technische universität
dortmund

Inserting a Vertex into a Planar Graph

Christian Wolf

Algorithm Engineering Report
TR08-1-002
April 2008
ISSN 1864-4503

Fakultät für Informatik
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Diplomarbeit

Inserting a Vertex into a Planar Graph

Christian Wolf

Januar 2008

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl für Algorithm Engineering (LS11)
Otto-Hahn-Str. 14
44227 Dortmund

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

Acknowledgement

True to Albert Schweitzer[1]

"Vergiß den Anfang nicht, den Dank!"

I want to thank all who have supported and affirmed me during the preparation of this thesis. I thank Prof. Dr. Petra Mutzel for the selection of topics. Especially, I thank my supervisors Dipl.-Ing. Markus Chimani and Dipl.-Inform. Carsten Gutwenger for answering my numerous questions, their helpful ideas and the constructive discussion rounds. Moreover I thank Christina Paßgang for her mental affirmation and encouragement. Particularly, I thank my parents for making my studies generally possible and for any kind of support in the meantime.

[1] Theologian, musician, philosopher, and physician (1875-1965).

Short Abstract

This diploma thesis presents a first algorithm for the optimization problem to compute a combinatorial embedding among all combinatorial embeddings of a planar graph wherein a new vertex and edges between it and vertices of the graph can be inserted into with the minimum number of crossings. The algorithm uses the dynamic programming paradigm and profits from the data structures SPQR- and BC-tree substantially. The solution of this problem and therewith the algorithm are of interest within a modified planarization method for drawing a non-planar graph. Thereby in contrast to the common planarization method a planar representation of the non-planar graph to be drawn is obtained by deleting vertices including their incident edges and reinserting them in a certain way. Although many optimization problems over the set of all combinatorial embeddings of a planar graph have emerged as $\mathcal{NP}$-hard, the algorithm succeeds in solving the problem investigated here in polynomial worst-case running time.

Kurzfassung

Diese Diplomarbeit stellt einen ersten Algorithmus für das Optimierungsproblem vor, eine kombinatorische Einbettung unter allen kombinatorischen Einbettungen eines planaren Graphen zu berechnen, in die ein neuer Knoten und Kanten zwischen diesem und den Knoten des Graphen kreuzungsminimal eingefügt werden können. Der Algorithmus benutzt das Paradigma der dynamischen Programmierung und profitiert wesentlich von den Datenstrukturen SPQR- und BC-Baum. Die Lösung dieses Problem und damit der Algorithmus sind im Rahmen einer modifizierten Planarisierungsmethode zum Zeichnen eines nicht-planaren Graphen von Interesse. Bei dieser wird im Gegensatz zur üblichen Planarisierungsmethode eine planare Repräsentation des nicht-planaren zu zeichnenden Graphen erhalten, indem Knoten inklusive ihrer inzidenten Kanten gelöscht und auf gewisse Weise wiedereingefügt werden. Obwohl sich viele Optimierungsprobleme über der Menge aller kombinatorischen Einbettungen eines planaren Graphen als $\mathcal{NP}$-schwierig erwiesen haben, gelingt dem Algorithmus die optimale Lösung des hier untersuchten Problems in polynomieller worst-case Rechenzeit.

Eidesstattliche Erklärung

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

(Christian Wolf)

Contents

List of Figures

List of Algorithms

1 Introduction

In many real-life situations graphical representations are used to illustrate information. The goal of information illustration is to clarify relations between single information objects, to make structures visible and to get a more holistic view on the information. Mind and concept maps are well known examples for information illustration. In [15] Fleischer and Hirsch choose another profane example to point out the importance for displaying information. They ask for an adequate representation of a subway map. Imagine, a subway map would be represented in textual form like "Train 1 runs from station A to station B via stations C, D, and E. At station D you can change to trains 3 and 11, etc.". This form of representation would be very confusing and information search would be rather difficult. Instead, a better way is to draw a nice "picture" which easily shows, e.g., where an interchange facility exists or which stations a train visits (cf. Figure 1).

In a more abstract mathematical way, mind, concept and subway maps are graphical representations of graphs. A *graph* is a structure consisting of a set of objects, called *vertices*, and a set of pairwise relations between vertices, called *edges*. In a mind or concept map the vertices are the individual information objects and the edges the connections between them. In a subway map the vertices are the stations and the edges are the rail links between the stations. The graphical representation of a graph is called a *graph drawing* or simply *drawing*. Usually, a graph is drawn into the plane by drawing a point, circle or any other geometric figure for each vertex and a curve between two geometric figures for each edge (cf. Figure 1.2).

Nowadays the spectrum of applications for graphs and especially graph drawings is wide. The following list gives an overview of important application fields:

- *Hard- and software development and specification*: In particular if huge software systems are developed, graphs and diagrams visualize relationships or data flows between objects. Examples are UML (Unified Modeling Language) diagrams like class, inheritance, activity or flow diagrams as well as finite state machines.

- *Database design:* (S)ER ((Structured) Entity Relationship) diagrams model the connection between tables of relational databases and how many records interact with others.

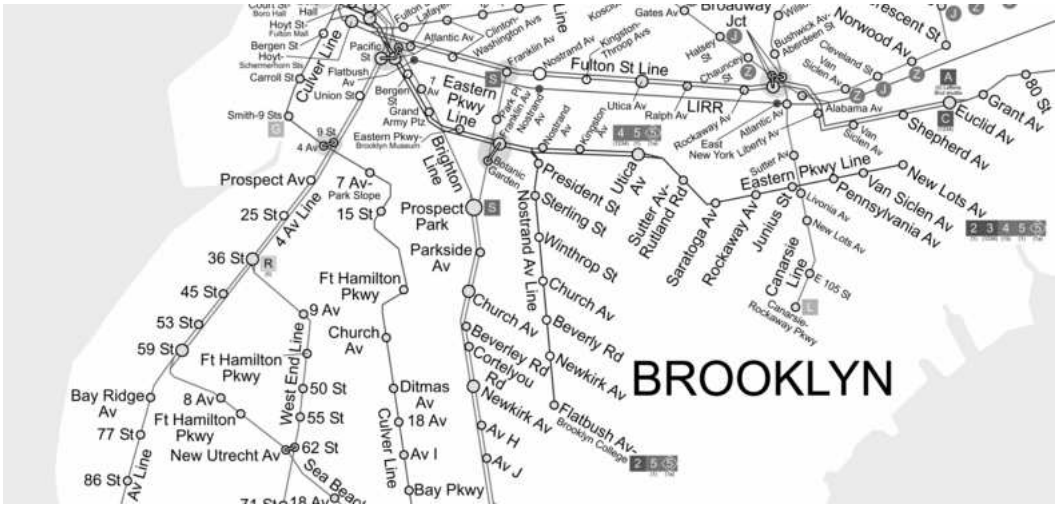


Figure 1.1: Typical subway map, here a detail of the subway map of New York City.

- *Computer science in general:* Petri nets as a representation of distributed systems, network design and visualization of networks, binary decision diagrams as a data structure for Boolean functions, model checking and reasoning.
- *Social sciences:* Here, graphs model genealogy trees and social networks for example.
- *VLSI (Very Large Scale Integration) Design:* Chip designer use graphs as wired schemes between electronic components.

With a growing complexity of graphs besides the additional demand for certain constraints on their drawings, it is desirable to generate such drawings automatically with the aid of computers and special graph drawing software. The relatively modern research field of *automatic graph drawing* addresses this task. It is concerned with the design, analysis, implementation and evaluation of appropriate graph drawing algorithms. Thereto automatic graph drawing combines graph theoretic, algorithmic and combinatorial aspects. Introductory information about automatic graph drawing can be found in, e.g., [6, 15, 27], further reading is given, e.g., in [24].

Depending on its purpose a drawing must fulfill certain criteria. There exists a range of commonly accepted criteria characterizing a "good" graph drawing, see the list below. Usually, these criteria are known under the term *aesthetic criteria*. But these criteria are also important from a technical view, for example, in VLSI design.

- *Minimum number of (edge) crossings:* Edges that cross each other make it difficult for the reader to follow a connection between two objects (cf. Figure 1.2(a), 1.2(b)). Increasing the number of edge crossings in a drawing decreases the understandability [29].

  In wired schemes an edge crossing represents a wire crossing. On a chip such wire crossings are resolved by routing one of the wires to a second layer immediately before the crossing. After the wire has passed the crossing point it can be routed back to the primary layer. Layer changes are realized with contact cuts that need a large area and thus increase the overall size of the layout.

- *Orthogonality:* Vertices and edges can be thought of as fixed on a grid. Edges consist only of vertical and horizontal segments (cf. Figure 1.2(c)). Orthogonal drawings usually look much tidier than drawings with arbitrarily curved edges. A plausible example for the application of orthogonal drawings is architectural floor plan design [15].

- *Minimum number of bends:* This criterion is particularly desirable in orthogonal drawings. The reader can follow an edge with few bends much more easier than an edge with an arbitrary number of bends.

  Moreover bend minimization is a technical criterion. In VLSI design bends in wires can cause technical problems [15].

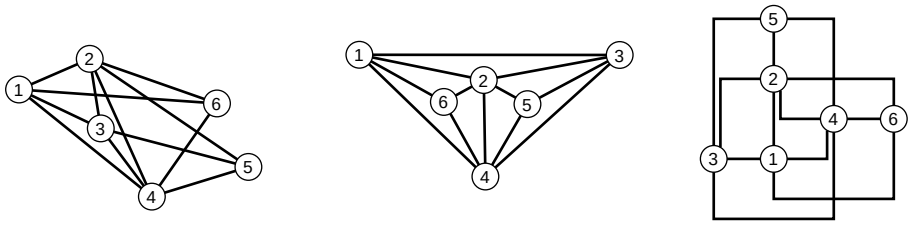
- *Angle maximization:* If edges incident to one vertex lie very closely and even parallel to each other, the reader cannot distinguish between single edges.

- *Symmetry:* The abstract graph itself can hide symmetries. So it is a task to make symmetries visible in a drawing. Otherwise they might be undiscovered.

- *Length minimization:* This criterion concerns the length of edges and distances between vertices. It is also another technical criterion. In VLSI design short edges are required to guarantee short circuit times and an overall small size of the chip.

- *Area minimization:* A drawing is more concise if the drawn vertices and edges fill the drawing area with homogeneous density.

- *Layered and hierarchical drawings:* Activity, flow or SER diagrams have a reading direction and therefore are typically visualized in hierarchies, i.e., vertices are restricted to distinct layers.



(a) (b) (c)

Figure 1.2: Different drawings of the same graph with different number of crossings. Drawing (a) has 6, drawing (b) has none and drawing (c) has 2 crossings.

The criterion of a minimum number of crossings in a drawing takes on a special position among the mentioned criteria. From the perspective of human cognition Purchase has verified the importance of this criterion with experimental studies [29, 30]. She has investigated the influence on the readability and interpretability of graph drawings with subjects and concludes that edge crossings affect the human reader at most [29]. In VLSI design, there is no doubt about the relevance of wired scheme layouts with a minimum number of crossings since nowadays thousands of transistor-based circuits have to be integrated into a single chip.

The class of graphs that can be drawn with no crossings at all, and therefore fulfill the crossing minimization criterion entirely, is the class of *planar graphs*. In graph theory as well as in the area of automatic graph drawing, scientists have spent a lot of research on planar graphs. Various algorithms for drawing planar graphs exist, for example, such ones that produce grid, straight line or orthogonal drawings of planar graphs. An overview about concepts of algorithms for drawing planar graphs is given by Weiskircher in [34].

Unfortunately, most graphs are non-planar and cannot be drawn without crossings. But the problem to compute a drawing with the minimum number of crossings, the so-called *crossing minimization problem*, is $\mathcal{NP}$-hard in general. This result goes back to Garey and Johnson. They have proved that the problem to compute the *crossing number*, that is the minimum number of crossings over all possible drawings of a graph, is $\mathcal{NP}$-complete [17]. Though several $\mathcal{NP}$-hard problems arising in automatic graph drawing have been solved in practice, the crossing minimization problem is extremely hard [7]. However, a first branch-and-cut approach that is able to compute the crossing number of sparse graphs on up to 40 nodes within a time limit of five minutes has been presented in [8]. An extended approach using the column generation technique is suggested in [10] and can compute the crossing number of larger graphs on up to ≈ 70 nodes within

the same time limit.

Nevertheless, in practice there is a need to tackle this problem and heuristics are employed. The most promising and frequently used heuristic is the *planarization method* introduced firstly by Batini et al. [1]. The planarization method performs two major steps and can be roughly described as follows. In the first step a set of edges in the non-planar graph which has to be drawn is deleted and a planar subgraph is obtained. In the second step, the deleted edges are reinserted one-by-one into this planar subgraph while trying to produce only few crossings. Crossings that appear in each edge reinsertion step are replaced by artificial vertices to maintain a planar subgraph for the next edge reinsertion step. Having added all edges in this way, the result is a planar graph which is also referred to as *planar representation* of the original graph. Then a drawing algorithm for planar graphs is applied to this planar representation. Finally, the artificial vertices are replaced by edge crossings and a drawing of the original graph with hopefully few crossings is obtained. Each these steps can be seen as a separate optimization problem, and there exist various solutions to solve them.

But the two steps of the planarization method are also realizable in another more drastic way. A planar subgraph of the non-planar graph to be drawn can be obtained by deleting a set of vertices together with their incident edges. Then the deleted vertices are reinserted one-by-one together with their incident edges at once. This thesis deals with one such vertex reinsertion step in two possible variants. The more interesting and more difficult one takes a possibly exponential number of drawings of the given planar (sub)graph in a certain topological sense into account. As will be shown both problem variants can be solved optimally in polynomial time. Vertex deletion and reinsertion can also be applied as a post-processing step after the usual planarization heuristic for improving the quality of the drawing.

In Chapter 2 we will introduce the formal fundamentals which are needed throughout the thesis. Important terms like *graph*, *k-connectivity* and *embedding* are defined as well as a crucial data structure, the *SPQR-tree data structure*. Chapter 3 throws light on the planarization heuristic in more detail with an emphasis on the edge reinsertion phase. Chapter 4 represents the main part of this thesis defining what we understand under the problem of *Inserting a Vertex into a Planar Graph* and especially presenting polynomial time algorithms for its two problem variants. Chapter 5 concludes this thesis by giving a summary of the results and an outlook of future work.

2 Preliminaries

This chapter introduces the formal fundamentals and notations which are essential throughout this thesis. Firstly, basic terms and a few statements of graph theory are given followed by some definitions around the term *embedding*. Further, the central data structure of this thesis, the SPQR-tree data structure, is defined and an exemplary generation of an SPQR-tree is given. Finally, some of its properties are discussed. Further definitions and notations are introduced in corresponding chapters.

2.1 Graph Theory

The following definitions are based on [20] unless stated otherwise. They might be slightly modified, merged or listed in an order that fits our purposes.

Definition 2.1 ((Multi)Graph) *Let V be a finite set and $E \subseteq \{\{v, w\} \mid v, w \in V,\ v \neq w\}$ be a (multi)set. Then $G = (V, E)$ is a (multi)graph.*

Let us consider a (multi)graph $G = (V, E)$. The elements $v \in V$ are called *vertices* and the elements $e \in E$ are called *edges* of G. Two vertices are *adjacent* if there exists an edge between them and two edges are adjacent if they share a common vertex. An edge $e = \{v, w\}$ *connects* the vertices v, w, also called *endpoints* of e, and v, w are said to be *incident* to e as well as e is said to be incident to v, w. The number of edges incident to vertex v is its *degree*, denoted with $d(v)$.

G is called directed if all $e \in E$ are directed and $e = \{v, w\}$ is said to be directed if v, w are ordered. Otherwise G is undirected. Note, the definition of a (multi)graph prohibits *self-loops*, that is a vertex must not relate to itself. A *subgraph* of G is another graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. We denote the subgraph relation with $G' \subseteq G$. G' is a *proper* subgraph of G if $G' \neq G$. If G' is a (proper) subgraph of G, then G is a *(proper) supergraph* of G'.

G' is a *maximal* (*minimal*) subgraph of G with a property Φ if G' satisfies Φ and if no other subgraph $G'' \subseteq G$ exists that is a proper supergraph of G' and satisfies Φ. A subset $V' \subseteq V$ *induces a subgraph* $G' = (V', E')$ of G with $E' := \{e = \{v, w\} \in E \mid v, w \in V'\}$ and a subset $E'' \subseteq E$ induces a subgraph $G'' := (V'', E'')$ with $V'' := \{v \in V \mid v \text{ is endpoint of an edge } e \in E''\}$. G is a *weighted* graph, if vertices or edges in G are weighted with any numbers.

A *walk* $\mathcal{W} = v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k$ of G is an alternating series of vertices $v_i \in V$ $(i = 0, \ldots, k)$ and edges $e_j = \{v_{j-1}, v_j\} \in E$ $(j = 1, \ldots, k)$ *connecting* v_0 with v_k. $\mathcal{W}$ can also be described by listing only its vertices or edges. $\mathcal{W}$ is *closed* if $v_0 = v_k$, otherwise it is *open*. $\mathcal{W}$ is a *trail* if all edges in it are distinct and a *path* if additionally all vertices in it are distinct. The *length of a path* is defined as the number of edges in it. A closed path is called a *cycle* for $k \geq 3$.

Definition 2.2 (k-Connectivity) *A graph $G = (V, E)$ is k-connected for $k > 0$ if for any pair of vertices $v, w \in V$ there exist k different paths connecting v and w that are pairwise vertex disjoint except in their endpoints.*

An equivalent definition of a k-connected graph G is that no set of $k-1$ elements exists, each a vertex or an edge, whose removal disconnects G. Such a set is called *separating* $(k-1)$*-set*. 1-connected, 2-connected, and 3-connected graphs are usually called connected, biconnected, and triconnected, respectively.

We denote by $G - x$ and $G - Y$ the deletion of either a vertex or an edge x in G and the deletion of either a set of vertices or edges Y in G, respectively. If a vertex is deleted also all its incident edges are deleted. Separating 1-sets and 2-sets of vertices are called *cut-vertices* and *separation pairs*, respectively. A maximal connected subgraph of a graph is called *connected component* and a maximal biconnected subgraph of a graph is called *biconnected component*, *bicomp* or *block*.

Note, a k-connected component for $k > 2$ is not just a maximal subgraph that is k-connected. Since we will deal at most with triconnected components (shortly *tricomps*), we now give their definition taken from [11]. The reader can find a generally recursive definition that creates the k-connected components of a graph for any $k > 2$ in [4]. Let G be a biconnected graph. If G is triconnected, then G itself is the unique tricomp of G. Otherwise, let the vertices u, v build a separation pair $\{u, v\}$ of G. We partition E into two disjoint sets E_1 and E_2 with $|E_1|, |E_2| \geq 2$ such that the subgraphs G_1 and G_2 induced by them have only vertices u and v in common. This decomposition process is continued recursively on $G_1 + e$ and $G_2 + e$ with edge $e = \{u, v\}$ until no decomposition is possible. The resulting graphs are each either a triconnected graph, or a multigraph consisting of two vertices with three multiple edges between them (triple bond), or a cycle of length three (triangle). The tricomps of G are obtained from such graphs by merging the triple bonds into maximal sets of multiple edges (bonds), and the triangles into maximal cycles (polygons).

Definition 2.3 (Tree) *A graph T is a tree if T is connected and cycle-free.*

Vertices of T are referred to as *nodes*. T can have a designated node called the *root* of T. Then T is called a *rooted tree* and has an orientation towards or

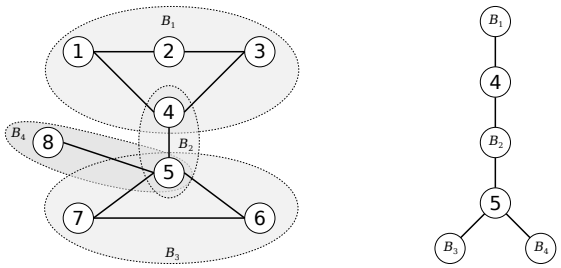


Figure 2.1: Example of a connected graph and its BC-tree. Cut-vertices are $4, 5$.

away from the root. The orientation can be represented by directed edges. If $\{v, w\}$ is a directed edge in a tree, then w is called a *child (node)* of v and v is called the *parent (node)* of w. A node v with $d(v) = 1$ is a so-called *leaf*. The tree representing the relationship between blocks and cut-vertices of a connected graph is the *block-cut-vertex-tree* or shortly *BC-tree* (cf. Figure 2.1).

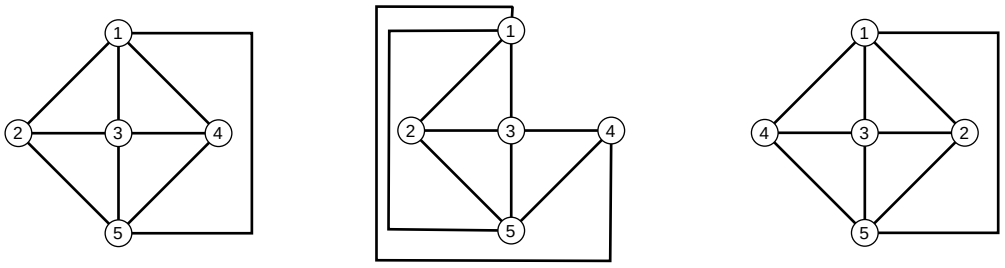
Definition 2.4 (BC-tree) *Let $G = (V, E)$ be a connected graph. A BC-tree of G contains a B-node for each block in G and a C-node for each cut-vertex v in G. A B-node b and a C-node c are connected by an edge if and only if c belongs to the corresponding block of b in G.*

So far we considered graphs as an abstract mathematical structure. As we already know from Chapter 1 graphs can have a graphical representation. Below we give a formal definition of a *drawing* and a few definitions concerning combinatorial aspects of drawings. The following four definitions are based on [35].

Definition 2.5 ((Planar) Drawing) *A drawing Γ of a graph $G = (V, E)$ is a function that maps each $v \in V$ to a point and each $\{v, w\} \in E$ to a curve $\Gamma(v, w)$ with endpoints $\Gamma(v)$ and $\Gamma(w)$ in the plane $\mathbb{R}^2$. Γ is* planar*, if all pairs of disjoint vertices are mapped to different points and curves does not cross each other except in their endpoints.*

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

Deleting a planar drawing of G induces *regions* in the plane. They are called *faces* and a distinction is made between bounded *internal faces* and one unbounded *external face*. We say edges and vertices defining a face *lie on the boundary* of it or *border* it. A face can be described by giving an anti-clockwise ordered list of the edges bordering that face.



(a) (b) (c)

Figure 2.2: Graph with different embeddings.

A planar drawing of G is also called a *planar geometric embedding* of G. G can have an infinite number of drawings because the plane is infinite. But different drawings can equal in a combinatorial sense. We call two planar drawings Γ and Γ' of G *weakly equivalent*, if for every $v \in V$, the circular order of the incident edges around v in clockwise order is the same in Γ and Γ'. If, additionally, the external faces of Γ and Γ' are equal, the two drawings are *strongly equivalent*.

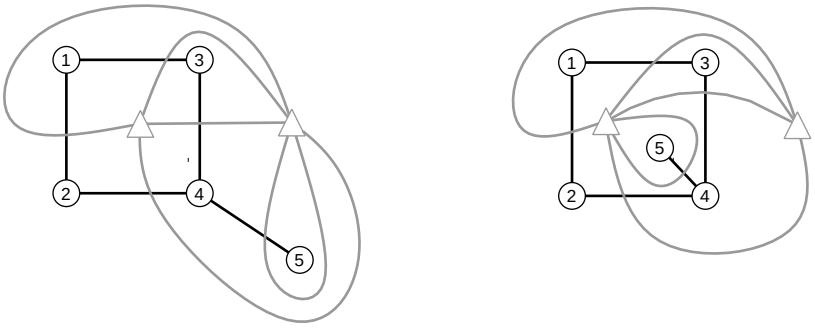
Definition 2.7 (Combinatorial embedding) *A combinatorial embedding* Π *of a planar graph* G *is an equivalence class of its weakly equivalent planar drawings.*

Definition 2.8 (Planar embedding) *A planar embedding* Π *of a planar graph* G *is an equivalence class of its strongly equivalent planar drawings.*

Drawings with the same combinatorial embeddings can look very different and drawings with different combinatorial embeddings can look similar. This is illustrated by Figure 2.2. The drawings in Figure 2.2(a) and 2.2(b) have the same combinatorial, but not the same planar embedding. The drawings in Figure 2.2(a) and 2.2(c) seem to have the same combinatorial embedding, in fact they have not.

The specification of a combinatorial or planar embedding as cyclic adjacency lists does not make any statements about geometric attributes like the position of vertices or the length of edges. From now on, we will not talk about a specific drawing of a graph. It suffices to give a combinatorial or planar embedding Π and to think of any planar drawing that *realizes* it, i.e., the drawing belongs to Π. We use the term embedding both for a combinatorial and planar embedding, if it is clear from context.

Definition 2.9 (Dual graph) *Let* $G = (V, E)$ *be a planar graph and* Π *be a combinatorial embedding of* G *inducing a face set* F*. Let* $F^* = \{f^* \mid f^* \text{ represents } f \in F\}$



(a) (b)

Figure 2.3: Example of a simple planar graph with two distinct combinatorial embeddings and their dual graphs.

and $E^* = \{\{f, f'\} \mid f, f' \in F^* \text{ join the same edge } e \in E\}$ *be. Then* $\Pi^* = (F^*, E^*)$ *is the (geometric) dual graph of* G *with respect to* Π.

Different combinatorial embeddings can have different dual graphs. Furthermore a dual graph is usually a multigraph and can contain self-loops (cf. Figure 2.3).

Planar graphs can also be characterized by KURATOWSKI's prominent theorem. A *subdivision* of an edge $\{v, w\}$ is the replacement of e by two new edges $\{v, v'\}, \{v', w\}$ and a new vertex v'. Two graphs G_1, G_2 are *homeomorphic* if both arise from a graph $G = (V, E)$ by a series of subdivisions of edges $e \in E$, for example two cycles are homeomorphic. A graph $G = (V, E)$ is *bipartite* if V can be partitioned into disjoint sets V_1, V_2 such that only edges $\{v, w\}$ with $v \in V_i$, $w \in V_j$, $i \neq j$ exist. K_n denotes the complete graph on n vertices and $K_{n,m}$ denotes the complete bipartite graph $G = (V_1 \dot{\cup} V_2, E), |V_1| = n, |V_2| = m$.

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

The size of a planar graph is linear in the number of its vertices. This can be concluded from the two following statements adopted from [13].

Theorem 2.2 (Euler's polyhedron formula for the plane) *Let* $G = (V, E)$ *be a planar connected graph with* $|V| \geq 1$, F *be the face set of any embedding of* G. *Then the following equation is true*

$$|V| - |E| + |F| = 2$$

Corollary 2.1 *A planar graph $G = (V, E)$ with $|V| \geq 3$ vertices has at most $3 \cdot |V| - 6$ edges.*

Corollary 2.2 *The size of a planar graph is $\mathcal{O}(|V| + |E|) = \mathcal{O}(|V|)$.*

2.2 Definition of the SPQR-tree Data Structure

An SPQR-tree is a data structure for decomposing a biconnected graph G with respect to its tricomps. For the first time, SPQR-trees were introduced by Di Battista and Tamassia based on ideas by Bienstock and Monma in 1989. Di Battista and Tamassia used SPQR-trees originally for incremental planarity testing [2]. Since then, the SPQR-tree data structure established itself as a crucial instrument for various problems mainly in the fields of planarity testing and graph drawing.

In [11] Di Battista and Tamassia use SPQR-trees for another on-line problem, the maintenance of tricomps. Their algorithm provides a couple of operations on a graph like vertex- and edge-insertion plus one operation, determining whether three vertex-disjoint paths between two vertices exist. Bertolazzi and Di Battista use the data structure in a branch-and-bound algorithm for computing an orthogonal drawing with the minimum number of bends of a biconnected graph. Gutwenger, Mutzel and Weiskircher utilize SPQR-trees for the problem of computing a crossing minimum drawing of a planar graph augmented by an additional edge such that all crossings involve this edge [19]. An overview about the applications of SPQR-trees in graph drawing is given by Mutzel in [28].

We will give a detailed definition below which is adopted from [11]. A slightly modified variant of this definition is given by Weiskircher in [35]. His definition is especially advisable for the reader who is inexperienced with SPQR-trees. But before we start with the main definition, a few more terms are necessary.

Let $G = (V, E)$ be a biconnected graph. A *split pair* of G is either a *separation pair* or a pair of adjacent vertices. A *split component* of a split pair $\{v, w\}$ is either an edge $\{v, w\}$ or a maximal subgraph $G' \subseteq G$ such that $\{v, w\}$ is not a split pair of G' (cf. Figure 2.4). Let $\{s, t\}$ be a split pair of G. A *maximal split pair* $\{v, w\}$ of G with respect to $\{s, t\}$ is such that, for any other split pair $\{v', w'\}$, vertices s, t, v and w are in the same split component. Let $e = \{s, t\} \in E$ be an edge, called the *reference edge*. The SPQR-tree $\mathcal{T}$ of G with respect to e describes a recursive decomposition of G induced by its split pairs:

Definition 2.10 (SPQR-tree) *$\mathcal{T}$ is a rooted ordered tree whose nodes μ are of four types: S, P, Q, R. Each node μ of $\mathcal{T}$ has an associated biconnected multi-*

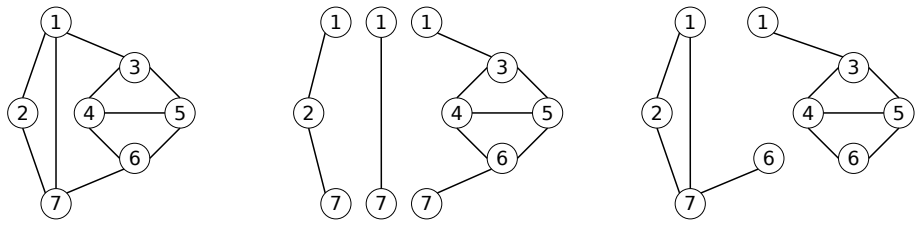


Figure 2.4: Biconnected graph (a), split components with respect to the split pair $\{1,7\}$ (b) and $\{1,6\}$ (as separation pair) (c).

graph, called the skeleton of μ and denoted by skeleton(μ). $\mathcal{T}$ is recursively defined as follows:

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

Parallel Case: *If the split pair $\{s,t\}$ has at least three split components $G_1,\ldots,G_k$ ($k \geq 3$), the root of $\mathcal{T}$ is a P-node μ. Skeleton skeleton(μ) consists of k parallel edges between s and t, denoted $e_1,\ldots,e_k$, with $e_1 = e$.*

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

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

Except for the trivial case, μ has children $\mu_1,\ldots,\mu_k$ in this order, such that μ_i is the root of the SPQR-tree of the (multi)graph $G_i + e_i$ with respect to reference edge e_i ($i = 1,\ldots,k$). The endpoints of edge e_i are called the poles *of node μ_i. Edge e_i is said to be the virtual edge of node μ_i in the skeleton of μ_i and of node μ in the skeleton of μ_i. We call node μ the pertinent node of e_i in the skeleton of*

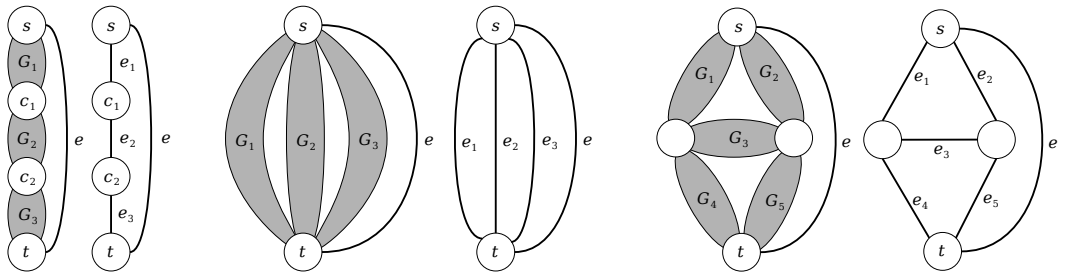


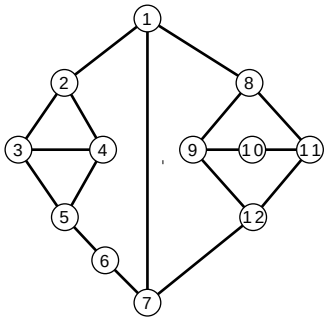
Figure 2.5: Pertinent graphs and their skeletons of an S- (a), P- (b) and R-node (c).

μ_i, and μ_i the pertinent node of e_i in the skeleton of μ. The virtual edge of μ in the skeleton of μ_i is called the reference edge of μ_i.

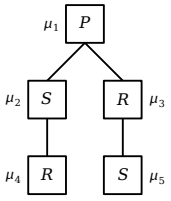
The tree so obtained has a Q-node associated with each edge of G, except the reference edge e. We complete the SPQR-tree by adding another Q-node, representing e, and making it the parent of μ so that it becomes the root.

We need some more handy notations according to an SPQR-tree $\mathcal{T}$. Let μ be a node in $\mathcal{T}$ and e be an edge in the skeleton $skeleton(\mu)$ and let ν be the pertinent node of e. Deleting edge $\{\mu, \nu\}$ in $\mathcal{T}$ splits $\mathcal{T}$ into two connected components. Let $\mathcal{T}_\nu$ be the connected component containing ν, also called *subtree of* $\mathcal{T}$ with root ν. The *expansion graph* of e, denoted with $expansion(e)$, is the graph induced by the edges of G contained in the skeletons of the Q-nodes in $\mathcal{T}_\nu$. $expansion^+(e)$ denotes $expansion(e) + e$. The *pertinent graph* of μ, denoted by $pert(\mu)$, is obtained from $skeleton(\mu)$ by replacing each edge in $skeleton(\mu)$ except for the reference edge with its expansion graph. Note, the expansion graph of e is the same graph as the pertinent graph of ν without its reference edge. A node in $\mathcal{T}$ whose skeleton contains the vertex v is called an *allocation node* of v. Schematic views of pertinent graphs and skeletons of the relevant node types are shown in Figure 2.5.

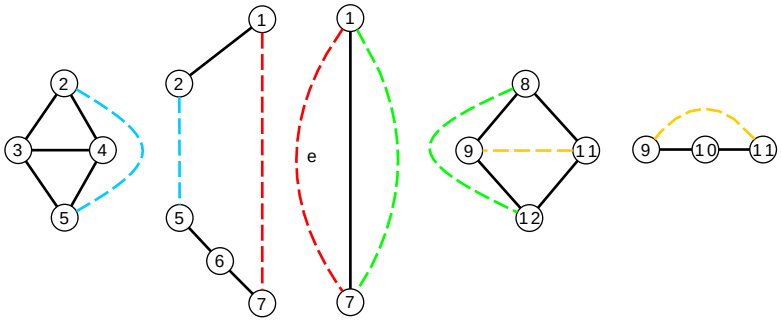
For simplicity reasons, if we deal with an SPQR-tree we will omit Q-nodes and thus will distinguish between virtual and *proper* edges in a skeleton which are edges contained in the underlying graph. Therewith all leaves and the root of an SPQR-tree are nodes of type S, P or R. According to this definition we have to adjust the definition of an expansion graph slightly. The expansion graph of the virtual edge e with pertinent node ν is the graph induced by the edges of G contained as proper edges in the skeletons of all nodes in $\mathcal{T}_\nu$. Figure 2.6 shows an SPQR-tree and clarifies important terms once again.



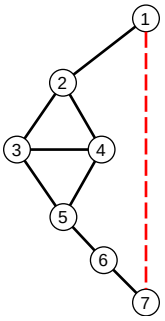
(a) Biconnected graph G.



(b) SPQR-tree $\mathcal{T}$ of G without Q-nodes.



(c) From left to right: skeleton graphs $skeleton(\mu_4)$, $skeleton(\mu_2)$, $skeleton(\mu_1)$, $skeleton(\mu_3)$ and $skeleton(\mu_5)$ associated with the nodes of $\mathcal{T}$. The coloured edges represent virtual edges. E.g., μ_4 and μ_2 are both allocation nodes for vertices 2 and 5.



(d) $expansion^{+}(e) = pert(\mu_2)$.

Figure 2.6: Illustration of certain notions concerning an SPQR-tree.

2.2.1 Example Decomposition

Now, we perform an example decomposition of a biconnected graph G shown in Figure 2.7(a) to illustrate the complex construction of an SPQR-tree. As reference edge for the first decomposition step, we choose the edge $e_0 := \{1, 2\}$. G has three split components with respect to e_0. Let C_1 be the left, $C_2 := e_0$ be the middle and C_3 be the right component in Figure 2.7(b). Hence, the parallel case applies and a P-node becomes the root of the current SPQR-tree with a skeleton consisting of three parallel edges between vertices 1 and 2 (cf. Figure 2.7(c)). Therewith the first decomposition step is completed. Now, the decomposition proceeds recursively for the graphs $G_1 = C_1 + e_1, G_2 = C_2 + e_2, G_3 = C_3 + e_3$ with $e_1 := e_2 := e_3 := e_0$ illustrated from left to right in Figure 2.7(d).

Let us now decompose G_3 with respect to e_3. G_3 has two split components. Let $C_{31} := e_3$ denote the left and C_{32} denote the right component in Figure 2.7(e). C_{32} owns a cut-vertex 5 and thus the series case applies. The skeleton of G_3 is G_3 itself. The blocks of C_{32} are the proper edges $\{1, 5\}$ and $\{2, 5\}$. Therefore the recursive decomposition of G_3 is completed and the SPQR-tree is extended by one S-node (cf. Figure 2.7(f)).

Decomposing G_1 with respect to e_1 generates again two split components. We denote by C_{11} the left and by C_{12} the right component in Figure 2.7(g). C_{11} is biconnected and so the rigid case applies. Now, the maximal split pairs with respect to e_1 have to be determined. The split pairs of G_1 are all edges in G_1 plus the separation pair $\{1, 3\}$. None of the split components of G_1 with respect to $\{1, 3\}$ contains the vertices $1, 2$ and one of the vertices $6, 7$ or 8 (cf. Figure 2.7(h)). Hence all edges with endpoints $6, 7$ or 8 are not maximal split pairs. $\{1, 3\}, \{2, 3\}, \{1, 4\}$ and $\{2, 4\}$ are the maximal split pairs. For each such pair π we have to compute a graph G_π that is the union of the split components of G_1 with respect to π but the one containing e_1. $G_{\{2,3\}}, G_{\{1,4\}}$ and $G_{\{2,4\}}$ are the graphs induced by edges $\{2, 3\}, \{1, 4\}$ and $\{2, 4\}$, respectively. $G_{\{1,3\}}$ is the graph induced by vertex set $\{1, 3, 6, 7, 8\}$. We replace all these graphs by edges and obtain the skeleton of G_1 (cf. Figure 2.7(i)). The current SPQR-tree is extended by one R-node.

Finally, it remains to decompose $G_{\{1,3\}} + e$ with edge $e = \{1, 3\}$ recursively. In this decomposition one time the parallel case and three times the serial case applies. Figure 2.7(j) presents the final SPQR-tree.

2.2.2 Properties of an SPQR-tree

An important property of an SPQR-tree is that it represents all combinatorial embeddings of its underlying graph implicitly. In fact, this means that the SPQR-tree can be used to enumerate all combinatorial embeddings of its underlying

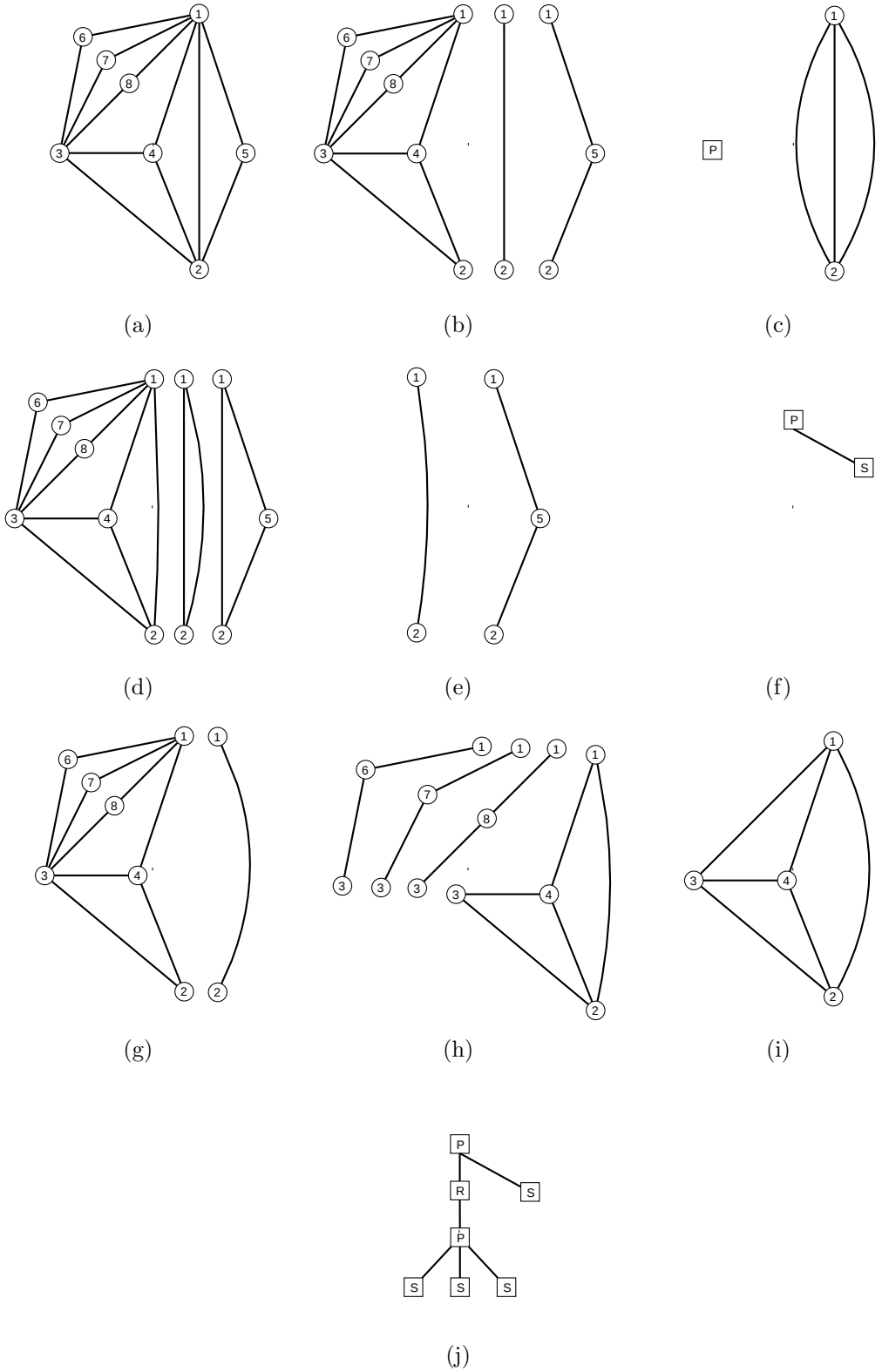


Figure 2.7: Graph and its SPQR-tree.

graph. This relies on the following theorem taken from [35].

Theorem 2.3 *Let G be a biconnected planar graph and let $\mathcal{T}$ be its SPQR-tree. A combinatorial embedding Π for G uniquely defines a combinatorial embedding of the skeleton of each node in $\mathcal{T}$. On the other hand, fixing the combinatorial embedding of the skeleton of each node in $\mathcal{T}$ uniquely defines a combinatorial embedding of G.*

Intuitively, each skeleton of an SPQR-tree $\mathcal{T}$ can be constructed from $\mathcal{T}$'s underlying graph G by replacing subgraphs of G with single edges. Hence, a combinatorial embedding of G defines a combinatorial embedding of each skeleton. From this also follows that each skeleton is a simplified view of G. Conversely, G can be constructed from T's skeletons by merging all skeletons together. This can be done by merging any two skeletons with the same reference edge until one skeleton is left. This skeleton is then *isomorphic* to G, i.e., there exists a one-to-one mapping between the vertices in the skeleton and G such that the adjacency in both graphs are the same. With this merging operation it is possible to construct a combinatorial embedding of G.

SPQR-trees are closely related to the classical decomposition of biconnected graphs into tricomps described by Hopcroft and Tarjan in [23]. Namely, the tricomps of a biconnected graph G are in one-to-one correspondence with the skeletons associated with the internal nodes of G's SPQR-tree. That is, R-skeletons correspond to triconnected graphs, S-skeletons to polygons (cycles) and P-skeletons to bonds [11].

Furthermore two S-nodes cannot be adjacent and the same is true for two P-nodes [11]. Since an S-node is a cycle, its skeleton has only one combinatorial embedding whereas the skeleton of an R-node has two combinatorial embeddings. In a skeleton of a P-node every permutation of its edges except the reference edge defines a combinatorial embedding. So, if an SPQR-tree has r R-nodes and l P-nodes whose skeletons own $p_1, \ldots, p_l$ edges, then the total number of combinatorial embeddings of its underlying graph is

$$2^r \prod_{1 \leq i \leq l} (p_i - 1)!$$

Therefore the number of combinatorial embeddings of a graph can be exponential.

If we think of an SPQR-tree as an acyclic connected graph instead of a rooted tree, then there is just one SPQR-tree for any biconnected graph. Hence, whatever edge is chosen as the reference edge for the decomposition, the result will always be the same [35].

According to its original definition an SPQR-tree of a biconnected graph $G = (V, E)$ has $\mathcal{O}(|V|)$ S-, P- and R-nodes and $\mathcal{O}(|E|)$ Q-nodes. It can be shown that

the number of edges and vertices in all skeletons grows linearly with the number of edges in the graph [35]. Therefore, the total size of an SPQR-tree is $\mathcal{O}(|V|+|E|)$. For planar graphs its size is even $\mathcal{O}(|V|)$ since a planar graph has at most $3 \cdot |V| - 6$ edges. An SPQR-tree can also be computed in time $\mathcal{O}(|V| + |E|)$. Again, for planar graphs the runtime is $\mathcal{O}(|V|)$. Gutwenger and Mutzel present in [19] a corrected linear time implementation based on the algorithm given in [23].

3 The Planarization Heuristic

We have already motivated in Chapter 1 that in automatic graph drawing it is essential to generate drawings with a minimum number of crossings. But unfortunately the problem to compute a drawing with the minimum number of crossings is $\mathcal{NP}$-hard in general. Basically, there are three heuristics that can be employed to tackle this problem. Thereto belong *force-directed* and *Sugiyama-like* methods [24] as well as the more promising and widespread *planarization heuristic*.

In this chapter we present the planarization method in more detail with an emphasis on its second major step, the edge reinsertion phase. Note, that the planarization method cannot be regarded as a self-contained graph drawing algorithm. It is rather a method which generates a *planar representation* of the non-planar graph to be drawn which then can be drawn by any planar graph drawing algorithm. Therewith the planarization approach can take advantage of many popular algorithms for drawing planar graphs that provide a great variety of styles of representations. The planarization technique goes back to ideas by Batini, Talamo and Tamassia [1, 31]. It is also part of the so-called *Topology-Shape-Metrics* model, a generic paradigm for computing orthogonal graph drawings [31].

3.1 The Two Major Steps

Basically, the planarization heuristic can be described by two major steps which are hard optimization problems on their own (see below). At first, for the sake of clarity we give a fine-grained description based on the one given by Ziegler in [36]:

1. Find a minimum set $F \subset E$ of edges in the non-planar graph $G = (V, E)$ to be drawn whose removal from G leads to a planar subgraph $G_p := G - F$ of G.

2. Determine a combinatorial embedding Π of G_p.

3. Reinsert all $e \in F$ into Π such that the number of crossings is minimized.

4. Replace the crossings by new artificial vertices.

5. Draw the resulting planar graph with a planar graph drawing algorithm.

6. Replace the artificial vertices by crossings.

3.1.1 First Step - Planarization

The first major step of the heuristic (step one in the above description) is known as the *Maximum Planar Subgraph Problem* (MPSP). We could equivalently formulate the first task of the planarization heuristic as finding a planar subgraph G_p of G with the maximum number of edges among all planar subgraphs of G. MPSP is an $\mathcal{NP}$-hard problem. However, there is an exact branch-and-cut algorithm by Jünger and Mutzel for the *weighted maximum planar subgraph problem*. The only difference to MPSP is that the input graph is edge-weighted and a planar subgraph with the maximum sum over all edge weights is desired. It is practically applicable as long as the number of deleted edges does not exceed ten [7]. So, with an edge weight of one for all edges an optimal solution of MPSP can be obtained by this algorithm.

In practice the computation of a maximal planar subgraph is more convenient. A maximal planar subgraph $G'_p = (V, E'_p)$ of a graph $G = (V, E)$ is a planar subgraph of G with the property that adding any edge in $E - E'$ would destroy the planarity of G'_p. The problem to find such a maximal planar subgraph is solvable in polynomial worst-case time. A simple approach is to start with a subgraph of G containing no edges and trying to insert edges of G one-by-one iteratively. After an edge is inserted, the current subgraph is tested for planarity. In the case it is non-planar, the edge is deleted from the subgraph and disregarded for the remaining steps. Since planarity can be tested in linear time in the number of vertices of a graph (e.g., [5]), the total running time is $\mathcal{O}(|V| \cdot |E|)$. Instead of starting this algorithm with no edges a spanning-tree of G can be utilized to save planarity testings. A better improvement is achieved with an on-line planarity testing algorithm using BC- and SPQR-trees [12]. It tests in amortized time $\mathcal{O}(log\,|V|)$ whether an edge can be inserted such that planarity is preserved plus necessary update operations. Altogether this leads to a total running time $\mathcal{O}(|E| \cdot log\,|V|)$.

3.1.2 Second Step - Edge Reinsertion

The second major step of the planarization heuristic (step three in the above description) is known as the *Constrained Crossing Minimization Problem* (CCMP) which has been thoroughly studied by Ziegler [36]. The formal definition of this problem claims that all $e \in F$ have to be inserted into the combinatorial embedding Π with the minimum number of crossings at once such that Π is preserved

and there are only crossings between edges in F and edges in $E - F$. Ziegler has proved CCMP to be $\mathcal{NP}$-hard and has developed a branch-and-cut algorithm that can solve CCMP optimally for small instances.

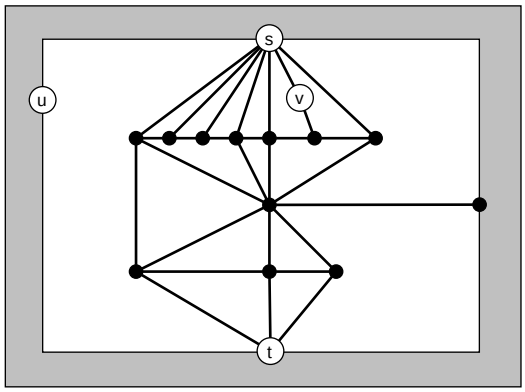
But even if both MPSP and CCMP are solved to optimality, the planarization method must not yield a crossing minimum drawing. The first reason is that we choose a maximum (maximal) planar subgraph whose planarity is preserved in the subsequent steps. But this subgraph can contain pairs of edges that have to cross in a crossing minimum drawing [36]. The second reason is that we fix one combinatorial embedding of this subgraph. But different combinatorial embeddings of the same subgraph allow solutions of CCMP with different qualities. The following example makes those facts clear. Consider the embedded planar subgraph G_p in Figure 3.1(a) taken from [22]. The gray shaded frame in Figure 3.1 stands for a sufficiently dense planar part of G_p. Suppose the first step of the planarization heuristic has removed only edge $e = \{u, v\}$ from the original graph to obtain G_p. Reinserting e into G_p with minimum crossings while preserving its embedding produces five crossings (cf. Figure 3.1(b)). But if the embedding of the dense planar part is not preserved and mirrored (can be thought of as a rotation along the vertical axis through s, t) whereby one crossing is accepted, then e can be inserted with only one crossing leading to two crossings at all (cf. Figure 3.1(c)).

3.2 Heuristics for CCMP

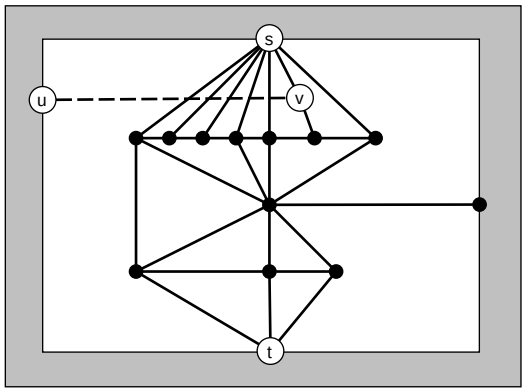
3.2.1 The Basic Heuristic

CCMP is solved heuristically as follows. According to [36], we call this procedure *basic heuristic*. The edges of F are reinserted one-by-one, instead of reinserting them at once, and the reinsertion of one edge $e = \{u, v\}$ into Π is transformed to a shortest path computation between u and v in the so-called *augmented dual graph*[1] $\Pi^*[\{u, v\}]$. $\Pi^*[\{u, v\}]$ arises from the dual graph $\Pi^* = (F^*, E^*)$ by adding the vertices u, v to F^* and by adding edges between u and all $f^* \in F^*$ and v and all $f^* \in F^*$ to E^* such that f^* represents the face in Π on whose boundary u and v, respectively, lie. Clearly, the insertion of e into Π with minimum crossings can be described by an ordered list of edges of G_p that e crosses. Since there is a dual edge in $\Pi^*[\{u, v\}]$ for each edge of G_p this can be done by a shortest path computation between u and v in $\Pi^*[\{u, v\}]$ where dual edges incident to u and v are traversed without costs. A formal proof of this equivalence can be found in [36].

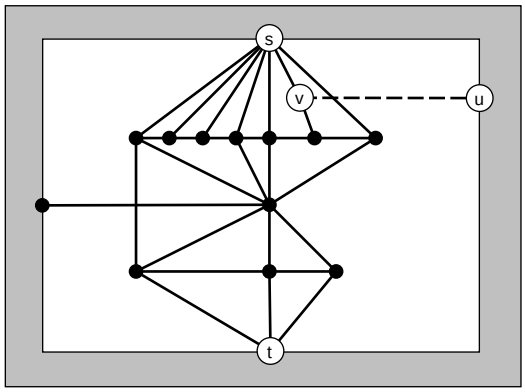
[1] Also known under the term *extended dual graph*.



(a)

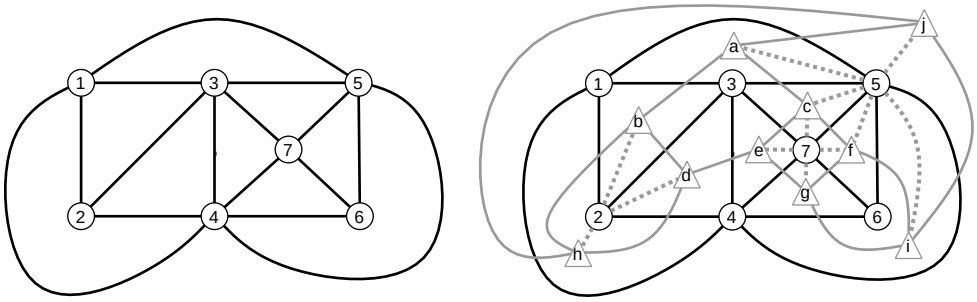


(b)



(c)

Figure 3.1: Embedding Π of a graph (a) into which the edge $\{u, v\}$ has to be reinserted. (b) and (c) show each a resulting embedding with 5 and 2 crossings while the planarity of Π is one time preserved and the other time disregarded.



(a) (b)

Figure 3.2: Example of a graph (a) and its augmented dual graph (b) with additional vertices $2, 5$ and 7.

Below a formal definition of the augmented dual graph, a subsequent example, and the algorithm presenting the basic heuristic for CCMP are given.

Definition 3.1 (Augmented dual graph) *Let $G = (V, E)$ be a planar graph, $V' \subseteq V$, Π be an embedding of G and $\Pi^* = (F^*, E^*)$ be the dual graph of G with respect to Π. Further let $F_v = \{\{v, f^*\} \mid f^* \in F^*, v \text{ lies on the boundary of } f\}$ be for every $v \in V'$. Then $\Pi^*[V'] = (F^* \cup V', E^* \cup (\bigcup_{v \in V'} F_v))$ is the augmented dual graph of G with respect to Π and V'.*

Figure 3.2(a) shows a graph G with an embedding Π and Figure 3.2(b) shows the augmented dual graph of G with respect to Π and vertex set $\{2, 5, 7\}$. We assume that the edge $e = \{2, 7\}$ has to be inserted. We observe that $P = 2, d, e, 7$ is the only shortest path between 2 and 7 in the augmented dual graph that does not use 5 as internal vertex. The length of P is one. The edge $e' = \{d, e\}$ used by P indicates that e can be inserted into Π by crossing the dual edge $\{3, 4\}$ of e'. For example, the path $P' = 2, h, j, 5, f, 7$ is not a proper shortest path that can be transformed to a list of edges that e crosses. So generally, a shortest path between two additional vertices in an augmented dual graph must not contain any additional vertex as internal vertex.

Algorithm 3.1: Basic heuristic for CCMP.

Input: Embedding Π of a planar subgraph G_p, edge set F with endpoints in G_p
Output: Embedding Π' of a planar graph G'_p

$\Pi' := \Pi$
$G'_p := G_p$

while $F \neq \emptyset$ **do**
  Choose edge $e = \{u, v\} \in F$
  $F := F - \{e\}$
  Compute the augmented dual graph $\Pi^*[\{u, v\}]$
  Insert e into Π by computing a shortest path in $\Pi^*[\{u, v\}]$ between u and v
  Replace crossings by artificial vertices to obtain a new planar graph G'_p and an embedding Π' of G'_p
end while
Replace all artificial vertices by crossings
return Π'

3.2.2 Refinements of the Basic Heuristic

The quality of the basic heuristic for CCMP depends on the order in which edges in F are reinserted. But even if all different insertion orders and furthermore all shortest paths between the endpoints of an edge are taken into account, it is not guaranteed to solve CCMP to optimality. Ziegler presents in [36] an example for which this fact is true.

The basic heuristic can be improved by the following variations:

Permutation Heuristic. Algorithm 3.1 is modified such that it expects an ordered list of the edges in F and reinserts the edges according to this list. We generate n lists each saving a randomly computed permutation of the edges in F and call the modified Algorithm 3.1 for each list. Clearly, the result of this heuristic is the solution with the smallest number of crossings.

Shortest First Heuristic. This heuristic inserts that edge among the remaining edges to be inserted with the shortest shortest path. This approach is based on the assumption that when inserting an edge with a large number of crossings also the number of crossings for remaining edges to be inserted increases.

Remove and Reinsert Heuristic. This heuristic is a post-processing strategy and it addresses the following drawback. After an edge is reinserted it is not regarded anymore. However, it is possible that this edge has been reinserted early in the reinsertion phase and might be involved in many crossings at the end of the reinsertion phase. Hence, it can be profitable to delete and directly reinsert this edge again. Thereby the edge can be reinserted with the same number of crossings as before and thus it is guaranteed that the quality of the overall solution is only improved. This is done for all edges of

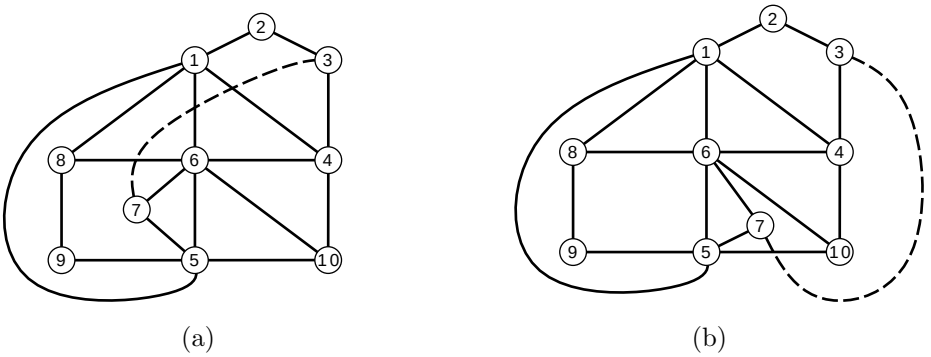


Figure 3.3: The minimum number of crossings needed when inserting an edge depends on the chosen embedding.

F. Having performed one run, the situation might have changed again for all edges of F. So, it is reasonable to iterate the procedure until no more improvement is achieved.

Another improvement in one single edge reinsertion step is to take all combinatorial embeddings of the current subgraph into account. More formally, this problem can be defined as follows. Let $G = (V, E)$ be a planar (sub)graph and $e = \{u, v\} \notin E$ be an edge with endpoints $u, v \in V$. Find a combinatorial embedding Π of G among all embeddings of G such that e can be inserted into Π with the minimum number of crossings. Figure 3.3 shows a simple example in which the choice of the combinatorial embedding has an impact on the number of crossings produced when inserting the edge $\{3, 7\}$. When choosing the embedding of Figure 3.3(a) the dashed edge cannot be inserted with less than two crossings whereas the embedding of Figure 3.3(b) allows to add it with one crossing. Gutwenger, Mutzel and Weiskircher present in [19] a linear time algorithm based on SPQR-trees for the solution of this problem.

In [9] Gutwenger and Mutzel present an extensive experimental study of crossing minimization heuristics. They have studied the effects of various methods for computing the maximal planar subgraph and for edge reinsertion with different post-processing strategies based on the remove and reinsert heuristic. Gutwenger and Mutzel conclude that the remove and reinsert heuristic with deleting and reinserting also edges of the planar subgraph "helps a lot". But also starting with a "good" planar subgraph and performing edge reinsertion with a variable embedding are worthwhile.

4 VIP - The Vertex Insertion Problem

This chapter is the main part of this thesis. First of all in Section 4.1 we define and motivate what the *Vertex Insertion Problem*[1] is about. Thereto belong two variants of this problem which will be defined formally and referred to as *VIP-FIX* and *VIP-VAR*. The main focus lies on the latter problem. Thereafter in Section 4.2 two polynomial time algorithms solving VIP-FIX are presented. Section 4.3 is dedicated to VIP-VAR. We present two algorithms that can compute a lower bound of the costs of VIP-VAR. Moreover in Section 4.3.2 we develop an algorithm for a certain subproblem arising in our polynomial time algorithm with respect to a biconnected graph. This algorithm is given in Section 4.3.3. At last in Section 4.3.4 a polynomial time algorithm with respect to a connected graph is developed.

4.1 Problem Definition and Motivation

Chapter 3 presented the planarization heuristic which is one of the best methods in practice to tackle the crossing minimization problem. In its original form, the heuristic uses edge deletion and reinsertion "only". However, the problems relating to these two major steps, the Maximum Planar Subgraph and Constraint Crossing Minimization Problem, are both proved to be $\mathcal{NP}$-hard. But the planarization method is also conceivable in another way and its two steps can be substituted as follows. Firstly, a preferably small set of vertices including their incident edges is deleted to obtain a planar subgraph of the given non-planar graph. We cannot hope to compute such a vertex set of minimum size efficiently. This is due to the fact that the problem of deciding for an integer k whether a non-planar graph can be made planar by deleting at most k vertices is $\mathcal{NP}$-complete [25]. But this step can be done heuristically. For example, a greedy algorithm deletes vertices of minimum (or maximum) degree until the obtained subgraph is planar. Conversely, it can start with an empty graph and extends the graph with vertices of maximum (or minimum) degree until no further vertex can be added. A more sophisticated algorithm is proposed by Edwards and

[1] We use Vertex Insertion Problem as a synonym for the problem of Inserting a Vertex into a Planar Graph.

Farr in [14]. Their algorithm provides an induced planar subgraph of at least $3 \cdot |V|/(d_{max}+1)$ vertices in a graph of maximum degree d_{max} and in a runtime of $\mathcal{O}(|V| \cdot |E|)$. Afterwards the deleted vertices are reinserted one-by-one each with their incident edges at once. Thereby the inserted edges should produce as few crossings as possible with edges in the current (sub)graph. Again crossings are replaced by artificial vertices to obtain a planar graph for the next vertex reinsertion step. This *Vertex Insertion Problem* can be formulated in two variants. The first one is to insert a vertex and its incident edges into the current graph with respect to a fixed combinatorial embedding of it. Formally, we define this problem variant as follows.

Problem 4.1 (VIP-FIX) *Let $G = (V, E)$ be a connected planar graph, $W \subseteq V$ be a non-empty vertex set and Π be a fixed combinatorial embedding of G. The Vertex Insertion Problem Fix is the problem to insert a vertex $\vartheta \notin V$ and all edges in $\{\{\vartheta, w\} \mid w \in W\}$ into Π with the minimum number of crossings.*

The second variant of the Vertex Insertion Problem takes all combinatorial embeddings of the current graph into account. Formally, we define this problem variant as follows.

Problem 4.2 (VIP-VAR) *Let $G = (V, E)$ be a connected planar graph and $W \subseteq V$ be a non-empty vertex set. The Vertex Insertion Problem Variable is the problem to find a combinatorial embedding Π among all combinatorial embeddings of G such that a vertex $\vartheta \notin V$ and all edges in $\{\{\vartheta, w\} \mid w \in W\}$ are inserted into Π with the minimum number of crossings.*

In both problems we implicitly demand that there are only edge crossings between edges in E and edges to be inserted. Consequently, we denote the vertex to be inserted by ϑ and the set of vertices becoming adjacent to ϑ by W[2]. We call a vertex $w \in W$ an *affected vertex* and the graph into which ϑ has to be inserted is also called *input graph*. Furthermore we easily recognize that the subproblem in VIP-VAR to insert ϑ and its incident edges into the determined embedding crossing minimally is exactly VIP-FIX. Therefore in our following considerations we rather associate with VIP-VAR the problem to find an appropriate combinatorial embedding.

Since the input graph G of VIP-VAR is finite, and thus the set of all combinatorial embeddings of G is finite, VIP-VAR is a combinatorial optimization problem. The basic set is given by the set of combinatorial embeddings of G. Feasible solutions for VIP-VAR are combinatorial embeddings that permit planar drawings of G. The *costs of a solution* are the number of crossings produced by inserted

[2]Additionally, we can claim $|W| > 2$ because otherwise the problem can be solved trivially ($|W| = 1$) or is equivalent to the problem in [19] ($|W| = 2$).

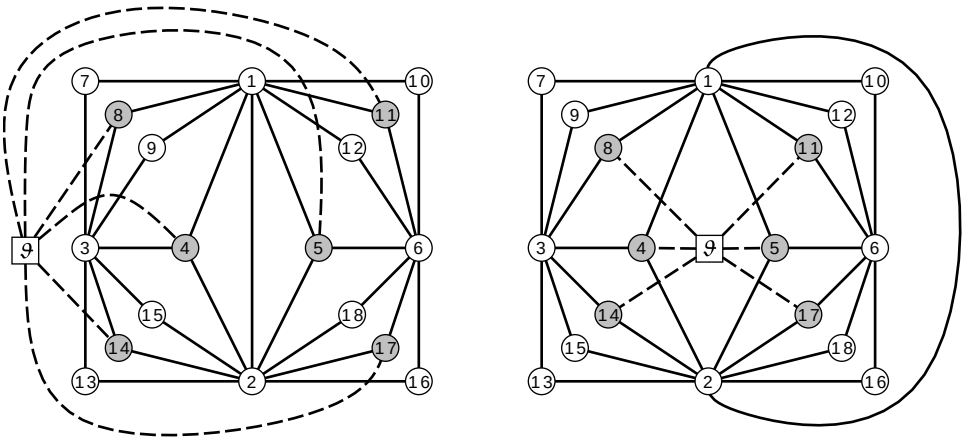


Figure 4.1: Biconnected graph with affected vertices $4, 5, 8, 11, 14, 17$. The embedding (a) allows to insert ϑ and its incident edges with 10 crossings whereas these elements can be inserted into the embedding (b) with only 4 crossings.

edges. The aim of the problem is to minimize the costs and an *optimal solution* of VIP-VAR is a solution with minimum costs.

Clearly, since the number of crossings produced by inserted edges highly depends on the chosen embedding of the current graph, there is no doubt about a better quality of a solution of VIP-VAR compared to a solution of VIP-FIX (cf. Figure 4.1). One open question is left. What is the advantage of the modified planarization method in contrast to the usual one? The benefit is that in each vertex reinsertion step a certain amount of edges, namely the incident edges of the current considered vertex, is reinserted at once. Thus, no longer each edge is reinserted separately and this may lead to an embedding of the original graph with less crossings overall.

4.2 VIP with Fixed Embedding

As we know from Section 3.2, we can describe the insertion of one edge $e = \{u, v\}$ into a combinatorial embedding Π by giving an ordered list of edges that e crosses. We call such a list *edge insertion path*. Its formal definition is the following [35].

Definition 4.1 *Let $G = (V, E)$ be a connected planar graph and let Π be an embedding of G. The list $P = e_1, \ldots, e_k$ with $e_i \in E$ $(i = 1, \ldots, k)$ is an edge insertion path between two vertices $v_1, v_2 \in V$ of 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 satisfied:

1. *There is a face in Π with e_1 and v_1 on its boundary*
2. *There is a face in Π with e_k and v_2 on its boundary*
3. *$e_1^*, \ldots, e_2^*$ is a path in the dual graph Π^**

We have seen that this list can be determined by computing a shortest path $P^* = u, f_1^*, \ldots, f_k^*, v$ between u and v in the augmented dual graph $\Pi^*[\{u, v\}]$ and by replacing the dual edges $\{f_1^*, f_2^*\}, \ldots, \{f_{k-1}^*, f_k^*\}$ with their primal edges.

Basically, to solve VIP-FIX with inputs $G = (V, E), \Pi$ and W we have to compute several shortest edge insertion paths between ϑ and all $w \in W$ with the additional difficulty that ϑ has to be inserted itself. Since ϑ can only be inserted into a face f of Π and all the edges to be inserted will originate only in ϑ and thus need not cross among themselves, the basic solution of VIP-FIX is to compute shortest dual edge insertion paths between f^* and all $w \in W$ (we denote one such shortest path and its length by $P(f^*, w)$ and $|P(f^*, w)|$, respectively) for all dual vertices $f^* \in F^*$ in the augmented dual graph $\Pi^*[W] = (F^* \cup W, \tilde{E}^*)$ of G with respect to Π and W. Algorithm 4.1 acts like this. Note, that a shortest path $P(w, f^*)$ must not contain any $w' \in W$ as internal vertex. Let $\tilde{V} := F^* \cup W$. If, e.g., Dijkstra's algorithm is employed to compute a shortest path together with efficient data structures, this leads to a total running time $\mathcal{O}(|\tilde{V}| \cdot log\,|\tilde{V}| \cdot |F^*|)$. We can generalize this running time to $\mathcal{O}(|V|^2 \cdot log\,|V|)$ because $|F^*| \leq |\tilde{V}| = \mathcal{O}(|V|)$ holds and also the computation of $\Pi^*[W]$ needs time $\mathcal{O}(|V|)$. If we use a breadth-first search (BFS) instead of Dijkstra's algorithm we achieve an improvement of factor $log\,|V|$ such that a solution is computable in time $\mathcal{O}(|V|^2)$.

Algorithm 4.1: Computes shortest dual edge insertion paths naively.

Input: Embedding Π of a connected planar graph $G = (V, E)$, $W \subseteq V$
Output: List L of shortest dual edge insertion paths

```
1: c := ∞
2: L := ()
3: Compute Π*[W] = (F* ∪ W, Ẽ*)
4: for all f* ∈ F* do
5:     c' := 0
6:     L' := ()
7:     for all w ∈ W do
8:         Compute P(f*, w) in Π*[W] and append it to L'
9:         c' := c' + |P(f*, w)|
```

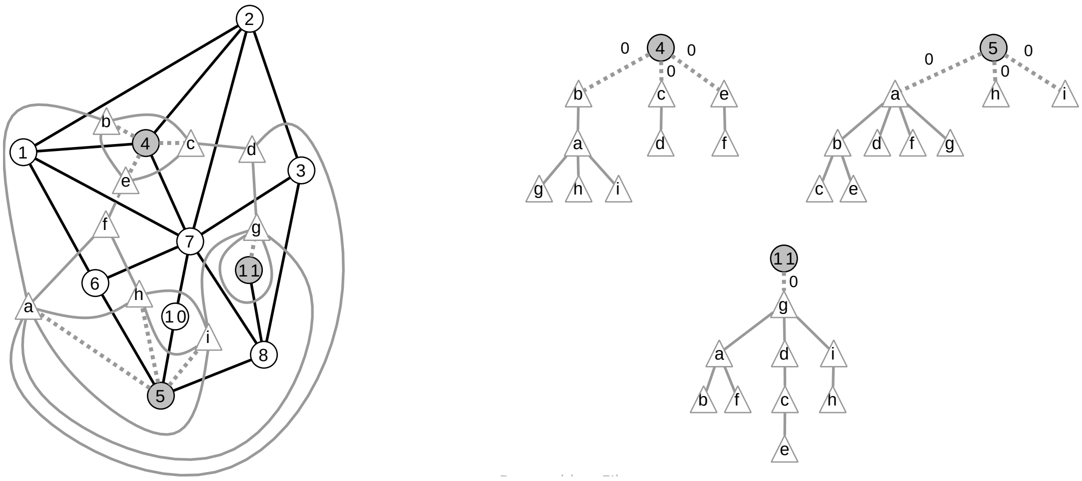
```
10:     end for
11:     if c' < c then
12:         L := L'
13:         c := c'
14:     end if
15: end for
16: return L
```

A more efficient approach is Algorithm 4.2 (cf. also Figure 4.2). The algorithm processes a BFS in $\Pi^*[W]$ starting from each $w \in W$ and utilizing two associative data structures $\mathcal{C}$ and $\mathcal{L}$. $\mathcal{C}(f^*)$ is the sum of the length of shortest paths between $f^* \in F^*$ and the current processed $w \in W$. $\mathcal{L}(f^*)$ is a list of all shortest dual edge insertion paths between f^* and the current processed $w \in W$. Whenever a vertex $f^* \in F^*$ is reached during a BFS, $\mathcal{C}(f^*)$ and $\mathcal{L}(f^*)$ are updated. Because $\Pi^*[W]$ is planar one BFS requires time $\mathcal{O}(|\tilde{V}|)$ and therefore the total running time is $\mathcal{O}(|\tilde{V}| \cdot |W|) = \mathcal{O}(|V| \cdot |W|)$. Since $|W| \leq |V|$ the same worst-case runtime is guaranteed as with Algorithm 4.1. But if $|W|$ is sufficiently small this algorithm can even have linear runtime.

Algorithm 4.2: Computes shortest dual edge insertion paths by processing several breadth-first searches.

Input: Embedding Π of a connected planar graph $G = (V, E)$, $W \subseteq V$
Output: List L of shortest dual edge insertion paths

```
 1: c := ∞
 2: L := ()
 3: C := L := ∅                                  ▷ associative data structures
 4: Compute Π*[W] = (F* ∪ W, Ẽ*)
 5: for all f* ∈ F* do
 6:     C(f*) := 0
 7:     L(f*) := ()
 8: end for
 9: for all w ∈ W do
10:     Process a BFS in Π*[W] starting from w and let P(w, f*):
          ↪ C(f*) := C(f*) + |P(w, f*)| and append P(w, f*) to L(f*)
11: end for
12: f*_min := nil
13: for all f* ∈ F* do
14:     if C(f*) < c then
15:         c := C(f*)
```



(a) (b)

	a	b	c	d	e	f	g	h	i
4	1	0	0	1	0	1	2	2	2
5	0	1	2	1	2	1	1	0	0
11	1	2	2	1	3	2	0	2	1
$\sum$	2	3	4	3	5	4	3	4	3

(c)

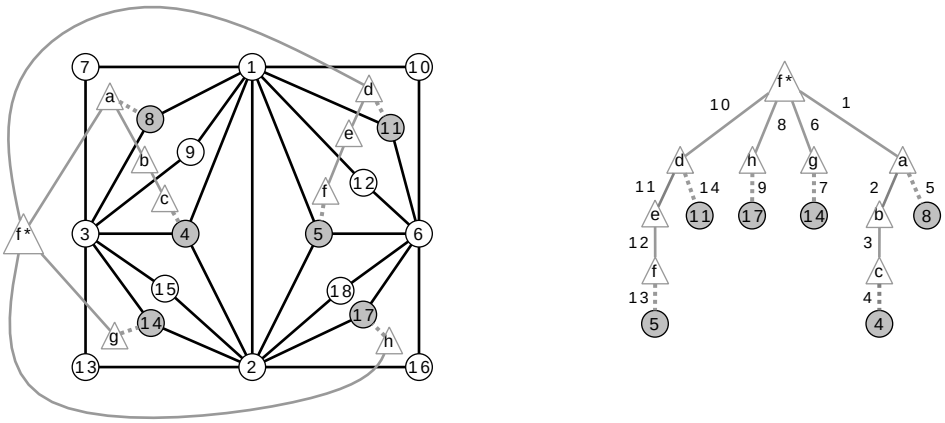
Figure 4.2: Example illustrating the approach of Algorithm 4.2. (a) shows a graph with its augmented dual graph with respect to affected vertices $4, 5$ and 11, (b) possible BFS trees outgoing from these vertices and (c) a distance matrix for each affected vertex. As can be seen the new vertex would be inserted into face a.

16: $f^*_{min} := f^*$
17: **end if**
18: **end for**
19: **return** $L := \mathcal{L}(f^*_{min})$

Clearly, only by computing the shortest dual edge insertion paths, we have not solved VIP-FIX entirely. It remains to construct the shortest edge insertion paths, to replace all crossings described by these by artificial vertices and to insert ϑ itself. With Π as fixed embedding of G and $G_0 := G$, for the i-th edge insertion path $P_i = e_1, \ldots, e_k$ between the vertices ϑ and $w_i \in W$ we build a graph G_i from G_{i-1} by splitting each edge e_j in P with a new vertex x_j and insert new edges forming a path $\vartheta, x_1, \ldots, x_k, w_i$. If the edge e_j is subdivided into two new edges e_{j_1}, e_{j_2}, it must be ensured that either e_{j_1} or e_{j_2} is associated with e_j. Otherwise unprocessed edge insertion paths may contain invalid edges. Note that it is necessary to process the edge insertion paths in a specific order due to the fact they can share common edges (cf. Figure 4.1(a)). This order also induces the cyclic clockwise order of the edges around ϑ. Thereto we build a tree T induced by all the shortest dual edge insertion paths. The root of T becomes f^*. Let $g^*_1, \ldots, g^*_k$ be the children of f^* in cyclic clockwise order. We traverse T by processing a depth-first search on each subtree T_i of T with root g^*_i for $i = 1, \ldots, k$. The depth-first search always visits the left most unvisited subtree of the current inner node. Whenever a leaf $w_j \in W$ is reached, we build the corresponding primal edge insertion path and process it as just now described. Figure 4.3(a) shows the same biconnected graph as in Figure 4.1(a) with the tree that is induced by all shortest dual edge insertion paths. Figure 4.3(b) illustrates the tree again, this time with labeled edges indicating the traversing order of the tree. Hence the edge insertion paths between ϑ and 4, ϑ and 8,..., and ϑ and 11 are processed in this order. We can build T, e.g., by deleting every edge in the augmented dual graph $\Pi^*[W]$ that does not belong to a shortest dual edge insertion path. This takes time $\mathcal{O}(|V|)$. At most each edge of T is considered as many times as leafs in T exist. This is caused by constructing the primal edge insertion paths. There are $|W|$ leafs in T and $\mathcal{O}(|V|)$ edges. Hence processing all edge insertion paths takes time $\mathcal{O}(|V| \cdot |W|)$ and therefore VIP-FIX is solvable in an overall running time of $\mathcal{O}(|V| \cdot |W|)$.

4.3 VIP with Variable Embedding

Obviously, the concept of enumeration is a simple solution for VIP-VAR like for any other combinatorial optimization problem, since the set of feasible solutions



(a) (b)

Figure 4.3: Biconnected graph (a) with the tree (b) induced by all shortest dual edge insertion paths and a labelling of the tree edges indicating the tree's traversing order.

is finite. In the case that the input graph is biconnected we could enumerate all its embeddings with the help of its SPQR-tree (cf. Theorem 2.3). During this enumeration we save the current best embedding. But this is not a promising approach, since the number of embeddings of a graph is usually exponential. Moreover if the input graph is connected even the enumeration is a non-trivial problem. So both these facts make an efficient solution of VIP-VAR non-trivial.

4.3.1 Upper and Lower Bound

Let $G = (V, E)$ be the input graph of VIP-VAR. A trivial upper bound for the costs of an optimal solution of VIP-VAR is $|W| \cdot |E|$, because one inserted edge does not need to cross more than $|E|$ edges. We can improve this upper bound if we choose an arbitrary embedding of G and solve VIP-FIX with respect to that embedding.

Surprisingly, to compute a lower bound for the costs of an optimal solution of VIP-VAR we can employ the algorithm in [19] for finding a combinatorial embedding of a planar graph where one edge can be inserted with the minimum number of crossings (see also Section 3.2.2). For the moment let us refer to this problem as EIP-VAR. Let b denote the lower bound; initially $b = 0$ holds. A (greedy) algorithm computes in a first phase a $|W| \times |W|$ matrix $\Omega = (\omega_{ij})$, where ω_{ij} denotes the number of crossings in an optimal solution of EIP-VAR with input graph G into which edge $\{w_i, w_j\}$, $w_i, w_j \in W$ has to be inserted.

The entries ω_{ij} and ω_{ji} are equal and it suffices to compute the entries above the diagonal of Ω. After this, all vertices are regarded as *unmarked*. The second phase of the algorithm builds greedily pairs of unmarked vertices w_i, w_j with the maximum value ω_{ij} among all unmarked vertices, adds ω_{ij} on b and marks w_i, w_j. This is repeated until either all vertices are marked ($|W|$ is even) or one unmarked vertex remains ($|W|$ is odd). Constructing the matrix takes time $\mathcal{O}((|V| + |E|) \cdot |W|^2) = \mathcal{O}(|V| \cdot |W|^2)$ because one call of the algorithm solving EIP-VAR takes time $\mathcal{O}(|V| + |E|)$ and we have to compute $\frac{1}{2} \cdot (|W| - 1) \cdot |W|$ matrix entries. The second phase has a running time of $\mathcal{O}(|W^3|)$ since the matrix is traversed at most $|W|$ times. Since $|W| \leq |V|$ a total runtime of $\mathcal{O}(|V| \cdot |W|^2)$ is obtained.

We can argue the algorithm's correctness as follows. If ϑ is inserted into an optimal solution Π for VIP-VAR, there is a path w_i, ϑ, w_j between any two affected vertices w_i, w_j. Such a path causes at least as many crossings as the optimal edge insertion path between w_i, w_j in an embedding Π'. Due to this fact, the algorithm searches for unmarked vertices with the maximum value in Ω in each step of the second phase. The reason why the algorithm must take unmarked vertices can be explained as follows. Consider two edges $e_1 = \{w_i, w_j\}, e_2 = \{w_i, w_k\}$, $j \neq k$. Assume the algorithm for EIP-VAR with inputs G and e_1 in one and inputs G and e_2 in another call returns in both cases an identical embedding Π of G. Then it is possible that edge insertion paths $P_1 = w_i, e_1, \ldots, e_h, w_i$ and $P_2 = w_i, e'_1, \ldots, e'_j, w_k$ are not distinct, i.e., an index l exists such that $e_1 = e'_1, \ldots, e_l = e'_l$ is true. Clearly, ϑ is connected once to w_i in any solution of VIP-VAR. But adding values ω_{ij} and ω_{ik} may cause in counting l crossings twice, if, e.g., Π is also an optimal solution for VIP-VAR.

The second phase of this algorithm can be improved with the help of matchings. A set M of non-adjacent edges in a graph $G = (V, E)$ is called a *matching*. A matching M is called a *maximum (cardinality) matching* if there is no other matching M' with $|M'| > |M|$. If the edges $e \in E$ are weighted, we can ask for a *maximum weight (cardinality) matching*. That is a maximum matching M such that the sum of weights for $e \in M$ is maximum among all maximum matchings. Therefore, we can replace the greedy pair search in the second phase of our algorithm and compute a stronger bound by finding a maximum weight matching of the complete graph $K_{|W|}$ on all affected vertices. Edges between vertices w_i, w_j in $K_{|W|}$ are weighted with ω_{ij}. The algorithm of Gabow [16] computes a maximum weight matching for $K_{|W|}$ in time $\mathcal{O}(|W|^3)$. Thus, the total running time of our algorithm is maintained.

Usually, we cannot expect the computed lower bound to be sharp, even with the help of maximum weight matchings. For example, this is the case, if $|W|$ is odd and the matching algorithm remains one exposed vertex w_i but the edge

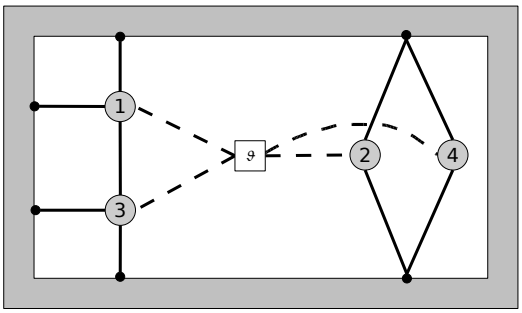


Figure 4.4: A counterexample that shows a graph for which the computed lower bound is not sharp.

$\{\vartheta, w_i\}$ produces crossings in an optimal solution of VIP-VAR. Figure 4.4 shows an example graph G for which the lower bound is not sharp even in the case where $|W|$ is even. The vertices $1, 2, 3, 4$ are affected and the shaded region represents a dense planar part of G. If this dense planar part is crossed by an edge, let at least two crossings occur. Assume further the graph resulting from G by deleting the vertices $2, 4$ is triconnected. Then the SPQR-tree of G consists of one R-node, one P-node and two S-nodes. Optimal solutions of EIP-VAR with input edge $\{i, j\}$ for all $i, j \in \{1, 2, 3, 4\}, i \neq j$ produce no crossings. But an optimal solution of VIP-VAR produces one crossing.

4.3.2 The Core Problem

In this section we deal with a certain optimization problem which arises as a subproblem in our holistic algorithm solving VIP-VAR. This problem forms the main difficulty and therefore can be called *core problem*. We first give some more definitions and notations whereupon the core problem is defined and an algorithm for it is stepwise developed.

Definition of the Core Problem

$W(G')$ and $W^-(G')$ denote the sets of affected vertices of the pertinent, skeleton or expansion graph G' including and excluding its possibly affected poles, respectively. From now on we implicitly interpret each virtual edge $e = \{u, v\}$ as a directed edge from u (v) to v (u). The same imaginary direction is interpreted for the reference edge of $skeleton(\mu)$ and $pert(\mu)$ with μ as pertinent node of e. We denote it by $\varepsilon_\mu := \{u, v\}$. $f_r(e)$ is the face to the right and $f_l(e)$ is the face to the left of e according to its imaginary direction. $f_r(\varepsilon_\mu)$ is the face which is separated by ε_μ and lies to the right of another edge $\{u, v'\}$ ($\{v, u'\}$) according

to its imaginary direction from u (v') to v' (u) if e has the imaginary direction from u (v) to v (u). $f_l(\varepsilon_\mu)$ is the counter face separated by ε_μ. Due to this definition a clear association between $f_r(e)$ and $f_r(\varepsilon_\mu)$, and between $f_l(e)$ and $f_l(\varepsilon_\mu)$ is established. As supplied before $P(x, y)$ is a (weighted) shortest path between the elements x and y in a graph and $|P(x, y)|$ its length (if $x = \emptyset$ or $y = \emptyset$ holds, $P(x, y)$ is an empty path with $|P(x, y)| := 0$).

Another fundamental concept used in our algorithms is that of the *traversing costs* of a skeleton edge. Thereto the following lemma is necessary.

Lemma 4.1 *Let μ be an inner node of a rooted SPQR-tree. The length of a shortest path $P(f_r^*(\varepsilon_\mu), f_l^*(\varepsilon_\mu))$ in the dual graph Λ^* that does not use ε_μ^* is independent of the embedding Λ of $pert(\mu)$.*

Definition 4.2 (Traversing Costs) *Let e be a skeleton edge. If e is a proper edge, then the traversing costs $c(e)$ of e are equal to 1. Else let μ be the pertinent node of e. The traversing costs $c(e)$ of e are then defined as the length of a shortest path $P(f_r^*(\varepsilon_\mu), f_l^*(\varepsilon_\mu))$ in the dual graph Λ^* that does not use ε_μ^* with Λ as any embedding of $pert(\mu)$.*

The above lemma and definition are based on [19]. Traversing costs of skeleton edges allow statements about the exact number of crossings produced when inserting an edge into an embedding of a skeleton. Let $\mathcal{T}$ be an SPQR-tree. The traversing costs of all skeleton edges for all skeletons associated with nodes of $\mathcal{T}$ are computed by a bottom-up traversal of $\mathcal{T}$. Let the current inner node in this traversal be μ with virtual edge e. Only if μ is an R-node a weighted shortest path computation is necessary to obtain $c(e)$. Else, if $skeleton(\mu)$ contains the edges $e_1, \ldots, e_k$ excluding the reference edge and μ is

- an S-node, then $c(e) := \min(\{c(e_1), \ldots, c(e_k)\})$.
- a P-node, then $c(e) := c(e_1) + \ldots + c(e_k)$.

The worst-case computation time of this algorithm eventuates if $\mathcal{T}$ consists only of R-nodes. The dual graph of an embedding of $skeleton(\mu)$ can be built in time $\mathcal{O}(|skeleton(\mu)|)$. The weighted shortest path computation can be done in time $\mathcal{O}(|skeleton(\mu)|)$ with the algorithm suggested in [33]. This leads to a total running time of $\mathcal{O}(|V|)$ since the overall size of all skeletons is $\mathcal{O}(|V|)$. The formal definition of the core problem is the following.

Problem 4.3 (Core Problem) *Let $\mathcal{T}$ be a rooted SPQR-tree, μ be an inner node of $\mathcal{T}$ and $\mathcal{W} := W^-(pert(\mu)) \neq \emptyset$. For $j = |\mathcal{W}|, \ldots, 0$ find an embedding $\Pi(j)$ among all embeddings of $pert(\mu)$ such that for an arbitrary partition*

$\mathcal{W}_1 \dot{\cup} \mathcal{W}_2 = \mathcal{W}$ *with* $|\mathcal{W}_1| = j$

$$\bar{C}(j) := \sum_{w \in \mathcal{W}_1} |P(w, f_r^*(\varepsilon_\mu))| + \sum_{w' \in \mathcal{W}_2} |P(w', f_l^*(\varepsilon_\mu))|$$

is minimum with $P(w, f_r^*(\varepsilon_\mu))$ *and* $P(w', f_l^*(\varepsilon_\mu))$ *as shortest paths in the augmented dual graph* $\Pi^*(j)[\mathcal{W}]$ *of* $pert(\mu)$ *with respect to* $\Pi(j)$ *and* $\mathcal{W}$ *that do not use* ε_μ^*.

A more illustrative characterization of this problem (and its solution) is to find a best embedding $\Pi(j)$ of $pert(\mu)$ for $j = 0, \ldots, |\mathcal{W}|$ such that vertices v_r and v_l can be inserted into $f_r(\varepsilon_\mu)$ and $f_l(\varepsilon_\mu)$, respectively, and edges between $v \in \{v_l, v_r\}$ and all $w \in \mathcal{W}$ can be inserted into $\Pi(j)$ with the minimum number of crossings under the premise that exactly j and $|\mathcal{W}| - j$ of these have endpoints v_r and v_l, respectively. Figure 4.5 shows an example. Figure 4.5(a) and Figure 4.5(b) show a biconnected graph with affected vertices 7, 8, 10 and 11 and its SPQR-tree with root μ_1 and drawn in skeletons, respectively. Figure 4.5(c) and 4.5(d) show the optimal subsolutions of Problem 4.3 with respect to μ_2 and $j = 2, 3$ with values $\bar{C}(2) = 3$ and $\bar{C}(3) = 4$, respectively. With respect to the imaginary direction from 5 to 6 of the virtual edge $\{5, 6\}$ the face $f_r(\varepsilon_{\mu_2})$ is defined by the edges $\{5, 6\}, \{6, 7\}$ and $\{7, 5\}$.

Solving the Core Problem

Our general approach for solving the core problem is to compute the value of a solution of it, i.e., the *cost vector*[3] $\bar{C}$, with dynamic programming and to save extra information to be able to construct the solution, i.e., the embeddings of $pert(\mu)$, itself. As will be clear in the next section where we present an algorithm for solving VIP-VAR for biconnected graphs, basically we are only interested in the cost vector and at most in one embedding of $pert(\mu)$. For the computation of $\bar{C}$ the cost vector for each child of μ has to be computed before. Then with further dynamic programming $\bar{C}$ is obtained. This induces a bottom-up approach. For the following consideration we presume that the traversing costs of all skeleton edges are given.

S-/R-node. Let μ be an S- or R-node and a j with $0 \leq j \leq |\mathcal{W}|$ be given. We show how to obtain a subsolution of the core problem with respect to μ and j and that our computation is correct. To make the argumentation easier we use the illustrative characterization of a solution of the core problem as shortly described. We first deal with the non-simple case in which μ is an inner node.

[3] In the whole chapter we do not use the standard mathematical notation for vectors and matrices since further vectors and matrices will be used with sub- and superscripts.

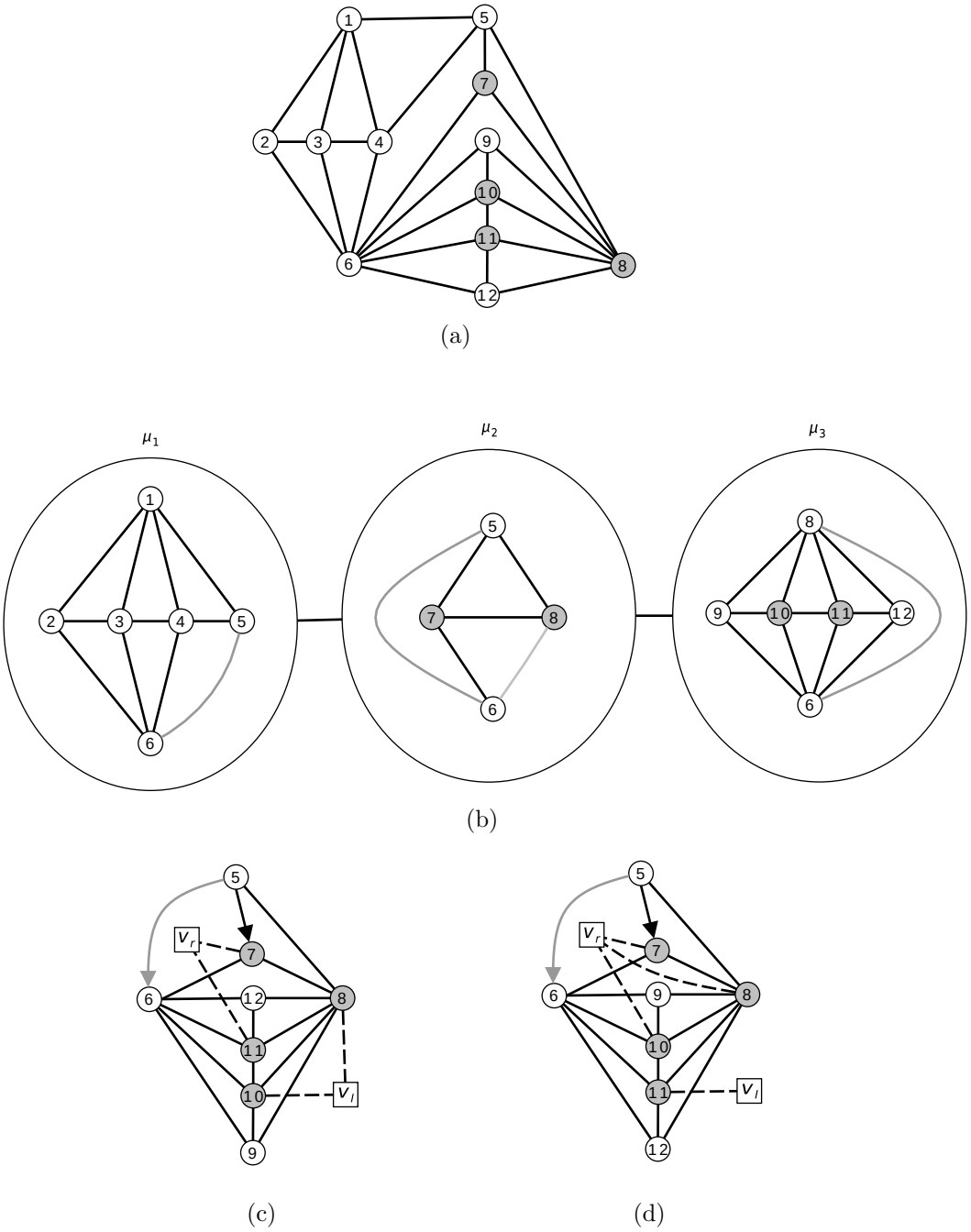


Figure 4.5: Biconnected graph (a), its SPQR-tree with root (b) and two embeddings of $pert(\mu_2)$ (c),(d) representing optimal subsolutions of Problem 4.3.

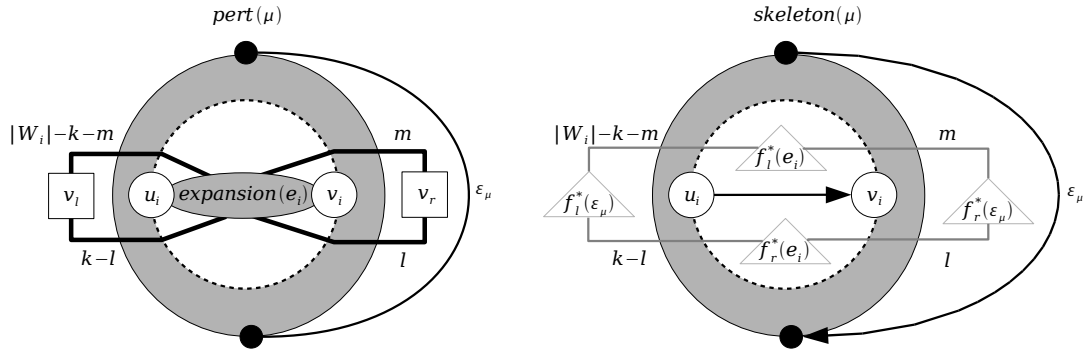


Figure 4.6: Schematic illustration concerning Equation 4.1.

Let $e_1, \ldots, e_{n\geq 1}, \varepsilon_\mu$ be the virtual edges of $skeleton(\mu)$ with pertinent nodes $\nu_1, \ldots, \nu_n$ such that $\mathcal{W}_1 := W^-(pert(\nu_1)), \ldots, \mathcal{W}_n := W^-(pert(\nu_n)) \neq \emptyset$. Further let $\mathcal{W}_0 := W^-(skeleton(\mu))$ and let the core problem be solved optimally with respect to $\nu_1, \ldots, \nu_n$. $\bar{C}_1, \ldots, \bar{C}_n$ denote the cost vectors associated with these pertinent nodes. An embedding of $pert(\mu)$ is created by fixing an embedding of $skeleton(\mu)$ and expanding all the virtual edges of $skeleton(\mu)$ except ε_μ. At first we assume the embedding Π of $skeleton(\mu)$ is given.

It is allowed to replace each virtual edge e of $skeleton(\mu)$ such that $W^-(expansion(e)) = \emptyset$ with an arbitrary embedding of $expansion(e)$. This follows from the following consideration. There is no edge inserted into an embedding of $pert(\mu)$ with an endpoint contained in $expansion(e)$. Thus at most $expansion(e)$ is crossed by an inserted edge. According to Lemma 4.1 such an edge can cross $expansion(e)$ with the same costs independent of the embedding of $expansion(e)$.

We make a case distinction. The first case is given by $n = 1, \mathcal{W}_0 = \emptyset$. When inserting edges between $v \in \{v_r, v_l\}$ and all $w \in \mathcal{W}_1$ then $0 \leq k \leq |\mathcal{W}_1|$ and $|\mathcal{W}_1| - k$ of these edges must cross the faces represented by $f_r(e_1)$ and $f_l(e_1)$ in Π, respectively, whereof $k - l$ and $0 \leq l \leq k$ have endpoint v_r and v_l respectively, and $|\mathcal{W}_1| - k - m$ and $0 \leq m \leq |\mathcal{W}_1| - k$ have endpoint v_r and v_l respectively such that $k - l + |\mathcal{W}_1| - k - m = |\mathcal{W}_1| - l - m = j$ (cf. Figure 4.6). Then the following steps obviously lead to an optimal solution:

(1) $i := 1$.

(2) Compute the dual graph Π^* of $skeleton(\mu)$ with respect to Π. Weight all edges except ε^*_μ of Π^* with the traversing costs of their primal edges.

(3) Compute $\ell_1 := |P(f_r^*(e_i),\, f_r^*(\varepsilon_\mu))|$, $\ell_2 := |P(f_r^*(e_i), f_l^*(\varepsilon_\mu))|$, $\ell_3 := |P(f_l^*(e_i), f_r^*(\varepsilon_\mu))|$, $\ell_4 := |P(f_l^*(e_i), f_l^*(\varepsilon_\mu))|$ (each weighted shortest path concerns Π^* and does not use ε_μ^*).

(4) Compute

$$C_i(j) := \min_{0\le k\le|\mathcal{W}_i|}(\bar{C}_i(k)+\min_{\substack{0\le l\le k,\\ 0\le m\le|\mathcal{W}_i|-k:\\ |\mathcal{W}_i|-l-m=j}}(\ell_1\cdot(k-l)+\ell_2\cdot l+\ell_3\cdot(|\mathcal{W}_i|-k-m)+\ell_4\cdot m)) \tag{4.1}$$

and let k' be the argument of the minimum function defining $C_i(j)$.

(5) Replace e_1 by the embedding of $pert(\nu_1) - e_1$ induced by the embedding $\Pi(k')$ of $pert(\nu_1)$ and each remaining virtual edge of $skeleton(\mu)$ except ε_μ by any embedding of its expansion graph.

The second case is given by $n > 1, \mathcal{W}_0 = \emptyset$. When inserting the edges between $v \in \{v_r, v_l\}$ and all $w \in \mathcal{W}$, then holds for $i = 1, \ldots, n$ that $0 \le k_i \le |\mathcal{W}_i|$ and $|\mathcal{W}_i| - k_i$ of these edges with endpoints $w \in \mathcal{W}_i$ must cross the faces represented by $f_r(e_i)$ and $f_l(e_i)$ in Π, respectively, whereof $k_i - l_i$ and $0 \le l_i \le k_i$ have endpoint v_r and v_l, respectively, and $|\mathcal{W}_i| - k_i - m_i$ and $0 \le m_i \le |\mathcal{W}_i| - k_i$ have endpoint v_r and v_l, respectively, such that $|\mathcal{W}_1| - l_1 - m_1 + \ldots + |\mathcal{W}_n| - l_n - m_n = j$ and $C_1(|\mathcal{W}_1| - l_1 - m_1) + \ldots + C_n(|\mathcal{W}_n| - l_n - m_n)$ crossings occur. Hence the following steps lead to an optimal solution since all possibilities are taken into account:

(1) For $i = 1, \ldots, n$, $j_i = 0, \ldots, |\mathcal{W}_i|$ compute $C_i(j_i)$ (as described in the previous case) and let k'_{j_i} be the argument of the minimum function defining $C_i(j_i)$.

(2) For $i = 1, \ldots, n$ find appropriate j_i such that $j_1 + \ldots + j_n = j$ and $C_1(j_1) + \ldots + C_n(j_n)$ is minimum.

(3) For $i = 1, \ldots, n$ replace e_i by the embedding of $pert(\nu_i) - e_i$ induced by the embedding $\Pi(k'_{j_i})$ of $pert(\nu_i)$ and replace each remaining virtual edge of $skeleton(\mu)$ except ε_μ by any embedding of its expansion graph.

Step (2) could be solved by trying all $|\mathcal{W}_1| \cdot \ldots \cdot |\mathcal{W}_n|$ possibilities in a naive manner. But this can be done more elegant as follows. Let $C_1^1 := C_1$. For $i = 2, \ldots, n$, $j_1^i = 0, \ldots, |\mathcal{W}_1| + \ldots + |\mathcal{W}_i|$ we compute

$$C_1^i(j_1^i) := \min_{\substack{0\le k\le|\mathcal{W}_1|+\ldots+|\mathcal{W}_{i-1}|\\ 0\le l\le|\mathcal{W}_i|:\\ k+l=j_1^i}}(C_1^{i-1}(k) + C_i(l)) \tag{4.2}$$

and save again all necessary integer arguments for step 3. The correctness easily follows.

The third case is given by $n > 1, \mathcal{W}_0 \neq \emptyset$. Now there must be additionally $0 \leq j_0 \leq |\mathcal{W}_0|$ and $|\mathcal{W}_0| - j_0$ inserted edges with endpoint v_r and v_l, respectively. It is easy to see that this case is similar to the previous one (it is helpful to imagine there is a further virtual edge e_0 such that $\mathcal{W}_0 = W^-(expansion(e_0))$) and that we can process the same steps as in the previous case except that i has to run from 0 to n, j_1^i has to be replaced by j_0^i running from 0 to $|\mathcal{W}_0| + \ldots + |\mathcal{W}_i|$ and $C_0(j_0)$ for $j_0 = 0, \ldots, |\mathcal{W}_0|$ has to be computed differently. Thereto the augmented dual graph $\Pi^*[\mathcal{W}_0]$ of $skeleton(\mu)$ with respect to Π and $\mathcal{W}_0$ is computed, all edges of $\Pi^*[\mathcal{W}_0]$ with endpoint $w \in \mathcal{W}_0$ are weighted with 0 and all other edges except ε_μ^* with the traversing costs of their primal edges.

The following equation defines the computation of $C_0(j_0)$. This time the shortest paths concern $\Pi^*[\mathcal{W}_0]$ and they do not use ε_μ^*. $\min\limits_{i}$ symbolizes the i-th smallest value of the minimum function.

$$C_0(j_0) := \begin{cases} 0 & j_0 = |\mathcal{W}_0| = 0 \text{ or} \\ & \mu \text{ is an S-node} \\ \sum_{w \in \mathcal{W}_0} |P(w, f_r^*(\varepsilon_\mu))| & j_0 = |\mathcal{W}_0| > 0 \\ C_0(j_0 + 1) + & \\ \min\limits_{w \in \mathcal{W}_0}^{|\mathcal{W}_0| - j_0} (|P(w, f_l^*(\varepsilon_\mu))| - |P(w, f_r^*(\varepsilon_\mu))|) & j_0 = |\mathcal{W}_0| - 1, \ldots, 0 \end{cases} \tag{4.3}$$

The equation can be explained like this. The case $j_0 = |\mathcal{W}_0| = 0$ holds trivially. The cases $j_0 = |\mathcal{W}_0| = 0$ and $j_0 = |\mathcal{W}_0| > 0$ should be clear. If μ is an S-node, the equation holds since all $w \in \mathcal{W}_0$ lie on the boundaries of $f_r(\varepsilon_\mu)$ and $f_l(\varepsilon_\mu)$. In the case $j \leq |\mathcal{W}_0| - k$ there must be $k \geq 1$ edges with endpoint v_r. Outgoing from $C_0(|\mathcal{W}_0|)$, we search for k endpoints in $\mathcal{W}_0$ which can be connected with v_l with the minimum number of crossings. This relates to add the k smallest values from the set $D := \{\{|P(w, f_l^*(\varepsilon_\mu))| - |P(w, f_r^*(\varepsilon_\mu))|\} \,|\, w \in \mathcal{W}_0\}$ to $C_0(|\mathcal{W}_0|)$. Since the computation of $C_0(j + 1)$ has already chosen the $k - 1$ smallest values from D, we need only to add the $|\mathcal{W}_0| - j_0 = |\mathcal{W}_0| - (|\mathcal{W}_0| - k) = k$-th smallest value from D to $C_0(j_0 + 1)$.

It remains to consider how an optimal solution can be computed if the embedding of $skeleton(\mu)$ is not given. We exploit the fact that $skeleton(\mu)$ has at most two embeddings which are mirror images of each other. Therefore we apply the computation with respect to j and $|\mathcal{W}| - j$ based on one arbitrarily fixed embedding of $skeleton(\mu)$. Then that embedding of $pert(\mu)$ which allows to insert the elements with less crossings is chosen. If embedding $\Pi(|\mathcal{W}| - j)$ is

the cheaper one, it must only be mirrored.

Finally let us consider the simple case in which μ is a leaf. It holds $pert(\mu) = skeleton(\mu)$ and thus we have to choose embeddings of $skeleton(\mu)$. If μ is of type S, then there is only one embedding of $skeleton(\mu)$ which is therefore the only solution. If μ is of type R, there are two embeddings of $skeleton(\mu)$ which are mirror images of each other. Then $C_0(j)$ and $C_0(|\mathcal{W}| - j)$ are computed with respect to any fixed embedding of $skeleton(\mu)$, the cheaper embedding is chosen and must possibly be mirrored.

Algorithm 4.3 presents a detailed algorithm for the computation of $\bar{C}$. The algorithm basically computes

1. C_0.
2. C_i for $i = 1, \ldots, n$ whereto $\bar{C}_i$ must be available.
3. C_0^i for $i = 1, \ldots, n$. C_0^i is obtained by combining C_0^{i-1} and C_i ($C_0^0 := C_0$).

to obtain

$$\bar{C}(j) := \min(C_0^n(j), C_0^n(|\mathcal{W}| - j)) \tag{4.4}$$

for $j = 0, \ldots, |\mathcal{W}|$. The algorithm also includes operations saving all the necessary information to construct an embedding $\Pi(j)$ of $pert(\mu)$ corresponding to $\bar{C}(j)$. Thereto a vector of lists I and a Boolean vector M are maintained. The list $I(j) = ((e_1, k'_{j_1}), \ldots, (e_n, k'_{j_n}))$ consists of 2-tuples and (e_i, k'_{j_i}) indicates that e_i has to be replaced by an embedding of $pert(\nu_i) - e_i$ induced by the embedding $\Pi(k'_{j_i})$ of $pert(\nu_i)$. $M(j) = true$ indicates that the embedding which has been constructed with respect to $\bar{C}(j)$ must be mirrored subsequently. $\mathcal{C}, \mathcal{M}$ and $\mathcal{I}$ are associative data structures. With $parent(\mu)$ as function returning the parent node of μ or nil if μ is the root of the SPQR-tree, $\mathcal{C}(\mu, parent(\mu)), \mathcal{I}(\mu, parent(\mu))$ and $\mathcal{M}(\mu, parent(\mu))$ return the cost vector $\bar{C}$, the vector of lists I and the Boolean vector M associated with μ, respectively. The reason why the pair $\mu, parent(\mu)$ is used to index each associative data structure will be explained in Section 4.3.3.

Algorithm 4.3: Computes the value of a solution of the core problem with respect to μ as S- or R-node.

Input: S- or R-node μ, $\mathcal{C}, \mathcal{I}, \mathcal{M}$

1: Let Π be the fixed embedding of $skeleton(\mu)$
2: $\mathcal{W}_0 := W^-(skeleton(\mu))$
3: Let C_0 be a vector of size $|\mathcal{W}_0| + 1$
4: Let I_0 be a vector of size $|\mathcal{W}_0| + 1$ initialized with () ▷ () is an empty list
5: **if** μ is an S-node or $\mathcal{W}_0 = \emptyset$ **then** ▷ Eq. 4.3

```
    for j_0 := 0, ..., |W_0| do
        C_0(j_0) := 0
    end for
else
    H := ∅                                                    ▷ Minimum heap
    C_0(|W_0|) := 0                                           ▷ Eq. 4.3
    Compute Π*[W_0]
    for all w ∈ W_0 do
        ℓ_r(w) := |P(f_r*(ε_μ), w)|   ▷ Weighted shortest path in Π*[W_0] not using ε*_μ
        ℓ_l(w) := |P(f_l*(ε_μ), w)|   ▷ Weighted shortest path in Π*[W_0] not using ε*_μ
        C_0(|W_0|) := C_0(|W_0|) + l_r(w)
        Insert ℓ_l(w) − ℓ_r(w) into H
    end for
    for j_0 := |W_0| − 1, ..., 0 do
        Extract the minimum value min from H
        C_0(j_0) := C_0(j_0 + 1) + min
    end for
end if
C_0^0 := C_0
I_0^0 := I_0

if n ≥ 1 then
    Compute Π* = (F*, E*)
    if μ is an S-node then
        min := ∞
        for all edges e ≠ ε_μ of skeleton(μ) do
            if c(e) < min then
                min := c(e)
            end if
        end for
        ℓ_r(f_r*(ε_μ)) := ℓ_l(f_l*(ε_μ)) := 0
        ℓ_r(f_l*(ε_μ)) := ℓ_l(f_r*(ε_μ)) := min
    else
        for all f* ∈ F* do
            ℓ_r(f*) := |P(f_r*(ε_μ), f*)|   ▷ Weighted shortest path in Π* not using ε*_μ
            ℓ_l(f*) := |P(f_l*(ε_μ), f*)|   ▷ Weighted shortest path in Π* not using ε*_μ
        end for
    end if

    for i := 1, ..., n do                                     ▷ Eq. 4.1
        Let C_i, I_i be vectors of size |W_i| + 1
        Initialize C_i with ∞
        for k := 0 to |W_i| do
```

```
47:             for l := 0 to k do
48:                 for m := 0 to |W_i| − k do
49:                     C̄_i := C(ν_i, parent(ν_i))
50:                     C' := C̄_i(k) + ℓ_r(f_r^*(e_i)) · (k − l) + ℓ_l(f_r^*(e_i)) · l+
                         ↪ ℓ_r(f_l^*(e_i)) · (|W_i| − k − m) + ℓ_l(f_l^*(e_i)) · m
51:                     if C' < C_i(|W_i| − l − m) then
52:                         C_i(|W_i| − l − m) := C'
53:                         I_i(|W_i| − l − m) := ((e_i, k))
54:                     end if
55:                 end for
56:             end for
57:         end for

58:         Let C_0^i be a vector of size |W_0| + ... + |W_i| + 1 initialized with ∞
59:         Let I_0^i be a vector of size |W_0| + ... + |W_i| + 1
60:         for k := 0, ..., |W_0| + ... + |W_{i−1}| do                    ▷ Eq. 4.2
61:             for l := 0, ..., |W_i| do
62:                 if C' := C_0^{i−1}(k) + C_i(l) < C_0^i(k + l) then
63:                     C_0^i(k + l) := C'
64:                     I_0^i(k + l) := I_0^{i−1}(k) + I_i(l)              ▷ List concatenation
65:                 end if
66:             end for
67:         end for
68:     end for
69: end if

70: I := I_0^n
71: Let C̄, M be vectors of size |W| + 1
72: for j := 0, ..., |W| do                                                ▷ Eq. 4.4
73:     if C_0^n(|W| − j) < C_0^n(j) then
74:         C̄(j) := C_0^n(|W| − j)
75:         M(j) := true
76:         C̄(|W| − j) := C_0^n(|W| − j)
77:         M(|W| − j) := false
78:         I(j) := I(|W| − j)
79:     end if
80: end for
81: C(μ, parent(μ)) := C̄
82: I(μ, parent(μ)) := I
83: M(μ, parent(μ)) := M
```

The worst-case running time of Algorithm 4.3 is $\mathcal{O}(|skeleton(\mu)| \cdot |W|^3)$. We analyze it by blocks of lines. We use $S := skeleton(\mu)$ as abbreviation.

- Lines 1–25: The augmented dual graph in line 12 can be computed in time $\mathcal{O}(|S|)$. The lengths of all weighted shortest paths in lines 14 and 15 can be computed by the algorithm in [33] in the augmented dual graph starting from the two faces that are separated by the reference edge in time $\mathcal{O}(|S|)$. At most $\mathcal{O}(|W|)$ values are inserted into, e.g., a Fibonacci heap in the loop in line 17. One insertion operation takes amortized time $\mathcal{O}(1)$. Extracting the minimum value out of the Fibonacci heap which contains at most $\mathcal{O}(|W|)$ elements takes time amortized $\mathcal{O}(log\,|W|)$ in line 20. Since the loop in line 19 iterates at most $\mathcal{O}(|W|)$ times, extracting all values takes time $\mathcal{O}(|W| \cdot log\,|W|)$. The running time of this block is $\mathcal{O}(|S| + |W| + |W| \cdot log\,|W|) = \mathcal{O}(|S|)| + |W| \cdot log\,|W|)$.

- Lines 27–42: The dual graph in line 27 can be computed in time $\mathcal{O}(|S|)$. The worst-case running time of this block is dominated by the else block in lines 37-42. Again the lengths of all shortest paths in line 39 and 40 can be computed in time $\mathcal{O}(|S|)$ in the dual graph. Hence the running time of this block is $\mathcal{O}(|S|)$.

- Lines 43–69: The outermost loop in line 43 iterates as many times as μ children has. The number of children of μ is bounded by $|S|$. The three nested loops in lines 46–48 obviously cause $\mathcal{O}(|W|^3)$ computation steps. The operations within these loops take constant time. The initializations of the vectors in lines 45 and 58 take time $\mathcal{O}(|W|)$. The running time of the two nested loops in lines 60 and 61 is $\mathcal{O}(|W|^2)$. The concatenation of the lists in line 64 takes constant time. Thus, this part has a running time of $\mathcal{O}(|S| \cdot (|W|^3 + |W|^2 + |W|)) = \mathcal{O}(|S| \cdot |W|^3)$.

- Lines 70–83: The loop in this block processes at most $\mathcal{O}(|W|)$ computation steps of which each has constant running time.

Therefore we obtain as worst-case running time:

$$\mathcal{O}(|S|)| + |W| \cdot log\,|W|) + \mathcal{O}(|S|) + \mathcal{O}(|S| \cdot |W|^3) + \mathcal{O}(|W|) = \mathcal{O}(|S| \cdot |W|^3)$$

P-node. Now let μ be a P-node and a j with $0 \leq j \leq |\mathcal{W}|$ be given. We have only to deal with the non-simple case in which μ is an inner node. The reason is that $W^-(pert(\mu)) = \emptyset$ always holds if μ is a leaf and of type P. Moreover let $e_1, \ldots, e_{n\geq 2}, \varepsilon_\mu$ be the edges of $skeleton(\mu)$ and let the core problem already be solved for all children of μ. If e_i is a virtual edge with pertinent node ν_i such

that $W^-(pert(\mu)) \neq \emptyset$, then again $\bar{C}_i$ denotes its cost vector. In contrast to the previous paragraph we define for $i = 1, \ldots, n$

$$\mathcal{W}_i := \begin{cases} \emptyset & e_i \text{ is a proper edge} \\ W^-(pert(\nu_i)) & e_i \text{ is a virtual edge with pertinent node } \nu_i \end{cases}$$

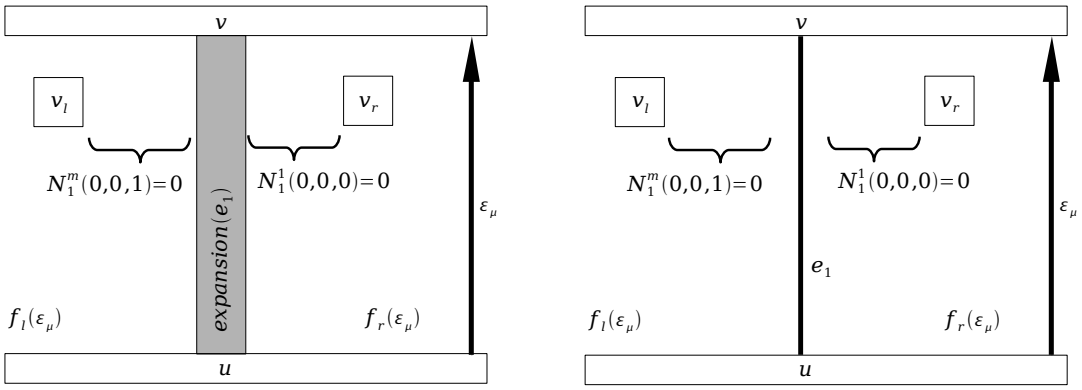
Again we use the illustrative description of a subsolution of the core problem with respect to μ and j and make some observations which allow to derive a dynamic programming algorithm. Thereto we consider $expansion(e_n)$ or e_n if it is a proper edge. Without loss of generality we can assume that ε_μ lies on the boundary of the external face of any embedding of $pert(\mu)$.

The observations are the following. With respect to the imaginary direction of ε_μ, to the right and left of $expansion(e_n)$ or e_n there are subgraphs of $pert(\mu)$. Such a subgraph is either empty, an expansion graph or an edge. We call the set of subgraphs to the right or left simply structure. Figure 4.7(c) illustrates this for another expansion graph. If e_n is a virtual edge and $W^-(expansion(e_n)) \neq \emptyset$ holds, then there are $0 \leq j_n \leq |\mathcal{W}_n|$ and $|\mathcal{W}_n| - j_n$ inserted edges connecting endpoints in $expansion(e_n)$ and v_r as well as v_l, respectively, crossing the structure to the right and left of $expansion(e_n)$. We state that the embedding and permutation of the subgraphs of the structure to the right and left of $expansion(e_n)$ or e_n is independent from the embedding of $expansion(e_n)$ or e_n. Especially, an inserted edge with endpoint in $expansion(e_n)$ causes the same number of crossings when changing the embedding (conclusion of Lemma 4.1) and/or permutation of a subgraph. This is the most important observation which the correctness of our algorithm is based on. Let f be the face bordered by edges of $expansion(e_n)$ or e_n and edges of the right/left structure. We define the thickness t of the right/left structure as the minimum number of crossings which arises when inserting a vertex into f and connecting it with v_r/v_l. So t is the sum of the traversing costs of the edges or virtual edges corresponding to the right/left structure. Clearly, $expansion(e_n)$ or e_n is possibly crossed by inserted edges outgoing from the right/left structure. The embedding of $expansion(e_n)$ or e_n is also irrelevant for these edges. If e_n or $expansion(e_n)$ is deleted from this embedding, obviously all the just mentioned observations are valid with respect to $expansion(e_{n-1})$ or e_{n-1}.

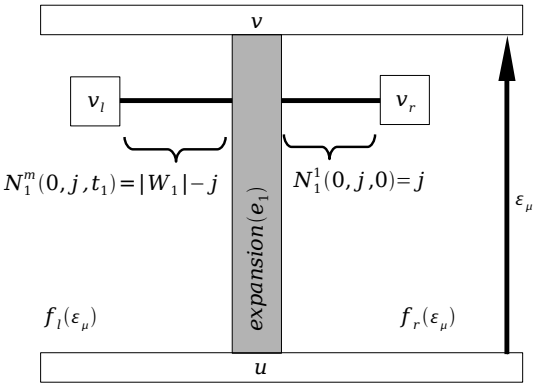
According to these observations the following algorithm solves the core problem with respect to μ:

(1) Solve the core problem with respect to μ under assumption that $skeleton(\mu)$ consists of $e_1, \ldots, e_{n-1}, \varepsilon_\mu$ only and the constraint that the right and left structure[4] of $expansion(e_{n-1})$ or e_{n-1} has thickness i and $t_{n-2} - i$ ($t_0 := 0$, $t_j := \sum_{1 \leq k \leq j} c(e_k)$ for $j = 1, \ldots, n-1$), respectively, for (adequate) $i =$

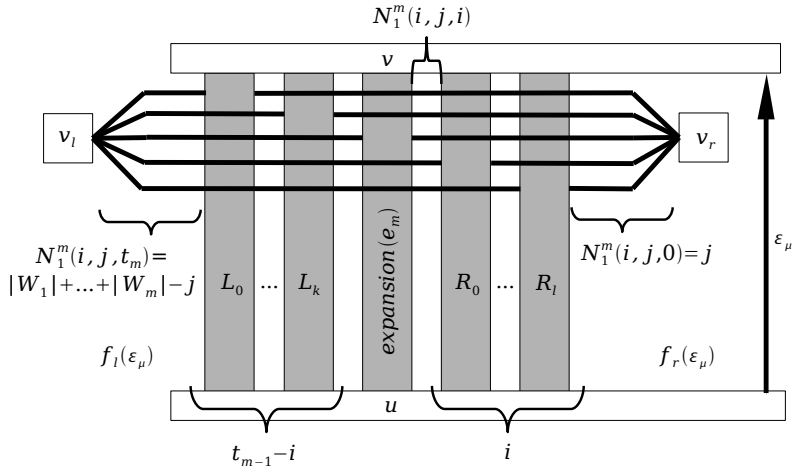
[4]Each structure must not contain a subgraph of one expansion graph.



(a)



(b)



(c) $L_0, \ldots, R_l$ represent each either an edge, expansion or empty graph.

Figure 4.7: Pertinent graphs with inserted elements visualizing the computation of a subsolution of the core problem with respect to a P-node.

$0, \ldots, t_{n-2}$. Let $\Pi(i, j_1^{n-1})$ for $j_1^{n-1} = 0, \ldots, |\mathcal{W}_1| + \ldots + |\mathcal{W}_{n-1}|$ be the one obtained embedding of the modified pertinent graph.

(2) For $i = 0, \ldots, t_{n-2}$, $j_1^{n-1} = 0, \ldots, |\mathcal{W}_1| + \ldots + |\mathcal{W}_{n-1}|$ extend $\Gamma := \Pi(i, j_1^{n-1})$ as follows:

(a) If e_n is a proper or virtual edge such that $W^-(expansion(e_n)) = \emptyset$, extend Γ for $i' = 0, \ldots, t_{n-1}$ by embedding e_n or $expansion(e_n)$ arbitrarily into that face f of Γ such that the structure to the right of e_n or $expansion(e_n)$ has thickness i'. Edges crossing f previously have to cross e_n or $expansion(e_n)$ with the minimum number of crossings. Each time save the best embedding.

(b) Else, extend Γ for $i' = 0, \ldots, t_{n-1}$, $j_n = 0, \ldots, |\mathcal{W}_n|$ by embedding $expansion(e_n)$ according to $\Pi(j_n)$ into that face f of Γ such that the structure to the right of $expansion(e_n)$ has thickness i' and by inserting j_n and $|\mathcal{W}_n| - j_n$ edges between appropriate affected vertices of $expansion(e_n)$ and v_r and v_l, respectively, such that there are $0 \leq j \leq |\mathcal{W}|$ and $|\mathcal{W}| - j$ edges with endpoint v_r and v_l, respectively. Inserted edges crossing f previously have to cross $expansion(e_n)$ with the minimum number of crossings. Each time save the best embedding.

It is easy to see that step 1 of this algorithm could be solved recursively. In the case that $n = 2$ holds, step 1 can be solved trivially.

In fact, it suffices again to compute the costs and to save extra information to be able to construct embeddings. As is indicated by the algorithm above, due to the additional constraint (a certain thickness to the right of an edge or expansion graph) we compute a $(t_{n-1} + 1 \times |\mathcal{W}_1| + \ldots + |\mathcal{W}_n| + 1)$ cost matrix C_1^n. From it, we obtain the actual cost vector for $j = 0, \ldots, |\mathcal{W}|$ as follows

$$\bar{C}(j) := \min_{0 \leq i \leq t_{n-1}} (C_1^n(i, j)) \tag{4.5}$$

C_1^n is computed stepwise, i.e., for $m = 2, \ldots, n$ the $(t_{m-1} + 1 \times |\mathcal{W}_1| + \ldots + |\mathcal{W}_m| + 1)$ cost matrix C_1^m (cf. step 1) is computed and built from the cost matrix C_1^{m-1} as basically described by step 2 in the above algorithm. C_1^1 is the initial cost matrix that can directly be built. Before we give all optimality equations, we have to explain the meaning of the $(t_{m-1} + 1 \times |\mathcal{W}_1| + \ldots + |\mathcal{W}_m| + 1 \times t_m + 1)$ auxiliary matrix N_1^m used for the computation of C_1^{m+1}. Thereto consider the embedding $\Pi(i, j)$ of the modified pertinent graph belonging to $C_1^m(i, j)$ into which v_r, v_l and appropriate edges are inserted. For $k = 0, \ldots, t_m$ the entry $N_1^m(i, j, k)$ saves the number of edges crossing that face f of $\Pi(i, j)$ such that the structure to the right of f has thickness k (cf. Figure 4.7(c)). If f is bordered by edges belonging to one expansion graph then $N_1^m(i, j, k)$ is defined as infinite.

The following equations define how to obtain the entries of the initial matrices for $j = 0, \ldots, |\mathcal{W}_1|$ and $k = 0, \ldots, t_1$. If e_1 is a proper edge the first two equations and if e_1 is a virtual edge such that $W^-(expansion(e_1)) = \emptyset$, the latter two equations are valid.

$$C_1^1(0, j) := 0 \tag{4.6}$$

$$N_1^1(0, j, k) := \begin{cases} 0 & k = 0, t_1 \\ \infty & 0 < k < t_1 \end{cases} \tag{4.7}$$

$$C_1^1(0, j) := \bar{C}_1(j) \tag{4.8}$$

$$N_1^1(0, j, k) := \begin{cases} j & k = 0 \\ \infty & 0 < k < t_1 \\ |\mathcal{W}_1| - j & k = t_1 \end{cases} \tag{4.9}$$

The next series of equations define how to obtain the entries of the matrices in the non-initial case, so for $m = 2, \ldots, n$, $i = 0, \ldots, t_m$, $j = 0, \ldots, |\mathcal{W}_1| + \ldots + |\mathcal{W}_m|$ and $k = 0, \ldots, t_m$. Again, if e_m is a proper edge the first two equations and if e_m is a virtual edge such that $W^-(expansion(e_m)) = \emptyset$, the first two equations and otherwise the latter two ones are valid. Note, in Equation 4.11 and 4.13 we use i', j' as the arguments defining $C_1^m(i, j)$.

$$C_1^m(i, j) := \min_{0 \le i' \le t_{m-2}} (C_1^{m-1}(i', j) + N_1^{m-1}(i', j, i)) \tag{4.10}$$

$$N_1^m(i, j, k) := \begin{cases} N_1^{m-1}(i', j, k) & 0 \le k \le i \\ \infty & i < k < i + c(e_m) \\ N_1^{m-1}(i', j, k - c(e_m)) & i + c(e_m) \le k \le t_m \end{cases} \tag{4.11}$$

$$C_1^m(i, j) := \min_{\substack{0 \le i' \le t_{m-2}, \\ 0 \le j' \le |\mathcal{W}_1| + \ldots + |\mathcal{W}_{m-1}|: \\ 0 \le j - j' \le |\mathcal{W}_m|}} \begin{pmatrix} C_1^{m-1}(i', j') + \bar{C}_m(j - j') + \\ N_1^{m-1}(i', j', i) \cdot c(e_m) + \\ i \cdot (j - j') + \\ (t_{m-1} - i) \cdot (|\mathcal{W}_m| - j + j') \end{pmatrix} \tag{4.12}$$

$$N_1^m(i, j, k) := \begin{cases} N_1^{m-1}(i', j', k) + (j - j') & 0 \le k \le i \\ \infty & i < k < i + c(e_m) \\ N_1^{m-1}(i', j', k - c(e_m)) + & \\ |\mathcal{W}_m| - j + j' & i + c(e_m) \le k \le t_m \end{cases} \tag{4.13}$$

For the explanation of Equation 4.6–4.9 confer to Figure 4.7(a) and 4.7(b)). We explain only Equation 4.12 whereof the explanation of Equation 4.13 follows. The remaining equations can be shown similarly. We orientate by step 2 of the above algorithm. The embedding $\Pi(i', j')$ (with inserted elements) of the modified pertinent graph belonging to $C_1^{m-1}(i', j')$ "is extended". Clearly, there are already $C_1^{m-1}(i', j')$ crossings. $expansion(e_m)$ is embedded according to $\Pi(j - j'')$ into the face of $\Pi(i', j')$ such that the structure to the right of $expansion(e_m)$ has thickness i. When inserting edges between appropriate affected vertices of $expansion(e_m)$ and v_r as well as v_l, the right and left structure each with thickness i and $t_{m-1} - i$ is crossed by these edges producing $i \cdot (j - j') + (t_{m-1} - i) \cdot (|\mathcal{W}_m| - (j - j''))$ crossings. $expansion(e_m)$ is crossed by inserted edges itself, namely by edges with and without endpoint in $expansion(e_m)$. These crossings relate to the term $N_1^{m-1}(i', j', i') \cdot c(e_m) + \bar{C}_m(j - j')$.

Algorithm 4.4 presents a detailed algorithm. This algorithm also maintains a vector I of lists. The list $I(j) = ((e'_1, k'_{j_1}), \ldots, (e'_n, k'_{j_n}))$ serves for the same purpose as described for Algorithm 4.3 and additionally encodes the embedding of $skeleton(\mu)$. That means, if $\varepsilon_\mu = \{u, v\}$ has the imaginary direction from u (v) to v (u), then $e'_1, \ldots, e'_n, \varepsilon_\mu$ is the cyclic clockwise order of the edges incident to u (v).

Algorithm 4.4: Computes the value of a solution of the core problem with respect to μ as P-node.

Input: P-node μ, $\mathcal{C}$, $\mathcal{I}$

1: Let $e_1, \ldots, e_n, \varepsilon_\mu$ be the edges of $skeleton(\mu)$
2: Let C_1^1, I_1^1 be $(1 \times |\mathcal{W}_1| + 1)$-matrices and N_1^1 be a $(1 \times |\mathcal{W}_1| + 1 \times t_1 + 1)$-matrix initialized with ∞
3: **if** e_1 is a proper edge **then**
4: $\quad C_1^1(0,0) := N_1^1(0,0,0) := N_1^1(0,0,1) := 0$ ▷ Eq. 4.6, 4.7
5: **else**
6: $\quad$ **for** $j = 0, \ldots, \mathcal{W}_1$ **do**
7: $\quad\quad$ **if** $W^-(expansion(e_1)) = \emptyset$ **then** ▷ Eq. 4.6, 4.7
8: $\quad\quad\quad C_1^1(0,j) := 0$
9: $\quad\quad\quad N_1^1(0,j,0) := N_1^1(0,j,t_1) := 0$
10: $\quad\quad\quad I_1^1(0,j) := ((e_1, 0))$
11: $\quad\quad$ **else** ▷ Eq. 4.8, 4.9
12: $\quad\quad\quad C_1^1(0,j) := \bar{C}_1(j)$
13: $\quad\quad\quad N_1^1(0,j,0) := j$
14: $\quad\quad\quad N_1^1(0,j,t_1) := |\mathcal{W}_1| - j$

15: $I_1^1(0,j) := ((e_1,j))$
16: **end if**
17: **end for**
18: **end if**

19: **for** $m = 2,\ldots,n$ **do**
20: Let C_1^m, A_1^m and I_1^m be $(t_{m-1}+1 \times |\mathcal{W}_1| + \ldots |\mathcal{W}_m| + 1)$-matrices initialized with ∞ and N_1^m be a $(t_{m-1}+1 \times |\mathcal{W}_1| + \ldots + |\mathcal{W}_m| + 1 \times t_m + 1)$-matrix ▷ A_1^m is an auxiliary matrix only used in this algorithm
21: **for** $i' := 0,\ldots,t_{m-2}$ **do**
22: **for** $j' := 0,\ldots,|\mathcal{W}_1| + \ldots + |\mathcal{W}_{m-1}|$ **do**
23: **for** $i := 0,\ldots,t_{m-1}$ **do**
24: **if** e_m is a proper edge or $W^-(expansion(e_m)) = \emptyset$ **then**
25: $C' := C_1^{m-1}(i',j') + N_1^{m-1}(i',j',i)$ ▷ Eq. 4.10
26: **if** $C' < C_1^m(i,j')$ **then**
27: $C_1^m(i,j') := C'$
28: $A_1^m(i,j') := (i',j',0)$
29: **end if**
30: **else**
31: **for** $j := 0,\ldots,|\mathcal{W}_m|$ **do**
32: $C' := C_1^{m-1}(i',j') + \bar{C}_m(j) + N_1^{m-1}(i',j',i) \cdot c(e_m) + i \cdot j + (t_{m-1} - i) \cdot (|\mathcal{W}_m| - j)$ ▷ Eq. 4.12
33: **if** $C' < C_1^m(i,j'+j)$ **then**
34: $C_1^m(i,j'+j) := C'$
35: $A_1^m(i,j'+j) := (i',j',j)$
36: **end if**
37: **end for**
38: **end if**
39: **end for**
40: **end for**
41: **end for**

42: **for** $i := 0,\ldots,t_{m-1}$ **do**
43: **for** $j := 0,\ldots,|\mathcal{W}_1| + \ldots + |\mathcal{W}_m|$ **do**
44: Let $A_1^m(i,j) = (i',j',j'')$
45: **for** $k := 0,\ldots,t_m$ **do** ▷ Eq. 4.11, 4.13
46: **if** $k \leq i$ **then**
47: $N_1^m(i,j,k) := N_1^{m-1}(i',j',k) + j''$
48: **else if** $i < k < i + c(e_m)$ **then**
49: $N_1^m(i,j,k) := \infty$
50: **else**
51: $N_1^m(i,j,k) := N_1^{m-1}(i',j',k - c(e_m)) + |\mathcal{W}_m| - j''$
52: **end if**

53: **end for**

54: **if** $i = 0$ **then**
55: $I_1^m(i,j) := I_1^m(i,j) + ((e_m, j''))$
56: **else if** $i = t_{m-1}$ **then**
57: $I_1^m(i,j) := ((e_m, j'')) + I_1^m(i,j)$
58: **else**
59: Let $I_1^{m-1}(i',j') = ((e'_1, k'_1), \ldots, (e'_{m-1}, k'_{m-1}))$
60: **for** $k := m-1, \ldots, 1$ **do**
61: **if** $i = c(e'_{m-1}) + \ldots + c(e'_k)$ **then**
62: $I_1^m(i,j) := (\ldots, (e'_{k-1}, k'_{k-1}), (e_m, j''), (e'_k, k'_k), \ldots)$
63: Break
64: **end if**
65: **end for**
66: **end if**
67: **end for**
68: **end for**
69: **end for**

70: Let $\bar{C}, I$ be vectors each of size $|\mathcal{W}| + 1$
71: **for** $j := 0, \ldots, |\mathcal{W}|$ **do**
72: $i' := 0$
73: **for** $i := 0, \ldots, t_{n-1}$ **do**
74: **if** $C_1^n(i,j) \leq C_1^n(i',j)$ **then**
75: $i' := i$
76: **end if**
77: **end for**
78: $\bar{C}(j) := C_1^n(i',j)$
79: $I(j) := I_1^n(i',j)$
80: **end for**
81: $\mathcal{C}(\mu, parent(\mu)) := \bar{C}$
82: $\mathcal{I}(\mu, parent(\mu)) := I$

The worst-case running time of Algorithm 4.4 is $\mathcal{O}(|skeleton(\mu)| \cdot t_{max}^2 \cdot |W|^2)$ with t_{max} as the maximum thickness of a skeleton associated with a P-node in the given SPQR-tree in the core problem. It is generated by the block in lines 19–69. As supplied before we use $S := skeleton(\mu)$ as abbreviation temporarily and analyze the running time by blocks of lines:

- Lines 2–18: The runtime of this block is $\mathcal{O}(t_{max} \cdot |W|)$ caused by the initialization of the auxiliary matrix N_1^1.

- Lines 20–41: The initialization of the three matrices needs time $\mathcal{O}(t_{max} \cdot |W|)$ and the four nested loops have a runtime of $\mathcal{O}(t_{max}^2 \cdot |W|^2)$ since $t_{m-2} \leq t_n \leq t_{max}$ and the runtime of the operations within the innermost loop can be estimated by $\mathcal{O}(1)$.
- Lines 42–68: $\mathcal{O}(t_{max} \cdot |W| \cdot (t_{max} + |S|)) = \mathcal{O}(t_{max}^2 \cdot |W|)$ due to $k \leq n \leq |S|$ (line 60) and $t_{max}+3 \geq |S|$. Again, this is simply the product of the number of steps in the nested loops.
- Lines 19–69: Having regard to the outermost loop of this block iterating $n \leq |S|$ times and the previous runtimes, we obtain a runtime of $\mathcal{O}(|S| \cdot (t_{max}^2 \cdot |W|^2))$ for this block.
- Lines 70–82: $\mathcal{O}(t_{max} \cdot |W|)$.

Constructing an Embedding. A detailed algorithm for constructing an embedding $\Pi(j)$ of $pert(\mu)$ belonging to $\bar{C}(j)$, i.e., a subsolution of the core problem, is given by Algorithm 4.5. This algorithm processes a recursive top-down replacement of the virtual edges of $skeleton(\mu)$ with embeddings of their expansion graphs. Thereto it utilizes the extra information stored during the computation of $\bar{C}$.

The runtime of Algorithm 4.5 is $\mathcal{O}(|V|^2)$. At most all skeleton edges are considered in all recursions together which takes time $\mathcal{O}(|V|)$ due to the size of $\mathcal{T}$. Mirroring the embedding of $pert(\mu)$ means mirroring all the skeletons associated with nodes in the subtree $\mathcal{T}_\mu$ of $\mathcal{T}$. Hence in the worst-case, the embedding of each skeleton is as many times mirrored as its associated node in $\mathcal{T}_\mu$ is deep. The maximum depth of $\mathcal{T}_\mu$ is $\mathcal{O}(|V|)$, since there are at most $\mathcal{O}(|V|)$ nodes in $\mathcal{T}_\mu$. Hence all the mirroring operations take time $\sum_{\mu' \text{ in } \mathcal{T}_\mu} |V| \cdot |skeleton(\mu')| = \mathcal{O}(|V|^2)$. The total worst-case running time is therefore $\mathcal{O}(|V|^2)$.

Algorithm 4.5: Computes an embedding $\Pi(j)$ of $pert(\mu)$ belonging to $\bar{C}(j)$.

Input: Rooted SPQR-tree $\mathcal{T}$, node μ of $\mathcal{T}$, integer $0 \leq j \leq |W^-(pert(\mu))|$, $\mathcal{I}$, $\mathcal{M}$
Output: Embedding $\Pi(j)$ of $pert(\mu)$

1: **if** $\mathcal{I}(\mu) = nil$ **then** ▷ No cost vector exists associated with μ
2: **return** any planar embedding $\Pi(j)$ of $pert(\mu)$
3: **end if**
4: $I := \mathcal{I}(\mu)$
5: Let $I(j) = ((e'_1, k'_{j_1}), \ldots, (e'_n, k'_{j_n}))$ and $\varepsilon_\mu = \{u, v\}$
6: **if** μ is a P-node **then**
7: Let $\Pi(j)$ be the embedding of $skeleton(\mu)$ such that $e'_1, \ldots, e'_n, \varepsilon_\mu$ is the cyclic

```
        ↪ clockwise order of the edges incident to u or v depending on the imaginary
        ↪ direction of ε_μ
 8: else
 9:     Let Π(j) be the fixed embedding of skeleton(μ)
10: end if
11: for i := 1, ..., n ≥ 1 do
12:     if e'_i is a virtual edge then
13:         Let ν'_i be the pertinent node of e'_i
14:         Call this algorithm recursively with inputs T, ν'_i, k'_{j_i}, I, M and obtain
              ↪ an embedding Π'(k'_{j_i}) of pert(ν'_i)
15:         Replace e'_i in Π(j) by the embedding of pert(ν'_i) − e'_i induced by Π'(k'_{j_i})
16:     end if
17: end for
18: if M := M(μ) ≠ nil and M(j) then
19:     Mirror Π_j
20: end if
21: for each virtual edge e of skeleton(μ) do
22:     Replace e in Π(j) by any planar embedding of expansion(e)
23: end for
24: return Π(j)
```

4.3.3 Solving VIP-VAR for Biconnected Graphs

Algorithm 4.6 presents the frame algorithm for solving VIP-VAR for a biconnected planar graph optimally in polynomial time. An overview about the relevant steps of this algorithm is the following:

1. First of all the SPQR-tree $\mathcal{T}$ of the biconnected planar input graph G is computed.

2. The algorithm roots $\mathcal{T}$ at each of its nodes once.

3. The solution value of the core problem is computed with respect to either each node of $\mathcal{T}$ except the root or each node of the bottom-up path from the previous to the current root except the current root. This depends on whether the loop iterates for the first time. Additionally, the traversing costs of the virtual edges of these nodes are computed.

4. Afterwards the insertion of ϑ into each face of an embedding of the root's skeleton is simulated and the overall costs are computed, i.e., the minimum number of crossings which occur if ϑ and its incident edges would be

inserted in reality. This computation is delegated to the function CrossingNumberSR in the case the current root is of type S or R and otherwise to the function CrossingNumberP. If the costs of such a solution are less than the best solution so far, necessary data structures are saved to provide a construction of an embedding of G later on.

5. Finally, an embedding of G is computed. This done by applying Algorithm 4.5.

(6. It remains to insert ϑ and its incident edges in fact as described in Section 4.2.)

Algorithm 4.6: Computes an optimal solution of VIP-VAR for a biconnected planar graph.

Input: Biconnected planar graph $G = (V, E)$, $W \subseteq V$, $W \neq \emptyset$
Output: Embedding of G

1: $\mu_r^{opt} := \mu_r^{old} := nil$
2: $c_1 := \infty$
3: $\mathcal{C} := \mathcal{I} := \mathcal{I}_{opt} := \mathcal{M} := \mathcal{M}_{opt} := \emptyset$
4: Compute the SPQR-tree $\mathcal{T}$ of G ▷ Fixes embeddings of all skeletons
5: **if** $\mathcal{T}$ consists of one node μ_r only **then**
6: **return** any embedding of $skeleton(\mu_r)$
7: **end if**
8: **for** each node μ_r of $\mathcal{T}$ **do**
9: Root $\mathcal{T}$ at μ_r
10: **if** $W = W(skeleton(\mu_r))$ **then**
11: Let Π be the fixed embedding of $skeleton(\mu_r)$
12: **for** each virtual edge e of $skeleton(\mu_r)$ **do**
13: Replace e in Π by any embedding of $expansion(e)$
14: **end for**
15: **return** Π
16: **end if**
17: **if** $\mu_r^{old} = nil$ **then**
18: Let $\nu_1, \ldots, \nu_{m \geq 2} = \mu_r$ be a bottom-up order[5] of all nodes of $\mathcal{T}$
19: **else**
20: Let $\mu_r^{old} = \nu_1, \ldots, \nu_{m \geq 2} = \mu_r$ be a bottom-up path in $\mathcal{T}$
21: **end if**

[5]The bottom-up order of these nodes is the order in which they are visited in a bottom-up traversal of $\mathcal{T}$. That means if a node is visited, all of its children have to be visited already before.

```
22:     for i := 1, ..., m − 1 do
23:         Let e_i be the virtual edge of ν_i
24:         Compute c(e_i)
25:         if W⁻(pert(ν_i)) ≠ ∅ and C(ν_i, parent(ν_i)) = nil then
26:             if ν_i is an S- or R-node then        ▷ Computing the cost vector w.r.t. ν_i
27:                 Call Algorithm 4.3 with inputs ν_i, C, I, M
28:             else
29:                 Call Algorithm 4.4 with inputs ν_i, C, I
30:             end if
31:         end if
32:     end for
33:     μ_r^old := μ_r
34:     if μ_r is an S- or R-node then
35:         c_2 :=CrossingNumberSR(μ_r, C, I)
36:     else
37:         c_2 :=CrossingNumberP(μ_r, C, I)
38:     end if
39:     if c_2 < c_1 then
40:         c_1 := c_2
41:         μ_r^opt := μ_r
42:         I_opt := I
43:         M_opt := M
44:     end if
45: end for
46: Root T at μ_r^opt
47: return the embedding of G obtained by calling Algorithm 4.5 with inputs
      ↪ T, μ_r^opt, I_opt, M_opt, 0

 1: function CrossingNumberSR(Root node μ_r of type S or R, C, I)
 2:     Let Π be the fixed embedding of skeleton(μ_r)
 3:     Let e_1, ..., e_{n≥1} be the virtual edges of skeleton(μ_r) with pertinent nodes ν_1, ..., ν_n
          ↪ such that W_1 := W⁻(pert(ν_1)), ..., W_n := W⁻(pert(ν_n)) ≠ ∅
 4:     Compute Π* = (F*, E*)
 5:     if W(skeleton(μ_r)) ≠ ∅ and μ_r is an R-node then
 6:         Compute Π*[W(skeleton(μ_r))]
 7:     end if
 8:     c_0 := ∞
 9:     Let I be a vector of size 1
10:     I(0) := ()                                          ▷ Empty list
11:     for all f* ∈ F* do                    ▷ Regard ϑ as inserted into f
12:         c_1 := 0
13:         I' := ()
14:         if W(skeleton(μ_r)) ≠ ∅ and μ_r is an R-node then
```

```
15:             for all w ∈ W(skeleton(μ_r)) do
16:                 c_1 := c_1 + |P(w, f*)|                ▷ Weighted shortest path
                                                             in Π*[W(skeleton(μ_r))]
17:             end for
18:         end if
19:         for all f̄* ≠ f* ∈ F* do
20:             ℓ(f̄*) := |P(f̄*, f*)|                       ▷ Weighted shortest path in Π*
21:         end for
22:         for i := 1, ..., n do
23:             c_2 := ∞
24:             C̄_i := C(ν_i)
25:             for j := 0, ..., |W_i| do
26:                 c_3 := C̄_i(j) + ℓ(f*_r(e_i)) · j + ℓ(f*_l(e_i)) · (|W_i| − j)
27:                 if c_3 < c_2 then
28:                     c_2 := c_3
29:                     j' := j
30:                 end if
31:             end for
32:             I' := I' + ((e_i, j'))
33:             c_1 := c_1 + c_2
34:         end for
35:         if c_1 < c_0 then
36:             c_0 := c_1
37:             I(0) := I'
38:         end if
39:     end for
40:     I(μ_r, parent(μ_r)) := I
41:     return c_0
42: end function
```

```
1: function CrossingNumberP(Root node μ_r of type P, C, I)
2:     Let u, v be the poles and e_1, ..., e_{n≥3} be the edges of skeleton(μ)
3:     c_0 := ∞
4:     c_1 := 0
5:     for i := 1, ..., n do
6:         c_1 := c_1 + c(e_i)
7:     end for
8:     for i := 1, ..., n − 1 do
9:         for j := i + 1, ..., n do
10:            Split μ into two new nodes μ_{r_1}, μ_{r_2}
11:            Let skeleton(μ_{r_1}) be induced by e'_1 := e_i, e'_2 := e_j and a new virtual
                 ↪ edge e'_3 := {u, v} with pertinent node μ_{r_2}
12:            Let skeleton(μ_{r_1}) have the embedding Π in which e'_1, e'_2, e'_3 is the cyclic
```

```
                ↪ clockwise order of these edges incident to u
13:             Let skeleton(μ_{r_2}) be induced by e_1, ..., e_n ∉ {e'_1, e'_2} and ε_{μ_{r_2}} := e'_3
14:             Let f be the face of Π defined by e'_1, e'_2     ▷ Regard ϑ as inserted into f
15:             Compute Π* = (F*, E*)
16:             for all f̄* ≠ f* ∈ F* do
17:                 ℓ(f̄*) := |P(f̄*, f*)|                          ▷ Weighted shortest path in Π*
18:             end for
19:             Call Algorithm 4.4 with inputs μ_{r_2}, C, I
20:             c(e'_3) := c_1 − c(e'_1) − c(e'_2)
21:             c_2 := 0;
22:             for l := 1, 2, 3 do
23:                 if e'_l is a proper edge then
24:                     k_l := 0
25:                     Continue
26:                 else if W := W⁻(expansion(e'_l)) ≠ ∅ then
27:                     Let ν_l be the pertinent node of e'_l
28:                     C̄_l := C(ν_l)
29:                     c_3 := ∞
30:                     for m := 0, ..., |W| do
31:                         c_4 := C̄_l(m) + ℓ(f*_r(e'_l)) · m + ℓ(f*_l(e'_l)) · (|W| − m)
32:                         if c_4 < c_3 then
33:                             c_3 := c_4
34:                             k_l := m
35:                         end if
36:                     end for
37:                     c_2 := c_2 + c_3
38:                 end if
39:             end for
40:             if c_2 < c_0 then
41:                 c_0 := c_2
42:                 Let I be a vector of size 1
43:                 I' := I(μ_{r_2})
44:                 I(0) := I'(k_3) + ((e'_1, k_1), (e'_2, k_2))          ▷ List concatenation
45:                 I(μ_r, parent(μ_r)) := I
46:             end if
47:         end for
48:     end for
49:     return c_0
50: end function
```

Correctness

The algorithm distinguishes between three cases where two of them are special cases.

Case 1. The first special case occurs if $\mathcal{T}$ consists of only one node μ (cf. lines 5–7). Then the optimal solution is any embedding of $skeleton(\mu)$. If μ is of type S- or P, then all affected vertices are part of each face of any embedding of $skeleton(\mu)$ and ϑ and its incident edges can be inserted without crossings. This is obviously optimal. If μ is of type R, then $skeleton(\mu)$ has only two embeddings which are mirror images of each other. Thus it is does not matter which is chosen.

Case 2. The other special case occurs if there exists one node μ of $\mathcal{T}$ such that $skeleton(\mu)$ contains all affected vertices (cf. lines 10–16). Then it is optimal to insert ϑ into a face of any embedding Π of $pert(\mu)$ which corresponds to a face of the embedding of $skeleton(\mu)$ induced by Π. If μ is of type S or P, ϑ and its incident edges can be inserted into any embedding of $pert(\mu)$ without crossings. If μ is of type R, the correctness follows from the consideration below and from the property that $skeleton(\mu)$ has only two embeddings which are mirror images of each other.

Let Π be an embedding of $pert(\mu)$ representing an optimal solution. Further on let ϑ and its incident edges be inserted into Π with the minimum number of crossings whereby ϑ is inserted into a face of an embedding of $expansion(e)$ with $e = \{u, v\}$ as virtual edge of $skeleton(\mu)$. f'_r and f'_l denote the two faces in Π corresponding to $f_r(e)$ and $f_l(e)$ in the embedding of $skeleton(\mu)$ induced by Π. If $\{u, v\} = W$, then obviously ϑ can be reinserted into f'_r or f'_l and its incident edges can be reinserted without crossings. Hence assume $\{u, v\} \neq W$. Then inserted edges between ϑ and all $w \in W - \{u, v\}$ must cross f'_r and f'_l on their insertion paths. Let E_r and E_l be the distinct sets of these inserted edges according to this property. $E_r = \emptyset, E_l \neq \emptyset$ or $E_r \neq \emptyset, E_l = \emptyset$ cannot hold because ϑ could be reinserted into f'_r and f'_l respectively and all crossings with respect to $expansion(e)$ could be saved. Thus $E_r \neq \emptyset$ and $E_l \neq \emptyset$ must hold. Consider the case $|E_r| = |E_l|$. For one $e \in E_r$ and one $e' \in E_l$ with insertion paths $P = e_1, \ldots, e_k$ and $P' = e'_1, \ldots, e'_l$, respectively, there exist indices $k' \leq k, l' \leq l$ such that $e_{k'}$ and $e'_{l'}$ lie on the boundary of f'_r and f'_l, respectively. Then $|\{e_1, \ldots, e_{k'}, e'_1, \ldots, e'_{l'}\}| \geq c(e)$ holds. It follows that ϑ can be reinserted into f'_r without loss of generality while producing the same number of crossings. All $e \in E_r$ need not cross $expansion(e)$ anymore and the length of their insertion paths is reduced by $|E_r| \cdot |P|$. We let all $e' \in E_l$ now cross $expansion(e)$ by an insertion path of length $c(e)$. We easily see that the case $|E_r| \neq |E_l|$ cannot hold

because a solution with less crossings would be possible. Clearly, the described property is valid for each virtual edge of $skeleton(\mu)$.

Case 3. This is the general case. The insertion of ϑ into an embedding of G, especially an embedding representing an optimal solution, corresponds to the insertion of ϑ into an embedding of a skeleton associated with a node of $\mathcal{T}$ and vice versa. Let Π be an embedding of G. Then ϑ is inserted into a face f' of Π. According to Theorem 2.3 we can uniquely define an embedding of the skeleton of each node in $\mathcal{T}$ by means of Π. Then there must be a face f of the embedding of a skeleton such that $f = f'$ or the replacement of each virtual edge bordering f by an embedding of its expansion graph creates f'. The converse direction follows easily.

The algorithm simulates the insertion of ϑ into each face of an appropriate embedding of the skeleton associated with each node of $\mathcal{T}$ and therefore takes all possibilities into account. Thereby it is no restriction to make the current considered node μ_r the root of $\mathcal{T}$. The reason is that $\mathcal{T}$ is the only SPQR-tree of G and rerooting it causes only that the meaning of edges in the skeleton associated with each node on the bottom-up path from the old to the new root changes, i.e., especially the reference edge becomes another.

Whenever ϑ is regarded as inserted into a face f of an embedding of $skeleton(\mu_r)$ the subproblem which arises and has to be solved for each child ν_i of μ_r with $\mathcal{W}_i := W^-(pert(\nu_i)) \neq \emptyset$ is the core problem. Thereto consider the virtual edge e_i of ν_i in $skeleton(\mu_r)$. When e_i is replaced by an embedding of $expansion(e_i)$ and edges between ϑ and all $w \in \mathcal{W}_i$ are inserted into the so obtained embedding then $0 \leq j \leq |\mathcal{W}_i|$ and $|\mathcal{W}_i| - j$ of these edges must cross the face corresponding to $f_r(e_i)$ and $f_l(e_i)$ in the embedding of $skeleton(\mu_r)$, respectively. With a solution of the core problem with respect to ν_i at hand, not only the minimum number of crossings produced with respect to $expansion(e_i)$ but also a best embedding of $expansion(e_i)$ is provided for each value of j. We have shown in detail how to solve the core problem for an S-, R- or P-node to optimality. One can easily see that there is no need to create any embedding until not all nodes are processed.

Clearly, in the first iteration we have to solve the core problem in a bottom-up order for all nodes of the SPQR-tree except the root whose pertinent graphs contain affected vertices. But afterwards it suffices to do so for all the nodes on the bottom-up path from the old to the new root since only the meaning of edges of their skeletons changes and therewith their associated cost vector as well as the traversing costs of their virtual edges. Now the reason why all the associative data structures are indexed with the pair $\mu, parent(\mu)$ becomes obvious. It serves for the *memoization* technique and avoids the multiple computation of the cost vector and the related data structures under the same conditions.

It remains to explain that the choice of the embedding of the root's skeleton is correct. In the case μ_r is an S- or R-node an arbitrary embedding of $skeleton(\mu_r)$ can be chosen for the reasons mentioned several times so far. In the case μ_r is a P-node we apply the trick of splitting μ_r into two artificial P-nodes μ_{r_1}, μ_{r_2} (cf. CrossingNumberP) based on the simple observation that $skeleton(\mu_r)$ consisting of a bundle of n edges has $\mathcal{O}(n^2)$ different faces where ϑ can be "inserted" into despite of its $(n-1)!$ embeddings. Therewith the task to create an embedding of $skeleton(\mu_{r_1})$ with μ_{r_1} as temporary new root becomes trivial and the core problem has to be solved with respect to μ_{r_2}.

Running Time

At first we give the running times of the two functions. The running time of CrossingNumberSR is $\mathcal{O}(|skeleton(\mu_r)|^2 \cdot |W|)$. The dual and augmented dual graph in lines 4 and 6 can be computed in time $\mathcal{O}(|skeleton(\mu_r)|)$. The number of steps of the outermost loop can be estimated by $|skeleton(\mu_r)|$ as well as of the loop in line 22. As multiple times mentioned the computation of the lengths of the weighted shortest paths in lines 16 and 20 take also time $\mathcal{O}(|skeleton(\mu_r)|)$. The running time of the loop in line 25 can be estimated by $\mathcal{O}(|W|)$.

The running time of CrossingNumberP is dominated by the multiple calls of Algorithm 4.4 for μ_{r_2} with a worst-case running time of $\mathcal{O}(|skeleton(\mu_{r_2})| \cdot t_{max}^2 \cdot |W|^2)$. The computation of the dual graph and the length of the shortest paths are negligible since $skeleton(\mu_{r_2})$ consists of three edges only. In fact it is not necessary to compute the dual graph. We easily see that the block in lines 22–39 has a running time of $\mathcal{O}(|W|)$. $n \leq |skeleton(\mu_r)|$ is valid and the two nested loops in lines 8 and 9 cause $|skeleton(\mu_r)|^2$ computation steps. We obtain $\mathcal{O}(|skeleton(\mu_r)|^2 \cdot (|skeleton(\mu_{r_2})| \cdot t_{max}^2 \cdot |W|^2 + |W|)) = \mathcal{O}(|skeleton(\mu_r)|^3 \cdot t_{max}^2 \cdot |W|^2)$.

In the first iteration of the outer loop of the main algorithm the traversing costs of all virtual edges are computed and furthermore for each node of the SPQR-tree either Algorithm 4.3 or 4.4 is called within the inner loop. The computation of the traversing costs takes time $\mathcal{O}(|V|)$. Algorithm 4.3 and 4.4 have running times of $\mathcal{O}(|skeleton(\nu_i)| \cdot |W|^3)$ and $\mathcal{O}(|skeleton(\nu_i)| \cdot t_{max}^2 \cdot |W|^2)$ with respect to the considered node ν_i. Hence we can estimate this portion of the running time by

$$\mathcal{O}(|V|) + \sum_{\mu \text{ in } \mathcal{T}} \mathcal{O}(|skeleton(\mu)| \cdot t_{max}^2 \cdot |W|^2) = \mathcal{O}(|V| \cdot t_{max}^2 \cdot |W|^2)$$

In all the following iterations of the outer and inner loop together either Algorithm 4.3 or 4.4 is called at most $d(\mu)$ times for each node μ of the SPQR-tree due to the fact that each neighbour of μ becomes the root and due to the memoization technique. Therefore in total we can estimate at most twice as many calls of

Algorithm 4.3 and 4.4 as there are nodes in the SPQR-tree. Taking again the worse running time of both algorithms and summing up leads to $\mathcal{O}(|V| \cdot t_{max}^2 \cdot |W|^2)$. Traversing all the bottom-up paths and computing all the traversing costs can be estimated by $\mathcal{O}(|V|^2)$. The two functions are also called as many times as there are nodes in the SPQR-tree. This causes a running time of

$$\sum_{\mu \text{ in } \mathcal{T}} \mathcal{O}(|skeleton(\mu)|^3 \cdot t_{max}^2 \cdot |W|^2) = \mathcal{O}(|V|^3 \cdot t_{max}^2 \cdot |W|^2)$$

The associative data structures $\mathcal{I}$ and $\mathcal{M}$ have each size $\mathcal{O}(|V|)$. Hence making copies of them takes time $\mathcal{O}(|V|^2)$. The same running time is true for the final creation of the embedding. The SPQR-tree which is built at the beginning of the algorithm causes a linear running time in the number of vertices of the graph. All in all we obtain a worst-case running time of

$$\mathcal{O}(|V|^3 \cdot t_{max}^2 \cdot |W|^2)$$

The theorem below summarizes our results.

Theorem 4.1 *Let $G = (V, E)$ be a biconnected planar graph and $W \subseteq V$ be a non-empty vertex set. Then VIP-VAR can be solved optimally with respect to G and W in time $\mathcal{O}(|V|^3 \cdot t_{max}^2 \cdot |W|^2)$ ($t_{max} \leq |V|$).*

The given runtime is also valid when taking the runtime of $\mathcal{O}(|V| \cdot |W|)$ for inserting the elements (cf. Section 4.2) into the so-obtained embedding into account. Moreover we have to remark that the worst-case runtime improves to $\mathcal{O}(|V| \cdot |W|^3 + |V|^2 \cdot |W|) = \mathcal{O}(|V|^2 \cdot |W|^2)$ ($|W| \leq |V|$) if the SPQR-tree consists of S- and R-nodes only.

4.3.4 Solving VIP-VAR for Connected Graphs

In this section we present an algorithm solving VIP-VAR for a connected planar graph $G = (V, E)$ and $W \subseteq V$. Basically, this algorithm is based on two algorithms solving VIP-VAR with respect to a block of G and the BC-tree of G. This algorithm has polynomial worst-case running time.

Let $G' = (V', E')$ be a biconnected graph and $W' \subseteq V'$. By $\mathcal{A}_1$ we denote the algorithm solving VIP-VAR with respect to G' and W' and assume further that $\mathcal{A}_1$ can process weighted $w' \in W'$. That means when computing the length of a dual shortest (sub)path with a weighted w' as endpoint this length is always multiplied with the weight of w'. By $\mathcal{A}_2$ we denote the algorithm for solving VIP-VAR with respect to G', W' and another input $v' \in V'$. $\mathcal{A}_2$ can process also weighted $w' \in W'$ like $\mathcal{A}_1$ and additionally assumes that the new vertex

to be inserted into an embedding of G' has to be inserted into a face on whose boundary v' lies. Moreover we assume that both algorithms not only return the embedding of G' but also the face in which the new vertex has to be inserted and the number of crossings which would occur when inserting that vertex and its incident edges into the returned embedding. We can adapt the algorithm(s) presented in the previous section to realize $\mathcal{A}_1$ and $\mathcal{A}_2$. Basically, thereto it is necessary to maintain larger cost vectors (whose components may contain ∞) and to save the face which the insertion vertex is regarded as inserted into. We can provide the same worst-case runtime of $\mathcal{O}(|V'|^3 \cdot t_{max}^2 \cdot |W'|^2)$ if we make W' a multiset which contains an affected vertex with weight ω exactly ω times.

Let T be the rooted BC-tree of G. $x(B)$ and $x(c)$ denote the nodes of T associated with the block B and the cut-vertex c, respectively. If x is a node of T, then $H(x)$ stands for the subgraph of G which is induced by the vertices of G associated with all nodes of the subtree of T with root x.

Now let Π be an embedding of G which represents an optimal solution of VIP-VAR with respect to G and W. Moreover let $B_1, \ldots, B_{n\geq 1}$ be the blocks of G and ϑ and its incident edges be inserted into Π with the minimum number of crossings. Below, by means of Π we give some facts whose correctness is easy to comprehend and which induce an optimal algorithm:

(1) ϑ is inserted into a face f' of Π. So there must be a face f of an embedding Π_i of a block B_i $(1 \leq i \leq n)$ induced by Π such that $f' = f$ or the removals of all subgraphs of G except B_i connected only through the cut-vertices (see step 2) lying on the boundary of f' let f arise.

(2) If there are cut-vertices $c_1, \ldots, c_{j\geq 1}$ in B_i, then for $k = 1, \ldots, j$ there exists a connected subgraph $H_k := H(x(c_k))$ (assuming B_i is the root of T) connected with the rest of G only through c_k. Additionally, H_k is composed by distinct subgraphs $\bar{H}_1 := H(x(\bar{B}_1)), \ldots, \bar{H}_{l\geq 1} := H(x(\bar{B}_l))$ with $x(\bar{B}_1), \ldots, x(\bar{B}_l)$ as the children of $x(c_k)$. $\bar{H}_m$ for $m = 1, \ldots, l$ must be embedded into a face of Π_i on whose boundary c_k lies (cf. Figure 4.8).

(3) If $\bar{H}_m$ does not contain affected vertices excluding the possibly affected c_k, then $\bar{H}_m$ is not crossed by any inserted edge. Obviously, an inserted edge can cross the face which $\bar{H}_m$ is embedded into always without crossing $\bar{H}_m$ itself.

(4) If $\bar{H}_m$ contains affected vertices excluding the possibly affected c_k, then all inserted edges with endpoint in $\bar{H}_m$ must cross the face which $\bar{H}_m$ is embedded into.

(5) If $\bar{H}_1, \ldots, \bar{H}_l$ are not embedded into a common face on whose boundary c_k lies, these graphs and therewith H_k can be embedded into a common face

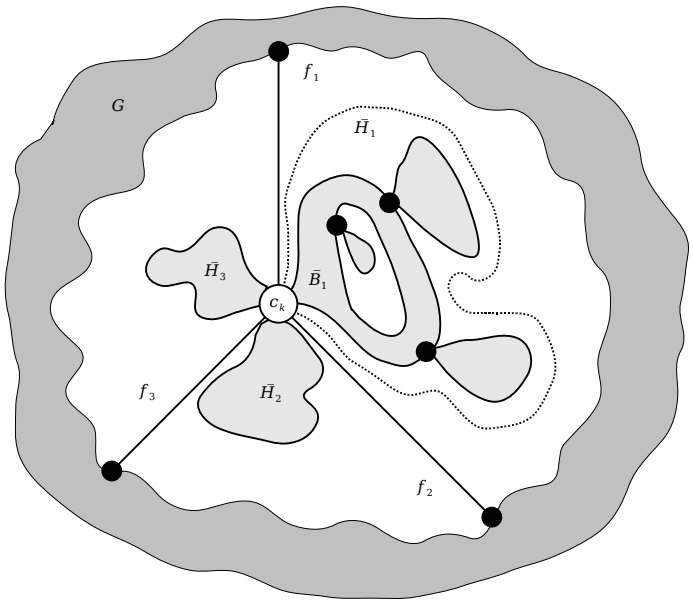


Figure 4.8: Distinct subgraphs $\bar{H}_1, \bar{H}_2$ and $\bar{H}_3$ sharing the common cut-vertex c_k.

f such that the total minimum number of crossings is preserved. f is the face on whose boundary c_k lies and that would be crossed when inserting an edge between ϑ and c_k with the minimum number of crossings.

(6) The embeddings of $B_1, \ldots, B_n$ can be fixed independently from each other and therewith the embedding of B_i can be fixed independently from the embedding of H_k and vice versa for all k.

Now we construct the algorithm based on these facts. According to fact 1 we assume for $i = 1, \ldots, n$ that the insertion of ϑ into an embedding of G relates to the insertion of ϑ into an embedding of B_i. According to fact 6 we compute an embedding Π_i of B_i and for all k an embedding Π_k of H_k separately and extend Π_i by Π_k appropriately, i.e., H_k is embedded according to Π_k into a certain face of Π_i on whose boundary c_k lies. If the costs of the resulting embedding, so the minimum number of crossings that would occur when inserting ϑ and its incident edges in fact, are less than the costs of the best solution so far, this embedding is saved. When all blocks are processed the best embedding is available.

Let us explain how to compute the embeddings. Thereto let B_i be the current block. Consider the case in which B_i contains all $w \in W$. Then it is not optimal to insert ϑ into a face of an embedding of H_k for all k. Illustratively, this can be reasoned as follows. According to fact 5 we assume H_k is embedded into a face

f of an embedding of B_i. According to fact 4 all inserted edges must cross f and produce crossings with respect to H_k. But at least these crossings can be saved if ϑ is reinserted into f. This holds for all k and it follows that ϑ can optimally be inserted into a face of an embedding of B_i. Therefore we can call Algorithm 4.6 with inputs B_i and W, obtain a best embedding Π_i of B_i and for all k we can embed H_k arbitrarily planar into a face of Π_i on whose boundary c_k lies.

Now consider the case in which B_i does not contain all $w \in W$. Then, without loss of generality, there is an index $1 \leq j' \leq j$ such that $H_{k'}$ for $k' = 1, \ldots, j'$ contains affected vertices excluding the possibly affected $c_{k'}$. According to fact 5, for all k' we can embed $H_{k'}$ into a face on whose boundary $c_{k'}$ lies. According to fact 4 all inserted edges with endpoints in $H_{k'}$ would cross this face. Therefore for all k' we can simply weight $c_{k'}$ with the number of affected vertices in $H_{k'}$ and make $c_{k'}$ an affected vertex. Then $\mathcal{A}_1$ can be called with respect to B_i and the affected vertices of B_i. Thus the number of crossings with respect to B_i is correctly paid attention to. Let the embedding Π_i of B_i and the face f of Π_i be the return values of $\mathcal{A}_1$. For all k' we compute the shortest path $P(f^*, c_{k'}) = f_0^*, \ldots, f_l^*, c_{k'}$ ($l \geq 0$, $f_0^* := f^*$) in $\Pi_i^*[\{c_k'\}]$, a best embedding $\Pi_{k'}$ of $H_{k'}$ (as will be shortly described) and extend Π_i by embedding $H_{k'}$ according to $\Pi_{k'}$ into f_l. According to fact 3, again we can embed H_k for $k = j' + 1, \ldots, j$ arbitrarily planar into a face of Π_i on whose boundary c_k lies.

The number of crossings with respect to B_i that would occur when inserting edges between ϑ and all affected vertices in $H_{k'}$ is already paid attention to. Since $H_{k'}$ will be embedded entirely into f_l it is easy to see that the problem to compute a best embedding of $H_{k'}$ is equivalent to solve VIP-VAR with respect to $H_{k'}$ and its affected vertices excluding the possibly affected $c_{k'}$ under an additional constraint. This constraint is that the vertex which is assumed to be inserted into an embedding of $H_{k'}$ has to be inserted into a face on whose boundary $c_{k'}$ lies. According to fact 2 we can solve this equivalent problem by solving it with respect to the distinct subgraphs of $H_{k'}$ that contain affected vertices. Without loss of generality let $1 \leq l' \leq l$ be the index such that $\bar{H}_{m'}$ for $m' = 1, \ldots, l'$ contains affected vertices. The embedding of $\bar{H}_{m'}$ could be computed recursively. According to the additional constraint and fact 1 the vertex assumed to be inserted into an embedding of $\bar{H}_{m'}$ is assumed to be inserted into an embedding of $\bar{B}_{m'}$ (e.g., cf. $\bar{B}_1$ in Figure 4.8). We recognize that facts 2–6 are valid for $\bar{B}_{m'}$ with respect to $\bar{H}_{m'}$. Hence we can compute the embedding of $\bar{H}_{m'}$ almost the same way as we do for G itself. That means according to fact 6 an embedding of $\bar{B}_{m'}$ and for each subgraph of $\bar{H}_{m'}$ connected with a cut-vertex of $\bar{B}_{m'}$ is computed and composed to an embedding of $\bar{H}_{m'}$. This time the computation of the embedding of a block is done by $\mathcal{A}_2$. Note, that it is essential to make the face of the embedding of the block, both returned by $\mathcal{A}_2$, the external face. Otherwise in a real insertion of the

elements into the obtained embedding of G there could be extra crossings. On the other hand it is no restriction since each block is biconnected and each face of its embedding can be made the external face while preserving its combinatorial embedding.

Algorithm 4.7 presents the algorithm as just now described more compactly and completely. It does not create embeddings recursively. Instead it processes a bottom-up traversal of T. Moreover we presume that $\mathcal{A}_2$ when called with respect to a block B, an empty set of affected vertices and a vertex v of B returns any planar embedding of B, any face of this embedding on whose boundary v lies and 0 as crossing number.

Algorithm 4.7: Computes an optimal solution of VIP-VAR for connected planar graphs.

Input: Connected planar graph $G = (V, E)$, $W \subseteq V$, $W \neq \emptyset$
Output: Embedding of G

1: Compute the BC-tree T of G
2: **if** T contains only one node **then** ▷ Special case 1
3: **return** the embedding obtained by calling Algorithm 4.6 with inputs G, W
4: **end if**
5: **for all** nodes x of T **do** ▷ Special case 2
6: **if** x is associated with block B and $V(B) \cap W = W$ **then**
7: Root T at x
8: Call Algorithm 4.6 with inputs B, W and obtain an embedding Π of B
9: Let $x(c_1), \ldots, x(c_{j \geq 1})$ be the children of x
10: **for** $k := 1, \ldots, j$ **do**
11: Extend Π by embedding $H(x(c_k))$ arbitrarily planar into a face of Π on whose boundary c_k lies
12: **end for**
13: **return** Π
14: **end if**
15: **end for**

16: $cr_{opt} := \infty$
17: **for all** nodes x of T **do**
18: **if** x is not associated with a block **then**
19: Continue
20: **end if**
21: $cr := 0$
22: Root T at x
23: $W' := \emptyset$ ▷ Set of artificially affected cut-vertices

24: $\Pi_{opt} := \emptyset$
25: **for all** nodes y of T in a bottom-up order **do**
26: **if** y is associated with the block B **then**
27: **if** $y = x$ **then**
28: Call $\mathcal{A}_1$ with inputs B, $V(B) \cap (W \cup W')$ and obtain an embedding Π of B, a face f of Π and a crossing number cr'
29: $cr := cr + cr'$
30: **else**
31: Call $\mathcal{A}_2$ with inputs B, $V(B) \cap (W \cup W')$, the cut-vertex associated with the parent of y and obtain an embedding Π of B, a face f of Π and a crossing number cr'
32: $cr := cr + cr'$
33: Make f the external face of Π
34: **end if**
35: **if** y has children $x(c_1), \ldots, x(c_{j \geq 1})$ **then**
36: Compute $\Pi^*[\{c_1, \ldots, c_j\}]$
37: **for** $k := 1, \ldots, j$ **do**
38: Compute $P(f^*, c_k) = f_0^*, \ldots, f_{l \geq 0}^*, c_k$ ($f_0^* := f^*$) in $\Pi^*[\{c_1, \ldots, c_j\}]$
39: Extend Π by embedding $H(x(c_k))$ according to $\Pi(x(c_k))$ into f_l
40: **end for**
41: $\Pi(y) := \Pi$
42: **end if**
43: **if** $y = x$ and $cr < cr_{opt}$ **then**
44: $\Pi_{opt} := \Pi$
45: $cr_{opt} := cr$
46: **end if**
47: **else if** y is associated with the cut-vertex c **then**
48: Let $\Pi := \emptyset$ be the empty embedding
49: Let $z_1, \ldots, z_{j \geq 1}$ be the children of y
50: **for** $k := 1, \ldots, j$ **do**
51: Extend Π by embedding $H(z_k)$ according to $\Pi(z_k)$ into the external face of Π
52: **end for**
53: $\Pi(y) := \Pi$
54: **if** $w := |W \cap V(H(y))| > 0$ **then**
55: Weight c with w
56: $W' := W' \cup \{c\}$
57: **end if**
58: **end if**
59: **end for**
60: **end for**
61: **return** Π_{opt}

In the following we verify the polynomial running time of our algorithm in the known manner by analyzing portions of its running time. We assume an adjacency list representation of the (combinatorial) embeddings using doubly linked lists.

We can compute the BC-tree of the graph in line 1 in linear time by a modified depth-first search [32]. The size of the BC-tree is linear in the number of the graph's vertices. This relies on the facts that the graph is planar and contains at most as many blocks as there are edges in it and, clearly, at most as many cut-vertices as there are vertices in it.

If the first special case in line 2 applies, Algorithm 4.6 is called which has a runtime of $\mathcal{O}(|V|^3 \cdot t_{max}^2 \cdot |W|^2)$. If the second case is true, Algorithm 4.6 is called as well, this time with respect to B and thus causing a runtime of $\mathcal{O}(|V(B)|^3 \cdot t_{max}^2 \cdot |W|^2) = \mathcal{O}(|V|^3 \cdot t_{max}^2 \cdot |W|^2)$. Any planar embedding of the subgraph in line 11 can be computed in linear time (see, e.g.,[26]) after its construction. There are as many subgraphs as B owns cut-vertices and each subgraph has a size less than $|V|$. Thus computing the embeddings of all those subgraphs can be estimated with $\mathcal{O}(|V|^2)$. Due to the adjacency list representation, the current embedding in line 11 can be extended by only little pointer rearrangement (list insertion and concatenation). Hence, the runtime of the algorithm in those special cases is $\mathcal{O}(|V|^3 \cdot t_{max}^2 \cdot |W|^2)$.

The code-block in lines 47–58 with respect to the loop in line 25 has a runtime of $\mathcal{O}(|V|)$. Each node associated with a block and cut-vertex is considered once. Again, because of the adjacency list representation, the construction of the embedding described in line 53 can be done with only little pointer rearrangement. We can further assume that the assignment of the embedding in line 53 is realized with a pointer assignment and thus causes constant runtime.

The augmented dual graph computation in line 36 can be realized in time $\mathcal{O}(|V(B)|)$. All shortest paths can be computed with one BFS in the augmented dual graph and need not to be computed iteratively as suggested in lines 37 and 38. This takes time $\mathcal{O}(|V(B)|)$ again. The construction of the embedding in line 39 can be estimated by constant time as already reasoned above. Paying attention to the loop in line 25, the operations in lines 35–46 obviously lead to a runtime portion of $\mathcal{O}(|V|)$.

Now let t_{max} be the maximum thickness associated with a P-node's skeleton among all P-nodes and $\omega(w)$ the weight of an affected vertex. The runtime of $\mathcal{A}_1$

as well as $\mathcal{A}_2$ with respect to B is then:

$$\begin{aligned}
& \mathcal{O}(|V(B)|^3 \cdot t_{max}^2 \cdot |\underbrace{V(B) \cap (W \cup W')}_{\text{becomes a multiset}}|^2) \\
= \; & \mathcal{O}(|V(B)|^3 \cdot t_{max}^2 \cdot (|\underbrace{V(B) \cap W)}_{\substack{\text{unweighted} \\ \text{vertices}}}| + |\underbrace{V(B) \cap W'}_{\substack{\text{weighted cut-} \\ \text{vertices of } B}}|)^2) \\
= \; & \mathcal{O}(|V(B)|^3 \cdot t_{max}^2 \cdot (\underbrace{|V(B) \cap W)|}_{\leq |W|} + \underbrace{\sum_{w \in V(B) \cap W'} \omega(w)}_{\leq |W|})^2) \\
= \; & \mathcal{O}(|V(B)|^3 \cdot t_{max}^2 \cdot |W|^2)
\end{aligned}$$

Over all iterations of the loop in line 25, we therefore obtain a runtime of

$$\mathcal{O}(|V|^3 \cdot t_{max}^2 \cdot |W|^2)$$

for the code-block in lines 27–34. We see that this is also the runtime of the code-block in lines 25–59.

It remains only to take the main loop in line 17 into account. This leads to the following total runtime with b as the number of the blocks of G:

$$\mathcal{O}(b \cdot |V|^3 \cdot t_{max}^2 \cdot |W|^2)$$

We conclude the chapter with the following theorem.

Theorem 4.2 *Let $G = (V, E)$ be a connected planar graph and $W \subseteq V$ be a non-empty vertex set. Then VIP-VAR can be solved optimally with respect to G and W in time $\mathcal{O}(b \cdot |V|^3 \cdot t_{max}^2 \cdot |W|^2)$ (b, $t_{max} \leq |V|$).*

5 Summary and Outlook

In this thesis we have dealt with the two variants VIP-FIX and VIP-VAR of the *Vertex Insertion Problem*. We could successfully develop polynomial time algorithms for both problems although especially VIP-VAR is formulated over the set of all combinatorial embeddings of a graph and therefore has a possibly exponential solution space. At the beginning, it was not clear at all if VIP-VAR is possibly an $\mathcal{NP}$-hard problem.

In the worst-case, the algorithm presented for VIP-FIX has quadratic runtime in the number of vertices of the given graph. In the case the number of affected vertices is small, it has even a linear runtime. We have achieved this runtime due to the usage of BFS for shortest path computations in the augmented dual graph in a non-naive strategy. Further on we have provided an approach that defines the order of inserting the new edges. We have seen that the order is essential for inserting all edges as desired.

In our consideration of VIP-VAR, firstly we have presented an algorithm for computing a lower bound of the costs (number of crossings) in an optimal solution of this problem. This result is rather enriching from theoretical point of view since we are able to solve VIP-VAR optimally. However, the interesting is that it uses the algorithm for the related *Edge Insertion Problem* (cf. Chapter 3, [19]) and the concept of maximum weight matchings.

Afterwards we have developed an algorithm based on SPQR-trees for solving VIP-VAR for biconnected graphs. The algorithm has polynomial worst-case runtime and it depends on the number of affected vertices and the maximum thickness of a skeleton associated with a P-node. If these values are negligible in real-world input instances, the algorithm runs in a worst-case time cubic in the number of vertices. If additionally P-nodes are absent the runtime improves to a worst-case runtime quadratic in the number of vertices. The special about the algorithm is its doubly dynamic programming approach and that it predominantly performs a pure cost computation before an embedding is constructed.

Subsequently, we have presented a polynomial time algorithm for solving VIP-VAR for connected graphs. The just mentioned remarks according the runtime of the algorithm for VIP-VAR for biconnected graphs also hold for this algorithm if additionally the number of the graph's blocks is sufficiently small. The algorithm reduces the problem by solving VIP-VAR with respect to each block of the connected graph in a certain manner. In addition it makes use of the BC-tree

data structure for managing the blocks and cut-vertices.

For various reasons it is advisable to implement all those algorithms. Therewith, first and foremost, an evaluation of their performance with respect to real-world input instances would be possible. As stated above, the worst-case runtime of both algorithms for VIP-VAR obliques a few values. So these factors need not to be that much significant in practice.

Of course it is also of high interest to apply these algorithms within the modified planarization method for comparing the quality of drawings against such ones created with the usual planarization method. As mentioned in Chapter 4, the benefit of the modified approach is that it reinserts a certain set of edges in each vertex reinsertion step at once. Hence it might produce drawings with less crossings overall. Within the modified planarization method it would also be interesting to vary the vertex reinsertion step. For example, the order of the vertices to be reinserted could be varied or each time that vertex with lowest degree among all vertices not yet reinserted could be reinserted first. Further on, an interesting question is if our approach can improve the quality of drawings when applying it as a post-processing strategy after the usual planarization method.

When implementing the modified heuristic there must be also a choice of "good" heuristics to compute a planar subgraph by deleting a preferably small set of vertices. We have proposed some simple concepts. However the more successful algorithm is probably the one given in [14] since it guarantees a certain size of the vertex induced subgraph. Of course, another interesting question is how the runtime between the modified and standard planarization approach differs.

In summary we can emphasize that the presented algorithms open up the possibility of an interesting modified planarization method and post-processing strategy within the usual planarization method. However their usefulness in practice concerning the quality of generated drawings and running times is outstanding and has to be proved.

Bibliography

[1] C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity relationship diagrams. *Journal of Systems and Software*, 4:163–173, 1984.

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

[3] 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.

[4] T. Biedl. Graph-theoretic algorithms. Lecture notes of a graduate course, University of Waterloo, `http://www.student.cs.uwaterloo.ca/~cs762/Notes/`, 2005.

[5] J.M. Boyer and W. Myrvold. Simplified O(n) planarity algorithms. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[6] F.J. Brandenburg, M. Jünger, and P. Mutzel. Algorithmen zum automatischen Zeichnen von Graphen. *Informatik Spektrum*, 20(4):199–207, 1997.

[7] C. Buchheim, D. Ebner, C. Gutwenger, M. Jünger, and P. Mutzel. Crossings and planarization. In *Handbook of Graph Drawing and Visualization*, chapter 4. 2006. To appear.

[8] C. Buchheim, D. Ebner, M. Jünger, G.W. Klau, P. Mutzel, and R. Weiskircher. Exact crossing minimization. In *Graph Drawing*, volume 3843 of *LNCS*, pages 37–48. Springer-Verlag, 2005.

[9] Gutwenger C. and Mutzel P. An experimental study of crossing minimization heuristics. In *Graph Drawing*, volume 2912 of *LNCS*, pages 13–24. Springer-Verlag, 2003.

[10] M. Chimani, C. Gutwenger, and P. Mutzel. Experiments on exact crossing minimization using column generation. In *Experimental Algorithms*, volume 4007 of *LNCS*, pages 303–315. Springer-Verlag, 2006.

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

[12] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal of Computing*, 25(5):956–997, 1996.

[13] R. Diestel. *Graphentheorie*. Springer-Verlag, 2006.

[14] K. Edwards and G. Farr. An algorithm for finding large induced planar subgraphs. In *Graph Drawing*, volume 2265 of *LNCS*, pages 75–83. Springer-Verlag, 2002.

[15] R. Fleischer and C. Hirsch. Graph drawing and its applications. In Kaufmann and Wagner [24], pages 1–22.

[16] H. Gabow. *Implementation of Algorithms for Maximum Matching on non-bipartite graphs*. PhD thesis, Stanford University, 1974.

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

[18] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In *Graph Drawing*, volume 1984 of *LNCS*, pages 77–90. Springer-Verlag, 2001.

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

[20] F. Harary. *Graphentheorie*. Oldenbourg Verlag, 1974.

[21] P. Hlineny. Crossing number is hard for cubic graphs. In *Mathematical Foundations of Computer Science 2004*, volume 3153 of *LNCS*, pages 772–782. Springer-Verlag, 2004.

[22] P. Hlineny and G. Salazar. On the crossing number of almost planar graphs. In *Graph Drawing*, volume 4372 of *LNCS*, pages 162–173. Springer-Verlag, 2007.

[23] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, pages 135–158, 1973.

[24] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *LNCS*. Springer-Verlag, 2001.

[25] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20:219–230, 1980.

[26] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–244, 1996.

[27] P. Mutzel. Zeichnen von Diagrammen - Theorie und Praxis. Research report, Max-Planck-Institut für Informatik, 1997.

[28] P. Mutzel. The SPQR-tree data structure in graph drawing. In *Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 34–46. Springer-Verlag, 2003.

[29] H. Purchase. Which aesthetic has the greatest effect on human understanding? In *Graph Drawing*, LNCS, pages 248–261. Springer-Verlag, 1997.

[30] H. C. Purchase, R. F. Cohen, and M. I. James. An experimental study of the basis for graph drawing algorithms. *Journal of Experimental Algorithmics*, 2, 1997.

[31] R. Tamassia, G.D. Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18:61–79, 1988.

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

[33] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[34] R. Weiskircher. Drawing planar graphs. In Kaufmann and Wagner [24], pages 23–45.

[35] R. Weiskircher. *New Applications of SPQR-Trees in Graph Drawing*. PhD thesis, Universität des Saarlandes, 2002.

[36] T. Ziegler. *Crossing Minimization in Automatic Graph Drawing*. PhD thesis, Max-Planck-Institut für Informatik, 2001.