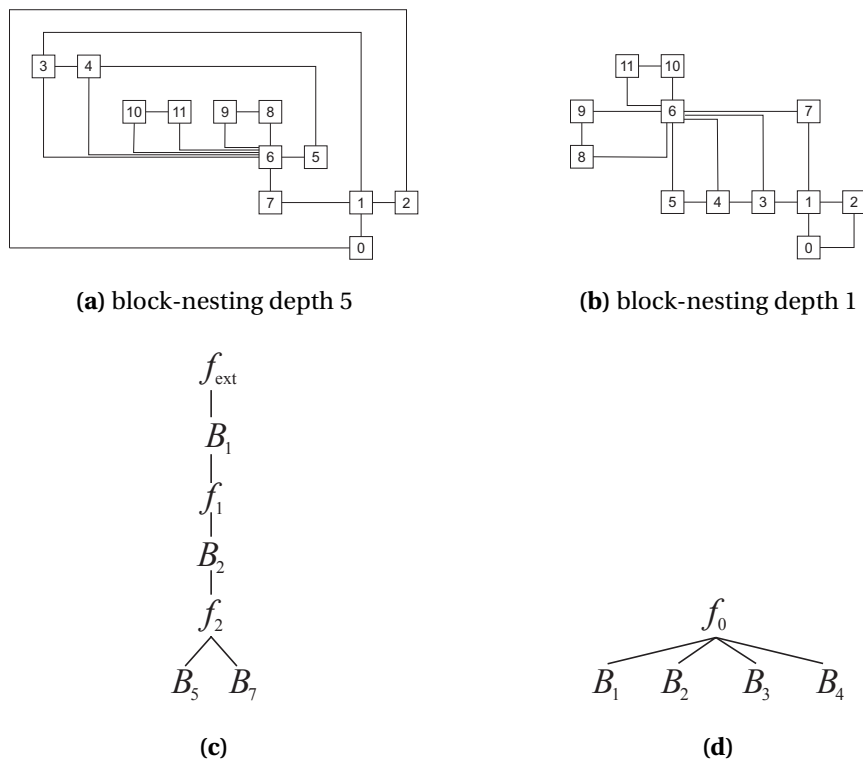


Figure 5.1: Two drawings of the same graph as shown in [Pizzonia and Tamassia, 2000] with their extended dual BC-trees shown below; both drawings have been computed by the GDToolkit system [GDToolkit]. The block-nesting depth of the planar embedding in (a) is five, whereas the one in (b) is only one.

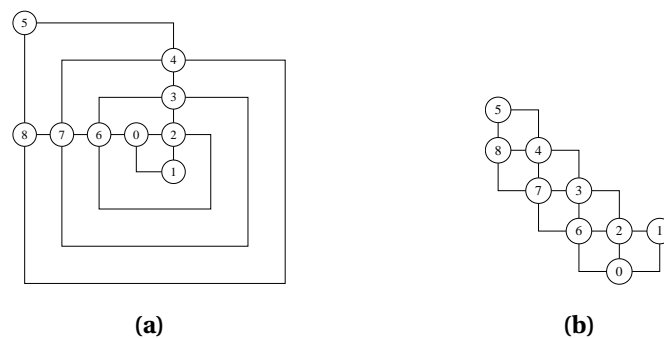


Figure 5.2: Two bend-minimal orthogonal drawings of a graph with different planar embeddings.

Hence, the second distance measure investigated in this chapter is the length of the external face cycle, that is, the degree of the external face, in the planar embedding. In the next section, we present a linear-time algorithm that computes a planar embedding with an external face of maximum degree among all possible planar embeddings.

The third distance measure we consider is a combination of the first two distance measures: We search for a planar embedding with an external face of maximum degree among all planar embeddings with minimum block-nesting depth. We conjecture that this measure provides, in general, the best planar embeddings among the considered distance measures in our paper, and in the literature, leading to improved planar layouts in practice.

5.1 Maximum External Face

Let $G = (V, E)$ be a planar and connected graph without self-loops. We consider the *maximum external face problem* defined as follows:

MAXIMUM EXTERNAL FACE (MEF)	
Instance:	a planar and connected graph G (without self-loops)
Solution:	a planar embedding (Π, f_0) of G
Maximize:	the degree of the external face f_0

Since every face in an embedding can be chosen as external face, we simply need to determine a combinatorial embedding with a face of maximum degree, which can then be set as external face. We first consider biconnected graphs and present a linear-time algorithm for finding an embedding with maximum external face, as well as an efficient algorithm for the restricted problem where the external face needs to contain a predefined vertex; the latter is an important sub-problem we need for generalizing the algorithm to connected graphs. For both algorithms, we also assume that we have given a weight function on the vertices; these weights are later used for modeling further blocks that are attached at a cut vertex v . Since we can embed all these blocks into the same face adjacent to v , we can thereby increase the degree of that face; the weight of a vertex v will tell us by how many edges we can increase the degree of a face containing v .

In our algorithms, we restrict us to computing only the degree of a maximum face, without actually producing the planar embedding. This makes our presentation much clearer and easier to understand. The general idea, how to obtain the corresponding planar embedding, will be evident from the presentation; a detailed description can be found in the master thesis of Thorsten Kerkhof [Kerkhof, 2007].

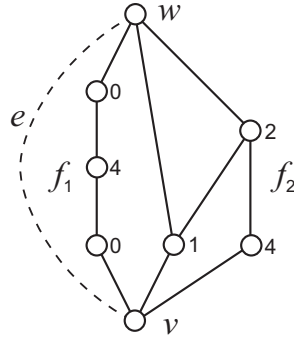


Figure 5.3: An embedding of $\text{expansion}^+(e)$ with given vertex weights. Assuming that v and w have weight zero, face f_1 has weight 9 and f_2 has weight 10; therefore edge e has component weight ≥ 10 .

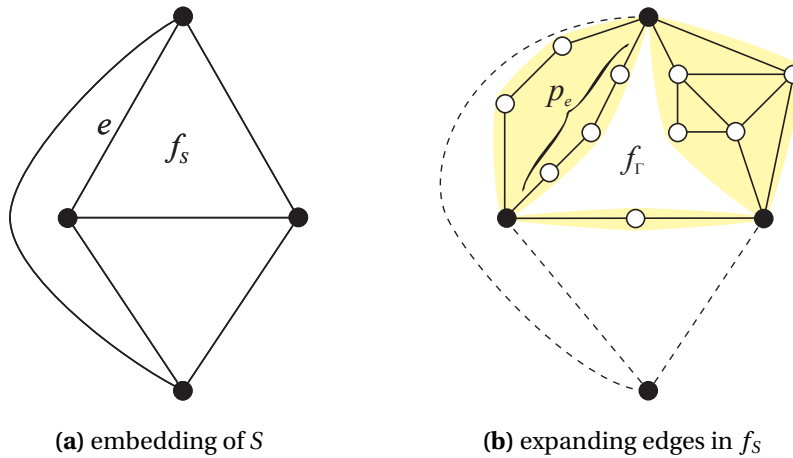


Figure 5.4: A face f_S in the skeleton S of an R-node (a) and the resulting face f_Γ after expanding all edges in f_S (b).

5.1.1 Biconnected Graphs

Let $B = (V_B, E_B)$ be a block of G with a weight function $w : V_B \rightarrow \mathbb{N}$, and let \mathcal{T}_B be the SPQR-tree of B . For each node $\mu \in \mathcal{T}$, we will also compute an edge weight $w_\mu(e) \in \mathbb{N}$ for each edge in $\text{skeleton}(\mu)$. The weight of a real edge is always one; the weights of virtual edges will be defined below. We define the *weight* of a face f in an embedding of $\text{skeleton}(\mu)$ to be $w_\mu(f) = \sum_{e \in f} w_\mu(e) + \sum_{v \in V(f)} w(v)$. We will use a similar definition for faces in embeddings of B ; in this case only the weights of vertices vary and all edges have weight one. The latter definition is also applied for embeddings of expansion graphs—again, all edges have weight one and the vertex weights of the given weight function w are used.

Consider an edge $e = (u, v)$ in a skeleton and let Γ_e be an embedding of $\text{expansion}^+(e)$ such that Γ_e has a face f^* containing e of maximum weight among all embeddings of $\text{expansion}^+(e)$. We call such an embedding an embedding of $\text{expansion}^+(e)$ with *maximum weight* and define the *component weight* of e to be $w(f^*) - 1 - w(u) - w(v)$; compare also Figure 5.3.

The general idea of the algorithm is as follows: Let S be a skeleton for which we have chosen an embedding Γ_S . In order to extend Γ_S to an embedding Γ of B , we have to choose embeddings of the graphs $\text{expansion}^+(e)$ for all edges $e \in S$. Each face f_S in Γ_S corresponds to a face f_Γ in Γ in which each skeleton edge $e \in f_S$ is replaced by a path p_e on the external face of its expansion graph; see Figure 5.4. We call this operation *expanding face* f_S to face f_Γ . Vice versa, for each face f'_Γ in Γ , we can find a face in a skeleton that can be expanded to f'_Γ . Face f_Γ is made as large as possible by embedding each expansion graph of an edge in f_S , such that the *weight* of the path p_e , which we define as the number of edges in p_e plus the weights of the interior vertices on p_e , is the component weight of e . We get the following Lemma:

Lemma 5.1. *Let B be a planar and biconnected graph with vertex weights $w : V(B) \rightarrow \mathbb{N}$ and \mathcal{T}_B its SPQR-tree. Then the following statements hold:*

- (i) *Each skeleton face f can be expanded to a face f' such that the weight of f' is the sum of the weights of the vertices in f plus the component weights of all edges in f .*
- (ii) *There exists an internal node $\mu \in \mathcal{T}_B$ and an embedding Γ_μ of $\text{skeleton}(\mu)$ such that there is a face f in Γ_μ that can be expanded to a face f^* of maximum weight. Moreover, all edges in f are expanded to expansion graphs with maximum weights.*

```

1: function COMPUTEEDGEWEIGHTS(block  $B$ ,  $w : V(B) \rightarrow \mathbb{N}$ )
2:   let  $\mu_r$  be the root of  $\mathcal{T}_B$  and  $v$  its (only) son
3:   GETCOMPONENTWEIGHTS( $B$ ,  $w$ ,  $v$ )
4:   SETREFWEIGHTS( $B$ ,  $w$ ,  $v$ , 1)
5:   for each R-node, precompute the weights of all skeleton faces
6:   for each P-node, precompute the two edges with largest weights
7:   for each S-node, precompute the weights of a skeleton face
8: end function

```

Listing 5.1: Computes the weight of each skeleton edge in \mathcal{T}_B ; performs some additional precomputations required in MAXIMUMFACEBLOCK and CONSTRAINTMAXFACEBLOCK.

In the following we show how to compute the component weights of all skeleton edges efficiently; see Algorithm COMPUTEEDGEWEIGHTS in Listing 5.1. We can compute the component weight of all non-reference skeleton edges by a bottom-up traversal of \mathcal{T}_B . Let e be an edge in $\text{skeleton}(\mu)$ and v the pertinent node of e . We denote with e_{ref} the reference edge of v and with W the sum of the weights of the two poles of μ . If we assume that the weights of all edges in $\text{skeleton}(v)$ except for e_{ref} are set to their component weights, we can compute the component weight of e by distinguishing four cases according to the type of v :

Q-node: The component weight of e is one.

S-node: The component weight of e is the weight of an arbitrary face in $skeleton(v)$ minus W .

P-node: The component weight of e is the weight of the heaviest edge different from e_{ref} in $skeleton(v)$.

R-node: The component weight of e is the weight of the heaviest face containing e_{ref} in $skeleton(v)$ minus W .

This computation is realized by Algorithm GETCOMPONENTWEIGHTS in Listing 5.2.

```

1: function GETCOMPONENTWEIGHTS(block  $B$ ,  $w : V(B) \rightarrow \mathbb{N}$ , node  $\mu$ )
2:   let  $S$  be the skeleton of  $\mu$  and  $e_{\text{ref}}$  its reference edge
3:   for all  $\mu \rightarrow v \in \mathcal{T}_B$  do
4:      $w_\mu(e_{S,v}) := \text{GETCOMPONENTWEIGHTS}(B, w, v)$ 
5:   end for
6:    $w_\mu(e_{\text{ref}}) := 0$ 
7:    $W :=$  sum of the weights of the two poles of  $\mu$ 
8:   switch type of  $\mu$ 
9:     case Q-node: return 1
10:    case S-node: return weight of an arbitrary face in  $S$  minus  $W$ 
11:    case P-node: return weight of the heaviest edge different from  $e_{\text{ref}}$  in  $S$ 
12:    case R-node: return weight of the heaviest face containing  $e_{\text{ref}}$  minus
       $W$ 
13:   end switch
14: end function

```

Listing 5.2: Sets the weights of all edges in $skeleton(\mu)$ except for the reference edge; returns the weight of the skeleton edge representing the pertinent graph of μ .

Once we have set the weights of all non-reference skeleton edges to their component weights, we can compute the component weights of the reference edges by a top-down traversal of \mathcal{T}_B . Since \mathcal{T}_B is rooted at a Q-node, the component weight of the reference edge of its only child is one. Consider now a node $\mu \in \mathcal{T}_B$ and let S be the skeleton of μ . We assume that the weight of each edge in S is set to its component weight. We want to compute the component weight of the reference edge of each child v of μ . We denote with $e_{S,v}$ the edge in $skeleton(\mu)$ whose pertinent node is v and distinguish three cases according to the type of μ :

S-node: Let W be the sum of the weights of all vertices and edges in S . The component weight of the reference edge of v is then W minus the weight of $e_{S,v}$ minus the weight of the two endpoints of $e_{S,v}$.

P-node: The component weight of the reference edge of v is the weight of the heaviest edge in S different from $e_{S,v}$.

R-node: Let f be the heaviest face in S containing $e_{S,v}$. The component weight of the reference edge of v is the weight of f minus the weight of $e_{S,v}$ minus the weights of the two endpoints of $e_{S,v}$.

This top-down traversal is performed by Algorithm SETREFWEIGHTS shown in Listing 5.3. The calling procedure COMPUTEEDGEWEIGHTS (see Listing 5.1) performs some additional precomputations, which are used later to implement Algorithm CONSTRAINTMAXFACEBLOCK(B, c) (see Listing 5.5) efficiently. This algorithm determines a face of maximum weight in a block B containing vertex c .

```

1: procedure SETREFWEIGHTS(block  $B$ ,  $w : V(B) \rightarrow \mathbb{N}$ , node  $\mu$ , int  $w_{\text{ref}}$ )
2:   let  $S$  be the skeleton of  $\mu$  and  $e_{\text{ref}}$  its reference edge
3:    $w_{\mu}(e_{\text{ref}}) := w_{\text{ref}}$ 
4:   switch type of  $\mu$ 
5:     case S-node:
6:        $W :=$  sum of the weights of all vertices and edges in  $S$ 
7:       for all  $\mu \rightarrow v \in \mathcal{T}_B$  do
8:         let  $e_{S,v} = (x, y)$  be the edge in  $S$  whose pertinent node is  $v$ 
9:         SETREFWEIGHTS( $B, w, v, W - w_{\mu}(e_{S,v}) - w(x) - w(y)$ )
10:      end for
11:     case P-node:
12:       let  $e_1$  be the heaviest and  $e_2$  the second heaviest edge in  $S$ 
13:       for all  $\mu \rightarrow v \in \mathcal{T}_B$  do
14:         let  $e_{S,v}$  be the edge in  $S$  whose pertinent node is  $v$ 
15:         if  $e_{S,v} = e_1$  then  $e_{\text{max}} := e_2$ 
16:         else  $e_{\text{max}} := e_1$ 
17:         SETREFWEIGHTS( $B, w, v, w_{\mu}(e_{\text{max}})$ )
18:       end for
19:     case R-node:
20:       Precompute the weight  $w_{\mu}(f)$  of each face  $f$  in  $S$ 
21:       for all  $\mu \rightarrow v \in \mathcal{T}_B$  do
22:         let  $e_{S,v} = (x, y)$  be the edge in  $S$  whose pertinent node is  $v$ 
23:         let  $f$  be the heaviest face in  $S$  containing  $e_{S,v}$ 
24:         SETREFWEIGHTS( $B, w, v, w_{\mu}(f) - w_{\mu}(e_{S,v}) - w(x) - w(y)$ )
25:       end for
26:     end case
27:   end switch
28: end procedure

```

Listing 5.3: Assigns w_{ref} to the weight of the reference edge in $\text{skeleton}(\mu)$ and recursively sets the length of the reference edges in all children of μ to their component weights.

Lemma 5.2. *Algorithm COMPUTEEDGEWEIGHTS(B, w) takes time linear in the size of B .*

Proof. GETCOMPONENTWEIGHTS is called for each node in \mathcal{T}_B (except for the root node) exactly once. Each call for a node μ takes—excluding the recursive calls—time linear in the size of $\text{skeleton}(\mu)$. SETREFWEIGHTS is also called for each tree

node (except for the root) exactly once. In each call for a node μ , we perform some precomputations that require time linear in the size of $\text{skeleton}(\mu)$. In the iteration over all children of μ , each step takes only constant time, excluding the recursive call. This proves that the calls of `GETCOMPONENTWEIGHTS` and `SETREFWEIGHTS` in `COMPUTEEDGEWEIGHTS` take time linear in the size of all skeletons. The final precomputations can also be performed in time linear in the size of all skeletons. Since the size of all skeletons in \mathcal{T}_B is linear in the size of B , the lemma follows. \square

According to Lemma 5.1, we need to find an embedding Γ_μ of the skeleton of a node μ with a face f of maximum weight among all possible embeddings of skeletons. This can be achieved by inspecting all skeletons S . If S is the skeleton of an R-node, S has only two embeddings which are mirror images of each other. The heaviest face we can produce is simply a heaviest face in an arbitrary embedding of S . If S is the skeleton of a P-node, say a bundle of parallel edges e_1, \dots, e_k , we can form any face consisting of two edges e_i and e_j with $i \neq j$. Hence, the heaviest face we can produce consists of the two heaviest edges in S . If S is the skeleton of an S-node, S has only a single embedding consisting of two faces with equal weight.

```

1: function MAXIMUMFACEBLOCK(block  $B$ ,  $w : V(B) \rightarrow \mathbb{N}$ )
2:   COMPUTEEDGEWEIGHTS( $B$ ,  $w$ )
3:    $w_{\max} := 0$ 
4:   for all  $\mu \in \mathcal{T}_B$  do
5:     let  $S$  be the skeleton of  $\mu$ 
6:     switch type of  $\mu$ 
7:       case R-node:
8:         let  $f$  be the heaviest face in  $S$ 
9:          $w_{\max} := \max(w_{\max}, w_\mu(f))$ 
10:      case P-node:
11:        let  $e_1$  and  $e_2$  be the two heaviest edges in  $S$ 
12:         $W := w_\mu[e_1] + w_\mu[e_2] + \text{weights of the two vertices in } S$ 
13:         $w_{\max} := \max(w_{\max}, W)$ 
14:      case S-node:
15:        let  $W$  be the weight of a skeleton face in  $S$ 
16:         $w_{\max} := \max(w_{\max}, W)$ 
17:      end case
18:    end switch
19:  end for
20:  return  $w_{\max}$ 
21: end function

```

Listing 5.4: Computes the weight of a heaviest face in an embedding of B .

This shows that we can find a tree node μ and an embedding Γ_μ of its skeleton with a face f that can be expanded to a maximum face of B ; see Algorithm MAX-

IMUMFACEBLOCK in Listing 5.4. According to Lemma 5.1, all edges in f have to be expanded to expansion graphs with maximum weight, which can be achieved by recursively traversing the tree nodes involved. We obtain the following theorem:

Theorem 5.1. *Let $B = (V_B, E_B)$ be a biconnected and planar graph with weights $w : V_B \rightarrow \mathbb{N}$. Then, procedure MAXIMUMFACEBLOCK computes the weight of a maximum weight face in a planar embedding of B in time $\mathcal{O}(|V_B| + |E_B|)$.*

In order to compute the weight of a maximum weight face containing a prescribed vertex v , we have to consider all allocation nodes of v in \mathcal{T}_B . Procedure COMPUTEEDGEWEIGHTS has already precomputed the weights of all skeleton faces in S- and R-nodes, as well as the weights of the two heaviest edges within a P-node. Therefore, we can compute the weight of a maximum weight face containing v very efficiently; see Algorithm CONSTRAINTMAXFACEBLOCK in Listing 5.5.

```

1: function CONSTRAINTMAXFACEBLOCK(block  $B$ ,  $w : V(B) \rightarrow \mathbb{N}$ , vertex  $v$ )
2:    $w_{\max} := 0$ 
3:   for all allocation nodes  $\mu$  of  $v$  in  $\mathcal{T}_B$  do
4:     let  $S$  be the skeleton of  $\mu$ 
5:     switch type of  $\mu$ 
6:       case P-node:
7:         let  $u$  be the vertex in  $S$  different from  $v$ 
8:         let  $e_1$  and  $e_2$  be the two heaviest edges in  $S$ 
9:          $w_{\max} := \max(w_{\max}, w(u) + w_{\mu}(e_1) + w_{\mu}(e_2))$ 
10:        case R-node:
11:          let  $f$  be the heaviest face in  $S$  containing  $v$ 
12:           $w_{\max} := \max(w_{\max}, w_{\mu}(f) - w(v))$ 
13:        case S-node:
14:          let  $f$  be an arbitrary face in  $S$ 
15:           $w_{\max} := \max(w_{\max}, w_{\mu}(f) - w(v))$ 
16:        end case
17:      end switch
18:    end for
19:    return  $w_{\max}$ 
20: end function

```

Listing 5.5: Computes the weight of a heaviest face in B containing v , where we ignore the weight of v itself; assumes that COMPUTEEDGEWEIGHTS(B, w) has already been called.

Lemma 5.3. *Let $B = (V_B, E_B)$ be a biconnected and planar graph with weights $w : V_B \rightarrow \mathbb{N}$ and assume that procedure COMPUTEEDGEWEIGHTS(B, w) has already been called. Then, procedure CONSTRAINTMAXFACEBLOCK computes the weight of a maximum weight face in a planar embedding of B in time $\mathcal{O}(n_v + m_v)$, where n_v denotes the number of allocation nodes of v and m_v denotes the total number of skeleton edges incident to representatives of v .*

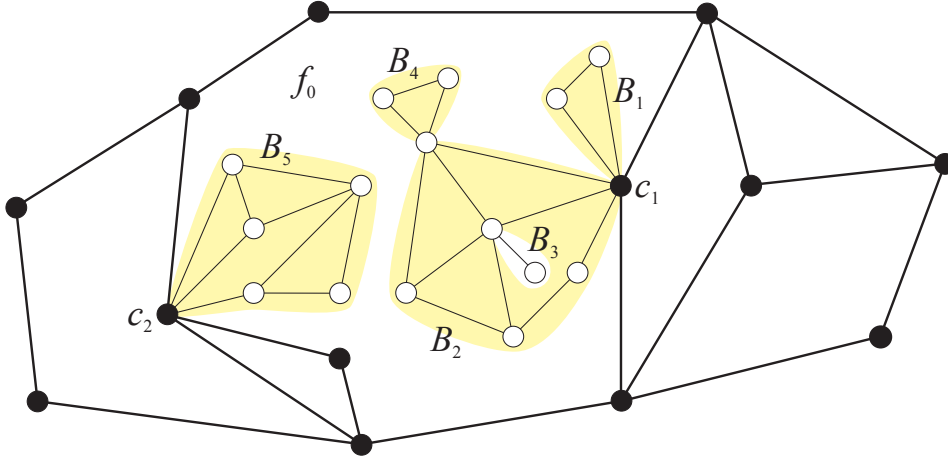


Figure 5.5: A fixed embedding Γ_0 of B_0 (thick edges), where the graphs G_{c_1, B_1} , G_{c_1, B_2} and G_{c_2, B_5} have been placed into face f_0 .

Proof. It is clear that it is sufficient to consider only skeletons containing v . We assume that the SPQR-tree data structure stores the allocation nodes of each vertex (if not, these can easily be computed in time linear in the size of B). Each S- and P-case in `CONSTRAINTMAXFACEBLOCK` takes only constant time, since we have precomputed the two heaviest edges in a P-node and the size of an S-node. For an R-node, we have precomputed the weights of the faces. We need to consider as many faces as there are skeleton edges incident to v , altogether at most m_v faces for the complete call. Hence, the running time is $\mathcal{O}(n_v + m_v)$. \square

5.1.2 Connected Graphs

Let \mathcal{B} be the BC-tree of G . Consider a block B and a cut-vertex c in G with $c \in B$. If we delete the edge connecting c with B in \mathcal{B} , that is, the edge $c \rightarrow B$ or $B \rightarrow c$, \mathcal{B} is split into two connected components, where one component \mathcal{B}_B contains B . We denote the graph induced by the edges in all blocks contained in \mathcal{B}_B with $G_{c, B}$.

The idea of the algorithm is as follows; compare Figure 5.5. Suppose we have fixed an embedding Γ_0 of a block B_0 . In order to extend Γ_0 to an embedding of G , we have to find an embedding $\Gamma_{c, B}$ with c on its external face for each graph $G_{c, B}$ with $c \in B$, $B \neq B_0$, and place $\Gamma_{c, B}$ into an adjacent face of c in Γ_0 . For a fixed face f_0 in Γ_0 , we can obviously enlarge f_0 as much as possible if we place all $\Gamma_{c, B}$ with $c \in f_0$ into f_0 . We denote with $\Psi_B(c)$ the sum of the weights of the maximum weight faces containing c of all $G_{c, B'}$ with $c \in B'$, $B' \neq B$. If c is not a cut-vertex in G , then $\Psi_B(c)$ is 0. We will use the value $\Psi_B(c)$ as the vertex weight of c in B_0 and apply the procedures developed in the previous section. The following lemma states the crucial results on which the correctness of our algorithm is based.

Lemma 5.4. (i) *Let B be a block of G and define the weight of a vertex v in B as $\Psi_B(v)$. If f_B is a face in an arbitrary embedding Γ_B of B , then there exists an embedding Γ of G with a face f , such that the degree of f equals the weight*

of f_B . Face f results from embedding each $G_{c,B'}$ with $c \in B$, $B' \neq B$, with a maximum weight external face containing c and placing it into f_B .

- (ii) Let f be a face in an embedding Γ of G . Then, there exists a block B of G , a weight w_v for each vertex $v \in B$ with $w_v \leq \Psi_B(v)$, and an embedding Γ_B of B with a face f_B such that the degree of f equals the weight of f_B .

Proof. (i) follows directly from the definition of $\Psi_B(c)$ and the fact that we can place each subgraph $G_{c,B'}$ into an arbitrary face adjacent to c in Γ_B .

Consider now (ii). Let B be an arbitrary block of G containing an edge in f . We denote with E_f the set of edges contained in both B and f , and with Γ_B the embedding of B induced by Γ . Then, the edges in E_B must form a face f_B in Γ_B . The edges on f_B appear in the same cyclic order as on f , but not necessarily consecutively. For each cut vertex c , a (possibly empty) sequence of edges from subgraphs $G_{c,B'}$ with $B' \neq B$ appears on f ; we denote the number of edges in this sequence with w_c and set $w_v = 0$ for all other vertices in B . Obviously, $w_v \leq \Psi_B(v)$ for all $v \in B$ and the weight of f_B (with respect to the vertex weights w_v) equals the degree of f . Hence, (ii) follows as well. \square

Theorem 5.2. For each block B of G , let $\Psi_B(v)$ be the weight of a vertex $v \in B$. Let B_{\max} be the block having the embedding Γ_{\max} with a heaviest maximum weight face f_{\max} among all blocks. Then, we can extend Γ_{\max} to a planar embedding of G with maximum external face f . Face f results from embedding each $G_{c,B'}$ with $c \in B_{\max}$ and $B' \neq B_{\max}$ with a maximum weight external face containing c and placing it into f_{\max} .

Proof. According to Lemma 5.4(i), we will find face f with the required degree in an embedding Γ of G . Lemma 5.4(ii) shows that a maximum face in G cannot be larger than f , hence Γ is an embedding of G with a face f of maximum degree. \square

Algorithm MAXIMUMFACE in Listing 5.6 shows how to compute the degree of the external face in a planar embedding of G with a maximum external face. We store in $w_B(v)$ the weight of vertex v in block B and set $ctrLength(B, c)$ to the weight of a maximum weight face of $G_{c,B}$ containing c . Function CONSTRAINT-MAXFACE(B, c) computes the weight of a maximum face in $G_{c,B}$ containing c and is called for each edge $c \rightarrow B \in \mathcal{B}$, thereby computing $ctrLength(B, c)$ for each edge $c \rightarrow B \in \mathcal{B}$. Then, function MAXIMUMFACEREC recursively traverses \mathcal{B} from top to bottom. When MAXIMUMFACEREC is called for a block B , the weight of each vertex $v \in B$ is already set to $\Psi_B(v)$ and we can call MAXIMUMFACEBLOCK to compute the weight of a maximum face in an embedding of B . For recursively traversing \mathcal{B} further, we first have to compute $ctrLength(B, c)$ for the edge $c \rightarrow B$ and set the weight of c in each block B' with $c \rightarrow B' \in \mathcal{B}$. Since $\Psi_{B'}(c)$ is the sum of all $\Psi_{\tilde{B}}(c)$ with $\tilde{B} \neq B'$, we can efficiently compute the weights by precomputing the sum W of all $\Psi_{\tilde{B}}(c)$. Function MAXIMUMFACEREC finally returns the maximum weight over all blocks and thus, by Theorem 5.2, the degree of a maximum face in an embedding of G .

```

1: function MAXIMUMFACE(graph  $G$ )
2:   Compute BC-tree  $\mathcal{B}$  of  $G$  and SPQR-tree  $\mathcal{T}_B$  for each block  $B \in \mathcal{B}$ .
3:   let  $B_{\text{root}}$  be the root block of  $\mathcal{B}$ 
4:   for all  $v \in B_{\text{root}}$  do
5:      $w_{B_{\text{root}}}(v) := \sum_{v \rightarrow B \in \mathcal{B}} \text{CONSTRAINTMAXFACE}(B, v)$ 
6:   end for
7:   return MAXIMUMFACEREC( $B_{\text{root}}$ )
8: end function

9: function CONSTRAINTMAXFACE(block  $B$ , vertex  $c$ )
10:  for all  $v \in B$  do
11:     $w_B(v) := \sum_{v \rightarrow B' \in \mathcal{B}} \text{CONSTRAINTMAXFACE}(B', v)$ 
12:  end for
13:  COMPUTEEDGEWEIGHTS( $B, w_B$ )
14:   $\text{cstrWeight}(B, c) := \text{CONSTRAINTMAXFACEBLOCK}(B, w_B, c)$ 
15:  return  $\text{cstrWeight}(B, c)$ 
16: end function

17: function MAXIMUMFACEREC(block  $B$ )
18:  COMPUTEEDGEWEIGHTS( $B, w_B$ )
19:   $w_{\text{max}} := \text{MAXIMUMFACEBLOCK}(B)$ 
20:  for all  $B \rightarrow c \in \mathcal{B}$  do
21:     $\text{cstrWeight}(B, c) := \text{CONSTRAINTMAXFACEBLOCK}(B, c)$ 
22:     $W := \text{cstrWeight}(B, c) + \sum_{c \rightarrow B' \in \mathcal{B}} \text{cstrWeight}(B', c)$ 
23:    for all  $c \rightarrow B' \in \mathcal{B}$  do
24:       $w_{B'}(c) := W - \text{cstrWeight}(B', c)$ 
25:       $w_{\text{max}} := \max(w_{\text{max}}, \text{MAXIMUMFACEREC}(B'))$ 
26:    end for
27:  end for
28:  return  $w_{\text{max}}$ 
29: end function

```

Listing 5.6: Planar embedding with maximum external face.

Theorem 5.3. *Let $G = (V, E)$ be a planar and connected graph. Then, Algorithm `MAXIMUMFACE(G)` computes a planar embedding of G with maximum external face in time $\mathcal{O}(|V| + |E|)$.*

Proof. We still have to prove the linear runtime. The computation of the BC-tree \mathcal{B} and the SPQR-trees \mathcal{T}_B require altogether time linear in the size of G , that is, $\mathcal{O}(|V| + |E|)$. Function `CONSTRAINTMAXFACE` is called for each block B (except B_{root}) exactly once. Each call takes time linear in the size of B —excluding recursive calls—, and thus altogether time $\mathcal{O}(|V| + |E|)$.

Function `MAXIMUMFACEREC` is called for each block $B = (V_B, E_B)$ exactly once. The calls to `COMPUTEEDGEWEIGHTS` and `MAXIMUMFACEBLOCK` take time $\mathcal{O}(|V_B| + |E_B|)$. `CONSTRAINTMAXFACEBLOCK` is called for each tree arc $B \rightarrow c$ in \mathcal{B} and each such call takes time $\mathcal{O}(n_c + m_c)$ by Lemma 5.3, where n_c denotes the number of allocation nodes of c and m_c the total number of skeleton edges incident to representatives of c . For all calls with B , this sums up to $\mathcal{O}(|V_B| + |E_B|)$. Therefore, the total runtime of `MAXIMUMFACE` is $\mathcal{O}(|V| + |E|)$. \square

All algorithms presented in this section can easily be generalized to graphs with predefined non-negative edge weights, in particular to edges with weights zero. We will use this in the following section. For ease of notation, we denote the corresponding algorithm for computing the edge weights with `COMPUTEEDGEWEIGHTS0`.

5.2 Minimum Block-Nesting Depth

In this section, we present a linear-time algorithm for minimizing the block-nesting depth. We define this problem formally as follows:

MINIMUM BLOCK-NESTING DEPTH EMBEDDING (MBNDE)	
Instance:	a planar and connected graph G
Solution:	a planar embedding (Π, f_0) of G
Minimize:	the block-nesting depth of (Π, f_0)

Let $G = (V, E)$ be a planar graph. Consider a block B with a planar embedding Γ_B . An *extension* of Γ_B denotes a planar embedding of G that results from embedding all graphs $G_{c, B'}$ with $c \in B$ and $B' \neq B$ such that c lies on the external face and placing them into a face of Γ_B .

Let $m_{c, B'}$ be the minimum block-nesting depth of a planar embedding of $G_{c, B'}$ with c on the external face. We define further:

$$\begin{aligned}
 m_B(c) &:= \max \{0\} \cup \{m_{c, B'} \mid c \in B', B' \neq B\} \\
 m_B &:= \max_{c \in B} m_B(c) \\
 M_B &:= \{c \in B \mid m_B(c) = m_B\}
 \end{aligned}$$

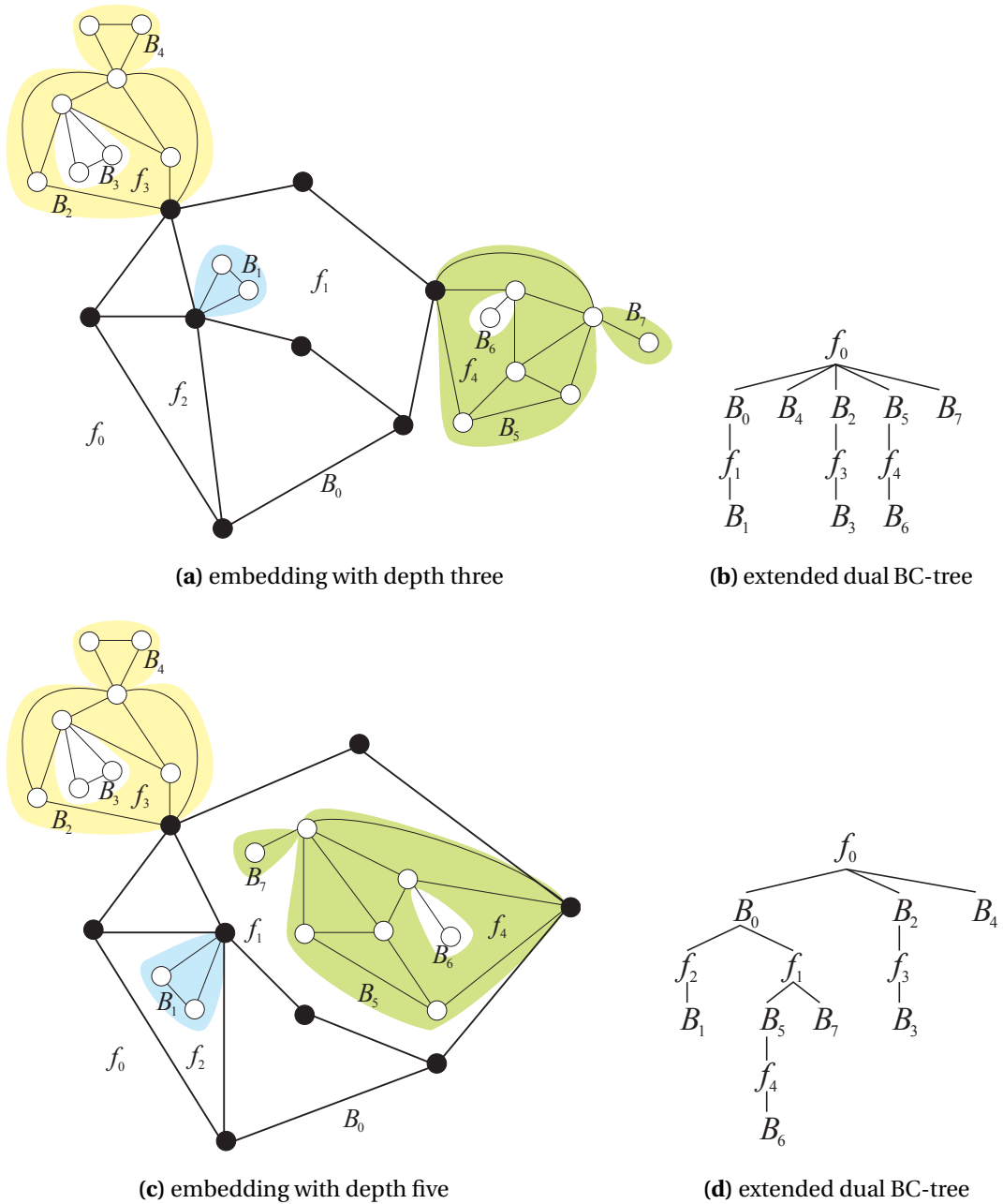


Figure 5.6: Two different extensions of an embedding Γ_0 of a block B_0 with their corresponding extended dual BC-trees.

That is, $m_B(c)$ is the largest block-nesting depth that occurs if we embed all the graphs $G_{c,B'}$ (with $c \in B'$) with minimal block-nesting depth among all embeddings with c on their external faces. Furthermore, m_B denotes the largest block-nesting depth we need to consider when extending an embedding of B and M_B is the set of cut vertices where this value occurs.

Then, we can easily derive the minimum block-nesting depth of an extension of an embedding of B . If it is possible to place all the vertices in M_B into a common face, then the minimum block-nesting depth is m_B , since the height of the

extended dual BC-tree does not increase. Otherwise, we have a path $f_{\text{ext}} \rightarrow B \rightarrow f' \rightarrow B'$ in the resulting extended dual BC-tree, where f_{ext} is the external face of the resulting embedding and f' is the non-external face into which a block B' with $m_{c,B'} = m_B$ has been placed. Hence, the block-nesting depth increases by two, that is, it is $m_B + 2$. This is also illustrated in Figure 5.6.

The following lemma formulates the resulting relationship between minimizing the block-nesting depth and maximizing the external face:

Lemma 5.5. *Assume we set the weight of all edges in B to 0 and the weight of a vertex v in B to 1 if $v \in M_B$ and 0 otherwise. Let (Γ^*, f^*) be a planar embedding of B with maximum external face f^* . Then, the minimum block-nesting depth of an extension of an embedding of B is*

$$\begin{cases} m_B & \text{if the weight of } f^* = |M_B| \\ m_B + 2 & \text{otherwise} \end{cases}$$

The problem of finding a planar embedding of a graph $G_{c,B}$ with minimum block-nesting depth such that c lies on the external face can be tackled in the same way. Based on this result, Algorithm MINIMUMDEPTH shown in Listings 5.7 through 5.9 proceeds similar to algorithm MAXIMUMFACE for connected graphs.

First, the values $m_{c,B}$ for all edges $c \rightarrow B$ in the BC-tree \mathcal{B} are computed by a bottom-up traversal of \mathcal{B} . The weight of each vertex $v \neq c$ in B is set according to Lemma 5.5 and a maximum face in B containing c is computed.

Afterwards, \mathcal{B} is traversed from top to bottom. When a block $B = (V_B, E_B)$ is visited, the values $m_{c,B'}$ are already computed for each cut-vertex c in G that is contained in B . According to Lemma 5.5, we set the vertex weights in B and compute the minimum block-nesting depth of an extension of B by finding a maximum face in B . Before we proceed with the descendants of B in the tree \mathcal{B} , we have to compute the values $m_{c,B}$ for all edges $B \rightarrow c \in \mathcal{B}$. We distinguish two cases:

$M_B = \{c_1, \dots, c_k\}$ with $k \geq 2$: We compute a maximum face in block B containing c using the vertex weights we have already assigned.

$M_B = \{c\}$: In this case, $m_2 = \max_{v \in V_B, v \neq c} m_B(v)$ is less than m_B and we cannot reuse the vertex weights. However, this case can occur at most once for each block which allows us to invest $\mathcal{O}(|B|)$ running time. We calculate new vertex weights according to $M_2 = \{c \in V_B \setminus \{v\} \mid m_B(c) = m_2\}$ and find a maximum face in B containing c using these vertex weights.

We obtain the following theorem.

Theorem 5.4. *Let $G = (V, E)$ be a planar and connected graph. Then, Algorithm MINIMUMDEPTH(G) computes a planar embedding of G with minimum block-nesting depth in time $\mathcal{O}(|V| + |E|)$.*

```

1: procedure MINIMUMDEPTH(graph  $G$ )
2:   Compute BC-tree  $\mathcal{B}$  of  $G$  and SPQR-tree  $\mathcal{T}_B$  for each block  $B \in \mathcal{B}$ .
3:   let  $B_{\text{root}}$  be the root block of  $\mathcal{B}$ 
4:   for all  $v \in B_r$  do
5:      $\text{maxCDepth}_{B_{\text{root}}}[v] := \max_{v \rightarrow B \in \mathcal{B}} \text{CONSTRAINTMINDEPTH}(B, v)$ 
6:   end for
7:    $(B, m) := \text{MINIMUMDEPTHREC}(B_{\text{root}})$ 
8: end procedure

9: function CONSTRAINTMINDEPTH(block  $B$ , vertex  $c$ )
10:  if  $B$  is a leaf in  $\mathcal{B}$  then
11:     $c\text{Depth}[B, c] := 1$ 
12:    return 1
13:  end if
14:  for all  $v \in B$ ,  $v \neq c$  do
15:     $\text{maxCDepth}_B[v] := \max_{v \rightarrow B' \in \mathcal{B}} \text{CONSTRAINTMINDEPTH}(B', v)$ 
16:  end for
17:   $m := \max\{\text{maxCDepth}_B[v] \mid v \in B, v \neq c\}$ 
18:   $M := \{v \in B \mid v \neq c, \text{maxCDepth}_B[v] = m\}$ 
19:  for all  $v \in B$  do
20:     $w_B(v) := \begin{cases} 1 & \text{if } v \in M \\ 0 & \text{if } v \notin M \end{cases}$ 
21:  end for

22:   $\text{COMPUTEEDGEWEIGHTS}_0(B, w_B)$ 
23:   $w_{\text{max}} := \text{CONSTRAINTMAXFACEBLOCK}(B, w_B, c)$ 
24:   $c\text{Depth}[B, c] := \begin{cases} m & \text{if } w_{\text{max}} = |M| \\ m + 2 & \text{otherwise} \end{cases}$ 
25:  return  $c\text{Depth}[B, c]$ 
26: end function

```

Listing 5.7: Planar embedding with minimum block-nesting depth.

```

1: function MINIMUMDEPTHREC(block  $B$ )
2:    $m_1 := \max_{v \in B} \text{maxCDepth}_B[v]$ 
3:    $M_1 := \{v \in B \mid \text{maxCDepth}_B[v] = m_1\}$ 
4:   for all  $v \in B$  do
5:      $w_B(v) := \begin{cases} 1 & \text{if } v \in M_1 \\ 0 & \text{if } v \notin M_1 \end{cases}$ 
6:   end for
7:   COMPUTEEDGEWEIGHTS0( $B, w_B$ )
8:    $(B^*, \ell^*) := (B, \text{MAXIMUMFACE}(B), w_B)$ 
9:    $m^* := \begin{cases} m_1 & \text{if } \ell^* = |M_1| \\ m_1 + 2 & \text{otherwise} \end{cases}$ 
10:  for all  $B \rightarrow c \in \mathcal{B}$  do
11:    if  $M_1 \neq \{c\}$  then
12:       $w_{\max} := \text{CONSTRAINTMAXFACEBLOCK}(B, w_B, c)$ 
13:       $cDepth[B, c] := \begin{cases} m_1 & \text{if } w_{\max} = |M_1 \setminus \{c\}| \\ m_1 + 2 & \text{otherwise} \end{cases}$ 
14:    else
15:       $m_2 := \max_{v \in B, v \neq c} \text{maxCDepth}_B[v]$ 
16:       $M_2 := \{v \in B \mid \text{maxCDepth}_B[v] = m_2\}$ 
17:      if  $m_2 = 0$  then
18:         $cDepth[B, c] := 1$ 
19:      else
20:        for all  $v \in B$  do
21:           $w'_B(v) := \begin{cases} 1 & \text{if } v \in M_2 \\ 0 & \text{if } v \notin M_2 \end{cases}$ 
22:        end for
23:        COMPUTEEDGEWEIGHTS0( $B, w'_B$ )
24:         $w_{\max} := \text{CONSTRAINTMAXFACEBLOCK}(B, w'_B, c)$ 
25:         $cDepth[B, c] := \begin{cases} m_2 & \text{if } \ell = |M_2| \\ m_2 + 2 & \text{otherwise} \end{cases}$ 
26:      end if
27:    end if
28:

```

▷ *continued on next page...*

Listing 5.8: Function MINIMUMDEPTHREC (part 1).

```

29:   for all  $c \rightarrow B' \in \mathcal{B}$  do
30:        $maxCDepth_B[c] := \max\{cDepth[B'', c] \mid \{B'', c\} \in \mathcal{B}, B' \neq B''\}$ 
31:        $(B'', m') := \text{MINIMUMDEPTHREC}(B')$ 
32:       if  $m' < m^*$  then  $(B^*, m^*) := (B'', m')$ 
33:   end for
34: end for
35: return  $(B^*, m^*)$ 
36: end function

```

Listing 5.9: Function MINIMUMDEPTHREC (part 2).

The following table summarizes the important variables used in Listings 5.7 through 5.9.

variable	purpose
$maxCDepth_B[v]$	Maximum of all minimum block-nesting depths of the graphs $G_{c,B'}$, $B' \neq B$ with v on the external face, or 0 if v is not a cut-vertex.
$cDepth[B, c]$	Minimum block-nesting depth of graph $G_{c,B}$ with c on the external face.

5.3 Minimum Block-Nesting Depth and Maximum External Face

Since MAXIMUMFACE and MINIMUMDEPTH proceed in a similar fashion, it is possible to combine both algorithms to a new algorithm MINDEPTHMAXFACE that computes a planar embedding with maximum external face among all planar embeddings with minimum block-nesting depths. This can be achieved by using a pair (d, w) as weight attribute, where the first component d refers to the 0/1 weight attribute used in MINIMUMDEPTH and w refers to the weight attribute used in MAXIMUMFACE. An edge in G has simply weight $(0, 1)$. The linear order on these pairs is defined lexicographically, that is,

$$(d, w) > (d', w') \iff d > d' \text{ or } (d = d' \text{ and } w > w')$$

Each time a maximum face in a block has to be computed, we first determine a maximum face according to the linear order defined above. If the resulting maximum face has weight (d^*, w^*) and d^* is the number of cut-vertices we want to place in a common face (for example, $d^* = |M_B|$ as in Lemma 5.5), we also have found a planar embedding with a maximum face among all planar embeddings with minimum block-nesting depth. Otherwise, the value d^* is irrelevant, since all extensions will have the same block-nesting depth and there might be planar embeddings with a larger external face. Therefore, we call the procedure

for finding a maximum face in a block again, this time respecting only the second component of the weight attribute.

Theorem 5.5. *Let $G = (V, E)$ be a planar and connected graph. Then, Algorithm $\text{MINDEPTHMAXFACE}(G)$ computes a planar embedding of G with maximum external face among all planar embeddings of G with minimum block-nesting depth in time $\mathcal{O}(|V| + |E|)$.*

5.4 Experimental Analysis

Pizzonia [2005] has conducted an experimental study on the effect of minimizing the block-nesting depth with respect to well-known quality measures like drawing area, number of bends, and total edge length. However, the study only focuses on the algorithm by Pizzonia and Tamassia [2000] for minimizing the block-nesting depth with fixed embeddings of blocks; it does not include the new algorithms introduced in this chapter. On the other hand, the Diploma thesis by Kerkhof [2007] (supervised by Petra Mutzel and Carsten Gutwenger) compares all these algorithms as well as some further extensions; furthermore, all algorithms are implemented in OGDF [Chimani et al., 2010]. We present some of the results in the following.²

Besides the Rome graphs, the study uses a newly generated benchmark set of graphs with many blocks, called the *Block graphs*. The latter benchmark set consists of 20 groups $i = 1, \dots, 20$, such that the graphs in each group i contain (at most) i blocks. For each group, 100 graphs have been generated, where each block is a planar graph with up to 30 vertices.

We consider 6 embedding algorithms:

- **SIMPLE**: the PQ-tree based planar embedding algorithm, where a largest face is used as external face.
- **MINDEPTHFIXBLOCKS**: the algorithm by Pizzonia and Tamassia [2000].
- **MAXIMUMFACE**: the embedding algorithm computing an embedding with maximum external face (see Section 5.1).
- **MINIMUMDEPTH**: the embedding algorithm computing an embedding with minimum block-nesting depth (see Section 5.2).
- **MINDEPTHMAXFACE**: the combination of maximum external face and minimum block-nesting depth (see Section 5.3).
- **MAXFACELAYERS**: an extension of the maximum external face algorithm [see Kerkhof, 2007]; inner faces are embedded such that the cycle we get after removing the external face is again as large as possible, and so forth.

²We thank Thorsten Kerkhof for providing the data.

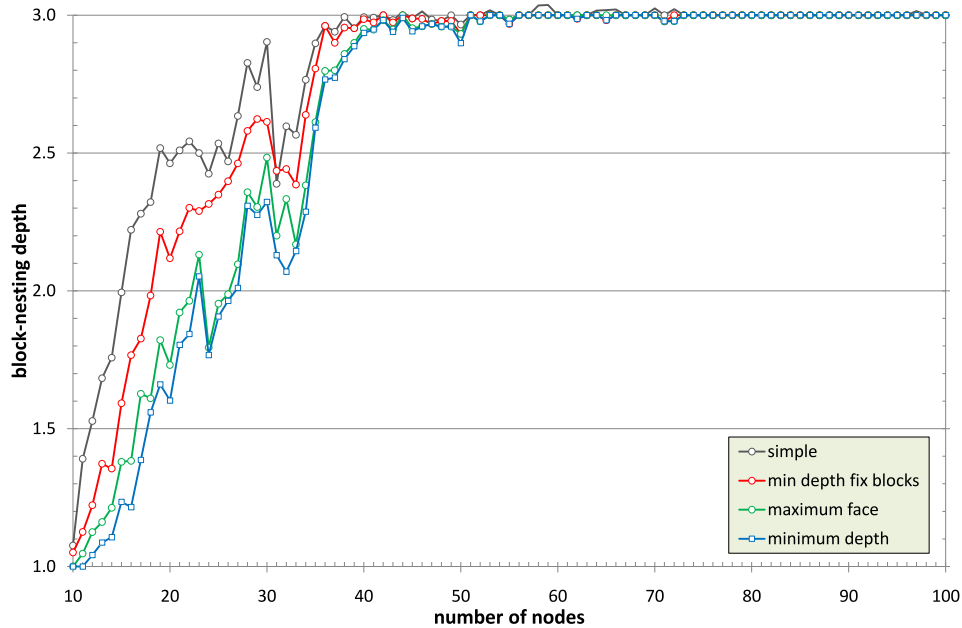


Figure 5.7: Average block-nesting depth (Rome graphs).

As drawing algorithm, we used the orthogonal layout algorithm in OGDF, which is based on a network-flow model for bend minimization combined with an edge routing heuristic for routing edges to high-degree vertices, and a sophisticated flow-based compaction phase.

5.4.1 The Rome Graphs

Figure 5.7 shows the average achieved block-nesting depth for the Rome graphs. Interestingly, all algorithms obtain an average block-nesting depth of about 3 for all graphs with 50 and more vertices. This seems to be caused by the special structure of the Rome graphs; moreover, we cannot expect a significant influence of minimizing the block-nesting depth for these graphs. For the smaller graphs, `MINDEPTHFIXBLOCKS` performs much worse than `MINIMUMDEPTH` and even `MAXIMUMFACE`, caused by the requirement to fix the embeddings of blocks.

Considering the important aesthetic criterion of total edge length (Figure 5.8), we observe that maximizing the external face results in a good improvement for smaller graphs with about 10–12% shorter edges; the effect is even better for `MAXFACELAYERS`. For larger graphs, the relative difference to `SIMPLE` decreases. In general, `MINDEPTHFIXBLOCKS` performs worst, even worse than `SIMPLE`.

Another important criterion is the area used by the drawing; see Figure 5.9. We distinguish between the area of the drawing's bounding box and the area used by all inner faces. We observe, that the effect of the choice of the embedding has a bigger effect on the area occupied by inner faces. Especially for the smaller graphs, we achieve improvements by about 20–25% on average; obviously, maximizing the external face is the essential criterion here.

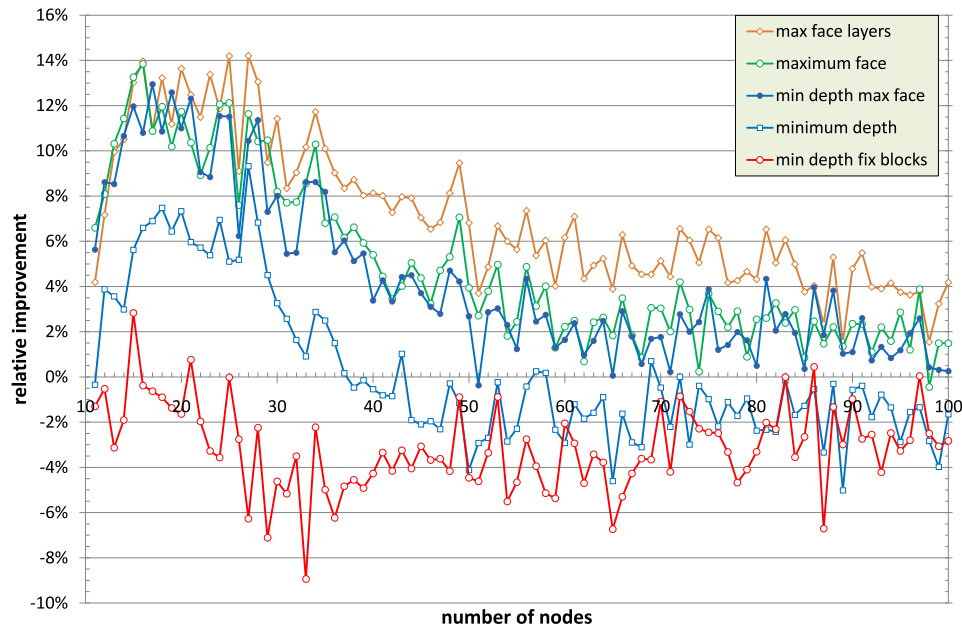
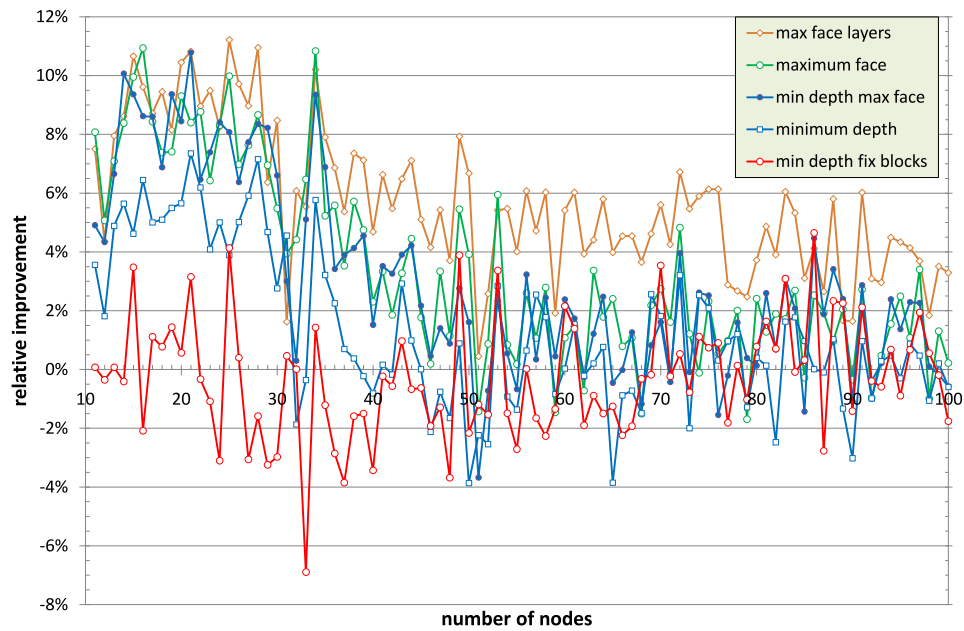


Figure 5.8: Average relative improvement for total edge length compared to simple embedding (Rome graphs).

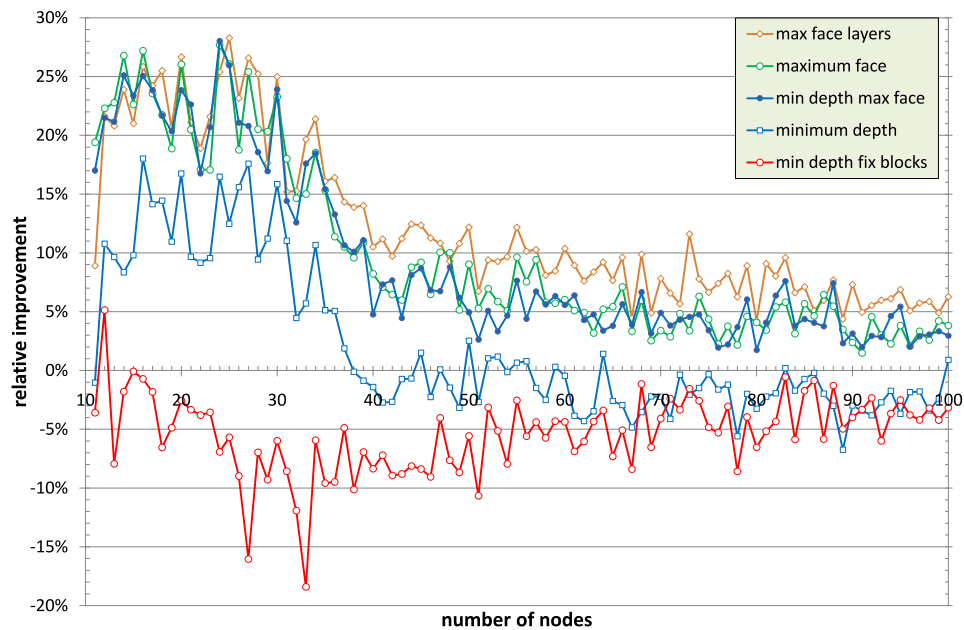
5.4.2 The Block Graphs

The special structure of the Rome graphs did not really allow us to study the effect of minimizing the block-nesting depth. For the Block graphs, the average block-nesting depth obtained by the various algorithms is much more diverse; see Figure 5.10. All algorithms perform significantly better than SIMPLE; MAXIMUMFACE and MINDEPTHFIXBLOCKS are more or less equal, with a gap of about 0.5 to the optimum MINIMUMDEPTH.

Figure 5.11 shows that a large external face leads to much less bends than, for example, just minimizing the block-nesting depth. The former algorithms save up to 15 bends on average for the larger graphs compared to SIMPLE; the algorithms minimizing the block-nesting depth only save up to about 4 bends, which is more or less marginal. Looking at the average total edge lengths (Figure 5.12), we see two clear winners, namely MAXFACELAYERS and MINDEPTHMAXFACE, requiring about 15% less than SIMPLE. It is interesting to observe that the combination of maximizing the external face and minimizing the block-nesting depth performs clearly better than just optimizing only one of these properties. Again, MINDEPTHFIXBLOCKS brings up the rear, but still reaches a good improvement of about 5–10%. Our findings are confirmed by the results for the area of inner faces. MAXFACELAYERS and MINDEPTHMAXFACE are again the winners, with a remarkable improvement of about 30%.



(a) area (bounding box)



(b) area (inner faces)

Figure 5.9: Average relative improvement for area of bounding box and inner faces compared to simple embedding (Rome graphs).

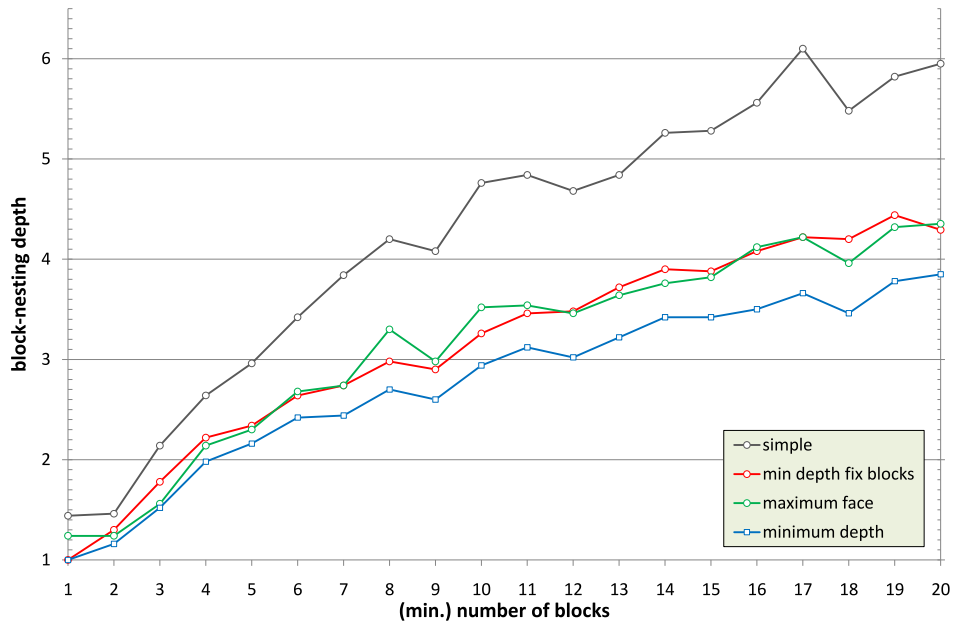


Figure 5.10: Average block-nesting depth (Block graphs).

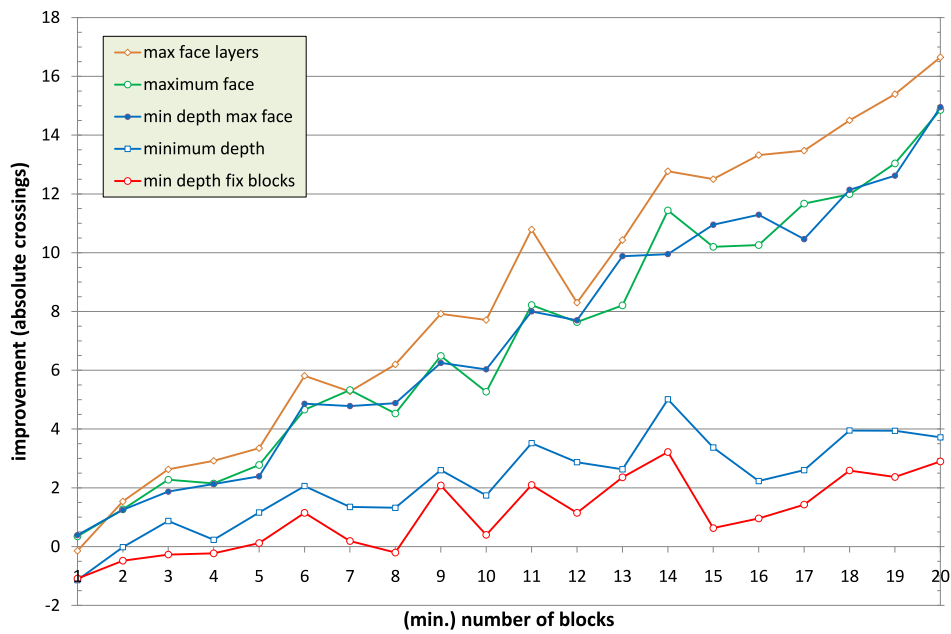


Figure 5.11: Average absolute improvement for number of bends (Block graphs).

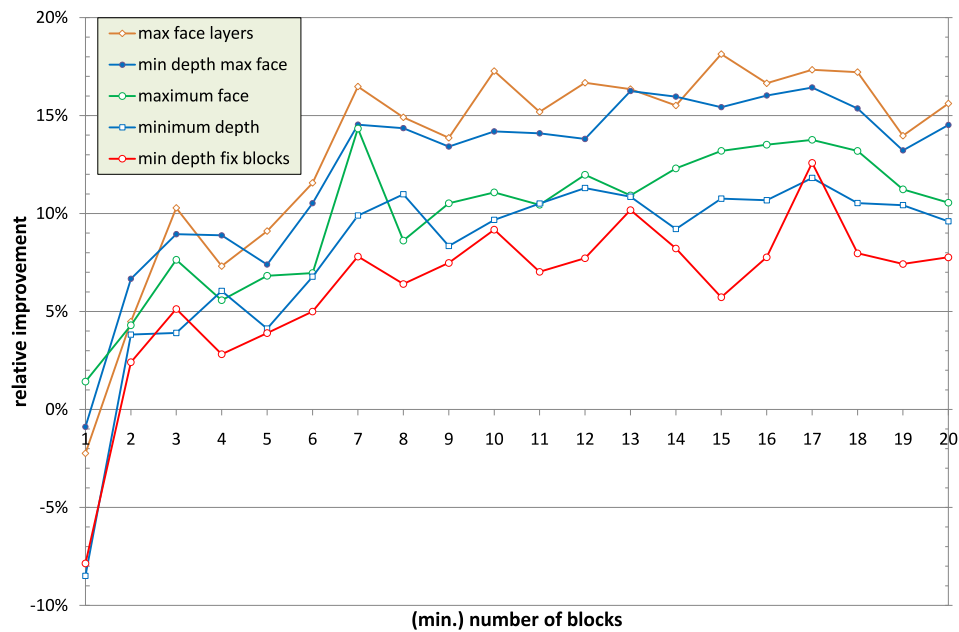


Figure 5.12: Average relative improvement for total edge length compared to simple embedding (Block graphs).

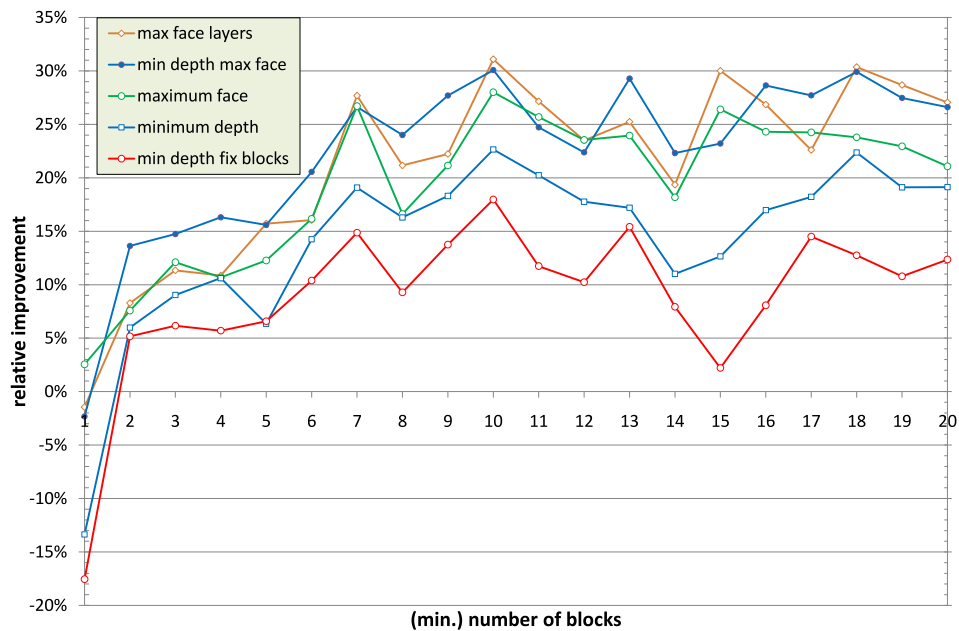


Figure 5.13: Average relative improvement for area of inner faces (Block graphs).

5.4.3 Summary

Our experimental study has shown that the choice of the embedding can have a considerable influence on the quality of the resulting drawing when applying the topology-shape-metrics approach. While the larger Rome graphs seem to have some kind of special structure diminishing the influence of the embedding to some amount, experiments with graphs from the Block library prove that the choice of the embedding is also important for large graphs; for the Block graphs the relative improvement achieved by the advanced embedding algorithms stays almost constant as the graphs get larger.

Maximizing the degree of the external face turns out to be more important if we want to have a small number of bends, short edges, or a small area. However, we think that a small block-nesting depth improves readability, since it reveals the structure of the graph in a better way. Together with the observation that the combination of a large external face and a small block-nesting depth sometimes performs even significantly better, we recommend to use `MINDEPTHMAXFACE` as embedding algorithm in the topology-shape-metrics approach; a very good alternative is `MAXFACELAYERS`, which can also be combined with minimizing the block-nesting depth; see [Kerkhof, 2007].

Chapter 6

Embedding Constraints

As far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality.

ALBERT EINSTEIN (1879 – 1955)

In many application domains information visualization is based on graph representations. Examples include software engineering, database modeling, business process modeling, VLSI-design, and bioinformatics. The computation of concise graph layouts by automatic layout systems facilitates the readability and immediate understanding of the displayed information. However, these layout systems need to take into account application specific as well as user-defined layout rules in addition to the aesthetic criteria we typically try to optimize in graph drawing. In database diagrams, for example, links between attributes should enter the tables only at the left or right side of the corresponding attributes, the placement of reactants in chemical reactions or biological pathways should reflect their role within the displayed reactions, and in UML class diagrams, generalization edges should leave a class object at the top and enter a base class object at the bottom. Many of these layout rules impose restrictions on the admissible embeddings for a drawing. Even more important is the possibility to use drawing restrictions in order to express the user's preferences and to guide the layout phase. A general survey of constraints in graph drawing algorithms is given by Tamassia [1998].

In this chapter, we consider restrictions on the allowed order of incident edges around a vertex, for example, to specify groups of edges that have to appear consecutively around the vertex or that have a fixed clockwise order in any admissible embedding. Such constraints occur, for example, in the form of *side constraints*, where incident edges are assigned to the four sides of a rectangular vertex, or *port constraints* where edges have prescribed attachment points at a vertex. In particular, we introduce three types of constraints which may be arbitrarily nested: *grouping*, *oriented* (prescribed clockwise order), and *mirror* constraints (prescribed reversible order). We call a planar embedding that fulfills the given set of constraints an *ec-planar* embedding.

Furthermore, it is desirable to integrate embedding constraint into the planarization approach for computing graph layouts with few crossings. The first step can be solved by successive ec-planarity testing. Our first contribution to embedding constraints therefore is a linear time algorithm for testing if a graph with a set of embedding constraints is ec-planar; see Section 6.3. The main challenge here is to incorporate oriented constraints, where a given clockwise order of (groups of) incident edges needs to be satisfied. Furthermore, we characterize all possible ec-planar embeddings using BC- and SPQR-trees, which also yields a linear time algorithm for computing an ec-planar embedding.

The second step of the planarization approach can be tackled by repeatedly solving the one-edge insertion problem, as introduced in Section 4.1 for the unconstrained case. The algorithm presented there essentially computes a shortest path Ψ between those nodes in the SPQR-tree \mathcal{T} of G whose skeletons contain v and w , respectively. The optimal insertion path is then constructed by simply concatenating locally optimal insertion paths of the tree nodes on Ψ . However, if embedding constraints have to be observed, that is, restrictions on the order of the edges around the vertices of G are given, locally optimal solutions need not lead to globally optimal solutions and the greedy approach cannot be applied anymore. The best local decision now depends on the decisions for other parts of the edge insertion path. Our second contribution is thus a linear-time algorithm to solve the *one-edge insertion problem with embedding constraints* (see Section 6.4): Given an ec-planar graph G with an additional edge e and a set of embedding constraints C for the graph $G + e$, compute an *ec-planar* embedding of G together with a crossing minimal edge insertion path for e that observes C .

Even though constraint handling is an important issue because of its relevance in practical applications, for example, in interactive graph drawing (see, for example, [Böhringer and Paulisch, 1990, North, 1996, Brandes and Wagner, 1997, Brandes et al., 2002]), there is only few previous work concerning constraints on the admissible embeddings of a graph. Di Battista et al. [2002] consider embedding constraints that appear in database schemas, where table attributes are arranged from top to bottom within a rectangular vertex representing a table, and links connecting attributes may attach at the left or right hand side of these attributes. The integer linear programming approach in [Eiglsperger et al., 2000] considers side constraints in the shape computation phase of orthogonal graph drawing. Dornheim [2002] studies the problem of computing embeddings satisfying topological constraints that consist of a cycle together with two sets of edges that have to be embedded inside or outside the cycle, respectively. Buchheim et al. [2006] describe how to adapt the planarization approach for directed graphs when incoming and outgoing edges have to appear consecutively around each vertex. Recently, Angelini et al. [2010a] presented a linear-time algorithm to test if a given embedding of a subgraph of G can be extended to an embedding of the entire graph G .

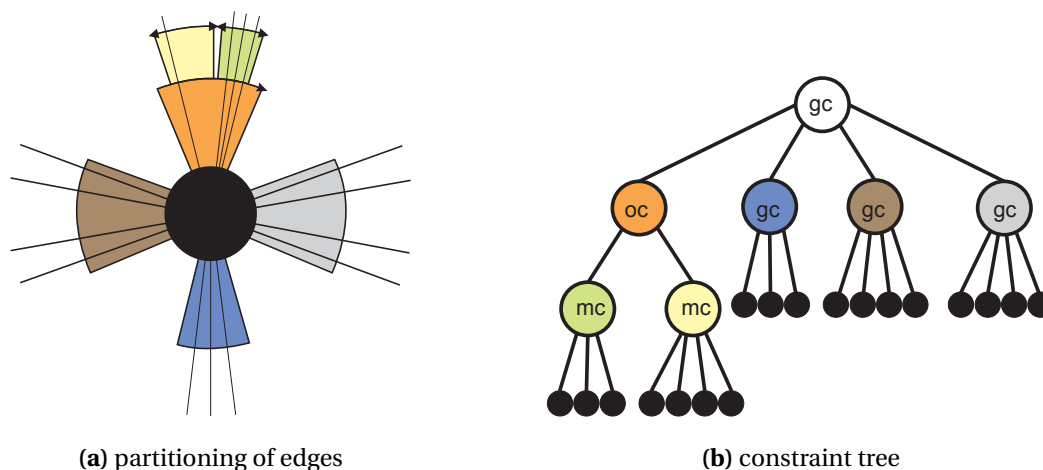


Figure 6.1: The hierarchical partitioning of edges imposed by an embedding constraint (a) and the corresponding constraint tree (b).

6.1 *ec-Constraints and ec-Planarity*

Let $G = (V, E)$ be a graph. An embedding constraint specifies the admissible clockwise order of the edges incident to a vertex in a combinatorial embedding of G . Here, we consider the case where a vertex has at most one embedding constraint and either all or none of the edges incident to a vertex are subject to embedding constraints.

An *embedding constraint* at a vertex $v \in V$ is a rooted, ordered tree T_v such that its leaves are exactly the edges incident to v . The inner nodes of T_v , also called *constraint-nodes* or *c-nodes* for short, are of three types: *oc-nodes* (oriented constraint-nodes), *mc-nodes* (mirror constraint-nodes), and *gc-nodes* (grouping constraint-nodes). Since T_v is an ordered tree, it imposes an order on its leaves and thus on the edges incident to v . We consider this order as a cyclic order and represent all *admissible* cyclic, clockwise orders of the edges incident to v by defining how the order of the children of c-nodes in T_v can be changed:

- *gc-node*: The order of children may be arbitrarily permuted.
- *mc-node*: The order of children may be reversed.
- *oc-node*: The order of children is fixed.

Figure 6.1 shows an example for an embedding constraint. A c-node with a single child is obviously redundant, therefore we demand that each c-node has at least two children. While gc- and mc-nodes alone resemble the concept of PQ-trees [Booth and Lueker, 1976], the additional concept of oc-nodes is necessary to model important constraints like, e.g., side constraints, and significantly complicates planarity testing.

Let C be a set of embedding constraints at distinct vertices of G . A combinatorial embedding Γ of G *observes* the embedding constraints in C , if for each

embedding constraint $T_v \in C$, the cyclic clockwise order of the edges around v in Γ is admissible with respect to T_v . A planar embedding observing the embedding constraints in C is an *ec-planar embedding* with respect to C , and (G, C) is *ec-planar*, if there exists an ec-planar embedding of G with respect to C .

6.2 ec-Expansion

A basic building block of the ec-planarity test is a structural transformation applied to a given graph G with embedding constraints C . For each embedding constraint T_v at vertex v , this transformation expands v according to the structure of T_v . We call the resulting graph the *ec-expansion* $E(G, C)$ of G with respect to C . The details of this transformation are given below.

6.2.1 Construction of the ec-Expansion

The *ec-expansion* $E(G, C)$ of G with respect to C is constructed as follows. Let $T_v \in C$ be an embedding constraint and T'_v the subgraph obtained from T_v by omitting its leaves. Recall that the leaves of T_v are exactly the edges incident to v . We replace v in G by the tree T'_v and connect the edges incident with v with the parents of the corresponding leaves. This transformation introduces a vertex in G for every c-node in T_v . Each vertex u corresponding to an oc- or mc-node is further replaced by a *wheel gadget* which is a wheel graph with $2d$ spokes, where e_1, \dots, e_d are the edges incident to u . Then, the respective wheel gadget consists of a cycle $x_1, y_1, \dots, x_d, y_d$ of length $2d$ and a vertex, called *hub*, incident to every vertex on the cycle; see Figure 6.2(a). The vertex u is replaced by this wheel gadget, such that e_i is connected to x_i for $1 \leq i \leq d$. According to the type of the expanded c-node, we distinguish between *O-hubs* (oc-nodes) and *M-hubs* (mc-nodes). We refer to the edges introduced during the ec-expansion as *expansion edges*. Figure 6.2(b) shows the expansion of a vertex according to the constraint tree shown in Figure 6.1(b).

The purpose of the wheel gadgets is to model the fixed order of the children of the corresponding c-node. Since a wheel gadget is a triconnected graph, it admits only two combinatorial embeddings that are mirror images of each other. The order in which non-gadget edges are attached to the wheel cycle is either the order given by the corresponding c-node, or the reverse order. Every face adjacent to the hub is a triangle. We call these faces *inner wheel gadget faces*.

Lemma 6.1. *Let $G = (V, E)$ be a graph with embedding constraints C . Then, its ec-expansion $E(G, C)$ has size $\mathcal{O}(|V| + |E|)$ and can be constructed in time $\mathcal{O}(|V| + |E|)$.*

Proof. Consider an embedding constraint $T_v \in C$. Since the leaves of T_v are in one-to-one correspondence to the edges incident to v and each c-node has at least two children, the size of T_v is linear in $\deg(v)$. We replace each oc- and mc-node μ by a wheel gadget with $4 \deg(\mu)$ edges. Thus, the expansion of vertex v creates $\mathcal{O}(\deg(v))$ edges, and the total number of additional edges in $E(G, C)$ is

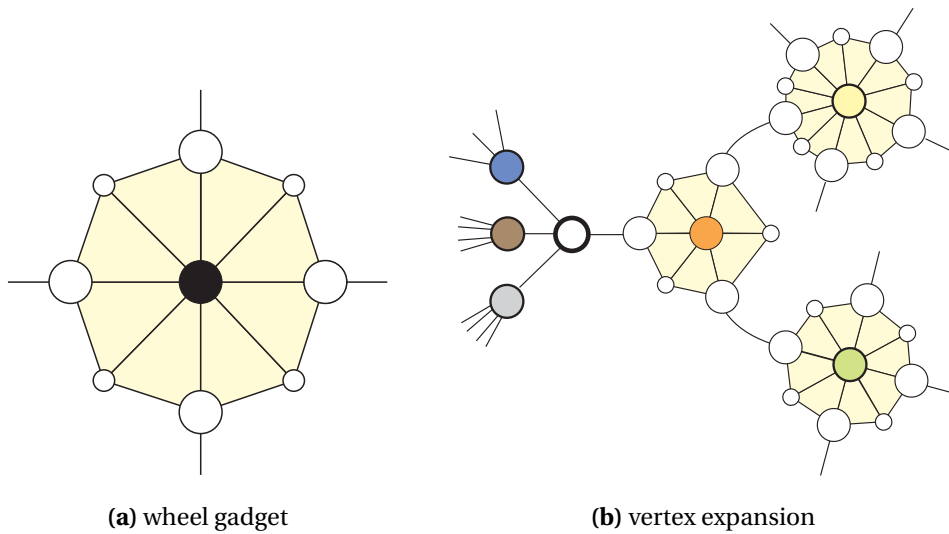


Figure 6.2: Expansion gadgets: (a) a wheel gadget replacing a vertex with degree 4; (b) vertex expansion according to the constraint tree in Figure 6.1(b) (the thick hollow vertex is the root).

bounded by $\sum_{v \in V} \mathcal{O}(\deg(v)) = \mathcal{O}(|E|)$. Therefore, the size of the expansion graph is $\mathcal{O}(|V| + |E|)$, and the expansion can obviously be computed in $\mathcal{O}(|E(G, C)|) = \mathcal{O}(|V| + |E|)$ time. \square

6.2.2 *ec-Expansion and ec-Planar Embeddings*

In this section we discuss the relationship between planar embeddings of the *ec-expansion* $E(G, C)$ and *ec-planar embeddings* of (G, C) . Though the *ec-expansion* serves as a tool for modeling the embedding constraints in C , a planar embedding of $E(G, C)$ needs to fulfill certain conditions in order to induce an *ec-planar embedding* of G with respect to C . We call a planar embedding Γ of $E(G, C)$ *ec-planar* if

- (a) the external face of Γ does not contain a hub;
- (b) every face incident to a hub is a triangle consisting solely of edges of the corresponding wheel gadget; and
- (c) each O-hub h is *oriented correctly*, that is, the cyclic, clockwise order of the edges around h in Γ corresponds to the order specified by the corresponding *oc-node*.

Let Γ be an *ec-planar embedding* of $E(G, C)$. We obtain an *ec-planar embedding* of (G, C) as follows. For each vertex v with corresponding embedding constraint in C , there is a connected subgraph G_v in $E(G, C)$ resulting from expanding v . Let $\tilde{G}_v \subset E(G, C)$ be the graph induced by the vertices not contained in G_v . The conditions above assure that the planar embedding Γ_v of G_v induced

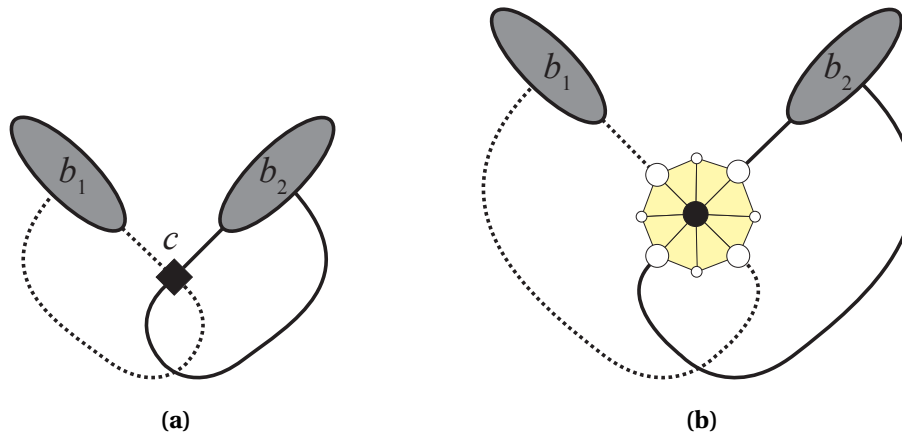


Figure 6.3: (a) a crossing required between edges of two different blocks caused by embedding constraints; (b) the expansion using a wheel gadget merges the two blocks into a single one (b).

by Γ is such that \bar{G}_v lies in the external face of Γ_v . The edges that connect G_v to \bar{G}_v correspond to the edges incident to v in G . Their cyclic clockwise order around G_v is admissible with respect to T_v , since the wheel gadgets fix the order of the edges specified by oc- and mc-nodes, and O-hubs are oriented correctly. We shrink G_v to a single vertex by contracting all edges in G_v while preserving the embedding, thus resulting in an admissible order of the edges around v .

If we have an ec-planar embedding of (G, C) , then the edges around each vertex v are ordered such that the constraints in T_v are fulfilled. It is easy to see that we can replace each such vertex v by the expansion graph corresponding to T_v in such a way that we obtain an ec-planar embedding of $E(G, C)$. Thus, we get the following result:

Lemma 6.2. *Let G be a graph with embedding constraints C . Then, (G, C) is ec-planar if and only if $E(G, C)$ is ec-planar. Moreover, every ec-planar embedding of $E(G, C)$ induces an ec-planar embedding of (G, C) .*

6.3 ec-Planarity Testing

It is well-known that planarity testing can be reduced to biconnected graphs, that is, it is sufficient to test the blocks of a graph independently. However, adding embedding constraints complicates this task. Let G be a graph with embedding constraints C . Consider a cut vertex c in G that connects two blocks b_1 and b_2 via the edge sets S_1 and S_2 , respectively; see Figure 6.3(a). If these edge sets are subject to embedding constraints that force the edges in S_1 and S_2 to be intermixed as in Figure 6.3(a), then the given graph is not ec-planar even if its blocks are ec-planar. We solve this problem by first applying the ec-expansion to the graph. This replaces the cut vertex c by a wheel gadget so that c does not separate b_1 and b_2 anymore; see Figure 6.3(b).

By Lemma 6.2, we know that it is sufficient to test the ec-expansion $E(G, C)$ for ec-planarity. In contrast to the graph G itself, the following lemma shows that we can test the blocks of $E(G, C)$ separately.

Lemma 6.3. *$E(G, C)$ is ec-planar if and only if every block of $E(G, C)$ is ec-planar.*

Proof. If $E(G, C)$ is ec-planar, then there is an ec-planar embedding of $E(G, C)$, and this embedding implies an ec-planar embedding for each block of $E(G, C)$.

Suppose now that each block of $E(G, C)$ is ec-planar. Consider a wheel gadget \mathcal{G} in $E(G, C)$. Since \mathcal{G} is triconnected, \mathcal{G} is completely contained in a single block B of $E(G, C)$. For each edge $(u, v) \in \mathcal{G}$, the pair $\{u, v\}$ is not a separation pair in B by construction, hence every inner wheel face of \mathcal{G} is also a face in every planar embedding of B . Moreover, the hub of \mathcal{G} is not a cut vertex of $E(G, C)$, since all its incident edges are in B .

We construct an ec-planar embedding of $E(G, C)$ as follows. We start with an arbitrary block B of $E(G, C)$. Let Π be an ec-planar embedding of B . In particular, the external face of Π is not an inner wheel face of a wheel gadget. We add the remaining blocks successively to Π . Let B' be another block of $E(G, C)$ that shares a vertex c with B , and let Π' be an ec-embedding of B' . We pick faces $f \in \Pi$ and $f' \in \Pi'$ that are adjacent to c and not inner wheel faces of a wheel gadget. This is possible, since the only vertices adjacent solely to inner wheel faces are the O- and M-hubs. Then, we insert Π' with f' as external face into the face f of Π . This results in an ec-planar embedding of $B \cup B'$. We can add the remaining blocks (if any) in the same way, resulting in an ec-planar embedding of $E(G, C)$. \square

If we can characterize all ec-planar embeddings of the blocks of $E(G, C)$, the construction in the proof of Lemma 6.3 also shows us how to enumerate all ec-planar embeddings of $E(G, C)$ by traversing its BC-tree. In the following, we devise such a characterization. Let B be a block of $E(G, C)$ and \mathcal{T} its SPQR-tree.

Observation 6.1. *Every wheel gadget \mathcal{G} is completely contained within the skeleton of an R-node. In particular, the hub of \mathcal{G} occurs only in the skeleton of a single R-node.*

Proof. \mathcal{G} is triconnected, and for each edge $(u, v) \in \mathcal{G}$, the pair $\{u, v\}$ is not a separation pair in B by construction. Therefore, all edges of \mathcal{G} occur in the same skeleton graph, which must be the skeleton of an R-node μ . The hub h of \mathcal{G} is only incident to edges of \mathcal{G} and no other edge of B , hence h occurs only in $\text{skeleton}(\mu)$. \square

If B is planar, then the skeleton of an R-node is a triconnected planar graph, thus having exactly two planar embeddings which are mirror images of each other. We call two O-hubs contained in the same skeleton S *conflicting* if none of the two planar embeddings of S orients both O-hubs correctly. The following theorem gives us an easy to check condition for ec-planarity and characterizes all possible ec-planar embeddings:

```

1: function ISECPANAR(Graph  $G$ , Constraints  $C$ )
2:   Construct ec-expansion  $E$  of  $(G, C)$ .
3:   if  $E$  is not planar then return false
4:   for each block  $B$  of  $E$  do
5:     Construct SPQR-tree  $\mathcal{T}$  of  $B$ .
6:     for each R-node  $\mu \in \mathcal{T}$  do
7:       if  $\text{skeleton}(\mu)$  contains two conflicting O-hubs then
8:         return false
9:       end if
10:    end for
11:  end for
12:  return true
13: end function

```

Listing 6.1: Ec-planarity testing.

Theorem 6.1. *Let G be a graph with embedding constraints C . Let B be a block of $E(G, C)$ and \mathcal{T} its SPQR-tree. Then, the following holds:*

- (a) *B is ec-planar if and only if B is planar and no skeleton of an R-node of \mathcal{T} contains conflicting O-hubs.*
- (b) *If B is ec-planar, then the embeddings of the skeletons of \mathcal{T} induce an ec-planar embedding of B if and only if each O-hub in the skeleton of an R-node is oriented correctly.*

Proof. If B admits an ec-planar embedding, then this embedding induces embeddings of the skeletons of \mathcal{T} such that every O-hub in the skeleton of an R-node is oriented correctly. In particular, no R-node skeleton contains conflicting O-hubs.

Suppose now that B is planar and no R-node skeleton contains conflicting O-hubs. For each R-node skeleton containing at least one O-hub, we can choose planar embeddings such that all O-hubs are oriented correctly within the skeletons. We have to show that the embeddings of the skeletons induce an ec-planar embedding of B , even if we choose arbitrary embeddings for the remaining skeletons. This holds, since every such embedding Π has the property that each O-hub is oriented correctly because wheel gadgets are completely contained within R-node skeletons by Observation 6.1, and inner wheel faces are preserved. We can pick any face of Π as external face which is not an inner wheel face (such a face always exists) and obtain an ec-planar embedding of B . \square

Algorithm ISECPANAR depicted in Listing 6.1 applies Theorem 6.1 and devises a linear time ec-planarity test, which can easily be extended so that it computes an ec-planar embedding as well.

Theorem 6.2. *Let $G = (V, E)$ be a graph with embedding constraints C . Then, Algorithm ISECPANAR tests (G, C) for ec-planarity in time $\mathcal{O}(|V| + |E|)$. Moreover,*

if (G, C) is ec-planar, an ec-planar embedding of (G, C) can also be computed in time $\mathcal{O}(|V| + |E|)$.

Proof. By Lemma 6.2 and 6.3, it is sufficient to test every block of $E(G, C)$ for ec-planarity. Hence, the correctness of Algorithm `ISECPLANAR` follows from Theorem 6.1.

Constructing the ec-expansion (Lemma 6.1) and testing planarity can be done in linear time. For each block B of $E(G, C)$, we construct its SPQR-tree, which requires linear time in the size of B ; see Section 3.4. The check for conflicting O-hubs is easy to implement: For each R-node skeleton S , we compute a planar embedding of S . If this embedding contains both correctly as well as not correctly oriented O-hubs, then there is a conflict, otherwise not. Since the total size of skeleton graphs is linear in the size of B and a planar embedding can be found in linear time, we need linear running time for each block. Hence, the total running time is linear in the size of $E(G, C)$ which is $\mathcal{O}(|V| + |E|)$ by Lemma 6.1.

In order to find an ec-planar embedding of G , we just have to compute embeddings of the skeleton graphs for each block as described in Theorem 6.1 and combine the embeddings as described in the proof of Lemma 6.3. \square

6.4 ec-Edge Insertion

6.4.1 ec-Edge Insertion Paths and ec-Traversing Costs

We first generalize the terms insertion path and traversing costs introduced in Section 4.1. Intuitively, the edges in an insertion path are the edges we need to cross when inserting an edge (x, y) into an embedding. Let $G + (x, y)$ be a graph with embedding constraints C . An *ec-edge insertion path* for (x, y) in an ec-planar embedding Π of G is a sequence of edges e_1, \dots, e_k of G satisfying the following two conditions:

1. There is a face $f_x \in \Pi$ with $x, e_1 \in f_x$, a face $f_y \in \Pi$ with $e_k, y \in f_y$, and faces $f_i \in \Pi$ with $e_i, e_{i+1} \in f_i$ for $1 \leq i < k$.
2. The edge order around x and y is admissible with respect to C if (x, y) leaves x via face f_x and enters y via face f_y .

Finding a shortest ec-insertion path in a fixed embedding Π is easy: We only need to identify the set of faces F_x incident to x where the insertion path may start, and F_y incident to y where it may end, and then find a shortest path in the dual graph of Π connecting a face in F_x with a face in F_y .

We are interested in the shortest possible ec-insertion path among all ec-planar embeddings of G , which we also call an *optimal ec-insertion path* in G . In particular, we need to identify the required ec-planar embedding of G . In order to represent all ec-planar embeddings of G , we apply Lemma 6.2 and use its ec-expansion instead. More precisely, we use the subgraph $K = E(G + (x, y), C) \setminus e$, where $e = (v, w)$ is the edge of $E(G + (x, y), C)$ connecting the expansion of x

with the expansion of y . An ec-insertion path in an ec-planar embedding of K is defined as before with the only difference that we replace the second condition with

- 2'. e_1, \dots, e_k contains no expansion edge of K .

It is easy to see that we can also use this definition for a subgraph B of K and two distinct vertices of B that are not hubs.

We adapt the notion of traversing costs defined to ec-planarity. Let e be a skeleton edge, and let Π be an arbitrary ec-embedding of the graph $\text{expansion}^+(e)$ with dual graph Π^* , in which all edges corresponding to gadget edges have length ∞ and the other edges have length 1. Let f_1 and f_2 be the two faces in Π separated by e . We denote with $P(\Pi^*, e)$ the length of the shortest path in Π^* that connects f_1 and f_2 and does not use the dual edge of e . Hence, we have $P(\Pi^*, e) \in \mathbb{N} \cup \{\infty\}$.

The following lemma follows analogously to Lemma 4.1.

Lemma 6.4. *Let μ be a node in \mathcal{T} and e an edge in $\text{skeleton}(\mu)$. Then, $P(\Pi^*, e)$ is independent of the ec-embedding Π of $\text{expansion}^+(e)$.*

Proof. Let m be the number of edges in $G_e = \text{expansion}^+(e)$ and G'_e be the graph obtained from G_e by replacing each gadget edge with $m + 1$ parallel edges. Then, each embedding Π of G_e corresponds to an embedding Π' of G'_e , and $P(\Pi^*, e)$ is ∞ if and only if the corresponding path in Π' is longer than m . Lemma 4.1 shows that for the general case, that is, without embedding constraints, $P(\Pi^*, e)$ is independent of the embedding Π . Applying this lemma and observing that the ec-embeddings of G_e are a non-empty subset of the embeddings of G_e yields the lemma. \square

Thus, we define the *ec-traversing costs* $c(e)$ of a skeleton edge e as $P(\Pi^*, e)$ for an arbitrary ec-embedding Π of $\text{expansion}^+(e)$. We formally define the one-edge insertion problem with embedding constraints as follows:

ONE-EDGE INSERTION PROBLEM WITH EMBEDDING CONSTRAINTS	
Instance:	an ec-planar graph $G = (V, E)$, two distinct vertices $x, y \in V$, a set of embedding constraints C for $G + (x, y)$
Solution:	an ec-planar embedding Π of (G, C) and an ec-edge insertion path p for (x, y)
Minimize:	the length of p

6.4.2 The Algorithm for Biconnected Graphs

The hard part is to find an ec-insertion path in a block B of K . Our task is to compute an optimal ec-insertion path between two nodes v, w of B . Algorithm `OPTIMALECBLOCKINSERTER` in Listing 6.2 and 6.3 solves this problem. In this algorithm, we use the notation $\min_{i,n} \Lambda$ which returns a tuple in the set Λ of n -tuples whose i -th component is minimal among all tuples in Λ .

OPTIMALECBLOCKINSERTER is called with a block B of an ec-planar ec-expansion and two distinct vertices v and w of B . Since we assume that B contains all gadget edges, we do not need to pass further constraint information for the edge (v, w) . In particular, using any insertion path in any ec-planar embedding of B that connects v and w and does not cross a gadget edge yields an ec-embedded planarization of $B \cup (v, w)$. Hence, we look for an ec-embedding of B that allows the insertion of the edge (v, w) with the minimum number of crossings.

First, we compute the SPQR-tree \mathcal{T} of B and embed the skeletons such that they imply an ec-embedding of B , that is, the R-node skeletons are embedded correctly. Then, the shortest path $\Upsilon := \mu_1, \dots, \mu_k$ between an allocation node μ_1 of v and μ_k of w is identified. In order to achieve a consistent orientation, we root \mathcal{T} such that Υ is a descending path in the tree, that is, μ_i is the parent of μ_{i-1} for $i = 2, \dots, k$. Note that the rooting of the SPQR-tree implies a direction of the skeleton edges: the edges in a skeleton with reference edge $e_r = (s, t)$ are directed such that the skeleton is a planar st -graph; see, for example, [Di Battista and Tamassia, 1996a]. This direction is necessary in order to identify the left and the right face of an edge.

The algorithm traverses the path Υ from μ_1 to μ_{k-1} and iteratively computes the lengths of the shortest ec-insertion paths that start from v and leave the pertinent graph P_i of μ_i to the left or to the right, respectively, where all ec-embeddings of P_i are considered. Here, left and right refer to the direction of the reference edge of μ_i . These lengths are maintained in the variables λ_ℓ and λ_r . Finally, when node μ_k is considered, this information is used to determine a shortest insertion path ending at w .

For each node μ_i , the following information is computed:

- ϕ_ℓ^i (respectively ϕ_r^i) indicates if the shortest ec-insertion path leaving P_i to the left (right) uses the shortest ec-insertion path that leaves P_{i-1} to the left (in this case the value is ℓ) or to the right (the value is r).
- Δ_ℓ^i (respectively Δ_r^i) is the subpath that is appended to the path leaving P_{i-1} when leaving P_i to the left (right).

These values are solely used for the purpose of creating the optimal ec-insertion path at the end of the function. If $s \in \{\ell, r\}$ denotes a side, we denote with \bar{s} the other side, that is, $\bar{\ell} = r$ and vice versa.

The body of the for-loop starts by expanding all edges of the skeleton S_i of μ_i except for edges representing v or w . The resulting graph is called G_i . If $1 < i < k$, then G_i will contain two virtual edges e_v (representing v) and e_w (representing w). Note that we obtain P_i (plus reference edge) by replacing e_v with P_{i-1} .

We distinguish according to the type of μ_i . If μ_i is a P-node, then the optimal ec-insertion path leaving P_{i-1} to the left (right) is also an optimal ec-insertion path leaving P_i to the left (right); we just need to permute the parallel edges in S_i such that e_v is the leftmost (rightmost) edge. Otherwise, we have four possibilities for extending an ec-insertion path leaving P_i . Such a path may start in a face left or right of e_v , and may end in a face left or right of e_w . In addition, we have to consider two special cases: if $i = 1$ then G_i contains v and

```

function OPTIMALECBLOCKINSERTER(Block  $B$  of  $K$ , vertex  $v$ , vertex  $w$ )
  Construct SPQR-tree  $\mathcal{T}$  of  $B$  such that the embeddings of the
  skeletons imply a feasible embedding of  $B$ .

  Find the shortest path  $\mu_1, \dots, \mu_k$  in  $\mathcal{T}$  between an allocation node
   $\mu_1$  of  $v$  and  $\mu_k$  of  $w$ .
  Root  $\mathcal{T}$  such that  $\mu_k$  becomes the parent of  $\mu_{k-1}$  (if  $k > 1$ ).

   $\lambda_\ell := \lambda_r := 0$      $\triangleright$  length of shortest insertion path leaving to the left/right

  for  $i = 1, \dots, k$  do
    let  $S_i = \text{skeleton}(\mu_i)$ 

    let  $G_i$  be the graph obtained from  $S_i$  by replacing each edge not repre-
    senting  $v$  or  $w$  with its expansion graph, and let  $\Pi_i$  be the embedding
    of  $G_i$  induced by the embeddings of the skeletons of  $\mathcal{T}$ .

     $\triangleright \phi_{l/r}^i$  indicates which insertion path of  $\mu_{i-1}$  is chosen.
     $\triangleright \Delta_{l/r}^i$  denotes the subpath within  $S_i$  when leaving left/right.

    if  $\mu_i$  is a P-node then
       $(\phi_\ell^i, \Delta_\ell^i) := (\ell, \epsilon); (\phi_r^i, \Delta_r^i) := (r, \epsilon)$      $\triangleright$  no crossings required
    else     $\triangleright$  S- or R-node
      if  $i = 1$  then
         $L_v := R_v :=$  the set of adjacent faces of the copy of  $v$  in  $S_i$ 
      else
        let  $e_v$  be the representative of  $v$  in  $S_i$ 
         $L_v := \{ \text{the left face of } e_v \}$ 
         $R_v := \{ \text{the right face of } e_v \}$ 
      end if

      if  $i = k$  then
         $L_w := R_w :=$  the set of adjacent faces of the copy of  $w$  in  $S_i$ 
      else
        let  $e_w$  be the representative of  $w$  in  $S_i$ 
         $L_w := \{ \text{the left face of } e_w \}$ 
         $R_w := \{ \text{the right face of } e_w \}$ 
      end if

       $\triangleright$  Compute shortest ec-insertion paths (from l/r to l/r) within  $G_i$ .
       $\triangleright$  Note:  $p_{\ell r} = p_{\ell \ell}$  and  $p_{rr} = p_{r\ell}$  if  $i \in \{1, k\}$ .
       $p_{\ell r} := \text{SHORTESTECINSPATH}(\Pi_i, L_v, L_w)$ 
       $p_{\ell \ell} := \text{SHORTESTECINSPATH}(\Pi_i, L_v, R_w)$ 
       $p_{rr} := \text{SHORTESTECINSPATH}(\Pi_i, R_v, L_w)$ 
       $p_{r\ell} := \text{SHORTESTECINSPATH}(\Pi_i, R_v, R_w)$ 

     $\triangleright$  continued on next page. . .
  
```

Listing 6.2: Computation of an optimal ec-insertion path (biconnected case).

```

    ▷ Collect possible solutions.
     $\Lambda_\ell := \{(\lambda_\ell + |p_{\ell\ell}|, \ell, p_{\ell\ell}), (\lambda_r + |p_{r\ell}|, r, p_{r\ell})\}$ 
     $\Lambda_r := \{(\lambda_\ell + |p_{\ell r}|, \ell, p_{\ell r}), (\lambda_r + |p_{rr}|, r, p_{rr})\}$ 
    if  $\mu_i$  is an R-node that can be mirrored then
       $\Lambda_\ell := \Lambda_\ell \cup \{(\lambda_\ell + |p_{rr}|, \ell, p_{rr}^*), (\lambda_r + |p_{\ell r}|, r, p_{\ell r}^*)\}$ 
       $\Lambda_r := \Lambda_r \cup \{(\lambda_\ell + |p_{r\ell}|, \ell, p_{r\ell}^*), (\lambda_r + |p_{\ell\ell}|, r, p_{\ell\ell}^*)\}$ 
    end if

    ▷ Pick best solution.
     $(\lambda_\ell, \phi_\ell^i, \Delta_\ell^i) := \min_{1,3} \Lambda_\ell$ 
     $(\lambda_r, \phi_r^i, \Delta_r^i) := \min_{1,3} \Lambda_r$ 
  end if
end for

  ▷ Build final ec-insertion path. Note:  $\lambda_\ell = \lambda_r$  always holds here!
   $s_k := \ell$  ▷ Start with empty path.
  for  $i := k$  downto 1 do ▷ Collect path backward.
     $p_i := \Delta_{s_i}^i$ ;  $s_{i-1} := \phi_{s_i}^i$ 
  end for

  return  $p_1 + \dots + p_k$ 
end function

```

Listing 6.3: Function OPTIMALECBLOCKINSERTER (part 2).

the ec-insertion path may start in any face adjacent to v ; if $i = k$ then G_i contains w and the ec-insertion path may end in any face adjacent to w . We compute the (at most) four possible shortest ec-insertion paths using the function $\text{SHORTESTECINSPATH}(\Pi, F_s, F_t)$. Here Π is an ec-embedding of an ec-expansion, F_s are the faces where the insertion path may start, and F_t are the faces where it may end. The ec-insertion path is found using a breadth-first search (BFS) in the dual graph of Π , where edges corresponding to gadget edges are removed (which means that it is forbidden to cross their primal counterparts). We call these shortest ec-insertion paths $p_{\ell\ell}, p_{\ell r}, p_{r\ell}, p_{rr}$, where $p_{\ell\ell}$ stands for the path starting in a face in L_v and ending in a face in R_w and so on. We have two choices for a shortest ec-insertion path leaving P_i to the left if we consider only the given embedding of the skeleton of μ_i :

- We leave P_{i-1} to the left (or start at v if $i = 1$) and end in a face in R_w (that is, we enter e_w from right). This path has length $\lambda_\ell + |p_{\ell\ell}|$.
- We leave P_{i-1} to the right (or start at v if $i = 1$) and end in a face in R_w (that is, we enter e_w from left). This path has length $\lambda_r + |p_{r\ell}|$.

For the shortest ec-insertion path leaving P_i to the right, we have two similar cases. Further choices are possible if μ_i is an R-node that can be mirrored. We could mirror the embedding of S_i , expand the skeleton edges as before such that we obtain an embedding $\tilde{\Pi}_i$, and compute the four paths in $\tilde{\Pi}_i$ again. Notice that

$\tilde{\Pi}_i$ is not simply the mirror image of Π_i . However, this is not necessary. We observe that, for example, the path $\tilde{p}_{\ell\ell}$ is obtained from p_{rr} by reversing the subsequences of edges that have been created by expanding a common skeleton edge of S_i . We call this path p_{rr}^* . A similar argumentation holds for $\tilde{p}_{\ell r}, \tilde{p}_{r\ell}, \tilde{p}_{rr}$. It follows that we have at most four possible choices for leaving P_i to the left and to the right, respectively. Among all possible choices, we pick the shortest one.

After processing all nodes μ_i , it is easy to reconstruct the best ec-insertion path from v to w using $\phi_{\ell/r}^i$ and $\Delta_{\ell/r}^i$. Notice that $\lambda_\ell = \lambda_r$ holds at the end, since $L_w^k = R_w^k$.

6.4.3 Correctness and Optimality

Lemma 6.5. *There exists an ec-embedding Π of B such that $p_1 + \dots + p_k$ is an ec-insertion path for v and w in B with respect to Π .*

Proof. Consider the path $\Upsilon = \mu_1, \dots, \mu_k$ computed by the algorithm. By construction of Υ , the skeleton of μ_1 contains v , the skeleton of μ_k contains w , and, for each $j = 2, \dots, k-1$, the skeleton of μ_j contains neither v nor w . Moreover, Υ does not contain a Q-node.

First, we prove the lemma for the case where Υ consists of a single node μ_1 . In this case, the skeleton of μ_1 contains both v and w . We distinguish two cases according to the type of μ_1 :

- (a) **μ_1 is a P-node.** Let Π be an arbitrary ec-embedding of B . Since v and w share a common face in Π , the empty path returned by the algorithm is an ec-insertion path for v and w in B with respect to Π ; see Figure 6.4(a).
- (b) **μ_1 is an S- or an R-node.** In this case the graph G_1 constructed by the algorithm is the original block B , since all skeleton edges are expanded. Moreover, Π_1 is an ec-embedding of B , and $p_{\ell r} = p_{\ell\ell}$ and $p_{rr} = p_{r\ell}$ are ec-insertion paths in B with respect to Π_1 . We do not need to consider the case where the embedding of the skeleton can be mirrored, since this will not yield a shorter path. Hence, p_1 is either $p_{\ell r}$ or p_{rr} and thus an ec-insertion path in B with respect to Π_1 .

Assume now that $k > 1$. For $i = 1, \dots, k$, we denote with H_i the pertinent graph of μ_i , with r_i the reference edge of μ_i in H_i , and, for $1 < i$, with e_i the edge in $\text{skeleton}(\mu_i)$ whose pertinent node is μ_{i-1} . Recall that $s_i \in \{\ell, r\}$ is the side of H_i where the computed insertion path shall leave. We show by induction over i that, for $1 \leq i < k$, there is an embedding Γ_i of H_i such that $p_1 + \dots + p_i$ is an ec-insertion path leaving H_i at side s_i . The embeddings $\Gamma_1, \dots, \Gamma_{k-1}$ are iteratively constructed during the proof. For our convenience, we denote with Γ_i^- the embedding of $H_i - r_i$ induced by Γ_i .

$i = 1$. Consider the different types for node μ_1 :

- (a) **μ_1 is a P-node.** This case does not apply, since μ_2 is not an allocation node of v .

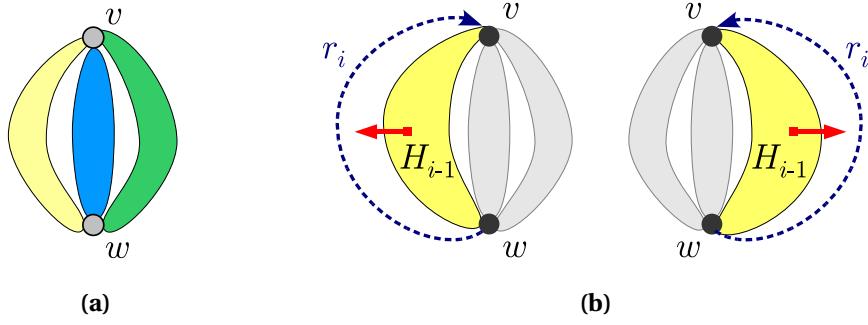


Figure 6.4: Proof of Lemma 6.5; $k = 1$ and μ_1 is a P-node (a), and μ_i is a P-node (b).

- (b) μ_1 **is an S- or an R-node.** In this case, $G_1 = H_1$, and p_1 is a path leaving either Π_1 or the mirror image of Π_1 to side s_1 . Hence, we set Γ_1 to Π_1 or its mirror image, respectively.

$1 < i < k$. We distinguish again between the types of μ_i .

- (a) μ_i **is a P-node.** In this case, $p_i = \epsilon$, that is, the empty path, since no further edges need to be crossed. The embedding Γ_i is obtained as follows. If $s_i = \ell$, we permute the edges in $\text{skeleton}(\mu_i)$ such that e_i is to the right of r_i ; otherwise, we permute the edges such that e_i is to the left of r_i . Then, we replace e_i by Γ_{i-1}^- , and the remaining edges $e \neq r_i$ in $\text{skeleton}(\mu_i)$ by an arbitrary embedding of $\text{expansion}(e)$; see Figure 6.4(b).
- (b) μ_i **is an S- or an R-node.** In this case, p_i is either $p_{s_{i-1}s_i}$ or $p_{\bar{s}_{i-1}\bar{s}_i}$; the latter case corresponds to mirroring the embedding of $\text{skeleton}(\mu_i)$ before.

We first restrict us to the case in which p_i is set to $p_{s_{i-1}s_i}$, that is, an *ec-insertion* path in the embedding Π_i that starts in a face at side s_{i-1} of e_i and ends in a face at side \bar{s}_i of edge r_i . We obtain Γ_i by replacing e_i by Γ_i^- in Π_i ; see Figure 6.5. Since the *ec-insertion* path $p_1 + \dots + p_{i-1}$ leaves Γ_i^- to the side s_{i-1} , $p_1 + \dots + p_i$ is an *ec-insertion* path leaving Γ_i to the side s_i .

Finally, assume that $p_i = p_{\bar{s}_{i-1}\bar{s}_i}$. Let $\tilde{\Pi}_i$ be the embedding that we obtain by first mirroring the embedding of $\text{skeleton}(\mu_i)$ and then expanding and embedding each skeleton edge not representing v or w as before. We observe that p_i is an *ec-insertion* path in $\tilde{\Pi}_i$ that starts in a face at side s_{i-1} of e_i and ends in a face at side \bar{s}_i of edge r_i ; see Figure 6.6. With the same argumentation as above, we obtain Γ_i by replacing e_i with Γ_{i-1}^- in $\tilde{\Pi}_i$.

To conclude the proof, we consider the node μ_k . We know that μ_k is either an S- or an R-node, and we may assume that $p_k = p_{s_{i-1}s_i}$, since $p_{\ell r} = p_{\ell \ell}$ and $p_{rr} = p_{r \ell}$ holds for $i = k$. Hence, p_k is an *ec-insertion* path in Π_k that starts in

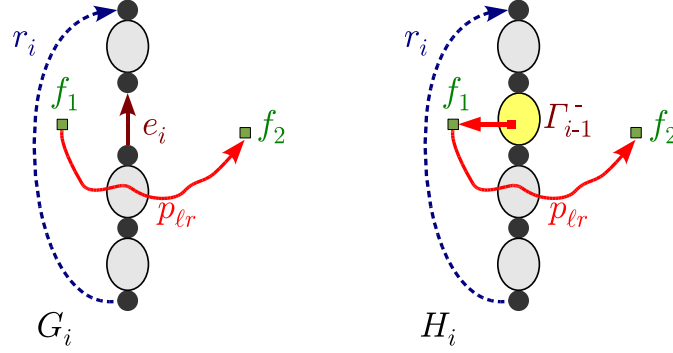


Figure 6.5: Proof of Lemma 6.5; μ_i is an S-node, $L_v = R_w = \{f_1\}$, $R_v = L_w = \{f_2\}$.

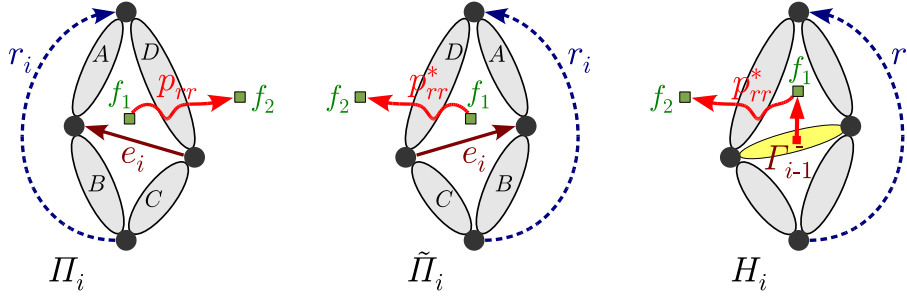


Figure 6.6: Proof of Lemma 6.5; μ_i is an R-node, $R_v = \{f_1\}$, $L_w = \{f_2\}$.

a face at side s_{i-1} of e_i and ends in a face adjacent to the copy of w in G_k . We obtain Π by replacing e_k with Γ_{k-1}^- in Π_k , and thus $p_1 + \dots + p_k$ is an ec-insertion path for v and w in Π . \square

Lemma 6.6. *Let Π' be an arbitrary ec-embedding of B and let p' be a shortest ec-insertion path for v and w in B with respect to Π' . Then $|p'| \geq |p_1 + \dots + p_k|$.*

Proof. Let G_i , S_i , and s_i be as defined in OPTIMALECBLOCKINSERTER, and let λ_ℓ^i and λ_r^i be the value of λ_ℓ and λ_r , respectively, after the i -th iteration of the for-loop. For $i = 1, \dots, k$, we denote with H_i the pertinent graph of μ_i . Observe, that Π' induces embeddings of G_i and S_i . Accordingly, we denote the induced embedding of G_i with Π'_i , and of S_i with Σ'_i .

Since p' is a shortest ec-insertion path, it does not visit a face twice. Therefore, we can subdivide p' into $p' = p'_1 + \dots + p'_k$ such that p'_i contains exactly the edges of p' that are in G_i , for $1 \leq i \leq k$. This follows from the fact that H_i shares only two vertices with the rest of the graph and p' does not visit a face twice. For $1 \leq i < k$, we denote with $s'_i \in \{\ell, r\}$ the side at which the ec-insertion path $p'_1 + \dots + p'_i$ leaves H_i in Π' .

We show by induction over i that $\lambda_{s'_i}^i \leq |p'_1 + \dots + p'_i|$.

$i = 1$. If $k = 1$, then $G_1 = B$ and the proposition follows immediately, so assume $k > 1$. If μ_1 is not an R-node, then $\lambda_{s'_1} = 0$ and the proposition follows immediately. Otherwise, the algorithm also computes the shortest ec-insertion path leaving at side s'_1 in Σ'_1 , where the costs of the edges are their

traversing costs. Since the traversing costs are independent of the embedding by Lemma 6.4, we get $\lambda_{s'_1}^1 \leq |p'_1|$.

$1 < i < k$. Assume now that $\lambda_{s'_j}^j \leq |p'_1 + \dots + p'_j|$ for $1 \leq j < i$. We distinguish two cases:

- (a) μ_i **is a P-node**. In this case, we have $s'_{i-1} = s'_i$, since $p_i + \dots + p_k$ does not contain an edge of H_{i-1} . This yields

$$\lambda_{s'_i}^i = \lambda_{s'_{i-1}}^{i-1} \leq |p'_1 + \dots + p'_{i-1}| \leq |p'_1 + \dots + p'_i|.$$

- (b) μ_i **is an S- or an R-node**. Observe that p'_i is an ec-insertion path in Π'_i starting in the face at side s'_i of the edge representing v and ending in a face at side \bar{s}'_{i+1} of the edge representing w if $i < k$, or a face adjacent to w otherwise. This implies an ec-insertion path in Σ'_i , where the costs of a skeleton edge are its traversing costs. On the other hand, the algorithm computes a shortest ec-insertion path in Σ'_i , since the traversing costs of a skeleton edge are independent of the embedding by Lemma 6.4. Thus, we get $\lambda_{s'_i} - \lambda_{s'_{i-1}} \leq |p'_i|$, and hence

$$\lambda_{s'_i} \leq \lambda_{s'_{i-1}} + |p'_i| \leq |p'_1 + \dots + p'_i|.$$

Finally, we get $|p_1 + \dots + p_k| = \lambda_{s'_i}^i \leq |p'|$ and the lemma holds. \square

Theorem 6.3. *Let $B = (V, E)$ be a block of K and let v and w be two distinct vertices of B . Then, function `OPTIMALECBLOCKINSERTER` computes an optimal ec-insertion path for v and w in B in time $\mathcal{O}(|E|)$.*

Proof. The correctness and optimality of the algorithm follows from Lemma 6.5 and Lemma 6.6. Constructing the SPQR-tree and embedding the skeleton graphs takes time $\mathcal{O}(|E|)$. Let $G_i = (V_i, E_i)$ be the graph considered in each iteration of the for-loop. Then, each iteration takes time $\mathcal{O}(|E_i|)$, since `SHORTESTECINSPATH` takes only time linear in the size of G_i by applying BFS. Moreover, the set E_i consists of some edges E'_i of G plus at most two virtual edges (the representatives of v and w). Thus, $|E_1| + \dots + |E_k| = \mathcal{O}(|E|)$, and hence we get a total running time of $\mathcal{O}(|E|)$. \square

6.4.4 Generalization to Connected Graphs

The edge insertion algorithm can easily be generalized to connected graphs by using the same technique as in Section 4.1.3 for the unconstrained case; see Algorithm `OPTIMALECINSERTER` in Listing 6.4. For each block B_i on the path from v to w in the block-vertex tree \mathcal{B} of G , we compute the optimal ec-edge insertion path p_i between the representatives of v and w with a corresponding ec-planar embedding Π_i . Then, we concatenate these ec-edge insertion paths building the optimal ec-edge insertion path for v and w .

```

1: function OPTIMALECINSERTER(ec-expansion  $G$ , vertex  $v$ , vertex  $w$ )
2:   Compute the block-vertex tree  $\mathcal{B}$  of  $G$ .
3:   Find the path  $v, B_1, c_1, \dots, B_{k-1}, c_{k-1}, B_k, w$  from  $v$  to  $w$  in  $\mathcal{B}$ .
4:   for  $i := 1, \dots, k$  do
5:     let  $x_i$  and  $y_i$  be the representatives of  $v$  and  $w$  in  $B_i$ 
6:      $p_i :=$  OPTIMALECBLOCKINSERTER( $B_i, x_i, y_i$ )
7:   end for
8:   return  $p_1 + \dots + p_k$ 
9: end function

```

Listing 6.4: Computation of an optimal ec-insertion path.

The proof of Lemma 4.4 uses induction over the number of blocks on the path from v to w in \mathcal{B} . We briefly recall this proof. Let B_1, \dots, B_k be the blocks on this path and let H_i be the union of the blocks B_1 to B_i . Let Π_i be an embedding of B_i such that p_i is an optimal edge insertion path for the representatives x_i and y_i in B_i with respect to Π_i . Let Ψ_i denote the concatenation $p_1 + \dots + p_i$.

An embedding Γ_i for H_i with an optimal edge insertion path Ψ_i can be iteratively constructed by combining the embedding Γ_{i-1} for H_{i-1} and the embedding Π_i for block B_i . Both y_{i-1} and x_i denote the same vertex in G and there exist optimal edge insertion paths Ψ_{i-1} for v_1 and y_{i-1} as well as p_i for x_i and y_i . Therefore there is a face $f \in \Gamma_{i-1}$ that contains y_{i-1} and either v_1 if Ψ_{i-1} is empty or the last edge in Ψ_{i-1} . Similarly, there is a face $f' \in \Pi_i$ that contains x_i and either y_i if p_i is empty or the first edge in p_i . We can directly concatenate the two paths if both faces coincide. This can be achieved by choosing f as the external face of Γ_{i-1} and placing this embedding of H_{i-1} into face f' of Π_i . Then Ψ_i is an optimal ec-insertion path for v_1 and y_i in H_i with respect to Γ_i .

We need to show that—under the presence of embedding constraints—ec-planarity is preserved, that is, Γ_k is still an ec-planar embedding. The only critical point in each step is the selection of f as the external face; but this does not change the clockwise order of the edges around the vertices of G . Furthermore, we ensure in the computation of the ec-edge insertion paths p_i that we do not cross any expansion edges. Hence, we know that the paths Ψ_{i-1} and p_i do not start or end in a face containing a hub. Therefore, the ec-planarity conditions are still fulfilled and Γ_k is ec-planar.

It is obvious that $p_1 + \dots + p_k$ is an ec-edge insertion path for v and w with respect to an embedding Π that results from inserting the remaining blocks not contained in H_k (as shown in the proof of Lemma 6.3) into Γ_k . The length of the computed ec-edge insertion path is obviously minimal, since a shorter path would imply that there exists a shorter path within a block. The block-vertex tree of a graph can be constructed in linear time and the running time of OPTIMALECBLOCKINSERTER(B_i, x_i, y_i) is linear in the size of the block B_i (Theorem 6.3), thus yielding linear running time for OPTIMALECINSERTER.

Together with Lemma 6.1, we obtain the following result:

Theorem 6.4. *Let $G = (V, E)$ be a graph with embedding constraints C and $e = (v, w) \in E$ such that $G - e$ is ec-planar. Then, we can compute an optimal ec-edge insertion path for (v, w) in $G - e$ in $\mathcal{O}(|V| + |E|)$ time.*

6.5 Summary and Discussion

We introduced a flexible concept of embedding constraints which allows us to model a wide range of constraints on the order of edges incident to a vertex. We presented a linear time algorithm for testing ec-planarity, as well as a characterization of all possible ec-embeddings. The latter is in particular important for developing algorithms that optimize over the set of all ec-planar embeddings. We showed that optimal edge insertion can still be performed in linear time when embedding constraints have to be respected. In order to devise practically successful graph drawing algorithms, the following problems should be considered:

- Develop faster algorithms for finding ec-planar subgraphs.
- Solve the so-called *orientation problem* for orthogonal graph drawing, for example, allow us to fix some edges to attach only at the top side of a rectangular vertex. The problem arises when angles are assigned at each vertex between adjacent edges to fix the assignment to the vertex's sides, for example, in network flow based drawing approaches. The vertices then need to be oriented such that the edges that are assigned to the same sides at different vertices are aligned.
- In some applications, only a subset of the edges is subject to embedding constraints at a vertex v , that is, some edges can attach at arbitrary positions. Hence, we wish to extend the concept of embedding constraints for so-called *free edges* that are not contained in the tree T_v .

Chapter 7

Conclusion

Science is a wonderful thing if one does not have to earn one's living at it.

ALBERT EINSTEIN (1879 – 1955)

The planarization method—in combination with the topology-shape-metrics approach—is one of the most important methods for drawing graphs. It is by far the best method with respect to the fundamental aesthetic criterium of minimizing the number of crossings in a drawing.

The main goal of this thesis was to improve this method for practical applications. Therefore, we developed efficient heuristics for tackling two major problems: crossing reduction and finding good planar embeddings. We improved the crossing reduction step by presenting a new linear-time algorithm for solving the one-edge insertion problem optimally. In the original version of the planarization method, this problem was only solved for a given fixed embedding, thus producing lots of unnecessary crossings when choosing a bad initial embedding. Beforehand, it was not even clear if the one-edge insertion problem was solvable in polynomial time. We could also show that further pre- and postprocessing methods, as well as random permutations lead to excellent results. We presented two new methods, namely the non-planar core reduction and the incremental postprocessing strategy, which—besides the optimal one-edge insertion algorithm—both are essential for achieving the best quality. In an experimental study, we showed that these heuristics get close to optimal results for real-world-like graphs (the established Rome graphs) as well as for graphs from graph classes with known crossing numbers (a new benchmark set introduced in this thesis). In case of the Rome graphs, we could show that even the quality achieved by exact crossing minimization methods [Buchheim et al., 2008, Chimani et al., 2008] will not be significantly better in practice.

Even when finding a planarized representation with only few crossings, the resulting drawings can look quite bad if the planar drawing algorithm uses an inappropriate planar embedding. Since optimizing aesthetic criteria like, for example, the number of bends over all embeddings is computationally hard (even for 4-planar graphs), it is worthwhile to come up with other properties that make up good planar embeddings. Here, we discussed the block-nesting depth—thus

generalizing the result by Pizzonia and Tamassia [2000]—and the degree of the external face. For both problems, we presented linear-time algorithms. As can be shown, such embeddings can improve layout quality significantly; see [Kerkhof, 2007].

Though the planarization method is the superior approach in many cases, the planarization approach still lacks acceptance in practice. One reason is the high implementation effort requiring expert knowledge. With the public availability of our algorithms in the OGDF library, we try to overcome this dilemma. Another problem is that in many application domains further restrictions on admissible drawings exist. Typically, such constraints can easily be integrated into simpler drawing algorithms. Therefore, we incorporated embedding constraints into the planarization method. Although these constraints cannot model all restrictions we find in application domains, it is a good step in this direction, hopefully helping to bring the planarization method into practice. In Chimani and Gutwenger [2007], we also extended the planarization method for drawing hypergraphs, for example, we study the insertion of hyperedges. The concept of hyperedges is also present in various applications.

The fundamental basis for all these algorithms is the SPQR-tree data structure. Though this data structure is simply an enriched version of the triconnected components and there exists a linear-time algorithm for finding the triconnected components [Hopcroft and Tarjan, 1973a] since the early 1970s, there was no linear-time implementation for constructing SPQR-trees so far. The reason was that the paper by Hopcroft and Tarjan contained various faulty parts, thus inhibiting linear-time implementations for triconnectivity decomposition in general and SPQR-trees in particular. In this thesis, we gave a corrected version of their algorithm and applied it to SPQR-trees. The resulting implementation is fast and stable in practice. Moreover, we made it publicly available in graph drawing libraries, at first in AGD and later in OGDF. This data structure is not only very useful for countless graph drawing problems, but has also applications in completely different areas, for example, wave digital filters [Fränken et al., 2003, 2005] or information flow security [McCamant and Ernst, 2008, McCamant, 2008]. Since our first publication of a corrected version of the Hopcroft and Tarjan algorithm in [Gutwenger and Mutzel, 2001], further linear-time implementations of SPQR-trees were made possible, for example, recently an implementation in the Java graph drawing library *yFiles* [Mader, 2008].

Although the planarization method is now a practical tool for drawing sparse graphs of medium size, there is still work to do. We do not think that it is important to improve the performance of this method for denser graphs, since the topology-shape-metrics approach in general is not the proper method for drawing such graphs: Other methods like, for example, multi-level force directed algorithms will give better results in most cases and scale even for graphs with 100,000s of edges. However, the performance of the planarization method should still be improved for large sparse graphs. In particular, the postprocessing methods used in the crossing minimization heuristics should be faster. One possible approach could be the usage of dynamic updates with the optimal one-edge in-

sersion algorithm. This is in particular difficult when supporting the full range of postprocessing, since in this case, we also need to support edge deletions. Further ideas could be to exploit multi-core processors by parallelization (this is obviously easy when applying several permutations in the crossing minimization heuristic) or speed-up techniques for finding shortest paths in the one-edge insertion algorithm. An important and challenging problem is also to extend our concept of embedding constraints to free edges, that is, edges that can be attached everywhere at a vertex.

From the theoretical point of view, there are several interesting yet also challenging open problems to consider. First of all, instead of just inserting a single edge optimally, we could insert a vertex with all its incident edges optimally. In [Chimani, Gutwenger, Mutzel, and Wolf, 2009b], we have recently presented a polynomial-time algorithm for solving this problem. However, it heavily relies on dynamic programming and its running time is still far away from being practical. A yet open problem is the optimal insertion of k edges at once, where $k \geq 2$ is a constant. Notice that even for the special case, where the edges to be inserted are all incident to a common vertex, an iterative application of the one-edge insertion algorithm will not yield an optimal result. For example, the insertion of the first edge could fix the embedding of an R-node (one of the two choices we have), but this choice is inappropriate for the remaining edges. Even more complications arise when considering P-nodes. However, the optimal insertion of several edges at once could significantly improve the final planarized representation and probably reduce the effort for postprocessing, hence we think this is an open problem that could improve the planarization method also in practice.

Bibliography

- J. Adamsson and R. B. Richter. Arrangements, circular arrangements and the crossing number of $C_7 \times C_n$. *Journal of Combinatorial Theory, Series B*, 90:21–39, 2004. doi:10.1016/j.jctb.2003.05.001.
- P. Angelini, G. Di Battista, F. Frati, V. Jelinek, J. Kratochvil, M. Patrignani, and I. Rutter. Testing planarity of partially embedded graphs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*, pages 202–221, 2010a. URL http://www.siam.org/proceedings/soda/2010/SODA10_019_angelinip.pdf.
- P. Angelini, G. Di Battista, and M. Patrignani. Finding a minimum-depth embedding of a planar graph in $O(n^4)$ time. *Algorithmica*, 2010b. doi:10.1007/s00453-009-9380-6. To appear.
- L. Auslander and S. V. Parter. On imbedding graphs in the plane. *Journal of Mathematics and Mechanics*, 10:517–523, 1961.
- C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity relationship diagrams. *Journal of Systems and Software*, 4:163–173, 1984. doi:10.1016/0164-1212(84)90006-2.
- C. Batini, E. Nardelli, and R. Tamassia. A layout algorithm for data-flow diagrams. *IEEE Transactions on Software Engineering*, 12(4):538–546, 1986.
- L. W. Beineke and R. D. Ringeisen. On the crossing numbers of products of cycles and graphs of order four. *Journal of Graph Theory*, 4:145–155, 1980. doi:10.1002/jgt.3190040203.
- P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*, 27(1):132–169, 1998. doi:10.1137/S0097539794279626.
- P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. *IEEE Transactions on Computers*, 49(8):826–840, 2000. doi:10.1109/12.868028.
- S. N. Bhatt and F. T. Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28(2):300–343, 1984. doi:10.1016/0022-0000(84)90071-0.

- D. Bienstock and C. L. Monma. On the computational complexity of embedding planar graphs to minimize certain distance measures. *Algorithmica*, 5(1):93–109, 1990. doi:10.1007/BF01840379.
- K.-F. Böhringer and F. N. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, pages 43–51. ACM Press, 1990. URL <http://doi.acm.org/10.1145/97243.97250>.
- K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976. doi:10.1016/S0022-0000(76)80045-1.
- J. M. Boyer and W. Myrvold. On the cutting edge: Simplified $O(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004. URL <http://www.cs.brown.edu/sites/jgaa/accepted/2004/BoyerMyrvold2004.8.3.pdf>.
- U. Brandes and D. Wagner. A bayesian paradigm for dynamic graph layout. In *Proceedings of the 5th International Symposium on Graph Drawing (GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 236–247. Springer-Verlag, 1997. doi:10.1007/3-540-63938-1_66.
- U. Brandes, M. Eiglsperger, M. Kaufmann, and D. Wagner. Sketch-driven orthogonal graph drawing. In *Proceedings of the 10th International Symposium on Graph Drawing (GD 2002)*, volume 2528 of *Lecture Notes in Computer Science*, pages 131–148. Springer-Verlag, 2002. doi:10.1007/3-540-36151-0_1.
- C. Buchheim, M. Jünger, A. Menze, and M. Percan. Bimodal crossing minimization. In *Proceedings of the 12th Annual International Conference on Computing and Combinatorics (COCOON 2006)*, volume 4112 of *Lecture Notes in Computer Science*, pages 497–506. Springer-Verlag, 2006. doi:10.1007/11809678_52.
- C. Buchheim, M. Chimani, D. Ebner, C. Gutwenger, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. A branch-and-cut approach to the crossing number problem. *Discrete Optimization*, 5(2):373–388, 2008. doi:10.1016/j.disopt.2007.05.006.
- S. Cabello and B. Mohar. Crossing and weighted crossing number of near-planar graphs. In *Proceedings of the 16th International Symposium on Graph Drawing (GD 2008)*, volume 5417 of *Lecture Notes in Computer Science*, pages 38–49. Springer-Verlag, 2009a. doi:10.1007/978-3-642-00219-9_5.
- S. Cabello and B. Mohar. Crossing number and weighted crossing number of near-planar graphs. *Algorithmica*, 2009b. doi:10.1007/s00453-009-9357-5. To appear.

- S. Cabello and B. Mohar. Adding one edge to planar graphs makes crossing number hard. In *Proceedings of the 2010 annual symposium on Computational geometry (SCG 2010)*, pages 67–76. ACM, 2010. URL <http://doi.acm.org/10.1145/1810959.1810972>.
- G. Călinescu, C. G. Fernandes, U. Finkler, and H. Karloff. A better approximation algorithm for finding planar subgraphs. *Journal of Algorithms*, 27(2):269–302, 1998. doi:10.1006/jagm.1997.0920.
- N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and System Sciences*, 30(1):54–76, 1985. doi:10.1016/0022-0000(85)90004-2.
- M. Chimani. *Computing Crossing Numbers*. PhD thesis, Fakultät für Informatik, Technische Universität Dortmund, 2008.
- M. Chimani and C. Gutwenger. Non-planar core reduction of graphs. In P. Healy and N. S. Nikolov, editors, *Proceedings of the 13th International Symposium on Graph Drawing (GD 2005)*, volume 3843 of *Lecture Notes in Computer Science*, pages 223–234. Springer-Verlag, 2005. doi:10.1007/11618058_21.
- M. Chimani and C. Gutwenger. Algorithms for the hypergraph and the minor crossing number problems. In T. Nishizeki, editor, *Proceedings of the 9th International Symposium on Algorithms and Computation (ISAAC 2007)*, volume 4835 of *Lecture Notes in Computer Science*, pages 184–195. Springer-Verlag, 2007. doi:10.1007/978-3-540-77120-3_18.
- M. Chimani and C. Gutwenger. Non-planar core reduction of graphs. *Discrete Mathematics*, 309(7):1838–1855, 2009. doi:10.1016/j.disc.2007.12.078.
- M. Chimani, C. Gutwenger, and P. Mutzel. Experiments on exact crossing minimization using column generation. In M. Serna, editor, *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA 2006)*, volume 4007 of *Lecture Notes in Computer Science*, pages 303–315. Springer-Verlag, 2006. doi:10.1007/11764298_28.
- M. Chimani, C. Gutwenger, and P. Mutzel. On the minimum cut of planarizations. *Electronic Notes in Discrete Mathematics*, 28:177–184, 2007. doi:10.1016/j.endm.2007.01.036.
- M. Chimani, P. Mutzel, and I. Bomze. A new approach to exact crossing minimization. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008)*, volume 5193 of *Lecture Notes in Computer Science*, pages 284–296. Springer-Verlag, 2008. doi:10.1007/978-3-540-87744-8_24.
- M. Chimani, C. Gutwenger, and P. Mutzel. Experiments on exact crossing minimization using column generation. *ACM Journal of Experimental Algorithms*, 15:3.4–3.18, 2009a. URL <http://doi.acm.org/10.1145/1498698.1564504>.

- M. Chimani, C. Gutwenger, P. Mutzel, and C. Wolf. Inserting a vertex into a planar graph. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*, pages 375–383. ACM Press, 2009b. URL http://www.siam.org/proceedings/soda/2009/SODA09_042_chimanim.pdf.
- M. Chimani, C. Gutwenger, M. Jünger, K. Klein, P. Mutzel, and M. Schulz. *The Open Graph Drawing Framework*. Technische Universität Dortmund, University of Cologne, 2010. URL <http://www.ogdf.net>.
- E. Dahlhaus. A linear time algorithm to recognize clustered planar graphs and its parallelization. In C. L. Lucchesi and A. V. Moura, editors, *LATIN '98: Theoretical Informatics, Third Latin American Symposium*, volume 1380 of *Lecture Notes in Computer Science*, pages 239–248. Springer-Verlag, 1998. doi:10.1007/BFb0054325.
- C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 2009. URL <http://www.dis.uniroma1.it/~challenge9/>.
- G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS '89)*, pages 436–441, 1989.
- G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In M. S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 598–611. Springer-Verlag, 1990. doi:10.1007/BFb0032061.
- G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996a. doi:10.1137/S0097539794280736.
- G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996b. doi:10.1007/BF01961541.
- G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry*, 7(5–6):303–326, 1997. doi:10.1016/S0925-7721(96)00005-3.
- G. Di Battista, W. Didimo, M. Patrignani, and M. Pizzonia. Drawing database schemas. *Software: Practice and Experience*, 32(11):1065–1098, 2002. doi:10.1002/spe.474.
- W. Didimo, 1999. Dipartimento di Informatica e Automazione, Università di Roma Tre, Rome, Italy, Personal Communication.

- W. Didimo and G. Liotta. Computing orthogonal drawings in a variable embedding setting. In L. Chwa, editor, *Proceedings of the 9th International Symposium on Algorithms and Computation (ISAAC '98)*, volume 1533 of *Lecture Notes in Computer Science*, pages 79–89. Springer-Verlag, 1998. doi:10.1007/3-540-49381-6_10.
- R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, third edition, 2005.
- H. N. Djidjev. A linear algorithm for the maximal planar subgraph problem. In *Proceedings of the 4th Workshop on Algorithms and Data Structures (WADS '95)*, volume 955 of *Lecture Notes in Computer Science*, pages 369–380. Springer-Verlag, 1995. doi:10.1007/3-540-60220-8_77.
- C. Dornheim. Planar graphs with topological constraints. *Journal on Graph Algorithms and Applications*, 6(1):27–66, 2002. URL <http://jgaa.info/accepted/2002/Dornheim2002.6.1.pdf>.
- M. E. Dyer, L. R. Foulds, and A. M. Frieze. Analysis of heuristics for finding a maximum weight planar subgraph. *European Journal of Operational Research*, 20(1):102–114, 1985. doi:10.1016/0377-2217(85)90288-7.
- P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994. doi:10.1007/BF01187020.
- M. Eiglsperger, U. Fößmeier, and M. Kaufmann. Orthogonal graph drawing with constraints. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms (SODA 2000)*, pages 3–11. SIAM, 2000.
- L. Euler. Demonstratio nonnullarum insignium proprietatum, quibus solida hedris planis inclusa sunt praedita. *Novi Commentarii academiae scientiarum imperialis Petropolitanae*, 4:140–160, 1758.
- G. Exoo, F. Harary, and J. Kabell. The crossing numbers of some generalized Petersen graphs. *Mathematica Scandinavica*, 48:184–188, 1981.
- U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F. J. Brandenburg, editor, *Proceedings of the Symposium on Graph Drawing (GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 1996. doi:10.1007/BFb0021783.
- D. Fränken, J. Ochs, and K. Ochs. Generation of wave digital structures for connection networks containing ideal transformers. In *IEEE Intern. Symp. on Circuits and Systems (ISCAS)*, volume 3, pages 240–243, 2003.
- D. Fränken, J. Ochs, and K. Ochs. Generation of wave digital structures for networks containing multiport elements. *IEEE Transactions on Circuits and Systems, Part I: Regular Papers*, 52:586–596, 2005. doi:10.1109/TCSI.2004.843056.

- D. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacement. *SIAM Journal on Computing*, 22(3):587–616, 1993. doi:10.1137/0222040.
- M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY, 1979.
- M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983. doi:10.1137/0604033.
- A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM Journal on Computing*, 31(2):601–625, 2002. doi:10.1137/S0097539794277123.
- GDToolkit. Graph drawing toolkit: An object-oriented library for handling and drawing graphs. URL <http://www.dia.uniroma3.it/~gdt>.
- A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. In *Graph and Combinatorics Conference, Office of Naval Research Logistics Proj., Contract NONR 1858-(21)*. Dept. of Math., Princeton Univ., 1963. 2 unnumbered pp.
- M. Grohe. Computing crossing numbers in quadratic time. *Journal of Computer and System Sciences*, 68(2):285–302, 2004. doi:10.1016/j.jcss.2003.07.008.
- M. Gronemann. Engineering the fast-multipole-multilevel method for multicore and SIMD architectures. Master’s thesis, Technische Universität Dortmund, 2009.
- C. Gutwenger and P. Mutzel. A linear time implementation of SPQR trees. In J. Marks, editor, *Proceedings of the 8th International Symposium on Graph Drawing (GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer-Verlag, 2001. doi:10.1007/3-540-44541-2_8.
- C. Gutwenger and P. Mutzel. An experimental study of crossing minimization heuristics. In G. Liotta, editor, *11th Symposium on Graph Drawing 2003, Perugia*, volume 2912 of *Lecture Notes in Computer Science*, pages 13–24. Springer-Verlag, 2003a. URL <http://www.springerlink.com/content/6ac0pmpyqwcg0txr>.
- C. Gutwenger and P. Mutzel. Graph embedding with minimum depth and maximum external face. In G. Liotta, editor, *11th Symposium on Graph Drawing 2003, Perugia*, volume 2912 of *Lecture Notes in Computer Science*, pages 259–272. Springer-Verlag, 2003b. URL <http://www.springerlink.com/content/dbcv5uptvh1uyt92>.
- C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*, pages 246–255. ACM Press, 2001. URL <http://portal.acm.org/citation.cfm?id=365455>.

- C. Gutwenger, M. Jünger, S. Leipert, P. Mutzel, M. Percan, and R. Weiskircher. Advances in C -planarity testing of clustered graphs. In M. Goodrich and S. Kobourov, editors, *Proceedings of the 10th International Symposium on Graph Drawing (GD 2002)*, volume 2528 of *Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, 2002. doi:10.1007/3-540-36151-0_21.
- C. Gutwenger, M. Jünger, S. Leipert, P. Mutzel, M. Percan, and R. Weiskircher. Subgraph induced planar connectivity augmentation. In H. L. Bodlaender, editor, *Proceedings of the 29th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'2003)*, volume 2880 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 2003. URL <http://www.springerlink.com/content/wgj7w9rhbgd4yv9m>.
- C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005. doi:10.1007/s00453-004-1128-8.
- C. Gutwenger, K. Klein, and P. Mutzel. Planarity testing and optimal edge insertion with embedding constraints. In M. Kaufmann and D. Wagner, editors, *Proceedings of the 14th International Symposium on Graph Drawing (GD 2006)*, volume 4372 of *Lecture Notes in Computer Science*, pages 126–137. Springer-Verlag, 2006. doi:10.1007/978-3-540-70904-6_14.
- C. Gutwenger, K. Klein, and P. Mutzel. Planarity testing and optimal edge insertion with embedding constraints. *Journal of Graph Algorithms and Applications*, 12(1):73–95, 2008. URL <http://jgaa.info/accepted/2008/GutwengerKleinMutzel2008.12.1.pdf>.
- S. Hachul and M. Jünger. Large-graph layout algorithms at work: An experimental study. *Journal of Graph Algorithms and Applications*, 11(2):345–369, 2007. URL <http://www.cs.brown.edu/sites/jgaa/accepted/2007/HachulJuenger2007.11.2.pdf>.
- G. Henschel. *Die wirrsten Grafiken der Welt*. Hoffmann und Campe Verlag, 2003.
- P. Hliněný and G. Salazar. On the crossing number of almost planar graphs. In M. Kaufmann and D. Wagner, editors, *Proceedings of the 14th International Symposium on Graph Drawing (GD 2006)*, volume 4372 of *Lecture Notes in Computer Science*, pages 162–173. Springer-Verlag, 2006. doi:10.1007/978-3-540-70904-6_17.
- D. A. Holton and J. Sheehan. *The Petersen Graph*, volume 7 of *Australian Mathematical Society Lecture Notes*. Cambridge University Press, 1993.
- J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973a. doi:10.1137/0202012.
- J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973b. doi:10.1145/362248.362272.

- J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974. doi:10.1145/321850.321852.
- R. Jayakumar, K. Thulasiraman, and M. N. S. Swamy. $O(n^2)$ algorithms for graph planarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(3):257–267, 1989. doi:10.1109/43.21845.
- M. Jünger and P. Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16(1):33–59, 1996. doi:10.1007/BF02086607.
- M. Jünger and P. Mutzel, editors. *Graph Drawing Software*. Mathematics and Visualization. Springer, 2004.
- M. Jünger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design*, 17(7):609–612, 1998. doi:10.1109/43.709399.
- G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996. doi:10.1007/BF02086606.
- T. Kerkhof. Algorithmen zur Bestimmung von guten Graph-Einbettungen für orthogonale Zeichnungen. Master’s thesis, University of Dortmund, 2007.
- M. Klešč. On the crossing numbers of Cartesian products of stars and paths or cycles. *Mathematica Slovaca*, 41:113–120, 1991. URL <http://dml.cz/dmlcz/136523>.
- M. Klešč. The crossing numbers of certain Cartesian products. *Discussiones Mathematicae Graph Theory*, 15(1):5–10, 1995.
- M. Klešč. The crossing number of $K_{2,3} \times P_n$ and $K_{2,3} \times S_n$. *Tatra Mountains Mathematical Publications*, 9:51–56, 1996.
- M. Klešč. The crossing numbers of products of 5-vertex graphs with paths and cycles. *Discussiones Mathematicae Graph Theory*, 19:59–69, 1999a.
- M. Klešč. The crossing number of $K_5 \times P_n$. *Tatra Mountains Mathematical Publications*, 18:605–614, 1999b.
- M. Klešč. The crossing numbers of Cartesian products of paths with 5-vertex graphs. *Discrete Mathematics*, 233(1–3):353–359, 2001. doi:10.1016/S0012-365X(00)00251-X.
- M. Klešč. Some crossing numbers of products of cycles. *Discussiones Mathematicae Graph Theory*, 25:197–210, 2005.
- M. Klešč and A. Kocúrová. The crossing numbers of products of 5-vertex graphs with cycles. *Discrete Mathematics*, 307(11–12):1395–1403, 2007. doi:10.1016/j.disc.2005.11.077.

- M. Klešč, R. B. Richter, and I. Stobert. The crossing number of $C_5 \times C_n$. *Journal of Graph Theory*, 22(3):239–243, 1996. doi:10.1002/(SICI)1097-0118(199607)22:3<239::AID-JGT4>3.0.CO;2-N.
- C. Kuratowski and G. T. Whyburn. Sur les éléments cycliques et leurs applications. *Fundamenta Mathematicae*, 16:305–331, 1930.
- J. A. La Poutré. Alpha-algorithms for incremental planarity testing. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC '94)*, pages 706–715, 1994. doi:10.1145/195058.195439.
- T. Lengauer. Hierarchical planarity testing algorithms. *Journal of the ACM*, 36(3):474–509, 1989. doi:10.1145/65950.65952.
- A. Liebers. Planarizing graphs — A survey and annotated bibliography. *Journal of Graph Algorithms and Applications*, 5(1):1–74, 2001. URL <http://www.cs.brown.edu/sites/jgaa/accepted/01/Liebers01.5.1.pdf>.
- P. C. Liu and R. C. Geldmacher. On the deletion of nonplanar edges of a graph. *Congressus Numerantium*, 24:727–738, 1979.
- S. MacLaine. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3(3):460–472, 1937. doi:10.1215/S0012-7094-37-00336-3.
- M. Mader. Planar graph drawing. Master's thesis, University of Konstanz, 2008.
- S. Masuda, T. Kashiwabara, K. Nakajima, and T. Fujisawa. On the NP-completeness of a computer network layout problem. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 292–295, 1987.
- S. Masuda, K. Nakajima, T. Kashiwabara, and T. Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Transactions on Computers*, 39(1):124–127, 1990. ISSN 0018-9340. doi:10.1109/12.46286.
- S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 193–205. ACM Press, 2008. doi:10.1145/1375581.1375606.
- S. A. McCamant. *Quantitative Information-Flow Tracking for Real Systems*. PhD thesis, Massachusetts Institute of Technology, 2008. URL <http://groups.csail.mit.edu/pag/pubs/infowflow-mccamant-phdthesis.pdf>.
- K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996. doi:10.1007/BF01940648.
- K. Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

- G. L. Miller and V. Ramachandran. A new graph triconnectivity algorithm and its parallelization. *Combinatorica*, 12(1):53–76, 1992. doi:10.1007/BF01191205.
- P. Mutzel and R. Weiskircher. Optimizing over all combinatorial embeddings of a planar graph. In G. Cornuéjols, R. Burkard, and G. Woeginger, editors, *Proceedings of the 7th Conference on Integer Programming and Combinatorial Optimization (IPCO '99)*, volume 1610 of *Lecture Notes in Computer Science*, pages 361–376. Springer Verlag, 1999. doi:10.1007/3-540-48777-8_27.
- P. Mutzel and R. Weiskircher. Computing optimal embeddings for planar graphs. In D. Z. Du, P. Eades, V. Estivill-Castro, X. Lin, and A. Sharma, editors, *Proceedings of the 6th Annual International Conference on Computing and Combinatorics (COCOON '00)*, volume 1858 of *Lecture Notes in Computer Science*, pages 95–104. Springer Verlag, 2000. doi:10.1007/3-540-44968-X_10.
- P. Mutzel and R. Weiskircher. Bend minimization in orthogonal drawings using integer programming. In O. H. Ibarra and L. Zhang, editors, *Proceedings of the Eighth Annual International Conference on Computing and Combinatorics (COCOON 2002)*, volume 2387 of *Lecture Notes in Computer Science*, pages 484–493. Springer-Verlag, 2002. doi:10.1007/3-540-45655-4_52.
- P. Mutzel and T. Ziegler. The constrained crossing minimization problem. In J. Kratochvil, editor, *Proceedings of the 7th International Symposium on Graph Drawing (GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 175–185. Springer-Verlag, 1999. doi:10.1007/3-540-46648-7_18.
- E. Nardelli and M. Talamo. A fast algorithm for planarization of sparse diagrams. Technical Report R. 105, IASI-CNR, Rome, 1984.
- S. C. North. Incremental layout in dynadag. In *Proceedings of the Symposium on Graph Drawing (GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418. Springer-Verlag, 1996. doi:10.1007/BFb0021824.
- M. Pizzonia. Minimum depth graph embeddings and quality of the drawings: An experimental analysis. In P. Healy and N. S. Nikolov, editors, *Proceedings of the 13th International Symposium on Graph Drawing (GD 2005)*, volume 3843 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 2005. doi:10.1007/11618058_36.
- M. Pizzonia and R. Tamassia. Minimum depth graph embedding. In M. Paterson, editor, *Proceedings of the 8th Annual European Symposium on Algorithms (ESA 2000)*, volume 1879 of *Lecture Notes in Computer Science*, pages 356–367. Springer-Verlag, 2000. doi:10.1007/3-540-45253-2_33.
- B. R. Richter and G. Salazar. The crossing number fo $P(N, 3)$. *Graphs and Combinatorics*, 18(2):381–394, 2002. doi:10.1007/s003730200028.

- R. D. Ringeisen and L. W. Beineke. The crossing number of $C_3 \times C_n$. *Journal of Combinatorial Theory, Series B*, 24(2):134–136, 1978. doi:10.1016/0095-8956(78)90014-X.
- R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987. doi:10.1137/0216030.
- R. Tamassia. Constraints in graph drawing algorithms. *Constraints*, 3(1):87–120, 1998. doi:10.1023/A:1009760732249.
- R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, 1988. doi:10.1109/21.87055.
- R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.
- P. Turán. A note of welcome. *Journal of Graph Theory*, 1(1):7–9, 1977. doi:10.1002/jgt.3190010103.
- W. T. Tutte. *Connectivity in graphs*, volume 15 of *Mathematical Expositions*. University of Toronto Press, 1966.
- I. Vrt'o. Crossing numbers of graphs: A bibliography, 2009. URL <ftp://ftp.ifi.savba.sk/pub/imrich/crobib.pdf>.
- J. Širáň. Additivity of the crossing number of graphs with connectivity 2. *Periodica Mathematica Hungarica*, 15(4):301–305, 1984. doi:10.1007/BF02454162.
- R. Weiskircher. *New Applications of SPQR-Trees in Graph Drawing*. PhD thesis, Universität des Saarlandes, 2002.
- H. Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34(2):339–362, 1932a. URL <http://www.jstor.org/stable/1989545>.
- H. Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54(1):150–168, 1932b. URL <http://www.jstor.org/stable/2371086>.
- T. Ziegler. *Crossing Minimization in Automatic Graph Drawing*. PhD thesis, Max-Planck-Institut für Informatik, Saarbrücken, 2000.

Index

- , 8
- $D(v)$, 47
- \cap , 8
- \cup , 8
- \emptyset , 7
- \Rightarrow^* , 8
- $\phi(e)$, 49
- \rightarrow , 9
- $\Psi_B(c)$, 138
- \subseteq , 8
- \times , 20
- $\xrightarrow{*}$, 9
- $cr(G)$, 13
- G^* , 12
- K_n , 7
- $P(m, \ell)$, 20
- $T(G)$, 31
- $V(E')$, 8
- $v \hookrightarrow w$, 10
- $v \rightarrow w$, 10

- Ackermann function, 16
- acyclic, 9
- $Adj(v)$, 47
- adjacent
 - faces, 11
 - vertices, 7
- aesthetic criteria, 1
- allocation node, 34
 - proper, 34
- ancestor, 9
- approximation factor, 17
- arc, 9
 - B-arc, 10
 - incoming, 9
 - outgoing, 9
 - T-arc, 10
- articulation point, 8

- artificial graphs, **100**, 122
- atoms, 24

- B-node, 22
- BC-tree, 16, **22**
- bend minimization, 74
- biconnected, 8
- block, **22**, 22–24
 - graph, 22
- Block graphs, 147
- block-nesting depth, **129**, 141
- bond, 30
 - triple, 40
- branch graph, 24
- branch-and-cut, 15
- bridge, **8**, 22

- C-node, 22
- c-nodes, 157
- CCMP, 17
- child, 9
- CMP, 73
- coarseness, **13**, 89
- compaction, 3
- component, 8
- component weight, 132
- connected
 - 0-connected, 8
 - 1-connected, 8
 - k -connected, 8
 - triply, 24
- constrained edge insertion, 17
- constraint nodes
 - oriented, 157
- constraint-nodes, 157
 - grouping, 157
 - mirror, 157
- constraints, 155

- crossing, 11
- crossing minimization, 73
 - approximation, 74
 - constrained, 99
 - NP-hard, 74
 - problem, 73
- crossing number, 13, 89
 - weighted, 93
- crossing number problem, 13, **73**
- crossing vertex, 13
- crossing weight, 93
- CrossingMinimizationModule, 100
- curve, 11
- cut, 9
 - capacity, 9
 - minimum, **9**, 89
 - st*-cut, **9**, 89
- cut vertex, **8**, 22, 23
- cycle, 8
 - directed, 9
- $\deg(f)$, 11
- $\deg(v)$, 7
- degree
 - face, 11
 - vertex, 7
- density, 64
- depth, 128
 - node, 9
 - rooted tree, 9
- depth-first search, 10
- descendant, 9
 - first, 52
- DFS, **10**, 22–24
 - number, 10, 46
 - tree, **10**, 22, 52
- digraph, 9
- DIMACS library, 65
- drawing
 - orthogonal, 1
- DynamicPlanarSPQRTree, 64
- DynamicSPQRTree, 64
- ec-edge insertion path, 163
 - optimal, 163
- ec-expansion, 158
- ec-planar, **158**, 160, 161
 - ec-expansion, 159
 - embedding, 158
- ec-traversing costs, 164
- edge, 7
 - boundary edges, 11
 - multiple, 7
 - real, 34
 - virtual, 26, **27**, 27, 34
- edge insertion
 - fixed embedding, **17**, 97
 - general framework, 97
 - variable embedding, 97
- edge insertion path, 76
 - length, 76
 - optimal, **76**, 85
- edge insertion problem, 14, **17**
- EIP, 14, **17**
- embedding, 12
 - combinatorial, **11**, 26
 - planar, 12
- embedding constraint, 157
- endvertex, 7
- ESTACK, 59
- Euler, 12
 - formula, 12
- expanding, 34
- expanding a face, 133
- expansion edges, 158
- expansion graph, 34
- $expansion(e)$, 34
- $expansion^+(e)$, 34
- extended dual BC-tree, 129
- extended dual graph, 17
- extension, 141
- face, 11
 - external, 11
- FastPlanarSubgraph, 100
- FixedEmbeddingInserter, 100
- forest, 8
- free edges, 173
- gauge, 128
- gc-nodes, 157
- graph, 7

- Cartesian product, 19
 - complete, 7
 - connected, 8
 - contains, 8
 - directed, 9
 - disconnected, 8
 - drawing, 11
 - dual, 12
 - empty, 7
 - planar, 1, **11**, 22
 - planar drawing, 11
 - simple, 7
 - underlying undirected, 9
- Hachul library, 65
- hub, 158, 161
- in-degree, 9
- incident, 7
- incremental planarity testing, 16
- $\text{indeg}(v)$, 9
- independent, 8
- inner wheel gadget faces, 158
- Kandinsky algorithm, 127
- leaf, 9
- length
 - cycle, 8
 - path, 8
- lowpt-value, 23
- lowpt1-value, 47
- lowpt2-value, 47
- M-hub, 158
- MacLane, 24, 31
- maximal planar subgraph, **15**, 16, 96
- MaximalPlanarSubgraphSimple, 100
- maximum external face problem, 131
- maximum planar subgraph, 13
- maximum planar subgraph problem, 14, **15**, 96
- maximum weight
 - of embedding, 132
- MBNDE, 141
- mc-nodes, 157
- MEF, 131
- Menger's theorem, 8
- merge operation, 41
- $\text{mincut}_{s,t}(G)$, 9
- minimum block-nesting depth, 141
- MPSP, 14, **15**
- near-planar graph, 86
- neighbor, 7
- node, 9
- non-planar core, **92**, 100
 - virtual edge, 92
- non-planar core reduction, 89, 109
- NonPlanarCore, 100
- O-hub, 158
 - conflicting, 161
 - oriented correctly, 159
- observes embedding constraints, 157
- oc-nodes, 157
- OEIP, 75
 - approx. for crossing number, 88
 - arbitrary graphs, 86
 - biconnected graphs, 78
 - connected graphs, 84
- OGDF, 63, 100, 148
- one-edge insertion problem, 75
 - with embedding constraints, 156, **164**
- orientation problem, 173
- orthogonalization, 2
- out-degree, 9
- $\text{outdeg}(v)$, 9
- outerplanarity, 128
- P-node, 34
- parent, 9
- path, 8
 - directed, 9
 - simple, 8
- PATHSEARCH, 49
- pertinent graph, 32, 34
- pertinent node, 34
- $\text{pertinent}(v)$, 34
- PertinentGraph, 64
- Petersen graph, 20
 - generalized, **20**, 102
- planar 2-component, 91

- maximal, 91
- planar st -component, 91
 - trivial, 91
- planar subgraph, 13, 14
- planarBiconnectedGraph(), 64
- planarity test, 16
- planarization method, 3, **13**, 74, 88
- planarized representation, **14**, 73
- point, 11
- poles, 34
- polygon, 30
- port constraints, 155
- PQ-tree, 16

- Q-node, 34
- QueryPerformanceCounter(), 66

- R-node, 34
- radius, 128
- randomBiconnectedGraph(), 64
- reference edge, 33
- representative, 84
- Rome graphs, **64**, 88, 100, 102, 148
- root, 9

- S-node, 34
- segment, 42
- self-loop, 7
- separation pair, 26
 - type-1, 42, 55
 - type-2, 42, 55
- side constraints, 155
- sink, 9
- Skeleton, 63
- skeleton, 33
- skeleton edge
 - represents, 78
- skewness, **13**, 89
- source, 9
- source node, 9
- spanning tree, 15, 16
- SPLIT, 61
- split class, 26
- split class graph, 26
- split components, 40
- split graph, **27**, 28
 - size, 28
- split operation, 27
- split pair, 26, 32
 - dominated, 32
 - maximal, 33
- SPQR-tree, 16, **34**
 - dynamic updates, 32
 - pre-SPQR-tree, 33
 - size, 39
- standard method, 107
- StaticPlanarSPQRTree, 64
- StaticSPQRTree, 63
- subdivision, 8
- subgraph, 7
 - spanning, 8
- SubgraphPlanarizer, 100
- supergraph, 8

- target node, 9
- thickness, **13**, 89
- topology-shape-metrics approach, 2
- traversing costs, **77**, 89
- traversing path, **77**, 89
- tree, 8
 - rooted, 9
- triangle, 40
- tric. components, **28**, 24–31, 35
 - size, 31
 - unique, 31
- triconnected, 8
- TSTACK, 59
- Turan, Paul, 73
- Tutte, 26, 31
- Tutte split, **27**, 28, 30, 40

- VariableEmbeddingInserter, 100
- vertex, 7
 - inner, 8

- wall, 86
 - poles, 86
- wheel gadget, **158**, 161
- width, 128