Algorithms for Drawing Planar Graphs

Algoritmen voor het Tekenen van Planaire Grafen

(met een samenvatting in het Nederlands)

Proefschrift
ter verkrijging van de graad van doctor
aan de Rijksuniversiteit te Utrecht
op gezag van de Rector Magnificus, Prof. Dr J.A. van Ginkel,
ingevolge het besluit van het College van Dekanen
in het openbaar te verdedigen
op maandag 14 juni 1993 des namiddags te 2.30 uur

door

Goossen Kant

geboren op 3 januari 1967
te Rijswijk (N.Br.)

Promotor: Prof. Dr J. van Leeuwen
Co-promotor: Dr H.L. Bodlaender

Faculteit Wiskunde en Informatica

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Kant, Goossen

Algorithms for drawing planar graphs / Goossen Kant. -
Utrecht : Universiteit Utrecht, Faculteit Wiskunde en
Informatica
Proefschrift Rijksuniversiteit Utrecht. - Met samenvatting
in het Nederlands.
ISBN 90-393-0416-5
Trefw.: algoritmen / grafentheorie.

This research was supported by the ESPRIT Basic Research Actions of the EC under contract No. 3075 (project ALCOM) and contract No. 7141 (Project ALCOM II).

Preface

Many times it was pretty hard for me to explain to my family and friends what the precise contents is of the work of a PhD student. Moreover, in general it becomes even more difficult to explain the research problems we try to solve in theoretical computer science. I often started these discussions by complaining of the readability of electrical diagrams, which you receive when buying a television. This thesis gives more insight in the broad area of automatically drawing networks or graphs. Many involved problems and different kinds of representations are discussed. I hope that you will enjoy this research field in graph algorithms as much as I do. I also hope that you will detect (after inspecting the different delivered drawing results) that this drawing topic is not so theoretical at all.

It is a pleasure to thank the people who supported me during the preparation of this thesis. Especially I want to thank my supervisor Jan van Leeuwen for introducing me in this beautiful and fascinating field of graph drawing. I really appreciate the various discussions we had (including computer science) and his trust in me after I found again another error in this – and other – work. Jan van Leeuwen involved me in organizing various events and gave me a lot of opportunities to travel to workshops and conferences in foreign parts of this world. In these four PhD years, I have been to a huge number of major European cities, and also to Ottawa, Montreal, New York, Pittsburgh, and even to the magnific island Barbados. I also like to thank Hans Bodlaender for giving a lot of helpful comments. He also delivered considerable contributions to the research that is reported in this thesis, and was co-author of some corresponding reports.

Several persons commented on previous versions of parts of this thesis. In particular, I want to thank Therese Biedl, Marek Chrobak, Xin He, Tsan-sheng Hsu, Siebren van de Kooij, Klaus Kriegel, Petra Mutzel, and Martin van Trigt and the anonymous referees of conferences and journals. Many results in this thesis have profited from their suggestions and remarks. I like to thank the members of the review-committee, Giuseppe Di Battista, Kees Hoede, Jan Karel Lenstra, Mark Overmars and Roberto Tamassia for their careful proofreading of this thesis. I appreciate the help of Piet van Oostrum, Maarten Pennings, Otfried Schwartzkopf and Nico Verwer, who answered all my questions about LaTeX and related issues. But also thanks to all other collegues. The Vakgroep is not only a stimulating environment for doing research, it is also a pleasant place to work.

I also want to thank my family and friends for their moral support, and being there when I needed them. In particular, I want to thank my father and mother for taking care of me, and stimulating me the many days I worked at home. To them I dedicate this thesis. But most important, I like to thank the almighty God. He gave me joy, happiness, strength, health, intelligence, and so many other blessings all the days of my life.

Algorithms
for
Drawing Planar Graphs

voor mijn vader en moeder

Gebruik de tijd om te werken – het is de prijs voor succes.
Gebruik de tijd om te denken – het is de bron van kracht.
Gebruik de tijd om te spelen – het is het geheim van de eeuwige jeugd.
Gebruik de tijd om te lezen – het is de fontein der wijsheid.
Gebruik de tijd om vriendelijk te zijn – het is de weg naar geluk.
Gebruik de tijd om te dromen – zo worden idealen geboren.
Gebruik de tijd om te beminnen en bemind te worden – het is het voorrecht van verloste mensen
Gebruik de tijd om rond te kijken – de dag is te kort voor zelfzuchtigheid.
Gebruik de tijd om te lachen – het is de muziek van de ziel.
Gebruik de tijd voor God – het is de enige duurzame belegging in het leven.

Uit: "Planning voor alledag", een Telos-uitgave

Contents

Part A

Introduction

Chapter 1

Drawing Planar Graphs

In many applications graphical representations are used for displaying information. The function of these representations is to clarify or to display the structure of the information in a compact and relatively small space. Many times one picture says more than thousand words, but the picture has to be clear and readable. Almost everybody is aware of schemas, using rectangles with information, and lines and arrows connecting them. Just think about the schematic representation of the organizational structure of a company. Or consider all relations and links in a database or other huge software program, which must be shown in a convenient way. Also a plan for a project has to show clearly the underlying relationships, e.g., which parts of the project should be done at the same time or consecutively. Representing all this information in a schematic diagram helps to manage the project. In many contexts the design of such a graphical representation is supported by a computer, i.e., the computer is used to calculate the coordinates of the different objects. What kind of problems can occur?

If a schema is small, then it can easily be drawn by hand. The problem becomes more and more difficult if one tries to draw the electrical diagrams of electrical applications in a readable form. The electrical schema of e.g. a television set gives a readable idea how the different components in the television are connected with each other. Needless to say, the actual placement inside the application is completely different. This is a real-life example that shows how computers are used to compute a drawing of a huge network.

On the other hand, a computer is also used to calculate the optimal placement of the components inside the electrical application. The application contains a numerous amount of small electrical components, which must be connected with each other. These components have to be placed on a *chip*, such that the number of crossings between the connections is as small as possible, and the required area of the chip must not become too large. To compute an optimal placement of the components on a chip by hand requires an incredible amount of time. The problem becomes even more complex when several additional constraints have to be satisfied as well, e.g., the number of bends and the total length of the connections must

be minimized as well. These questions arise in the design of Very Large Scale Integration (VLSI) chips.

In a more mathematical abstract setting, the components are called *vertices* or *nodes* and the connections between the components are called *edges*. A *graph* is a set of vertices and a set of edges between the vertices. Many real-life examples with objects and connections (or relations) between the various objects can be represented by a graph.

This thesis is devoted to research in automatic drawings and graphical representation of graphs. The examples mentioned above give a good insight in the questions and optimization criteria, involved in the methods, or *algorithms*, to layout any given graph. However, these optimization criteria are not always well defined. The aesthetic criteria of "readability" or "a nice drawing" cannot always be expressed clearly in mathematical formulas. One mathematical optimization criterion can be a good choice for one structure but can lead to an unattractive drawing in other cases. Many times a good drawing fulfills a combination of optimization criteria. Nevertheless, we will assume that if a graph can be drawn without any pair of crossing edges, then we draw it accordingly. The graphs, which can be drawn without any pair of crossing edges, are called *planar graphs*. As announced in the title, we focus our attention on planar graphs in this thesis. This problem is not new: the *drawing problem* is a classical theme in the context of planar graphs. Nowadays, an abundant amount of research and literature is reported all over the world. See the recent annotated bibliography of Di Battista, Eades & Tamassia [18] for an overview.

Several algorithms are constructed to test whether a graph is planar or not (see e.g. [9, 50]). Even for planar graphs, various relevant additional constraints are developed and shown to be useful in the applications. The following are just some of the major criteria mentioned in the ever growing collection of relevant literature in the subject of drawing (planar) graphs.

- Minimizing the total number of bends in the edges (or draw the graph with no bends at all, i.e., straight-line edges)
- Minimizing the total used area
- Placing the vertices and bends on grid coordinates
- Maximizing the minimum angles between consecutive edges
- Maximizing the minimum distance between the vertices

In the area of VLSI design several additional constraints are given, e.g., separating the graph in circuits layers, via placement, and typical network models. See the book of Lengauer ([77]) for a more detailed overview of the questions and problems appearing in this specific area.

In some applications a direction is given to each edge. For this special case more specific constraints are developed, e.g., the endpoint of the edge must be

placed higher than the beginpoint. Drawings satisfying this constraint are called *upward drawings*. Recently several (graph-theoretic) papers appeared in this area [4, 5, 20, 22]. In this thesis we focus our attention on undirected planar graphs. Let us distinguish in more detail the different classes of undirected planar graphs and the corresponding representation models, known for this model. The relationships between the different classes of planar graphs and the corresponding drawing algorithms are a key in this thesis. We consider the following four different classes: (i) trees, (ii) biconnected planar graphs, (iii) triconnected planar graphs, and (iv), triangulated planar graphs. Here we give a brief overview; in Chapter 9 several of these algorithms are outlined in more detail, which are relevant for our research.

First, when the graph is a tree, then in most drawings the vertices are placed along horizontal lines according to their level (distance from the root), and a minimum separation distance between two consecutive vertices on the same level such that the width of the drawing is small. In Figure 1.1 we give some output drawings of existing algorithms. For tree drawing algorithms the reader is referred to [40, 93, 101, 112, 120].

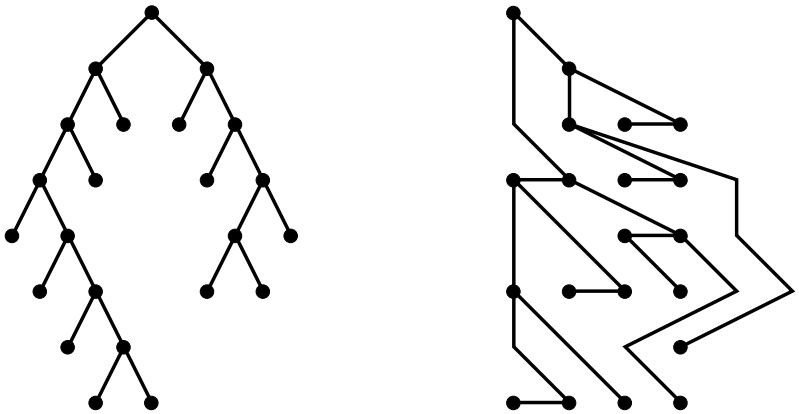
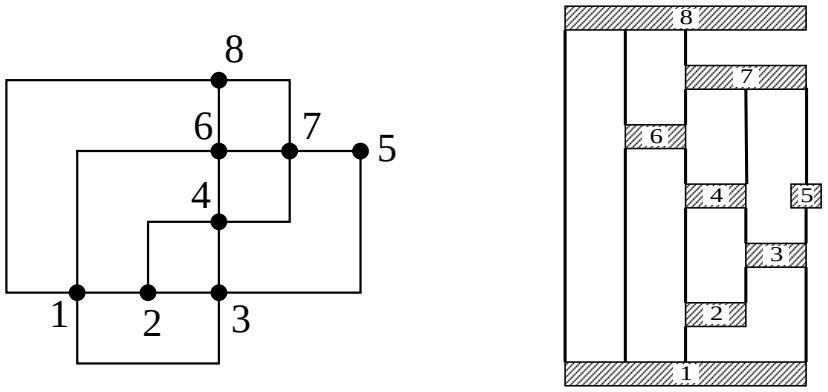


Figure 1.1: Output of tree-drawing algorithms (see [93, 40]).

In the case of biconnected planar graphs two important representation models are presented. The first one is the *orthogonal drawing*, in which the vertices are represented by points and edges by alternatingly horizontal and vertical segments, connecting the endpoints. Such a representation is only possible when every vertex has at most 4 incident edges. The model of orthogonal drawings has important applications in VLSI-design and in drawings of electrical diagrams. Therefore research with special attention to grid size and number of bends gains a lot of interest from both the theoretical and practical point of view. Storer [100], Tamassia [102] and Tamassia & Tollis [105] presented the major orthogonal drawing algorithms. In Figure 1.2(a) an example is given. The second model is the *visibility representation*. Vertices are represented by horizontal segments and edges by vertical segments, only

touching the two segments, representing the two endpoints. An important advantage of this model is that relevant additional information, assigned to the vertices, can be placed inside the horizontal segments (or rectangles). Moreover, since the edges are straight-line vertical segments, this yields very readable and practical pictures in general, appearing in a broad context of applications. In Figure 1.2(b) an idea of this representation is given, as outlined by Rosenstiehl & Tarjan [96] and Tamassia & Tollis [104].



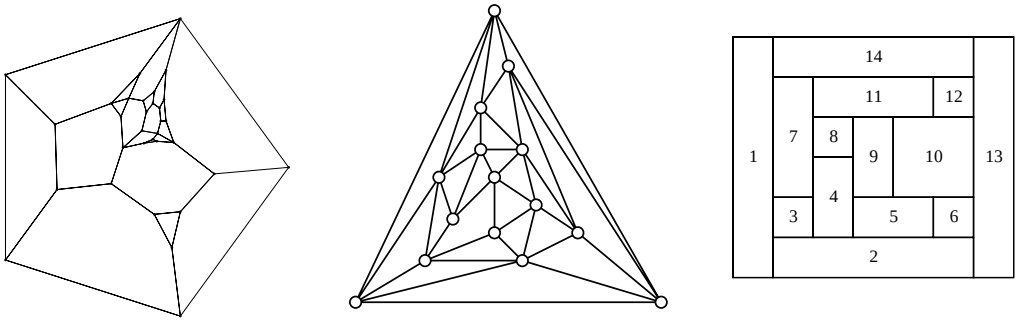
(a) Orthogonal drawing. (b) Visibility representation.

Figure 1.2: Output of biconnected planar graph drawing algorithms

A third important class is the class of triconnected planar graphs. Most algorithms in this thesis require a triconnected planar graph as input. If a planar graph is triconnected, then a drawing is possible with vertices represented as points and edges as straight lines such that every face is drawn as a convex polygon. This so-called *convex drawing* is an important representation in graph theory. Wagner [113], Fáry [31] and Stein [99] independently proved that every planar graph can be drawn with vertices represented as points and edges as straight lines. (Sometimes this is also called a *Fáry drawing*.) Tutte [110] and Thomassen [109] considered the graph-theoretic backgrounds of convex drawings. Tutte [111] presented an algorithm for drawing triconnected planar graphs convexly. The algorithm of Chiba et al. [14] draws a planar graph with convex faces if this is possible.

When every face of the graph is a triangle, then the graph is called a *triangular planar graph*. This forms the last class we distinguish. Recently, several algorithms are presented to represent a graph as a straight-line drawing, requiring a triangular planar graph as input. We mention here the work of Chrobak & Payne [15], de Fraysseix, Pach & Pollack [34], Van Haandel [43], Read [92] and Schnyder [98]. The algorithms, described in [15, 34, 43, 98], places the vertices on an $O(n) \times O(n)$ grid. Since all coordinates are integers, we can draw the resulting picture in precisely the same way on a screen or paper, where the resolution is fixed. If the triangular planar

graph is 4-connected, then we can represent every vertex v of G as a rectangle $R(v)$, such that every edge (u, v) in G corresponds to a common boundary of $R(u)$ and $R(v)$. This is called a *rectangular dual*. In Figure 1.3 an idea of the mentioned representations is given.



(a) Convex drawing (b) Straight-line drawing (c) Rectangular dual

Figure 1.3: Different types of drawing.

However, what to do when a given graph does not fulfill these additional connectivity requirements? How can we still use the drawing algorithm we want to apply? The typical solution we present here is to add edges to the graph such that the augmented graph is still planar and satisfies the connectivity property. We apply the drawing algorithm on the augmented graph and compute the coordinates of the vertex and edge representations. Finally we draw the graph, but the added edges are suppressed. Corresponding to the distinguished classes of planar graphs, we have the planar augmentation problem with respect to biconnectivity, triconnectivity and triangular planar graphs.

We show that the problem of deciding whether adding at most K edges to a graph such that the resulting graph is planar and biconnected is NP-complete. We present an $O(n \cdot \alpha(n, n))$ time algorithm for the planar biconnectivity augmentation algorithm, adding at most two times the minimum number of edges. We also present an $O(n)$ time algorithm for the planar triconnectivity augmentation algorithm, adding at most $\frac{3}{2}$ times the minimum number of edges. Triangulating a planar graph G can be done in linear time. A linear time triangulation algorithm is presented, with the property that the maximum degree of the triangulation is at most $\frac{3}{2}$ time the maximum degree of the input graph. It is shown that the problem of deciding whether G can be triangulated such that the maximum degree of the triangulation is at most K is NP-complete. Graph augmentation algorithms receive more and more attention in the recent literature, but these algorithms do not deal with the additional planarity constraint.

The thesis is divided into three parts.

Part A, of which this introduction is the first chapter, introduces the reader to the area of planar graphs. We briefly outline the techniques of testing whether a graph is planar, and if so, how to embed a planar graph. In case the graph is not planar, we explain how to delete a small number of edges such that the graph is planar. Moreover a technique is introduced for splitting a planar graph into its bi- and triconnected components, and how to compute a special ordering (called *canonical ordering*) on the vertices of a planar graph. The last two techniques will play a major role in Part B and C.

Part B concerns itself with the problem of augmenting planar graphs. In particular, we consider the problem of adding a minimum number of edges such that a planar graph is biconnected or triconnected, and still planar. These augmentation problems turn out to be very hard to solve in polynomial time. Therefore approximation algorithms are presented, adding only a constant times the minimum number of edges. We also consider the problem of adding edges such that we obtain a triangular planar graph, while minimizing the maximum degree. This problem appears in the area of constructing a straight-line drawing. In the special case that the input graph is outerplanar, i.e., all vertices share one face, then all augmentation problems mentioned above can be solved in polynomial time. Chapter 7 is devoted to the case of outerplanar graphs.

Part C concerns the major theme of this thesis, namely, the drawing of planar graphs. This part requires no background of Part B and, hence, can be read completely independent of Part B. We start in Part C by giving a more detailed survey of the existing drawing algorithms, which are relevant for our work. We also introduce a new ordering on the vertices and faces of a triconnected planar graph, called the *lmc-ordering*. This ordering leads to a new drawing framework for many representation models. The most important drawing results are the following (in all cases the planar graph is drawn without crossing edges):

1. Every triconnected planar graph can be drawn with straight-line edges on a grid of size at most $(n-2) \times (n-2)$ such that every interior face is convex (Chapter 10).

2. Every planar graph can be drawn on a grid of size at most $(2n-6) \times (3n-6)$ with at most $5n-15$ bends and minimum angle at least $\frac{2}{3d+1}$, such that every edge has at most 3 bends and length at most $2n$ (Chapter 10).

3. Every triconnected planar graph with degree at most 4 can be drawn orthogonally on a grid of size at most $n \times n$ with at most $\lceil \frac{3}{2}n \rceil + 4$ bends such that every edge has at most 2 bends if $n > 6$ (Chapter 11).

4. Every planar graph with degree at most 3 can be drawn orthogonally on a grid of size at most $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$ with at most $\lfloor \frac{n}{2} \rfloor + 1$ bends, with the property that there is a spanning tree of $n-1$ straight-line edges, while all non-tree edges have at most one bend if $n > 4$ (Chapter 11).

5. Every planar graph with degree at most 3 can be drawn with straight-line edges and vertices represented as points, such that the minimum angle between any two consecutive edge is at least $\frac{\pi}{4}$ if the graph is triconnected, and at least $\frac{\pi}{3}$, otherwise (Chapter 12).

6. There is a simple linear time algorithm for constructing a rectangular dual of a 4-connected triangular planar graph (Chapter 13).

7. A visibility representation of a planar graph can be constructed on a grid of size at most $(n-1) \times (n-1)$, if the graph is 4-connected, and on a grid of size at most $(\lfloor \frac{3}{2}n \rfloor - 3) \times (n-1)$, otherwise (Chapter 13).

For almost all cases, the given bounds improve previous bounds, known in the literature, and sometimes match existing lower bounds. The outlined algorithms are quite easy to implement. A discussion of the advantages and disadvantages of the representation models is included as well. Moreover, results, based on experiments of implementations of several described algorithms, are given in Chapter 14. Part C ends with an evaluation of these graph drawings and some conclusions.

Chapter 2

Backgrounds

This chapter gives the basic definitions and offers a brief description of a number of techniques that have been developed in the theory of planar graphs. We also describe a number of the major algorithms that are used in this thesis. More precisely, an outline is given of the methods for testing planarity, embedding planar graphs, planarization of graphs, splitting a graph into bi- and triconnected components, and a special ordering on the vertices and faces of a triconnected planar graph. The aim is not to give a complete survey of these techniques in their most general and sophisticated form; it is rather meant as a starting point for our work. In particular, we assume for many algorithms described in Part B and C, that a planar embedding of a (planar) graph is given along with a description of the bi- and triconnected components.

If the reader wants to know more, he/she should follow the pointers to the literature that are given. We also mention the work of Even [28] and Nishizeki & Chiba [86]. Both books deliver a broad spectrum of techniques with respect to planarity, embeddings and planar graphs.

2.1 Terminology

In this section we give a first introduction to the terminology of graphs, which will be used throughout the thesis. More explicit definitions with respect to graph theory, graph augmentation and graph layout are given in the corresponding chapters. Using these introduction a more detailed overview is given of testing the planarity, embedding of planar graphs and splitting the graph into subgraphs.

Definition 2.1.1 *A graph is a structure $G = (V, E)$ in which V is a finite set of vertices and $E \subseteq V \times V$ is a finite set of edges (unordered pairs). Given a graph G, $|V|$ is denoted by n and $|E|$ by m.*

Two vertices of a graph G are called *adjacent* if there is an edge with these vertices as end vertices. We define the degree of vertex v, denoted by $deg(v)$, to be the number

of incident edges of v. We define $\Delta(G) = \max\{deg(v)|v \in G\}$. A *path* between two vertices x and y is an alternating sequence of vertices and edges such that x and y are at the end of this sequence and each edge in the sequence is preceded and followed its end vertices. More precisely, $x = x_0, e_1, x_1, e_2, \ldots, e_k, x_k = y$ is a path between x and y, if $e_i = (x_{i-1}, x_i) \in G$ (for $1 \leq i \leq k$). If $x = y$ and $k > 0$, then there is a path with the same begin- and endpoint. Such a path is called a *cycle*. If the vertices on the path have degree 2, then the path is called a *chain*. Two vertices are called *connected* if there exists a path between them. A graph is called *connected* if there is a path between every pair of vertices, otherwise the graph is called *disconnected*. A cycle of G consisting of 3 edges is called a *triangle*. Let $G - \{v\}$ denote the graph after deleting vertex v with all its incident edges. If $G - \{v\}$ is disconnected, then v is called a *cutvertex* of G, and each component of $G - \{v\}$ is called a *v-block*. If (x, y) is an edge such that $(V, E - \{(x, y)\})$ is disconnected, then (x, y) is called a *bridge* of G. Let us consider the connectivity aspects of graphs and the involved

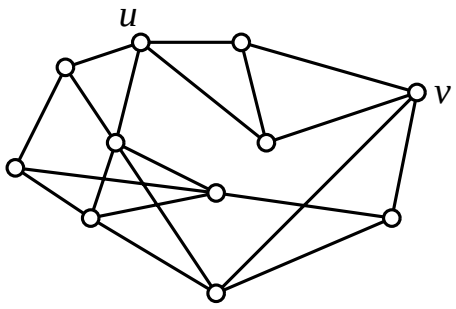


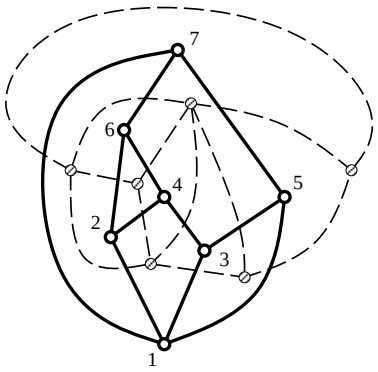
Figure 2.1: A nonplanar biconnected graph with separation pair $\{u, v\}$.

definitions in more detail. If G is connected and contains no cutvertices, then G is called *biconnected* or *2-connected*. If G is connected and contains no bridges, then G is called *bridge-connected* or *2-edge-connected*. In general, G is *k-connected* if there is no set of $k - 1$ vertices, whose removal disconnects G. Such a set is called a separating $(k - 1)$ set. By Menger's theorem, G is k-connected if there exist k vertex-disjoint paths between any two vertices of G. A 2-set of vertices is called a *separation pair* or *cutting pair*. A maximal biconnected subgraph is called a *biconnected component* or *block*. In Section 2.4 a detailed discussion is given on the algorithmic aspects of splitting a graph into subgraphs, which are biconnected or triconnected. This splitting plays a major role in the augmentation algorithms in Part B of this thesis.

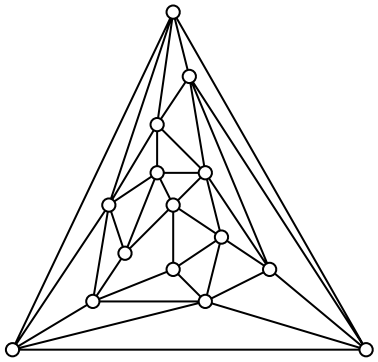
In this thesis we only consider planar graphs in detail, which are defined as follows:

Definition 2.1.2 *A graph is called* planar *if it can be drawn in the plane such that there is no pair of crossing edges.*

A *planar embedding* is a representation of a planar graph in which at every vertex all edges are sorted in clockwise order when visiting them around the vertex with respect to the planar drawing. A graph with a given, fixed planar embedding is also called a *plane graph*. A *face* of a plane graph is any topologically connected region surrounded by edges of the plane graph. The one unbounded face of a plane graph is called the *outerface* or *exterior face*. All other faces are called *interior faces*. Edges and vertices, belonging to the outerface, are called *exterior edges* and *exterior vertices*, respectively. The other edges and vertices are called *interior edges* and *interior vertices*. An interior edge, connecting two exterior vertices, is called a *chord*. If every face in G is a triangle, then G is called a *triangulated, triangular* or *maximal planar graph*.



(a) A triconnected planar graph and its dual.



(b) A triangulated planar graph.

Figure 2.2: Different types of graphs.

The dual graph G^* of G is defined as follows: every vertex v_{F_k} of G^* represents a face F_k of G. All faces of G are represented in this way. There is an edge (v_{F_k}, v_{F_j}) in G^* if F_k and F_j have an edge in common in G. Hence $deg(v_{F_k})$ in G^* is equal to the number of edges belonging to F_k in G. By Euler's formula: $m - n - f + 2 = 0$ for every planar graph, where f is the number of faces. It implies that $m \leq 3n - 6$. Using this formula and the definition of the dual graph, more observations can be made. For the number of vertices n_{G^*} of the dual graph G^* of G, $n_{G^*} = m - n + 2$ holds. It also easily follows that every planar graph has a vertex v with $deg(v) \leq 5$. Deleting the neighbors of v disconnects G, hence every planar graph is at most 5-connected. We call a graph G *k-planar*, if G is planar and the maximum degree of any vertex in G is at most k. An embedded k-planar graph is called a *k-plane graph*. A triangulated planar graph has exactly $3n-6$ edges. It is also called *maximal planar* since adding any edge to it destroys the planarity. Every triangulated planar graph is triconnected [41]. A triconnected planar graph G has the important property of

having a unique embedding, i.e., in any planar embedding of G, the edges around each vertex have the same order (up to reversing all adjacency lists) [12].

In this thesis we also consider two important subclasses of planar graphs, namely, the *outerplanar graphs* and *trees*. A planar graph is called outerplanar if it can be drawn as a planar graph with all vertices occurring on one face, the outerface. A graph is outerplanar if and only if its blocks are outerplanar. A block of an outerplanar graph essentially is a cycle with non-intersecting chords. If the outerplanar graph G is biconnected and every interior face is a triangle, then G is called a *maximal outerplanar graph* or *mop*. Adding any edge to a mop destroys the outerplanarity. An outerplanar graph has at most $2n-3$ edges, and a mop has exactly $2n-3$ edges. A tree is an undirected, connected, acyclic graph. A tree is outerplanar. Consider the dual graph G^* of an outerplanar graph G. Let v_{out} be this vertex of G^*, representing the outerface of G. It follows that $G^* - \{v_{out}\}$ is a tree with maximum degree 3.

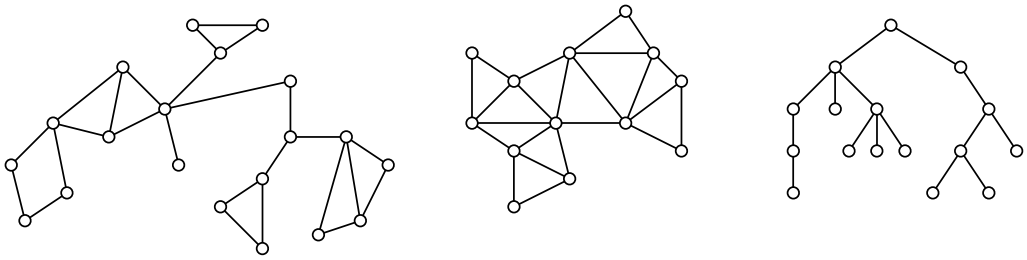


Figure 2.3: An outerplanar graph, a mop and a tree.

Throughout the thesis a graph $G = (V, E)$ is represented as a collection of adjacency lists. For each $v \in V$, the adjacency list $adj(v)$ contains (pointers to) all the vertices u for which there is an edge $(u, v) \in E$. The record containing u in $adj(v)$ contains a *crosspointer* to the record containing v in $adj(u)$. For testing planarity we assume that the vertices in each adjacency list are stored in arbitrary order. The sum of the lengths of all the adjacency lists is $2m$, since for every edge (u, v), u appears in v's adjacency list and vice versa. Hence the adjacency-list representation has the desirable property that the amount of memory it requires is $O(m+n)$ [1]. However, in the augmentation and drawing algorithms we assume that a planar embedding is given, i.e., the neighbors of v are stored in $adj(v)$ in clockwise order when visiting them around v with respect to a planar drawing. This has the nice property that for every edge (u, v), we can find in $O(1)$ time the "next" and "previous" edges of u and v in the planar embedding.

Constructing the adjacency lists $adj(v)$ including crosspointers of a given planar embedding of G can be done on-line as follows in $O(m)$ time. Assume that the

[1] We assume that the reader is familiar with the O-, Ω-, and Θ-notation, see e.g. [17]

input-format is as follows:

- On the first line n, the number of nodes.
- On the next n lines we have on line i ($1 \leq i \leq n$) the neighbors of vertex v_i in clockwise order.

The adjacency lists $adj(v_i)$ are represented as queues. For each vertex we also introduce a list $loweradj(v_i)$, containing pointers to the place of vertex v_i in $adj(v_j)$, with $j < i$. We also introduce an array *bucket*, for finding an element of $loweradj(v_i)$ in $O(1)$ time in step i. An important fact is that when the neighbors of vertex v_i are read, we already have $adj(v_j)$ for the vertices $v_j, j < i$. The algorithm becomes as follows:

```
MakeGraph(G);
  Readln(n);
  for i := 1 to n do initialize loweradj(v_i) and adj(v_i) to ∅ rof;
  for i := 1 to n do
    for all elements x in loweradj(v_i) do
      let x be a crosspointer to v_i in adj(v_j); bucket[j] := x
    rof;
    for every neighbor v_j of v_i, read in order from input do
      Enqueue(v_j, adj(v_i));
      if j > i then Enqueue(v_i, loweradj(v_j)) with pointer to adj(v_i)
      else place crosspointer's between adj(v_i) and bucket[j]
    rof
  rof;
End MakeGraph
```

An idea of this algorithm is the following: before the neighbors of v_i are read, the elements of $loweradj(v_i)$ are put in *bucket*. More precisely, if $bucket[j] = x$, then x is a pointer to record v_i in $adj(v_j)$ with $j < i$. When neighbor v_j of v_i is read from the input, then v_j is added to $adj(v_i)$. If $j < i$ then (cross)pointers between $adj(v_i)$ and x are added, with $bucket[j] = x$; if $j > i$ then v_i is added to $loweradj(v_j)$.

In several planar algorithms it is also necessary to have pointers from the vertices and edges to the faces they belong to. Given a planar embedding, e.g. as constructed by MakeGraph, it is rather easy to deliver this. We start by visiting vertex v_1 and process the faces incident to v_1. For each such face F with boundary vertices $v_1 = u_1, u_2, \ldots, u_p$ (in clockwise order around F) we do: we introduce a record for face F and an *edge-list*, containing the edges (u_i, u_{i+1}) $(1 \leq i < p)$ and (u_p, u_1). We also mark vertex u_{i+1} in $adj(u_i)$ (for $1 \leq i < p$) and u_1 in $adj(u_p)$ as being visited, and set pointers from them to the record of F. We continue by visiting all vertices $v_1, \ldots, v_n$. If some vertex v_j in $adj(v_i)$ is not marked visited, then from (v_i, v_j) we traverse in clockwise order a face, not visited yet. Since every face is traversed once, since every record in every adjacency list is marked visited only once, this yields a linear time algorithm.

2.2 Testing and Embedding Planar Graphs

2.2.1 Introduction

A major theme in graph theory is the study of planar graphs. Before we consider the problem of drawing a planar layout of a graph, the question arises how one can actually determine whether a given graph is planar or not. This classical problem in graph theory has a fundamental answer in the form of Kuratowski's Theorem [74]: a graph G is planar if and only if it has no subgraph "homeomorphic" to $K_{3,3}$ or K_5. ($K_{3,3}$ is the complete bipartite graph on 2 sets of 3 vertices and K_5 is the complete graph on 5 vertices, see Figure 2.4.) This characterization seems far from a

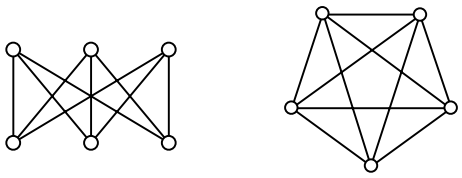


Figure 2.4: Forbidden homeomorphic subgraphs of planar graphs.

feasible computational recipe for testing planarity, and a different approach is called for. A simple observation shows that we can actually restrict the planarity test and, later, the design of a suitable drawing algorithm to the biconnected components of graphs: a graph is planar if and only if its biconnected components are. (This follows because the biconnected components of a graph can intersect in at most one vertex.) Hopcroft & Tarjan [107, 49] proved that the connected, biconnected and triconnected components of a graph can be determined by an algorithm that operates in $O(m+n)$ time on a graph (see also Section 2.4).

Nowadays, several algorithms are known for testing the planarity of graphs, based on one of two global techniques, namely the "edge addition" method and the "vertex addition" method. These terms refer to the principles used in the algorithms. The edge addition algorithm is originally due to Auslander & Parter [2] and a linear time implementation was developed by Hopcroft & Tarjan [50]. After this first approach, several related and simplified versions appeared. We mention here the work of Williamson [121], the algorithm of de Fraysseix & Rosenstiehl [35], and the recent algorithm by Hsu & Shih [54], which seems to be a very simple and fast method to test whether a graph is planar or not.

2.2.2 Testing Planarity Using PQ-trees

In this section we outline the vertex addition algorithm as presented by Lempel, Even & Cederbaum [76], and improved later to a linear time algorithm by Booth &

Lueker [9], using a novel data structure called the *PQ-tree*. Except the leaves, the PQ-tree consists of two types of nodes: The *P-nodes*, representing the cutvertices of the graph, and the *Q-nodes*, representing the blocks of the graph. There is an edge between a P-node and a Q-node, if the corresponding cutvertex belongs to that block. We will verify later that the order of children at each node is a core item in this data structure. (There is a high correspondence between the PQ-tree and the BC-tree, described in Section 2.4.) We maintain the children in a doubly linked list. The left- and rightmost child of a Q-node, and all children of a P-node, have a pointer to their parent. Every node has a pointer to its left- and rightmost child. Using this algorithm it is relatively easy to compute a planar embedding for a planar graph. There is a good related planarization algorithm, i.e., an algorithm to delete a small number of edges from a non-planar graph to obtain planarity. Both features of this algorithm are outlined in Section 2.2.3, and 2.3.2, 2.3.3, respectively.

We henceforth assume that G is biconnected. The planarity testing algorithm of Lempel, Even & Cederbaum first labels in linear time the vertices of G as $v_1, v_2, \ldots, v_n$, using what is called an *st-numbering* [30]. An st-numbering numbers the vertices of G such that $(v_1, v_n) \in E$ and every vertex v_i $(1 < i < n)$ has edges to some vertices v_k and v_l with $k < i < l$. Planar graphs that are numbered or directed in this way are called *planar st-graphs*. Planar st-graphs have many properties which have shown to be useful for planar graph drawing algorithms. An overview of these properties and the nice consequences are enumerated in Chapter 9.

Let $G_k = (V_k, E_k)$ be the subgraph of G induced on the vertices $v_1, v_2, \ldots, v_k$, i.e., $V_k = \{v_1, \ldots, v_k\}$, and $(v_i, v_j) \in E_k$, if $i \leq k$ and $j \leq k$. If $k < n$ then there exists an edge of G with one endpoint in V_k and the other in $V - V_k$. Let G'_k be the graph formed by adding to G_k all these edges (v_i, v_j), with $i \leq k$ and $j > k$. Any edge (v_i, v_j) of this kind is called a *virtual edge*, and v_j is called a *virtual vertex*. The virtual vertices are kept separate, i.e., there may be several virtual vertices with label (v_{k+1}), each with exactly one entering edge from V_k. Let B_k (the *bush form*) be an embedding of G'_k such that all the virtual vertices are placed on the outerface. Lempel, Even & Cederbaum [76] showed that an st-graph G is planar if and only if for every B_k, $2 \leq k \leq n-2$, there exists a planar drawing B'_k isomorphic to B_k such that in B'_k all virtual vertices labeled (v_{k+1}) appear consecutively.

The *PQ-tree* T_k corresponding to the bush form B_k consists of three types of vertices: (i) *Leaves* in T_k represent virtual edges (v_i, v_j) in B_k, with $i \leq k < j$, and are labeled with (v_j); (ii) *P-nodes* in T_k represent cutvertices in B_k, and (iii) *Q-nodes* of T_k represent the blocks in B_k. G, G_k, B_k and the corresponding PQ-tree are illustrated in Figure 2.5. (In this drawing a P-node is denoted by a circle, a Q-node is denoted by a rectangle.)

A few definitions are now in order. A node x in T_k is said to be *full* if all its descendant leaves are labeled (v_{k+1}); x is said to be *empty* if none of its descendant leaves is labeled (v_{k+1}); otherwise x is *partial*. If x is full or partial, then x is called a *pertinent node*. A pertinent leaf with label (v_{k+1}) is always full. The *frontier of* T_k

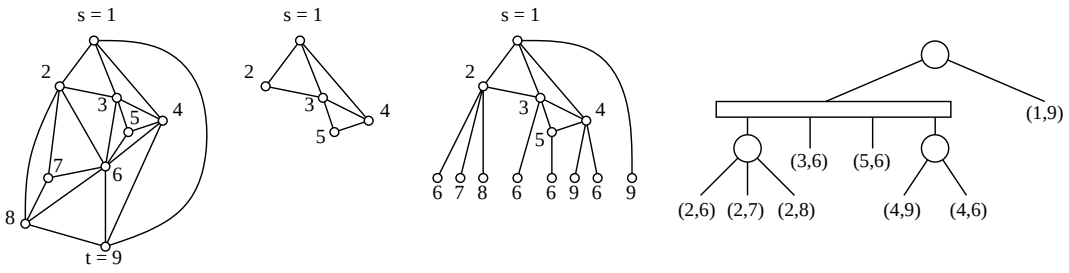


Figure 2.5: Example of G, G_k, B_k and corresponding PQ-tree (from [12]).

is the sequence of all the descendant leaves of T_k read from left to right. Similarly, the *frontier of a node* x is the sequence of all descendant leaves of x read from left to right. The *pertinent subtree* of T_k is the smallest connected subtree which contains all leaves with label (v_{k+1}). The root of the pertinent subtree, say x_r, is called the *pertinent root*. x_r is the least common ancestor of all leaves with label (v_{k+1}). Two PQ-trees are considered to be *equivalent* if one can be obtained from the other by performing one or more of the following types of operations.

- Reversing the order of the children of a Q-node.
- Permuting the children of a P-node.

The underlying idea is that all v-blocks, connected at a cutvertex v, can be arbitrarily permuted without destroying the planarity. Hence the children of a P-node can be arbitrarily permuted. In the same way every block can be reversed with respect to a cutvertex, hence the children of a Q-node can be reversed. An important operation during the algorithm is that in every step the P- and Q-nodes of degree 2 are removed from the PQ-tree, while connecting their neighbors.

It is shown in [9] that B'_k exists if and only if T_k can be converted into an equivalent PQ-tree T'_k such that all leaves with label (v_{k+1}) (corresponding to the incoming edges of v_{k+1}) appear consecutively in the frontier of T'_k. This follows precisely the intuition of planarity, i.e., all edges of v_{k+1} must end in a single endpoint, v_{k+1}, without losing planarity. Booth & Lueker have defined a set of patterns and replacements by means of which T'_k can be reduced into a PQ-tree T^*_k in which all full nodes appear as one consecutive sequence of children of a single node. To construct T_{k+1} from T_k, they first reduce T_k to T^*_k and then replace all full nodes by a P-node, whose children are all leaves, corresponding to the outgoing edges of v_{k+1}. The algorithm of Booth & Lueker starts with T_1 and constructs the sequence of PQ-trees $T_1, T_2, \ldots$. If the graph G is planar, then the algorithm terminates after constructing T_{n-1}; otherwise it terminates after detecting the impossibility of reducing some T_k into T^*_k. The crucial result in the complexity analysis of this algorithm is stated in the following theorem [9]:

Theorem 2.2.1 *The sum of the sizes of all the pertinent nodes in the PQ-trees $T_1, T_2, \ldots, T_{n-1}$ when the algorithm is run on an arbitrary graph is $O(m+n)$.*

2.2.3 Constructing Planar Embeddings Using PQ-trees

Testing the planarity of a planar graph and constructing a planar embedding seems to imply more difficulties than expected at first sight. In particular, modifying the "edge-addition" planarity testing algorithm of Hopcroft & Tarjan [50] such that it outputs a planar embedding seems to be "fairly complicated", according to Chiba et al. [12]. Nevertheless, several years after the publication of Hopcroft & Tarjan [50], embedding algorithms based on this planarity testing algorithm have been presented, independently by Mutzel [84] and Cai, Han & Tarjan [10].

In this section we describe briefly a simple modification of the Booth & Lueker planarity testing algorithm to obtain a planar embedding algorithm for planar graphs, as described by Chiba et al. [12]. They (and also, independently, Rosenstiehl & Tarjan [96] and Tamassia & Tollis [104]) observed the following interesting characteristic of planar st-graphs.

Lemma 2.2.2 *Consider an embedding of a planar graph G obtained by the Booth & Lueker algorithm. Let v_i be a vertex of G. All neighbors v_j with $j < i$ appear consecutively around v as do all the neighbors v_j with $j > i$.*

The embedding algorithm consists of two stages: (i) constructing an upward embedding of the upward graph G_{up} of G, and (ii), constructing the entire embedding. G_{up} is the graph G in which every edge (v_i, v_j) is directed $v_i \rightarrow v_j$, iff $i < j$. Let the adjacency lists $adj_{up}(v_i)$ store G_{up}. The head v_i appears in $adj_{up}(v_j)$, but the tail v_j does not appear in $adj_{up}(v_i)$ for every directed edge (v_i, v_j). When G is planar, then we can reduce the tree T_k in step k of the Booth & Lueker algorithm to a tree T_k^* in which all the leaves, labeled (v_{k+1}), appear as children of a single node, say node x_k. $adj_{up}(v_{k+1})$ is obtained by scanning the leaves labeled v_{k+1} from left to right (or vice versa) in T_k^*.

If $adj_{up}(v_{k+1})$ is correctly determined in step k then, by counting the number of subsequent reversions of node x_k, one can correct the direction of $adj_{up}(v_{k+1})$ by reversing $adj_{up}(v_{k+1})$ if the number is odd. The algorithm of Chiba et al. [12] does not determine the direction of $adj_{up}(v_{k+1})$ in T_k, but adds a new special node as one of x_k's children to the PQ-tree at an arbitrary position. The new node is called a *direction indicator*, also labeled v_{k+1}, and depicted by a triangle, as illustrated in Figure 2.6. The direction indicator v_{k+1} plays two roles. The first is to trace the subsequent reversions of $adj_{up}(v_{k+1})$. The indicator will be reversed with each reversion of x_k. The second is to transfer the relative direction of node v_{k+1} to its siblings.

When computing the maximal consecutive sequence of pertinent leaves, the presence of direction indicators is ignored. In the vertex addition step of step k we

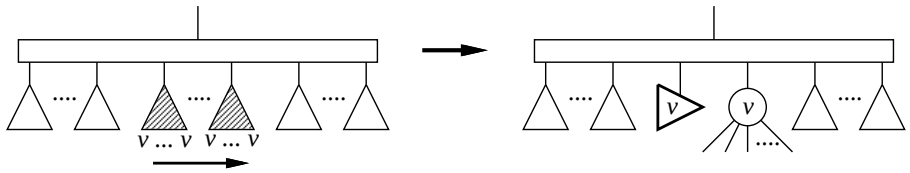


Figure 2.6: Adding a direction indicator in the PQ-tree (from [12]).

traverse the sequence with pertinent leaves and direction indicators. We store the contents of the leaves (which are edges (v_i, v_{k+1}) with $i < k+1$) and the direction indicators in $adj(v_{k+1})$. After step $n-1$ the adjacency lists are scanned in reverse order. When we visit the direction indicator of vertex v_i in $adj(v_j)$ (with $j > i$) in a direction opposite to the direction indicated by the triangle, then $adj(v_i)$ is reversed. The algorithm UPWARDEMBED outlines these ideas.

```
UPWARDEMBED(G);
  assign st-numbers to all the vertices of G;
  construct the initial PQ-tree T_1;
  for k := 1 to n − 1 do
    { reduction step }
      construct T_k^* from T_k by applying the template matchings
      to the PQ-tree, ignoring the direction indicators in it, such that
      the leaves labeled (v_{k+1}) occupy consecutive positions;
    { vertex addition step }
      let l_1, l_2, ..., l_i be the leaves labeled (v_{k+1}) and direction
      indicators scanned in this order;
      delete l_1, l_2, ..., l_i from the PQ-tree and store the contents in adj_up(v_{k+1});
      if the pertinent root r is not full then
        add an indicator (v_{k+1}) directed from l_1 to l_i to the PQ-tree as a child of r;
      replace all full nodes by a new P-node, with all outgoing edges
      of vertex v_{k+1} appearing as children of the new P-node
  rof;

  {correction step}
  for k := n downto 1 do
    for each element x in adj_up(v_k) do
      if x is a direction indicator then
        if direction of x is opposite to that of adj_up(v_k) then reverse adj_up(x);
        delete x from adj_up(v_k)
    rof
  rof;
END UPWARDEMBED
```

Finally the upward embedding $adj_{up}(v)$ is extended to a complete embedding. This is obtained by a simple depth-first search from v_n, and adding w to the top of $adj_{up}(v)$ when directed edge (w, v) is visited. Due to the characteristics of the st-numbering it can be proved [12] that this leads to a correct planar embedding. Let all vertices be marked *new* and all adjacency lists $adj_{up}(v)$ be copied to $adj(v)$, then the embedding is completed by calling DFS(v_n).

```
DFS(w)
  mark vertex w old;
  for each v in adj_up(w) do
    add vertex w to the top of adj(v);
    if v is marked new then DFS(v)
  rof
END DFS
```

Theorem 2.2.3 ([12]) *There is a linear time and space algorithm to test whether a graph is planar, and if so, it outputs a planar embedding.*

2.3 Planarization of Graphs

2.3.1 Introduction

If the graph G is nonplanar, then G can only be drawn in the plane with crossings. One reasonable objective for a readable representation is to look for a drawing which minimizes the number of crossings. Unfortunately, deciding whether a graph can be drawn with at most K crossings is NP-complete (the CROSSING NUMBER PROBLEM, see [59]). Another strategy for drawing G is to delete some edges from it such that it becomes planar, draw the resulting graph, and add the deleted edges again in the drawing. Unfortunately again, deciding whether the deletion of at most K edges makes the graph planar is NP-complete (the PLANAR SUBGRAPH PROBLEM, see problem [GT27] in [38]). On the other hand, making the graph planar by deleting edges yields an approximation algorithm for the crossing number problem. Therefore, the interest for computing planar subgraphs of a graph is growing, with more and more efficient and sophisticated algorithms documented in the recent literature. However, there exist graphs with n vertices and $\Theta(\sqrt{n})$ crossings, which become planar after deleting already one edge. *Graph planarization* defines the process of deleting (a small number of) edges to obtain a planar subgraph.

Several *heuristic algorithms* have been described for planarization. Several of these algorithms have been developed from a practical point of view. They contain no theoretical time and performance bounds, and the results are based on experiments. As an illustration of a simple and good heuristic, we outline the approach of

Goldschmidt & Takvorian [41], which consists of two phases: (i) devise an ordering of the set of vertices of G, $v_1, v_2, \ldots, v_n$, and draw them in this order on a horizontal line; (ii) try to draw a maximum number of edges in E above or below the line such that no two edges intersect. More precisely, partition the edges of G into three sets E_1, E_2 and E_3 in such a way that $|E_1| + |E_2|$ is maximum and that, given any four vertices with $v_{i_1} < v_{i_2} < v_{i_3} < v_{i_4}$, there are no two edges (v_{i_1}, v_{i_3}) and (v_{i_2}, v_{i_4}) both in E_1 or in E_2. Clearly the subgraph consisting of the edges in $E_1 \cup E_2$ is planar. For Phase (i) they use an approximation algorithm for finding an Hamiltonian circuit. For Phase (ii) they introduce a new graph H. Every edge (v_i, v_j) of G is represented by a vertex in H. There is an edge between two vertices in H iff for the corresponding edges $(v_i, v_j), (v_k, v_l)$ $i < k < j < l$ holds. Constructing E_1 and E_2 is achieved by computing independent sets. Computing a maximum independent set in H can be achieved in polynomial time, and the corresponding edges in G yields a planar subgraph. See [41] for more details and Figure 2.7 for an idea of the algorithm.

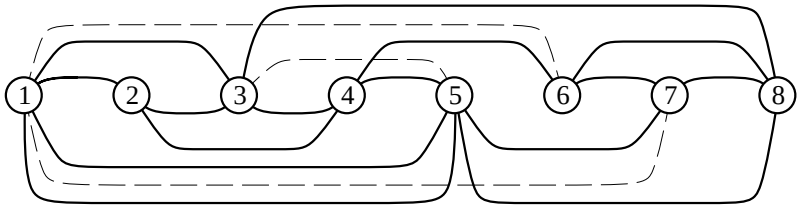


Figure 2.7: A simple heuristic for planarization of graphs.

Since the maximum planar subgraph problem is NP-complete, the current research emphasizes the problem of computing a *maximal planar subgraph*. A maximal planar subgraph $G_p = (V, E_p)$ of $G = (V, E)$ is a planar subgraph with the property that adding any edge $e \in G - G_p$ to G_p destroys the planarity. A first attempt to compute G_p is by incremental planarity testing: start with $E_p = \emptyset$, and test for every edge e whether $G_p \cup e$ is still planar. If so, add e to G_p. Continue until no further edge can be added. Using any of the prescribed planarity testing algorithms this leads to an $O(mn)$ time maximal planarization algorithm. Di Battista & Tamassia [21] developed an algorithm that tests in $O(\log n)$ time worst-case whether e can be added to G_p such that the resulting graph is planar. Adding e to G_p and updating the data structure that they need requires $O(\log n)$ time amortized. This leads to an $O(m \log n)$ time maximal planarization algorithm.

Very recently, La Poutré [75] presented a new data structure for maintaining planar graphs, in which one can test in $O(\alpha(m, n))$ time worst-case whether an edge e can be added to a planar graph G with m edges and n vertices while preserving planarity. $\alpha(m, n)$ is the functional inverse of Ackermann's function, which is no larger than 4 in all practical situations. The time of adding an edge e to G is $O(\alpha(n, n))$, amortized over $O(n)$ edges. This leads to an $O(m \cdot \alpha(m, n))$ maxi-

mal planarization algorithm. (This also improves an independently obtained result by Westbrook [119].) We also mention the work of Cai, Han & Tarjan [10], who presented an $O(m \log n)$ maximal planarization algorithm, based on the planarity testing algorithm of Hopcroft & Tarjan, but not using the incremental planarity testing approach.

In this thesis we focus attention on the planarization algorithm due to Ozawa & Takahashi [90], and described in more detail by Jayakumar et al. [58]. The algorithm is based on Booth & Lueker's planarity testing algorithm. The underlying argument for studying this framework is the following: in every step a next vertex v_i is added, and we determine the *minimum* number of edges $(v_j, v_i), j < i$, whose deletion yields a planar graph on $v_1, \ldots, v_i$. After deleting these edges in every step we obtain a planar subgraph G_p. The hope is that G_p contains more edges than the planar subgraphs obtained by incremental planarity testing, as described by Di Battista [21] and La Poutré [75]. We implemented the planarization algorithm of Jayakumar et al. [58], and compared the delivered planar subgraphs with the planar subgraph, obtained by incremental planarity testing. Indeed, for graphs which are "almost planar", we observed that the "vertex addition" method of Jayakumar et al. outputs denser planar subgraphs than the "edge addition" method. The experimental comparisons are summarized in Figure 2.10. However, the resulting planar subgraph G_p of the "vertex addition" method is not necessarily planar, as wrongly announced in [90]. In Section 2.3.3 we give a brief description how to obtain a maximal planar subgraph G'_p, containing G_p.

2.3.2 Planarization Using PQ-trees

In this section we discuss the basic principle of an approach for planarization, due to Ozawa & Takahashi [90] and also studied by Jayakumar et al. [56, 58]. Following these papers, we classify the nodes of a PQ-tree as follows:

Type W: A node is said to by type W, if its frontier consists of only empty leaves.

Type B: A node is said to be type B, if its frontier consists of only full leaves.

Type H: A node X is said to be type H if the subtree rooted at x can be arranged such that all the descendant pertinent leaves of x appear consecutively either at the left end or at the right end of the frontier.

Type A: A node x is said to be type A if the subtree rooted at x can be arranged such that all the descendant pertinent leaves of x appear consecutively in the middle of the frontier with at least one non-pertinent leaf appearing at each end of the frontier.

The central concept of the planarization algorithm is stated in the following theorem of [58], which essentially is a restatement of the principle on which the Booth & Lueker's planarity testing algorithm is based.

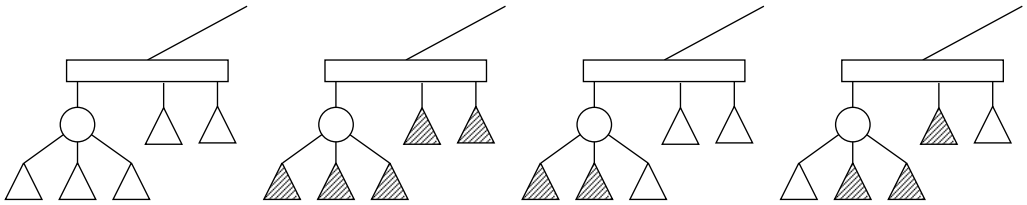


Figure 2.8: Nodes of type W, B, H, and A, respectively.

Theorem 2.3.1 ([58]) *A graph G is planar if and only if the pertinent roots of all subtrees in $T_2, T_3, \ldots, T_{n-1}$ of G are type B, H or A.*

A PQ-tree is called *reducible* if its pertinent root is type B, H or A; otherwise it is called *irreducible*. A graph G is planar iff all the trees T_k are reducible. If any T_k is irreducible, we make it reducible by appropriately deleting some leaves with label (v_{k+1}) from it. For a node x in an irreducible PQ-tree T_k, let the w-, h- and a-number be the minimum number of descendant leaves of x, which must be deleted from T_k such that x becomes type W, H and A, respectively. This is denoted by $[w, h, a]$. (Note that a partial node can not be made type B, because in this case we have to delete empty children, which is not allowed.) After computing $[w, h, a]$ for the pertinent root r of the PQ-tree, we set the type of r to H or A (according to the minimum of the h- and a-number) and traverse the tree top-down to determine the type of each pertinent node. The leaves of type W are removed from the tree. The result is a reducible tree.

We now concentrate on the computation of the $[w, h, a]$-numbers for every pertinent node x. To this end we process T_k bottom-up from the pertinent leaves to the pertinent root r, i.e., when we compute $[w, h, a]$ for node x, then the $[w, h, a]$ numbers of all pertinent children of x is already computed. For every node x with numbers $[w, h, a]$ it follows that $a \leq h \leq w$. If x is a leaf, then $[w, h, a] = [1, 0, 0]$, so assume x is not a leaf. Let $x_1, \ldots, x_p$ be the pertinent children of x, each child x_i with h- and w-number h_i and w_i. When one pertinent child of x is made type H and all other pertinent children of x type W, then the h-number of x is $\min_{1 \leq i \leq p}\{h_i + w_1 + \ldots + w_{i-1} + w_{i+1} + \ldots + w_p\} = w_1 + \ldots + w_p - \min_{1 \leq i \leq p}\{w_i - h_i\}$. The computation of the w- h- and a-numbers follows in a similar way as follows:

- The w-number for x is simply $\sum_y w$, over all y that are a pertinent child of x.
- We make a P-node x type H by making all full children type B, one partial child type H and all other partial children type W.
- We can make a P-node x type A in two different ways. We can make one partial child of x type A and make all other pertinent children of x type W, or we can make two partial children type H, all full children type B and make all other pertinent children type W.

- To make a Q-node x type H, we traverse the children of x from left to right and find the maximal consecutive sequence of pertinent children such that only the rightmost node in this sequence may be partial, all other nodes must be full. The same is done by traversing the children of x from right to left.

- We make a Q-node x type A by finding a maximal consecutive sequence of pertinent children of x such that all the nodes of this sequence except the leftmost and rightmost nodes are full. The endmost nodes of this sequence may be full or partial. Instead of this we could also make one of the pertinent children of x type A and make all the other pertinent children type W.

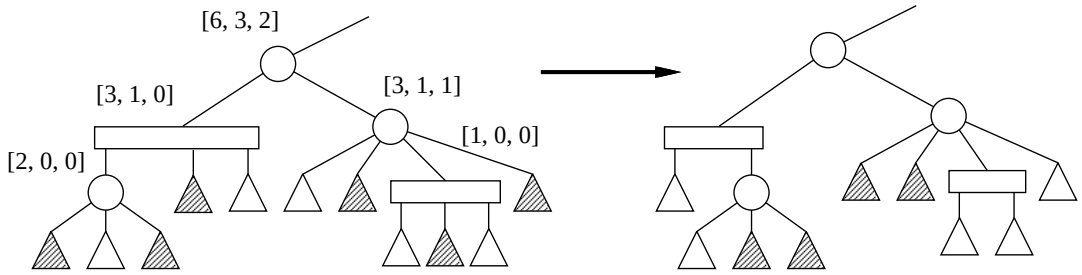


Figure 2.9: Computing the $[w, h, a]$-numbers.

```
PLANARIZE
  Construct the initial tree T_1 = T_1^*;
  for k := 1 to n − 1 do
    { make T_k reducible }
      compute the [w, h, a]-number for all pertinent nodes in T_k;
      if min{h, a} for the pertinent root r is not zero then
        make r type H or A corresponding to the minimum of h and a;
        traverse T_k top-down and determine the type of each pertinent node;
        delete leaves of type W from T_k;
    { reduction step }
      construct T_k^* from T_k by applying the replacements
      to the PQ-tree such that all remaining leaves with label
      (v_{k+1}) appear as a consecutive sequence in T_k^*;
    { vertex addition step }
      replace all full nodes of T_k^* by a new P-node x with all
      outgoing edges of vertex v_{k+1} appearing as children of x
  rof;
END PLANARIZE
```

Computing the $[w, h, a]$-number for node x can be done in $O(p(x))$ time, with $p(x)$ the number of pertinent children of x if x is a P-node, and with $p(x)$ the number of children of x if x is a Q-node. The number of children of all Q-nodes in any PQ-tree is at most n [58]. Since in any PQ-tree there are at most n pertinent leaves, n P-nodes and n Q-nodes, the total work for computing the $[w, h, a]$-numbers and making the tree reducible is $O(n)$ in each step, hence the algorithm takes $O(n^2)$ time in total.

Theorem 2.3.2 ([58]) *Algorithm* PLANARIZE *determines a planar subgraph G_p of the nonplanar graph G in $O(n^2)$ time.*

2.3.3 Maximal Planarization

Unfortunately, the algorithm PLANARIZE does **not** necessarily return a maximal planar subgraph, and in [56] a counterexample is given. If G is a complete graph K_n or a complete bipartite graph $K_{m,n}$, then PLANARIZE yields a maximal planar subgraph with $3n-6$ and $2(m+n)-4$ edges, respectively, which is best possible (see Jayakumar et al. [56] and Kant [66], respectively). In [58], Jayakumar et al. propose an $O(n^2)$ time algorithm that, given a biconnected planar subgraph G_p, outputs a maximal planar subgraph G'_p of G, with $G_p \subseteq G'_p$. However, as proved in Kant [66], this algorithm is incorrect, and several counterexamples are included in [66]. Kant also presents a new algorithm to augment G_p to a maximal planar subgraph G'_p of G, containing G_p, even when G_p is not biconnected. In this section we give the main ideas of this algorithm, for details the reader is referred to [66].

The idea is to do the planarization algorithm again, and to distinguish leaves $l_{e'}$, storing edges $e' \in G_p$, which we call *preferred leaves*, and leaves l_e storing edges $e \in G - G_p$, which we call *potential leaves*. In step k we compute an equivalent tree T_k^* of T_k in which all preferred leaves with label (v_{k+1}) form one consecutive sequence. Potential leaves with label (v_{k+1}) are **not** removed from T_k^*. Let x_k be the new P-node after the vertex addition step in T_k^*, with all outgoing edges of v_{k+1} appearing as children of x_k. To indicate the place of the sequence of incoming edges of v_{k+1} in T_k^*, we place adjacent to x_k a new node, called the *sequence indicator*, denoted by $< k+1 >$. (This idea is inspired by the direction indicator, described in Section 2.2.3). We call potential leaves and sequence indicators *empty leaves*. When computing the consecutive sequence of preferred leaves of v_{k+1} in T_k, we ignore the presence of empty leaves.

Observe that an edge $e = (v_i, v_j) \in G - G_p$ can be added to G_p without destroying the planarity, if in T_k between the corresponding potential leaf l_e and sequence indicator $< j >$, only empty leaves $l_{e_1}, \ldots, l_{e_r}$ appear. This follows because after deleting the leaves $l_{e_1}, \ldots, l_{e_r}$, T_k can be reduced such that l_e is an adjacent sibling of $< j >$. $< j >$ denotes the place of the consecutive sequence of the incoming edges of vertex v_j in PQ-tree T_{j-1}. (v_i, v_j) is an incoming edge of v_j as well. Hence the consecutive sequence can be enlarged, i.e., we can add (v_i, v_j) to G_p without

destroying the planarity. Such a pair of potential leaf and corresponding sequence indicator with only empty leaves in between, is called a *near pair*. (A more formal and precise definition of a near pair is given in [66].)

We compute in each step k the maximal consecutive sequence of preferred leaves with label (v_{k+1}), thereby ignoring the presence of empty leaves. When reducing the tree from T_k to T_k^*, we test for near pairs, that are part of the sequence. For every near pair $l_e, < j >$, we delete all empty leaves between $< j >$ and l_e from T_k, and add e to G_p. After reducing all near pairs, we delete all potential leaves and sequence indicators from T_k, which are part of the consecutive sequence. For every deleted sequence indicator $< j >$, we remove all corresponding potential leaves from T_k, since after deleting $< j >$, they cannot form a near pair anymore.

The maximal planarization algorithm can now be described at a high level as follows:

```
MaximalPlanarize
  assign st-numbers to the vertices of G;
  Planarize(G);
  construct the initial PQ-tree T_1;
  for k := 1 to n − 1 do
    {computing step}
      compute the maximal pertinent sequence in tree T_k
      of incoming edges of vertex v_{k+1} in G_p;
    {reduction step}
      apply the template matchings in the PQ-tree, and apply an additional
      step to reduce near pairs in the maximal pertinent sequence;
    {vertex addition step}
      for all deleted sequence indicators < j >,
      remove the corresponding potential leaves from T_k;
      replace all the full nodes in T_k by a P-node x with all
      outgoing edges of vertex v_{k+1} appearing as children of x;
      add the sequence indicator < k + 1 > as a sibling of x in T_k
  rof;
End MaximalPlanarize
```

Reducing the near pairs is complex and uses several extra pointers and a new type for a node in the PQ-tree. Explaining this in detail is beyond the scope of this thesis, and we only announce the following result:

Theorem 2.3.3 ([66]) *There is a maximal planarization algorithm based on PQ-trees, that requires $O(n^2)$ time and space.*

As noticed earlier, the strength of the PQ-tree algorithms is that we can compute for every vertex v_{k+1} the minimum number of edges which have to be deleted to

obtain planarity in step k. In particular, for graphs which are "almost planar", this approach seems to delete less edges than testing planarity incrementally per edge. We applied both approaches on randomly generated triangular planar graphs with n vertices and $3n-6$ edges, to which we randomly added k edges. The experimental results are given in Figure 2.10. Here v-TEST denotes the planarization algorithm described in Section 2.3.2, and e-TEST denotes the incremental planarity testing algorithm. The numbers shown denote the number of edges in the resulting planar subgraphs.

k	100 nodes		200 nodes		300 nodes		400 nodes		500 nodes	
	v-TEST	e-TEST	v-TEST	e-TEST	v-TEST	e-TEST	v-TEST	e-TEST	v-TEST	e-TEST
35	189	187	504	325	731	480	730	656	1189	804
75	207	293	438	326	683	474	917	637	1147	738
115	203	238	428	318	644	482	894	746	1107	763
155	205	239	421	324	630	487	829	635	1073	800
190	212	161	396	319	668	472	820	638	1037	791
230	222	163	387	318	624	471	823	636	1020	778
265	226	162	412	319	606	476	815	632	1048	769
305	232	162	405	325	606	467	805	633	1011	750
345	245	162	409	319	615	470	789	634	1008	786
375	234	166	398	317	567	480	795	617	965	804

Figure 2.10: Comparison between v-TEST and e-TEST on almost planar graphs.

We conclude from Figure 2.10 that for almost planar graphs the planarization algorithm based on vertex addition is preferred above the edge addition approach, when we search for a minimal number of deleted edges. We did the same test for random graphs. In this case the vertex addition approach seems only to be interesting for sparse graphs.

2.4 Biconnected and Triconnected Components

Until now, we have described how to test planarity, how to construct a planar subgraph in case the graph is nonplanar, and how to compute a planar embedding. In Part B and Part C we consider augmentation and drawing algorithms of planar graphs. An important issue in these algorithms is to determine whether the graph is bi- or triconnected, and if not, to split the graph into bi- and triconnected components. To this end we present two trees in this section. The first tree is the *BC-tree* T_{BC} of G for splitting G into its biconnected components, introduced by Harary [45]. This tree is also denoted as the *block tree*, $bc(G)$ or the *2-block tree* in [53]. The second tree is the *SPQR-tree* T_{SPQR} of G for splitting the biconnected components of G into triconnected components, introduced by Di Battista & Tamassia [21]. Related trees are the *2-subgraph tree* and the *3-block tree* [51, 52]. If a graph G is not biconnected, then we construct a BC-tree of G, and to each block of G we associate the SPQR-tree of this block. Both trees can be constructed in linear time.

2.4.1 The BC-tree

The *biconnected components* of a connected graph (also called *blocks*) are: (a) its maximal biconnected subgraphs, and (b) its bridges together with their endpoints (trivial blocks). In T_{BC}, every block is represented by a B-node and each cutvertex of G is represented by a C-node. There is an edge between a C-node u and a B-node b in T if and only if u belongs to the corresponding block of b in G. Every path in the BC-tree contains alternatingly B- and C-nodes. A *pendant* or a *pendant block* is a block which contains exactly one cutvertex, i.e., whose corresponding B-node is a leaf in T_{BC}. Let $p(v)$ of a cutvertex v denote the number of pendants, connected at v, i.e., the number of leaves, which are children of C-node v. Let p be the number of pendants of G, i.e., the number of leaves in T_{BC}. Let $d(v)$ denote the number of components of $G - \{v\}$, i.e., the graph after deleting cutvertex v, i.e., the degree of C-node v in T_{BC}. Each component of $G - \{v\}$ is called a v-block. Let $d = \max_{v \in V}\{d(v)\}$. Using a depth-first search algorithm as described by Tarjan [107], the tree T_{BC} can be constructed in linear time. In Figure 2.11 an example is given (from [53]).

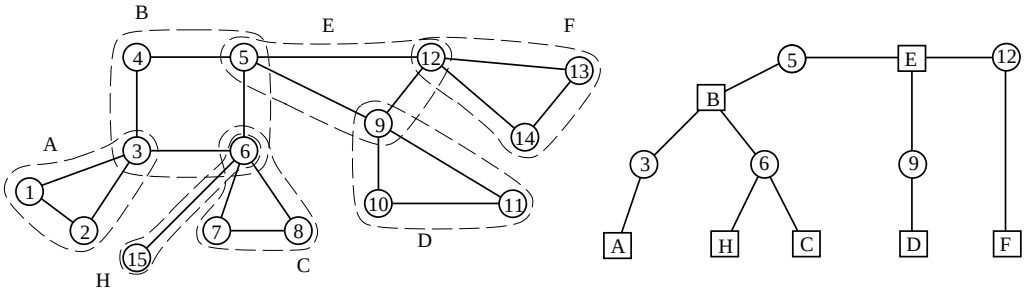


Figure 2.11: A graph G and its corresponding BC-tree (from [53]).

If G is outerplanar, then all vertices belong to the outerface. When constructing T_{BC} for G, we assume that the cutvertices $c_1, \ldots, c_k$ connected at a certain B-node b in T_{BC} appear in the order they appear on the outerface of the corresponding block of G. This means that in T_{BC}, the order of children of every B-node is fixed, and for every C-node any order of children is allowed. Such a "fixed" BC-tree can be constructed in linear time as well.

2.4.2 The SPQR-tree

The *3-connected* or *triconnected components* of a biconnected graph G are defined as follows (see also [49, 21]): if G is triconnected, then G itself is the unique triconnected component of G. Otherwise, let $\{u, v\}$ be a separation pair of G, i.e., deleting u and v from G disconnects G. We partition the edges of G into two disjoint subsets E_1

and E_2 ($|E_1|, |E_2| \geq 2$), such that the subgraphs G_1 and G_2 induced by E_1 and E_2 only have u and v in common. We continue the decomposition process recursively on $G_1' = G_1 + (u, v)$ and $G_2' = G_2 + (u, v)$ until no further decomposition is possible. The added edges (u, v) are called *virtual edges*. This procedure is called *splitting*. Each of the resulting graphs is either a triconnected simple graph, a set of three parallel edges (*triple bond*), or a cycle of length three (*triangle*). The *triconnected components* of G are obtained from such graphs by *merging* the triple bonds into maximal sets of parallel edges (*bonds*), and the triangles into maximal simple cycles (*polygons*). Merging is the procedure opposite to *splitting*.

We now describe the SPQR-tree T_{SPQR} which is used for splitting the biconnected components of G into triconnected components. To each B-node in T_{BC} an SPQR-tree T_{SPQR} is associated for the triconnected components of this block. For simplicity, we now assume that G is 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 $\{u, v\}$ is either an edge (u, v) of G or a maximal subgraph G' of G such that $\{u, v\}$ is not a split pair of G'. Let $\{s, t\}$ be a split pair of G. A split pair $\{u, v\}$ is a *maximal split pair* of G with respect to $\{s, t\}$ when for any other split pair $\{u', v'\}$, vertices u, v, s and t are in the same split component of $\{u', v'\}$.

Let e be an edge of G between vertices s and t, called the *reference edge*. T_{SPQR} of G with respect to e describes a recursive decomposition of G induced by its split pairs. T_{SPQR} is a rooted ordered tree whose nodes are of four types: S, P, Q and R. Each node b of T has an associated biconnected multigraph, called the *skeleton* of b, and denoted by *skeleton*(b). Also, it is associated with an edge in the skeleton of the parent b' of b, denoted by *edge*(b) . T_{SPQR} and the types of its nodes are recursively defined as follows.

Trivial Case: If G consists of exactly two parallel edges between s and t, then T_{SPQR} 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 $G_1, \ldots, G_k$ ($k \geq 3$), then the root T_{SPQR} is a P-node b. Graph *skeleton*(b) consists of k parallel edges between s and t, denoted by $e_1, \ldots, e_k$, with $e_1 = e$.

Series Case: In case the split pair $\{s, t\}$ has exactly two split components, one of them is the reference edge e, and we denote the other split component by G'. If G' has cutvertices $c_0, \ldots, c_k$ ($k \geq 2$) that partition G into its blocks $G_1, \ldots, G_k$, in this order from s to t, then the root of T_{SPQR} is an S-node b. Graph *skeleton*(b) 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: Otherwise, 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 T_{SPQR} is an R-node b. Graph *skeleton*(b) is obtained from G by replacing each subgraph G_i with the edge e_i between s_i and t_i.

In the last three cases (series, parallel and rigid), b has children $b_1, \ldots, b_k$ (in this order), such that b_i is the root of T_{SPQR} of the decomposition tree of graph $G_i \cup e_i$ with respect to reference edge e_i $(i = 1, \ldots, k)$. The virtual edge of node b_i is edge e_i in *skeleton*(b). The endpoints of e_i are the vertices of the split pair, and are also called the *poles* of node b_i. The tree so obtained has a Q-node associated with each edge of G, except the reference edge e. We complete T_{SPQR} by adding another Q-node, representing the reference edge e, and making it the parent of b such that it becomes the root. An example of an SPQR-tree is shown in Figure 2.12.

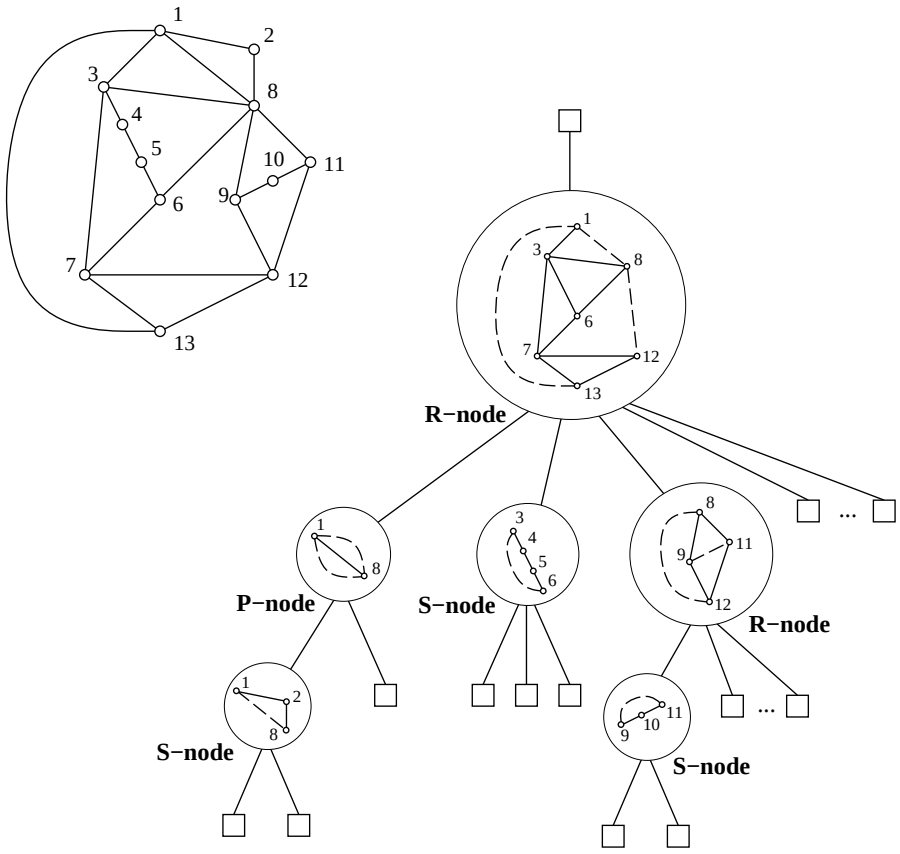


Figure 2.12: A graph and its SPQR-tree (from [21]).

The next three lemmas follow directly from the above definitions:

Lemma 2.4.1 ([21]) *Let b be a node of T_{SPQR}. We have:*

- *if b is an R-node, then skeleton(b) is a triconnected graph;*
- *if b is an S-node, then skeleton(b) is a cycle;*

- *if b is a P-node, then skeleton(b) is a triconnected multigraph consisting of a bundle of parallel edges;*
- *if b is a Q-node, then skeleton(b) is a biconnected multigraph consisting of two parallel edges.*

Lemma 2.4.2 ([21]) *The skeletons of the nodes of T_{SPQR} are homeomorphic to subgraphs of G. Also, the union of the sets of split pairs of the skeletons of the nodes of T_{SPQR} is equal to the set of split pairs of G.*

Lemma 2.4.3 ([21]) *T_{SPQR} of G has m Q-nodes and $O(n)$ S-, P- and R-nodes. Also the total number of vertices of the skeletons stored at the nodes of T_{SPQR} is $O(n)$.*

The following lemma is important for our algorithms, and can be proved by using the linear time algorithm of Hopcroft & Tarjan [49] for splitting a graph into its triconnected components.

Lemma 2.4.4 ([21]) *T_{SPQR} of G can be constructed in $O(m+n)$ time.*

It is possible to show that SPQR-trees of the same graph with respect to different reference edges are isomorphic and are obtained one from another by selecting a different Q-node as the root. SPQR-trees are closely related to the classical decomposition of biconnected graphs into triconnected components [49]. Namely, the triconnected components of a biconnected graph G are in one-to-one correspondence with the internal nodes of the SPQR-tree: The R-nodes correspond to triconnected graphs, the S-nodes to polygons, and the P-nodes to bonds. The SPQR-trees of planar graphs are introduced in [21] and are applied to the problem of on-line planarity testing. Notice that if $\{s,t\}$ is a split pair and $(s,t) \in E$, then there is a P-node b_i, with poles s and t, and one child of b_i corresponds with edge (s,t). We will also use the following lemma:

Lemma 2.4.5 *If b is an S-node, then parent(b) is not an S-node.*

Proof: Let b be an S-node with poles s,t. Suppose that $b' = parent(b)$ is also an S-node with poles s',t'. But replacing (s,t) by $skeleton(b)$ in $skeleton(b')$ gives a cycle with poles s',t'. This contradicts with the fact that the triconnected components are unique. □

In all our algorithms the Q-nodes, representing the edges of G, are not used, so we delete these from the SPQR-tree. We denote by $pertinent(b_i)$ the subgraph of G which corresponds with the subtree of the SPQR-tree rooted at b_i. Hence we can define $pertinent(b_i)$ recursively as the the subgraph of G obtained by replacing all virtual edges e_j by $pertinent(b_j)$ in $skeleton(b_i)$, if b_j is a child of b_i in T_{SPQR} and $e_j = edge(b_j)$.

Lemma 2.4.6 *If b is a P-node, then $deg(b) \geq 2$ in T_{SPQR}.*

Proof: Let b be a P-node, then *skeleton*(b) is a bond. At least two edges of this bond are virtual, hence correspond with triconnected components, whose corresponding node is connected with b in the SPQR-tree. □

The last lemma implies that if b is a leaf in T_{SPQR}, then b is an S- or R-node. In our algorithms, G is planar. By a theorem of Chiba et al. [12], triconnected planar graphs have a unique embedding. Hence, if b is a S- or R-node in the SPQR-tree T_{SPQR}, then *skeleton*(b) has a unique embedding. If b is a P-node, then we may permute the parallel edges between the two poles s, t in the skeleton of b.

2.5 The Canonical Ordering

A last tool we need in our algorithms is an ordering on the vertices and faces of a graph. Hereto we introduce in this section the *canonical ordering* for triconnected planar graphs. The canonical ordering plays a major role in Chapter 6 for triangulating planar graphs, and in Part C for drawing triconnected planar graphs on a grid. The canonical ordering generalizes the canonical ordering for triangulated planar graphs, described by de Fraysseix, Pach & Pollack [34], and also generalizes the st-ordering, defined by Even [30]. The st-ordering is used in several drawing algorithms [20, 21, 22, 23, 96, 105, 104, 106]. Let an embedding of a triconnected planar graph G be given. G_k denotes the subgraph of G, induced on the vertices $v_1, \ldots, v_k$.

(Canonical Ordering)
Let G be a triconnected plane graph with an edge (v_1, v_2) on the external face. Let $\pi = (V_1, \ldots, V_K)$ be an ordered partition of V, that is, $V_1 \cup \ldots V_K = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. Define G_k to be the subgraph of G induced by $V_1 \cup \ldots \cup V_k$, and denote by C_k the external face of G_k. We say that π is a *canonical ordering* of G if:

- V_1 consists of $\{v_1, v_2\}$.
- V_K is a singleton $\{v_n\}$, where v_n lies on the outerface and is a neighbor of v_1.
- Each C_k $(k > 1)$ is a cycle containing (v_1, v_2).
- Each G_k is biconnected and internally triconnected, that is, removing two internal vertices of G_k does not disconnect it.
- For each k in $2, \ldots, K-1$, one of the two following conditions holds:

  (a) V_k is a singleton, $\{z\}$, where z belongs to C_k and has at least one neighbor in $G - G_k$.

(b) V_k is a chain, $(z_1, \ldots, z_\ell)$, where each z_i has at least one neighbor in $G - G_k$, and where z_1 and z_ℓ each have one neighbor on C_{k-1}, and these are the only two neighbors of V_k in G_{k-1}.

Theorem 2.5.1 *Every triconnected planar graph G with pre-defined v_1, v_2, v_n has a canonical ordering.*

Proof: Let G be a triconnected planar graph with v_1, v_2 and v_n given in advance. The decomposition of the vertices in $V_1, \ldots, V_K$ will be defined by reverse induction. Assume that v_2 and v_n are neighbors of v_1 and belong to the outerface of a planar embedding of G. Notice that by triconnectivity of G, the graph $G_{n-1} = G - \{v_n\}$ is biconnected and the outerface C_{n-1} is a cycle, containing (v_1, v_2).

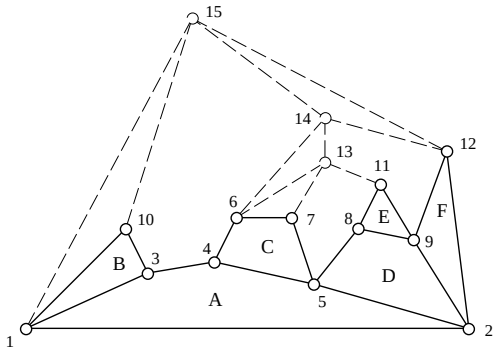
Let $2 < k < K$ be fixed. Assume that V_i has already been determined for every $i > k$ such that the subgraph G_k satisfies the conditions of the canonical ordering. Notice that if there are vertices $v \in G_k$ of degree 2 then $v \in C_k$. Notice also that by triconnectivity of G there are at least 3 vertices $c_\alpha, c_\beta, c_\gamma \in C_k$ having edges to vertices in $G - G_k$. Assume w.l.o.g. that $c_\alpha \neq v_1, v_2$. If G_k is triconnected then we can take $V_k = \{c_\alpha\}$ because by triconnectivity, c_α has at least three neighbors in G_k and $G_k - c_\alpha$ is biconnected.

Assume further that G_k is not triconnected, hence G_k contains separation pairs. Let v_x, v_y be a separation pair, and let G_1, G_2 be two components of $G_k - \{v_x, v_y\}$. Since G is triconnected, there is a path P between G_1 and G_2, not visiting v_x and v_y. In G_k, v_x and v_y are forming a separation pair, hence the edges of path P are removed in G_k. Since we defined the ordering by reverse induction, we removed only vertices and edges from the outerface. Hence path P goes between two vertices $c_{x'}, c_{y'}$, belonging to C_k with $c_{x'} \in G_1$ and $c_{y'} \in G_2$. This yields that v_x and v_y belong to C_k and one path between v_x and v_y on C_k is part of G_1; the other path on C_k between v_x and v_y is part of G_2. This holds for every separation pair v_x, v_y, hence all vertices of the separation pairs belong to C_k.

Let c_a, c_b be a separation pair such that $b - a$ is minimal. If $deg(c_{a+1}) > 2$ then there is a vertex c_α, $a < \alpha < b$, with at least one edge to a vertex deleted in step $j > k$, otherwise the graph $G - \{c_a, c_b\}$ is disconnected, which contradicts the triconnectivity of G. By minimality of $b-a$, c_α is not part of a separation pair in G_k, hence $G_k - c_\alpha$ does not have a cutvertex and the outerface of $G_k - c_\alpha$ is biconnected. We take $V_k = \{c_\alpha\}$ in the ordering.

Assume now that there is no separation pair c_a, c_b with $deg(c_{a+1}) > 2$. Then $b = a + 2$, because c_a and c_{a+2} are the only neighbors of c_{a+1} in G_k. Let now $1 \leq a' \leq a$ and $b \leq b' \leq r$ be such that all vertices $c_{a'+1}, c_{a'+2}, \ldots, c_{b'-1}$ have degree 2, and $deg(c_{a'}) > 2$ if $a' > 1$ and $deg(c_{b'}) > 2$ if $b' < r$. Notice that $v_1, v_2 \notin \{c_{a'+1}, \ldots, c_{b'-1}\}$ and every vertex $c'_i, a' < i < b'$, has an edge to $G - G_k$. If edge $(c_{a'}, c_{b'}) \in G$, then $G_k - \{c_{a'+1}, \ldots, c_{b'_1}\}$ is biconnected and we set $V_k = \{c_{a'+1}, \ldots, c_{b'-1}\}$.

Assume finally that $(c_{a'}, c_{b'}) \notin G$. Let F be the face in G_k, containing the vertices $c_{a'}, c_{a'+1}, \ldots, c_{b'}$. We claim that the path P in F between $c_{a'}$ and $c_{b'}$ not containing



vertex	*sepf*	*visited*
1	2	1
2	3	0
3	2	0
4	2	0
5	3	0
6	1	2
7	1	1
8	2	0
9	2	0
10	1	1
11	1	1
12	1	2

face	*outv*	*oute*
A	5	1
B	3	2
C	4	3
D	3	1
E	3	2
F	3	2

Figure 2.13: A graph with canonical ordering and corresponding variable-values at some step.

$c_{a'+1}, \ldots, c_{b'+1}$, does not contain another vertex on the outerface of G_k. Suppose not, i.e, the path P contains a vertex c_d on the outerface of G_k. Suppose w.l.o.g. $1 \leq d < a'$. But now it follows that $c_{d+1}, \ldots, c_{a'-1}$ is a chain of vertices of degree 2. Since $(c_d, c_{a'}) \notin G$ it follows that $deg(c_{a'}) = 2$, which is a contradiction. Hence $G_k - \{c_{a'+1}, \ldots, c_{b'_1}\}$ is biconnected and we set $V_k = \{c_{a'+1}, \ldots, c_{b'-1}\}$. □

If $V_k\{z_1, \ldots, z_\ell\}$ with $\ell > 1$, then precisely one face is added, otherwise one vertes is added to G_{k-1}. The algorithm for computing the canonical ordering is based on the proof of Theorem 2.5.1: We start with the entire graph G, and in each step we delete a face or vertex. For this we introduce a variable *outv*(F) and *oute*(F) for each face F, denoting the number of vertices and edges of F belonging to the current outerface. We also introduce a variable *sepf*(v) for every vertex v, denoting the number of different faces, containing a separation pair with vertex v. By the proof of Theorem 2.5.1, both v and w are part of the current outerface. We call the corresponding faces *separation faces*. Using these variables we can prove the following theorem.

Theorem 2.5.2 *For every triconnected planar graph a canonical ordering can be computed in linear time and space.*

Proof: Let an embedding of the triconnected planar graph G be given. Let every vertex v and edge e have pointers to the faces they belong to. Initially all variables *outv*(F), *oute*(F) and *sepf*(v) are set to 0. We take an arbitrary face F_{out} as outerface during the algorithm. Assign v_1 with neighbors v_2 and v_n on F_{out}. In every step we remove vertices from G and update F_{out}. For every vertex $v \in F_{out}$ and every edge $e \in F_{out}, e \neq (v_1, v_2)$, we increase *outv*$(F_v)$ and *oute*(F_e). for every $F_v \neq F_{out}$ where v belongs to, and $F_e \neq F_{out}$, where e belongs to.

For every vertex v which becomes part of F_{out} we have to compute *sepf*(v). Consider for this problem a face F where v belongs to. We claim that F is a

separation face if and only if $outv(F) \geq 3$ or if $outv(F) = 2$ and $oute(F) = 0$. This follows because precisely in these cases, F has at least two non-adjacent vertices on the outerface which makes this face a separating face. To compute $sepf(v)$ we count the number of incident faces F of v with $outv(F) \geq 3$ or $outv(F) = 2$ and $oute(F) = 0$. A face becomes at most once a separation face, because when $outv(F)$ or $oute(F)$ decreases then a vertex or edge from F is deleted and F is added to F_{out}. Every face F with $outv(F) = oute(F) + 1$ and $oute(F) \geq 2$ can be the next face in our ordering, because in this case the vertices of F, belonging to the outerface, form a consecutive sequence. Otherwise a vertex v, $v \neq v_1, v_2$, with $sepf(v) = 0$ and $visited(v) \geq 1$ (with $visited(v)$ the number of deleted neighbors of v) can be the next vertex v_k in our ordering. By Theorem 2.5.1, such a face or vertex exists.

The time complexity of the algorithm is the following: every vertex v has $deg(v)$ neighbors and belongs to $deg(v)$ faces. When v becomes part of F_{out} then updating $outv(F_v)$ for all incident faces of v requires $O(deg(v))$ time in total. Computing $sepf(v)$ requires $O(deg(v))$ time if v becomes part of F_{out}, and $O(1)$ if v was already part of F_{out} and is incident to a vertex, deleted in this step. Updating $oute(F_e)$ for an edge e, which becomes part of F_{out}, requires $O(1)$ time. When F becomes a separation face then $sepf$ of the other vertices of F, part of the outerface, must be increased by one. (This are at most two vertices and this happens only once, hence requires constant time.) Deleting a vertex or face can be done in time, constant in the number of deleted edges. Since $\sum deg(v) = 2m$, and $m = O(n)$, this yields a linear time and space algorithm. □

See Figure 2.13 for a graph with corresponding values of the variables $visited(v)$, $sepf(v)$, $outv(F)$ and $oute(F)$.

2.6 Augmentation and Drawing Algorithms

We now have all ingredients, which are necessary as a starting point for the drawing algorithms introduced in this thesis. However, several drawing algorithms require a biconnected, and sometimes even a triconnected planar graph. Therefore, we devote complete Part B to the problem of adding (a small number of) edges to a planar graph so the resulted augmented graph is biconnected, triconnected or triangulated. Efficient algorithms are described, often yielding performance ratio's which are tight, and with a complexity only an additive or multiplicative factor from optimal. In Chapter 3 an overview of our work and the relation to existing algorithms is given.

Using the tools of this chapter in combination with the augmentation tools described in Part B, we can start the investigation of representing or drawing a planar graph. Part C is devoted to this problem. In Chapter 9, we start with an overview of good existing algorithms. Several of these algorithms will be improved or generalized in the remaining chapters of Part C. We do not have the intention of being complete in this overview, and to present all different representation models known

in the literature. Only those algorithms, relevant for our work, are described. Nevertheless, this includes all main representations known to date. We mention the convex, orthogonal and hexagonal drawings, visibility representations, rectangular duals, drawings with minimum angle at least a constant. Some further representations, not mentioned here, are summarized in [18]. In this paper, Di Battista, Eades, Tamassia & Tollis give an annotated bibliography with more than 250 references, and several applications in which drawing algorithms appear. We refer the reader to this work for a more detailed survey of the current field of graph drawings. Additionally, in [103], Tamassia, Di Battista & Batini give another survey. This survey emphasizes the practical applications.

Part B

Augmenting Planar Graphs

Chapter 3

Introduction

Many problems concerning the planarity of graphs arise from the wish to draw the graph in an elegant way. In Part C we consider several different techniques to obtain a graphical layout of planar graphs. These techniques rely on the underlying structure and characteristics of the input graph. Moreover, almost always the drawing algorithm requires additional constraints of the planar graph G with respect to connectivity. As a first example, consider the graph drawing algorithms of Di Battista, Tamassia & Tollis [22], Rosenstiehl & Tarjan [96] and Tamassia & Tollis [104]. They all assume that the input graph is biconnected. In this case an st-ordering of the vertices can be constructed, which leads to an interesting linear time framework for a broad area of drawing applications. In Chapter 9 we outline these algorithms. As a second example, consider the convex drawing algorithm of Tutte [110, 111]. This algorithm is only valid for triconnected planar graphs. In Part C we introduce a completely new framework for drawing planar graphs, based on a so-called lmc-ordering. This framework leads to various graph representation and drawing applications. This ordering requires as input a triconnected planar graph. As a last example we mention the several recent algorithms for drawing a planar graph with straight lines on a grid [15, 34, 43, 98]. In this case the input graph has to be triangulated, i.e., every face has to be a triangle.

If the input graph does not satisfy these connectivity constraints, dummy edges have to be added to make it satisfy the constraints. These edges are only used to obtain the desired degree of connectivity for the purposes of the algorithm. The added edges are suppressed in the final drawing. The goal is to preserve the original graph as much as possible in the drawing of the augmented graph. Indeed, when we delete a large number of edges from an elegant drawing, the resulting drawing, i.e., the drawing of the original graph, can be less readable. For this reason, we consider the problem of finding the minimum number of edges that must be added to the input graph such that a biconnected or triconnected planar graph is obtained. This problem is not interesting if the graph must be triangulated, since every triangulated planar graph has $3n-6$ edges. In this case we inspect the problem how to augment a planar graph to a triangulated planar graph while minimizing the maximum degree.

Eswaran & Tarjan posed the problem of finding a minimum set of edges to augment a graph to a biconnected graph [27]. The solution of this problem heavily depends on the structure of the BC-tree $bc(G)$, defined in Section 2.4. Assume G is connected. Observe that every pendant block must receive an augmenting edge to satisfy the biconnectivity requirement for G. If p is the number of leaves of $bc(G)$, then at least $\lceil \frac{p}{2} \rceil$ augmentation edges are necessary. If a C-node v has degree $d(v)$ in $bc(G)$, then $d(v) - 1$ edges must be added between the v-blocks such that when we delete vertex v, the graph is still connected. Let d be the maximum degree of the C-nodes in $bc(G)$, and q the number of isolated nodes in $bc(G)$. Then the following theorem of Eswaran & Tarjan is obtained:

Theorem 3.0.1 ([27]) $\max\{d-1, q + \lceil \frac{p}{2} \rceil\}$ *edges are necessary and sufficient to make G biconnected.*

A linear time algorithm for biconnectivity augmentation based on this theorem was presented by Rosenthal & Goldner [97]. The algorithm contains a small error. Hsu & Ramachandran corrected it and simplified the algorithm such that also a fast parallel implementation is possible [53]. Eswaran & Tarjan [27] proved that the weighted variant of the biconnectivity augmentation problem is NP-complete. An approximation algorithm for the weighted biconnectivity augmentation problem is presented by Frederickson & Ja'Ja [36]. Khuller & Thurimella [72] simplified this algorithm, yielding an $O(m + n \log n)$ algorithm to biconnect G such that the total weight of the added edges is at most two times the optimal weight.

The triconnectivity augmentation problem has been investigated by Hsu & Ramachandran [52]. The algorithms for this problem are based on the structure of what they call a *3-block tree*, which is very close to the SPQR-tree. Assume G is biconnected, and let the SPQR-tree T_{SPQR} of G be given. We change T_{SPQR} such that every vertex v with $deg(v) = 2$ is also represented by one, unique, S-node b_i, with a virtual edge between the neighbors of v. We remove all Q-nodes from the SPQR-tree, since they are not necessary in the construction. Notice that after deleting the Q-nodes, every leaf in T_{SPQR} is either an R-node or represents a vertex v of degree 2.

Let p be the number of leaves of T_{SPQR}. Then at least $\lceil \frac{p}{2} \rceil$ edges are necessary to make G triconnected. If a P-node b_i has degree $d(b_i)$ in T_{SPQR}, then $d(b_i) - 1$ edges must be added between the different triconnected components, having the poles of b_i as cutting pair. Let d be the maximum degree of the P-nodes in T_{SPQR}. Then the following variant of Theorem 3.0.1 can be proved in the triconnected case.

Theorem 3.0.2 ([52]) $\max\{d-1, \lceil \frac{p}{2} \rceil\}$ *edges are necessary and sufficient to make a biconnected graph G triconnected.*

Hsu & Ramachandran [52] also presented an algorithm to augment a general graph to a triconnected graph, by adding a minimum number of edges. Very recently, Hsu presented a $O(m + n \cdot \alpha(m, n))$ algorithm to 4-connect a triconnected

graph by adding a minimum number of edges [51]. For the problem of finding a smallest augmentation for a graph to reach a given edge connectivity property, several polynomial time algorithms are known, see e.g. Eswaran & Tarjan [27], Frank [33], Frederickson & Ja'Ja [36], Kuller & Thurimella [72], Naor et al. [85], and Watanabe et al. [114, 115, 117].

In Part B we study these problems with the additional requirement that the augmented graphs have to be planar. In Chapter 4 we consider the planar biconnectivity augmentation problem. We prove that the problem of deciding whether adding at most K edges to a planar graph G yields a biconnected planar graph is NP-complete. We present an approximation algorithm for this problem, working in $O(n \cdot \alpha(n,n))$ time and it adds at most two times the minimum number of edges. A nice feature is that every vertex receives at most two augmentation edges. We extensively make use of incremental planarity testing (see e.g. [21, 75]), which finds a very nice application in this context. The approximation algorithm can also be used for making a planar graph bridge-connected while preserving planarity, and it follows that the same complexity and performance bound holds for this problem as well.

In Chapter 5 we consider the planar triconnectivity augmentation problem for biconnected planar graphs. Up to now it is not known whether the decision variant of this problem is NP-complete. We present an approximation algorithm for triconnecting a biconnected planar graph while preserving planarity. Inspecting the SPQR-tree and the planarity aspects in more detail leads to a linear time algorithm, which adds at most $\frac{3}{2}$ times the minimum number of edges. Every vertex v receives at most $\max\{2, \lceil \frac{deg(v)}{2} \rceil\}$ extra edges.

In Chapter 6 we consider the problem of triangulating a planar graph while minimizing the maximum degree. We show that the decision variant of this problem is NP-complete for biconnected planar graphs. We present a linear time algorithm that triangulates a triconnected planar graph such that every degree increases by at most 8. This algorithm uses the canonical ordering described in Section 2.5. Combining this result with results on triconnecting a planar graph yields that every connected planar graph can be triangulated such that the maximum degree is at most $\lceil \frac{3}{2}\Delta(G) \rceil + 11$, where $\Delta(G)$ denotes the maximum degree of G. This differs only an additive constant from the worst-case lower bound. Several simple triangulation algorithms, working in linear time and space, are included as well.

In Chapter 7 we consider the augmentation problems for the case that the input graph is outerplanar. Outerplanar graphs are an interesting class of planar graphs, since all vertices are on one common face. Several problems, which are NP-hard for planar graphs, become easily solvable for outerplanar graphs, e.g., the CHROMATIC NUMBER PROBLEM. Also the augmentation problems mentioned above can be solved optimally in polynomial time, if the input graph is outerplanar. In particular, we present a linear time algorithm to augment an outerplanar graph by adding a minimum number of edges such that the augmented graph is bridge-connected

and planar. We also show that biconnecting an outerplanar graph while preserving planarity by adding a minimum number of edges can be achieved in linear time, by modifying the algorithms of Hsu & Ramachandran [53]. Augmenting an outerplanar graph to a triconnected planar graph by adding a minimum number of edges can also be done in linear time. For this we use the algorithm for bridge-connecting outerplanar graphs. We also present a polynomial time algorithm to triangulate the interior faces of an outerplanar graph while minimizing the maximum degree. This algorithm is based on dynamic programming and can also be used to triangulate one face of a planar graph while minimizing the maximum degree.

The last chapter of Part B contains several concluding remarks. Observations with respect to bridge-connectivity and triconnecting arbitrary planar graphs are included as well.

Chapter 4

The Planar Biconnectivity Augmentation Problem

4.1 Preliminaries

In this section we present various observations and first results concerning the problem of augmenting planar graphs to satisfy biconnectivity and planarity constraints. Recall the definitions in Section 2.4 with respect to $bc(G), p(v), d(v)$ and v-blocks. Let p be the number of leaves of $bc(G)$, and q be the number of isolated vertices. If G is disconnected, we connect G by applying the following technique of Eswaran & Tarjan [27]. Let $bc(G)$ have t trees, numbered from 1 to t. For each i with $1 \leq i \leq 2t$, let $v(i)$ be a set of vertices of $bc(G)$ such that

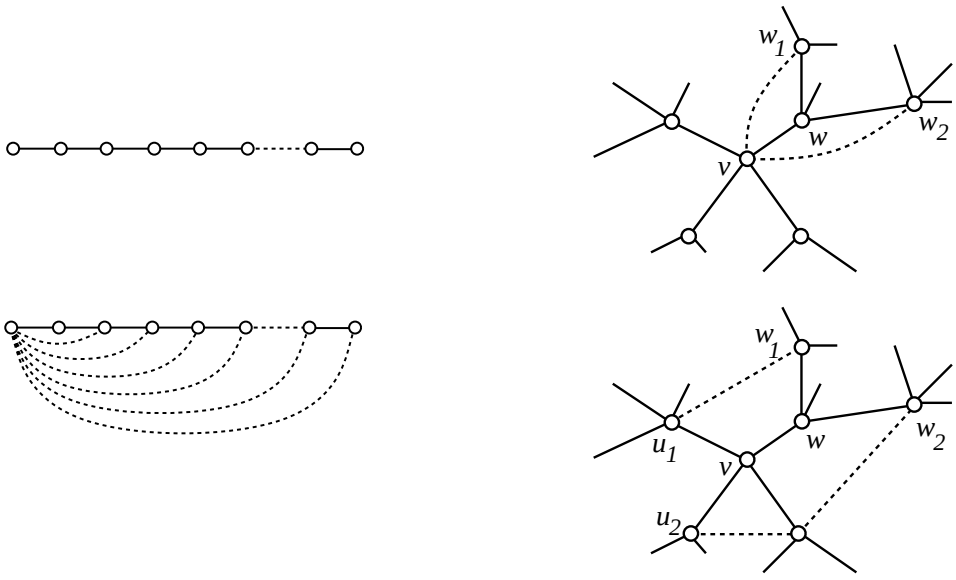
1. $v(2i-1)$ and $v(2i)$ are each a pendant or an isolated vertex in the ith tree of $bc(G)$, for each $1 \leq i \leq t$.

2. $v(2i-1) = v(2i)$ if and only if the ith tree of $bc(G)$ is an isolated vertex.

It easily follows that $bc(G) \cup \{(v(2i), v(2i+1)) | 1 \leq i < t\}$ is a tree having $p' = p + 2q - 2(t-1)$ pendants and no isolated vertices [27]. We call the corresponding algorithm for determining the trees of $bc(G)$, numbering the pendants and adding the edges $(v(2i), v(2i+1))$ for $1 \leq i < t$, CONNECT(G). We will use the algorithm CONNECT in Chapter 7 as well, when the aim is to bridge-connect, biconnect or triconnect an outerplanar graph by adding a minimum number of edges.

A simple linear time algorithm for biconnecting G while preserving planarity is the following approach due to Read [92]: if for any pair of consecutive neighbors u, w of v, u and w belong to different blocks, then the edge (u, w) is added. This collapses the two blocks into one. Applying this to all consecutive neighbors of every vertex yields a biconnected planar graph. Unfortunately, this algorithm can increase the degree of a single vertex by $O(n)$, as shown in Figure 4.1(a). To avoid this, the algorithm of Read is modified in four ways: (i) we change the embedding of G such that all neighbors of v, belonging to the same block, appear in a consecutive

sequence in $adj(v)$; (ii) we inspect the vertices in depth-first order; (iii) we test during the algorithm whether an added edge can be removed without destroying biconnectivity; (iv) let v have $d(v)$ v-blocks, then we observe that adding $d(v)-1$ edges between the v-blocks has the effect that v is not a cutvertex anymore. The modification of the algorithm of Read can be described as follows:



(a) One vertex gets $n-2$ extra edges.

(b) Every vertex gets at most 2 extra edges.

Figure 4.1: Making graphs biconnected.

BICONNECT(G)
  construct a planar embedding of G such that all neighbors of v_i,
  belonging to the same block, appear consecutively in $adj(v_i)$;
  number the cutvertices v_i of G in depth-first order;
  for every cutvertex v_i (in increasing v_i-number) **do**
    let $adj(v_i) = \{u_1, \ldots, u_k\}$, with u_1 and u_k belonging to different blocks;
    for $j := 1$ **to** $k-1$ **do**
      if u_j and u_{j+1} belong to different blocks **then**
        add an edge (u_j, u_{j+1}) to G;
        if (v_i, u_j) (or (v_i, u_{j+1})) was added to G earlier **then**
          remove (v_i, u_j) (or (v_i, u_{j+1}), resp.) from G
    rof
  rof;
END BICONNECT

Lemma 4.1.1 *In linear time a planar embedding of a planar graph G can be constructed with the property that for every vertex v, all neighbors of v, belonging to the same block, appear consecutively.*

Proof: Every edge belongs to a unique block. After labeling the edges with the numbers of the blocks, we sort $adj(v_i)$ such that all neighbors of v_i, belonging to the same block, appear consecutively in $adj(v)$. We do it such that if neighbors v_j and v_k belong to the same block and initially v_j appeared before v_k in $adj(v_i)$, then after sorting v_j is before v_k in $adj(v_i)$. Since the ordering between the edges in the same block is not disturbed, and there are no edges between the different blocks except the incident edges of v_i, it follows that the new embedding is planar. □

Lemma 4.1.2 *Algorithm* BICONNECT(G) *gives a biconnected planar graph.*

Proof: Let $B_1, \ldots, B_l$ be the v_i-blocks, in clockwise order around cutvertex v_i in BICONNECT(G), with $u_1 \in B_1$ and $u_k \in B_l$. Then an edge is added between B_j and B_{j+1}, for $1 \leq j < l$. Hence after the augmentation by algorithm BICONNECT(G), all neighbors of v_i belong to one common block. Thus v_i is not a cutvertex anymore. Consider consecutive neighbors u_j and u_{j+1} of v_i. If (v_i, u_j) was added in BICONNECT(G) then there was a path from v_i to u_j initially. But then there was a path initially from u_{j+1} to u_j, not using the edge (v_i, u_{j+1}). Adding (u_j, u_{j+1}) implies a cycle containing the vertices u_j, v_i, u_{j+1}, not containing (v_i, u_j), hence (v_i, u_j) can be removed without destroying the biconnectivity. Similar for (v_i, u_{j+1}). □

Lemma 4.1.3 *In algorithm* BICONNECT(G) *every vertex receives at most 2 extra incident edges.*

Proof: Assume w.l.o.g. that vertex v_i is the only vertex of a block. When visiting a vertex w of which v_i is a neighbor, v_i receives at most two incident augmenting edges, say (w_1, v) and (w_2, v). By the depth-first order v will be visited before the other neighbors of v. If v belongs to at least two blocks, then edges are added between neighbors of v, and the edges (w_1, v) and (w_2, v) are removed. In this case v is on the outerface of a block. The two neighbors of v on the outerface, say u_1 and u_2, can be cutvertices. By the algorithm, v receives at most 1 incident edge when visiting u_1, say to u_1', and at most 1 incident edge when visiting u_2, say to u_2'. If v receives another incident edge, e.g. when visiting u_1', then edge (v, u_1') is removed. Hence v receives at most two incident augmenting edges (see Figure 4.1(b)). □

Lemma 4.1.4 *Let b be the number of blocks of G.* BICONNECT(G) *adds at most $b - 1$ edges to G.*

Proof: Let T_{BC} be the BC-tree of G. Root T_{BC} at an arbitrary B-node, say b_r. Every C-node v has degree $d(v)$ in T_{BC} and has only B-nodes as neighbors in T_{BC}. When visiting C-node v in BICONNECT(G), at most $d(v)-1$ edges are added, i.e., the number of children of C-node v in T_{BC}. Hence for every B-node (except the root b_r) at most one edge is added in BICONNECT(G). b is the number of blocks of G, i.e., the number of B-nodes in T_{BC}, thus BICONNECT(G) adds at most $b-1$ edges. □

Corollary 4.1.5 *There is a linear time and space algorithm to augment a planar graph such that it is biconnected and planar and the degree of every vertex increases by at most 2.*

However, it is not difficult to modify the example in Figure 4.1(a) such that BICONNECT(G) adds $O(n)$ edges to achieve biconnectivity, while one edge would already be sufficient to satisfy biconnectivity. In the remaining part of this chapter we try to biconnect G by adding a minimum number of edges.

4.2 NP-completeness

Theorem 4.2.1 *The problem of deciding whether adding at most K edges to a connected planar graph $G=(V,E)$ can lead to a biconnected planar graph is NP-complete.*

Proof: Clearly the problem is in NP: guess $L \leq K$ edges to be added to G, add the edges to G in some way and check in polynomial time whether the resulting graph is biconnected and planar.

To prove the NP-hardness, we show that 3-PARTITION (which is well-known to be NP-complete in the strong sense [38]) is reducible to the planar biconnectivity augmentation problem. Let an instance of 3-PARTITION be given, i.e., a set A of $3m$ elements $a_1,\ldots,a_{3m}$, a bound $B \in Z^+$ and a size $s(a_i) \in Z^+$ for each $a_i \in A$ such that $B/4 < s(a_i) < B/2$ and $\sum_{a_i \in A} s(a_i) = mB$. The question is whether A can be partitioned into m disjoint sets $A_1, A_2, \ldots, A_m$ such that, for $1 \leq i \leq m$, $\sum_{a \in A_i} s(a) = B$ (note that each A_i must therefore contain exactly three elements from A). To reduce 3-PARTITION to the planar biconnectivity augmentation problem we construct a planar graph as follows:

Introduce a vertex x and for each $i, 0 \leq i < m$, introduce vertices b_i, c_i, and the edges $(b_i, b_{i+1}), (b_i, x)$, (b_i, c_i) and (b_{i+1}, c_i) (additions modulo m). Introduce for each vertex c_i B additional edges to B new vertices. Introduce $3m$ new vertices $a_1,\ldots,a_{3m}$ connected at x. Each vertex a_i gets $s(a_i)$ additional edges to $s(a_i)$ new vertices (see Figure 4.2). Let G be the resulting graph.

Clearly G has $2mB$ pendant blocks, so at least mB edges are necessary to make G biconnected. G can be made biconnected without destroying planarity by adding

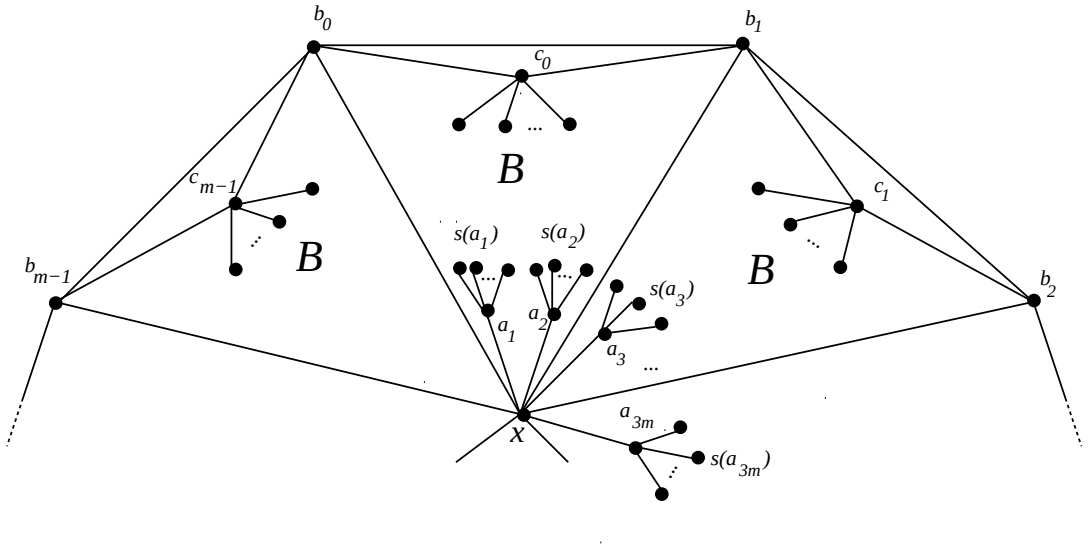


Figure 4.2: Construction of the graph for the NP-completeness proof.

exactly mB edges, if and only if it is possible to have a matching edge from each pendant of cutvertex a_i to a unique pendant of a cutvertex c_i. This can be done, if and only if for each c_i, the B pendants are matched with pendants of some vertices $a_{i_1}, a_{i_2}, a_{i_3}$, i.e., if and only if there exists a partition of A into m disjoint sets $A_1, \ldots, A_m$ such that for all $i, 1 \leq i \leq m, \sum_{a \in A_i} s(a) = B$. As G can be constructed in time, polynomial in m and B, this is a polynomial time transformation from the strongly NP-complete 3-PARTITION problem to the planar biconnectivity augmentation problem, hence the latter is NP-complete. □

4.3 Approximation Within 2 Times Optimal

As the planar biconnectivity augmentation problem is NP-complete, we focus our attention on approximation algorithms for the problem. In this section we present an $O(n \cdot \alpha(n, n))$ algorithm for augmenting a planar graph G to a biconnected planar graph G', while keeping the number of added edges within 2 times optimal. We may assume w.l.o.g. that G is connected, otherwise we can apply the algorithm CONNECT to G.

Let T_{BC} be the BC-tree of G. We root T_{BC} at a B-node b_r. Let for each block B_i in G, b_i denote the corresponding B-node in T_{BC}, and let $c_i = \mathit{parent}(b_i)$ in T_{BC}. We add pointers from the children to their parent in T_{BC}, and we store the children of a node in a doubly linked list. If we want to add an edge $e = (v, w)$ to G, then we have to test whether $G \cup e$ is still planar. We denote this test by

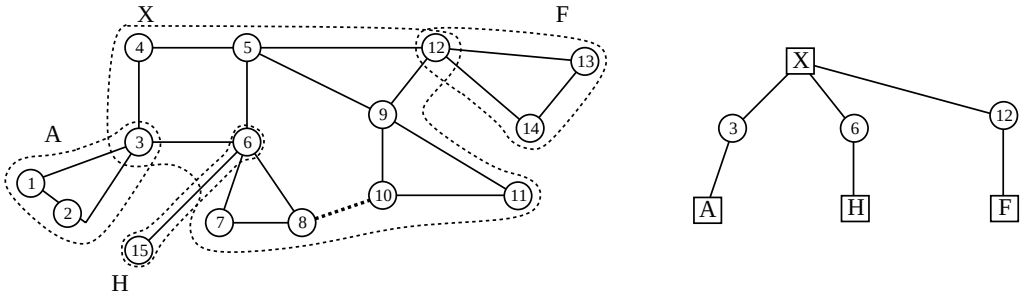


Figure 4.3: Updating T_{BC} after adding edge $(8, 10)$ to the graph of Figure 2.11.

PLANAR(v, w). This test is called *incremental planarity testing* in the literature, and several sophisticated algorithms are known for it ([21, 119, 75]). The best algorithm (in a theoretical sense) for this problem is given by La Poutré [75]: it requires $O(\alpha(q, n))$ time worst-case for the q^{th} query of testing whether an edge e can be added while preserving planarity. The time of adding e to G is $O(\alpha(n, n))$ time, amortized over $O(n)$ edges.

Let a planar embedding of G be given. We call a vertex v on the outerface of a block B_i an *outside vertex*, if $v \neq parent(b_i)$. The idea is to search for an arbitrarily outside vertex v of a pendant block B_i this B-node b_j such that: (i) b_j is as closest as possible to b_r in T_{BC}, and (ii), after changing the embedding, v is an outside vertex of B_j. Let $b_k = parent(c_j)$, and let w be an outside vertex of B_k, with $(w, c_j) \in G$. v is an outside vertex of B_j iff PLANAR$(v, w) = true$. Let $d(c_i)$ be the degree of C-node c_i in T_{BC}. If $p(c_i) = d(c_i) - 1$, then v has only leaves as children, and we first add edges between the $p(c_i)$ corresponding pendant blocks to coalesce them into one block. When we add an edge between two vertices of different blocks, say B_1 and B_2, then all blocks on the path between the corresponding nodes of B_1 and B_2 in T_{BC} become part of one block. More precisely, T_{BC} must be updated as follows, given by Rosenthal & Goldner:

Lemma 4.3.1 ([97]) *Given a graph G and its BC-tree T_{BC}, consider the cycle C created by adding an edge between two vertices v and w of T_{BC}. Let G' be the graph obtained from G by adding an edge between v' and w' in G where v' and w' are non-cutvertices in the blocks represented by v and w respectively. Let T'_{BC} be the BC-tree of G'. The following relations hold between T_{BC} and T'_{BC} (see also Figure 4.3):*

1. *Vertices and edges of T_{BC} that are not in the cycle C remain the same in T'_{BC}.*
2. *All B-nodes in T_{BC} that are in the cycle C contract to a single B-node b' in T'_{BC}.*
3. *Any C-node in C with degree equal to 2 is eliminated.*

4. *A C-node x in C with degree greater than 2 remains in T'_{BC} with edges incident on nodes not in the cycle. The node x also attaches to the B-node b' in T'_{BC}.*

The algorithm can now be described as follows:

```
2*OptBiconnect(G);
  compute the BC-tree T_BC of G; root T_BC at b_r;
  while T_BC is not a single node b_r do
    let c_i be a C-node in T_BC with p(c_i) = d(c_i) − 1;
    coalesce pendant blocks of c_i into one by adding p(c_i) − 1 edges;
    let B_i be the coalesced block; let v be an outside vertex of B_i;
    repeat
      w := outside vertex of B_j, with b_j = parent(c_i);
      c_i := parent(b_j)
    until b_j = b_r or Planar(v, w) = false;
    add (v, w) to G;
    update T_BC
  od;
End 2*OptBiconnect
```

Theorem 4.3.2 *Algorithm* 2*OptBiconnect(G) *can be implemented to run in* $O(n \cdot \alpha(n,n))$ *time.*

Proof: We represent T_{BC} by a Union-Find structure. Every node in T_{BC} has a *parent*-pointer, and represents a set of nodes, initially only one B- or C-node. The data structure required for the incremental planarity tester of La Poutré is initialized by adding incrementally every edge of the original planar graph G to the data structure. This costs $O(n \cdot \alpha(n,n))$ time in total [75]. During the algorithm we maintain the set of cutvertices c_i with $p(c_i) = d(c_i) - 1$. After determining such a cutvertex c_i, we first add $p(c_i) - 1$ edges between the pendant blocks, connected at c_i. We update T_{BC} by applying $p(c_i) - 1$ unions on these pendant blocks in the underlying Union-Find structure, which decreases the number of nodes in T_{BC} by $p(c_i) - 1$. Hence this requires $O((p(c_i) - 1) \cdot \alpha(n,n))$ time amortized.

Next we compute a path P from the coalesced pendant block B_i to B_j such that Planar$(v,w) = true$, with $v \in B_i$ and $w \in B_j$. Let $|P|$ denote the length of P, then $|P|$ is even, because T_{BC} consists of alternatingly B- and C-nodes. P is computed by testing $\frac{|P|}{2} + 1$ times whether Planar$(v,w) = true$. Using the algorithm of La Poutré, this requires $O(|P| \cdot \alpha(n,n))$ time amortized. Updating T_{BC} follows by a union of all B-nodes on the path between b_i and b_j in T_{BC}. We test for every C-node c on P, whether $deg(c)$ was 2, and if so, we eliminate c. The total time for updating T_{BC} by collapsing path P is $O(|P| \cdot (1 + \alpha(n,n)))$. The number of nodes in T_{BC} decreases by at least $\lceil \frac{|P|}{2} \rceil$.

If at the end G is biconnected, then T_{BC} is a single leaf, thus the total work during all augmentation steps is $O(n \cdot \alpha(n,n))$. □

Lemma 4.3.3 *Every vertex receives at most 2 extra incident edges.*

Proof: When we add $p(c_i) - 1$ edges between the pendant blocks, say in order $B_1, \ldots, B_{p(c_i)-1}$, connected at c_i, then B_1 and $B_{p(c_i)-1}$ get one incident edge, the other blocks $B_2, \ldots, B_{p(c_i)-2}$ two. Then we add an edge between a vertex of B_1 and a neighbor w of cutvertex c in block B_j in 2*OPTBICONNECT. After this addition it follows that (c,w) becomes a chord, i.e., is not on the outerface of the coalesced block. Hence in the next step we can take another neighbor w' of c on the outerface of B_j, and if $c = c_i$, we can take a neighbor of c on the outerface of $B_{p(c_i)-1}$. □

Theorem 4.3.4 *Algorithm* 2*OPTBICONNECT *adds at most 2 times the minimum required number of edges to G in $O(n \cdot \alpha(n,n))$ time.*

Proof: By Theorem 3.0.1, the number of required edges is at least $\lceil \frac{p}{2} \rceil$, with p the number of pendant blocks, i.e., the number of leaves in T_{BC}. In 2*OPTBICONNECT, the $p(c_i)$ pendants of c_i are coalesced into one by adding $p(c_i) - 1$ edges. The coalesced pendant receives one incident augmenting edge to a block B_j, with b_j an ancestor of c_i in T_{BC}.

A B-node $b_j \neq b_r$ becomes a leaf in T_{BC}, if there is no descendant pendant block for which an outside vertex can become an outside vertex of B_j. Hence no edge (v,w) can be added to G without destroying planarity, with $v \in B_i$, b_i a descendant of b_j in T_{BC}, and $w \in B_l$, b_l not a descendant of b_j in T_{BC}. Thus also in the optimal solution b_j becomes a leaf, and an extra edge is required for b_j. If there are s non-leaf B-nodes, becoming a leaf during the augmentation algorithm, then at least $\lceil \frac{p}{2} \rceil + \lceil \frac{s}{2} \rceil$ edges are required in the optimal solution to preserve the planarity. In 2*OPTBICONNECT we add an edge from each leaf, hence $p + s$ edges are added, which completes the proof. □

Figure 4.4 shows an example of the algorithm 2*OPTBICONNECT(G) as it applies to the graph of Figure 2.11, with $B = root(T_{BC})$.

4.4 A Special Case

In this section we consider a special case of the problem of adding edges to a planar graph, such that the resulting graph is biconnected and still planar. Recall from Theorem 4.2.1 that the problem of deciding whether adding at most K edges to a connected planar graph $G = (V,E)$ can lead to a biconnected planar graph is NP-complete. The problem appears to be solvable in polynomial time when the input graph G preserves a special structure:

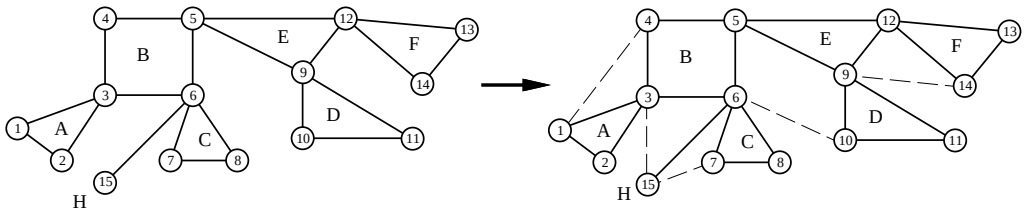


Figure 4.4: Biconnecting the graph of Figure 2.11 by 2*OptBiconnect.

Theorem 4.4.1 *If all cutvertices of G are part of one triconnected component, then a minimum number of edges, whose addition to G gives a biconnected planar graph can be determined in $M(n, n^2)$ time, where $M(n, m)$ is the time required to compute a maximum cardinality matching on a graph with n vertices and m edges.*

Proof: Let $G' \subseteq G$ be the triconnected component, containing all cutvertices. By the definition of triconnected components (see Section 2.4), G' is either a pair of vertices a, b with several multiple edges (a, b), or a cycle, or a triconnected graph. If G is a pair of vertices a, b with multiple edges, then the problem is easy to resolve in linear time: add $\min\{p(a), p(b)\}$ edges between pendants of a and b, where $p(v)$ is the number of pendants connected at v. The remaining $\max\{p(a), p(b)\} - \min\{p(a), p(b)\}$ pendants receive one incident edge to the remaining part of the graph. If G' is a cycle, then we can use the algorithm OuterBiconnect of Chapter 7. This algorithm biconnects an outerplanar graph in linear time by adding a minimum number of edges, while preserving planarity. This algorithm can also be used to biconnect a graph, where all pendants are part of one outerplanar graph.

Assume now that G' is a triconnected graph. Hence G' has one, unique, planar embedding, which is the key to obtain an optimal solution. We construct a graph H by representing every cutvertex v of G by $p(v)$ vertices in H. We add an edge (v_i, v_j) in H if and only if v_i and v_j are on a common face in G' and do not represent the same vertex in G.

Every vertex in H corresponds to one pendant in G, and every edge in H corresponds to an edge, which can be added between two pendants. The aim is to add as many as possible edges between pendants without destroying planarity. An edge between pendants of v and w is only possible, if v and w are on a common face, i.e., if there is an edge $(v, w) \in H$. We try to find a subset $E' \subseteq E_H$ such that $|E'|$ is minimal and for each vertex $v \in H$ there is an edge $e \in E'$ such that v is an endpoint of e. Hereto we compute a *maximum cardinality matching* M in H. A maximum cardinality matching M is a maximum set of edges $M \subseteq E_H$ such that each vertex v is an endpoint of at most one edge $e \in M$. Every edge $(v, w) \in M$ corresponds to an edge between pendants of v and w in G. Let $V' \subseteq V_H$ be the set of vertices, which do not have an incident edge in M. Every pendant, corresponding

to a vertex $v \in V'$ must receive an additional edge to an arbitrary vertex such that planarity is maintained, because every pendant must get at least one augmenting edge. It directly follows that the computed set of edges is minimum.

If more edges are added in one face, then crossings may occur between the added edges. To remove these crossings, we apply the algorithm OUTERBICONNECT of Chapter 7. This algorithm biconnects an outerplanar graph in linear time by adding a minimum number of edges, while preserving planarity. To apply the algorithm OUTERBICONNECT here, we remove all added edges in one face F, and we apply the algorithm OUTERBICONNECT to F. This gives a minimum set of added edges in F without crossings. Applying OUTERBICONNECT to all faces in G gives a biconnected planar graph without crossings.

Notice that $|V_H| = O(n)$, but $|E_H|$ can be $O(n^2)$, which completes the proof. □

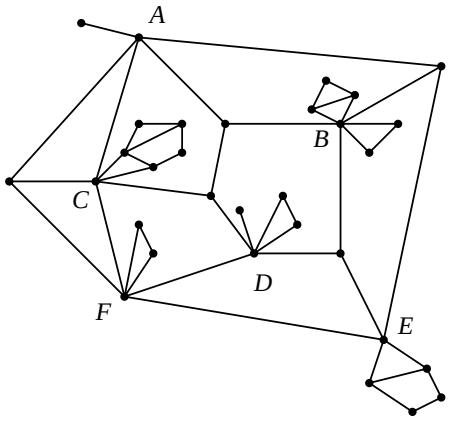
The fastest algorithm (in a theoretical sense) is presented by Micali & Vazirani, who proved that $M(n,m) = O(m\sqrt{n})$ [83], hence this yields an $O(n^{2.5})$ time algorithm. In Figure 4.5 an example of this special case is given.

4.5 The Planar Bridge-Connectivity Augmentation Problem

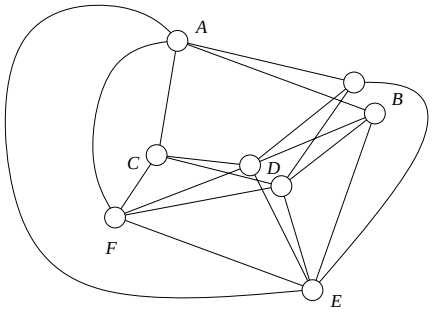
In this section we show that the algorithm for the planar biconnectivity problem can also be used to make a connected graph G bridge-connected with the same complexity and performance bound. Recall that a bridge is an edge (u,v) such that $G - \{(u,v)\}$ is disconnected. A *bridge-connected graph* is a graph without bridges. A maximal bridge-connected subgraph of G is called a *bridge-block*.

Let T_{BC} be the BC-tree of G. Every bridge in G is a trivial block, and is represented by a B-node in T_{BC}. We now merge two B-nodes in T_{BC} into one, if the corresponding blocks belong to the same bridge-block. We remove the C-nodes of degree 1 from T_{BC}. After this merging and deleting process, every B-node corresponds with a bridge or a bridge-block, and every C-node is an endpoint of at least one bridge. Let T_{BBC} be the transformed tree. We call T_{BBC} the *bridge-block tree*. T_{BBC} is a tree, and can be constructed in $O(m+n)$ time by using depth-first search, as described by Tarjan [107, 108]. Any set of edges which bridge-connects T_{BBC} corresponds to a set of edges which bridge-connects G.

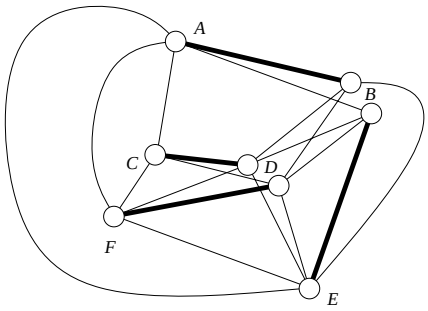
Let a *pendant bridge-block* correspond to a leaf in bridge-block tree T_{BBC}. Let $p(c)$ be the number of leaves of cutvertex c in T_{BBC}. If $p(c) > 2$, then we can add edges between corresponding pendant bridge-blocks, connected at c, until one ($p(c)$ is odd) or two ($p(c)$ is even) pendant bridge-blocks at c remain. Every added edge removes two bridges. We continue this, until one or two pendant bridge-blocks remain at c, because there has to go an edge from a pendant bridge-block of c to a bridge-block, not connected at c to achieve bridge-connectivity (if $c \neq root(T_{BBC})$).



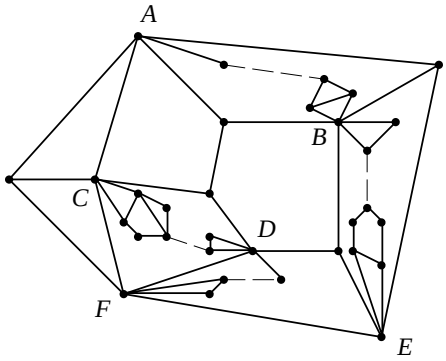
(a) The initial graph G.



(b) The matching graph H.



(c) Maximum matching in H.



(d) The optimal augmentation.

Figure 4.5: Biconnecting a planar graph using a maximum matching graph.

So assume further that $p(c) \leq 2$ for every vertex $c \in T_{BBC}$. Bridge-connecting G is obtained by adding edges between bridge-blocks of G while preserving planarity. Doing this within two times optimal follows by applying 2*OptBiconnect(G), but now using T_{BBC}. We call the resulting algorithm 2*OptBridgeConnect(G).

The problem of finding in polynomial time a minimum number of edges, that when added to a given planar graph yields a bridge-connected planar graph, remains as an interesting open problem.

Chapter 5

The Planar Triconnectivity Augmentation Problem

5.1 Preliminaries

In this chapter we consider the question how to augment a biconnected graph such that the augmented graph is triconnected and still planar. Triconnected planar graphs have nice characteristics, e.g., they have only one embedding in the plane and they can be drawn with convex faces, by a result of Tutte [110]. These characteristics are used in Section 10.2 to obtain a linear time algorithm for drawing a triconnected planar graph convexly on an $(n-2) \times (n-2)$ grid. Thomassen [109] characterized the class of planar graphs which can be drawn with convex faces. If a graph G does not satisfy these constraints, we can try to draw G with a minimum number of non-convex faces. Unfortunately, this problem is rather difficult, as stated in the following theorem.

Theorem 5.1.1 *The problem of deciding whether a biconnected planar graph can be drawn with $\geq K$ convex faces is NP-complete.*

Proof: (i) The problem is in NP: assume G has $F \geq K$ faces. Pick $L \leq F - K$ faces, triangulate these faces by adding a vertex in each face with edges to all other vertices, belonging to this face. Check whether the resulting graph satisfies the constraints of Theorem 5.1 of [109], which is a characterization of the planar graphs that can be drawn with convex faces. If this is the case then G can be drawn with $\geq F - L$ convex faces. This checking is possible in polynomial time by the algorithm of Chiba et al. [14].

(ii) To prove the NP-hardness we use a transformation from the VERTEX COVER PROBLEM on triconnected planar graphs (which can easily be shown to be NP-complete, by modifying the graph G in the reduction in the proof in [39], such that G is triconnected). Let a triconnected planar graph $G = (V_G, E_G)$ and a positive integer $K \leq n$ be given. The question in the VERTEX COVER PROBLEM is whether

there is a subset $V' \subseteq V$ with $|V'| \leq K$ such that for each edge $(u, v) \in E$ at least one of u and v belongs to V'. Inspect the dual graph G^* of G. The vertices of the dual graph G^* of a planar graph G are the faces of G. There is an edge in G^* if and only if the corresponding faces of the endpoints have a common edge in G. Note that G^* is also triconnected and, hence, has exactly one planar embedding. We construct a new graph G_1^* from G^* by changing every edge (a, b) in G^* by $(a, ab_1), (ab_1, b), (a, ab_2)$ and (ab_2, b) in G_1^*, thereby introducing two new vertices, ab_1 and ab_2. Note that G_1^* is biconnected.

We now claim that G_1^* has a planar embedding with $\geq K$ convex faces, if and only if G has a vertex cover of size $\leq K$. Suppose that $(a, b) \in F_1$ and $(a, b) \in F_2$ in G^*. In any drawing of G_1^*, F_1 or F_2 cannot be drawn convexly. Since the goal is to obtain a minimum number of non-convex faces, the problem is to find in G^* the smallest set S of faces such that for every edge (a, b) in G^*, F_1 or F_2 belongs to S, if (a, b) belongs to F_1 and F_2 in G^*. The faces of S are drawn non-convex in G_1^*. The faces F_1 and F_2 with common edge (a, b) correspond to an edge (v_{F_1}, v_{F_2}) in G. For every edge $(v_{F_1}, v_{F_2}) \in G$, v_{F_1} or v_{F_2} belongs to S. Hence S is a vertex cover of G. There is a vertex cover in G of size $\leq K$ if and only if there is a set S of faces of size $\leq K$ in G^*. But this exists if and only if there is a set of non-convex faces of size $\leq K$ in G_1^*. The construction is easily computable in polynomial time, so the problem whether there exist a drawing such that the biconnected planar graph can be drawn with $\leq K$ non-convex faces is NP-complete. □

If we augment G by adding k edges to obtain a triconnected planar graph G', then G contains at most k non-convex faces, since deleting any augmented edge joins two (convex) faces into one (possibly non-convex) face.

In this chapter we consider the problem of augmenting a biconnected planar graph G to a triconnected planar graph G' by adding as few as possible edges. The problem of deciding in polynomial time whether adding $\leq K$ edges to a biconnected planar graph can make the augmented graph planar and triconnected remains as an interesting open problem. We present an algorithm, that adds a number of edges that is bounded by only a constant times optimal, and the increase of the maximum degree is only an additive constant from an existential lower bound. Recall the definitions in Section 2.4 concerning the triconnected components and the SPQR-tree T_{SPQR}. As described in Chapter 3, we delete the Q-nodes from T_{SPQR}, and assume that every vertex v of degree 2 is represented by an S-node in T_{SPQR}. Let T_{SPQR} be rooted at an arbitrary node. The goal is to add a minimum number of edges between the triconnected components, which are leaves in T_{SPQR}.

To preserve planarity, edge can only be added between two triconnected components if they share a face in a planar embedding of G. Let b_l be a node in T_{SPQR}, and let $B_l = \mathit{skeleton}(b_l)$. If b_l is a P-node, B_l is a bond; if b_l is an S-node, B_l is a cycle, and if b_l is an R-node, B_l is a triconnected graph. Cycles and triconnected graphs have a unique embedding.

Assume $edge(b_l)$ belongs to faces F' and F'' in $skeleton(parent(b_l))$, then for every augmenting edge (v', w') with $v' \in pertinent(b_l)$ and $w' \in G-$ $pertinent(b_l)$, v' and w' share F' or F'' in a planar embedding of G. Suppose there are x edges in F' and y edges in F'' between vertices in $pertinent(b_l)$ and vertices in $G-$ $pertinent(b_l)$. The following lemma is crucial in our algorithm.

Lemma 5.1.2 *We can change the augmentation edges while preserving triconnectivity such that there are at most 2 augmentation edges in F' and at most 2 augmentation edges in F'' between $pertinent(b_l)$ and $G-$ $pertinent(b_l)$.*

Proof: By induction on the depth of the subtree of the SPQR-tree, rooted at b_l. Let $B_l = skeleton(b_l)$. If the depth $= 0$, then b_l is a leaf, and B_l is either a vertex v of degree 2, or a triconnected graph. Since only one augmenting edge from B_l to $G - B_l$ is sufficient to triconnect B_l, the lemma follows.

Let now b_l be an arbitrary vertex in the SPQR-tree, and assume the lemma holds for all triconnected components B_i with b_i in the subtree of T_{SPQR}, rooted at b_l. Assume there are $k \geq 3$ edges between $pertinent(b_l)$ and $G-$ $pertinent(b_l)$ in face F', with k odd. (A similar argument follows when k is even, and for face F''.) We prove that instead of adding k edges between $pertinent(b_l)$ and $G-$ $pertinent(b_l)$, we can add $k-1$ edges between vertices of $pertinent(b_l)$ and one between $pertinent(b_l)$ and $G-$ $pertinent(b_l)$ in F' while preserving the triconnectivity.

Let $v_1, \ldots, v_k$ be the vertices (in clockwise order) from $pertinent(b_l)$, which must get an incident edge in F_1 to $G-$ $pertinent(b_l)$. Let B_i be the triconnected component corresponding to v_i (for $1 \leq i \leq k$), which must get an augmenting edge to $G-$ $pertinent(b_l)$. See Figure 5.1(a). Let $e_i = edge(b_i)$, for $1 \leq i \leq k$. Assume edges $e_{i_j} \ldots, e_{i_{j+1}-1}$ also share face F_j in B_l. Let there be k' such faces, then $i_1 = 1$ and $i_{k'+1} - 1 = k$. Inspect the sequence of vertices $v_{i_j}, \ldots, v_{i_{j+1}-1}$. For $i_j \leq \alpha < i_{j+1} - 2$ we do: if $B_\alpha = B_{\alpha+1}$, then the edge $(v_{\alpha+1}, v_{\alpha+2})$ is added and we union the components $B_{\alpha+1}$ and $B_{\alpha+2}$ into one triconnected component. (By induction hypothesis $B_{\alpha+1} \neq B_{\alpha+2}$.) See Figure 5.1(b). Let (after renumbering) $v_{i'_j}, \ldots, v_{i'_{j+1}-1}$ be the vertices among $v_{i_j}, \ldots, v_{i_{j+1}-1}$, which must still get an augmenting edge (numbered from v_{i_j} to $v_{i_{j+1}-1}$). The underlying idea is that after these unions, only $B_{i'_{j+1}-1}$ must get at most two outgoing edges, the other components $B_{i'_j}, \ldots, B_{i'_{j+1}-2}$ must get one outgoing edge, for each $j, 1 \leq j \leq k'$.

We now add the augmentation edges $(v_{i'_j}, v_{i'_{j+1}-1}), (v_{i'_j+1}, v_{i'_{j+1}-2})$, etc., until one or two vertices remain, which must get an augmenting edge. By swapping the triconnected components $B_{i'_j}, \ldots, B_{i'_{j+1}-1}$ these edges can be added in face F_j. If $i'_{j+1} - 1 - i'_j$ is even, then one vertex, with index $\frac{i'_{j+1}-1+i'_j}{2}$, does not receive an augmenting edge, otherwise two vertices, with indices $\lfloor \frac{i'_{j+1}-1+i'_j}{2} \rfloor$ and $\lceil \frac{i'_{j+1}-1+i'_j}{2} \rceil$, do not receive an augmenting edge. Notice that the components $B_{i'_j}, \ldots, B_{i'_{j+1}-1}$ become one triconnected graph after adding these edges (see Figure 5.1(c)).

Let $v_1, \ldots, v_p$ $(p \leq k')$ be the vertices, in clockwise order around F', which must get an edge to $G-$ $pertinent(b_l)$. If for any i, $1 \leq i < p$, e_i and e_{i+1} are on the

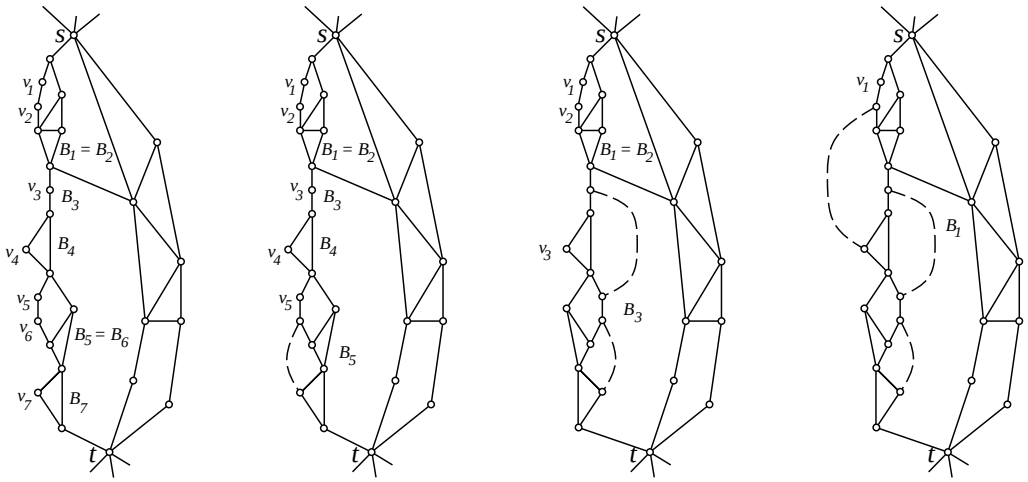


Figure 5.1: Example for the proof of Theorem 5.1.2.

common face F_j initially, then we add (v_{i+1}, v_{i+2}) in face F'. Let (after renumbering) $v_1, \ldots, v_{p'}$ $(p' \leq p)$ be the vertices which must get an augmenting edge. Notice that p' is odd, since k was assumed to be odd, and by every added edge e, two vertices of $pertinent(b_l)$ do not need an edge to $G-$ $pertinent(b_l)$. We add the edges (v_{2i}, v_{2i+1}) in F', with $1 \leq i \leq \frac{p'-1}{2}$. v_1 is the only remaining vertex among $v_1, \ldots, v_k$, which must get an augmenting edge to $G-$ $pertinent(b_l)$. See Figure 5.1(d).

Notice that every vertex v_i $(1 \leq i \leq k)$ receives one augmenting edge. Observe also that if $e_{i_j}, \ldots, e_{i_{j+1}-1}$ were on a common face before the augmentation, then there is a vertex $v_x, i_j \leq x \leq i_{j+1} - 1$, having an edge to a vertex $v_y, y < i_j$ or $y > i_{j+1} - 1$. This proves the triconnectivity. □

5.2 An Approximation Algorithm for the Planar Triconnectivity Augmentation Algorithm

In Section 5.1 it is shown that for every node $b_i \in T_{SPQR}$, with $edge(b_i)$ belonging to faces F_{i_1} and F_{i_2} in $skeleton(parent(b_i))$, at most x augmentation edges in F_{i_1} and y augmentation edges in F_{i_2} between vertices of $pertinent(b_i)$ and $G-$ $pertinent(b_i)$ are necessary to achieve triconnectivity, with $0 \leq x, y \leq 2$, and $x + y > 0$. Notice that more optimal solutions are possible. Let $aug(b_i)$ denote the set of pairs of integers (x, y), with $0 \leq x, y \leq 2$, with the property that adding x edges in F_{i_1} and y edges in F_{i_2} between $pertinent(b_i)$ and $G-$ $pertinent(b_i)$ triconnects $pertinent(b_i)$ with $G-$ $pertinent(b_i)$. For all $(x_j, y_j) \in aug(b_i)$, $x_j + y_j$ is even, or for all $(x_j, y_j) \in aug(b_i)$,

$x_j + y_j$ is odd.

The global idea of the algorithm is as follows: we start at the leaves b_i of T_{SPQR}, and for each leaf b_i we compute $aug(b_i)$. $skeleton(b_i)$ of a leaf $b_i \in T_{SPQR}$ is either a vertex v with $deg(v) = 2$ or a triconnected graph. In both cases $skeleton(b_i)$ must receive one augmenting edge, thus $aug(b_i) = \{(1,0)\}$. Consider now nodes $b_i \in T_{SPQR}$, for which $aug(b_j)$ is computed, if b_j is a child of b_i. The computation of $aug(b_i)$ heavily depends on the type of node $b_i \in T_{SPQR}$. For each type of node b_i in T_{SPQR} we present an algorithm for computing $aug(b_i)$, using only the information of the sets $aug(b_j)$ of the children b_j of b_i in T_{SPQR}. After computing $aug(b_r)$ with $b_r = root(T_{SPQR})$, we fix one pair $(x,y) \in aug(b_r)$ for b_r. From this pair (x,y) the augmentation pairs (x_i, y_i) for the children b_i of b_r follow. We traverse the tree top-down and repeat this argument. Finally the pairs (x_l, y_l) for all leaves b_l in T_{SPQR} are computed. Using this information, the augmenting edges can be added to achieve triconnectivity. We will discuss this problem of computing $aug(b)$ in more detail when considering the different types of the nodes in T_{SPQR}. The algorithm can now be described at a high level as follows:

TRICONNECT(G);
  construct the SPQR-tree T_{SPQR} of G;
  remove the Q-nodes from T_{SPQR};
  root T_{SPQR} at an arbitrary vertex b_r;
  for every leaf b_i **do** $aug(b_i) := \{(1,0)\}$ **rof**;
  repeat
    let $b_i \in T_{SPQR}$ such that $aug(b_j)$ has been computed for all children b_j of b_i;
    compute $aug(b_i)$ as follows according to the type of b_i:
      S-node : SERIES(b_i);
      P-node : PARALLEL(b_i);
      R-node : RIGID(b_i)
  until $b_i = b_r$;
  traverse T_{SPQR} top-down and add the augmentation edges;
END TRICONNECT

We now only have to explain how the three algorithms SERIES(b_i), PARALLEL(b_i) and RIGID(b_i) work. Given the computed set $aug(b_i)$ it is quite easy to determine and add the augmentation edges in $O(skeleton(b_i))$ time while preserving planarity.

5.2.1 The Series Case

Let b be an S-node, and $B = skeleton(b)$. Then B is a cycle. Let $b_1, \ldots, b_k$ be the children of b in T_{SPQR}, with $edge(b_i)$ in clockwise order when walking around the cycle B. Let $B_i = pertinent(b_i)$ with $1 \leq i \leq k$. We change $aug(b_i)$ by a pair $(x,y) \in aug(b_j)$, for which $x + y$ is maximal. Let s and t be the poles of B. Assume s and t belong to F_{left} and F_{right} in $skeleton(parent(b))$. The idea is to add alternatingly edges in F_{left} and F_{right} such that if there is an edge added between a

vertex in B_i and a vertex in B_j $(j > i)$ in face F_{left}, then there is an edge added in F_{right} between a vertex of B_x and B_y with $i \leq x \leq j$ and $y < i$ or $y > j$. Hereto we use counters n_{left} and n_{right}. These numbers denote the number of vertices of $B_1, \ldots, B_{i-1}$, which must get an augmenting edge to $G - \{B_1, \ldots, B_{i-1}\}$ in F_{left} and F_{right}, respectively. The following algorithm makes this idea to compute *one* pair of $aug(b)$ more precise. (Later we show how to compute the complete set $aug(b)$.) Let b have children $b_1, \ldots, b_k$ in order as described above.

SERIES(b)
  for $i := 1$ **to** k **do**
    let (x_i, y_i) be this pair in $aug(b_i)$, for which $x_i + y_i$ is maximal
  rof;
  $n_{left} := x_1; n_{right} := y_1; i := 2;$
  while $i \leq k$ **do**
    $n_{left} := |n_{left} - y_i|;$
    $n_{right} := n_{right} + x_i;$
    $n'_{left} := 0; j := i;$
    while $n'_{left} < n_{left}$ **and** $j < k$ **do**
      $j := j + 1;$
      $n'_{left} := n'_{left} + x_j;$
      $n_{right} := n_{right} + y_j$
    od;
    $n_{left} := n'_{left} - n_{left};$
    while $n_{right} > 2$ **do** $n_{right} := n_{right} - 2$ **od**;
    $i := j + 1;$
    swap(*left*, *right*)
  od;
  $aug(b) := \{(n_{left}, n_{right})\};$
END SERIES

When the statement $n_{left} := |n_{left} - y_i|$ is executed, we assume that $|n_{left} - y_i|$ edges are added between vertices of B_i and $B_j, j < i$, which must receive augmenting edges. When the statement $n_{left} := n'_{left} - n_{left}$ is executed, we assume that n_{left} edges are added between vertices of $B_\alpha, \alpha \leq i$ and $B_\beta, i < \beta \leq j$. When $n_{right} := n_{right} - 2$ is executed, then edges are added between consecutive vertices in F_{right}, which must get an augmenting edge and do not belong to the same B_i. swap(*left*, *right*) means that F_{left} and F_{right} are swapped. Also n_{left} and n_{right} are swapped.

Lemma 5.2.1 *At the entry of every* **while***-loop:* $0 < n_{left} \leq 2$.

Proof: Initially $n_{left} = x_1$ and $0 < x_i \leq 2$, since $x_i \geq y_i$. During every outermost **while**-loop, we set $n_{right} := n_{right} + x_i$ and decrease n_{right} if necessary until $0 < n_{right} \leq 2$. □

Lemma 5.2.2 *pertinent*(b) *is triconnected after applying* SERIES(b).

Proof: We prove by induction that after step i of the outermost **while**-loop, the components $B_1, \ldots, B_j$ are triconnected when we add the additional edges with respect to $aug(b_j)$. Initially $i = 2$ and B_1 is triconnected by adding x_i edges in n_{left} and y_i edges in n_{right}.

Assume $B_1, \ldots, B_{i-1}$ are made triconnected by adding edges between vertices in $B_1, \ldots, B_{i-1}$, and by adding n_{left} edges in F_{left} and n_{right} edges in F_{right} to vertices in $G - \{B_1, \ldots, B_{i-1}\}$. In the remaining part of the proof, let n_{left} and n_{right} denote the value at the entry of the outermost **while**-loop. We add $\min\{y_i, n_{left}\}$ edges between vertices of B_i and $B_1, \ldots, B_{i-1}$. If $n_{left} = y_i$, then $j = i$, and $B_1, \ldots, B_i$ are triconnected, since there are y_i edges from B_i to $B_1, \ldots, B_{i-1}$, and there are $x_i + n_{right}$ edges (with $x_i + n_{right} > 0$) from $B_1, \ldots, B_i$ to $B_{i+1}, \ldots, B_k$.

Suppose $|n_{left} - y_i| > 0$, hence $|n_{left} - y_i|$ edges will go in F_{left} from $B_1, \ldots, B_i$ to $B_{i+1}, \ldots, B_j$. If $y_i > 0$, then also one edge will go from B_i to $B_1, \ldots, B_{i-1}$, yielding triconnected components $B_1, \ldots, B_j$. If $y_i = 0$, then n_{left} edges go from $B_1, \ldots, B_{i-1}$ to $B_{i+1}, \ldots, B_j$, since $n_{left} > 0$ by Lemma 5.2.1. Since $x_i > 0$, the following cases can occur in F_{right}: if $x_i + n_{right} \geq 3$, then an edge between B_i and $B_1, \ldots, B_{i-1}$ will be added, and $x_i + n_{right}$ decreases by 2. If $x_i + n_{right} + y_{i+1} + \ldots + y_j \geq 3$, then an edge between $B_1, \ldots, B_i$ and $B_{i+1}, \ldots, B_j$ is added. One or two edges will be added between $B_1, \ldots, B_j$ and $B_{j+1}, \ldots, B_k$. This implies that if $y_i = 0$, then also the components $B_1, \ldots, B_j$ are triconnected. □

Lemma 5.2.2 implies that (n_{left}, n_{right}) is a correct member of $aug(b)$. If $3 \leq n_{left} + n_{right} \leq 4$, then it is rather easy to add one additional edge between two vertices inside *pertinent*(b), such that $1 \leq n_{left} + n_{right} \leq 2$. This gives a second correct member of $aug(b)$. Observe that if $(x_1 + y_1) + (x_2 + y_2) + \ldots + (x_k + y_k)$ is even (odd), then $n_{left} + n_{right}$ is even (odd, respectively).

To compute the complete set $aug(b)$, the following is done. If there are children b_j of b_i, with $\{(2,2),(1,1)\}$ or $\{(2,1),(1,0)\} \subseteq aug(b_i)$, then SERIES$(b)$ is applied again, with (x_i, y_i) changed into $(1,1)$ or $(1,0)$, for exactly one child b_i of b. This leads to a complete and optimal set $aug(b)$, completing the following lemma:

Lemma 5.2.3 SERIES(b) *computes in linear time the set* $aug(b)$, *and* SERIES(b) *also delivers the corresponding sets with minimum number of edges, which must be added between* *pertinent*(b_i) *and* *pertinent*(b_j), *with* b_i *and* b_j *children of node* b *in* T_{SPQR}.

In Figure 5.2 an example of applying the algorithm SERIES(V') is given.

5.2.2 The Parallel Case

Let b be a P-node with children $b_1, \ldots, b_k$. Let $B_i =$ *pertinent*(b_i), with $1 \leq i \leq k$. b is a bond, i.e., a pair of vertices s, t, with $k+1$ parallel edges in between. (One virtual

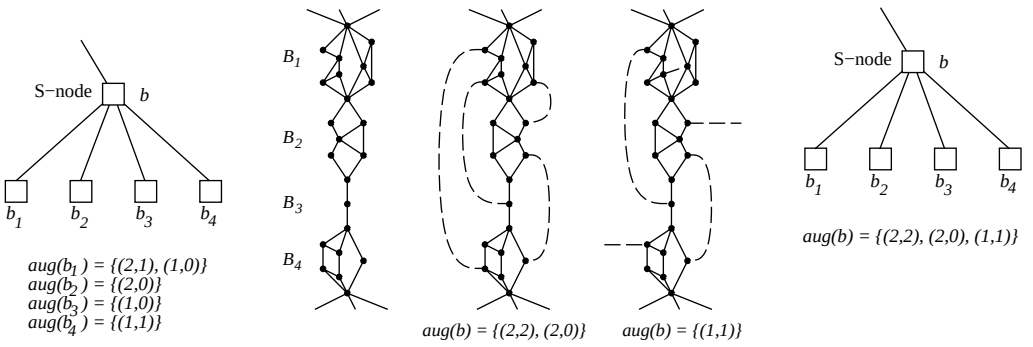


Figure 5.2: Applying the algorithm SERIES(b).

edge corresponds with *parent*(b).) We have to find an optimal order, say $b_1, \ldots, b_k$, such that adding a minimum number of edges between B_j and B_{j+1}, $1 \leq j < k$ makes $B_1, \ldots, B_k$ triconnected. If $(\alpha, \beta) \in aug(b_j)$, and $(\beta, \gamma) \in aug(b_{j+1})$, then adding β edges in between leads to a united component B' with $(\alpha, \gamma) \in aug(b')$. If $\beta = 0$, then $B_1, \ldots, B_j$ are only connected via s and t to $B_{j+1}, \ldots, B_k$. A third vertex-disjoint path from B_j to B_{j+1} cannot visit s and t, hence has to go via B_1 and B_k, implying that between every pair of components B_i, B_{i+1} $(i \neq j)$ there has to be an augmenting edge. Also B_1 and B_k must get augmenting edges to $G-$ *pertinent*(b). Such a place, where no augmenting edges between B_j and B_{j+1} occur, is called a *gap*. Between the components $B_1, \ldots, B_k$ at most one gap may occur to achieve triconnectivity. If $(s,t) \in G$, then there has to be a gap, say between B_j and B_{j+1}, and (s,t) is placed in the face in between.

We first place the triconnected components in one set K_j as follows:

if $(1,1) \in aug(b_i)$ **then** add b_i to K_1 **else**
  if $(2,2) \in aug(b_i)$ **then** add b_i to K_2 **else**
    if $(2,1) \in aug(b_i)$ **then** add b_i to K_3 **else**
      if $(1,0) \in aug(b_i)$ **then** add b_i to K_4 **else**
        if $(2,0) \in aug(b_i)$ **then** add b_i to K_5

Let $k_i = |K_i|$, $1 \leq i \leq 5$. If $k_4 + k_5 \leq 2$, then these components imply one gap; if $k_4 + k_5 > 2$, this gives rise to more than one gap, and we have to add extra edges to get only one gap. If $k_5 > 2$, we change $k_5 - 2$ times $(2,0)$ into $(2,1)$ and add the component to K_3. This implies $k_5 - 2$ extra edges, one for each component, added to K_3. Hence we assume $k_5 \leq 2$. If $k_4 > 2 - k_5$ then we change $k_4 - 2 + k_5$ times $(1,0)$ into $(1,1)$, and add the component to K_1 (and add the corresponding extra edges). After these two operations $k_4 + k_5 \leq 2$ holds. By adding augmentation edges between triconnected components of K_1 we can collapse them into one triconnected component B' with $aug(b') = \{(1,1)\}$. By adding edges between triconnected components of K_2 we get one united triconnected component

B'' with $aug(b'') = \{(2,2)\}$. Let the term (α,β)-component denote a component b_i with $(\alpha,\beta) \in aug(b_i)$, corresponding to the set K_i, to which b_i is added.

We place the components as follows: If $k_1 > 0$, we place all components of K_1 consecutively, say in the order $B_{j_1+1},\ldots,B_{j_1+k_1}$. Between each consecutive pair B_{j_1+i} and B_{j_1+i+1}, $1 \le i < k_1$, one augmenting edge is required. Also all components of K_2 are placed consecutively, say at $B_{j_2+1},\ldots,B_{j_2+k_2}$. Between each consecutive pair B_{j_2+i}, B_{j_2+i+1}, $1 \le i < k_2$, two augmenting edges are required.

If k_3 is even, we collapse the components of K_3 into one $(1,1)$-component, i.e., place them at $B_{j_1+k_1+1},\ldots,B_{j_1+k_1+k_3}$, such that between B_i and B_{i+1} alternatingly one or two augmenting edges are added. We can also collapse the components into one $(2,2)$-component, i.e., placing them at $B_{j_2+k_2+1},\ldots,B_{j_2+k_2+k_3}$. If k_3 is odd, we can collapse them into one $(2,1)$-component, thereby connecting the $(1,1)$-components with the $(2,2)$-components, i.e., at places $B_{j_1+k_1+1},\ldots,B_{j_1+k_1+k_3}$, and set $j_2 = j_1+k_1+k_3$. If $k_4 = 1$, we place the component of K_4 before the components of K_1, i.e., at place j_1. If $k_5 = 1$, we place the component of K_5 at place k_2+j_2+1. If $k_4 = 2$, we place the two components of K_4 between two components of K_1. If $k_5 = 2$, we place the two components of K_5 between two components of K_2.

Distinguishing the different values of $k_1,\ldots,k_5$ leads to the following algorithm:

PARALLEL(b);
  if $k_4 = 2$ **then**
    collapse the components of K_4 into one $(1,1)$-component and add it to K_1;
  if $k_5 = 2$ **then**
    collapse the components of K_5 into one $(2,2)$-component and add it to K_2;
  if $k_4 = 1$ **and** $k_5 = 1$ **then**
    collapse the components of K_4 and K_5 into one $(2,1)$-component and add it to K_3;
  if k_3 is odd **then**
    collapse all components of K_3 into one $(2,1)$-component; $k_3 := 1$
  else
    if $k_2 > 0$ **then**
      collapse all components of K_3 into one $(2,2)$-component and add it to K_2
    else
      collapse all components of K_3 into one $(1,1)$-component and add it to K_1;
    $k_3 := 0$;
  do depending on the values of k_4 and k_5 the following:
    $k_4 = 1$: **if** $k_2 > 0$ or $k_3 > 0$ **then** $aug(b) := \{(2,0)\}$ **else** $\{(1,0)\}$;
    $k_5 = 1$: **if** $k_1 > 0$ or $k_3 > 0$ **then** $aug(b) := \{(1,0)\}$ **else** $\{(2,0)\}$;
    otherwise : **if** $k_3 = 1$ **or** ($k_1 > 0$ **and** $k_2 > 0$) **then** $aug(b) := \{(2,1)\}$
      else if $k_1 > 0$ **then** $aug(b) := \{(1,1)\}$ **else** $\{(2,2)\}$
  od;
END PARALLEL

More optimal solutions are possible: If there was a component b_i with $(2,1)$ and $(1,0) \in aug(b_i)$, then we can change $(2,1)$ into $(1,0)$. Otherwise, if there is a

component b_i with $(2,2)$ and $(2,0) \in aug(b_i)$, then we change $(2,2)$ into $(2,0)$. This leads to other optimal pairs for $aug(b)$, i.e., this implies that $|aug(b)| \geq 2$. If for all $(x,y) \in aug(b)$, $x > 0$ and $y > 0$ and there is no gap, i.e., $k_4 = k_5 = 0$, then we have to inspect the case when edge $(s,t) \in G$. In this case, one extra edge is required, if there is a component with $(1,l) \in aug(b_i)$ for some child b_i of b, otherwise two extra edges are required. The following lemma can now be verified:

Lemma 5.2.4 *Algorithm* PARALLEL(b) *computes the correct set* $aug(b)$ *for P-node* b.

5.2.3 The Rigid Case

The case in which b is an R-node is more difficult. Let $B = skeleton(b)$. Assume b has children $b_1, \ldots, b_k$ in T_{SPQR}. Let $B_i = pertinent(b_i)$, for $1 \leq i \leq k$. Let $e_i = edge(b_i)$, belonging to faces F_{i_1} and F_{i_2} in B. If $(\alpha, \beta) \in aug(b_i)$ and $(\beta, \gamma) \in aug(b_j)$, then β augmenting edges can be added between vertices B_i and B_j, if they share a face. If $\beta = 2$ and there are two components b_{j_1}, b_{j_2}, with $(1,l) \in aug(b_{j_1})$, $aug(b_{j_2})$, then we can also add an edge between a vertex of B_i and B_{j_1} and between a vertex of B_i and B_{j_2}. An added edge (v_i, v_j) is called a *matching edge*, if both v_i and v_j must get an incident augmenting edge to admit triconnectivity. If only one of them must receive an augmenting edge, then (v_i, v_j) is called an *extra edge*. The problem of constructing the set $aug(b)$ can now be described as matching vertices with each other, corresponding to the sets $aug(b_i)$ $(1 \leq i \leq k)$. To this end one pair $(x_i, y_i) \in aug(b_i)$ is fixed for each child b_i of b. x_i and $y_i = 1$ or 2, and this is denoted by a **1** of b_i or **2** of b_i. Matching a **1** of b_i with a **2** of b_j means that x_i (or y_i) $= 1$, and x_j (or y_j) $= 2$, and we can add augmenting edges between the corresponding vertices of b_i and b_j to make the graph triconnected. The vertex or vertices of B_i, which must get an edge to $G - B_i$, are denoted by v_i (in the case of one augmenting edge) or by v_{i_1} and v_{i_2} (in the case of two augmenting edges). The number of matching and extra edges is computed as follows:

1. Fix one pair $(x_i, y_i) \in aug(b_i)$, $1 \leq i \leq k$. Assign x_i to F_{i_1} and y_i to F_{i_2}, where F_{i_1} and F_{i_2} are the faces to which $edge(e_i)$ belongs in B. Hence there must be x_i vertices of $pertinent(b_i)$ in F_{i_1} and y_i of $pertinent(b_i)$ in F_{i_2}, which must get an augmenting edge to $G-$ $pertinent(b_i)$.

2. In a face, it is possible to match:

   (a) a **1** with a **1**: one matching edge

   (b) a **1** with a **2**: one matching edge, one extra edge

   (c) a **2** with a **2** or with two **1**'s: two matching edges

   (d) three **2**'s with each other: three matching edges

For each unmatched **1**, one extra edge, and for each unmatched **2**, two extra edges are required. The problem now is to assign the **1**'s and **2**'s to the faces in such a way, that a minimum number of extra edges is required. An interesting, but still open question is to decide whether this problem can be solved in polynomial time. Let $e = edge(b)$, and let F_{left} and F_{right} be the left and right face of e in the embedding of $skeleton(parent(b))$, if $b \neq root(T_{SPQR})$. At least one edge has to go from B to $G-B$. If all elements in F_{left} or F_{right} are matched, then by deleting one augmenting edge in F_{left} or F_{right}, there are two vertices which can get an edge to $G-B$. If there is one unmatched vertex $v \in F_{left}$ or F_{right}, then an edge from v to $G-B$ can be added. Using these observations, the set $aug(b)$ can be computed.

In Figure 5.3 an input graph and an augmentation are given. Assume we fix an arbitrary pair $(x_i, y_i) \in aug(b_i)$ for each b_i. In the optimal solution at least $\frac{1}{2}\sum_{1\leq i\leq k}(x_i + y_i)$ augmentation edges are required. Let us assign now x_i to F_{i_1} and y_i F_{i_2}, and add x_i augmentation edges in F_{i_1} and y_i augmentation edges in F_{i_2} to vertices of $G - B_i$. This can be done while preserving planarity, and we add $\sum_{1\leq i\leq k}(x_i + y_i)$ augmenting edges, i.e., at most two times the minimum number. This leads to the following lemma:

Lemma 5.2.5 *There is a linear time algorithm to augment a biconnected planar graph to a triconnected planar graph by adding at most two times the minimum required number of edges.*

In the remaining part of this section we present an algorithm, working in linear time, which adds at most $\frac{3}{2}$ times the minimum number of augmenting edges.

If $aug(b_i) = \{(1,1)\}$ or $\{(2,2)\}$, then in both cases the same number is assigned to F_{i_1} and F_{i_2}. If $|aug(b_i)| \geq 2$ or $aug(b_i) = \{(\alpha, \beta)\}$, with $\alpha \neq \beta$, then different numbers can be assigned to F_{i_1} and F_{i_2}. The idea is to consider this component b_i, with $(2, \beta) \in aug(b_i)$, and add 2 augmenting edges from $pertinent(b_i)$ in either F_{i_1} or F_{i_2} to two vertices of $G-$ $pertinent(b_i)$, which must get an augmenting edge.

To this end we introduce for each face F two sets, $2set(F)$ and $1set(F)$. Let for each $b_i, (\alpha_i, \beta_i) \in aug(b_i)$ be this pair, for which $\alpha_i + \beta_i$ is maximum. We insert b_i in $\alpha_i set(F_{i_1})$ and $\alpha_i set(F_{i_2})$, if $\alpha_i > 0$. We insert b_i in $\beta_i set(F_{i_1})$ and $\beta_i set(F_{i_2})$, if $\beta_i > 0, \beta_i \neq \alpha_i$. Assume $|\alpha set(F_{i_1})| \geq 2$, and $b_i \in \alpha set(F_{i_1})$ and $b_j \in \alpha set(F_{i_1})$. Then we can add α edges between vertices of B_i and B_j in face F_{i_1}. Let $aug(b_i) = \{(\alpha, \beta)\}$, and let $edge(b_i) \in F_{i_1}$ and F_{i_2}. The α of b_i is used, thus b_i must be removed from $\alpha set(F_{i_1})$ and $\alpha set(F_{i_2})$. The β of b_i cannot be assigned to F_{i_1}, hence b_i must be deleted from $\beta set(F_{i_1})$. Let $assign(F)$ denote the set of vertices, which must get an augmenting edge in face F, then we add the corresponding α vertices of b_i to $assign(F_{i_1})$. The same is done for b_j.

The algorithm is now as follows: We first initialize the sets $1set(F)$ and $2set(F)$, and inspect for every $b_i \in 2set(F)$, whether we can add 2 edges between $pertinent(b_i)$ and two vertices of $G-$ $pertinent(b_i)$, which must get an augmenting edges. In other words, we try to match a **2** with a **2** or a **2** with two **1**'s. We can describe this as given in RIGIDHIGH.

RIGIDHIGH(b);
  for all faces F **do** initialize $1set(F)$ and $2set(F)$ to $\emptyset$ **rof**;
  for all b_i **do**
    let $(\alpha_i, \beta_i) \in aug(b_i)$, with $\alpha_i + \beta_i$ maximum;
    INSERT($b_i, \alpha_i set(F_{i_1})$) and INSERT($b_i, \alpha_i set(F_{i_2})$), if $\alpha_i > 0$;
    INSERT($b_i, \beta_i set(F_{i_1})$) and INSERT($b_i, \beta_i set(F_{i_2})$), if $\beta_i \neq \alpha_i$ and $\beta_i > 0$
  rof;
  for all faces F with $|2set(F)| \geq 2$ **do**
    for all $b_i \in 2set(F)$ **do**
      add corresponding vertices v_{i_1} and v_{i_2} to $assign(F)$;
      update the $1set$'s and $2set$'s
    rof
  rof;
  for all faces F with $|2set(F)| = 1$ **and** $|1set(F)| \geq 2$ **do**
    let $2set(F) = \{b_i\}$, and $b_j, b_k \in 1set(F)$ with $j \neq i$ and $k \neq i$;
    add corresponding vertices v_{i_1}, v_{i_2}, v_j and v_k to $assign(F)$;
    update the $1set$'s and $2set$'s
  rof;
END RIGIDHIGH

Suppose that after RIGIDHIGH(b), a **2** of b_i is not matched, i.e., $b_i \in 2set(F_{i_1})$ and $b_i \in 2set(F_{i_2})$. This implies that there is at most one vertex in F_{i_1} (or in F_{i_2}) to which we can add an edge from $pertinent(b_i)$. This gives rise to at least one extra edge for b_i. However, if initially $|aug(b_i)| \geq 2$, then we change $aug(b_i)$ into $\{(1,0)\}$ or $\{(1,1)\}$, if this pair belongs to $aug(b_i)$. After this changing, we have to add at most one edge from $pertinent(b_i)$ in F_{i_1} or F_{i_2}. We update the corresponding $2set(F_{i_1}), 1set(F_{i_1}), 2set(F_{i_2})$ and $1set(F_{i_2})$. The aim is to match a **2** with a **1**, otherwise a **1** is matched with a **1**, until no matching is possible anymore.

RIGIDLOW(b);
  while not all faces F have $|2set(F)| + |1set(F)| < 2$ **do**
    if $|2set(F)| = 1$ **and** $|1set(F)| = 1$ **then**
      let $2set(F) = \{b_i\}$, and $b_j \in 1set(F)$ with $j \neq i$;
      add corresponding vertices v_{i_1}, v_{i_2} and v_j to $assign(F)$
    else
      let $b_i, b_j \in 1set(F), j \neq i$;
      add corresponding vertices v_i and v_j to $assign(F)$;
    update the $1set$'s and $2set$'s
  od;
  for each $2set(F) = \{b_i\}$ **or** $1set(F) = \{b_i\}$ **do**
    add corresponding vertices to $assign(F)$;
    update the $1set$'s and $2set$'s
  rof;
END RIGIDLOW

Let $assign(F) = \{v_1, \ldots, v_k\}$, in clockwise order around face F. Adding the augmentation edges in F without introducing crossings is done as follows: if a pair of vertices v_i, v_{i+1} belong to the same B_j, with $parent(b_j) = b$, then we add (v_{i+1}, v_{i+2}) in F. Let after renumbering $v_1, \ldots, v_{k'}$ be the remaining vertices in clockwise order around F which must receive an augmenting edge (with $k' \leq k$). Then we add the edges (v_{2i-1}, v_{2i}), for $1 \leq i \leq \lfloor \frac{k'}{2} \rfloor$. If k' is odd, then one extra edge is required for vertex $v_{k'}$. If $k = 2$ and v_1, v_2 belong to the same B_i, then an extra edge is required for both v_1 and v_2. Applying this approach leads to a maximum number of augmentation edges in F, given the vertices in $assign(F)$.

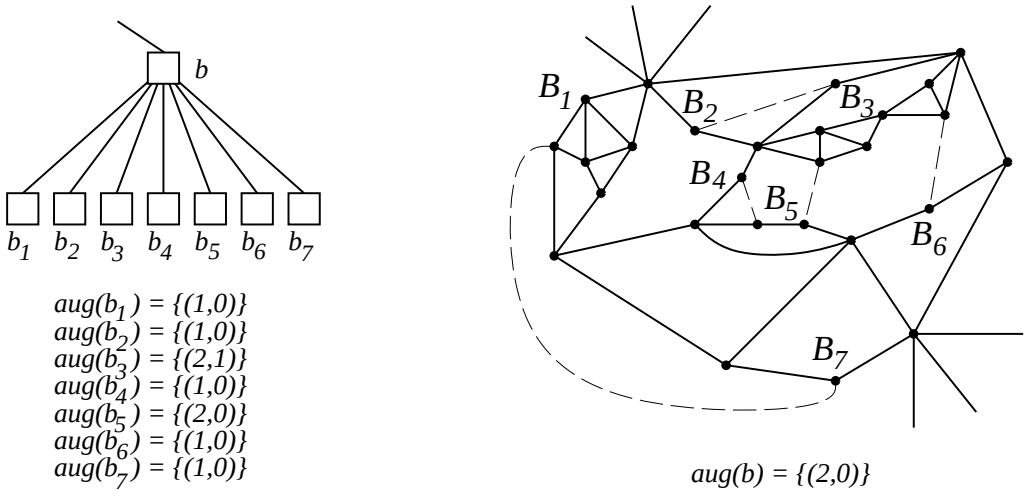


Figure 5.3: Applying RIGID(b).

Let RIGID(b) denote the algorithm, consisting of RIGIDHIGH(b), RIGIDLOW(b), and adding the corresponding edges in the faces of $skeleton(b)$. To prove the approximation ratio of the algorithm RIGID(b), we use the following lemma:

Lemma 5.2.6 *We can assign every extra edge, required by applying* RIGID(b) *and not required in the optimal solution, to a unique vertex, endpoint of a matching edge in* RIGID(b).

Proof: Inspect a b_i with an unmatched **2** in RIGID(b), which is matched in the optimal solution. If the **2** of b_i is matched with a **2** of b_j, then in RIGID(b) the **2** of b_j is matched in another face, hence we assign the two extra edges of b_i to the two corresponding matched vertices v_{j_1}, v_{j_2} of b_j. If the **2** of b_i is matched with two **1**'s, say from b_j and b_k, then in RIGID(b) the **1** of b_j and the **1** of b_k are matched in other faces, hence we assign the two extra edges of b_i to the corresponding vertices v_j and v_k.

Inspect a b_i with a **2**. Suppose this **2** is matched with a **1** in RIGID(b). If this **2** of b_i was matched with a **2** of b_j in the optimal solution, then the **2** of b_j is matched in another face. We assign one extra edge of b_i to one vertex v_{j_1} of b_j. Suppose that the **2** of b_i was matched with two **1**'s in RIGID(b), say from b_j and b_k. b_j and b_k share F initially. They cannot be both unmatched now, because then we can match b_j with b_k in face F. Assume b_j is matched, then we assign the extra edge of b_i to v_j. Assume finally that a **1** of b_i is unmatched, while it was matched with a **1**, say of b_j, in the optimal solution. But now again b_j is matched, otherwise the **1** of b_j was matched with b_i, hence the extra edge of b_i is assigned to v_j. □

Lemma 5.2.7 RIGID(b) *adds at most* $\frac{3}{2}$ *times the minimum number of edges to obtain triconnectivity.*

Proof: We can assign every extra edge to an endpoint of a matching edge. If there are k matching edges, then there are at most $2k$ extra edges. This implies $2k$ vertices, endpoints of matching edges and $2k$ vertices, endpoints of extra edges, which must get an augmenting edge. In the optimal solution, each of these $4k$ vertices corresponds to the endpoint of a matching or extra edge, hence there are at least $2k$ edges are added in the optimal solution. Since we now have $2k$ extra edges and k matching edges, and in the optimal solution at least $2k$ edges added, this yields a performance ratio of $\frac{3}{2}$. □

Lemma 5.2.8 *Algorithm* RIGID(b) *can be implemented to run in* $O(n_B)$ *time, with* n_B *the number of vertices in skeleton*(b).

Proof: Initializing and constructing the sets $1set(F)$ and $2set(F)$ can be done in $O(n_B)$ time, because we can construct the embedding of the planar graph in $O(n_B)$ time [12], and the time for updating $1set(F)$ and $2set(F)$ is $O(1)$ for every child b_i of b in T_{SPQR}. Each b_i belongs to at most 4 sets. We use cross-pointers between b_i and $\alpha set(F)$, if $b_i \in \alpha set(F)$. Using these pointers we can update the $1set$'s and $2set$'s in constant time, after chosing b_i and b_j for a matching. After constructing the sets $assign(F)$ we can compute the matching edges in $O(|assign(F)|)$ time, for each face F. □

The lemmas of this section leads to the main result of this section:

Theorem 5.2.9 *The algorithm* TRICONNECT(G) *augments in linear time a biconnected planar graph to a triconnected planar graph, which adds at most* $\frac{3}{2}$ *times the minimum required number of edges.*

5.3 Triconnecting While Minimizing The Maximum Degree

In this section we focus our attention on the problem of triconnecting a planar graph while minimizing the maximum degree. Let us first consider the increase of the degree of every vertex in the algorithm TRICONNECT(G) as described in Section 5.2. Again we distinguish matching edges and extra edges as defined in Section 5.2. If (v_i, v_j) is an extra edge, then v_i gets an extra incident edge, while v_i did not need an extra edge to achieve triconnectivity. Indeed, v_i might get several incident extra edges. (Notice that every vertex gets at most one incident matching edge.)

Let us be more precies about this. We inspect the different types of node b in the SPQR-tree: If b is an S-node, then no extra edge is added. If b is a P-node, then an extra edge between triconnected components B_j and B_{j+1} is added, if in the optimal permutation β of B_j and α of B_{j+1} is assigned to F_j, $\beta \neq \alpha$, with F_j the face between B_j and B_{j+1}. If $\beta = 1$ and $\alpha = 1$, then the increase of $deg(v)$ of the corresponding vertices is one. If $\beta = 2$, then b_j is not a leaf in T_{SPQR} and, hence, has descendants, say b_{j_1} and b_{j_2}. Let $B_{j_1} = pertinent(b_{j_1})$ and $B_{j_2} = pertinent(b_{j_2})$. They both must get an edge to $G - B_j$. But we can add an edge between B_{j_1} and B_{j_2} in face F_j, and add an edge from arbitrarily B_{j_1} or B_{j_2} to $G - B_j$. If B_{j_1} and B_{j_2} are just vertices of degree 2, then $deg(v_{j_1})$ or $deg(v_{j_2})$ increases by two. A similar argument follows when $\alpha = 2$. Hence the increase for all vertices, receiving incident edges in PARALLEL(b) is at most 2. In the remaining part of this section we focus our attention on the case in which b is an R-node.

Let b be an R-node. Extra edges in face F are required, when $|2set(F)| = 1$ or $|1set(F)| = 1$, say $2set(F) = \{b_j\}$. This implies again that b_j is not a leaf in T_{SPQR}. Let b_{j_1} and b_{j_2} be descendants of b_j such that B_{j_1} and B_{j_2} must get edges to $G - B_j$. We can add an edge between B_{j_1} and B_{j_2} in face F, and an edge from arbitrarily B_{j_1} or B_{j_2} to $G - B_j$ makes B_j triconnected. If F is a triangle in B on the vertices u, v, w, and u and v are the poles of B_j, then the augmenting edge from B_{i_1} or B_{i_2} to $G - B_j$ goes to w. w receives at most one extra edge in face F. Since w belongs to $deg(w)$ faces, w can receive $deg(w)$ augmenting edges. Since the initial graph is biconnected, $deg(w) \geq 2$. This completes the following lemma.

Lemma 5.3.1 TRICONNECT(G) *triconnects in linear time a biconnected planar graph G to a triconnected planar graph G' by adding at most $\frac{3}{2}$ times the minimum number of edges such that $\Delta(G') \leq 2\Delta(G)$.*

The idea in this section is to change the algorithm TRICONNECT(G) a little such that for the triconnected planar graph G', $\Delta(G') \leq \lceil \frac{3}{2}\Delta(G) \rceil$ holds. Hereto we only have to consider the case that b is an R-node, because in all other cases, every vertex receives at most two edges. Let b_j be a child of b, with $aug(b_j) = \{(\alpha_j, \beta_j)\}$. Let s_j, t_j be the poles of b_j, and assume that $(s_j, t_j) \in F_{j_1}$ and F_{j_2} in B. There must

always go at least one edge from B_j to $G - B_j$. We distinguish two cases for the extra edges:

1. In RIGID(b) a matching edge, say (v_j, v'_j), goes from B_j to $G-B_j$. If $\alpha_j+\beta_j > 1$, then extra edges are required in B_j. But if $\alpha_j + \beta_j > 1$, then b_j is not a leaf in T_{SPQR}. Let b' be a descendant of b_j in T_{SPQR}, which must get an incident edge to $G-B_j$. It is not difficult to add this edge, say (v', w') such that $v' \in B'$ and $w' \in B_j - B'$. We do this for all vertices in B_i, which must get an augmenting edge to $G - B_j$. We can do this such that B_j becomes triconnected, because there is an edge (v_j, v'_j) to $G - B_j$.

   Hence there are no extra edges from B_j to $G - B_j$.

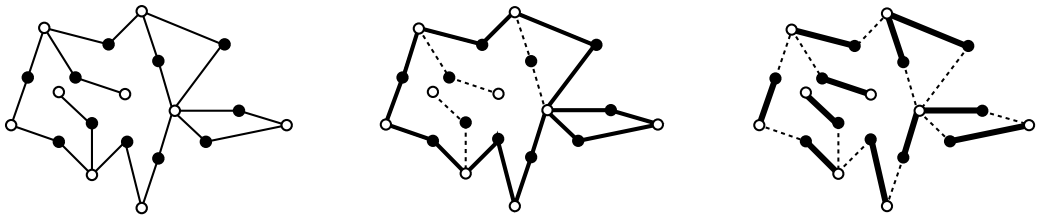
2. In RIGID(b) no matching edges are added between B_j and $G-B_j$. Then $\alpha_j+\beta_j$ extra edges between B_j and $G - B_j$ are required in TRICONNECT(G). Similar as described before, we add $\alpha_j + \beta_j - 1$ extra edges with both endpoints in B_j, such that adding one extra edge from B_j to $G - B_j$ makes B_j triconnected.

We apply this approach to all children of vertex b in T_{SPQR}. After this, there are children b_j of b in T_{SPQR}, which must receive one extra edge from B_j to $G-B_j$. Let F_{i_1} and F_{i_2} be the triangles, say on vertices s_j, t_j, v_{j_1} and s_j, t_j, v_{j_2}. v_{j_1} or v_{j_2} must get an augmenting edge. The aim is to the edges such that $deg(v_{j_1})$ and $deg(v_{j_2})$ do not increase too much. Hereto a graph H is constructed as follows:

Every face $F \in B$ is represented by an f-vertex v_F in H, and every b_i is represented by a b-vertex v_{b_i} in H, if B_i must get one augmenting edge from B_i to $G-B_i$. We add the edge (v_F, v_{b_i}) to H, iff $e_i \in F$ in the embedding of B. We delete all isolated f-vertices from H. This leads to a bipartite planar graph H on b- and f-vertices. We now compute a subset of edges $M \subseteq E_H$ such that every b-vertex b_j has one incident edge $(b_j, u_i) \in M$, and the f-vertex have as few as possible incident edges in M. The edge $(v_F, v_{b_i}) \in M$ implies that we add an edge from B_i to $G - B_i$ in face F. If F is a triangle in B, then there is only one vertex $v \in F$, with $v \notin B_i$, because the other two vertices in F are the poles of B_i.

M is constructed as follows: Using a simple modification of Eulers technique to find an Eulerian cycle in a graph, the *elementary cycles* C_{elem} are extracted from H. An elementary cycle is a cycle that uses each edge at most once. Thus $H - C_{elem}$ consists of paths P with disjoint begin- and endpoints (see Figure 5.4(b)). From every cycle of C_{elem} C_{elem} and every path P we add alternatingly one edge to M and one not. Recall that H is bipartite, and that b-vertices have degree 2. Hence for every vertex in C_{elem} and every internal vertex of a path P, one incident edge is in M and the other is not. But also for every b-vertex, exactly one incident edge is in M, hence satisfying the constraints (see Figure 5.4(c)). For every vertex $v \in H$, at most $\lceil \frac{deg_H(v)}{2} \rceil$ neighbors are in M. This observation completes the following theorem:

Theorem 5.3.2 *Algorithm* TRICONNECT(G) *augments a biconnected planar graph G to a triconnected planar graph G' in linear time by adding at most $\frac{3}{2}$ times the minimum number of edges, such that $\Delta(G') \leq \max\{2, \lceil \frac{3}{2}\Delta(G) \rceil\}$.*



(a) example of H with white and black vertices

(b) The elementary cycles and the disjoint paths

(c) The set M of edges.

Figure 5.4: The construction of H and M.

This bound with respect to the maximum degree matches the lower bound, as stated in the following theorem:

Theorem 5.3.3 *For every $\Delta > 3$ there exists a biconnected planar graph G with the property that for every triconnected planar graph G', $G \subseteq G'$: $\Delta(G') \geq \lceil \frac{3}{2}\Delta(G) \rceil$.*

Proof: Let $\Delta = \Delta(G) \geq 4$. Construct the graph G_Δ consisting of two vertices A and B, and $2\Delta - 1$ vertices $p_0, \ldots, p_{2\Delta-2}$. There is an edge (p_i, p_{i+1}) for $0 \leq i < 2\Delta - 2$. There are edges (A, p_i) and (B, p_i), for i even and $0 \leq i \leq 2\Delta - 2$. For Δ odd (implying $\Delta \geq 5$), a separate vertex C is inserted with edges (p_0, C) and $(C, p_{2\Delta-2})$. See Figure 5.5.

In every face F_i with vertices p_i, p_{i+1}, p_{i+2}, A and in every face F_i' with vertices p_i, p_{i+1}, p_{i+2}, B, one extra edge must be added (with i even). When we add an edge (p_i, p_{i+2}) in F_i, then an edge (p_{i+1}, B) must be added in F_i'. Since there are $\Delta - 1$ faces F_i, this means that the total increase of the degree of A and B is $\Delta - 1$. If Δ is odd, then an additional edge has to go from C to A or B. Hence the degree of A or B increases by at least $\lceil \frac{\Delta}{2} \rceil$. □

We will use the algorithm TRICONNECT in Chapter 6 to triangulate a planar graph while minimizing the maximum degree.

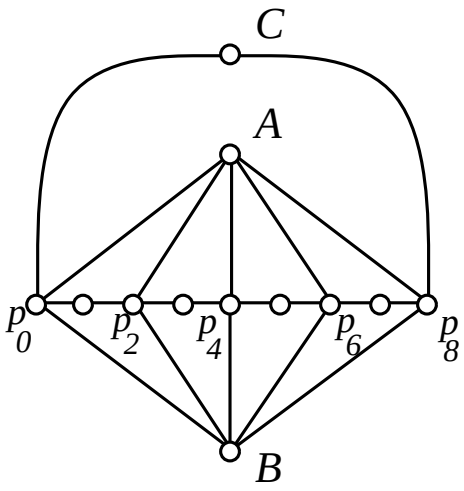


Figure 5.5: Illustration of Theorem 5.3.3.

Chapter 6

Triangulating Planar Graphs

In this chapter we consider the problem of triangulating planar graphs. A planar graph is triangular (or triangulated or maximal planar) when every face has exactly three vertices. If a planar graph is not triangular, then there is a face F having at least four different vertices, say v_1, v_2, v_3, v_4 in this order around the face. By planarity constraints it follows that $(v_1, v_3) \notin G$ or $(v_2, v_4) \notin G$. Adding the missing edge in face F and repeating this argument to all faces which are not triangles leads to a triangular planar graph. Simple and elegant linear time algorithms based on this idea are documented in the literature. In this section we outline two approaches, and we present a new algorithm for triangulating planar graphs, based on the canonical ordering introduced in Section 2.5. Assume G is a biconnected planar graph (otherwise we can use an augmentation algorithm, described in Chapter 4. Let an arbitrary planar embedding of G be given.

The first algorithm is due to Read [92], and modified by De Fraysseix such that it works in linear space. It can be described as follows: visit all vertices of G. For every pair of consecutive neighbors u, w of a current visited vertex v, add (u, w) to G if w is not adjacent to v in $adj(u)$. This gives a triangle on the vertices u, w, v. Applying this to all vertices leads to a triangulated planar graph G' which, however, may contain multiple edges. Let there be k edges (u, w) in G'. Removing one edge (u, w) leads to a face F with four vertices, say v, u, v', w, and by planarity, $(v, v') \notin G'$. Replacing (u, w) by (v, v') in F removes one multiple edge. The multiple edges can be detected by a bucketsort [1]. Replacing each edge requires constant time, hence this leads to a linear time, linear space triangulation algorithm.

Hagerup & Uhrig [44] changed this algorithm such that no multiple edges are introduced in G' at all. This is obtained by marking the neighbors of vertex v, which we are currently visiting in the algorithm. Process the faces incident on v. For each such face F with boundary vertices $v = u_1, u_2, \ldots, u_p$ we do: if $p = 3$, then F is a triangle. Otherwise, if u_3 is not marked, add an edge (u_1, u_3), mark u_3 and continue triangulating the face with boundary vertices $u_1, u_3, u_4, \ldots, u_p$. Since u_3 was not marked, $(u_1, u_3) \notin G$, and thus by adding (u_1, u_3), no multiple edge is introduced. If u_3 was marked, then $(u_1, u_3) \in G$, and by planarity constraints, $(u_2, u_4) \notin G$.

Add an edge (u_2, u_4), and continue triangulating the face with boundary vertices $u_1, u_2, u_4, u_5, \ldots, u_p$. When all incident faces of v are visited, we unmark the neighbors of v. After visiting all vertices of G, a planar triangular graph is obtained [44].

In this section another simple linear time triangulation algorithm is introduced. This algorithm has the interesting side-effect that it directly computes a canonical ordering for triangular planar graphs, as defined in [34]. For completeness, we restate here this theorem:

Theorem 6.0.4 *Let G be a triangular planar graph embedded in the plane with outerface u, v, w. There exists an ordering of the vertices $v_1 = u, v_2 = v, v_3, \ldots, v_n = w$ meeting the following requirements for every $k, 4 \leq k \leq n$.*

1. *The subgraph $G_{k-1} \subseteq G$ induced by $v_1, v_2, \ldots, v_{k-1}$ is biconnected, and the boundary of its outerface is a cycle C_{k-1} containing the edge (u, v);*

2. *v_k is on the outerface of G_k, and its neighbors in G_{k-1} form an (at least 2-element) subinterval on the path $C_{k-1} - (u, v)$.*

Assume G is biconnected and let a planar embedding of G be given. Following the idea of Theorem 6.0.4 we start with edge (v_1, v_2) and add a vertex v_k in each step $k, 3 \leq k \leq n$, such that we can construct G_k with all interior faces triangulated. To this end we search for a vertex v_k such that all neighbors v_l with $l < k$ are on C_{k-1}. We call the edges (v_α, v_β) with $\alpha < k \leq \beta$ *outgoing edges* in step k, i.e., the outgoing edges are edges from vertices on the outerface C_{k-1} to $G - G_{k-1}$. Let i be as small and j be as high as possible such that (v_k, c_i) and $(v_k, c_j) \in G$, with c_i and c_j belonging to the outerface $C_{k-1} : c_1, c_2, \ldots, c_r$. If $j > i$, then all vertices $c_{i+1}, \ldots, c_{j-1}$ may not have outgoing edges to vertices v_l with $l > k$. We add edges from $c_{i+1}, \ldots, c_{j-1}$ to v_k, if these edges are not present already. If $j = i$, then we assume that v_k is consecutive to c_{i-1} or c_{i+1} in $adj(c_i)$. We can add an edge from c_{i-1} or c_{i+1} to v_k while preserving planarity. If all interior faces in G_{k-1} were triangulated, then it follows that all faces in G_k are also triangulated. We call c_i the *leftvertex* of v_k and c_j the *rightvertex* of v_k. The vertices $c_{i+1}, \ldots, c_{j-1}$ are called *internal vertices*. Assume v_k has edges to vertices v_l with $l > k$. We call the vertex $v_l, l > k$, with v_l consecutive to c_i (to c_j) in $adj(v_k)$ *leftup*(v_k) (*rightup*(v_k), respectively). If v_k has only one neighbor v_l with $l > k$, then *leftup*(v_k) = *rightup*(v_k). If v_k has no neighbors v_l with $l > k$, then *leftup*$(v_k) = nil$ and *rightup*$(v_k) = nil$.

For the algorithm we introduce a linked list, called *readylist*, containing these vertices, which can be the next one in the ordering of triangulating. We also introduce two variables for each vertex v_k in G: *old*(v_k), denoting the number of vertices v_l with $l < k$, and *visit*(v_k). We increase *visit*(v_k) by one, if there is a pair of vertices c_α, c_β on C_k, with *rightup*$(c_\alpha) = v_k$, *leftup*$(c_\beta) = v_k$ and all vertices c_γ on C_k, $\alpha < \gamma < \beta$, do not have outgoing edges. We claim that if this is the case, then c_α

and c_β are consecutive in $adj(v_k)$. Suppose not, thus there is at least one neighbor of v_k, say w, between c_α and c_β in $adj(v_k)$. w is not part of C_k, since all vertices c_γ on C_k, $\alpha < \gamma < \beta$, do not have outgoing edges. Hence there is no path from w to c_α than by going via v. This yields that $G - \{v\}$ is disconnected, which contradicts the fact that G is biconnected.

We use the variables $old(v)$ and $visit(v)$ to determine whether there is precisely one consecutive sequence of lower-numbered neighbors in $adj(v)$, which is the case when $old(v) = visit(v) + 1$. The algorithm CANONICALTRIANGULATE can now be described as follows:

```
CANONICALTRIANGULATE(G);
  initialize old(v) and visit(v) to 0 for all v ∈ G;
  start with an edge (v1, v2);
  increase old(v) by one, for all neighbors of v1 and for all neighbors of v2;
  add rightup(v1) and leftup(v2) to readylist;
  for k := 3 to n do
    delete arbitrary vertex vk from readylist;
    for all neighbors vl, l > k of vk do
      delete vl from readylist (if present); old(vl) := old(vl) + 1
    rof;
    let ci be the leftvertex and cj be the rightvertex of vk;
    if vk = leftup(ci) and i = j and ci ≠ v1 then j := i; i := i − 1 else
      if vk = rightup(ci) and i = j and cj ≠ v2 then i := j; j := j + 1;
    while old(ci) = deg(ci) and ci ≠ v1 do i := i − 1 od;
    while old(cj) = deg(cj) and cj ≠ v2 do j := j + 1 od;
    add edges from ci, ..., cj to vk (if not present already);
    let cl = rightup(ci) and cr = leftup(cj);
    if old(vk) = deg(vk) then
      if cl = cr then visit(cl) := visit(cl) + 1
    else
      if cl = leftup(vk) then visit(cl) := visit(cl) + 1;
      if cr = rightup(vk) then visit(cr) := visit(cr) + 1;
    if old(cl) = visit(cl) + 1 then add cl to readylist;
    if old(cr) = visit(cr) + 1 then add cr to readylist
  rof;
END CANONICALTRIANGULATE
```

See Figure 6.1 for an illustrating example. To prove that this algorithm indeed works, we need the following lemma:

Lemma 6.0.5 *readylist is not empty at the start of step* $k, 3 \leq k \leq n$.

Proof: The initial graph is biconnected, hence in every step k there are at least two vertices on C_{k-1}, having edges to vertices outside G_{k-1}. If there is a vertex

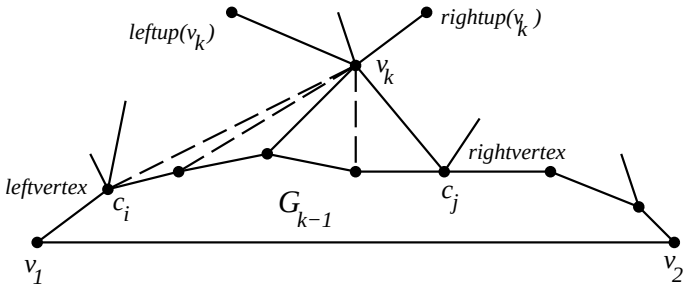


Figure 6.1: Triangulating a planar graph while computing a canonical ordering.

$v \in G - G_{k-1}$ with $v = \mathit{leftup}(c_\alpha)$ or $v = \mathit{rightup}(c_\alpha)$ for some c_α on C_{k-1}, and $\mathit{old}(v) = 1$, then by definition of *visit*, $\mathit{visit}(v) = 0$. Hence we can choose $v_k = v$. Assume further that for every vertex v, with $v = \mathit{leftup}(c_\alpha)$ or $v = \mathit{rightup}(c_\alpha)$ for some c_α on C_{k-1}, that $\mathit{old}(v) > 1$.

Consider now this pair of vertices c_α, c_β on C_{k-1} ($\beta > \alpha$) with $\beta - \alpha$ minimal, $v = \mathit{rightup}(c_\alpha)$ and $(c_\beta, v) \in G$, but c_β not consecutive to c_α in $\mathit{adj}(v)$. By biconnectivity of G it follows that there is at least one outgoing edge on C_{k-1} between the outgoing edges (c_α, v) and (c_β, v). Let (c_γ, v') be such an edge, with $v' = \mathit{leftup}(c_\gamma)$ and $\alpha < \gamma \leq \beta$. By definition $v' \neq v$. By planarity, v' has no neighbors c_δ on C_{k-1}, with $\delta \leq \alpha$ or $\delta > \beta$. We assumed $\mathit{old}(v') > 1$, thus v' has a neighbor c_δ, with $\alpha < \delta \leq \beta$. But by minimality of $\beta - \alpha$ it follows that all neighbors of v' on C_{k-1} form a consecutive sequence in $\mathit{adj}(v')$. Hence we can choose $v_k = v'$. □

Lemma 6.0.6 *The ordering in which the vertices are added is a canonical ordering.*

Proof: In every step k after the augmentation, v_k has edges to vertices $c_i, \ldots, c_j$ on C_{k-1}. If $i = j$, then by the algorithm $v_k = \mathit{leftup}(c_i)$ or $v_k = \mathit{rightup}(c_i)$, because only vertices, which are $\mathit{rightup}(c_i)$ or $\mathit{leftup}(c_j)$, are added. If $v_k = \mathit{leftup}(c_i)$ then $c_i \neq v_k$, and if $v_k = \mathit{rightup}(c_i)$, then $c_i \neq v_2$. Hence after the augmentation $j > i$ holds, and v_k has edges to $c_i, c_{i+1}, \ldots, c_j$. v_k becomes part of C_k and receives at least one edge to a vertex v_l with $l > k$, if $k < n$. This precisely corresponds to the canonical ordering of de Fraysseix, Pach & Pollack [34], as presented in Theorem 6.0.4. □

Theorem 6.0.7 *The algorithm* CanonicalTriangulate(G) *triangulates in linear time a biconnected planar graph, while computing a canonical ordering.*

Proof: The correctness of the algorithm and the canonical ordering follows from the previous lemmas. We only have to prove that it works in linear time. But this follows directly, by introducing a linked list for maintaining C_{k-1} in every step,

and by using the pointers *leftup*(v), *rightup*(v) and the counters *old*(v) and *visit*(v) for every vertex v. □

The algorithm CANONICALTRIANGULATE is a good and simple method for computing a canonical ordering of a graph while triangulating the graph. We can use this algorithm as a first step to draw a planar graph planar with straight lines on an $(n-2) \times (n-2)$ grid (see Section 10.2.1), or to compute a visibility representation (see Section 10.4).

Unfortunately, all triangulation algorithms mentioned above do not triangulate the planar graph such that the maximum degree is minimum. The remaining part of this paper is devoted to this problem. We show that the problem of deciding whether a biconnected planar graph can be triangulated such that the maximum degree is at most K is NP-complete. An approximation algorithm is presented, working only an additive constant from optimal, when the input graph is triconnected, and only an additive constant from an existential lower bound, when the input graph is biconnected.

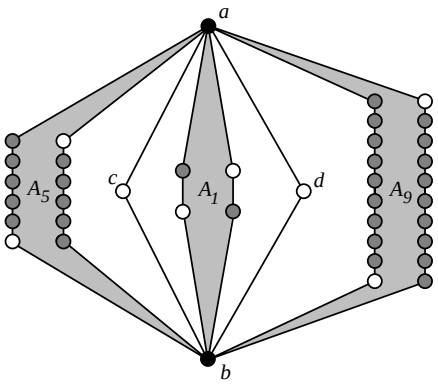
6.1 NP-completeness

In this section we prove the following theorem:

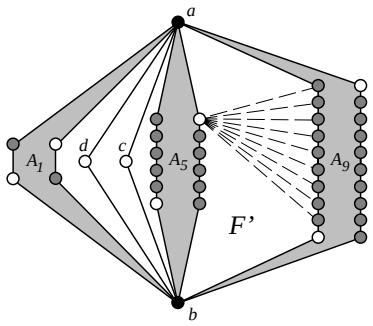
Theorem 6.1.1 *The problem of deciding whether a biconnected planar graph can be triangulated such that the maximum degree is $\leq K$ is NP-complete.*

Proof: The problem is in NP: guess $3n-6-m$ additional edges, and test in polynomial time whether G is planar and has maximum degree $\leq K$.

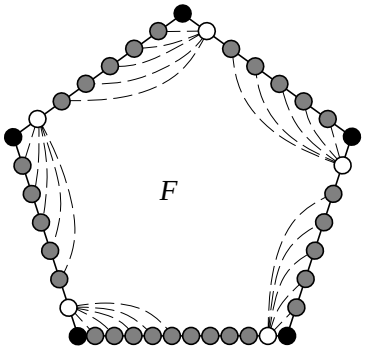
To prove the NP-hardness we use a reduction from the 3-coloring problem for triconnected planar graphs. It is well-known that deciding whether a planar graph can be colored with three colors such that every pair of adjacent vertices have different colors is NP-complete [39]. The proof in [39] can be modified such that the NP-completeness of the 3-coloring problem also follows for triconnected planar graphs. We omit the details here. Let a triconnected planar graph G and a constant $K \geq 7$ times the number of vertices in the largest face in G be given. G has a unique embedding. Let G^* be the dual graph of G: every vertex of G^* corresponds to a face of G and there is an edge between two vertices in G^*, if and only if the corresponding faces in G share an edge. G^* is triconnected and planar as well. We change graph G^* into a graph G_1^* as follows: every edge $(a,b) \in G^*$ is replaced by three components A_1, A_3, A_5 and two vertices c, d with edges to a and b as shown in Figure 6.2(a). We construct A_i $(i = 1,3,5)$ such that all interior faces are triangles and all interior vertices (vertices not belonging to the outerface of A_i) have degree $\leq K$. The outerface of A_i consists at each side of a and b of i consecutive vertices of degree $K-1$, say $v_1, \ldots, v_i$ at one side and $w_1, \ldots, w_i$ at the other side. of degree $K-1$. Each side also contains a vertex of degree $K-i-2$, say v and w.



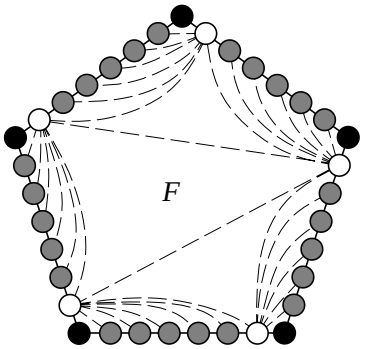
(a) The replacement of edge (a, b) in G.



(b) Every component in F must have the same orientation.



(c) Every face must have the same kind of components.



(d) If a face F consists of one type of components, then a triangulation of F exists with degree $\leq K$.

Figure 6.2: Figures for the NP-completeness proof.

v and w are adjacent to a and b, and to $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$, respectively. We augment the components such that the vertices a and b have degree K. See Figure 6.2(a), where the dotted areas imply the triangulated components A_1, A_3, A_5, hence there are additional edges in it, not shown in the picture. It is not hard to find the precise constructions for A_1, A_3, A_5, although a little tedious. We omit the precise construction. Notice that vertex a (and b) receives at least two incident edges from each component A_i. a has also incident edges to c and d and there is no edge (a, b) in G_1^* Thus $\Delta(G_1^*) \geq 7\Delta(G^*)$.

We add vertices inside some A_i-components with edges to a and b in such a way that the degree of all vertices a and $b \in G^*$ is K in G_1^* and the degree of the other

external vertices of the A_i-components does not change. We call the vertices $v \in G_1^*$ with $deg(v) < K-1$ *white*, with $deg(v) = K-1$ *grey* and with $deg(v) = K$ *black*, and they are drawn accordingly in Figure 6.2. Notice that the vertices of G^* are black in G_1^* and the other vertices in G_1^* are initially grey or white. Suppose G^* has a triangulation with maximum degree $\leq K$. Fix such a triangulation, and construct a planar embedding of the triangulated graph.

Lemma 6.1.2 *Between every two components A_i, A_j with common vertices a, b there must be a vertex c or d.*

Proof: Suppose not, i.e., there are two components A_i, A_j, adjacent to each other between two vertices a and b in G_1^*. Let F be the face between A_i and A_j. If there are two grey vertices adjacent to a or b in F, then one of these grey vertices must get two extra edges by the triangulations, hence will get degree $> K$, which is not allowed. So assume that adjacent to a and b there is a white and a grey vertex. If we want to triangulate F such that every vertex has degree $\leq K$, then each consecutive sequence of grey vertices must get incident edges to a common white vertex. Let v, w be the white vertices and $v_1, \ldots, v_i$, $w_1, \ldots, w_j$ be the grey vertices of F_1. Let $j > i$, then we must assign $i+2$ edges between v and $w_1, \ldots, w_{i+2}$ and i edges must be assigned between w and $v_1, \ldots, v_i$. After this assignment, v and w_{i+2} are both black. Since $j \geq i+2$, F is not completely triangulated. But v and w_{i+2} are now neighbors in F, and one of them must get an extra incident edge. This contradicts the assumption that we can triangulate F such that the maximum degree is $\leq K$. Thus between every two components A_i, A_j of vertices a and b, there must be a vertex c or d. □

This means that for every edge $(a, b) \in G^*$, belonging to faces F_1, F_2, we have to *assign* one A_i-component of a, b to F_1 and one A_j-component of a, b to F_2, with $i \neq j$. Assigning A_i to F_1 means that we construct a planar embedding such that the exterior vertices of A_i at one side between a and b belong to face F_1. Triangulating the faces F, to which c and d belong, can simply be done by adding edges from vertex c or d to all other vertices of face F.

Lemma 6.1.3 *For every face $F \in G_1^*$ only one type component A_i can be assigned to F.*

Proof: Suppose we can triangulate a face F with different components A_i, A_j assigned to it, such that the maximum degree is still $\leq K$. First we notice that every black vertex $v \in F$ may not get an incident augmenting edge, hence there must come an edge between its two neighbors $v_1, v_2 \in F$. Since for any triangulation, v_1 or v_2 must get in total at least two augmenting edges, one of them must be white. This means that every component in F must have the same orientation, i.e., when walking around F we visit alternatingly a black vertex, a white vertex and a sequence of grey vertices (see Figure 6.2(b)). Notice that after adding (v_1, v_2) for every black

vertex v in F, the size of F decreases, but since v_1 or v_2, say v_1, was grey, v_1 is black now, and we repeat our argument to all black vertices v_1. Inspect two adjacent components A_i, A_j of a black vertex v. Let $j > i$. After assigning $i+2$ edges from the white vertex w of A_i to the grey vertices $w_1, \ldots, w_{i+2}$ of A_j, w and $w_1, \ldots, w_{i+2}$ are all black now. But similar to Lemma 6.1.2, since $j \geq i+2$, F is not completely triangulated yet, thus w or w_{i+2} must receive at least one extra edge (see Figure 6.2(c)). This contradicts the assumption that we could triangulate F such that the maximum degree is $\leq K$. □

If only one type of components A_i is assigned to a face $F \in G_1^*$ then we can triangulate F as follows: from every white vertex we add i edges to the grey vertices of the next component in the circular order of F. In the reduced face F we assign edges between every two consecutive white vertices. This triangulates F completely and the degree of every vertex $v \in F$ becomes K. An example is given in Figure 6.2(d).

From Lemma 6.1.2 and 6.1.3 and the construction in Figure 6.2(d) it follows that we can triangulate G_1^* with maximum degree $\leq K$ if and only if we can assign to every face $F \in G_1^*$ only one type of components A_i, $i = 1, 3, 5$, i.e., if and only if we can assign one number $i, i = 1, 3, 5$, to every face in G^* such that every two faces, sharing a common edge in G^*, have a different number, i.e, if and only if there exists a coloring of G with three colors such that every pair of two neighbors v, w, have a different color. As G^* and G_1^* can be constructed in time polynomial in K, n and m, there is a polynomial time transformation from the NP-complete problem of 3-coloring triconnected planar graphs to the problem of triangulating a planar graph while minimizing the maximum degree, hence the latter is NP-complete. □

6.2 Triangulating Planar Graphs While Minimizing the Maximum Degree

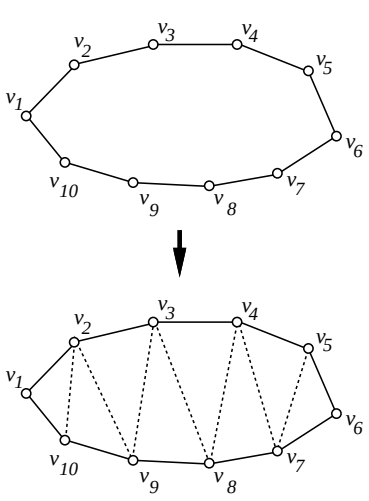
6.2.1 The Algorithm

In this section we present an approximation algorithm for triangulating planar graphs while minimizing the maximum degree. The algorithm will lead to a proof of the following theorem:

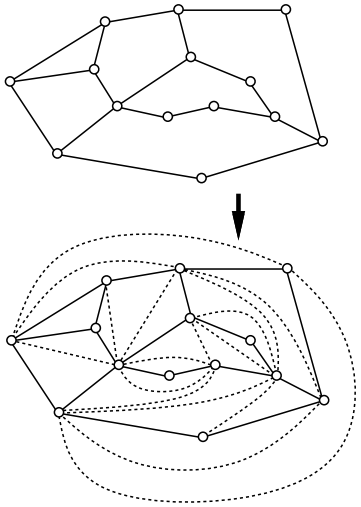
Theorem 6.2.1 *There is a linear time and space algorithm to triangulate a connected planar graph G such that for the triangulation G_1^* of G, $\Delta(G_1^*) \leq \lceil \frac{3}{2}\Delta(G) \rceil + 11$.*

An important procedure in the algorithm is to triangulate a face by adding edges in a "zigzag"-form, as shown in Figure 6.3. It can be described as follows:

ZIGZAG(F, v_1, v_p);
($*$ F is a face of p vertices, numbered $v_1, \ldots, v_p$ in order $*$)
add edges $(v_p, v_2), (v_2, v_{p-1}), (v_{p-1}, v_3), (v_3, v_{p-2}), \ldots, (v_{\lfloor \frac{p}{2} \rfloor}, v_{\lfloor \frac{p}{2} \rfloor + 2})$;
END ZIGZAG



(a) Zigzagging one face.



(b) Zigzagging the planar graph.

Figure 6.3: Example of zigzagging a face and zigzagging a planar graph.

By applying this method to a face, the degree of v_1 and $v_{\lfloor \frac{p}{2} \rfloor + 1}$ does not increase, the degree of v_p and $v_{\lfloor \frac{p}{2} \rfloor}$ (p even) or $v_{\lfloor \frac{p}{2} \rfloor + 2}$ (p odd) increases by 1, and all other degrees increase by 2 (see Figure 6.3(a)).

Let G be a connected planar graph to be triangulated. A simple technique for triangulating G is to apply ZIGZAG to all faces of G. However, since a vertex v belongs to $deg(v)$ faces and $deg(v)$ can increase by two in every face, this may lead to a maximum degree of $3\Delta(G)$. Moreover, this algorithm may imply multiple edges, which are not allowed (see Figure 6.3(b)). To circumvent this we first add edges to G such that G is planar and triconnected (see Chapters 4 and 5), because then the following important lemma holds:

Lemma 6.2.2 *Triangulating a triconnected planar graph G cannot cause multiple edges.*

Proof: Assume not, thus there are two vertices u and v with at least two edges (u, v) in the triangulation of G. Let the second edge (u, v) be added in face F. This implies that u and v are not neighbors in F, thus there are vertices between the two edges (u, v), after adding (u, v) in F. Let w be such a vertex. By planarity of G

and the two edges (u, v) it follows that all paths from w to other vertices of G go via u and v. But this yields that $G - \{u, v\}$ is disconnected, which contradicts the triconnectivity of G. □

The idea for triangulating G is as follows: we compute a "special" ordering of the faces of G, and when applying ZIGZAG(F, v, w) to every face F of G, we start with some "special" vertex v. For the ordering of the faces, we use the definition of the canonical ordering for triconnected planar graphs, as described by Kant [66]: Let an embedding of a triconnected planar graph G be given. G_k denotes the subgraph of G, induced on the vertices $v_1, \ldots, v_k$.

Theorem 6.2.3 ([66]) *The vertices of a triconnected planar graph G can be ordered in a sequence $v_1, \ldots, v_n$ such that v_2 and v_n are neighbors of v_1 and are on a common face, and for every $k, k > 3$:*

1. *v_k is on the outerface of G_k and has at least two neighbors in G_{k-1}, which are on the outerface of G_{k-1}. v_k has at least one neighbor in $G - G_k$. G_k is biconnected,*

2. *or there exists an $l \geq 1$ such that $v_k, \ldots, v_{k+l}$ is a chain on the outerface of G_{k+l} and has exactly two neighbors in G_{k-1}, which are on the outerface of G_{k-1}. Every vertex $v_k, \ldots, v_{k+l}$ has at least one neighbor in $G - G_{k+l}$. G_{k+l} is biconnected.*

Suppose we add v_k, and let $C_{k-1} : v_1 = c_1, \ldots, c_r = v_2$ be the outerface of G_{k-1}. Let c_i be the leftmost neighbor of v_k (called *leftvertex*) and c_j be the rightmost neighbor of v_k on C_{k-1} (called *rightvertex*). Vertices $c_{i+1}, \ldots, c_{j-1}$ of C_{k-1} are called *internal*. The vertices, added in step k, are called *new* in step k. Every vertex, not belonging to the outerface of G, is exactly once new and once internal. Vertices on the outerface of G, i.e., belong to the face, containing v_1, v_2 and v_n, are once new, and do not become internal. Every vertex can be more than once the left- or rightvertex. If we triangulate a face F_k with leftvertex c_i and rightvertex c_j with $j = i + 1$, then c_i or c_j must get an augmenting edge, since they are neighbors on C_{k-1}. To count the number of incident edges of vertex v, added when v was the left- or rightvertex, we introduce a variable *extra*(v), initially 0. Every time when v receives an incident edge and v is the left- or rightvertex, we increase *extra*(v) by one. (This is not expressed in the algorithm TRIANGULATE(G).)

Adding an edge between the neighbors of v in face F is denoted by ADDNEIGHBORS(F, v). By this method *deg*(v) does not increase in face F. ZIGZAG(F, v, w) does not increase *deg*(v), but increases *deg*(w) by one. Let there be K steps in total. In step 1 and 2 vertex v_1 and v_2 are added; in step K vertex v_n is added. The algorithm becomes as follows:

TRIANGULATE(G);

2*OPTBICONNECT(G);
TRICONNECT(G);
compute a canonical ordering $v_1, \ldots, v_n$ of G and start with (v_1, v_2);
for $k := 3$ **to** K **do**
  if we add a face F_k with leftvertex c_i and rightvertex c_j **then**
    if $extra(c_i) \leq extra(c_j)$ **then**
      ADDNEIGHBORS(F_k, c_j); ZIGZAG(F_k, c_i, v) with $(v, c_i) \in F_k$
    else
      ADDNEIGHBORS(F_k, c_i); ZIGZAG(F_k, c_j, v) with $(v, c_j) \in F_k$
  else
    let $c'_1, \ldots, c'_t$ $(t > 2)$ be the neighbors of v_k from left to right on C_{k-1};
    let F'_l be the face, containing v_k, c'_l, c'_{l+1}, for $1 \leq l \leq t-1$;
    ZIGZAG(F'_1, c'_1, v_k);
    for $l := 2$ **to** $t-2$ **do** ADDNEIGHBORS(F'_l, v_k); ZIGZAG(F'_l, c'_l, c'_{l+1}) **rof**;
    ADDNEIGHBORS(F'_{t-1}, c'_{t-1}); ZIGZAG(F'_{t-1}, c'_t, v_k)
rof;
triangulate the outerface F by ZIGZAG(F, v_1, v_2);
END TRIANGULATE

Using the adjacency lists $adj(v)$ for the embedding, it is not very difficult to implement the algorithm such that it works in linear time and space. In the remaining part of this paper we consider the increase of $deg(v)$.

6.2.2 Counting the Increase of $deg(v)$

In this section we inspect the total increase of every vertex v in TRIANGULATE(G). We distinguish the cases that v is new, internal or a left- or rightvertex. The increase of $deg(v)$, when v is a left- or rightvertex, is given by $extra(v)$. When v is part of the outerface F, then v does not become internal, but by ZIGZAG(F, v_1, v_2), $deg(v)$ increases by at most 2. To count the increase of the degree of all other vertices, we have the following lemma:

Lemma 6.2.4 *In every step k the degree of the new and internal vertices increases by at most 3.*

Proof: If we add only one face F_k in step k, then we start with applying ADDNEIGHBORS(F_k, c_i) or ADDNEIGHBORS(F_k, c_j), which increases the degree of the neighbors of c_i or c_j in face F_k by one. Then ZIGZAG(F_k, c_i, v) or ZIGZAG(F_k, c_j, v) is applied, which increases the degree of every internal and every new vertex of face F_k by at most 2.

If we add a vertex v_k with neighbors $c'_1, \ldots, c'_t$, then $deg(c_\alpha), c_\alpha \neq c'_l, (1 \leq l \leq t)$, increases by at most 3 by ADDNEIGHBORS(F'_l, v_k) and ZIGZAG(F'_l, c'_l, c'_{l+1}). In face F'_l, $(2 \leq l \leq t-2)$, $deg(c'_l)$ and $deg(c'_{l+1})$ increase by one by ADDNEIGHBORS(F'_l, v_k), and $deg(c'_{l+1})$ increases one by ZIGZAG(F'_l, c'_l, c'_{l+1}). $deg(c'_2)$ increases by at most 2

in F_1' by ZIGZAG(F_2', c_1', v_k). $deg(c_{t-1}')$ does not increase in F_{t-1}', because of the call ADDNEIGHBORS(F_{t-1}', c_{t-1}'). Hence $deg(c_l')$, $2 \leq l \leq t-1$, increases by at most 3. $deg(v_k)$ increases by one in F_1' by ZIGZAG(F_1', c_1', v_k). $deg(v_k)$ does not increase in the faces $F_2', \ldots, F_{t-2}'$. In F_{t-1}', $deg(v_k)$ increases by one in ADDNEIGHBORS(F_{t-1}', c_{t-1}') and by one in ZIGZAG(F_{t-1}', c_t', v_k), which completes the proof. □

To count the increase of $extra(v)$ during all steps of TRIANGULATE(G), we consider the outerface C_k after adding the new vertices in any step k:

Lemma 6.2.5 *For every four consecutive vertices $c_\alpha, c_{\alpha+1}, c_{\alpha+2}, c_{\alpha+3}$ on C_k, $k \geq 3$, the following holds:*
If $extra(c_{\alpha+1}) = 2$ then either $extra(c_\alpha) = extra(c_{\alpha+2}) = 0$ or $extra(c_\alpha) = extra(c_{\alpha+3}) = 0$ and $extra(c_{\alpha+2}) = 1$. If $extra(c_{\alpha+1}) = extra(c_{\alpha+2}) = 1$, then we have $extra(c_\alpha) = extra(c_{\alpha+3}) = 0$.

Proof: By induction. When starting the algorithm $extra(v_1) = extra(v_2) = 0$ holds, thus then the lemma is true. Assume the lemma holds for outerface C_{k-1}.

If we add a face F_k in step k with two (or more) vertices v_{k_1}, v_{k_2} with leftvertex c_i and rightvertex $c_j = c_{i+1}$, then the lowest *extra*-value of c_i and c_j, say $extra(c_i)$, increases by one. If $extra(c_i) = 1$, then by the induction step, we obtain the following *extra*-values for $c_{i-1}, c_i, v_{k_1}, v_{k_2}, c_j, c_{j+1}$: 0, 2, 0, 0, $extra(c_j)$, 0. If $extra(c_i) = 0$, then we obtain: 0, 1, 0, 0, $extra(c_j)$, 0. Hence also on C_k the lemma holds.

If we add only one vertex v_k with leftvertex c_i and rightvertex c_j, then we do not increase $extra(c_i)$ and $extra(c_j)$, and we initialize $extra(v_k)$ to 0. Hence then also the lemma holds. □

Lemma 6.2.6 *For every pair of consecutive vertices $c_\alpha, c_{\alpha+1}$ on C_k holds that $extra(c_\alpha) + extra(c_{\alpha+1}) \leq 3$.*

Proof: Suppose not. Let k be the smallest possible integer such that for two consecutive vertices $c_\alpha, c_{\alpha+1}$ on C_k: $extra(c_\alpha) + extra(c_{\alpha+1}) > 3$, thus $extra(c_\alpha) + extra(c_{\alpha+1}) = 4$ on C_k, and on C_{k-1}, $extra(c_\alpha) + extra(c_{\alpha+1}) = 3$. But by Lemma 6.2.5, $extra(c_{\alpha-1}) = extra(c_{\alpha+2}) = 0$ on C_{k-1}. Thus both c_α and $c_{\alpha+1}$ have neighbors with smaller *extra*-value. The added face in C_k cannot have leftvertex c_α and rightvertex $c_{\alpha+1}$, because then vertices with *extra*-value 0 come between them. Hence $extra(c_\alpha)$ and $extra(c_{\alpha+1})$ do not increase in step k. This contradicts the assumption that in step k, $extra(c_\alpha) + extra(c_{\alpha+1}) > 3$. □

Corollary 6.2.7 *For every vertex $v \in G$, $extra(v) \leq 2$.*

Thus every vertex v receives at most 3 edges when v is new, at most 2 edges when v is left- or rightvertex, and finally at most 3 edges when v is internal. This implies that TRIANGULATE(G) triangulates in linear time and space a triconnected planar

graph G such that every vertex receives at most 8 extra incident edges. If G is not triconnected, then we do the following: We apply the algorithm BICONNECT(G), described in Chapter 4, to biconnect G, while increasing the degree of every vertex by at most 2. Then we apply the algorithm TRICONNECT(G) to triconnect G, while increasing the degree of every vertex by at most $\lceil \frac{\Delta(G)}{2} \rceil$. Applying the algorithms BICONNECT, TRICONNECT and TRIANGULATE completes Theorem 6.2.1, which says that we can triangulate in linear time and space a connected graph G such that for the maximum degree of the triangulation G': $\Delta(G') \leq \lceil \frac{3}{2}\Delta(G) \rceil + 11$.

For a lower bound of the increase of $deg(v)$ we can use Theorem 5.3.2: It is easy to prove that for any triangulation G' of the biconnected planar graph G described in Theorem 5.3.2, that $\Delta(G') \geq \lceil \frac{3}{2}\Delta(G) \rceil$. Hence our approach works only an additive constant from an existential lower bound. The same holds for the triconnected case, in which case we have the following theorem:

Theorem 6.2.8 *For every $\Delta = \Delta(G) \geq 5$, there exist triconnected planar graphs G such that for every triangulation G' of G: $\Delta(G') \geq \Delta(G) + 3$.*

Proof: Construct a triconnected planar graph G as follows. Take $V = \{v\} \cup \{v_i | 1 \leq i \leq \Delta\} \cup \{v_{ij} | 1 \leq i \leq \Delta, 1 \leq j \leq \Delta - 1\}$, thus $|V| = \Delta^2 + 1$. Connect v to the Δ vertices $v_1, \ldots, v_\Delta$. Connect v_i to $\Delta - 1$ vertices v_{ij}, $1 \leq j \leq \Delta - 1$. Connect v_{ij} to $v_{i(j+1)}$ if $j < \Delta - 1$ and to $v_{(i+1)1}$ otherwise (see Figure 6.4). Notice that G is triconnected. Suppose we can triangulate G such that $\Delta(G') \leq \Delta(G) + 2$. Thus v can get at most two augmenting edges. Let $F_1, \ldots, F_\Delta$ be the adjacent faces of v. F_i consists of 5 vertices, thus every triangulation of F_i consists of two augmenting edges, incident to one vertex $w \in F_i$, which we call *marked*, to its non-neighbors of F_i. Hence except for at most two neighbors v_i, v_j of v, all other $\Delta - 2$ neighbors of v must be marked. But marking both v_i and v_{i+1} increases $deg(v_i)$ by 3. Hence only half of the neighbors of v can be marked, which is larger than $\Delta - 2$ for $\Delta \geq 5$. Thus $\Delta(G') \geq \Delta + 3$, for $\Delta \geq 5$. □

It is an interesting open problem to close this gap between the lower bound of 3, and the upper bound of 8, delivered by the algorithm TRIANGULATE.

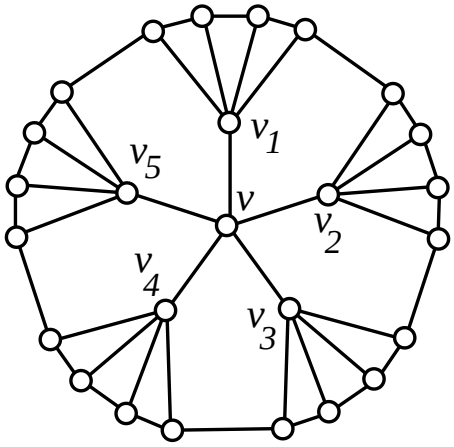


Figure 6.4: Illustration of Theorem 6.2.8.

Chapter 7

Augmenting Outerplanar Graphs

7.1 Introduction

Outerplanar graphs are an interesting subclass of planar graphs, because all vertices share one face, namely, the outerface. There are two specific drawing techniques for outerplanar graphs. The first method is to place all vertices of a biconnected outerplanar graph on the corner points of a regular n-gon (in clockwise order around the face). Every interior edge is a chord, and drawn as a straight line inside the n-gon. Though this drawing is planar and convex, the minimum angle between two consecutive incident edges of a vertex can be $O(\frac{1}{n})$ and the ratio between the length of the longest and smallest edge can be $O(n)$. Another drawing algorithm for outerplanar graphs requires as input a maximal outerplanar graphs (mop), i.e., all interior faces are triangles. It starts with drawing a face F, say with vertices u, v, w, which has at least one edge, say (u, v), on the outerface. It draws (u, v) horizontally and draws w above (u, v) such that $\mathit{length}(u, w) = \mathit{length}(v, w)$, and $\angle vuw = \angle uvw$. From (u, w) and (v, w) the remaining faces of the outerplanar graph are drawn recursively. See figure 7.1 for an example. The drawing can be constructed such that the minimum angle is $\frac{\pi}{2(d+2)}$, by a result of Malitz & Papakostas [80]. However, the ratio between the longest and smallest edge can be $O(2^n)$. This drawing construction has also been applied succesfully by Lin & Skiena [78], to draw a polygon P on a grid of polynomial size, where the visibility graph of P is a maximal outerplanar graph.

Let us first consider the problem of augmenting an outerplanar graph such that the resulting graph is biconnected, and still outerplanar. If G is disconnected, then the algorithm CONNECT(G) of [27], described in Chapter 4, is applied. This algorithm preserves the outerplanarity. Hence we may assume from now on that G is connected and outerplanar. The algorithm of Read [92], and the modification BICONNECT(G) of it described in Chapter 4, can be used to biconnect G. If b is the number of blocks, then at most $b-1$ blocks are added (see Lemma 4.1.4. Let v be a cutvertex with $d(v)$ v-blocks. If u and w appear consecutively in $adj(v)$ and u and

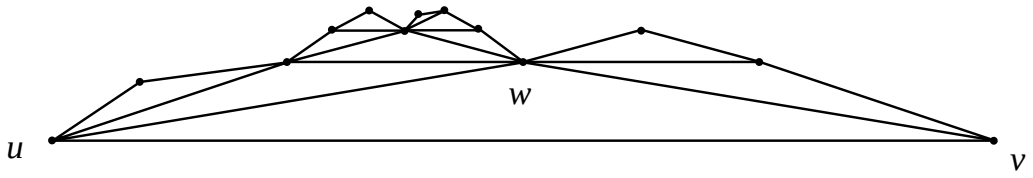


Figure 7.1: Drawing an outerplanar graph.

w belong to different blocks, then (u, w) is added. In this way $d(v) - 1$ edges are added. There remains one pair of consecutive neighbors u and w in $adj(v)$, belonging to different blocks, where no edge (u, w) is added. (u, v) and (v, w) belong to the outerface before the augmentation, hence also after the augmentation, v belongs to the outerface. This completes the following theorem:

Theorem 7.1.1 *If the input graph G is outerplanar, then the augmented graph obtained by* BICONNECT(G) *is biconnected and outerplanar.*

Every biconnected graph is also bridge-connected, thus this algorithm can be used to bridge-connect G as well. In both cases the maximum increase of the degree of every vertex is at most 2. Notice that if G is not a triangle, then G cannot be triconnected. This follows because if G contains no chords, then G is a cycle, and deleting any pair of two non-adjacent vertices disconnects G. If G contains a chord (u, v), then $G - \{u, v\}$ is disconnected.

Biconnecting (or bridge-connecting) G by a minimum number of edges while maintaining outerplanarity seems to be a much harder problem. Even for trees T with p leaves and maximum degree d it is not very difficult to construct examples in which $\frac{p}{2}$ edges are sufficient to biconnect (or bridge-connect) T, and examples in which $p - d$ edges are necessary to biconnect (or bridge-connect) T, while preserving outerplanarity. Also there exist outerplanar graphs G with b blocks and 2 pendants, for which $b - 1$ edges are necessary to biconnect G (in which case BICONNECT(G) gives an optimal solution). However, it seems to be difficult to prove in general that a solution is also optimal.

To apply the huge number of planar graph drawing algorithms, we consider the problem of augmenting outerplanar graphs such that the augmented graph is bridge-connected, biconnected or triconnected and planar. We show that the number of added edges is equal to the number of edges, which we would add without the additional planarity constraint. Hence the presented solutions are optimal. The algorithms are simple and can easily be implemented to run in linear time. For drawing biconnected and triconnected planar graphs many advanced algorithms are known, which can thus be used to draw the (augmented) outerplanar graph. Triangulating outerplanar graphs such that the augmentation is a mop can easily be done in linear time. We show that we can do this triangulation in polynomial

time such that the maximal degree is minimized. This result has nice features for the algorithm of Malitz & Papakostas [80].

7.2 Bridge-Connectivity

In this section we consider the problem of how to add a minimum number of edges to an outerplanar graph G, such that the augmented graph G' is bridge-connected and planar. Bridge-connecting G is equal to bridge-connecting the forest $bc(G)$. The algorithm for bridge-connecting $bc(G)$ is inspired by the general bridge-connecting algorithm of Eswaran & Tarjan [27]. They gave the following lower bound on the number of edges needed to make $bc(G)$ bridge-connected:

Theorem 7.2.1 ([27]) *At least $\lceil \frac{p}{2} \rceil + q$ edges are needed to make $bc(G)$ bridge-connected, if $p + q > 1$.*

If G is not connected, then we can first apply the algorithm CONNECT(G) of Chapter 4. Hence we assume from now on that G is connected. Let T_{BC} be its BC-tree. The children of each B-node are in the order of visiting them around the corresponding block. It remains to find a set of $\lceil \frac{p}{2} \rceil$ edges to bridge-connect a tree with p leaves and a fixed order of the children of each B-node. Eswaran & Tarjan make the graph bridge-connected by adding the edges $(v_i, v_{\lfloor \frac{p}{2} \rfloor + i})$ $(1 \leq i \leq \lceil \frac{p}{2} \rceil)$ with $v_1, \ldots, v_p$ the leaves of T_{BC} enumerated from left to right. Of course this destroys the planarity and therefore we apply a complete new strategy.

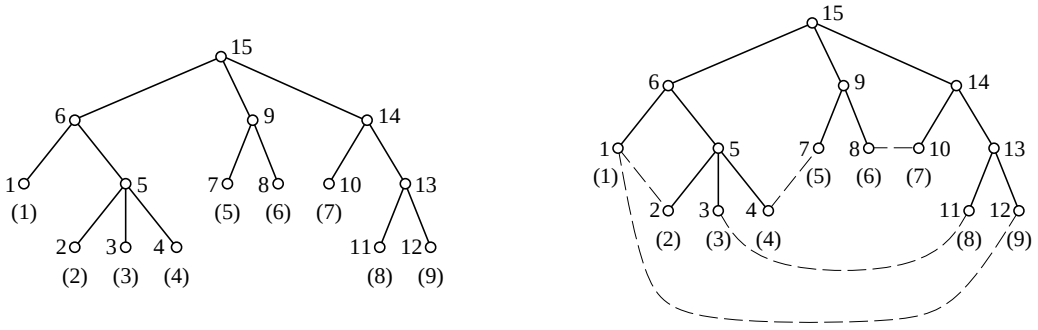
Root T_{BC} at an arbitrary non-leaf B-node. We number the nodes in *postorder*, which means that we traverse the tree by a depth-first search traversal, where we first visit the children of a node for numbering, before numbering the node. We visit the children from left to right, which means that the leftmost descendant leaf gets number 1 and the root gets number n. Let $v_1, \ldots, v_n$ be the vertices where vertex v_i has postorder-number i. Let $v(1), \ldots, v(p)$ be the leaves of T_{BC}, numbered from left to right. The following lemma is easy to prove:

Lemma 7.2.2 *The descendants of any node have consecutive numbers in any postorder numbering.*

The idea of the algorithm is as follows: we visit the nodes in increasing postorder numbering. If v_i is a leaf $v(k)$ and k is odd, then we simply add an edge $(v(k-1), v(k))$. If v_i is an internal node, then we test whether there exists an augmenting edge $(v(\alpha), v(\beta))$, with $v(\alpha)$ a descendant leaf of v_i and $v(\beta)$ not. If there is no such edge, then we change some added edges to obtain this. To this end a variable x is introduced, which increases monotone from 1 to p. Later it is proved that if $v(x)$ is a descendant leaf of v_i, then there exists an augmenting edge $(v(\alpha), v(\beta))$, with $v(\alpha)$ a descendant leaf of v_i and $v(\beta)$ not. This precisely corresponds to the definition of the bridge-connectivity.

The algorithm OUTERBRIDGECONNECT makes this idea more precise. (The command LCA$(v(a), v(b))$ denotes the least common ancestor of leaves $v(a)$ and $v(b)$.)

OUTERBRIDGECONNECT
  $A := \emptyset$; { A becomes the set of augmenting edges. }
  $x := 1$;
  for $i := 2$ **to** n **do**
    if v_i is a leaf $v(k)$ **then**
      if k is odd **then** $A := A \cup \{(v(k-1), v(k))\}$
    else
      let $v(j_1), \ldots, v(j_h)$ be the descendant leaves of v_i;
      if j_1 is even **and** $x < j_1$ **then**
        $A := A - \{(v(j_1), v(j_2))\} \cup \{(v(x), v(j_1))\}$; $x := j_2$
  rof;
  if $x = 1$ **or** $x = p$ **then** $A := A \cup \{(v(1), v(p))\}$ **else**
    let $(v(1), v(y)) \in A$;
    if LCA$(v(y), v(x))$ is an ancestor of LCA$(v(x), v(p))$ **then**
      $A := A - \{(v(1), v(y))\} \cup \{(v(y), v(x)), (v(1), v(p))\}$
    else
      $A := A \cup \{(v(x), v(p))\}$;
END OUTERBRIDGECONNECT



(a) Numbering of the nodes and leaves.

(b) The bridge-connected augmentation.

Figure 7.2: Bridge-connecting an outerplanar graph.

In Figure 7.2 an example is given of the algorithm OUTERBRIDGECONNECT. It follows from the algorithm that $|A| = \lceil \frac{p}{2} \rceil$, but to prove that G indeed is bridge-connected and planar, we need the following lemma of Tarjan [108]:

Lemma 7.2.3 *Let $G' = (V, E \cup A)$, and let (v, w) be an edge of G'. Then (v, w) is a bridge of G' if and only if v is the parent of w in T_{BC} and there is no edge $(\alpha, \beta) \in A$ such that α is a descendant of w in T_{BC} and β is not a descendant of w in T_{BC}.*

Lemma 7.2.4 *If at some step during the algorithm, $v(x)$ is a descendant of $w \neq v_1$, then in the augmented graph w has a descendant leaf, which has an edge to a leaf, not a descendant of w.*

Proof: Let us call the added edges $(v(x), v(j_1))$ in OUTERBRIDGECONNECT *special*. Consider a vertex $w \neq v_1$ with descendant leaves $v(j_1), \ldots, v(j_h)$ from left to right. Notice that for every special edge $(v(a), v(b))$ a is odd and b is even, and the next special edge starts in $v(b+1)$. Assume first that at the end $x > j_h$ holds. If j_h is odd then later the special edge $(v(j_h), v(j_h + 1))$ is added, and $v(j_h + 1)$ is not a descendant of w. If j_h is even, then let $(v(a), v(b))$ be the special edge with $b \leq j_h$ as high as possible. When $(v(a), v(b))$ is added, then x is set to $b + 1$. Since j_h and $b + 1$ are both even it follows that $x \leq j_h$. At the end $x \geq j_h + 2$ holds, thus there is a special edge $(v(a'), v(b'))$ with $a' \leq j_h < b'$.

Assume now that at the end $x \leq j_h$. If $x = 1$ then $p > j_h$, and the edge $(v(x), v(p))$ is added to A. Otherwise let $(v(1), v(y)) \in A$. Assume first that $j_1 \leq y$. If $j_h = p$ then $j_1 > 1$ thus the added edge $(v(1), v(y))$ or $(v(1), v(p))$ satisfies the lemma. If $p > j_h$ then the added edge $(v(x), v(p))$ to A satisfies the lemma. If $j_1 > y$ then if $p > j_h$ then the edge $(v(y), v(x))$ is added to A, otherwise the edge from $v(x)$ to $v(y)$ or $v(p)$ satisfies the lemma. □

Lemma 7.2.5 *$G' = (V, E \cup A)$ is bridge-connected.*

Proof: Let (v, w) be any edge of G' such that $v = \mathit{parent}(w)$ in T_{BC}. Let the descendant leaves of w be $\{v(j_1), \ldots, v(j_h)\}$ from left to right. If $j_1 = 1$ then initially $v(x)$ is a descendant of w and we can apply Lemma 7.2.4, so assume $j_1 > 1$. Every leaf gets an incident augmented edge, hence if $j_1 = j_h$ then the lemma directly follows, thus assume $j_h > j_1$. If j_1 is odd, then the edge $(v(j_1 - 1), v(j_1))$ is added. If later $(v(j_1 - 1), v(j_1))$ is removed, then x is set to $j_1 + 1$. Hence in this case $v(x)$ is a descendant of w, and we can apply Lemma 7.2.4. Otherwise, if j_1 is even and $x < j_1$, then the edge $(v(x), v(j_1))$ is added, and $v(x)$ is not a descendant of w. If $x \geq j_1$, then $v(x)$ is a descendant leaf of w, and we can apply Lemma 7.2.4. This completes the proof. □

Constructing the graph $bc(G)$ can be done in linear time, as well as adding the edges such that the graph is connected. Computing the postorder numbering of T_{BC} can easily be performed in linear time. At each internal node v we store a pointer to its leftmost leaf. Using this plus one pointer to leaf $v(x)$ the total required time is $O(1)$ when visiting v_i in BRIDGECONNECT. After visiting v_n we have to decide

whether $\text{LCA}(v(y), v(x))$ is an ancestor of $\text{LCA}(v(x), v(p))$. We do this by looking whether there is a vertex v' on the path between v_1 and $v(p)$ such that that for the leftmost leaf $v(z)$ of v', $y < z \leq x$ holds. $\text{LCA}(v(y), v(x))$ is an ancestor of $\text{LCA}(v(x), v(p))$, if and only if there exists such a vertex v'. This leads to the main theorem of this section:

Theorem 7.2.6 *The planar bridge-connectivity augmentation problem can be solved in linear time and space for outerplanar graphs.*

7.3 Biconnectivity

For the planar biconnectivity augmentation algorithm we assume that the input outerplanar graph G is connected (otherwise we can apply CONNECT to G). Let T_{BC} be the BC-tree of G. We first study some properties of T_{BC}. The following definitions are from Hsu & Ramachandran [53].

Definition 7.3.1 *A node v of T_{BC} is called* massive *if and only if v is a C-node with $d(v) - 1 > \lceil \frac{p}{2} \rceil$. A node v of T_{BC} is* critical *if and only if v is a C-node with $d(v) - 1 = \lceil \frac{p}{2} \rceil$. T_{BC} is* critical *if and only if there exists a critical C-node in T_{BC}.*

Definition 7.3.2 *T_{BC} is* balanced *if and only if G has not a massive C-node. A graph G is balanced if and only if T_{BC} is balanced.*

Definition 7.3.3 (the leaf-connecting condition) *Two leaves u_1 and u_2 of T_{BC} satisfy the leaf-connecting condition if and only if the path P from u_1 to u_2 in T_{BC} contains either (1) two nodes of degree more than 2, or (2) one B-node of degree more than 3.*

Some first observations concerning T_{BC} are the following:

Lemma 7.3.1 ([97]) *There can be at most one massive node in T_{BC}. If there is a massive node in T_{BC}, then there is no critical node in T_{BC}, and there can be at most two critical nodes in T_{BC}, if $p > 2$.*

Lemma 7.3.2 ([53]) *Let u_1 and u_2 be two leaves of T_{BC} satisfying the leaf-connecting condition (definition 7.3.3). Let α and β be non-cutvertices in blocks of G represented by u_1 and u_2 respectively. Let G' be the graph obtained from G by adding an edge between α and β and let P represent the path between u_1 and u_2 in T_{BC}. The following three conditions are true.*

- *$p' = p - 2$.*
- *If v is a C-node in P with degree greater than 2 in T_{BC}, then the degree of v decreases by 1 in $bc(G')$.*

- *If v is a C-node in P with degree equal to 2, then v is eliminated in $bc(G')$.*

The idea for obtaining the lower bound of $\max\{d-1, \lceil \frac{p}{2} \rceil\}$ edges (see Theorem 3.0.1) without the requirement of planarity is to add an edge between two leaves y and z under the conditions that the path P between y and z passes through all critical vertices and the new block tree has two less leaves if T_{BC} has more than 3 leaves. Thus the degree of any critical vertex decreases by 1 and the tree remains balanced (see [53, 97]). In our case we also have to choose y and z such that the augmented graph remains planar. We show that this can be achieved and that the general lower bound stated in Theorem 3.0.1 is achieved for our problem.

The algorithm of Hsu & Ramachandran [53] for biconnecting graphs by adding a minimum number of edges consists of three stages. Stage 1 is making G connected, for which we used CONNECT of Chapter 4. We will describe now Stage 2 and 3 for our problem of biconnecting outerplanar graphs. Stage 2 eliminates the massive nodes and Stage 3 makes the graph biconnected. Stage 2 can easily be modified for our problem but for Stage 3 we have to modify the algorithm extensively.

7.3.1 Stage 2

Suppose there is a massive node v^* in T_{BC}. (If no massive node v^* exists, no action is taken at Stage 2.) Let γ be the number of components of $T_{BC} - \{v\}$ containing only one leaf. Call such components *1-chains*. There are $d(v^*) - \gamma$ components, each containing at least 2 leaves, so $p \geq \gamma + 2(d(v^*) - \gamma)$. We pick v^* as the root of T_{BC} and number the leaves of T_{BC} from left to right by $v(1), \ldots, v(p)$. We now add 2δ edges such that as many 1-chains as possible are coalesced into one by the following algorithm, with $\delta = d(v^*) - 1 - \lceil \frac{p}{2} \rceil$.

$i := 1; A := \emptyset$; { A becomes the set of augmenting edges. }
while $|A| < 2\delta$ **and** $i < n$ **do**
  if $v(i)$ and $v(i+1)$ are both 1-chains **then** $A := A \cup \{(v(i), v(i+1))\}$;
  $i := i + 1$
od;
$i := 1$;
while $|A| < 2\delta$ **do**
  if $v(i)$ is a 1-chain **and** $(v(i), v(i+1)) \notin A$ **then** $A := A \cup \{(v(i), v(i+1))\}$;
  $i := i + 1$
od

Let G' be the augmented graph by adding the 2δ edges of A between the corresponding pendants. We do this by adding edges between exterior vertices on the last blocks of the 1-chains, which preserves the planarity. Let T'_{BC} be the BC-tree of G' with p' leaves, with $p' = p - 2\delta$, and let $d'(v)$ denote the d-value of v in T'_{BC}.

Lemma 7.3.3 ([97]) *For all C-nodes v of T'_{BC}, $d'(v) - 1 \leq \lceil \frac{p'}{2} \rceil$ holds.*

Proof: For v^* holds that $d'(v^*) - 1 = d(v^*) - 1 - 2\delta = \lceil \frac{p}{2} \rceil - \delta = \lceil \frac{p'}{2} \rceil$. Consider a C-node $v \neq v^*$, and suppose that $d'(v) - 1 > \lceil \frac{p'}{2} \rceil$. Now $p \geq d(v^*) + d(v) - 2$ $= (d(v^*) - 1) + (d(v) - 1) > (\lceil \frac{p}{2} \rceil + \delta) + \lceil \frac{p'}{2} \rceil = (\lceil \frac{p}{2} \rceil + \delta) + (\lceil \frac{p}{2} \rceil - \delta) = 2\lceil \frac{p}{2} \rceil \geq p$, which is a contradiction. □

7.3.2 Stage 3

In Stage 3 we have to deal with a graph G where T_{BC} is balanced. (Assume we have performed Stage 2.) The idea is to add an edge between two leaves y and z under the conditions that the path P between y and z passes through all critical nodes (at most 2) and the BC-tree of the augmented graph has two leaves less if T_{BC} has more than 3 leaves. Thus the degree of any critical node decreases by one and the tree remains balanced. We also want y and z to be leaves that satisfy the leaf-connecting condition, because then we can use Lemma 7.3.2 and it will lead to the desired lower bound of $\lceil \frac{p}{2} \rceil$ edges to biconnect a balanced graph. Moreover, this path must be such that adding an edge between two non-cutvertices of the blocks represented by y and z does not destroy planarity. Let us be more precise about this. We call two leaves y and z in T_{BC} *adjacent* if after adding (y, z) to T_{BC} all other leaves of T_{BC} are either inside or outside the new created cycle C. If y and z are adjacent, then we can add the edge (y', z'), with y' and z' non-cutvertices in the blocks of B-nodes y and z. Hence in every step we look for two adjacent leaves y and z, satisfying the leaf-connecting condition and the path P between them in T_{BC} passes through all critical nodes. We will show that these pairs always exist.

Since the nodes with degree 2 are of no interest in the algorithm, we eliminate them from the BC-tree by contracting their two incident edges into one. We do this also during the algorithm. In particular, if the root of the BC-tree, say b^*, gets degree 2, then we take one of the children of b^* the root and we eliminate b^* from the tree. This can easily be done in $O(1)$ time. The algorithm becomes as follows:

OUTERBICONNECT(G)
(* G has at least 3 nodes and T_{BC} is balanced; *)
  Let T_{BC} be rooted at an arbitrary B-node b^*;
  while $p \geq 2$ **do**
    if $d = 2$ **then** let v be a B-node (different from b^*) with degree > 2
    else let v be a C-node with the largest degree in T_{BC};
    use algorithm PATHFINDER(v) to find a node w with degree > 2 and two
    adjacent leaves y and z such that the path between them passes through v and w;
    find non-cutvertices α and β in the corresponding blocks of G represented
    by y and z respectively;
    add an edge between α and β and update T_{BC}
  od;
END OUTERBICONNECT

We now describe the procedure PATHFINDER, that finds a node w and the two adjacent leaves y and z whose path P between them passes through v and w. Recall from Section 7.2 that any order of children of a C-node is allowed, but the children of a B-node may only be swapped from a left-to-right order into a right-to-left order. We construct the following data structure for T_{BC} (which is almost equal to the construction of PQ-trees, introduced in [9], and explained in Section 2.2.2)

- Every node v in T_{BC} is represented by a record. If v is not a leaf, then v has a pointer to its leftmost and rightmost child, called *l-child* and *r-child*, respectively.
- The children of each node are stored in a doubly linked list.
- If a node is a child of a C-node or the left- or rightmost child of a B-node, then it has a parent-pointer to it, otherwise this pointer is *nil*.

Since we may permute the children of a C-node in any order, we sort the children of each C-node v such that all non-leaf children occur at one side, say starting from the leftmost child of v. The idea is to walk from v towards root b^*, until the parent-pointer is *nil* or b^*. Let w be the highest node reached from v to b^*, then we change the tree such that the path from w to v goes via *l-child* pointers only. We reach leaf y now by following *l-child* pointers from v and leaf z by following *r-child* pointers from the left sibling of w if w is not the leftmost child of its parent, otherwise we take the *r-child* pointers from the parent of w.

If there are only three leaves, then we can reduce T_{BC} to a new tree with two leaves by picking any pair of leaves in T_{BC} and connecting them. We know that we can reduce a BC-tree of 2 leaves into a single node by connecting the two leaves. So assume further that $p > 3$. Then the algorithm can be described as follows (swap(a, b) swaps the contents of a and b):

```
PATHFINDER(node v);
(* v is the C-node with largest degree in T_BC or a B-node with degree > 2; *)
  w := v;
  while parent(w) ≠ nil and parent(w) ≠ b* do
    if w ≠ l-child(parent(w)) then swap(w, l-child(parent(w)));
    w := parent(w)
  od;
  y := v;
  while y is not a leaf do
    if leftmost child of y is a leaf then swap(l-child(y), r-child(y));
    y := l-child(y)
  od;
  if w ≠ l-child(parent(w)) then z := left sibling of w
  else z := r-child(parent(w));
```

while z is not a leaf **do**
  if rightmost child of z is a leaf **then** swap(*l-child*(z), *r-child*(z));
  z := *r-child*(z)
od;
END PATHFINDER

Lemma 7.3.4 *y and z are adjacent.*

Proof: Let w be the vertex obtained in OUTERBICONNECT. Let w' be the parent of w. From w' there are two paths downwards: one path via w and v to leaf y, following only the leftmost child pointers. If w is the leftmost child of w', then the path to leaf z follows the rightmost child pointers of w', otherwise the path to leaf z follows the rightmost child pointers from the left sibling of w. Hence in both cases all other leaves are on one side of the cycle, obtained by adding (y, z) to T_{BC}. □

This means that we can add an edge (α, β) between two non-cutvertices of the blocks of y and z without destroying the planarity.

Lemma 7.3.5 *y and z satisfy the leaf-connecting condition.*

Proof: Let v be the starting node in PATHFINDER, then $d(v) > 2$. When we stop in the **while**-loop of PATHFINDER, then either *parent*(w) = b^* or *parent*(w) = *nil* and, hence, has degree > 2 and is part of path P. Assume w.l.o.g. that $d(b^*) < 3$, thus b^* has degree 1, because we eliminated all nodes with degree 2 from T_{BC}. w has degree at least 3 because all non-leaf vertices have degree at least 3. If $w \neq v$ then we have two nodes with degree > 2 on the path, so assume $w = v$. Hence v is the only child of b^*. If v is a C-node and there is another node v' with degree > 2 in T_{BC}, then v' is the leftmost or rightmost child of v, because the children of each C-node in T_{BC} are sorted such that all non-leaf children occur at one side. But path P is constructed by taking from v the *l-child* pointers and taking from b^* the *r-child* pointers. Hence v' is part of P. Assume finally that v is a B-node (hence $d = 2$). If v has degree > 3 then $w = v$ satisfies the leaf-connecting condition, so assume v has only two children. Since v is the only child of b^*, both children of v are part of P. Since $p > 3$, it follows that at least one of them is not a leaf, i.e., has degree at least 3 and is part of P. Hence leaves y and z always satisfy the leaf-connecting condition. □

Lemma 7.3.6 *For outerplanar graphs G with T_{BC} balanced, the algorithm* OUTERBICONNECT *finds a set of $\lceil \frac{p}{2} \rceil$ edges such that, when this set is added to G, a biconnected planar graph is obtained.*

Proof: Assume w.l.o.g. that $p > 3$. In this case, a critical node must have degree more than 2.
Case 1: T_{BC} has two critical nodes v and w. All other non-leaf vertices have degree 2 and, hence, are eliminated from the tree. Since PATHFINDER will find another node with degree > 2 if present, both v and w will be part of P.
Case 2: T_{BC} has only one critical node v. OUTERBICONNECT takes the C-node with highest degree, thus v is part of P. Since T_{BC} is balanced and $p > 3$, there exists a node $w \neq v$ with degree more than 2, otherwise v is massive. PATHFINDER find a node w with degree > 2 by Lemma 7.3.5.
Case 3: T_{BC} has no critical node. OUTERBICONNECT will take the C-node with highest degree if $d \geq 2$, otherwise it takes a B-node with degree ≥ 3. In both cases PATHFINDER finds a node $w \neq v$ with degree > 2 on path P.

In all three cases, we find two nodes of degree more than 2 or a B-node of degree more than 3. Thus by Lemma 7.3.2, the number of leaves in the new BC-tree reduces by two. When v or w is critical, the value of d reduces by 1. Thus the BC-tree remains balanced. Hence the lower bound of Theorem 3.0.1 is achieved by the algorithm. □

For finding the C-node with highest degree, we maintain an array *bucket*, and initially store in $bucket[i]$ all C-nodes with degree i. We use an extra pointer *ptr* for finding the C-node with highest degree, starting at $bucket[d]$. The degree of the nodes only decrease, hence *ptr* moves monotone from $bucket[d]$ to $bucket[3]$. Hence the total work for finding the C-node with highest degree is $O(d)$ in total. When the degree of a C-node c decreases, we can remove c from $bucket[d(c)]$ and store c in $bucket[d(c) - 1]$ in $O(1)$ time. If all buckets are empty, then there is no C-node in T_{BC}, because a C-node has not degree 1 and we eliminated all nodes of degree 2. We now have to take a B-node with degree > 2. Notice that either b^* or one of its children must have degree > 2, hence we can easily find such a B-node in $O(1)$ time.

If $d(b^*) \geq 3$, then the lowest common ancestor of y and z is always a B-node b_1. Let w_1, w_2 be two children of b_1, part of P. We walk from y to z and make every C-node a child of b_1 and we make all children of a B-node children of b_1. We store them between w_1 and w_2 in the order we visit them between y and z. Since all B-nodes on P are now eliminated and every C-node on P is now child of P, the degree of several nodes on P is decreased in the updated tree. Since only the degree of nodes on P decreases, we test these nodes for degree 2, because then we eliminate them. If the lowest common ancestor of y and z is a C-node c_1, then we take a new B-node b_1, and do the same as above, and add finally this B-node as child of c_1 in the tree T_{BC}.

Lemma 7.3.7 *Algorithm* OUTERBICONNECT *runs in* $O(n)$ *time.*

Proof: The BC-tree can be built in $O(n)$ time, and has $O(n)$ nodes. A linear time bucket-sort routine is used to sort the degree of the C-nodes. By the algorithm

PATHFINDER, the path P between the adjacent leaves y and z is found in $O(|P|)$ time. By Theorem 4.3.1, the number of times a node is visited is no more than its degree. Since the summation of the degrees of all nodes in a tree with n nodes is $O(n)$, the lemma follows. □

Lemma 7.3.7 completes the following result:

Theorem 7.3.8 *There is a linear time and space algorithm to augment outerplanar graphs by adding a minimum number of edges such that the resulting graph is biconnected and planar.*

7.4 Triconnectivity

7.4.1 Triconnecting Biconnected Outerplanar Graphs

We now consider the problem of how to augment an outerplanar graph G by adding a minimum number of edges such that the augmented graph G' is triconnected and planar. To this end we first restrict our attention to biconnected outerplanar graphs, which are cycles with non-intersecting chords. Every vertex with degree 2 must get an additional edge. Every biconnected outerplanar graph has at least two vertices v, with $deg(v) = 2$ [82]. Let K be the number of vertices of degree 2, then we will show that the number of edges to be added to achieve triconnectivity is equal to $\lceil \frac{K}{2} \rceil$, which is optimal.

It is not very difficult to construct an example where the outerplanar embedding (all vertices occuring on the outerface) has to be changed to obtain an optimal augmentation. As an example, consider the cycle on vertices $v_1, \ldots, v_n$ and let n vertices $w_1, \ldots, w_n$ be given with w_i connected to v_i and v_{i+1} $(1 \leq i < n)$. Vertex w_n is connected to v_1 and v_n. The resulting graph G is outerplanar and each vertex w_i must receive an augmenting edge. Hence $\lceil \frac{n}{2} \rceil$ edges must be added. Assume edge (w_i, w_j) is added, with $j > i$ and $j - i$ as small as possible. If $j = i + 1$ then it follows that $G - \{v_i, v_{i+2}\}$ is disconnected. If $j > i + 1$ then we cannot add an edge from w_{i+1} without crossing edge (w_i, w_j). Hence we cannot always maintain the original outerplanar embedding for constructing an optimal augmentation.

Let G be an outerplanar graph with a given outerplanar embedding. Assume the vertices are numbered $v_1, \ldots, v_n$ along the outerface. We now construct the dual graph G^* of G: every interior face F of G is represented by a vertex w_F in G^*. There is an edge between two vertices $(w_F, w_{F'})$ in G^*, iff F and F' share an edge in G. We also represent every vertex v of degree 2 by a vertex v in G^*, and add the edge (v, v_F), iff $v \in F$ in G. Observe that G^* is a tree and the leaves of G^* are the vertices of degree 2 in G.

Notice also that if v_i and v_j share an interior face and $j - i > 1$, then $G - \{v_i, v_j\}$ is disconnected. If $j - i > 1$, then there is a vertex v_k with $deg(v_k) = 2$ and $i < k < j$. Assume v_i and v_j share face F then vertex w_F has descendant leaf v_k. If we add

edges to G^* such that there is a path P from v_k to a vertex v_l, not a descendant of w_F and $w_F \notin P$, then it follows that v_i and v_j are not a cutting pair anymore. This observation leads to the following lemma:

Lemma 7.4.1 *If G^* is balanced, then applying* OUTERBICONNECT *to G^*, and then adding the corresponding edges in G gives a triconnected graph.*

Proof: Let G' be the augmented graph of G after applying OUTERBICONNECT to G^*. Consider two vertices v_i and v_j. They split the outerface of G into two paths: a path P_1 from v_{i+1} to v_{j-1}, and a path P_2 from v_{j+1} to v_{i-1}. If v_i and v_j do not share an interior face, then there is chord (v_k, v_l) inbetween, connecting P_1 with P_2. Hence $G - \{v_i, v_j\}$ is connected. Assume further that v_i and v_j share an interior face F and let $j > i+1$. Let v_k be a vertex with degree 2 and $i < k < j$. If we delete w_F in the augmentation G'^* of G^*, then the graph G'^* is still connected. Let (v_a, v_b) be an added edge in G'^*, with v_a a descendant of w_F and v_b not a descendant of w_F in G^*. This implies that $i < a < j$ and $b < i$ or $b > j$. There is an edge between P_1 and P_2 in G', hence v_i and v_j are not a cutting pair in G'. Since this holds for every pair v_i, v_j $(j > i)$, sharing an interior face, this proves the lemma. □

In the algorithm OUTERBICONNECT the order of the children of a C-node is changed from time to time, to preserve the planarity. How can we deal with this problem here, since the embedding of G^* (hence the order of the children) is fixed. The solution we present here is as follows: We root G^* at an arbitrary vertex of degree at least 2, say b_r. We assume that every internal vertex of G^* is a C-node, and every leaf of G^* is a B-node. However, the order of children of each vertex is fixed. After finding v with maximal $d(v)$ we walk towards b_r, until the current vertex is not the leftmost child of its parent, or the parent is b_r. If the parent is not b_r, then it has degree at least 3, which is sufficient to satisfy the leaf-connecting condition. Hence Lemma 7.3.2 still holds, and of course, planarity is preserved.

The only point is now when the graph is not balanced, or in other words: how to implement Stage 2? Recall that a 1-chain is a component of $T_{BC} - \{v\}$, containing only one leaf. The solution we present is to add edges between the 1-chains inside the shared interior face. More precisely, let v^* be the massive node in G^*, i.e., $d(v^*) > \lceil \frac{p}{2} \rceil - 1$. Root G^* at v^* and let $v(1), \ldots, v(\gamma)$ be the leaves from left to right in G^*, which are 1-chains. We now add an edge between 1-chain $v(\lceil \frac{\gamma}{2} \rceil - i)$ and $v(\lceil \frac{\gamma}{2} \rceil + i)$, for $1 \leq i < \delta$ with $\delta = d(v^*) - 1 - \lceil \frac{p}{2} \rceil$. Since $p \geq 2d(v^*) - \gamma$ (see Section 7.3.1) it follows that $\delta \leq \lceil \frac{\gamma}{2} \rceil - 1$.

Let G' be the augmented graph by adding the δ edges of A between the corresponding pendants. We do this by swapping the 1-chains inside interior face F and adding edges between vertices on the last blocks of the 1-chains inside face F^*, which preserves the planarity. For updating G^* we simply delete the two 1-chains from G^*. This decreases the number of leaves and the degree of v^* by two. Let G_1^* be the resulting tree with p' leaves, with $p' = p - 2\delta$, and let $d'(v)$ denote the d-value of v in G_1^*. The following variant of Lemma 7.3.3 can be proved.

Lemma 7.4.2 *For all nodes v of G_1^*, $d'(v) - 1 \leq \lceil \frac{p'}{2} \rceil$ holds.*

Proof: For v^* holds that $d'(v^*) - 1 = d(v^*) - 1 - 2\delta = \lceil \frac{p}{2} \rceil - \delta = \lceil \frac{p'}{2} \rceil$. Consider a $v \neq v^*$, and suppose that $d'(v) - 1 > \lceil \frac{p'}{2} \rceil$. Now $p \geq d(v^*) + d(v) - 2 = (d(v^*) - 1) + (d(v) - 1) > (\lceil \frac{p}{2} \rceil + \delta) + \lceil \frac{p'}{2} \rceil = (\lceil \frac{p}{2} \rceil + \delta) + (\lceil \frac{p}{2} \rceil - \delta) = 2\lceil \frac{p}{2} \rceil \geq p$, which is a contradiction. □

After this we apply the algorithm OUTERBICONNECT on the graph G'^* as described above. The corresponding edges are added in the outerface of G. Since this algorithm works in linear time and space, we obtain the following result:

Lemma 7.4.3 *There exists a linear time algorithm to augment a biconnected outerplanar graph G by adding a minimum number of edges such that the augmented graph G' is triconnected and planar.*

7.4.2 Triconnecting Outerplanar Graphs

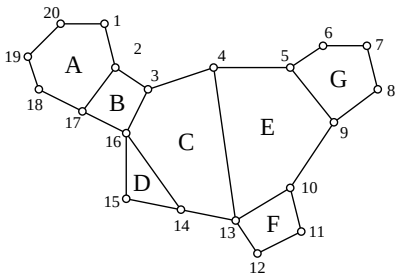
To triconnect outerplanar graphs G, which are not necessarily biconnected, we use the technique of the previous section. If G is not connected, then the algorithm CONNECT(G) is applied to connect the components with each other, so assume that G is connected.

We now recognize the blocks of G and build the BC-tree T_{BC}. Notice that if a block B_i contains two cutvertices, c_1 and c_2, then deleting c_1 and c_2 disconnects G. To be more precise about the blocks, which must receive an augmenting edge, we state the following lemma, which can be verified quite easily.

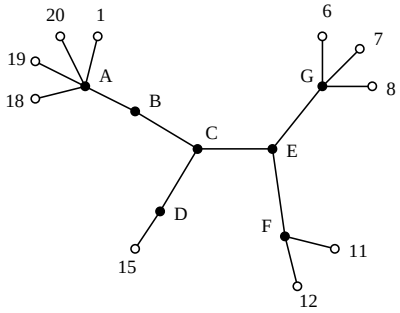
Lemma 7.4.4 *A block B_i in an outerplanar graph needs at least one augmenting edge in any triconnectivity augmentation if and only if there is a vertex of degree 2 in B_i or its corresponding B-node has degree 2 in T_{BC}.*

Let one arbitrary B-node b^* be the root of T_{BC}. Again the order of the children of a B-node is fixed. We assume that the left-to-right-order of the children of a B-node corresponds to walking counterclockwise around the outerface of the corresponding block.

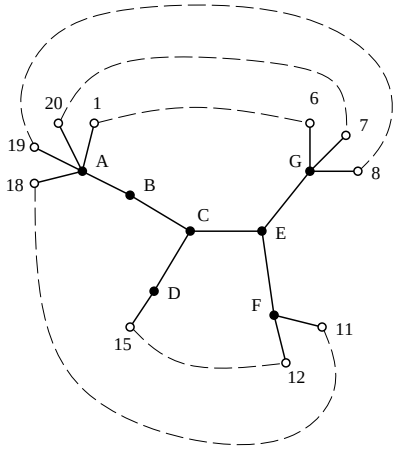
The idea is to start with the pendant blocks B_i, and to apply BICOUTERTRICONNECT on B_i. But now there must go at least two edges from vertices of B_i to vertices, not part of B_i. Indeed, for this we do not add the last augmentation edge to B_i. This implies that two or three vertices of B_i remain with degree 2. Let B_j be the block, corresponding to the parent of the parent of the B-node of B_i in T_{BC}. When applying OUTERBICONNECT on B_j, we represent B_i by a vertex v_{B_i} with k leaves as children in the dual graph B_j^* of B_j, where k is the number of vertices of degree 2.



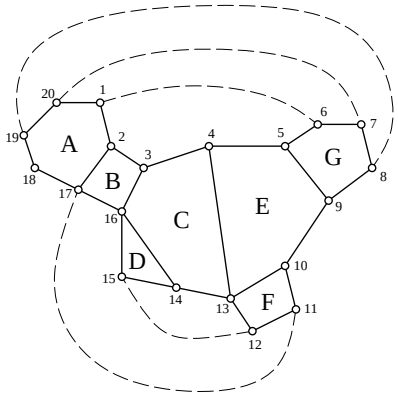
(a) The initial graph G.



(b) The tree G^*.



(c) Applying OuterBiconnect to G^*.



(d) The optimal triconnectivity augmentation.

Figure 7.3: Triconnecting an outerplanar graph.

Notice that always at least two edges have to go from B_i or descendants of B_i to vertices, not descendants of B_i. If pendant block B_i consists of only one single vertex v in G or if B_i has only one vertex v of degree 2, then we represent this by 2 leaves as children of v_{B_i}, because there has to go two edges from v to vertices, not part of B_i. Since every vertex v_{B_i} of a pendant block B_i has 2 or 3 leaves, no new 1-chains or massive nodes are introduced in B_j^*.

After visiting the pendant blocks we visit these blocks B_j, for which all descendant B-nodes are visited by OUTERBICONNECT. We compute the dual graph B_j^* in which every descendant B-node of the corresponding block B_i is represented by a vertex v_{B_i}. v_{B_i} has k children, where k is the number of edges, which has to go to the remaining part of the graph. We apply OUTERBICONNECTon B_j^*. In the same way, two or three edges have to go from B_j or its descendants to the remaining part of the graph. We apply this algorithm bottom-up until we are at the root of T_{BC}. In each step we visit a block B_i and the total time is linear in the number of vertices of B_i+ the number of children of the B-node of B_i. Combining this observation with Lemma 7.4.1 yields the following theorem.

Theorem 7.4.5 *There is a linear time and space algorithm to augment an outerplanar graph G by a minimum number of edges such that the resulted graph is triconnected and planar.*

7.5 Triangulating Outerplanar Graphs

7.5.1 Triangulating One Face of a Planar Graph

Now we consider the following problem: given a planar graph G with a planar embedding, triangulate a face F of G while minimizing the maximum degree. We show by using dynamic programming that this problem can be solved exactly in polynomial time. We use this algorithm to triangulate all interior faces of an outerplanar graph while minimizing the maximum degree. Dynamic programming has been applied for several related triangulation algorithms of polygons, described by Edelsbrunner et al. [24, 25, 26]. They present polynomial time algorithms (using dynamic programming) for triangulating polygons while maximizing the minimum height, minimizing the maximum slope, or minimizing the maximum eccentricity. Unfortunately, these algorithms cannot be translated directly to our problem.

Let a face F of G on k vertices $v_1, \ldots, v_k$ be given (numbered clockwise around the face). Every vertex v_i of F has degree ≥ 2. Some edges (v_i, v_j) with $v_i, v_j \in F$ may occur. These edges are called *forbidden edges* and are embedded outside F. Notice that in every triangulation the vertices v_1 and v_k have a common neighbor v_p $(2 \leq p \leq k-1)$ inside face F, which splits the face F into two faces F_1 (with vertices $v_1, \ldots, v_p$) and F_2 (with vertices $v_p, \ldots, v_k$). If (v_1, v_p) or (v_p, v_k) is already present outside F, (thus $2 < p < k-1$), then this triangulation is not possible, since it would imply multiple edges. Otherwise, we recursively triangulate the faces

F_1 and F_2. Let F_1' and F_2' denote the triangulated faces of F_1 and F_2, then the highest degree in F_1' and F_2' is important, but moreover, since F_1 and F_2 share v_p, the increase of $deg(v_p)$ in F_1' and F_2' must be added to $deg(v_p)$ in F. When we examine triangulations of a face F_{ij}, formed by vertices $v_i, v_{i+1}, \ldots, v_j$, we inspect the different values of increases of $deg(v_i)$, $deg(v_j)$ and $deg(v_p)$ in F_1 and F_2. See figure 7.4.

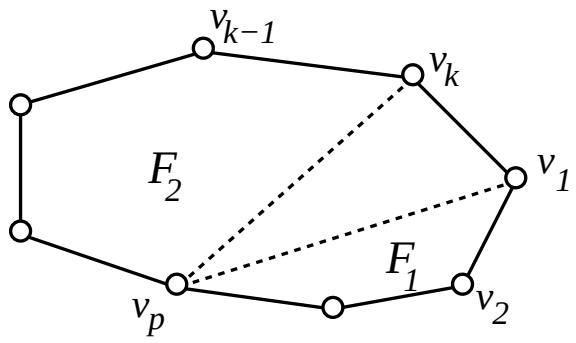


Figure 7.4: Recursive definition of the triangulation of one face.

Notice that when $p = 2$ (or $k-1$) then the edge (v_1, v_2) (or (v_{k-1}, v_k)) is already present, hence need not to be added. To deal with this, we delete the edges (v_i, v_{i+1}) $(1 \leq i < k)$ and decrease $deg(v_i)$ by 2. Let $incr(v_i)$ denote the increase of $deg(v_i)$ when triangulating F_1 to F_1' (assuming $v_i \in F_1$). For a triangulation of a face F_{ij} we have to store the different increases of $deg(v_i)$ and $deg(v_j)$ in a table. Let $D[i, j, i', j']$ be the maximum degree of F_{ij} by a triangulation with $incr(v_i) = i'$ and $incr(v_j) = j'$. If such a triangulation does not exist, $D[i, j, i', j'] = \infty$. A simple analysis shows the following recursive formula if $i < j - 1$:

$$
\begin{aligned}
D[i,j,i',j'] := \min \{ \max \{ & \; D[i,p,i'-1,p'], D[p,j,p'',j'-1], \\
{\scriptstyle i<p<j} \quad & \; deg(v_i) + i' + 1, \\
{\scriptstyle p',p''} \quad & \; deg(v_j) + j' + 1, \\
{\scriptstyle (i,p) \text{ and } (j,p)} \quad & \; deg(v_p) + p' + p'' + 2 \\
{\scriptstyle \text{not forbidden}} \quad & \; \} \\
\} &
\end{aligned}
$$

If $i = j - 1$ then for all $i', j' \geq 0$: $D[i, j, i', j'] = \max\{deg(v_i), deg(v_j)\}$. We want to compute $\min_{i',j'}\{D[1, k, i', j'], deg(v_1) + i' + 1, deg(v_k) + j' + 1\}$. We do this by using dynamic programming, based on the above formulas, and some other ideas which help to decrease the running time of the algorithm.

Theorem 7.5.1 *There is an exact $O(k^3 \Delta(G) \log \Delta(G))$ time algorithm to triangulate one face of k vertices of a graph G such that the maximum degree of the triangulation G' is minimized.*

Proof: Let a face F be given. By Theorem 5.3.2 and 6.2.1, we know that we can triangulate F such that the maximum degree is at most $\lceil \frac{3}{2}\Delta(G) \rceil + O(1)$. We do not compute all values of $D[i,j,i',j']$ as this would be too time-consuming, but instead apply binary search on the maximum degree. So we must test for $O(\log \Delta(G))$ values of K whether a triangulation with maximum degree $\leq K$ exists. Let $D_K[i,j,i',j']$ $= true \iff D[i,j,i',j'] \leq K$. Suppose K is fixed. Note that it is sufficient to know: for all i', $1 \leq i' \leq K - deg(v_i)$, the smallest value of j' such that $D_K[i,j,i',j'] = true$ and for all j', $1 \leq j' \leq K - deg(v_j)$, the smallest value of i' such that $D_K[i,j,i',j'] = true$. Denote these smallest values with $F_K[i,j,i']$ and $G_K[i,j,j']$. (If such j' or i' not exist, $F_K[i,j,i'] = \infty$, or $G_K[i,j,j'] = \infty$.)

Now $D_K[i,j,i',j'] = true$, if and only if there exists a p, $i < p < j$, (v_i, v_p) and (v_j, v_p) not forbidden and $F_K[p,j,K - deg(v_p) - F_K[i,p,i'] - 2] \leq j'$. (The increase of the degree of v_p in face F_{pj} may not be larger than $K - deg(v_p) - F_K[i,p,i'] - 2$. $F_K[i,p,i']$ edges are used for face F_{ip}, and there are edges $(i,p),(p,j)$.) So $F_K[i,j,i'] = \min\{F_K[p,j,K - deg(v_p) - F[i,p,i'] - 2] | i < p < j, (i,p),(p,j)$ not forbidden$\}$. The latter formula allows us to compute all values of $F_K[i,j,i']$ $(1 \leq i \leq k, 0 \leq i \leq p)$ in $O(k^3 K)$ time. When F_K is computed, one easily determines whether a triangulation with maximum degree $\leq K$ exists. Using binary search on K, we can implement it such that the total runtime becomes $O(k^3 \Delta(G) \log \Delta(G))$. □

7.5.2 Triangulating Outerplanar Graphs

In this section we consider the problem of triangulating the interior faces of an outerplanar graph G. If G is biconnected, then the augmentation G' is a mop. Every triangulated polygon is a mop [88]. Also recognizing outerplanar graphs is based on recognizing mops [82].

First, we remark that we can treat all blocks separately. The only relevant information for a block B for $G - B$ is cutvertex v, belonging to both B and $G - B$. If B is connected by a bridge with $G - B$, then no information of the triangulation of B is relevant for $G - B$. Hence assume that G is bridge-connected, otherwise we can apply the following method to the bridge-connected components of G. We construct a tree T, where every interior face F of G is represented by a node v_F in T. We have two types of edges:

- add an edge (v_{F_1}, v_{F_2}), if F_1 and F_2 share an edge. By definition, F_1 and F_2 belong to the same block.
- add an edge (v_{F_1}, v_{F_2}), if F_1 and F_2 share a vertex in G, and F_1 and F_2 belong to different blocks of G.

We now have a tree where each node v_F represents a face F. Root T at an arbitrary node v_F. Let the parent of node v_{F_a} be v_{F_b}, then we call F_b the *parent-face* of F_a.

The common edge (or common node if F_a and F_b belong to different blocks) is called the *parent-edge* (or *parent-node* respectively) of F_a. For every edge (v_i, v_j) of face F_a that is a parent-edge of some other face F_b, define $D[v_i, j, i', j']$ to be the maximum degree if F_b and all their descendants in H are triangulated, such that the degree increase of v_i is at most i', and the degree increase of v_j is most j'. For every vertex v_i of face F_a that is parent-node of at least one other face F_b, define $D[v_i, i']$ to be the maximum degree, if these faces F_b and all their descendants in H are triangulated, such that the degree increase of v_i is at most i'.

We now use a procedure, similar to the one, described in Section 7.5. Apply binary search on the maximum degree K. For a fixed K, define $F_K[i, j, i']$ and $G_K[i, j, j']$ as in the proof of Theorem 7.5.1. Define $H_K(i) = \min\{i' | D[i, i'] \leq K\}$. Compute the tables or values for F_K, G_K and H_K bottom up in the tree H. When we deal with a face, we add $H_K(i)$ to the degree of v_i. Let the tables $F_K[i, j, i']$ and $G_K[i, j, j']$ play the same role as they played in the proof of Theorem 7.5.1. With minor modifications of this algorithm, one can compute the table $F_K[v'_i, v'_j, i']$ and $G_K[v'_i, v'_j, j']$ for the parent-edge (v'_i, v'_j) of F_a, or compute the contribution to $H_K[v'_i]$ for the parent-node v'_i of F_a. We omit some easy details here. Applying a similar proof as in Theorem 7.5.1, the following result can be obtained:

Theorem 7.5.2 *There is an $O(n^3 \Delta(G) \log \Delta(G))$ algorithm to triangulate all interior faces of an outerplanar graph while minimizing the maximum degree.*

Chapter 8

Conclusions

In Part B we developed efficient algorithms for several augmentation problems for planar and outerplanar graphs. For outerplanar graphs G, optimal $O(n)$ time and space algorithms are given for making G bridge-connected, biconnected or triconnected by adding a minimum number of edges, while preserving planarity. In all three cases, the degree of every vertex increases by at most 2. Triangulating all interior faces of an outerplanar graph while minimizing the maximum degree can be achieved in polynomial time as well.

For planar graphs we presented augmentation algorithms, adding only a constant times the minimum number of edges to preserve the planarity and different connectivity constraints. In Figure 8.1 an overview is given of the presented algorithms of Part B. Except for the algorithm TRIANGULATE, the maximum increase of the degree is equal to the worst-case lower bound. In TRIANGULATE the maximum increase of the degree is at most an additive constant from optimal. To our knowledge, this is the first time that augmentation problems are considered with the additional planarity constraint. The augmentation problems arise in the area of graph drawings, in which the input planar graph must be biconnected, triconnected or triangulated. In Chapter 13 we will also present an algorithm for 4-connected triangular planar graphs, but unfortunately, we cannot augment every planar graph by adding edges such that it becomes a 4-connected triangular planar graph. Therefore, we did not consider this problem in Part B.

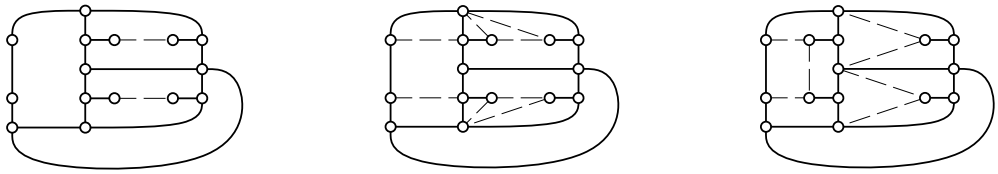
Augmentation algorithms, not dealing with the planarity constraint, are becoming a core area in the literature on graph algorithms. We mention here the work on augmentation algorithms, with respect to vertex-connectivity [27, 36, 51, 52, 53, 72, 97, 114, 117] and edge-connectivity [33, 72, 85, 115, 118]. Among these papers, optimal linear time algorithms are given for augmenting graphs with respect to bridge-connectivity [27], biconnectivity [53, 97] and triconnectivity [52].

However, up to now we did not consider the problem of augmenting an arbitrary non-biconnected planar graph to a triconnected planar graph by adding a minimum number of edges. As announced in Chapter 5, the decision variant of this problem is still open. One approach to solve this problem is to make the graph biconnected, e.g.

algorithm	time complexity	space compl.	input graph G	added edges	lower bound for $\Delta(G')$	$\Delta(G')$
CONNECT	$O(n)$	$O(n)$	disconnected	optimal	$\Delta(G)+2$	$\Delta(G)+2$
BICONNECT	$O(n)$	$O(n)$	connected	—	$\Delta(G)+2$	$\Delta(G)+2$
2*OPTBICONNECT	$O(n \cdot \alpha(n,n))$	$O(n)$	connected	2*OPT.	$\Delta(G)+2$	$\Delta(G)+2$
2*OPTBRIDGECONNECT	$O(n \cdot \alpha(n,n))$	$O(n)$	connected	2*OPT.	$\Delta(G)+2$	$\Delta(G)+2$
TRICONNECT	$O(n)$	$O(n)$	biconnected	$\frac{3}{2}$*OPT.	$\lceil \frac{3}{2}\Delta(G) \rceil$	$\lceil \frac{3}{2}\Delta(G) \rceil$
TRIANGULATE	$O(n)$	$O(n)$	triconnected	optimal	$\Delta(G)+3$	$\Delta(G)+8$

Figure 8.1: Overview of algorithms in Part B.

by applying 2*OPTBICONNECT, and then making the graph triconnected, e.g. by applying TRICONNECT. This of course does not necessarily lead to a triconnected planar graph with a minimum number of edges, but moreover, there are planar graphs for which any optimal augmentation to a biconnected planar graph does not lead to an optimal augmentation to a triconnected planar graph. In Figure 8.2 an example is given. Also it is open whether applying 2*OPTBICONNECT and TRICONNECT leads to a solution for the general planar triconnectivity problem with the number of added edges only a small constant times optimal.



(a) Optimal biconnectivity augmentation

(b) From biconnectivity to triconnectivity

(c) Optimal triconnectivity augmentation

Figure 8.2: Counterexample for triconnecting non-biconnected planar graphs.

When the input graph G is embedded, i.e., given as a plane graph G, then the augmentation problem with respect to biconnectivity becomes easily solvable. In a plane graph every pendant is already assigned to a face. In each face we have to find an optimal matching between the pendants, while preserving planarity. This problem is equal to applying OUTERBICONNECT to each face of the graph, leading to an optimal linear time algorithm for biconnecting general plane graphs while preserving planarity.

When the plane graph G must be augmented to a triconnected plane graph, then the problem is more complicated. We cannot simply use the algorithm for triconnecting outerplanar graphs to every face F of G. Because when we triconnect the components inside F, then also edges can be added outside F. We have to place

these edges outside F in such a way that triconnecting the adjacent faces of F can also be done by adding a minimum number of edges. Notice also that Lemma 5.1.2 does not hold anymore for a plane graph. Though it is not very hard to change the algorithm TRICONNECT such that it works within $\frac{3}{2}$ times optimal in $O(n)$ time for plane graphs, it seems pretty hard to obtain an optimal solution in polynomial time.

We also considered the problem of triangulating planar graphs G. We proved that triangulating a biconnected planar graph G such that the maximum degree is $\leq K$ is NP-hard. Jansen proved that triangulating a plane graph, with coordinates given for every vertex, such that the triangulated plane graph has maximum degree $\leq K$ is also NP-hard [55]. On the positive side, we presented a linear time algorithm for triangulating a planar graph, such that the maximum degree is at most an additive constant from an existential lower bound. For triconnected planar graphs, this increase of the degree is at most a constant from optimal, and it is interesting to close this gap, or in other words: is the problem of triangulating a planar graph such that the maximum degree is $\leq K$ also NP-hard for triconnected planar graphs?

Part C

Drawing Planar Graphs

Chapter 9

Drawing Algorithms

During the last decades many drawing algorithms have been described in the literature, both from the theoretical and the practical point of view. The problem of *nicely* drawing a graph in the plane has received increasing attention due to the large number of applications. Examples include VLSI layout, algorithm animation, visual languages and CASE tools. In Chapter 1 some more detailed examples are presented. Several representations are possible. Typically, vertices are represented by distinct points in a line or plane, and are sometimes restricted to be grid points. (Alternatively, vertices are sometimes represented by line segments [58, 89, 96, 104].) Edges are often constrained to be drawn as straight lines [15, 31, 32, 34, 58, 80, 89, 96, 98, 104] or as a contiguous line segments, i.e., when bends are allowed [100, 102, 105, 106]. The objective is to find a layout for a graph that optimizes some cost function such as area, minimum angle, number of bends, or that satisfies some other constraint. In [18], Di Battista, Eades, Tamassia & Tollis give a good annotated bibliography with more than 250 references including several pointers to applications in which drawing algorithms appear. In this section we describe several techniques in more detail, which deal with undirected planar graphs. We do not have the intention of being complete in our overview, but we try to give the more recent general techniques that leads to interesting theoretical and practical bounds. The algorithms serve as a starting point for the new results, presented in Part C.

9.1 Straight-line Drawings

By a result, independently obtained by Wagner [113], Fáry [31] and Stein [99], every planar graph can be drawn in the plane with straight-line edges. This is also obtained by the following constructive proof, due to Read [92]: assume G is a triangular planar graph. (If G is not triangulated, then in linear time we can add edges such that G is triangulated, see Chapter 6.) By planarity one can verify that every vertex v of G has at least one neighbor u such that u and v have exactly two neighbors in common.

If v does not belong to the outerface, then we contract edge (u, v), i.e., we add edges from u to all neighbors of v, which are not a neighbor of u yet, and remove vertex v. We draw the reduced graph G' with $n-1$ vertices recursively. Afterwards we remove the added edges from u to the (previous) neighbors of v and place v inside the corresponding face such that v is *visible* from its previous neighbors. This gives a drawing with straight-line edges. Using the observation that every planar graph G has a vertex v with $deg(v) \leq 5$, we can implement this algorithm such that it runs in $O(n)$ time, requiring $O(n^2)$ space [92].

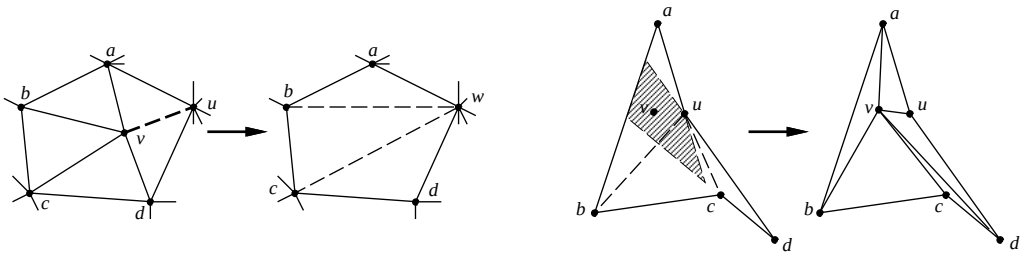


Figure 9.1: The basic concept of Read's drawing algorithm.

A drawback of this algorithm is that vertices are placed on real coordinates. Moreover, it sometimes distributes the vertices unevenly, thus requiring high-resolution display devices for a drawing, since the vertices can be placed very close to each other (this is called *clustering*).

Using a more advanced and deeper characterization of planar graphs one can draw every triangular planar graph planar with straight lines such that the vertices are placed on grid coordinates. A method for this, described by de Fraysseix, Pach & Pollack [34] will be outlined in Chapter 10. Independently, Schnyder [98] obtained a linear time algorithm to draw a triangular planar graph on an $(n-2) \times (n-2)$ grid, based on a novel representation of triangulated planar graphs, called the *barycentric representation*. The vertices are widely distributed on the grid, and there is a lower bound on the minimum edge length. Planar drawings require an $\Omega(n^2)$ area in the worst-case [34]. However, a drawback of all these drawing algorithms is that the minimum angle between lines can be very small, which makes the drawing unattractive. In [80], Malitz & Papakostas showed that every d-planar graph G can be drawn in the plane such that the minimum angle is at least α^d radians, where $0 < \alpha < 1$ is a constant (approximately 0.15). This follows by the remarkable result that one can represent every vertex v of a triangular planar graph G by a closed disc $D(v)$ such that if $(u, v) \in G$, then the discs $D(u)$ and $D(v)$ *touch* each other (a so-called *disc-packing*). A disc packing D induces a planar graph G in the obvious way: place a vertex at the center of each disc and for each pair of touching discs, create an edge between the vertices at the centers of the two discs. Unfortunately, the proof is non-constructive, and the minimum angle is quite small. A polynomial-time

approximation of a disc-packing realization, and a nice generalization to triconnected planar graphs is announced by Mohar (personal communication).

On the other hand, G can be drawn non-planar with straight lines such that the minimum angle is at least $\Omega(\frac{1}{d})$, by a result of Formann et al. [32]. They also proved that deciding whether a graph with maximum degree 4 can be drawn with minimum angle $\geq \frac{\pi}{2}$ is NP-complete. In Kant [66] it it is proved that deciding whether a biconnected planar graph can be drawn **planar** with the minimum angle $\geq K$ is NP-complete.

9.2 Convex Drawings

Another way for drawing planar graphs is by drawing it with convex faces, i.e., a planar straight-line drawing such that all internal face boundaries are convex polygons. This problem of obtaining convex drawings was first studied in more detail by Tutte [110]. Tutte also gave a simple method for finding a convex drawing. Here the external face is any prescribed convex polygon and the position $P(v) = (x(v), y(v))$ of each vertex v is given by

$$x(v) = \frac{1}{deg(v)} \sum_{(v,w)\in E} x(w) \qquad\qquad y(v) = \frac{1}{deg(v)} \sum_{(v,w)\in E} y(w)$$

Using Gaussian elimination the coordinates can be found by a simple algorithm, working in $O(n^3)$ time, and requiring $O(n^2)$ space. Using a more sophisticated sparse matrix elimination scheme which relies on the planar separator theorem, this leads to an $O(n\sqrt{n})$ algorithm, requiring only $O(n \log n)$ space [79].

Thomassen [109] characterized the class of planar graphs that admit a convex drawing. We do not give the full characterization here, but it can be described as the class of biconnected planar graphs, where "almost" all separation pairs are part of the outerface. Based on this characterization, Chiba et al. [14] present an $O(n)$ time recursive algorithm, which can be outlined as follows: assume an outerface F of the biconnected planar graph G has been chosen, assume that all vertices $v \in F$ are placed, and that vertices of degree two are eliminated, while connecting their neighbors. (The vertex of degree 2 can later be placed on the straight-line segment joining the two vertices adjacent to it.) The remaining part of the algorithm is as follows:

ConvexDraw(G); { assume $n \geq 4$, otherwise G is drawn as a triangle }
  let v be a vertex, which is a corner point of the outerface;
  let $G' := G - \{v\}$;
  let $B_1, \ldots, B_p$ be the blocks of G';
  let for each B_i, v_i and v_{i+1} be two cutvertices of B_i, with $(v, v_i), (v, v_{i+1}) \in E$;
  place the vertices on the outerface of every B_i on a convex area
  inside triangle v, v_i, v_{i+1} such that:

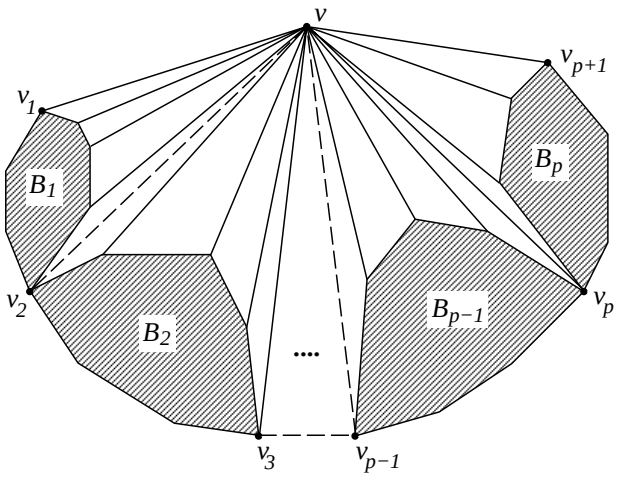


Figure 9.2: Drawing a planar graph convex (from [14]).

- the vertices adjacent to v are corner points of a convex polygon;
- the other vertices are on straight-line segments of this polygon;

for each block B_i **do** ConvexDraw(B_i) **rof**;
End ConvexDraw

See Figure 9.2 for an idea of the algorithm, and see [14] for a complete description. In [13], Chiba et al. extended the algorithm to general planar graphs such that the outerface of each drawn triconnected component is drawn as a convex polygon. Unfortunately, experiments showed that this algorithm sometimes distributes unevenly, thus requiring high-resolution display devices.

9.3 Drawing Planar Graphs Using the *st*-Numbering

Several drawing algorithms for planar graphs are based on the *st-numbering*. An *st*-numbering is a numbering of the vertices $v_1, \ldots, v_n$ of G such that $(v_1, v_n) \in E$ and every vertex v_i $(1 < i < n)$ has edges to vertices v_k and v_l, with $k < i < l$. This is only possible when G is biconnected, hence assume G is biconnected. (Otherwise, dummy edges can be added to G to make G biconnected while preserving planarity. See Chapter 4 for an extensive investigation of this augmentation problem. The dummy edges are suppressed in the final drawing.) The *st*-numbered graph is called an *st-graph* .

Let the edges (v_i, v_j) be directed $v_i \to v_j$, if $j > i$. Let a planar embedding of G be given. G has exactly one source (vertex $s = v_1$) and one sink (vertex $t = v_n$).

The orientation is also called a *bipolar orientation* [96]. We first restate and extend Theorem 2.2.2.

Theorem 9.3.1 ([104, 122]) *All the entering edges of any vertex in G appear consecutively in the rotation around v, as do all the exiting edges, in any embedding of G. The boundary of every face consists of exactly two directed paths in G.*

We define *left*(e) (*right*(e)) to be the face to the left (right) of e. The face separating the incoming from the outgoing edges in the clockwise direction is called *left*(v) and the other separating face is called *right*(v). The highest and lowest numbered vertex of the face is denoted by *high*(F) and *low*(F), see Figure 9.3.

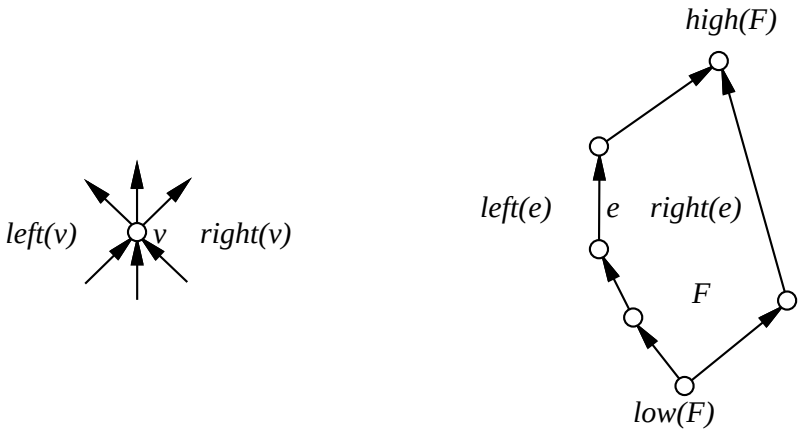


Figure 9.3: Properties of planar st-graphs.

The edges of the dual graph G^* of G are directed as follows: if F_l and F_r are the left and right face of some edge (v, w) of G, then the dual edge is directed from F_l to F_r if $(v, w) \neq (s, t)$ and from F_r to F_l, if $(v, w) = (s, t)$. The orientation of G^* is also a bipolar orientation, with source s^* the right face of (s, t) and sink t^* the left face of (s, t).

9.3.1 Visibility Representation

In a *visibility representation* of a planar graph vertices are represented as horizontal segments and edges as vertical segments such that each edge segment has its endpoints on the segments associated with its incident vertices and does not cross any other vertex segment. Otten & van Wijk [89] introduced this representation, which has applications to circuit schematics, and showed that every planar graph admits one. Here we describe in more detail the algorithm of Rosenstiehl & Tarjan [96]. (This algorithm was found independently by Tamassia & Tollis [104].)

For drawing the graph we first construct an st-numbering, and direct the edges from s to t. Then we compute the dual graph G^*, and direct the edges in G^* from source s^* to sink t^* as described above. For each vertex v of G, let $d(v)$ denote the length of the longest path from s to v. Let $D = d(t)$. For each vertex F of G^*, let $d^*(F)$ denote the number of nodes on the longest path from s^* to F. Let $d^*(s^*) = 1$ and $D^* = d^*(t^*)$. These lengths can easily be computed in linear time and give the coordinates in the visibility representation. The width of the drawing becomes $D^* - 1$ and the height becomes D.

VISIBILITY(G);
  compute an st-numbering for G;
  construct the planar st-graph G and its dual G^*;
  compute $d(v)$ for all vertices in G and $d^*(F)$ for vertices in G^*;
  for each vertex $v \neq s, t$ **do**
    draw a horizontal line between $(d^*(\mathit{left}(v)), d(v))$ and $(d^*(\mathit{right}(v)) - 1, d(v))$;
  rof;
  for vertex s, draw a horizontal line between $(0, 0)$ and $(D^* - 1, 0)$;
  for vertex t, draw a horizontal line between $(0, D)$ and $(D^* - 1, D)$;
  for each edge $(u, v) \neq (s, t)$ **do**
    draw a line between $(d^*(\mathit{left}(u, v)), d(u))$ and $(d^*(\mathit{left}(u, v)), d(v))$;
  rof;
  for edge (s, t), draw a line between $(0, 0)$ and $(0, D)$;
END VISIBILITY

Theorem 9.3.2 ([96, 104]) *The algorithm* VISIBILITY(G) *computes in linear time a visibility representation of a biconnected planar graph on a grid of size at most* $(2n - 5) \times (n - 1)$.

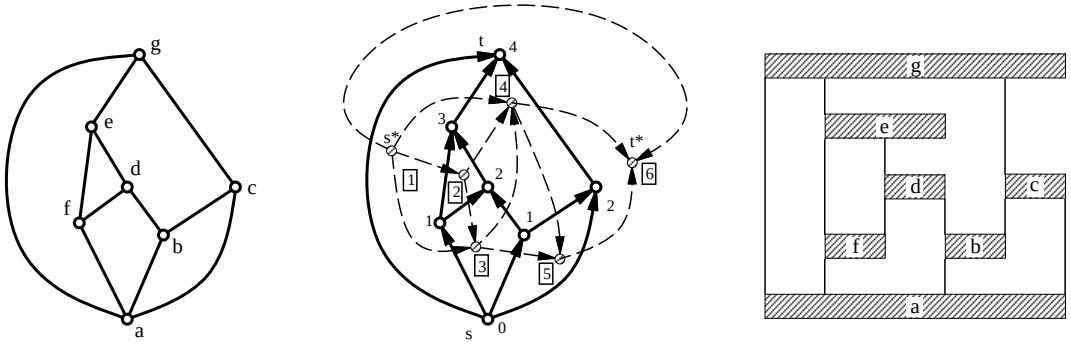


Figure 9.4: Constructing a visibility representation.

The steps are also illustrated in Figure 9.4. It follows that the vertices v of G, $v \neq s, t$, of degree 2, are represented by a single point in the visibility representation

(e.g., see Figure 9.7). In Chapter 13 it is shown that the visibility representation of a 4-connected planar graph can be constructed on a grid of size at most $(n-1)\times(n-1)$. Using this result it is shown in Chapter 14 that every planar graph can be represented by a visibility representation on a grid of size at most $(\lfloor \frac{3}{2}n \rfloor - 3)\times(n-1)$. To achieve a linear time complexity for this algorithm, we show that constructing the 4-block tree of a triangulated planar graph can be done in linear time.

This simple algorithm has also been applied successfully by Di Battista & Tamassia [20], who showed that by using this type of construction we can construct in $O(n)$ time an *upward drawing* of G on a grid of size at most $(2n-5)\times(n-1)$ such that there are at most $4n-10$ bends. Recall that a drawing is upward if for every directed edge $(v,w) : y(w) > y(v)$. The result for upward drawings is obtained by constructing a visibility representation of G, and then collapsing every horizontal segment into one point. Except for $n-1$ edges, all other edges will get at most two bends. There are at most $2n-5$ edges, leading to at most $4n-10$ bends. This number can be improved to $\frac{1}{3}(10n-31)$ bends [20], and even to $2n-5$ bends [22]. Di Battista, Tamassia & Tollis used the algorithm VISIBILITY(G) also for computing a *constrained visibility representation* of graphs. In a constrained visibility representation the edges of some given edge-disjoint paths are vertically aligned [22].

9.3.2 Orthogonal Drawings

The algorithm VISIBILITY(G) can also be used to construct an orthogonal drawing of G. An orthogonal drawing is a planar drawing where vertices are represented by points and every edge is an alternating sequence of horizontal and vertical segments. The heuristic algorithm of Tamassia & Tollis [105] for minimizing the number of bends in orthogonal drawings is as follows:

ORTHOGONAL(G);
  , := VISIBILITY(G);
  for each $v \in$, , convert it to a node as indicated in Figure 9.5 **rof**;
  optimize drawing , by the following optimizations (see Figure 9.6):
    $T1$: move consecutive bends of 90^o and 270^o in one edge by straight line;
    $T2$: rotate vertex, if all incident edges have an angle of 90^o;
    $T3$: replace a vertex of degree ≤ 3 to incident bend;
  construct a grid embedding;
END ORTHOGONAL

Theorem 9.3.3 ([105]) *There is a linear time algorithm to compute an orthogonal representation of a biconnected planar graph G with at most $n+2$ bends if $\Delta(G) \leq 3$, and with at most $2n+4$ bends if $\Delta(G) \leq 4$.*

In Figure 9.7 an example is given of the complete process of constructing an orthogonal drawing, using a visibility representation. The same bound on the number

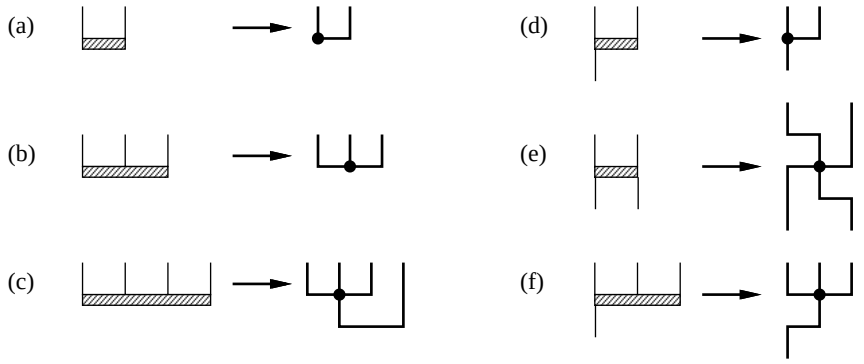


Figure 9.5: Converting a visibility representation into an orthogonal drawing.

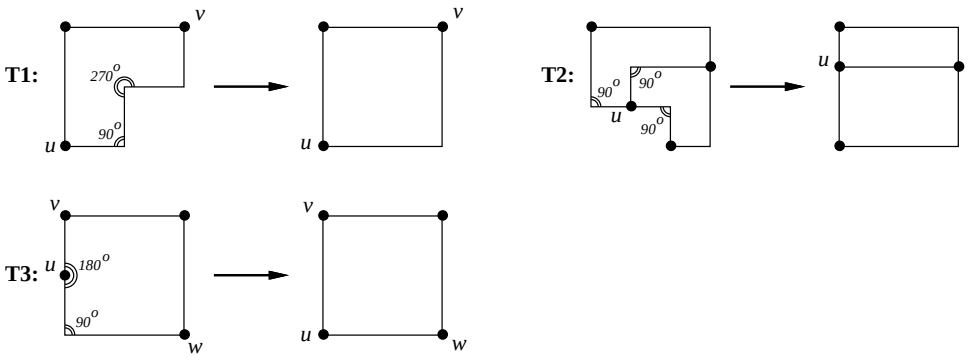


Figure 9.6: Optimizing the orthogonal drawing.

of bends was already obtained by Storer [100]. He presented several polynomial-time heuristics for minimizing the number of bends for 3- and 4-planar graphs. Storer also proved several optimization problems concerning grid size and number of bends to be NP-complete. Storer showed that there are 3-plane graphs for which any orthogonal layout requires an $\frac{n}{2} \times \frac{n}{2}$ grid. A lower bound of $\frac{n}{2} + 1$ bends for drawing 3-planar graphs orthogonal is included in [100] as well. There are also 4-plane graphs requiring an $(n-1) \times (n-1)$ grid, and there are 4-plane graphs, requiring $2n-2$ bends in any orthogonal drawing [100]. On the other hand, Tamassia [102] showed that given a 4-plane graph G, the problem of computing an orthogonal drawing with minimum number of bends can be solved in $O(n^2 \log n)$ time. This fascinating result follows by reducing the problem to an instance of a maximum-flow problem.

Indeed, this result only holds for embedded planar graphs. To compute the precise minimum number of bends, all different planar embeddings have to be checked. Very recently, Di Battista et al. presented polynomial-time algorithms for bicon-

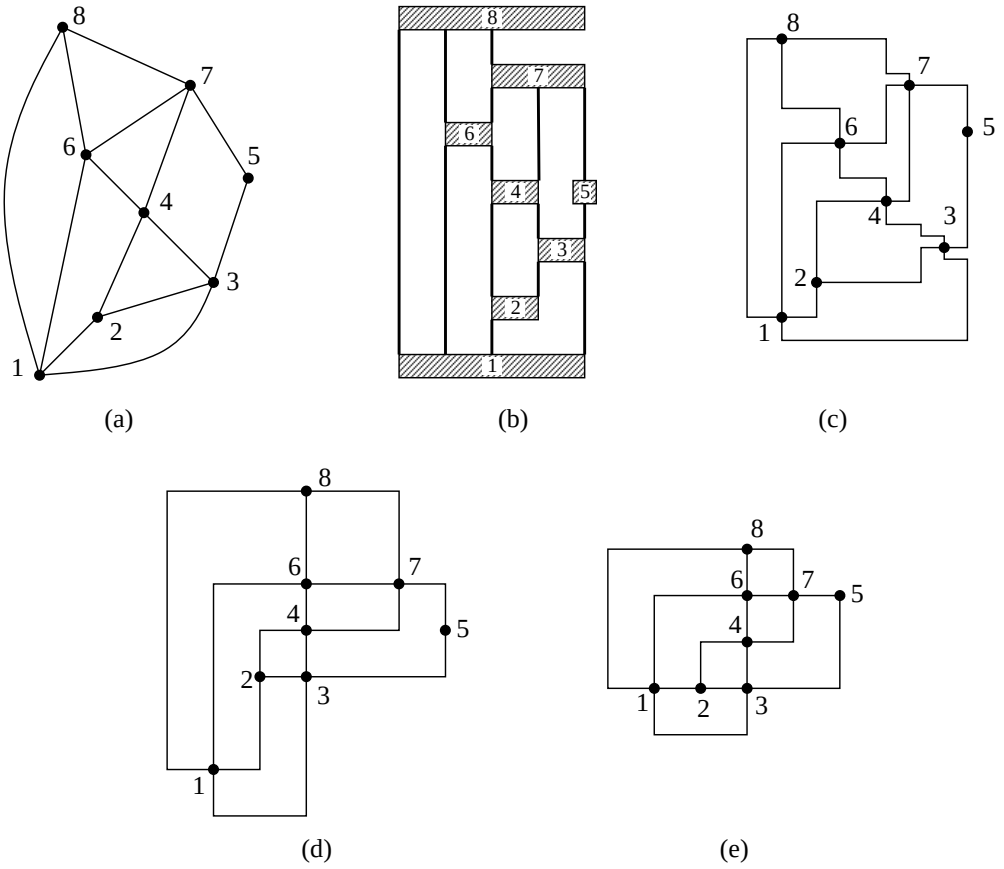


Figure 9.7: Example of constructing an orthogonal drawing (from [105]).

nected 3-planar graphs and series-parallel graphs, delivering an orthogonal drawing with a minimum number of bends [19]. Here the initial embedding is not fixed.

9.4 Overview of Part C

In Chapter 10 we refine the canonical ordering, introduced in Section 2.5 to a so-called *leftmost canonical ordering* or *lmc*-ordering. Every *lmc*-ordering is also an *st*-numbering. If the input graph is triangulated, then the *lmc*-ordering is a canonical ordering for triangulated planar graphs, as introduced by de Fraysseix, Pach & Pollack [34]. The *lmc*-ordering leads to a linear time drawing framework, and can be used for a broad range of drawing representations of (triconnected) planar graphs. We show in Chapter 10 that the planar straight-line grid drawing algorithm of de Fraysseix, Pach & Pollack [34] can be implemented to run in linear time, and the grid size can be decreased to $(n-2) \times (n-2)$. (In [15] an alternative

linear time implementation is described, requiring an $(2n-4) \times (n-2)$ grid, where the planar graph has to be triangular.) For our algorithm it is already sufficient that the input graph is triconnected instead of triangulated. In particular we show that, using the *lmc*-ordering, all interior faces can be made convex. This algorithm outperforms the algorithms of [111, 14] and is not only of theoretical interest, but also leads to more pleasing pictures (see Figure 10.3). It also gives a new and rather simple proof of the fact that every triconnected planar graph admits a convex planar drawing. It is also shown that every triconnected planar graph G can be drawn in the plane on an $(2n-6) \times (3n-6)$ grid with minimum angle $> \frac{1}{d-2}$ radians, such that in each edge there are at most 3 bends, and at most $5n-15$ bends in total. The vertices and bends are placed on grid points. This seems to be the first practical drawing algorithm for drawing planar graphs with vertices vertices represented by grid points and edges by polylines, having very reasonable bounds on the grid size, on the number of bends, and on the minimum angle.

In Chapter 11 we prove that if G is triconnected and 4-planar, then G can be drawn with at most $\lceil \frac{3}{2}n \rceil + 4$ bends on an $n \times n$ grid. This improves the best known bound of $2n+4$ bends considerably in the triconnected case. We also present a class of triconnected 4-plane graphs, for which any orthogonal layout requires at least $\frac{4}{3}(n-1)+2$ bends. For any 3-planar graph G we show that G can be drawn with at most $\lfloor \frac{n}{2} \rfloor + 1$ bends on an $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$ grid. A nice characteristic of the drawing is that it has a spanning tree using $n-1$ straight-line edges, and that all non-tree edges have at most 1 bend (if $n > 4$).

In Chapter 12 we present a linear time algorithm to draw triconnected 3-planar graphs on a hexagonal grid of size $\frac{n}{2} \times \frac{n}{2}$. Using this algorithm we show that every 3-planar graph with > 4 vertices can be drawn with straight-line edges such that the minimum angle is $\geq \frac{\pi}{4}$ if G is triconnected, and $\geq \frac{\pi}{3}$, otherwise. As a side result we prove that every triconnected 6-planar graph can be drawn on an $O(n) \times O(n)$ hexagonal grid such that every edge has at most 4 bends.

In Chapter 13 we consider the problem of representing every vertex v of G as a rectangle $R(v)$ such that if $(v,w) \in G$, then $R(v)$ and $R(w)$ share a boundary. This is called the *rectangular dual* problem. Bhasker & Sahni [7] described a very complicated linear time algorithm for this problem, based on a so-called *regular edge labeling*. He showed that using this labeling, the coordinates can be computed in a simple and elegant way, using topological sort [47]. We present a simple linear time algorithm to compute a regular edge labeling, based on the canonical ordering for 4-connected triangular planar graphs. This completes a new and rather easy approach for the rectangular dual problem. Applying this ordering to the visibility representations leads to a grid size of at most $(n-1) \times (n-1)$ for representing 4-connected planar graphs as a visibility representation. This improves the best known grid bounds by a factor 2. Using this result it is shown that every planar graph can be drawn as a visibility representation, using a grid of size at most $(\lfloor \frac{3}{2}n \rfloor - 3) \times (n-1)$, improving the best known bound of $(2n-5) \times (n-1)$ considerably.

Part C ends with several related results, conclusions and closing observations in

Chapter 14. Chapter 14 also contains some open problems and it introduces some interesting related fields for further research.

Chapter 10

The Drawing Framework and Convex Drawings

10.1 The Drawing Framework

In this section we refine the canonical ordering, described in Section 2.5, to a *leftmost canonical* ($lmc-$) *ordering* for triconnected planar graphs. Using this refinement, we obtain a linear time framework to represent triconnected planar graphs in various ways on a grid. Among these representations, we discuss in detail the convex straight-line grid drawings, visibility representations, orthogonal and hexagonal drawings, and the computation of rectangular duals.

10.1.1 The lmc-Ordering

Recall from Section 2.5 that a canonical ordering of a triconnected planar graph is a sequence of sets $V_1, V_2, \ldots, V_K$ of vertices such that $V_1 = \{v_1, v_2\}$, $V_K = \{v_n\}$, and v_2 and v_n are neighbors of v_1 and share a face (the outerface). In step k, $1 < k < K$ the vertex set V_k is added. If V_k is a singleton, z, then z belongs to C_k and has at least one neighbor in $G - G_k$; if V_k is a chain, $\{z_1, \ldots, z_\ell\}$, then each vertex z_i $(1 \leq i \leq \ell)$ has a neighbor in $G - G_k$, and only z_1 and z_ℓ have a neighbor on C_{k-1}. In each step, the graph G_k, consisting of the vertices of $V_1, \ldots, V_k$, is biconnected.

The general idea for drawing the graph is to start with edge (v_1, v_2), and add in step k the vertices of V_k. In step K vertex v_n is added. Assume w.l.o.g. that in step 1 v_1 is drawn most left and v_2 most right. Let $C_{k-1} : c_1 = v_1, c_2, \ldots, c_q = v_2$ be the vertices from left to right on the outerface of G_{k-1}. When adding the vertices of V_k let c_l and c_r be two neighbors of V_k on G_{k-1}, with l and r as small and as big as possible, respectively. We call c_l the *leftvertex* and c_r the *rightvertex*. Edges to lower (higher) numbered neighbors of vertex v are called *incoming* (*outgoing*) *edges* of v, and $in(v)$ and $out(v)$ denote the corresponding number.

We place the vertices in such a way on the grid such that when adding V_k, the corresponding incoming edges have downwards direction. Moreover, we want to

maintain the invariant that the vertices $c_1, \ldots, c_q$ of C_k remain "visible from the top" during each step. This implies that after adding V_k, vertices $c_r, \ldots, c_q$ must be "shifted to the right", as well as several interior vertices of G_{k-1}. However, updating all x-coordinates of the vertices in G_k in each step implies a quadratic running time. To avoid this, we use *lazy evaluation*:

We compute the exact coordinates of a vertex only when they are necessary to compute the coordinates of other vertices. This means that only the exact coordinates of the vertices on the outerface are essential during the insertions. As a first step towards this process, we refine the canonical ordering to the *leftmost* canonical ordering, which we will call the *lmc*-ordering from now on.

Definition 10.1.1 *A canonical ordering is a leftmost canonical (lmc-)ordering if we can add in any step k a vertex set V_k with leftvertex c_l or a vertex set $V_{k'}$ with leftvertex $c_{l'}$, and $l < l'$ holds, then $k < k'$.*

In other words, we take this vertex set V_k, for which the corresponding leftvertex c_l is minimal with respect to l. By planarity it follows that also the corresponding rightvertex, say c_r, is minimal with respect to r.

To compute the *lmc*-ordering, we maintain a list *Outerface-Stack* for the vertices on the outerface from left to right, implemented as a stack, and initialized as $\{v_2\}$. Also the vertex sets V_k of the canonical ordering, with pointers to its left- and rightvertex are stored. Notice that V_k can be added in step k', if all incoming edges of V_k are part of $C_{k'-1}$. We now delete vertices from the top from *Outerface-Stack*, until we find a vertex c_r on top, which is the rightvertex of a vertex set V_k, not added yet. Let $V_k = \{z_1, \ldots, z_\ell\}$ from left to right, then we add $z_\ell, \ldots, z_1$ in this order to *Outerface-Stack*. We repeat this step with the updated *Outerface-Stack*, until all sets V_k are added. When a vertex c_r is deleted from *Outerface-Stack*, then c_r is not the rightvertex of some set V_k (which is not added yet), because otherwise all other incoming edges of V_k are left from c_l on the current outerface, which would imply that V_k could be added. Hence every vertex set V_k will be added once to the ordering. This implies that every vertex will once be added and once be deleted from *Outerface-Stack*. Since the vertices are added from left to right to *Outerface-Stack* the vertex sets V_k are added in a leftmost order.

Theorem 10.1.1 *Given a canonical ordering, an lmc-ordering can be computed in linear time.*

Proof: The correctness is shown above. Regarding the time complexity, every vertex v is once added to, and once deleted from *Outerface-Stack*. Testing whether vertex c_r on top of *Outerface-Stack* is the rightmost vertex of some set V_k (not added yet) requires constant time, which completes the proof. □

In Figure 10.1 an example of the *lmc*-ordering is given which will serve as an example for almost all drawing algorithms, presented in this paper.

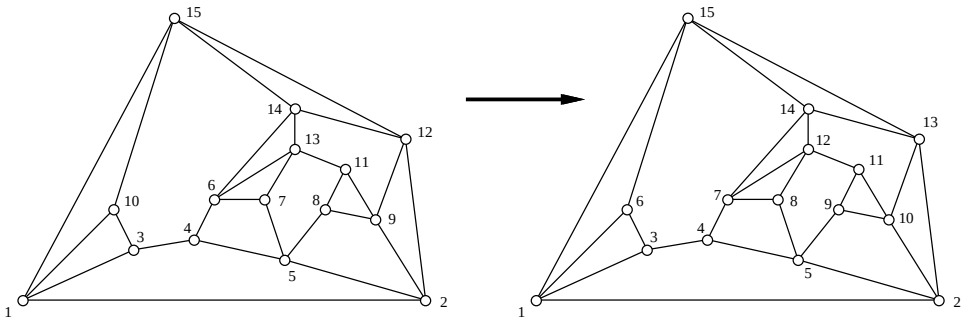


Figure 10.1: From a canonical ordering of the graph in Figure 2.13 to an *lmc*-ordering.

In the drawing algorithms we distinguish the insertcoordinates of v_k (when we insert v_k by the *lmc*-ordering) and the endcoordinates of v_k (in the complete drawing). We introduce a boolean variable *correct*(v), denoting whether $x(v)$, with $v \in C_k$, has been recalculated. We also introduce a counter for each vertex v, called *shift*(v). *shift*(c_r) denotes the value, which must be added to all $x(c_k)$, with $j \leq k \leq r$, where $c_1, \ldots, c_r$ is the current outerface C_k. When a vertex v is added by the *lmc*-ordering, we set *correct*$(v) =$ *false* and *shift*$(v) = 0$. Inspect step k. Let c_r be the rightvertex of V_k. We walk along the outerface from c_r towards c_1 until we find the first *true* marked *correct*(c_α). Then we walk back from c_α to c_r. When visiting c_β $(\alpha < \beta < r)$ we add $\sum_{\alpha < i \leq \beta}$ *shift*(c_l) to $x(c_\beta)$ and set *correct*(c_β) to *true*, because $x(c_\beta)$ is recalculated. We add $\sum_{\alpha < i < r}$ *shift*(c_l) to *shift*(c_r). This approach is correct since the following two lemmas hold in step k:

Lemma 10.1.2 *All vertices c_β, $\alpha < \beta \leq r$, have correct$(c_\beta) =$ false.*

Proof: Suppose not. Inspect the first time that a vertex c_γ on the outerface is encountered for which *correct*$(c_\gamma) =$ *false* and *correct*$(c_{\gamma+1}) =$ *true*. *correct*$(c_{\gamma+1}) =$ *true* means that in a step $k' < k$, $x(c_{\gamma+1})$ and *correct*$(c_{\gamma+1})$ are recomputed, due to the insertion of a vertex set $V_{k'}$ with leftvertex $c_{l'}$, and $l' \geq \gamma + 1$. But *correct*$(c_\gamma) =$ *false* means that in step k we add V_k with rightvertex c_γ. This contradicts the definition of the *lmc*-ordering. □

Lemma 10.1.3 *$r > \alpha$ holds for rightvertex c_r.*

Proof: Suppose not. *correct*$(c_\alpha) =$ *true* means that in a step $k' < k$, we updated $x(c_\alpha)$, due to the insertion of $V_{k'}$ with leftvertex $c_{l'}, l' \geq \alpha$. Adding V_k with rightvertex c_r in step k implies $l' > r$. Since $k' < k$, this contradicts the definition of the *lmc*-ordering. □

Lemma 10.1.4 *The total time for visiting the* false *marked vertices and updating shift*(v), $x(v)$ *and correct*(v) *for all vertices* v *is* $O(n)$.

Proof: When we insert V_k in step k with leftvertex c_l and rightvertex c_r, extra time is required for walking towards c_1 to find the first *true* marked *correct*(c_α). All *correct*-values of the vertices $c_\alpha, \ldots, c_{r-1}$ are marked *true* after visiting them. If in a step $k' > k$, *correct*(c_β) becomes *false* again (with $\alpha \leq \beta \leq r-1$), then a vertex set $V_{k'}$ with rightvertex $c_{k'} = c_\beta$ is added, with $k' \leq l$. This contradicts the *lmc*-ordering, hence every *correct*(c_β) becomes once *false* and becomes *true* after visiting c_β again. Updating requires constant time, hence the total time for visiting the *false* marked vertices and updating *shift*$(v), x(v)$ and *correct*(v) for all vertices is $O(n)$. □

These three lemmas show that in any step k we can compute the up-to-date x-coordinates of the vertices $c_1, \ldots, c_r$ of C_{k-1} when adding V_k, where c_r is the rightvertex of V_k. Let $P(v) = (x_{insert}(v), y_{insert}(v))$ be the coordinates of v at the time of adding v.

However, how can we compute the final x-coordinates of the vertices. Indeed, hereto we have to traverse the vertices of V_k in decreasing order, i.e., from V_K to V_1, and set initially *shift*$(v) = 0$ for all $v \in V$. When considering the vertices of $V_k = \{z_1, \ldots, z_\ell\}$, we set *shift*$(c_l)$ = *shift*(z_1), with $l < i < r$, and c_l and c_r the left- and rightvertex, resp., of V_k, and $c_1, \ldots, c_q$ the outerface C_{k-1} of G_{k-1}. *shift*(c_l) is not updated (because c_l is not shifted when adding V_k initially). Since c_r is also part of some outerface $C_{k'-1}$, $k' > k$, *shift*(c_r) could already be greater than zero at the moment of visiting V_k. The question arises whether this value was also added to the vertices of V_k or not. If this was the case, then this shift-value should not be added to *shift*(c_r) again. How can we solve this problem?

The solution is as follows: to compute the right shift of c_r we distinguish the shifts added to $c_{l+1}, \ldots, c_{r-1}$ and to c_r, by introducing a new variable, *rshift*(v). When considering V_k for computing the final x-coordinates, and c_r must be shifted a value x' more to the right than c_{r-1}, then we add *rshift*$(z_1) + x'$ to *rshift*(c_r). The final coordinates of the vertices $z_1, \ldots, z_\ell$ of V_k is now given by $x_{insert}(z_i)$+ *shift*(z_i)+ *rshift*$(z_i), 1 \leq i \leq \ell$.

The technique for computing the final coordinates corresponds to the idea of computing the insert-coordinates: when the vertices of V_k are shifted to the right in a later step, then also the vertices $c_{l+1}, \ldots, c_r$ must be shifted to the right as well.

All this work can be done in linear time totally. We call the method for computing the insert- and endcoordinates, using the *shift*(v) values the *shift-method*. This method will serve as a general framework for planar graph drawings on a grid. The idea of shifting vertices is widely used, e.g., in the grid drawing algorithm of Chrobak & Payne [15]. A generalization of the latter technique is described in Section 10.5.2.

In Section 10.2 we use the *lmc*-ordering to draw a triconnected planar graph with convex faces on an $(2n-4) \times (n-2)$ grid. In Section 10.2.1 we show how

the grid size can be reduced to $(n-2) \times (n-2)$. In Section 10.5.2 optimizations are given for computing an *lmc*-ordering and computing the coordinates. Several related applications of the *lmc*-ordering are given in this chapter as well. In Chapter 11 and 12 we use the *lmc*-ordering for drawing planar graphs orthogonally and on an hexagonal grid.

10.2 Convex Drawings

The *lmc*-ordering is a generalization of the canonical ordering of de Fraysseix et al. [34]. We can apply the *lmc*-ordering and the shift-method to get a linear implementation of the straight-line grid drawing algorithm of triangulated planar graphs [34]. (In [15] another linear implementation of [34] is described, assuming that the input graph is triangulated.) Moreover, we will show that this algorithm can be modified such that we can draw every triconnected planar graph with convex faces on a grid.

The algorithm of [34] is as follows: it maintains a straight-line embedding during every step k of the *lmc*-ordering such that

1. v_1 is at $(0,0)$, v_2 is at $(2k-4,0)$.
2. If $v_1 = c_1, c_2, \ldots, c_r = v_2$ is the outerface of G_k in step k, then $x(c_1) < x(c_2) < \cdots < x(c_r)$.
3. The edges (v_l, v_{l+1}) have slopes $+1$ or -1.

Assume first that G is triangulated, in which case we can add a vertex v_k in every step k of the *lmc*-ordering [34]. Let $L(v)$ be a set of vertices. The idea of the algorithm is the following: when we add vertex v_k with leftvertex c_l and rightvertex c_r then all vertices $c_{l+1}, \ldots, c_{r-1}$ are shifted one to the right, and the vertices $c_r, \ldots, c_r$ are shifted two to the right (and of course, several internal vertices of G_{k-1} have to be shifted to the right as well). The crossing point of the line with slope $+1$ from c_l and the line with slope -1 from c_r denotes the place for vertex v_k. All vertices $c_l, \ldots, c_r$ are visible from this point, see Figure 10.2 for the corresponding picture. In particular, the algorithm is as follows:

{ In every step k, let $c_1, \ldots, c_r$ be the outerface,
and c_l and c_r are the left- and rightvertex of v_k, resp.}
let $\mu(p_1, p_2)$ be the crossing point of line of slope $+1$ from p_1
and line of slope -1 from p_2.

$P(v_1) := (0,0); L(v_1) := \{v_1\};$
$P(v_2) := (2,0); L(v_2) := \{v_2\};$
$P(v_3) := (1,1); L(v_3) := \{v_3\};$
for $k := 4$ **to** n **do**
  for $v \in \bigcup_{l=j}^{r} L(c_l)$ **do** $x(v) := x(v) + 2$ **rof**;

for $v \in \bigcup_{l=i+1}^{j-1} L(c_l)$ **do** $x(v) := x(v) + 1$ **rof**;
$P(v_k) := \mu((x(c_l), y(c_l)), (x(c_r), y(c_r)))$;
$L(v_k) := \{v_k\} \cup \bigcup_{l=i+1}^{j-1} L(c_l)$
rof

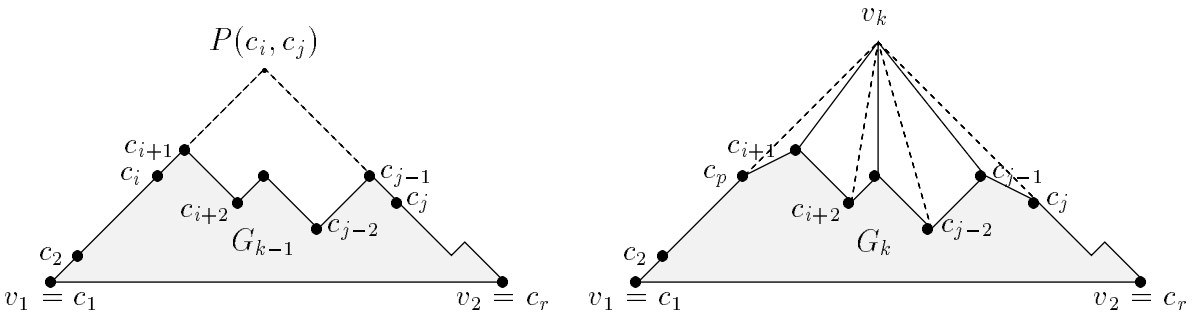


Figure 10.2: Idea of the straight-line drawing algorithm.

The correctness of this algorithm is proved in [34]. Shiftvalues of 1 and 2 occur in the algorithm and we update the corresponding variables *shift*(v) and *rshift*(v) in a similar way. The complete algorithm can now be implemented as follows:

LINEARSTRAIGHT-LINEDRAW(G);
$P(v_1) := P(v_2) := (0, 0)$;
for $k := 3$ **to** n **do**
update $x(c_l)$ and *shift*(c_r);
shift$(c_r) :=$ *shift*$(c_j) + 2$;
$P(v_k) := \mu((x(c_l), y(c_l)), (x(c_r) +$ *shift*$(c_j), y(c_j)))$
rof;
shift$(v) :=$ *rshift*$(v) := 0$ for all $v \in V$;
for $k := n$ **downto** 2 **do**
for every internal vertex v_i of v_k **do** *shift*$(v_i) :=$ *shift*$(v_k) +$ *rshift*$(v_k) + 1$ **rof**;
rshift$(c_r) :=$ *rshift*$(c_j) +$ *rshift*$(v_i) + 2$;
$x(v_k) := x_{insert}(v_k) +$ *shift*$(v_k) +$ *rshift*(v_k)
rof;
END LINEARSTRAIGHT-LINEDRAW

Moreover, using the *lmc*-ordering it is already sufficient that the planar graph is triconnected, because when adding $V_k = \{z_1, \ldots, z_\ell\}$, we can draw $z_1, \ldots, z_\ell$ on a horizontal line with distance two in between. This yields that edge (c_l, z_1) has a slope $+1$, edges $(z_1, z_2), \ldots, (z_{\ell-1}, z_\ell)$ have slope 0 and length 2, and edge (z_ℓ, c_r) has a slope -1. It is easy to see that this still gives a correct straight-line drawing on an $(2n - 4) \times (n - 2)$ grid.

In the remaining part of this section we modify this new algorithm a little such that all interior faces are convex.

Let V_k be $\{z_1\}$, and let $c_{i_1}, \ldots, c_{i_s}$ be the vertices on C_{k-1}, adjacent to z_1. ($i_1 = l$ and $i_s = r$.) Let F_j $(1 \leq j < s)$ be the face formed by the edges $(z_1, c_{i_j}), (z_1, c_{i_{j+1}})$ and the path B_j between c_{i_j} and $c_{i_{j+1}}$.

Lemma 10.2.1 *Each path B_j has the following pattern:*

- *From c_{i_j} to some vertex c_{α_j}, a sequence D_j, $|D_j| \geq 1$, of vertices with strictly decreasing y-coordinate.*
- *Two vertices $c_{\alpha_j}, c_{\beta_j}$ with same y-coordinate.*
- *From c_{β_j} to $c_{i_{j+1}}$, a sequence U_j of vertices with strictly increasing y-coordinate.*

Proof: By definition of the *lmc*-ordering, every vertex $v \in V_k$ has a neighbor $w \in V_{k'}$, with $k' > k$. By definition of the algorithm LINEARSTRAIGHT-LINEDRAW it follows that $y(w) > y(v)$. In step k the vertices $c_{i_j+1}, \ldots, c_{i_{j+1}-1}$ have already higher placed neighbors. Let c_{α_j} be the lowest placed vertex, with $i_j \leq \alpha_j < i_{j+1}$ and α_j minimal. If there is a vertex c_{β_j} with $y(c_{\beta_j}) = y(c_{\alpha_j})$, then $\beta_j = \alpha_j + 1$, because otherwise there would be a vertex c_γ, $\alpha_j < \gamma < \beta_j$, which does not have a higher placed neighbor. From c_{i_j} to c_{α_j} the vertices have strictly decreasing y-coordinate and from c_{α_j+1} to $c_{i_{j+1}}$ the vertices have strictly increasing y-coordinate. □

Notice that all edges on C_{k-1} have slope $+1$, 0 and -1 before adding V_k. When adding z_1, we shift $c_{\alpha_1}, \ldots, c_{\beta_{s-1}}$ to right by one, and shift $c_{\beta_{s-1}+1}, \ldots, c_{i_s}$ to right by two. As explained above, we draw z_1 at point $\mu(c_{i_1}, c_{i_s})$.

When $V_k = \{z_1, \ldots, z_\ell\}$, we add only one face F_1. Let B_1 be the path of C_{k-1} between c_l and c_r. B_1 also has the pattern of Lemma 10.2.1. We shift c_{α_1} and c_{β_1} to right by one, and $c_{\beta_1+1}, \ldots, c_r$ to right by $2p$. $z_1, \ldots, z_\ell$ are placed as explained above.

This yields the following slopes after adding V_k:

- The slope of edge $(c_{\alpha_1-1}, c_{\alpha_1})$ is in the range $[-1, 0)$.
- The slope of edge $(c_{\beta_{s-1}}, c_{\beta_{s-1}+1})$ is in the range $(0, +1)$.
- All other slopes on C_{k-1} are not changed.
- The slopes of the incident edges of V_k are in the range $(-\infty, -1] \cup [+1, \infty)$.

This implies that the faces $F_1, \ldots, F_{s-1}$ are convex when inserting V_k at step k. To preserve convexity during the other steps $k' > k$, we add edges from c_{i_j} to $c_{\beta_j}, \ldots, c_{i_j-2}$ $(1 \leq j \leq s)$. This does not destroy planarity and implies that if c_{i_j} is shifted to right in some step $k' > k$, then also $c_{\beta_j+1}, \ldots, c_{i_j-1}$ is shifted to right with the same value. The modified graph is still called G. Now we can prove the following lemma.

Lemma 10.2.2 *The faces remain convex during the algorithm.*

Proof: Assume $V_k = \{z_1\}$, and let c_{i_j} $(1 \leq j \leq s)$, $c_{\alpha_j}, c_{\beta_j}$ and F_{i_j} $(1 \leq j < s)$ be as defined above. (The proof is analogous when $|V_k| > 1$.) Consider a step $k' > k$. If c_{i_s} is shifted to the right, then by the added dummy edges, also $c_{\beta_{s-1}+1}, c_{i_s-1}$ are shifted to the right with the same value, thereby preserving planarity in the only relevant face, F_{s-1}. If z_1 is shifted to right, then the vertices $c_{i_2}, \ldots, c_{i_s}$ are shifted to right as well, and if c_{i_j} is shifted, then $c_{\beta_{j-1}+1}, \ldots, c_{i_j-1}$ and $c_{i_j+1}, c_{\beta j+1}$ are shifted to right with the same value. This yields planarity in the faces $F_1, \ldots, F_{s-1}$. It also has the consequence that if c_i is shifted to right $(i_1 \leq i < i_s)$, then $c_{i'}$, $i < i' \leq i_s$ is shifted to right with at least the same value.

We use this observation for the case that c_{i_1} is shifted to right in some later step k. Then z_1 is also shifted to right by at least the same value. Hence the vertices $c_{\beta_1+1}, \ldots, c_{i_2}$ are shifted to right by at least the value of the shift of $c_{i_1}, \ldots, c_{\beta_1}$. This preserves the planarity in F_1 and completes the proof. □

Finally we remove the added dummy edges from c_{i_j} to $c_{\beta_j}, c_{\beta_j+1}, \ldots, c_{i_j-2}$ $(1 \leq j \leq s)$.

Theorem 10.2.3 *There is a linear time and space algorithm to draw a triconnected planar graph convexly with straight-line edges on an* $(2n-4) \times (n-2)$ *grid.*

Our algorithm not only outperforms the algorithms of [110, 13], it is also much easier to implement than the algorithm of [13]. However, a drawback of the algorithm is that the drawings are not strictly convex, as those of [110, 13]. On the positive side, this algorithm gives a new proof that every triconnected planar graph admits a planar drawing, in which every interior face is convex. The outerface is a triangle. With respect to the tightness of the grid size we note that every strictly convex drawing of a cycle with n vertices requires an $\Theta(n^3)$ grid [78]. In Figure 10.3, the straight-line convex drawing of the graph in Figure 10.1 is given.

10.2.1 Convex Drawings on an $(n-2) \times (n-2)$ Grid

In this section we describe a method for decreasing the grid size by a factor 2. This optimization is joint work with Chrobak, and based on an important observation made by Schnyder. The idea is to maintain the outerface C_{k-1} during the drawing algorithm, such that every edge has a slope, which is $-1, 0$, or in the range $[+1, \infty)$, instead of $-1, 0, +1$, as described in the previous section.

Let $V_k = \{z_1, \ldots, z_\ell\}$, which we add from leftvertex c_l to rightvertex c_r in step k. If c_l has no edges to vertices added in a step $k' > k$, then we can place z_1 right above c_l, thus $x(z_1) := x(c_l)$, otherwise we set $x(z_1) := x(c_l) + 1$. We shift c_r ℓ to the right, and the crossing point of the line with slope -1, starting from $P(c_r)$, with the vertical line of $x(z_1) + \ell - 1$ gives the place for z_ℓ. We place z_i $(1 \leq i < \ell)$ at $(x(z_\ell) - \ell + i - 1, y(z_\ell))$, and add ℓ to *shift*(c_r). Testing whether c_l has an edge to

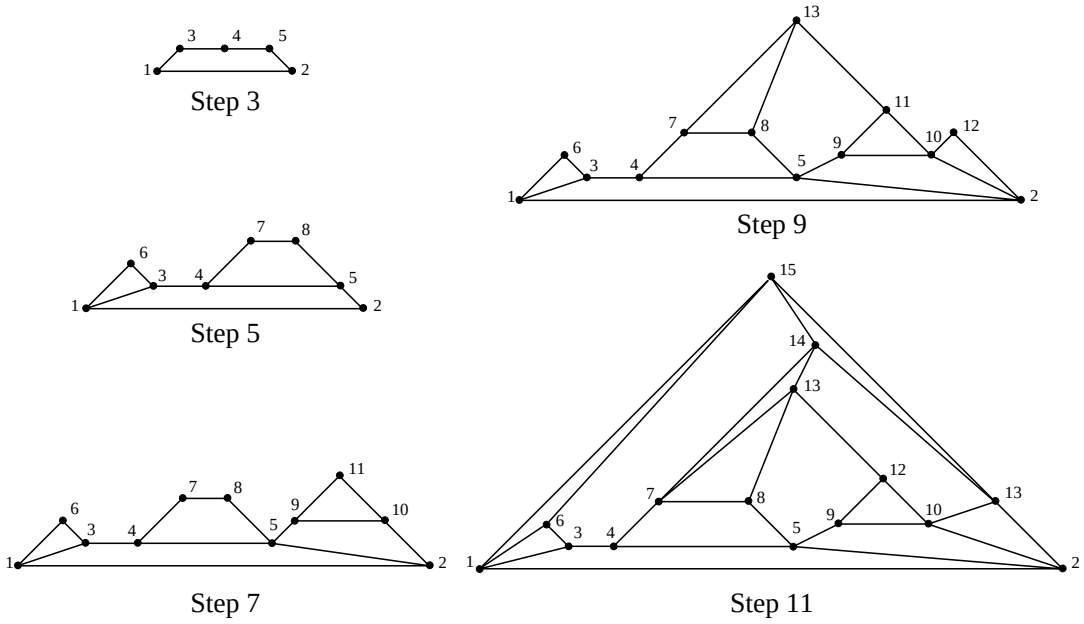


Figure 10.3: Convex drawing of the graph of Figure 10.1.

a vertex added in a step $k' > k$ is easy, because the embedding of the planar graph is given. If c_{l-1} is adjacent to z_1 in $adj(c_l)$, then there is no such vertex, otherwise there is one. In Figure 10.4 it is demonstrated that the neighbors c_α of v_k on C_{k-1}, with $i \leq \alpha \leq j$, are visible from $P(v_k)$. This follows from the fact that if for a vertex c_α, $x(c_{\alpha+1}) = x(c_\alpha)$, then c_α does not have an edge to v_k. The slope of (c_l, z_1) is 0 or in the range $[+1, \infty)$, the slopes of the edges (z_i, z_{i+1}) is 0 $(1 \leq i < p)$, and the slope of the edge (z_ℓ, c_r) is -1.

Lemma 10.2.4 *Applying the modification, described above, in the convex grid drawing algorithm, gives a correct straight-line drawing of a triconnected planar graph G on an $(n-1) \times (n-1)$ grid.*

Proof: We only have to prove the size of the grid. We start with horizontal edge (v_1, v_2) of length 1. Assume that we add in step k the vertex set $V_k = \{z_1, \ldots, z_\ell\}$ $(\ell \geq 1)$ from leftvertex c_l to rightvertex c_r. This increases the width by one, and the height by at most ℓ. This gives a total length of $n-1$, and since the line through v_2 and v_n has slope -1, and $x(v_n) = 0$, it follows that the height is also $n-1$. □

We now modify the algorithm in the same way as in the previous section such that all interior faces are convex. Assume we add a vertex v_k. Let c_p and c_q be two vertices on C_{k-1}, such there is no neighbor c_l of v_k, with $p < l < q$. Let F' be the face, containing c_p, c_q and v_k.

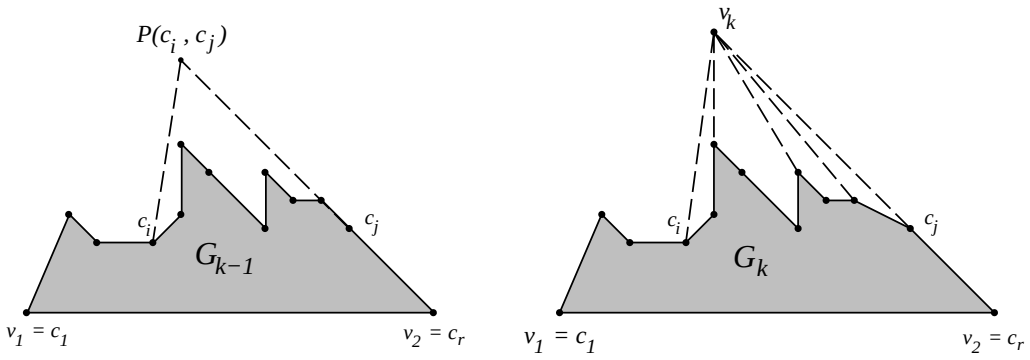


Figure 10.4: Convex drawings on an $(n-2) \times (n-2)$ grid.

Lemma 10.2.5 *There are constants α, β, with $p \leq \alpha \leq \beta \leq q$, and $\beta = \alpha$ or $\beta = \alpha + 1$ such that before adding v_k:*

- *all edges $(c_a, c_{a+1}), p \leq a < \alpha$, have slope -1;*
- *all edges $(c_b, c_{b+1}), \beta < b < q$, have slope ∞;*
- *the edge $(c_\beta, c_{\beta+1})$ has a slope in the range $(0, \infty)$;*
- *if $\beta = \alpha + 1$, then (c_α, c_β) has a slope 0.*

Proof: We first observe that every vertex v has a neighbor, say w, which is added in a later step and has $y(w) \geq y(v)$. In step k the vertices $c_{p+1}, \ldots, c_{q-1}$ have already higher placed neighbors. Let c_β be the lowest placed vertex, with $p \leq \beta \leq q$. If $\beta > p$ and $(c_{\beta-1}, c_\beta)$ is horizontal, then from $c_{\beta-1}$ to c_p the vertices are strictly increasing in Y-direction, thus have slope -1. If $\beta > p$, then c_b is the leftvertex of $c_{b+1}, \beta \leq b < q$. By the drawing algorithm, we place $x(c_{b+1}) = x(c_b)$, thus the edges (c_b, c_{b+1}) are vertical. If $\beta = p$, then c_β has neighbor v_k. This yields $x(c_{\beta+1}) = x(c_\beta) + 1$. In this case $(c_\beta, c_{\beta+1})$ has a slope in the range $[+1, \infty)$. □

To achieve a convex face F', we do the following. Let v_k be an added vertex with leftvertex c_l and rightvertex c_r. Only $x(c_r)$ increases when adding v_k. Also it follows that if *shift*(c_l) or *shift*(v_k) increases, then the *shift* increases for the vertices $c_l, \ldots, c_r$. The only point is when *shift*(c_r) increases, and *shift*(v_k) does not increase. Let c_p be a neighbor of v_k, with $p < j$ as large as possible, and let β be as small as possible, with $y(\beta + 1) > y(\beta)$ and $p \leq \beta < j$. We add edges from c_r to $c_\beta, \ldots, c_{r-1}$ in step k. This does not destroy planarity in the embedding, and it means in the algorithm, that if we shift c_r, then we also shift $c_{\beta+1}, \ldots, c_{r-1}$ to the right. This implies that still $x(c_{\beta+1}) = x(c_{\beta+2}) = \ldots = x(c_r)$ holds, and thus Lemma 10.2.5

holds after adding v_k. This implies that face F' is convex when inserting v_k in step k. The same values follow when we add a chain $z_1, \ldots, z_\ell$ instead of one vertex v_k, because in this case $z_1, \ldots, z_\ell$ are placed on a horizontal line. We again apply Lemma 10.2.2, which says that the internal faces remain convex during the algorithm. After this we delete the added edges from c_r to $c_\beta, c_{\beta+1}, \ldots, c_{r-1}$.

Now we sketch how to modify the algorithm in order to reduce the grid size to $(n-2) \times (n-2)$. First we pick v_n to be the neigbhor of v_2 different from v_1 on the outer face of G. We construct a canonical decomposition and run the previous algorithm for $K-1$ steps. In the last step, having already embedded G_2, we set $P(v_n) = (1, n-2)$, and we *do not* shift any vertices to the right.

In order to show correctness, we only need to show that adding v_n will result in a correct, convex embedding. By Lemma 10.2.5 and the algorithm, before adding v_n we have $x(c_1) = x(c_2) = \ldots = x(c_l) = 0$ and $x(c_r) = n-2$, where $c_r = v_2$. The edge with slope -1 from v_2 contains the point $(1, n-3)$. This implies that all vertices $c_l, \ldots, c_r$ are visible from $(1, n-2)$. The convexity of the outer face follows from the choice of v_n. Consequently, we obtain the following theorem.

Theorem 10.2.6 *There is a linear time and space algorithm to draw a triconnected planar graph convexly into the* $(n-2) \times (n-2)$ *grid.*

At the end of this chapter a complete pseudo-Pascal code for the $(n-2) \times (n-2)$ convex grid drawing algorithm is given, including some other optimizations, described in Section 10.5.2. The grid size matches the best known grid bounds for drawing a planar graph planar on a grid (see Schnyder [98]). Moreover, it gives a new proof that every triconnected planar graph admits a planar drawing, in which every interior face is convex. However, the drawing is not *strictly convex*, i.e., there might be angles of size π. When we want to obtain a strictly convex drawing, then the gridsize becomes larger, since any drawing of a cycle of length n already requires a grid of size $\Omega(n^3)$ by a result of Lin & Skiena [78]. Unfortunately, we see no simple way to change our convex drawing algorithm such that it avoids angles with size π.

Chiba et al. showed that if a graph can be drawn convex, then after eliminating the vertices of degree 2, almost all triconnected components have an edge on the boundary of the outerface. But using our algorithm, we can draw biconnected planar graphs, where every triconnected component has an edge on the outerface, convex as well. This follows by observing that adding a vertex v_{n+1} with edges to all vertices on the outerface gives a triconnected planar graph. Applying the algorithm on the augmented graph, and finally removing vertex v_{n+1} gives the desired result.

The vertices with degree 2 seems to be a much harder problem. How can we place these vertices back after applying ConvexSmallGridDrawing on the reduced graph? In particular, let v have neighbors u and w. Let the greatest common divisor of $|x(w) - x(u)|$ and $|y(w) - y(u)|$ be 1, then there is no interior grid point on the drawing of edge (u, w). Hence the problem of placing v seems to be hard when placing the vertices on grid coordinates.

10.3 The mixed model

In this section we use the drawing framework, introduced in Section 2.5, to draw any triconnected d-planar graph G on an $(2n-6) \times (3n-9)$ grid such that there are at most $5n-15$ bends and the minimum angle is at least $\frac{2}{d}$ radians. All vertices and bend coordinates will be placed on grid points only. Every edge will have at most three bends and length $O(n)$.

Every edge, say (u, v) will have the following format. From u it goes to an *outpoint* of u, say b_o, from b_o it goes in vertical direction to a point, say b', from b' it goes in horizontal direction to an *inpoint*, say b_i of v, and from b_i it goes to v. Important question is here what the coordinates of the in- and outpoints are, and how to compute them.

Let $out_l(v) = \lfloor \frac{out(v)-1}{2} \rfloor$ and $out_r(v) = \lceil \frac{out(v)-1}{2} \rceil$. Similar for $in(v)$, i.e., $in_l(v) = \lfloor \frac{in(v)-3}{2} \rfloor$ and $in_r(v) = \lceil \frac{in(v)-3}{2} \rceil$. The idea is to place the outpoints of v on the following places: the diagonal lines from $(x(v) - out_l(v), 1)$ to $(x(v)-1, y(v)+ out_l(v))$, and from $(x(v)+ out_r(v), 1)$ to $(x(v)+1, y(v)+ out_r(v))$, and using $(x(v), y(v)+ out_l(v))$. For the inpoints it is defines similarly: the diagonal lines from $(x(v)- in_l(v), -1)$ to $(x(v) - 1, y(v)- out_l(v))$, and from $(x(v)+ out_r(v), -1)$ to $(x(v) + 1, y(v)- out_r(v))$, and the three points $(x(v)- out_l(v), y(v))$, $(x(v), y(v)- out_l(v))$ and $(x(v)+ out_r(v), y(v))$. See Figure 10.5 for some examples, and the corresponding bounding boxes.

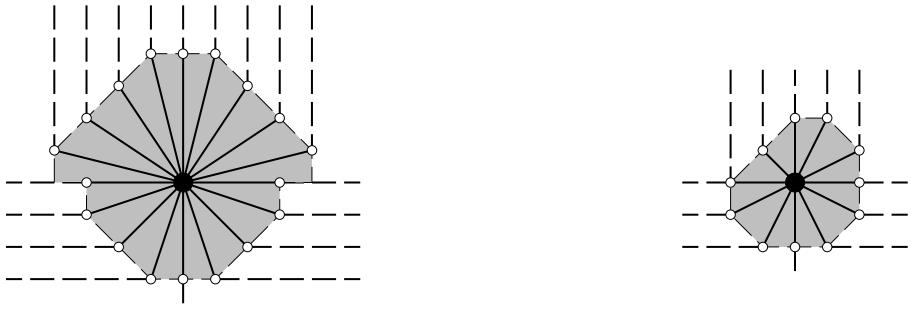


Figure 10.5: Examples of bounding boxes.

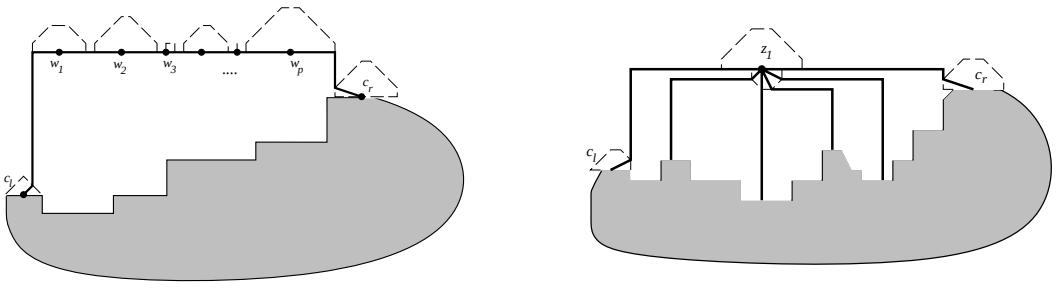
The width of the bounding box is $\max\{out(v) - 1,\ in(v) - 3\}$, the height is $in_l(v)+ out_l(v)$. The idea is to insert the vertices of the canonical ordering such that the bounding boxes do not intersect or touch. Also outgoing edges may not cross or overlap, i.e., this means that for every pair of consecutive vertices c_l, c_{l+1} on the outerface, $x(c_{l+1}) > x(c_l) + out_r(c_l) + out_l(c_{l+1})$ must hold. We can now explain the different adding steps as follows:

Adding a vertex v

For the y-direction, we simply set $y(v) = \max\{y(c_l)+out_l(c_l)+1, y(c_r)+out_l(c_r)+1\}$. In x-direction, the problem is a little more difficult: let $u_1, \ldots, u_{in(v)}(v)$ be neighbors of v, corresponding with the left-to-right order of incoming edges of v. Let $u = u_{in_l(v)}$. then we want to have $x(v) = x(u)$, but also (for the outgoing edges), we want to have $x(v) > x(c_l) + out_r(c_l) + out_l(v)$. Hence we set $x(v) = \max\{x(u), x(c_l) + out_r(c_l) + out_l(v)$, and shift $u_{in_l(v)}, \ldots, u_{in(v)-1}$ to the right (if $x(c_l)+out_r(c_l)+out_l(v) > x(u)$). For the rightvertex c_r we set $x(c_r) = \max\{x(c_r), x(v) + out_r(v) + out_l(c_r) + 1\}$.

Adding $z_1, \ldots, z_\ell$, $\ell > 1$.

Now every vertex $z_1, \ldots, z_\ell$ has precisely two incoming edges, hence they are placed on a horizontal line at height $\max\{y(c_l) + out_l(c_l), y(c_r) + out_l(c_r)\} + 1$. In the x-direction we don't have to deal with the incoming edges, hence we set $x(z_1) = x(c_l) + out_r(c_l) + out_l(z_1) + 1$, we set $x(z_i) = x(z_{i-1}) + out_r(z_{i-1}) + out_l(z_i) + 1$ $(1 < i \leq p)$, and we set $x(c_r) = \max\{x(c_r), x(z_\ell) + out_r(z_\ell) + out_l(c_r) + 1\}$. Figure 10.6 makes this more precise.



(a) Adding one vertex.

(b) Adding more vertices.

Figure 10.6: Adding vertices in the mixed model.

Analysis of the algorithm

Using the shift-technique, explained in section 2, it is not difficult to obtain a linear time and space algorithm, satisfying the constraints with respect to width and height, as given in the two relevant steps. Therefore we now consider in detail the number of bends, the size of the minimum angle and the total grid size.

Lemma 10.3.1 *The size of the minimum angle is $\frac{2}{\Delta}$, where Δ is the maximum degree of G.*

Proof: Let v have maximum degree. The minimum angle, say α, is reached at an outpoint which is neighbored to a horizontal line. If $deg(v) < 6$, then it is easily proved, hence assume $deg(v) \geq 6$. The size of the angle is $\arctan(\frac{1}{out_r(v)}) \geq \arctan(\lfloor\frac{2}{deg(v)-2}\rfloor)$. Using the potence series of the arctan we know that for $|x| < 1$, $\arctan(x) = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \ldots \geq x - \frac{1}{3}x^3$. Since $\lfloor\frac{2}{deg(v)-2}\rfloor \geq \frac{2}{deg(v)-1}$, we obtain that $\alpha \geq \frac{2}{deg(v)-1} - \frac{1}{3}(\frac{2}{deg(v)-1})^3 \geq \frac{2}{deg(v)}$, which completes the proof. □

Lemma 10.3.2 *The gridsize is at most* $(2n-6) \times (3n-6)$.

Proof: For the width notice that $x(c_{i+1}) = x(c_i) + out_r(c_i) + out_l(c_{i+1})$ holds in every step on the outerface if $y(c_{i+1}) \neq y(c_i)$ and $x(c_{i+1}) = x(c_i) + out_r(c_i) + out_l(c_{i+1})$ otherwise. If $y(c_{i+1}) = y(c_i)$ then (c_i, c_{i+1}) is not an outgoing edge of any vertex. Let us call (c_i, c_{i+1}) in this case *unmarked*. Counting leads to a horizontal distance of at most $\sum_{1 \leq i < n} out_l(v_i) + out_r(v_i)$+number of unmarked edges $= \sum_{1 \leq i < n}(out(v_i) - 1) = 2n - 6$.

Adding a vertex v_k requires more increase in height per vertex than adding a face, hence assume we add a vertex v_k in every step. Let the incoming edges of vertex v_k come from vertices $u_1, \ldots, u_r$, then $y(v_k) \geq \max_{1 \leq i \leq p}\{y(u_i)\} + \max\{1 + in_r(v), out_r(u_1), out_r(u_r)\}$. The increase for every vertex v_k during the insertions is at most $1 + in_r(v_k) + out_r(v_k)$. Summarizing this for all vertices leads to a total distance in Y-direction of at most $3n - 6$ units. □

Lemma 10.3.3 *There are at most* $5n - 15$ *bends. Every edge has at most 3 bends and length* $O(n)$.

Proof: All outgoing edges of vertex v, except the one going straight upwards, requires one bend in worst-case to go in vertical direction. We assign these bends to the insertion step of v. Adding a face requires less bends per vertex than adding a vertex, so assume we only add vertices v_k. If $in(v_k) = 2$ and $y(c_l) \geq y(c_r)$, then there will come at most 1 bend in (c_l, v_k) and 2 bends in (c_r, v_k). (A similar holds when $y(c_l) \geq y(c_r)$.) In each edge, one bend was already assigned to the insertion step of c_l and c_r, hence adding v_k with $in(v_k) = 2$ requires at most one bend for the incoming edges. If $in(v_k) \geq 3$, then at most $2 \cdot in(v_k) - 4$ extra bends are required for the incoming edges. Edge (v_1, v_2) requires no bends. Counting this leads to totally at most $5n - 15$ bends. Every edge goes at most once vertical and once horizontal, hence requiring 3 bends in worst-case and by Lemma 10.3.2, has length $O(n)$. □

Theorem 10.3.4 *There is a linear time and space algorithm to draw a triconnected d-planar graph planar on an* $(2n-6) \times (3n-9)$ *grid with at most* $5n-15$ *bends and minimum angle* $> \frac{1}{d-2}$, *in which every edge has at most 3 bends and length* $O(n)$.

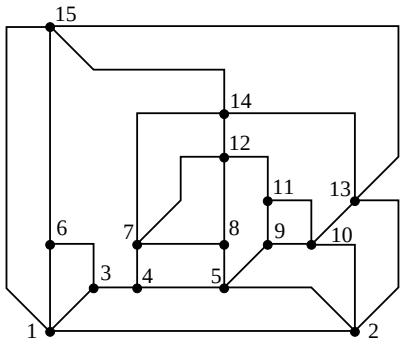


Figure 10.7: Drawing the graph of Figure 10.1 with bends.

In Figure 10.7 the drawing of the graph of Figure 10.1 is given.

Notice that in Chapter 5 we proved that every graph G can be augmented by adding edges to a triconnected planar graph G' such that $\Delta(G') \leq \lceil \frac{3}{2}\Delta(G) \rceil + 3$. This yields the following theorem.

Theorem 10.3.5 *There is a linear time and space algorithm to draw a planar graph planar on an* $(2n-6) \times (3n-9)$ *grid with at most* $5n-15$ *bends and minimum angle* $> \frac{4}{3\Delta(G)+1}$*, in which every edge has at most 3 bends and length* $O(n)$*.*

10.4 Visibility Representations

Since every *lmc*-ordering is also an *st*-ordering, we can use the *lmc*-ordering in various drawing applications, where the *st*-ordering is used. We now focus the attention on the construction of a visibility representation by the *lmc*-ordering. In a visibility representation every vertex is mapped to a horizontal segment, and every edge is mapped to a vertical line, only touching the two vertex segments of its endpoints. In figure 10.8 an example is given. The visibility representation is interesting of its practical consequences. The interesting fact of using the *lmc*-ordering is that now we have the shift-values on the edges instead of on the vertices, and the algorithm becomes quite simple and may lead to more compact representations.

Hereto, when adding V_k, the y-coordinates of the vertices, already placed, are interesting, as well as the x-coordinates of (c_l, v_k) and (c_r, v_k), where c_l and c_r are the left- and rightvertex of V_k, resp. Therefore we associate an y-coordinate, $y(v)$ to every vertex, and an x-coordinate, $x(u,v)$, and a *shift*-variable, *shift*(u,v), to every edge (u,v).

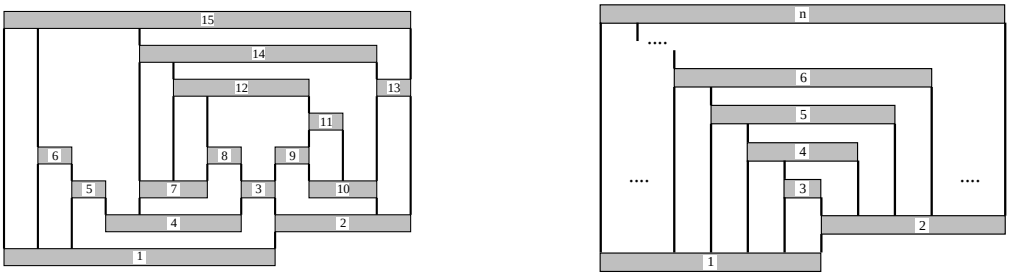
The horizontal segment, representing a vertex v must have length at least $out(v)-1$. If we add $V_k = \{z\}$, then the horizontal segment, representing z goes from $(x(z,c_l), y(z))$ to $(x(z,c_r), y(z))$. If $V_k = \{z_1, \ldots, z_\ell\}$, $\ell > 1$, then we can draw the

horizontal segments of $z_1, \ldots, z_\ell$ alternatingly on some height Y and $Y+1$, as shown in Figure 10.8. The implementation of the complete algorithm follows now directly and is left to the reader.

Since $in(v) \geq 2$ for every vertex v, this means that adding v_k increases the height by 1 and the width by $\max\{0, out(v_k) - 2\}$. If for every vertex v_i, $out(v_i) \geq 2$ holds, then this leads to a visibility representation on a grid of size at most $(n-1) \times (n-1)$. Indeed, we prove in Section 13.4 that a canonical ordering of a 4-connected triangular planar graph is possible, in which every vertex v_k has $out(v_k) \geq 2$. This decreases the width by a factor 2 with respect to the grid size for visibility representations of 4-connected planar graphs. Several related compressing and optimization techniques are possible, leading in general to more compact layouts than the algorithms in [57, 89, 96] in general. In particular, when we add a face F_k in lmc-VISIBILITY(G), we can do it such that the Y-direction increases by at most 2. Moreover, compared with [96], we do not have to compute the dual graph.

Theorem 10.4.1 *There is a linear time algorithm to construct a visibility representation of a planar graph on a grid of size at most* $(2n-5) \times (n-1)$.

In Figure 10.8 a visibility representation of the graph in Figure 10.1 is given, and a visibility representation of a graph is given, requiring an $(2n-5) \times (n-1)$ grid. In Chapter 13 a linear time algorithm is presented, constructing a visibility representation of a planar graph on a grid of size at most $(\lfloor \frac{3}{2}n \rfloor - 2) \times (n-1)$.



(a) Visibility representation of the graph in Figure 10.1.

(b) Graph, requiring a grid of size $(2n-5) \times (n-1)$.

Figure 10.8: Visibility representations.

10.5 Improvements of the lmc-Ordering

In this section we present two important optimizations of our drawing framework. The first improvement follows by proving that an lmc-ordering of G defines an lmc-ordering on the dual graph G^* of G. In particular, if G is a triconnected 3-planar

graph, then G^* is a triangular planar graph, and we can use a much simpler algorithm of de Fraysseix, Pach & Pollack [34] to compute a canonical ordering for G^*. From this canonical ordering of G^* we can derive an *lmc*-ordering for G in an efficient way. The second improvement is that the *shift*-method, as introduced by Chrobak & Payne [15] for a linear implementation of the straight-line grid drawing algorithm of de Fraysseix, Pach & Pollack can be adopted in our framework. More precisely, we can generalize this technique, based on the canonical ordering of triangular planar graphs, such that it works for the canonical ordering of triconnected planar graphs. This means that there is no need to refine the canonical ordering for triconnected planar graphs to an *lmc*-ordering. This yields a new linear time framework, and the constant in the $O(n)$ seems to be smaller in the generalized Chrobak & Payne framework than by applying the *lmc*-ordering. As an example, we show the complete convex drawing algorithm of triconnected planar graphs at the end of this chapter, using the modified Chrobak & Payne approach.

10.5.1 Duality Aspects

Let G be a triconnected planar graph. Let G^* be the dual graph of G, i.e., every face of G is a vertex in G^*, and there is an edge (u, v) in G^*, if the corresponding faces share an edge in G. Assume an *lmc*-ordering is given on the vertices of G. We construct a labeling on the faces of G as follows: if k is the smallest integer such that all vertices of face F belong to G_k in the *lmc*-ordering, then we set $label(F) = K + 3 - k$. Let $(v_1, v_n) \in F', F''$, with F' the outerface. We set $label(F') = 1$ and $label(F'') = 2$.

Theorem 10.5.1 *The labeling of the faces of G corresponds to an lmc-ordering of the dual graph G^*.*

Proof: We first prove by reverse induction on the steps of the *lmc*-ordering that the assigned labeling corresponds to the canonical ordering of the dual graph G^*. Let w_F denote the vertex in G^*, corresponding to face F in G. Let G^* denote the induced subgraph on the vertices w_F with $label(F) \leq i$ in G. We start with deleting v_n from G. (v_1, v_n) belongs to F' and F'', and $label(F') = 1$ and $label(F'') = 2$. Since $deg(v_n) \geq 3$, there are $deg(v_n) - 2$ remaining faces F_i' in G, with $label(F_i') = 3$. By duality aspects, the corresponding vertices $w_{F_i'}$ form a consecutive chain from $w_{F'}$ to $w_{F''}$. Hence G_3^* satisfies the constraints of Theorem 2.5.1.

Let k be fixed, $3 < k < K$, and assume that face F_i or vertex v_i has already been determined for every $i > k$ such that G^*_{K+3-i} satisfies the constraints of Theorem 2.5.1. Consider the 2 cases in step k: deleting a vertex v_k or a face F_k from G_k. Assume first that we delete v_k with p lower-numbered neighbors. Then $p - 1$ faces F_i' are deleted from G_k, which all have $label(F_i') = K + 3 - k$. By construction of G^*, there are edges $(w_{F_i'}, w_{F_{i+1}'})$ $(1 \leq i < p - 1)$, hence it follows that each $w_{F_i'}$ has degree 2 in G^*_{K+3-k}. Each $w_{F_i'}$ has a neighbor with higher label, since F_i' contains

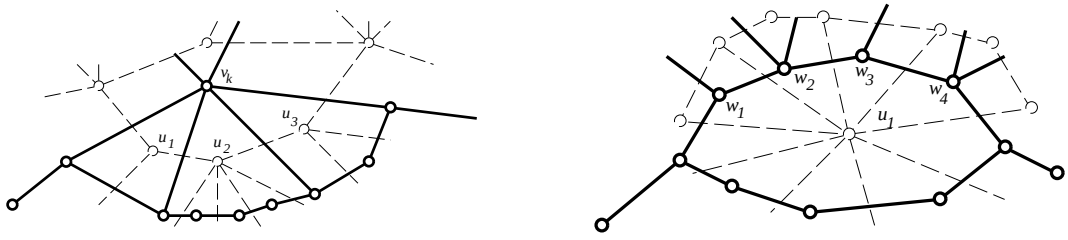
an edge of C_{k-1}. $w_{F'_1}$ and $w_{F'_{p-1}}$ have neighbors with lower label, because F'_1 and F'_{p-1} contain an edge of C_k.

If we delete a face F_k in step k, then w_{F_k} is added to G^*_{K+3-k}. w_{F_k} has at least two neighbors with lower label, and at least one neighbor with higher label. In both cases one easily verifies that the added chain or vertex to G^*_{K+3-k} is on the outerface. G^*_{K+3-k} is biconnected and by induction G^*_{K+3-k} satisfies the constraints of Theorem 2.5.1.

We end with edge (v_1, v_2) in G. Assume (v_1, v_2) belongs to F' (the outerface) and F''', then $label(F''') = K$ by definition. But since v_2 and v_n are neighbors of v_1 and belong to the outerface it follows that $w_{F'''}$ and $w_{F''}$ are both neighbors of $w_{F'}$ in G^*, and belong to a common face, which completes the proof that the assigned labeling is a canonical ordering for G^*.

In the same way, by considering the dual graph G^*, it easily follows that the canonical ordering is leftmost, because if we delete a vertex v_k or face F_k from G_k, then the corresponding face or vertex in G^* is leftmost with respect to the vertices $w_{F'}$ and $w_{F''}$. □

Notice that in [96] a similar result is obtained for the *st*-ordering. Hence it seems that the *lmc*-ordering is a powerful generalization of the *st*-ordering in the triconnected case. (See also the illustration in Figure 10.9.)



(a) Adding a vertex v_k. (b) Adding a chain $w_1, \ldots, w_4$.

Figure 10.9: The dual graph also implies an *lmc*-ordering.

If G is triconnected and 3-planar, then G^* is a triangulated planar graph. In this case we can use the canonical ordering of de Fraysseix, Pach & Pollack [34] to compute an *lmc*-ordering of G^*. The f vertices of G^* are numbered $v_1, \ldots, v_f$ by a simple linear time framework. Every vertex v_i in G^* corresponds to a face F_i in G which must be added. This optimization can be used for computing the *lmc*-ordering in Section 11.2 and Section 12.1, where we consider triconnected 3-planar graphs.

10.5.2 A New *shift*-Method

In this section we explain another technique for computing the coordinates, based on the *shift*-method, described by Chrobak & Payne [15]. They introduced this technique for implementing the straight-line grid drawing algorithm of de Fraysseix, Pach & Pollack [34] in linear time. We show here that this technique can be changed such that it works for our drawing algorithms for triconnected planar graphs as well. Using this technique there is no need to verify the canonical ordering for triconnected planar graphs to the *lmc*-ordering. The crucial observation in [15] is that when we draw v_k, it is not necessary to know the exact positions of c_l and c_r. If we only know their y-coordinates and their relative x-coordinates, i.e., if we know $x(c_r)-x(c_l), y(c_l)$ and $y(c_r)$, then we can compute $y(v_k)$ and the x-offset of v_k *relative* to c_l, that is $x(v_k) - x(c_l)$.

To obtain this, a tree T is constructed during the algorithm. In the first phase we add new vertices, compute their x-offsets and their y-coordinates, and update the x-offsets of one or two vertices. In the second phase, we traverse the tree and compute the final x-coordinates by accumulating the offsets. Suppose that vertex v is a T-ancestor of vertex w. By the *cumulative offset* from v to w, denoted by $c(v,w)$, we mean the sum of offsets along the branch from v to w including that of w but excluding that of v. Note that, if w is a T-ancestor of vertex x, then $c(v,x) = c(v,w) + c(w,x)$. By adding the x-coordinate of the root v_1 of the tree to the cumulative offset from v_1 to a node, one can determine the node's proper x-coordinate. We store for each vertex v the following information:

$$
\begin{array}{rcl}
\mathit{left}(v) & = & \text{the left } T\text{-son of } v \\
\mathit{right}(v) & = & \text{the right } T\text{-son of } v \\
\Delta x(v) & = & x(v) - x(w),\ x\text{-offset of } v \text{ from its } T\text{-father } w \\
x(v) & = & x\text{-coordinate of } v \\
y(v) & = & y\text{-coordinate of } v
\end{array}
$$

If u, v are any two nodes, then let $\Delta x(u,v) = x(v) - x(u)$. In particular, $\Delta x(v) = \Delta x(u,v)$ where u is the father of v. We want to emphasize that the algorithm will store only $\Delta x(v)$ for each v; whenever the value of $\Delta x(u,v)$ is needed, where $v \neq \mathit{left}(u), \mathit{right}(u)$, it has to be computed by finding the lowest common ancestor w of u, v, adding all offsets on the path from w to v and subtracting all offsets on the path from w to u.

In terms of our tree T, when we add V_k, we need to shift $T(c_r)$ to the right. The crucial observation that leads to the linear-time algorithm is that it is not really necessary to know the exact positions of c_l and c_r at the time when we install $V_k = \{z_1, \ldots, z_\ell\}$. If we only know their y-coordinates and the offset $\Delta x(c_l, c_r)$ then for each $i > 1$ we can compute $y(z_i)$ and the x-offset of z_i relative to z_{i-1}, the x-offset of z_1 relative to c_l, and the x-offset of c_r relative to z_ℓ.

We will assume, for simplicity, that all links in T have been initialized to **nil**.

The algorithm consists of two phases. In the first phase we add new vertices, compute their x-offsets and y-coordinates. In the second phase, we traverse the tree and compute final x-coordinates by accumulating offsets.

We begin by embedding V_1 and V_2, where $V_1 = \{v_1, v_2\}$ and $V_2 = \{z_1, \ldots, z_\ell\}$:

for $i := 1$ **to** $\ell - 1$ **do** $right(z_i) := z_{i+1}$;
$right(v_1) := z_1$; $right(z_\ell) := v_2$;
$P(v_1) := (0, 0)$; $P(v_2) := (\ell + 1, 0)$;
for $i := 1$ **to** ℓ **do** $P(z_i) := (i, 1)$;

Now, for each $k = 3, 4, \ldots, K$, we proceed as follows. Let $c_1, \ldots, c_q$ be the outer face of G_{k-1}; and let c_l, c_r be the left- and rightvertex of $V_k = \{z_1, \ldots, z_\ell\}$ in G_{k-1}. Then execute the following steps.

$\alpha := \mu^+(p)$; $\beta := \mu^-(q)$;
Precompute offsets: compute $\Delta_i = \Delta x(c_l, c_i)$, for $i = l + 1, \ldots, r$;
Update vertex c_l: **if** $\alpha > l$ (and thus $r > l + 1$) **then begin**
    $right(c_\alpha) := left(c_l)$;
    if $left(c_l) \neq$ **nil then** $\Delta x(left(c_l)) := \Delta x(left(c_l)) - \Delta_\alpha$;
    $left(c_l) := right(c_l)$
end ;
$right(c_l) := z_1$;
Install V_k: **if** c_l is saturated **then** $\epsilon := 0$ **else** $\epsilon := 1$;
$\Delta x(z_1) := \epsilon$;
$y(z_1) := y(c_r) + \Delta_r - \ell + 1 - \epsilon$;
for $i := 2$ **to** ℓ **do begin**
    $right(z_{i-1}) := z_i$;
    $\Delta x(z_i) := 1$;
    $y(z_i) := y(z_1)$
end ;
$right(z_\ell) := c_r$;
if $\alpha < \beta$ **then begin**
    $left(z_1) := c_{\alpha+1}$;
    $\Delta x(c_{\alpha+1}) := \Delta_{\alpha+1} - \epsilon$;
    $right(c_\beta) :=$ **nil**;
end ;
Update node c_r: **if** $\beta + 1 < q$ **then begin**
    $right(c_{r-1}) := left(c_r)$;
    $\Delta x(left(c_r)) := \Delta x(left(c_r)) + \Delta x(c_r)$;
    $left(c_r) := c_{\beta+1}$;
    $\Delta x(c_{\beta+1}) := \Delta_{\beta+1} - \Delta_r$;
end ;
$\Delta x(c_r) := \Delta_r - \ell + 1 - \epsilon$;

At this point all y-coordinates and x-offsets have already been computed. All that remains to be done is to compute x-coordinates. In order to do so, we invoke AccumulateOffsets$(v_1, 0)$, where AccumulateOffsets is as follows:

```
procedure AccumulateOffsets(v: vertex, δ: integer);
begin
    if v ≠ nil then begin
        x(v) := δ + Δx(v);
        AccumulateOffsets(left(v), x(v));
        AccumulateOffsets(right(v), x(v))
    end
end
```

In Figure 10.10 the construction of the tree and the values of $\Delta x(v)$ are given for the example from Figure 10.1.

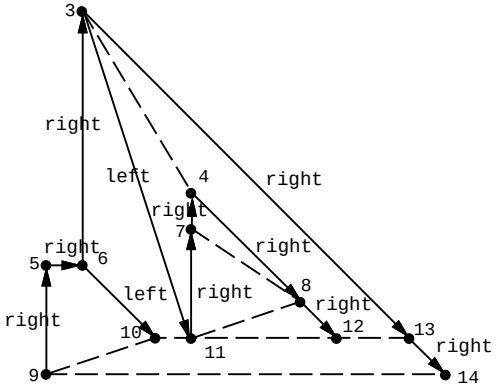
Notice that the algorithm also computes a spanning tree of a 3-connected planar graph with degree at most 3. This gives a new proof (and a linear-time algorithm) for a theorem of Barnette [3]. The general problem is NP-hard, i.e., given a graph, find a spanning tree with degree at most K ($K \geq 2$) (problem ND1 in [38]).

Our algorithm can also be generalized, using the following theorem of Thomassen:

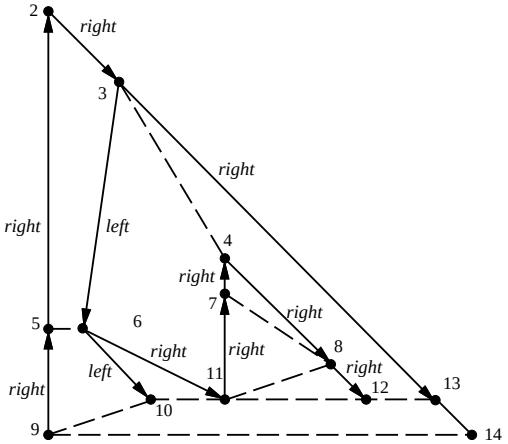
Theorem 10.5.2 *Let G be a plane graph with outer face S such that all vertices not in S have degree ≥ 3. Then G has a convex representation with outerface S if and only if G is internally 3-connected.*

If G satisfies the assumptions in the above theorem and $S = (u_1, \ldots, u_j)$, then adding a vertex z_0 with edges to $u_1, \ldots, u_j$ gives a triconnected graph G^*. By applying the algorithm to G^*, and not adding z_0 in the last phase, we obtain a straight-line and internally convex drawing for G. This yields the following theorem:

Theorem 10.5.3 *If a plane graph G with degree ≥ 3 is convex drawable, then the algorithm, modified as above, constructs in linear time an internally convex drawing of G into a $(n-1) \times (n-2)$ grid.*



(a) The tree of the graph G_7.



(b) The tree of the graph G.

step	adding vertices	$\Delta x(v)$													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
8	9, ..., 14									0	1	1	1	1	1
7	8								1	0	1	1	1	1	1
6	7							0	2	0	1	1	1	1	1
5	5, 6					0	1	0	2	0	2	1	1	1	1
4	4				0	0	1	0	3	0	2	1	1	1	1
3	3			0	0	0	1	0	3	0	2	3	1	2	1
2	2		0	2	0	0	-1	0	3	0	2	4	1	9	1

Figure 10.10: The tree T and $\Delta x(v)$.

Chapter 11

Orthogonal Drawings

In this chapter we consider the problem of drawing a planar graph G on a rectilinear grid with orthogonal edges, i.e., the edges are polygonal chains of horizontal and vertical segments. The vertices are represented by points. This problem has important applications in VLSI-design, and has received a lot of attention during the last years. In Section 9.3.2 several algorithms are described for orthogonal drawings of biconnected 3- and 4-planar graphs, originally coming from Storer [100], Tamassia [102], Tamassia & Tollis [105], and Tamassia, Tollis & Vitter [106].

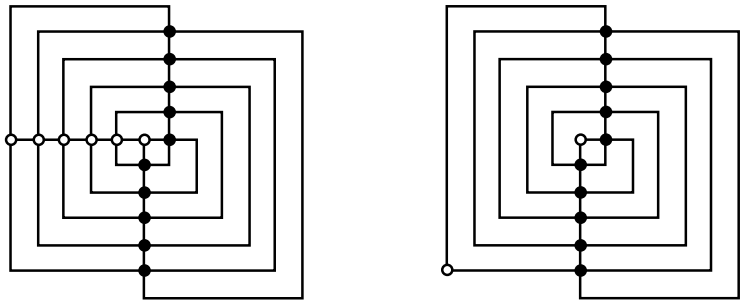
11.1 Orthogonal Drawings of 4-Planar Graphs

We first consider the problem when the input graph G is a triconnected 4-plane graph. Using a variant of Theorem 2 of [106], we can obtain the following lower bound:

Theorem 11.1.1 *There are embedded triconnected 4-planar graphs G_n with $3n+1$ vertices and $6n+1$ edges, for which any layout requires at least $4n+2$ bends.*

Proof: Consider the triconnected plane graph G_n with $3n+1$ vertices, and its layout in Figure 11.1(a), which has $4n+3$ bends. Notice that there are no bends in the edges between two white vertices. The vertices, which had degree 4 initially, have degree 2 now, and are deleted, while connecting the two incident edges. This leads to a biconnected planar graph G'_n with $2n+2$ vertices (see Figure 11.1(b)). It is shown in corollary 4 in [106] that the shown layout in Figure 11.1(b) of G'_n is best possible with respect to the minimum number of bends, which is $4n+2$. If there was a layout for G_n with fewer than $4n+2$ bends, then there was a better layout of G'_n with fewer than $4n+2$ bends, which contradicts Corollary 4 of [106]. □

Let G be a triconnected 4-planar graph. Let an *lmc*-ordering of G be given. We introduce a variable $mark(v_i)$ for each vertex v, which is important when adding $V_k = \{v\}$ to G_{k-1}. v has at most two outgoing edges, say to u_1 and u_2 (from left



(a) The graph G_5. (b) The graph G_5'.

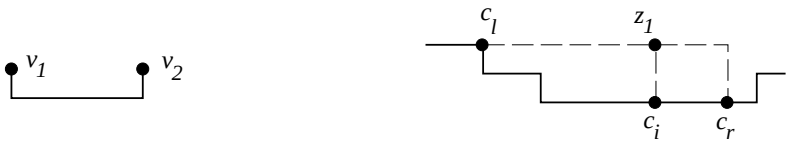
Figure 11.1: Lowerbound of $\frac{4}{3}(n-1)+2$ bends.

to right). If v is the rightvertex of u_1, and v is not the leftvertex of u_2, then we set $mark(v) = left$ otherwise we set $mark(v) = right$.

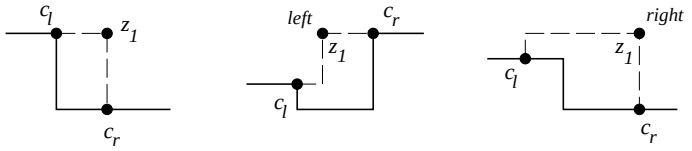
There are four directions to connect an edge at v, namely, *left*, *right*, *up* and *down* of v. A direction is called *free* if there is no edge connected in that direction of v yet. The idea for the algorithm is as follows: we add v to G_{k-1} such that $down(v)$ is not free in G_k. Let c_l and c_r be the left- and rightvertex of v. We connect (c_l, v) at $right(c_l)$, if it is free, otherwise at $up(c_l)$, if it is free, otherwise at $left(c_l)$. The opposite direction is followed for c_r. We want to add v such that when $mark(v) = left$ then $left(v)$ is free after addition. Since v is the rightvertex of u_1, we can use $left(v)$ for the edge (v, u_1) to the left. When $right(u_1)$ is used for (v, u_1), then no bends occur in (v, u_1), otherwise $up(u_1)$ is used, yielding one bend. Similar for $mark(v) = right$.

The algorithm, trying to achieve this as much as possible, can be described in a more elaborate way as follows:

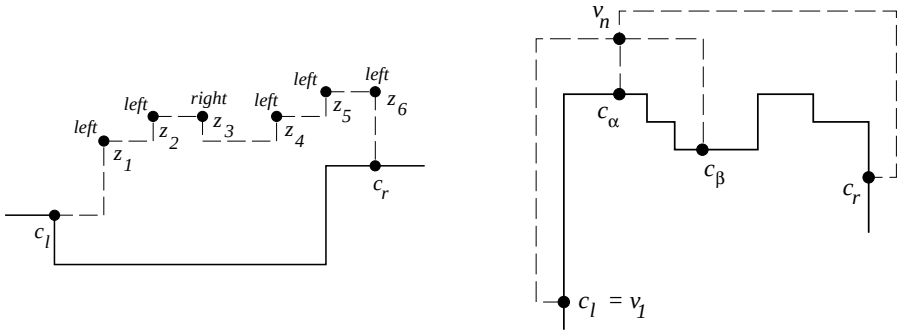
4-ORTHOGONAL(G);
  edge (v_1, v_2) via $down(v_1)$ and $down(v_2)$;
  for $k := 3$ **to** $K - 1$ **do**
    Let $V_k = \{z_1, \ldots, z_\ell\}$;
    • **if** $\ell = 1$ and $in(z_1) = \{c_l, c_i, c_r\}$ **then**
      (c_l, z_1) via $left(z_1)$;
      (c_i, z_1) via $down(z_1)$;
      (c_r, z_1) via $right(z_1)$;
    • **if** $\ell = 1$ and $in(z_1) = \{c_l, c_r\}$ **then**
      if $mark(z_1) = left$ or ($left(c_r)$ free **and** $right(c_l)$ not free) **then**
      (z_1, c_l) via $down(z_1)$ and (z_1, c_r) via $right(z_1)$
      else
      (z_1, c_l) via $left(z_1)$ and (z_1, c_r) via $down(z_1)$;
    • **otherwise** ($\ell > 1$)



Starting with (v_1, v_2). Adding z_1 with $in(z_1) = \{c_l, c_i, c_r\}$.



Adding z_1 with $in(z_1) = \{c_l, c_r\}$.



Adding $z_1, \ldots, z_\ell$. Adding v_n.

Figure 11.2: Adding vertices and faces to obtain an orthogonal drawing.

if *right*(c_l) is free **then** (z_1, c_l) via *left*(z_1) **else** via *down*(z_1);
for $i := 2$ **to** ℓ **do**
if *down*(z_{i-1}) is free **then** (z_{i-1}, z_i) via *down*(z_{i-1}) **else** via *right*(z_{i-1});
if *mark*(z_i) = *left* **then** (z_{i-1}, z_i) via *down*(z_i) **else** via *left*(z_i);
rof;
if *left*(c_r) is free **then** $(z_{\ell-1}, z_\ell)$ via *down*(z_ℓ); (z_ℓ, c_j) via *right*(z_ℓ) **else**
$(z_{\ell-1}, z_\ell)$ via *left*(z_ℓ); (z_ℓ, c_r) via *down*(z_ℓ)
rof;
edges from $c_l, c_{i_2}, c_{i_3}, c_r$ to v_n via *left*(v_n), *down*(v_n), *right*(v_n) and *up*(v_n), resp.;
END 4-ORTHOGONAL

See Figure 11.2 for an illustration of the different cases.

There are several ways for computing the coordinates. Here we briefly describe the method, given by Biedl & Kant [8]: Remark that the y-coordinate of a vertex is never changed later, so we only have to worry about the x-coordinates. The crucial observation is that we need not know the values of the x-coordinates of incoming edges of v_i when adding v_i. We can do the following trick: throughout the algorithm maintain a list *Columns*. Every embedded vertex v contains a pointer $x(v)$ to one element of *Columns*. Whenever we want to add a column, we add a new element in *Columns*. By storing a list as a sequence of pointers we can do so without changing any of the x-values of vertices, already visited. The final x-coordinates are computed by traversing *Columns* and assigning ascending values to each element. Every vertex and bend then checks the value of the element it points to and stores it as its x-coordinate. This yields a planar orthogonal drawing.

Lemma 11.1.2 *The number of bends is at most* $\lceil \frac{3}{2}n \rceil + 3$. *One edge has at most three bends, all other edges have at most 2 bends.*

Proof: Let $V_k = \{z_1, \ldots, z_\ell\}$. Assume first $\ell > 1$. For every vertex z_i, $down(z_i)$ and either $left(z_i)$ or $right(z_i)$ is used. The edge to $down(z_i)$ always requires one extra bend, the other ones not. If $mark(z_1) = right$, and $right(z_1)$ is used by (z_1, z_2), then later an extra bend is required for the outgoing edge via $left(z_1)$. A similar holds for z_ℓ when $mark(z_\ell) = left$. Hence this implies at most ℓ bends.

If $\ell = 1$ then we have at most two bends if $in(z_1) = 3$, so assume $in(z_1) = 2$. The incoming edge, using $down(z_1)$, gets no extra bends, the other edge gets one more bend. If $right(c_l)$ and $up(c_r)$ are both free, then both edges, (c_l, z_1) and (c_r, z_1) are straight lines (using $left(z_1)$ and $down(z_1)$, resp). However, if $mark(z_1) = left$, then an extra bend is required for the outgoing edge of z_1, using $right(z_1)$. We assign this extra bend to step k. Similar when $mark(z_1) = right$, hence in all cases at most one extra bend is introduced when $in(z_1) = 2$.

Also a similar assignment follows for edge (v_1, v_2): we assign the extra bends of edges, using the connection $left(v_1)$ or $right(v_2)$, to step 1. Summarizing this leads to the following table:

step	# vertices	# edges	# bends
$\ell = 1$, $in(z_1) = 2$	1	2	1
$\ell = 1$, $in(z_1) = 3$	1	3	2
$z_1, \ldots, z_\ell$	ℓ	$\ell + 1$	ℓ

Every vertex v has $deg(v) \leq 4$, thus $m \leq 2n$. Consider the steps $2, \ldots, K-1$ in which $n-3$ vertices and at most $2n-5$ edges are added. Adding $V_k = \{z\}$ with $in(z) = 3$ occurs at most $\lceil \frac{n}{2} \rceil - 2$ times, because then at most $\lceil \frac{3}{2}n \rceil - 6$ edges are added, and at most $\lfloor \frac{n}{2} + 1$ edges are added by at most $\lfloor \frac{n}{2} \rfloor - 1$ vertices. This yields at most $2(\lceil \frac{n}{2} \rceil - 2) + \lfloor \frac{n}{2} \rfloor - 1 + 8 = \lceil \frac{3}{2}n \rceil + 3$ bends. The edge, using $up(v_n)$ has at most three bends, all other edges have at most 2 bends. □

Lemma 11.1.3 *The gridsize is at most $n \times n$.*

Proof: The increase in height in step k, $1 < k < K$, is at most ℓ, where $V_k = \{z_1, \ldots, z_\ell\}$. In step 1 the increase in height is at most one, and in step K two, which proves the total height of n.

For the total width, we consider the different cases for step k, $1 < k < K$. Let $V_k = \{z_1, \ldots, z_\ell\}$. If $\ell > 1$, then $x(z_i) - x(z_{i-1}) = 1$ $(1 < i < \ell)$. If $x(z_\ell) - x(z_{\ell-1}) > 1$ then there is no increase in width at all in this step, since then $x(c_r) - x(c_l) > \ell - 1$. If (c_l, z_1) is horizontal, then this means an increase of one in width, if (c_l, z_1) is vertical and $mark(z_1) = right$, then one outgoing edge of z_1 has to go via $l(z_1)$, hence this means also an increase of one in width later. We assign this increase to step k. A similar holds for (z_ℓ, c_r). Since $x(c_r) \geq x(c_l) + 1$ in step $k - 1$, it follows that the increase in width is at most ℓ in step k.

If $\ell = 1$ and $in(z_1) = 2$, then the width increases by one, due to the fact that an extra column might be necessary for the outgoing edge of z_1 via $left(z_1)$ when $mark(z_1) = right$ (similar when $mark(z_1) = left$). If $\ell = 1$ and $in(z_1) \geq 3$ then the width does not increase.

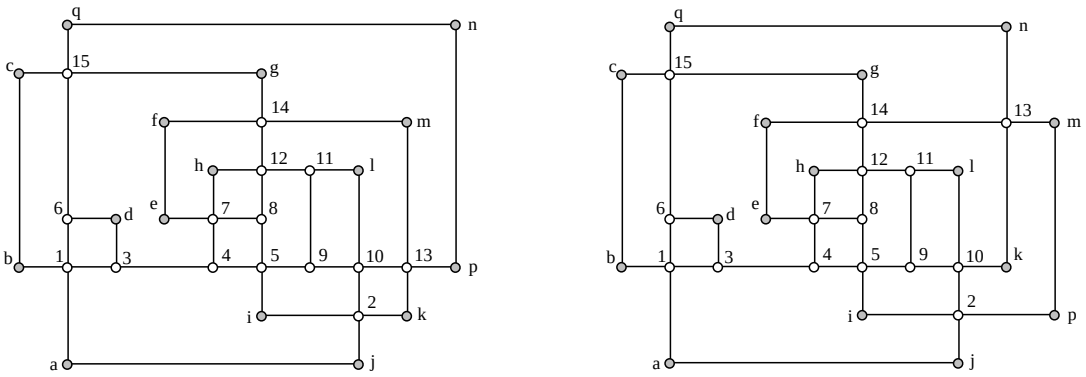
For (v_1, v_2) we assign the extra columns, required by the outgoing edges via $left(v_1)$ and $right(v_2)$ to step 1, yielding a starting width of three units. This gives the following table:

step	# vertices	increase in width
1, adding (v_1, v_2)	2	3
k, $\ell = 1$ and $in(z_1) = 2$	1	1
k, $\ell = 1$ and $in(z_1) = 3$	1	0
k, $\ell > 1$	ℓ	ℓ
K, adding v_n	1	0

This leads to a total width of at most n. □

Indeed, in our solution, the edge, using $up(v_n)$ has at most three bends, all other edges have at most two bends. How can we avoid the edge with three bends? Indeed, Even & Granot proved that any orthogonal drawing of the 4-planar triangulated planar graph on 6 vertices (octahedron) requires at least one edge with at least three bends [29]. In our case, if there is a vertex v with $deg(v) = 3$ then we can set $v_n = v$. Otherwise let $n > 6$. Then there is a face with at least 4 vertices, which we choose to be the outerface. Let $v_{n_1}, \ldots, v_{n_4}$ be the neighbors of v_n from left to right. If edge (v_n, v_{n_4}) uses $right(v_{n_4})$, then we change the four directions of v_{n_4} such that $up(v_{n_4})$ is used for (v_{n_4}, v_n). Since $mark(v_{n_4}) = right$ (by definition), it follows that at most one extra bend is introduced. Moreover, since $n_4 \neq 2$ it follows that all other edges still have at most 2 bends. This completes the following theorem.

Theorem 11.1.4 *There is a linear time and space algorithm to draw every triconnected 4-planar graph G orthogonally on an $n \times n$ grid with at most $\lceil \frac{3}{2}n \rceil + 4$ bends, such that every edge has at most two bends and length $O(n)$ if $n > 6$.*



(c) Orthogonal drawing.

(d) At most 2 bends in every edge.

Figure 11.3: Orthogonal drawing of the 4-planar graph of Figure 10.1.

In Figure 11.3 the orthogonal drawing of the graph of Figure 10.1 is given. In particular, in Figure 11.3(d) it is shown how to change v_k such that all edges have at most 2 bends. The previous bound on the number of bends was $2n + 4$, given by Tamassia & Tollis [105]. Hence our algorithm improves this result considerably for triconnected planar graphs. In the algorithm of Tamassia & Tollis, every edge gets at most 4 bends, hence we also improve this bound. Very recently, Biedl & Kant presented a linear time algorithm for constructing an orthogonal repesentation of a connected planar graph on an $n \times n$ grid, having at most $2n + 2$ bends, and every edge is bent at most twice. Notice that at most 2 bends in every edge is best possible, because if the planar graph contains a separating triangle on the vertices v_i, v_j, v_k, then at least one edge of the separating triangle has at least 2 bends in any orthogonal drawing.

11.2 Orthogonal Drawings of 3-Planar Graphs

11.2.1 Triconnected 3-Planar Graphs

In this section we present a linear time and space algorithm to draw every 3-planar graph with at most $\lfloor \frac{n}{2} \rfloor + 1$ bends on an $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$ grid. This improves all previous bounds (from [100, 105]) and matches the worst-case lower bounds and, hence, is best possible. An interesting side-effect is that there is a spanning tree using $n - 1$ straight-line edges. All $m - n + 1 \leq \lfloor \frac{n}{2} \rfloor + 1$ non-tree edges have at most one bend.

Assume first that G is triconnected. By Euler's formula, n is even, $m = \frac{3}{2}n$ and $f = \frac{n}{2} + 2$. Let an *lmc*-ordering of G be given. Similarly as in Section 11.1 there are four directions to connect an edge at v, namely, *left*(v), *up*(v), *right*(v) and *down*(v). Every vertex v (except v_1, v_2 and v_n) has one outgoing edge, and we connect this

edge via $up(v)$ at v. We start with placing v_1 and v_2 at $(0,1)$ and $(1,1)$. edge (v_1, v_2) goes via $down(v_1)$ and $down(v_2)$, hence via $(0,0)$ and $(1,0)$. In step 2, the vertices $z_1, \ldots, z_\ell$ of V_2 are placed on the horizontal line between v_1 and v_2, i.e., via $right(v_1)$ and $left(v_2)$. In every step k, $3 \leq k < K$, we place $z_1, \ldots, z_\ell$ also on a horizontal line of height $1 + \max\{y(c_l), y(c_r)\}$, with c_l and c_r the left- and rightvertex of V_k. If $\ell > 1$ then we shift the drawing such that $x(z_1) = x(c_l)$ and $x(z_\ell) = x(c_r)$. Since $in(V_k) = 2$ for $2 \leq k < K$ and $in(v_n) = 3$, it follows that $K = f$, with f the number of faces in G. Notice that $f = \frac{n}{2} + 2$ (n is even). The complete algorithm can now be described as follows:

3-ORTHOGONAL
  $P(v_1) := (0,1); P(v_2) := (1,1);$
  for $k := 3$ **to** $f - 1$ **do**
    assume we add $z_1, \ldots, z_\ell$ ($\ell \geq 1$), from c_l to c_r;
    $y(z_1) := \cdots := y(z_\ell) := 1 + \max\{y(c_l), y(c_r)\}$;
    update $x(c_l)$ and $shift(c_r)$;
    $x(z_1) := x(c_l)$;
    for $l := 2$ **to** $\ell - 1$ **do** $x(z_l) := x(z_1) + l - 1$ **rof**;
    if $\ell > 1$ **then** $x(z_\ell) := \max\{x(z_1) + \ell - 1, x(c_r) + shift(c_r)\}$;
    $shift(c_r) := \max\{shift(c_j), x(z_\ell) - x(c_j)\}$
  rof;
  $P(v_n) := (x(c_l), 1 + \max\{y(c_l), y(c_l), y(c_r)\})$, where $in(v_n) = \{c_l, c_l, c_r\}$;
  for $k := f$ **downto** 2 **do**
    assume we added $z_1, \ldots, z_\ell$ from c_l to c_r;
    for $i := 1$ **to** ℓ **do** $x(z_i) := x_{insert}(z_i) + \sum_{1 \leq j \leq i} shift(z_j)$ **rof**;
    if $\ell = 1$ **then** $shift(c_r) := shift(z_1)$ **else** $shift(c_r) := x(z_\ell) - x_{insert}(c_j)$
  rof;
END 3-ORTHOGONAL

Lemma 11.2.1 *The number of bends is at most* $\frac{n}{2} + 2$.

Proof: Since $m = \frac{3}{2}n$, we add at most $\frac{n}{2} - 2$ times a vertex v with $in(v) = 2$, each one introduces one bend. The edge (v_1, v_2) introduces 2 bends, as well as adding v_n. □

Lemma 11.2.2 *The gridsize is at most* $\frac{n}{2} \times \frac{n}{2}$.

Proof: Edge (v_1, v_2) gives 1 unit in X- and Y-direction. Then we add $\frac{n}{2} - 1$ times a face with $\ell \geq 1$ vertices, increasing the X-direction with at most $\ell - 1$ units and the Y-direction (except the first time) by 1 unit. Adding v_n increases the Y-direction by 1 unit. Counting this together leads to at most $\frac{n}{2}$ units in X-direction and $\frac{n}{2}$ units in Y-direction. □

In Figure 11.4(b), an example is given of a triconnected 3-planar graph.

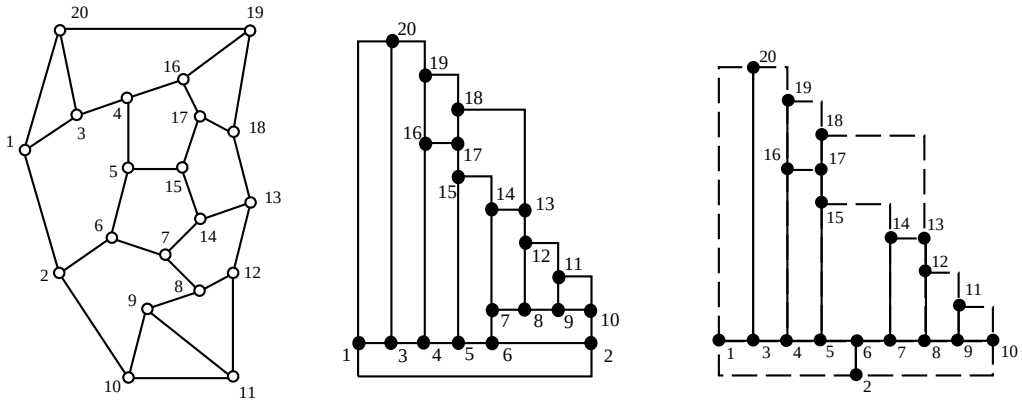


Figure 11.4: Orthogonal drawing of a triconnected 3-planar graph.

We can change the drawing as follows, such that there is one bend less, and there is a spanning tree, using only straight-line edges (if $n > 4$). Let the vertices of the first drawn face be numbered $v_1, v_i, v_{i-1}, \ldots, v_3, v_2$. We place $v_1, v_i, v_{i-1}, \ldots, v_3$ on a horizontal line, and place v_2 on $(x(v_3), y(v_3) - 1)$. Let F' be the other face, where (v_2, v_3) belongs to. Let $v_j, \ldots, v_k$ be the other vertices of F'. We draw $v_j, \ldots, v_k$ on a horizontal line on height $y(v_3)$, as shown in Figure 11.4(c). The remaining faces are drawn similar as before. Notice that using this strategy, every triconnected planar graph G with n vertices can be drawn orthogonally on a grid of size at most $\frac{n}{2} \times (\frac{n}{2} - 1)$, with at most $\frac{n}{2} + 1$ bends (n is even), in which there is a spanning tree, using only straight-line horizontal and vertical edges. All non-tree edges have at most one bend (if $n > 4$).

We notice that better bounds can be obtained if the dual graph G^* of G is a 4-connected planar graph in which all internal faces are triangles. It has been shown by Bhasker & Sahni [7] that in this case G can be drawn orthogonally in linear time such that there are at most 4 bends. In Chapter 13 we show that by changing the canonical ordering to 4-connected triangular planar graphs, and applying the drawing method of He [47], we can achieve the same orthogonal drawing in a very simple way in linear time and space.

11.2.2 Drawing Biconnected 3-Planar Graphs

In this section we generalize the results of Section 11.2.1 to biconnected 3-planar graphs G. Recall the definitions from Section 2.4 with respect to triconnected components, the SPQR-tree, $skeleton(b_i)$ and $pertinent(b_i)$. Let the triconnected components of G be given. From these components we construct the SPQR-tree T_{SPQR} of G. Let for each node b_i in T, s_i, t_i be the poles of b_i, and let $B_i = skeleton(b_i)$. We root T at an arbitrary S-node b_r. Since every vertex has degree 2 or 3 it follows

that the only bonds which can occur are triple bonds. Let b_i be a P-node in T. We claim that b_i has no R-nodes as neighbors. Assume not, and let b_j be an R-node, adjacent to P-node b_i in T. But in $pertinent(b_j)$ there are ≥ 3 paths between s_i and t_i. But then in G, $deg(s_i) > 3$ and $deg(t_i) > 3$. Thus two neighbors of b_i in T, say b_{i_1} and b_{i_2}, are S-nodes, and the third neighbor, say b_{i_3}, is a Q- or S-node. b_{i_3} is a child of b_i in T_{SPQR}, and we merge $skeleton(b_{i_3})$ with $skeleton(b_{i_2})$, where b_{i_2} is the other child of b_i. The merged subgraph is a cycle. If b_{i_3} is a Q-node, then it contains edge (s_i, t_i). If b_{i_3} is an S-node, then s_i and t_i are not adjacent. We represent the merged subgraph by an S'-node. In this way we remove all P-nodes from T_{SPQR}. We remove all Q-nodes from T_{SPQR} as well (they are leaves in T_{SPQR}). For every R-node b_i, $skeleton(b_i)$ is a triconnected 3-planar graph; for every S-node or S'-node b_i, $skeleton(b_i)$ is a cycle. Since $deg(v) \leq 3$ for every vertex, the neighbors of an R-node and the parent of an S'-node are S-nodes. The neighbors of an S-node and the children of an S'-node are R- or S'-nodes.

The drawing algorithm for biconnected 3-planar graphs follows the structure of T_{SPQR}. Every triconnected component B_i has exactly two vertices, s_i and t_i, in common with B_j, where $b_j = parent(b_i)$ in T_{SPQR}. The triconnected components B_i, with b_i a leaf in T_{SPQR}, are drawn as follows:

b_i **is an R-node** B_i is drawn by the algorithm 3-ORTHOGONAL with $v_1 = s_i$ and $v_2 = t_i$.

b_i **is an S-node** Let (s_i, s_i') and $(t_i, t_i') \in B_i$, $s_i' \neq t_i$ and $t_i' \neq s_i$. Let B_i' be $B_i - \{s_i, t_i\}$. If $|B_i'| \geq 4$, then draw B_i' as a rectangle such that s_i' and t_i' are placed in the lowerleft and lowerright corner, and 2 other vertices placed in the other two corner points. If $|B_i'| = 3$, then place s_i' in the upperleft corner, t_i' in the lowerright corner, and the other vertex is placed in the upperright corner. If $|B_i'| = 2$, B_i' is an edge, which we draw horizontal. For B_i, set $P(s_i) = (x(s_i'), y(s_i') - 1)$, if $|B_i'| \neq 3$, and $P(s_i) = (x(s_i'), y(s_i') - 2)$ if $|B_i'| = 3$. Set $P(t_i) = (x(t_i') + 1, y(t_i'))$.

b_i **is an S'-node** We draw B_i as a rectangle such that s_i and t_i are the lowerleft and lowerright corner. If one path between s_i and t_i in B_i contains at least 2 other vertices, then vertices are placed in the other 2 corner points.

Let $|B_i| = n'$. If b_i is an R-node, then n' is even. If b_i is an S'-node, then we can draw B_i such that $y(t_i) = y(s_i)$ and $x(t_i) = x(s_i) + \lfloor \frac{n'}{2} \rfloor$. The same equalities hold when b_i is an R-node after deleting (s_i, t_i). If b_i is an S-node, then $y(t_i) = y(s_i) + 1$ and $x(t_i) = x(s_i) + \lfloor \frac{n'}{2} \rfloor$, and all vertices $v \in B_i, v \neq s_i, t_i$, have $x(v) < x(t_i)$ and $y(v) > y(s_i)$. In all cases of b_i the used area is at most $\lfloor \frac{n'}{2} \rfloor \times (\lfloor \frac{n'}{2} \rfloor - 1)$.

We draw a triconnected component B_j, with b_j not a leaf in T_{SPQR}, after every triconnected component B_i is drawn, with $parent(b_i) = b_j$. Let $x(b_i) = x(t_i) - x(s_i)$ and $y(b_i) = y(t_i) - y(s_i)$ in the drawing of B_i, then the idea is to stretch the drawing of B_j such that in the drawing of B_j, $x(t_i) - x(s_i) = x(b_i)$ and $y(t_i) - y(s_i) = y(b_i)$.

Then we can place the drawing of B_i in the drawing of B_j without crossing edges. In particular, we want to stretch the drawing of B_j such that $x(b_j) \leq \lfloor \frac{|B_i|+|B_j|}{2} \rfloor$ and $y(b_j) \leq \lfloor \frac{|B_i|+|B_j|}{2} \rfloor - 1$.

If b_j is an S-node, then all edges of b_j are straight lines. Every child b_i of b_j is an R- or S'-node, thus $y(b_i) = 0$, and we can easily stretch the edge (s_i, t_i) in B_j to length $x(b_i)$. If b_j is an S'-node, then one edge with a bend may occur, if both paths between s_j and t_j have length ≤ 2. Let $P = s_j, v, t_j$ be such a path. But now it follows that if (s_j, v) is a virtual edge, then (v, t_j) is not virtual, otherwise $deg(v) \geq 4$. We can place v such that the incident virtual edge of v is a straight line. Hence we can draw B_j with all virtual edges drawn as straight lines. We can stretch (s_i, t_i) easily such that it has length $x(b_i)$. $y(b_i) \leq \lfloor \frac{|B_i|}{2} \rfloor - 1$, $x(b_i) \leq \lfloor \frac{|B_i|}{2} \rfloor$, and (s_i, t_i) had length ≥ 1, thus the increase in both X- and Y-direction, when inserting the drawing of B_i inside the drawing of B_j is at most $\lfloor \frac{|B_i|}{2} \rfloor - 1$. We can place the rectangle, representing B_i, on the outerface of the rectangle, representing B_j. In this way no crossings occur.

Assume finally that b_j is an R-node, thus B_j is a triconnected graph. This implies that b_i is an S-node, thus B_i is a cycle. Let $(s_i, s_i'), (t_i, t_i') \in B_i, s_i' \neq t_i$ and $t_i' \neq s_i$. Also $y(t_i') = y(s_i')$ holds in the drawing of B_i. Placing B_i depends on the different situations, which can occur for edge (s_i, t_i) in B_j. Figure 11.5 shows these replacements.

Situation (a) occurs when we add one vertex. Situation (b) and (c) are the cases for adding a chain with at least 3 vertices. Situation (d) occurs when we add a chain of length 2 in B_j. But here we may place a bend in the incoming edge of s_i, because we still have a path, using only straight-line edges from t_i to s_i. Also the increase in X-direction by adding a chain of length 2 is 1. Hence Lemma 11.2.1 and Lemma 11.2.2 still hold.

Situation (e) occurs when vertex $s_i = s_j$, i.e., a pole of B_j. A similar situation can occur for vertex t_i. From this replacement it follows that though the size of the total drawing is at most $\lfloor \frac{|B_i|+|B_j|}{2} \rfloor \times (\lfloor \frac{|B_i|+|B_j|}{2} \rfloor - 1)$, the poles s_j and t_j of B_j are not necessarily the corner points of this rectangle. But still $d(s_i), l(s_i)$ and $d(t_i), r(t_i)$ are free. $parent(b_j)$ is an S- or S'-node, and we can draw the edges from s_j via $d(s_j)$ and from t_j via $r(t_j)$ in the same way as described for the edges (s_i, s_i') and (t_i, t_i'). Hence though s_j and t_j are not the corner points of the area, we can add edges from s_j and t_j to the neighbors in B_j without crossing edges in the required area. Situation (f) occurs when (s_i, t_i) is a horizontal edge, belonging to the first added chain between s_j and t_j.

When (s_i, t_i) is the internal incoming edge of v_n of B_j, then we have 2 situations. Situation (g) occurs when $|B_j| > 4$. Then either $l(s_i)$ or $r(s_i)$ is free, and we can use a similar replacement. If $|B_j| = 4$, then $l(s_i)$ and $r(s_i)$ are not free, hence the only possible replacement is as shown in situation (h). This leads to a drawing of size $\lfloor \frac{|B_i|+|B_j|}{2} \rfloor \times (\lfloor \frac{|B_i|+|B_j|}{2} \rfloor - 1)$, and $x(t_j) = x(s_j) + \lfloor \frac{|B_i|}{2} \rfloor - 1$ holds. But adding B_i inside another triconnected graph by drawing the 2 connecting edges horizontal (as

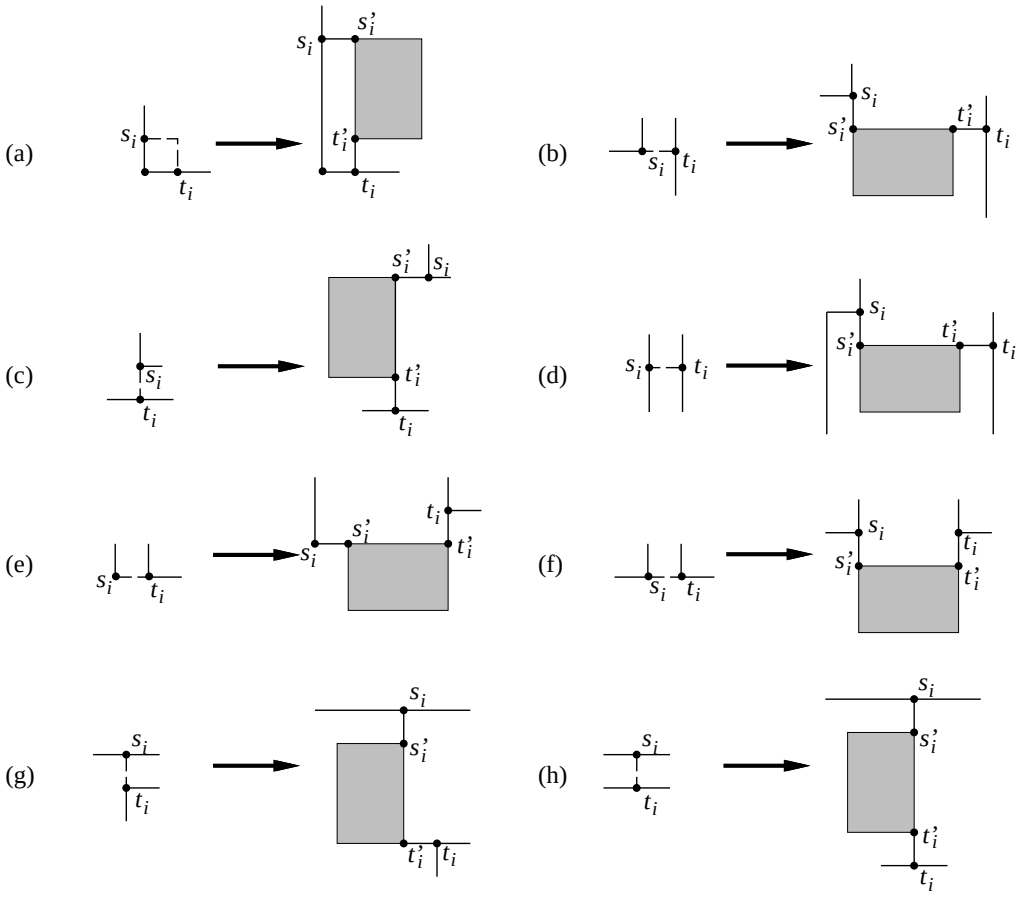


Figure 11.5: Replacing virtual edges by triconnected components.

in situation (h)) already solves the problem. This completes the following lemma:

Lemma 11.2.3 *After replacing a virtual edge (s_i, t_i) by the corresponding orthogonal drawing of B_i in an orthogonal drawing of B_j, the total required grid size is at most $\lfloor \frac{|B_i|+|B_j|}{2} \rfloor \times (\lfloor \frac{|B_i|+|B_j|}{2} \rfloor - 1)$.*

This means that after replacing all virtual edges of B_j by the triconnected components B_i, we obtain an orthogonal drawing of size $\lfloor \frac{n'}{2} \rfloor \times (\lfloor \frac{n'}{2} \rfloor - 1)$, with n' the number of vertices in *pertinent*(b_j). We continue this approach until we are at root b_r of T_{SPQR}. If G is not triconnected, then T_{SPQR} contains an S-node, and we assumed that b_r is an S-node.

One easily observes that no bends are introduced, when we consider an S- or S'-node. Hence the following theorem is obtained:

Theorem 11.2.4 *There is a linear time and space algorithm to draw a biconnected 3-planar graph on a grid of size at most $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$, with at most $\lfloor \frac{n}{2} \rfloor + 1$ bends,*

with the property that there is a spanning tree of $n-1$ straight-line edges, while all non-tree edges have at most 1 bend (if $n > 4$).

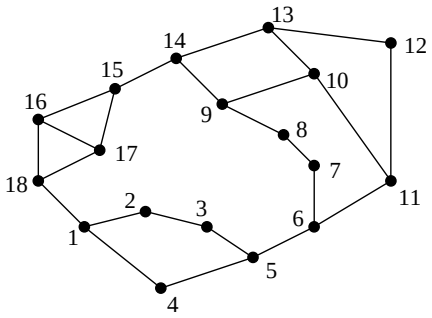
In Figure 11.6 an example is given of a biconnected 3-planar graph G, the SPQR-tree of G, and the corresponding orthogonal drawing of G.

11.2.3 Drawing General 3-Planar Graphs Orthogonally

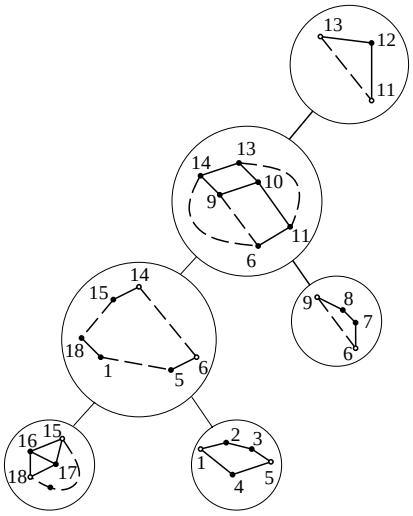
We extend the algorithm 3-ORTHOGONAL to draw arbitrary 3-planar graphs orthogonally. Assume all biconnected components B_i of G are drawn orthogonally on a grid of size at most $\lfloor \frac{|B_i|}{2} \rfloor \times (\lfloor \frac{|B_i|}{2} \rfloor - 1)$. We construct a BC-tree T_{BC} of G. Let $c_i = \mathit{parent}(b_i)$ in T_{BC}. We assume that cutvertex c_i of block B_i is drawn in one corner of the orthogonal drawing of B_i. Since c_i has degree 2 in B_i it follows that the root of the corresponding SPQR-tree of G_i is an S- or S'-node, hence placing c_i in the corner can easily be obtained. Let c_l be the other neighbor of c_i, then c_l is a cutvertex as well. c_l has one or two children in T_{BC}. We first draw these 2 corresponding blocks, and then merge it into one drawing, as shown in Figure 11.7.

There will be no extra bends included in the drawing. Also the required area for drawing blocks B_i and B_j is at most $\lfloor \frac{|B_i|+|B_j|}{2} \rfloor \times \lfloor \frac{|B_i|+|B_j|}{2} \rfloor$, which completes the following theorem:

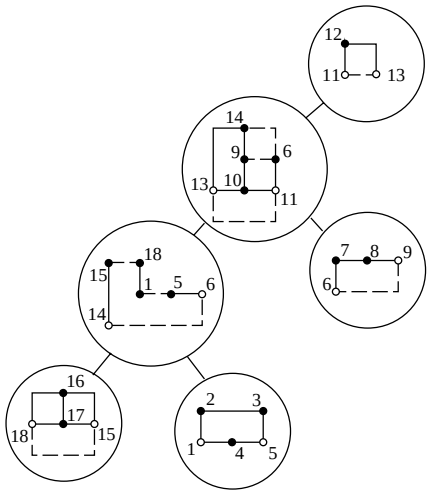
Theorem 11.2.5 *There is a linear time and space algorithm to draw any 3-planar graph on an $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$ grid with at most $\lfloor \frac{n}{2} \rfloor + 1$ bends, with the property that there is a spanning tree of $n-1$ straight-line edges, while all non-tree edges have at most 1 bend (if $n > 4$).*



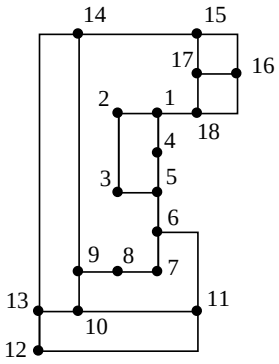
(a) The input graph G.



(b) The SPQR-tree of G.



(c) Drawing the triconnected components.



(d) The complete drawing.

Figure 11.6: Orthogonal drawing of a biconnected 3-planar graph.

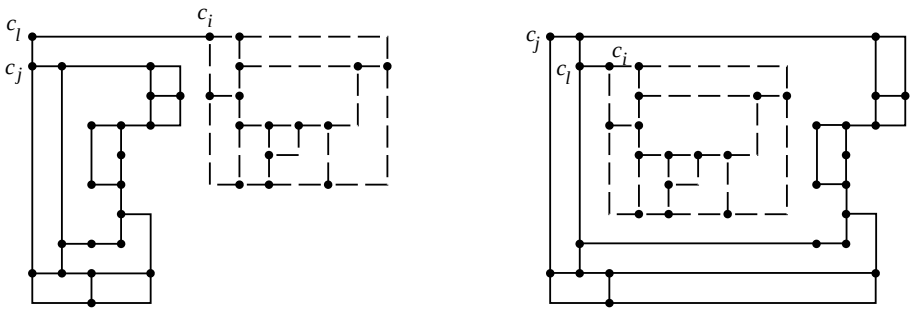


Figure 11.7: Orthogonal drawing of the blocks.

Chapter 12

Hexagonal Drawings

In Chapter 11 we considered the drawings of 3- and 4-planar graphs on a rectilinear grid. In this chapter we consider the drawings of 3-planar graphs on a *hexagonal grid*. In a hexagonal grid there are three directions of the lines (see Figure 12.1): 0 degree lines (here called X-direction), $\pi/3$ degree lines (here called Y-direction) and the $2\pi/3$ degree lines (here called Z-direction). We show in this chapter that we can draw any 3-planar graph on a hexagonal grid, such that at most one edge has bends. This bent edge cannot always be avoided. For instance, consider the graph K_4 in Figure 12.1.

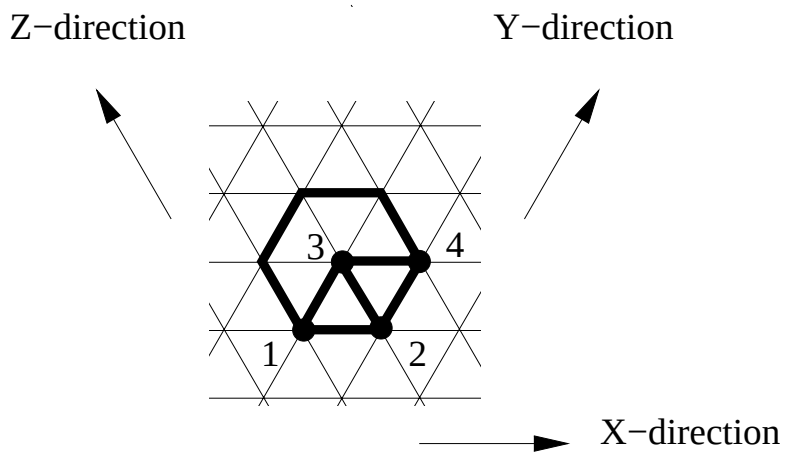


Figure 12.1: Drawing of a K_4 on a hexagonal grid.

Using the drawing on the hexagonal grid, we are able to prove the main result of this chapter, which says that every 3-planar graph G can be drawn with straight-line edges such that the minimum angle is $\geq \frac{\pi}{4}$ if G is triconnected, and $\geq \frac{\pi}{3}$, otherwise. This solves an open problem of Formann et al. [32].

12.1 Triconnected 3-Planar Graphs

Let G be a triconnected 3-planar graph with n vertices. Then n must be even, the number of edges $m = \frac{3}{2}n$ and the number of faces $f = \frac{n}{2} + 2$, since by Euler's formula, $m - n - f + 2 = 0$. Let an lmc-ordering of G be given. In each step k, $3 \leq k < n$ we add one face, F_k, and in the last step K we add v_n. This yields that $K = f$. Let F' be the outerface, and let $F'' \neq F'$ be the other face, containing (v_1, v_n). Let F_n be this incident face of v_n, which does not contain (v_1, v_n). Recall the definitions of the new and internal vertices, and the left- and rightvertex, given in Chapter 2.

Definition 12.1.1 *$E(F_k)$ is the set of edges of F_k, added in step k.*

Definition 12.1.2 *The* base-edge *of F_k, called $be(F_k)$, is the edge which belongs also to the lowest numbered face F_j that is adjacent to F_k.*

Let the base-edge of F_3 be (v_1, v_2). By definition $|E(F_k)| \geq 2$ for all faces $F_k, k \geq 3$, because we add at least one vertex v in step k. The base-edges play an important role in the drawing algorithm. First a length lth is assigned to each base-edge, calculated as follows:

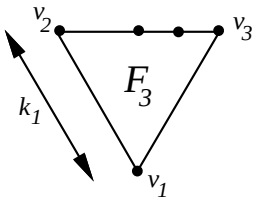
Set $lth(e) = 1$ for all edges $e \in G$;
for $k := f$ **downto** 3 **do** $lth(be(F_k)) := \sum_{e \in E(F_k)} lth(e) - 1$ **rof**;

For each edge e, we will show that the length of e in the resulting drawing, denoted by $length(e)$, is at least $lth(e)$. Let $k_1 = lth((v_1, v_2))$. For the coordinates $P(v_i) = (x(v_i), y(v_i))$ we use the units along the X- and Y-axis. The drawing is constructed as follows:

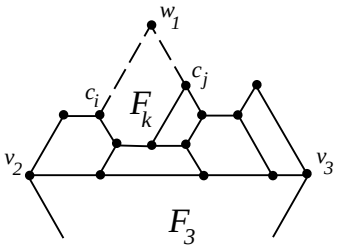
Let v_2 and v_3 be the two neighbors of v_1 in F_3. (v_2 and v_3 are not necessarily neighbors.) We start with drawing F_3 as a triangle with sizes k_1, i.e., $P(v_2) = (0,0), P(v_1) = (k_1, -k_1), P(v_3) = (k_1, 0)$, and the other vertices of F_3 are placed on the horizontal line between v_2 and v_3, such that the length of every edge e is $lth(e)$. These edges form the basis for adding the faces $F_4, \ldots, F_{f-1}$.

If $V_k = \{z\}$, then we walk from leftvertex c_l upwards in Y-direction and from rightvertex c_r upwards in Z-direction. The crossing point is the place for z (see Figure 12.2(b)). If $V_k = \{z_1, \ldots, z_\ell\}$, then we go from c_r one unit in Z-direction and from c_l in Y-direction to the same height (assume $y(c_r) \geq y(c_l)$) and add $z_1, \ldots, z_\ell$ on the horizontal line in between (see Figure 12.2(c)).

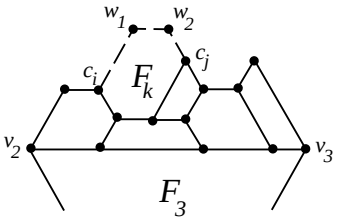
For computing $P(v_n)$ we ignore edge (v_1, v_n). Adding (v_1, v_n) is obtained by going from v_1 one step in X-direction, k_1 steps in Y-direction, k_1 steps in Z-direction and one step in negative X-direction to v_n (see Figure 12.2(d)). Adding the vertices $v_1, \ldots, v_{n-1}$ can be described as follows:



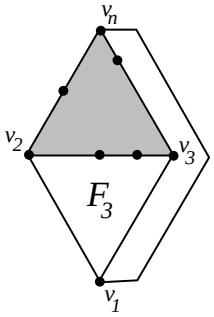
(a) The structure of F_3.



(b) Adding a face with one vertex to the current drawing.



(c) Adding a face with two vertices to the current drawing.



(d) The complete drawing.

Figure 12.2: Illustration of the algorithm HexaDraw.

```
HexaDraw
  P(v1) = (k1, −k1);
  let v2 = z1, ..., zℓ = v3 be the other vertices of F3;
  P(z1) := (0, 0);
  for i := 2 to ℓ do P(zi) := (x(zi−1) + lth((zi−1, zi)), 0) rof;
  for k := 4 to f − 1 do
    assume we add z1, ..., zℓ (ℓ > 1) from cl to cr;
    x(z1) := x(cl);
    if ℓ = 1 then
(*)   y(z1) := y(cr) + x(cr) − x(cl)
    else
(*)   y(z1) := y(z2) := ... := y(zℓ) := max{y(cl), y(cr)} + 1;
      for i := 2 to ℓ − 1 do x(zi) := x(zi−1) + lth((zi, zi−1)) rof;
      x(zℓ) := x(cr) + y(cr) − y(z1)
  rof;
End HexaDraw
```

It is easy to see that the algorithm can be implemented to run in linear time and space. To prove the correctness of the algorithm, we need the following lemmas.

Lemma 12.1.1 *At least one of the internal edges of a face F_k is horizontal.*

Proof: Suppose not. Let c_l be the leftvertex and c_r be the rightvertex of F_k. c_l and c_r had degree 2 before adding F_k and thus (c_l, c_{l+1}) must have Z-direction downwards and (c_{r-1}, c_r) must have Y-direction upwards, if they are not horizontal. But there cannot be a vertex c_α, $l < \alpha < r$ such that $(c_{\alpha-1}, c_\alpha)$ has Z-direction and $(c_\alpha, c_{\alpha+1})$ has Y-direction, because then by HEXADRAW, c_α would have degree 4. Thus there must be at least one horizontal internal edge when adding F_k. □

Lemma 12.1.2 *The internal edges of a new face F_k are: first ≥ 0 edges in Z-direction downwards, then one horizontal edge and then ≥ 0 edges in Y-direction upwards, in this order from left to right.*

Proof: If for an internal vertex c_α holds that $(c_{\alpha-1}, c_\alpha)$ is of Y-direction and $(c_\alpha, c_{\alpha+1})$ is of Z-direction then by definition c_α has degree 2 in G_k and, hence, cannot be internal. Similar when one or two of these edges are horizontal. Hence there is exactly one horizontal edge e. All left internal edges of e are in Z-direction and all edges right from e are in Y-direction. □

Lemma 12.1.3 *e is drawn horizontal $\Longleftrightarrow$ e is a base-edge.*

Proof: $\Longrightarrow$ Let $(c_\alpha, c_{\alpha+1})$ be the horizontal internal edge when adding F_k to G_{k+1}. By lemma 12.1.1, such an edge exists. All internal edges of F_k left (right) from c_α have Z-direction (Y-direction) upwards by lemma 12.1.2. But these edges are added after c_α, because c_α is the rightmost vertex of the face when adding $(c_{\alpha-1}, c_\alpha)$ by the algorithm HEXADRAW. Similarly for $c_{\alpha+1}$. But then $(c_\alpha, c_{\alpha+1})$ belongs to the lowest numbered adjacent face of F_k, hence $(c_\alpha, c_{\alpha+1})$ is the base-edge.

$\Longleftarrow$ Suppose e is a horizontal edge, belonging to F_i and F_j with $i > j$. Thus e is a horizontal internal edge of F_i. Suppose $be(F_i) = e'$ with $e' \neq e$. We know already that e' is horizontal, but then there are two horizontal internal edges when adding F_i. This contradicts lemma 12.1.2. □

Lemma 12.1.4 *For each edge e, $length(e) \geq lth(e)$.*

Proof: By induction on the faces F_k. The base-edge of F_3 is drawn with length k_1. $x(v_3) - x(v_2) = k_1$, which is equal to the sum of $lth(e)$ of all edges e between v_2 and v_3, hence the lemma is correct for F_3.

Assume the lemma is correct for $i = 3, \ldots, k-1$. We show that we add F_k by HEXADRAW such that $length(e) \geq lth(e)$ for every edge in G_k. Let $e' = (c_\alpha, c_{\alpha+1}) =$

$be(F_k)$. From c_α we have ≥ 0 edges in Z-direction upwards to leftvertex c_l and from $c_{\alpha+1}$ we have ≥ 0 edges in Y-direction upwards to rightvertex c_r. Assume w.l.o.g. that $y(c_l) \geq y(c_r)$ and that we add at least two vertices $z_1, \ldots, z_\ell$. From c_l we go one step in Y-direction to place z_1. From c_r we go in Z-direction to the same height to place z_ℓ. (c_l, z_1) and (z_ℓ, c_r) are not base-edges, thus $lth(e) = 1$ and thus $length(e) \geq lth(e)$ for (c_l, z_1) and (z_ℓ, c_r). Moreover, $x(z_\ell) - x(z_1) = x(c_r) + y(c_r) - y(z_1) - x(z_1) = x(c_r) - x(c_l) - (y(c_l) + 1 - y(c_r))$. Notice that from c_α to c_l we go in Z-direction upwards, thus $x(c_l) + y(c_l) = x(c_\alpha) + y(c_\alpha)$, and from $c_{\alpha+1}$ to c_r we go in Y-direction upwards, thus $x(c_r) + y(c_r) \geq x(c_{\alpha+1}) + y(c_{\alpha+1})$. Note also that $y(c_{\alpha+1}) = y(c_\alpha)$, as $(c_\alpha, c_{\alpha+1}) = be(F_k)$, hence horizontal. Thus $x(z_\ell) - x(z_1) = x(c_r) - x(c_l) - (y(c_l) + 1 - y(c_r)) \geq x(c_{\alpha+1}) - x(c_\alpha) - 1 \geq lth(e') - 1$ by induction. $lth((c_l, z_1)) = lth((z_\ell, c_r)) = 1$, thus $\sum_{1 \leq i < p} lth((z_i, z_{i+1})) = lth(e') - 1$, hence also in G_k all edges e have length at least $lth(e)$. □

Since $lth(e) \geq 1$ for all edges e, this lemma proves the correctness of the algorithm HexaDraw.

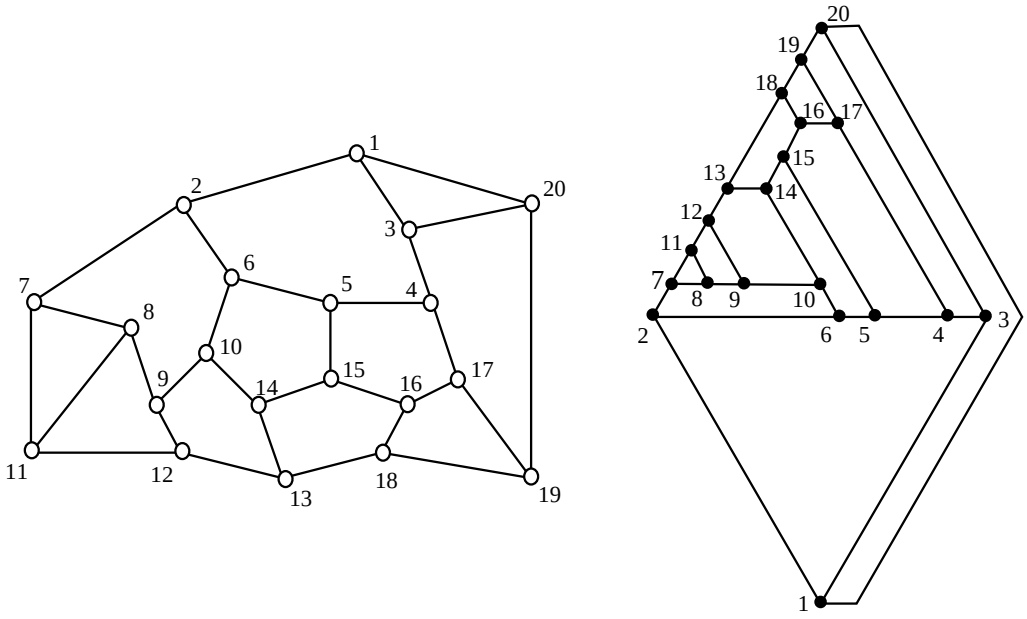


Figure 12.3: The graph of Figure 11.4 with the corresponding drawing by HexaDraw.

Lemma 12.1.5 *The size of the hexagonal grid is* $\frac{n}{2} \times \frac{n}{2}$.

Proof: There are $\frac{3}{2}n$ edges and $\frac{n}{2} + 2$ faces. F'' and F' do not have base-edges, hence there are $\frac{n}{2}$ base-edges. All $\frac{3}{2}n$ edges, except (v_1, v_n), are added to the lth of

a base-edge. The initial length 1 of a base-edge e is ignored when calculating $lth(e)$, and every $lth(e)$ is decreased by 1 in the calculation. Thus $k_1 = lth(be(F_3)) = \frac{3}{2}n - 1 - \frac{n}{2} - \frac{n}{2} = \frac{n}{2} - 1$. Adding (v_1, v_n) increases the size in both Y- and Z-direction by one. This leads to a drawing with sizes $\frac{n}{2}$ in each direction. □

In Figure 12.3 an example is given of a drawing of a triconnected 3-planar graph.

We can use the algorithm HEXADRAW as follows to draw a triconnected 6-planar graph G on a hexagonal grid: replace every vertex v_k with $deg(v_k) > 3$ by the cycle $C(v_k)$ of length $deg(v_k)$ where every vertex of $C(v_k)$ has an edge to a neighbor of v_k. Let G' be the resulting graph. G' is triconnected and 3-planar. We apply HEXADRAW to G' such that face F_k of the corresponding cycle $C(v_k)$ contains at least one internal point p_k of the hexagonal grid, i.e., the size of F in X-, Y- and Z-direction is ≥ 2. This follows when we set $lth(be(F)) = \sum_{e \in E(F_k)} lth(e)$, because then $lth(be(F)) \geq 2$, and we can place $z_1, \ldots, z_\ell$ such that $y(z_1) = \max\{y(c_l) + 2, y(c_r) + 2\}$. When we change G' into G, i.e., replacing cycle $C(v_k)$ by vertex v_k, we place v_k at an internal point p_k of face F_k.

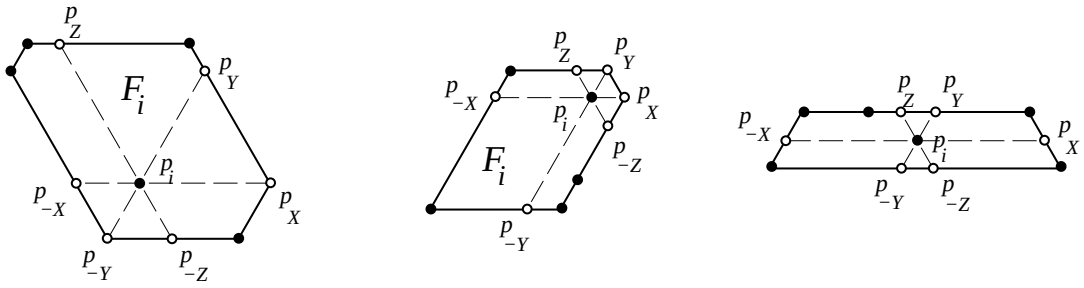


Figure 12.4: Placing p_k inside F_k.

Lemma 12.1.6 *We can choose p_k inside F_k such that all paths between p_k and the vertices of F_k are vertex-disjoint and have at most one bend.*

Proof: Let p_X and p_{-X} be these places on the hexagonal grid, by going from place p_k in X- and negative X-direction, respectively. Analog p_Y, p_{-Y}, p_Z and p_{-Z} are defined. Let $(c_\alpha, c_{\alpha+1})$ be the base-edge of F_k, and assume when we add F_k, we add the vertices $z_1, \ldots, z_\ell$ from c_l to c_r. Assume that $deg(v_k) = 6$, the cases $deg(v_k) = 4$ and $deg(v_k) = 5$ are similar.

If F_k is a 6-gon, then $i \neq \alpha, j \neq \alpha + 1$ and $\ell = 2$. Let $P(p_k) = (x(c_\alpha), y(c_r))$. It can easily be verified that this place is an internal point of F_k. $p_{-Y} = c_\alpha$ and $p_X = c_r$. $p_{-X} = c_l$, or on edge (c_l, c_α) or (c_l, z_1). $p_Y = z_2$, or on edge (z_2, z_1) or (z_2, c_r). $p_Z = z_1$, or on edge (z_1, c_l) or (z_1, z_2). $p_{-Z} = c_{\alpha+1}$, or on edge $(c_{\alpha+1}, c_\alpha)$ or $(c_{\alpha+1}, c_r)$. Hence we can place the edges $(p_k, c_\alpha), (p_k, c_{\alpha+1}), (p_k, c_r), (p_k, z_2), (p_k, z_1)$

and (p_k, c_l) via the points $p_{-Y}, p_{-Z}, p_X, p_Y, p_Z$ and p_{-X} respectively, such that they are vertex-disjoint, and each edge has at most one bend (see Figure 12.4).

If F_k is a 5-gon, then $\alpha = l + 2$ or $r = \alpha + 3$ or $\ell = 3$. Consider the case that $p = 3$, and $r = \alpha + 2$. (The other cases are similar by changing the different directions.) We place p_k at $(x(z_3), y(c_r))$. It follows that $p_X = c_r$ and $p_Y = z_3$. p_{-X} is on edge (c_α, z_1). p_{-Y} is on edge $(c_\alpha, c_{\alpha+1})$; $p_{-Z} = c_{\alpha+2}$, or on edge $(c_{\alpha+2}, c_{\alpha+1})$ or $(c_{\alpha+2}, c_{\alpha+3})$; $p_Z = z_1$, or on edge (z_1, z_2) or (z_1, c_l). Hence we can place the edges $(p_k, c_\alpha), (p_k, c_{\alpha+1}), (p_k, c_{\alpha+2}), (p_k, c_r), (p_k, z_2)$ and (p_k, z_1) via the points $p_{-Y}, p_{-Z}, p_X, p_Y, p_Z$ and p_{-X} respectively, such that they are vertex-disjoint, and each edge has at most one bend (see Figure 12.4).

If F_k is a 4-gon, then we have $p = 4$ or $r = \alpha+4$ or $l = \alpha-3$. Consider the case $p = 4$ in more detail. (The other cases go similar by changing the different directions.) Now $l = \alpha$ and $r = \alpha + 1$ holds, and $y(z_1) \geq y(c_l) + 2$, because we assumed that the length of F_i in each direction is at least 2, which is obtained by $lth(be(F_k)) := \sum_{e \in E(F_k)} lth(e)$. We set $P(p_k) = (x(z_3), y(c_k) + 1)$. Again, it is not difficult to verify that we can place the edges $(p_k, c_\alpha), (p_k, c_{\alpha+1}), (p_k, z_4), (p_k, z_3), (p_k, z_2)$ and (p_k, z_1) via the points $p_{-Y}, p_{-Z}, p_X, p_Y, p_Z$ and p_{-X} respectively, such that they are vertex-disjoint, and each edge has at most one bend (see Figure 12.4). This completes the proof, because F_k cannot be a triangle. □

In Figure 12.4 the different cases are illustrated. This leads to the following theorem:

Theorem 12.1.7 *There is a linear time algorithm to draw a triconnected 6-planar graph on a hexagonal grid of size at most* $(6n - 12) \times (6n - 12)$ *such that each edge has at most 4 bends.*

Proof: Every vertex v_k of G gives rise to at most $deg(v_k)$ vertices in G', thus $n_{G'} \leq 6n-12$. Applying HEXADRAW such that each face F_k has at least one internal point p_k can be obtained by setting $lth(be(F)) = \sum_{e \in E(F_k)} lth(e)$. This implies that every edge (except (v_1, v_n)) are added to lth of a base-edge. The initial length 1 of a base-edge e is ignored when calculating $lth(e)$. This yields that $k_1 = lth((v_1, v_2))$ in G' is $m_{G'} - 1 - f_{G'} + 2 = n_{G'} - 1 = 6n - 13$ (using the formula of Euler). This proves the grid size of $(6n - 12) \times (6n - 12)$. We place every vertex v_k at place p_k. This yields at most one bend to come from p_k to every vertex of F_k. At that vertex another bend is created. The same is done at the other endpoint of this edge, thus there are at most 4 bends in each edge. □

In [105] a linear time algorithm is presented to draw a 4-planar graph on an $O(n^2)$ rectilinear grid with at most 4 bends in each edge. Hence theorem 12.1.7 extends this result in a positive way to triconnected 6-planar graphs. Biedl (personal communication) showed that there exists a class of triconnected 6-planar graphs, requiring $O(n)$ bends in any hexagonal drawing.

We can use a modification of HEXADRAW, such that we obtain a straight-line drawing of a triconnected 3-planar graph G on an $\frac{n}{2} \times \frac{n}{2}$ rectilinear grid. For this goal take the Y-axis perpendicular to the X-axis, and let the Z-axis make an angle of degree $\pi/4$ with the X-axis. See Figure 12.5(a). We now do the algorithm HEXADRAW. The coordinates follow the X- and Y-direction. We finally move vertex v_1 to the point $(k_1 + 1, -1)$, leading to straight-line edges $(v_1, v_2), (v_1, v_3)$ and (v_1, v_n), as shown in Figure 12.5(b). The gridsize in X- and Y-direction is still the same, thereby proving the following theorem.

Theorem 12.1.8 *There is a linear time algorithm to draw a triconnected 3-planar graph planar with straight lines on an $\frac{n}{2} \times \frac{n}{2}$ grid.*

The best bound for the grid size of drawing planar graphs with straight-line edges on a grid is $(n-2) \times (n-2)$ (see Chapter 10), hence Theorem 12.1.8 improves this result by a factor 4 in the case of triconnected 3-planar graphs. Also here, every face is drawn convexly.

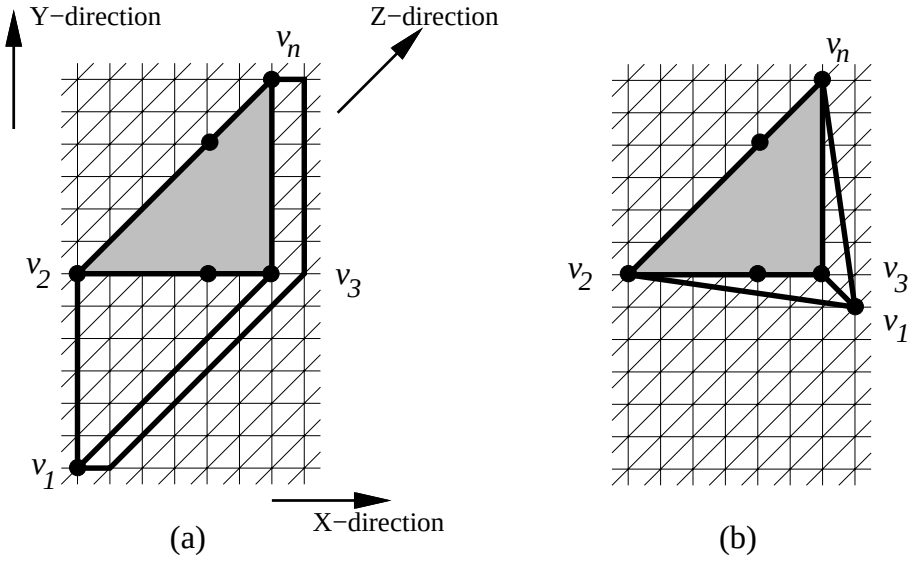


Figure 12.5: Drawing the triconnected 3-planar graph on gridcoordinates.

12.2 Drawing Graphs with Degree at most 3

In this section we show how the algorithm HEXADRAW can be used to draw non-triconnected 3-planar graphs on a hexagonal grid without bends at all. This extension is obtained in a similar way as in Section 11.2. In Section 11.2 we showed

how to use the algorithm for orthogonal drawings of triconnected 3-planar graphs to draw general 3-planar graphs orthogonally. We cannot add edges to G to obtain triconnectivity, since this may yield vertices v with $deg(v) > 3$. Assume w.l.o.g. that G is biconnected, otherwise the biconnected components of G can be drawn separately. (They are connected via bridges with each other, since $\Delta(G) \leq 3$).

Let G be a biconnected 3-planar graph, but not triconnected. Let T_{SPQR} be the SPQR-tree of G. Let for each node b_i in T_{SPQR}, s_i, t_i be the poles of b_i. Root T_{SPQR} at an arbitrary S-node. Since every vertex has degree 2 or 3 it follows that the only bonds which can occur are triple bonds. We merge the two components, children of a P-node, into one component, which is denoted by an S'-node (see also Section 11.2.2). In this way all P-nodes are removed from T_{SPQR}. We also remove all Q-nodes from T_{SPQR}. For every R-node b_i, $skeleton(b_i)$ is a triconnected 3-planar graph; for every S-node or S'-node b_i, $skeleton(b_i)$ is a cycle. Since $deg(v) \leq 3$ for every vertex, the children of an S'- and an R-node are S-nodes, and all neighbors of an S-node are R- or S'-nodes.

The idea for drawing G is as follows: We start with drawing these triconnected components B_i, for which b_i is a leaf in T_{SPQR}, with $B_i = skeleton(b_i)$. This is done as follows:

b_i **is an R-node** Draw B_i by using the algorithm HEXADRAW, with $s_i = v_1$ and $t_i = v_n$.

b_i **is an S-node** Draw all non-virtual edges of the cycle B_i as a horizontal line between s_i and t_i, with virtual edge (s_i, t_i), requiring two bends, below it.

b_i **is an S'-node** Draw the cycle B_i as a parallelepiped on the X- and Y-axis such that s_i and t_i are corner points. (If $|B_i| = 3$, then it is drawn as a triangle.)

Next the triconnected components B_j are drawn, for which all triconnected components B_i, b_i child of b_j in T_{SPQR}, are already drawn. We replace virtual edge (s_i, t_i) in B_j by the drawing of B_i. This is done by "stretching" the drawing of B_j such that the difference in coordinates of s_i and t_i corresponds to the difference in coordinates in the drawing of B_i. If b_i is an R- or S-node, then all edges, except the edge between the poles, are straight lines. If b_i is an S'-node, then all edges are straight lines. In all cases, the poles s_i and t_i are corner points of a rectangle on the X- and Y-axis. Using this strategy, it is quite easy to draw the triconnected components B_j, when b_j is not a leaf. Let b_i be a child of b_j. Let $x(b_i) = x(t_i) - x(s_i)$ and $y(b_i) = y(t_i) - y(s_i)$ in the drawing of B_i.

b_j **is an R-node** Apply HEXADRAW(B_j), in which every virtual edge (s_i, t_i) is changed such that $x(t_i) - x(s_i) = x(b_i)$ and $y(t_i) - y(s_i) = y(b_i)$ in the drawing of B_j.

b_j **is an S-node** The edges of B_j (except edge (s_j, t_j)) are drawn on a horizontal line initially, hence it is easy to change it such that for each virtual edge s_i, t_i, $x(t_i) = x(s_i) + x(b_i)$ and $y(t_i) = y(s_i) + y(b_i)$.

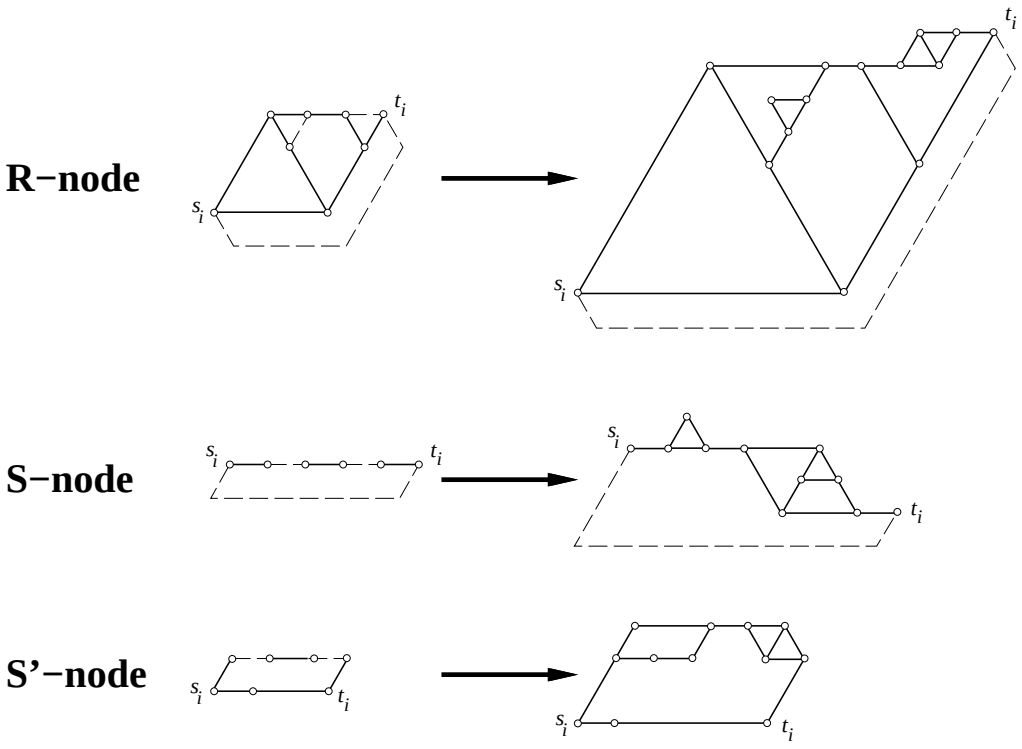


Figure 12.6: Drawing the triconnected components of a 3-planar graph.

b_j **is an S'-node** Draw B_j as a rectangle, but change the virtual edges (s_i, t_i), which are all straight-line edges, such that $x(t_i) - x(s_i) = x(b_i)$ and $y(t_i) - y(s_i) = y(b_i)$ in the drawing of B_j.

In Figure 12.6 the different cases for the drawing are given. We apply this drawing strategy for each node b_j in the SPQR-tree, until $b_j = \mathit{root}(T_{SPQR})$. In this case we assumed that b_j is an S-node. We draw the corresponding cycle B_j with straight lines on the hexagonal grid. Then we know the coordinates of all vertices of B_j. We place all triconnected components B_i, with $b_j = \mathit{parent}(b_i)$, in the drawing of B_j. We continue this until a complete drawing of G is obtained. This completes the following theorem:

Theorem 12.2.1 *There is a linear time algorithm to draw a non-triconnected 3-planar graph on a hexagonal grid with straight-line edges such that the minimum angle is* $\geq \frac{\pi}{3}$.

12.3 Drawings with Straight Lines

In this section we come to the main theorem of this chapter, which is a positive answer to the following question, posed by Formann et al. [32]:

> Does every planar graph with degree ≤ 3 have a planar embedding with straight-line edges such that the smallest angle is at least a constant, independent of the number of vertices?

If the 3-planar graph G is not triconnected, then by Theorem 12.2.1 we can draw G with straight-line edges such that the smallest angle is $\geq \frac{\pi}{3}$. This is also best possible by the following lemma:

Lemma 12.3.1 *There are 3-planar graphs with n vertices, for which in any layout the minimum angle α is at most $\frac{\pi}{3}$. If $n = 6$, then $\alpha \leq \frac{\pi}{4}$, if $n = 4$ then $\alpha \leq \frac{\pi}{6}$.*

Proof: If G contains a triangle, then the best way to draw this is by equal-sized angles of $\frac{\pi}{3}$. If G is a triconnected planar graph with 6 vertices, then the largest face has four vertices. Hence in any drawing of G the outerface F_{out} has at most four vertices. Each vertex of F_{out} has two internal incident angles. The sum of the angles in a 4-gon is 2π. Hence there is an angle with size $\leq \frac{2\pi}{8} = \frac{\pi}{4}$. If $n = 4$, then the largest face has three vertices, and the proof follows in a similar way. □

In this section we prove that every triconnected 3-planar graph can be drawn with straight-line edges such that the minimum angle is at least $\frac{\pi}{4}$, if $n \geq 6$. If $n = 6$ then a drawing with minimum angle $\geq \frac{\pi}{4}$ is easily constructed, so assume $n \geq 8$.

Let an *lmc*-ordering of G be given. let v_α and v_β be the other two neighbors of v_2, unequal to v_1. Let F_x be the face, containing (v_2, v_α) and (v_2, v_β). (It is easy to compute the *lmc*-ordering such that $x = 4$.) Let $k_2 = lth(be(F_x))$ and let $k_1 = \frac{n}{2} - 1 - k_2$. We assume $k_2 \geq k_1$, otherwise we simply put $k_2 = k_1$ (this only enlarges the drawing somewhat).

We change the algorithm HEXADRAW as follows: we draw edge (v_1, v_2) horizontal with length k_1. We draw face F_x as a triangle with sizes k_2, i.e., we draw v_β and v_α k_2 units from v_2 in Z- and Y-direction, respectively. We draw v_3 k_2 units from v_1 in Y-direction. The remaining vertices of F_3 and F_x are placed on the horizontal line between v_β and v_3 with respect to the length of the base-edges (see Figure 12.7). We now apply HEXADRAW to draw the remaining vertices. This gives a hexagonal drawing with bends only in (v_1, v_n). If F_3 is a triangle, then $v_\alpha = v_3$ and $k_1 = 0$.

Let $P(v_\beta) = (0, 0)$, then we change the drawing as follows: we set $P(v_1) = (k_1 + k_2, -k_1 - k_2)$ and $P(v_2) = (2k_1 + 2k_2, 0)$, as shown in Figure 12.7(b). As we used the underlying hexagonal grid, it follows that all angles have size at least $\pi/6$, and only the angles $\angle v_3 v_n v_1, \angle v_n v_1 v_3, \angle v_3 v_1 v_2$ and $\angle v_\alpha v_2 v_\beta$ can have size $< \pi/3$ (see the marked angles in Figure 12.7(b)). If $n = 4$, then 6 angles have size $\pi/6$. This completes the following result:

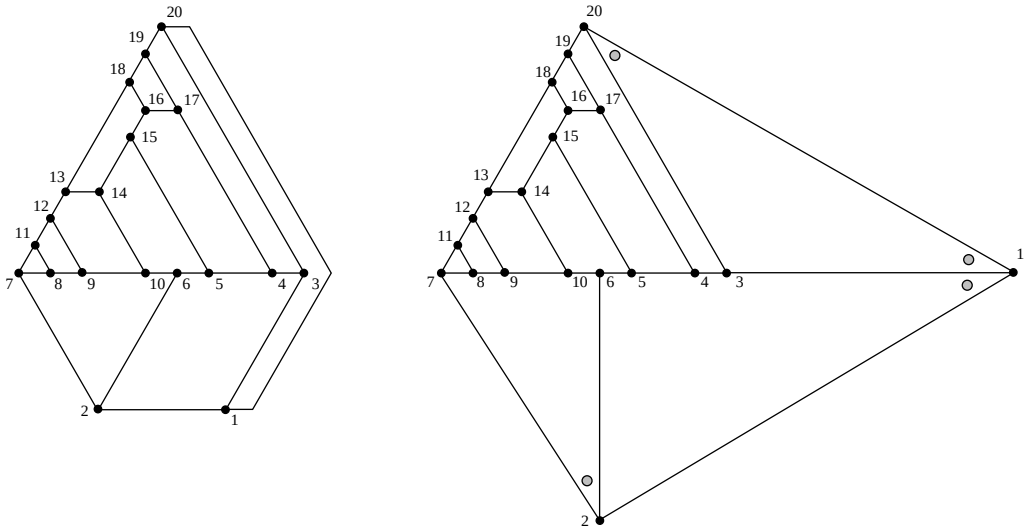


Figure 12.7: Changing the hexagonal drawing such that all angles $\geq \frac{\pi}{6}$.

Lemma 12.3.2 *There is a straight-line drawing of a triconnected 3-planar graph with $n \geq 6$ vertices in which every angle has size $\geq \pi/6$, and at most 4 angles have size $< \pi/3$.*

We now change the drawing algorithm a little such that the minimum angle is $\geq \frac{\pi}{4}$, if $n \geq 8$. Hereto we use the grid model of Figure 12.5. Assume that the outerface F'' has ≥ 5 vertices, and that F_x has ≥ 4 vertices (such a pair of adjacent faces always exists, if $n \geq 8$).

Let v_γ and v_δ be the other neighbors of v_β, unequal to v_2. Let $(v_\beta, v_\gamma) \in F_x$. Let F_y be the other face, containing (v_β, v_γ). By definition, $(v_\beta, v_\gamma) = be(F_y)$. Let $k_3 = lth(be(F_y)), k_2 = lth(be(F_x)) - k_3$, and $k_1 = \frac{n}{2} - 1 - k_2 - k_3$. We enlarge the value k_3 (and also possibly k_1 and k_2), such that $k_3 = 2k_1 + k_2$. We now place edge (v_δ, v_2) horizontal. We place the faces F_x, F_y and F_3 such that all vertices of these faces, except v_2 and v_δ, have the same y-value, and all angles have size $\geq \frac{\pi}{4}$. Hereto we set $P(v_\beta) = (k_3, -k_3)$ and $P(v_2) = (k_2 + 2k_3, -k_3)$, and we set $P(v_\delta) = (0,0), P(v_\gamma) = (k_3, 0), P(v_\alpha) = (k_2 + k_3, 0), P(v_3) = (k_1 + k_2 + k_3, 0)$ and $P(v_1) = (2k_1 + 2k_2 + 2k_3, 0)$, as shown in Figure 12.8. The other vertices of F_y, F_x and F_3 are placed on the horizontal line between v_δ and v_3 with respect to the length of the base-edges. The algorithm HEXADRAW is used to draw the remaining vertices. This gives $P(v_n) = (k_1 + k_2 + k_3, k_1 + k_2 + k_3)$. Since $k_3 = 2k_1 + k_2$ it follows that all edges are horizontal, vertical, or have slopes $+1$ or -1. This gives a drawing with minimum angle $\geq \frac{\pi}{4}$, as shown in Figure 12.8, and completes the proof of the following theorem:

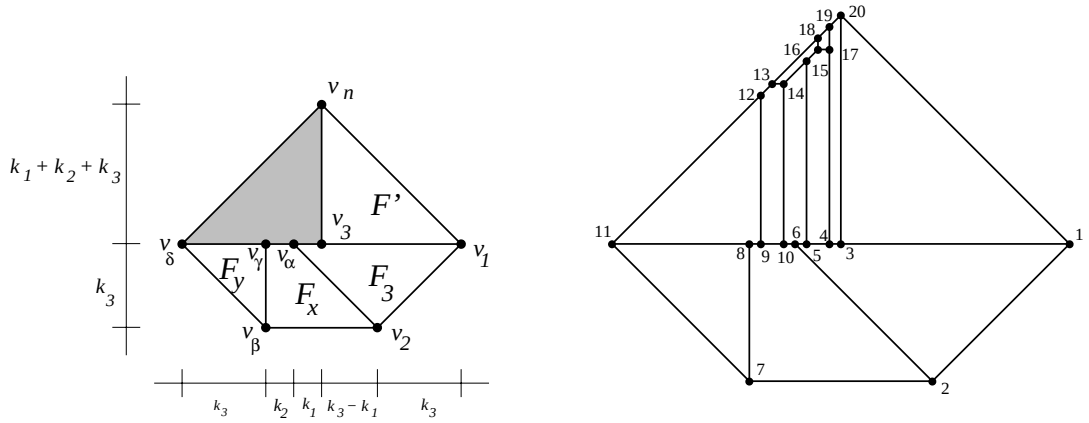


Figure 12.8: Drawing a triconnected 3-planar graph G with minimum angle $\geq \frac{\pi}{4}$.

Theorem 12.3.3 *Every triconnected 3-planar graph with $n \geq 6$ vertices can be drawn with straight-line edges on a grid of size at most $2n \times 2n$ grid such that the minimum angle is $\geq \frac{\pi}{4}$ and all interior faces are drawn convex.*

In Figure 12.8 an example is given of the graph of Figure 12.3.

12.4 Heuristics for Decreasing the Area

In this chapter we considered drawings of planar graphs with degree at most 3 on a hexagonal grid. A linear time algorithm HEXADRAW for this problem is described, leading to a linear-sized grid in the case of triconnected 3-planar graphs. However, in HEXADRAW we always go from the leftvertex in Y-direction and from the rightvertex in Z-direction to the same height, even when there is no reason to go upwards. For example, assume $y(c_l) > y(c_r)$ and (c_l, c_{l+1}) is in Z-direction downwards, then we can place the new vertices $z_1, \ldots, z_\ell$ of face F_k on a horizonal line on height $y(c_l)$ instead of $y(c_l)+1$. To obtain this we change the two lines with (*) both as follows in HEXADRAW:

$\vdots$
if $(y(c_l) > y(c_r)$ **and** $y(c_l) > y(c_{l+1}))$ **or** $(y(c_r) > y(c_l)$ **and** $y(c_r) > y(c_{r-1}))$ **then**
  $y(z_1) := y(z_2) := \ldots := y(z_\ell) := \max\{y(c_l), y(c_r)\}$
else
$\vdots$

In case we add one vertex z_1 and $(y(c_l) > y(c_r)$ **and** $y(c_l) > y(c_{l+1}))$ holds, then we set $x(z_1) := x(c_r) + y(c_r) - y(c_l)$. To prove that the drawing algorithm HEXADRAW works correct after this modification, we prove the following variant of lemma 12.1.2.

(Notice that lemma 12.1.1 still holds and that still holds: e is a base-edge $\Longrightarrow$ e is drawn horizontal.)

Lemma 12.4.1 *All internal edges of a face F_k left from $be(F_k)$ are horizontal or upwards in Z-direction. All internal edges of F_k right from $be(F_k)$ are horizontal or upwards in Y-direction.*

Proof: Suppose there are edges $(c_\alpha, c_{\alpha+1})$ and $(c_\beta, c_{\beta+1})$ in Y- and Z-direction at one side of a horizontal edge, with $\alpha + 1 \leq \beta$. If $\alpha + 1 = \beta$ then by definition c_β has degree 2 in G_{k+1} and, hence, must be a left- or rightvertex. If $\alpha + 2 = \beta$ then there is only one horizontal edge between the edges of Y- and Z-direction, thus $c_{\alpha+1}$ or c_β must have degree 2 in G_{k+1}. If $\beta > \alpha + 2$ then there are more horizontal consecutive edges. If these edges belong to one face of G_{k+1} then the internal vertices $c_{\alpha+2}, \ldots, c_{\beta-1}$ have degree 2 in G_{k+1}, otherwise again $c_{\alpha+1}$ or c_β has degree 2 in G_{k+1}.

Hence there is a horizontal edge $(c_\alpha, c_{\alpha+1})$ such that left from c_α all internal edges are horizontal or upwards in Z-direction. Right from $c_{\alpha+1}$ all internal edges are horizontal or upwards in Y-direction. Similarly to lemma 12.1.3 we can prove that we can choose α such that $(c_\alpha, c_{\alpha+1})$ is the base-edge. □

This lemma implies that in some cases we can decrease the total height considerably. Another optimization is the following. We can use the drawing of Figure 12.7(a) for the hexagonal grid drawing. If k_2 is small, then this decreases the height of the drawing. Let F_x be the face, containing v_2 and two neighbors of v_2, not equal to v_1. Finding an *lmc*-ordering such that k_2 is small is not easy in general, but it becomes solvable when F_x is a triangle. (When there is a triangle F_t in G, then it is not difficult to number G such that $F_t = F_x$.) Let v_α and v_β be the other two neighbors of v_2, unequal to v_1. Then $(v_\alpha, v_\beta) \in F_x$. Let also $(v_\alpha, v_\beta) \in F_y, F_y \neq F_x$. Then $be(F_y) \neq (v_\alpha, v_\beta)$, because F_y contains at least one edge e with $e \in F_3$. Thus $k_2 = 1$, and we obtain a drawing of G within a triangle with sides $\frac{n}{2}$.

The last optimization we notice is when $x(z_\ell) - x(z_1) > \sum_{1 \leq i < p} lth((z_i, z_{i+1}))$ at the moment of adding face F_k. This is the case when $y(c_r) > y(c_{\alpha+1})$ with $(c_\alpha, c_{\alpha+1})$ the base-edge of F_k and c_r the rightvertex of F_k. In HEXADRAW this leads to a drawing with $length((z_{p-1}, z_\ell)) > lth((z_{p-1}, z_\ell))$. We can now subtract $length((z_{p-1}, z_\ell)) - lth((z_{p-1}, z_\ell))$ from $lth((c_\alpha, c_{\alpha+1}))$. We update $lth((a, b))$ for all base-edges (a, b) by visiting the faces $F_{f-1}, \ldots, F_3$, in this order after HEXADRAW. Using the new lth's of the base-edges we again apply HEXADRAW to draw G in linear time on a hexagonal grid of smaller size.

In Figure 12.9, these optimizations have been applied to the example, given in Figure 12.3. Using these observations, we may use a smaller grid than $\frac{n}{2} \times \frac{n}{2}$ to draw triconnected planar graphs. Whether there exist triconnected 3-planar graphs for which any straight-line drawing requires an $\frac{n}{2} \times \frac{n}{2}$ grid remains as an open problem. In Figure 12.10(a) an example of a planar graph of degree 3 is given, requiring an

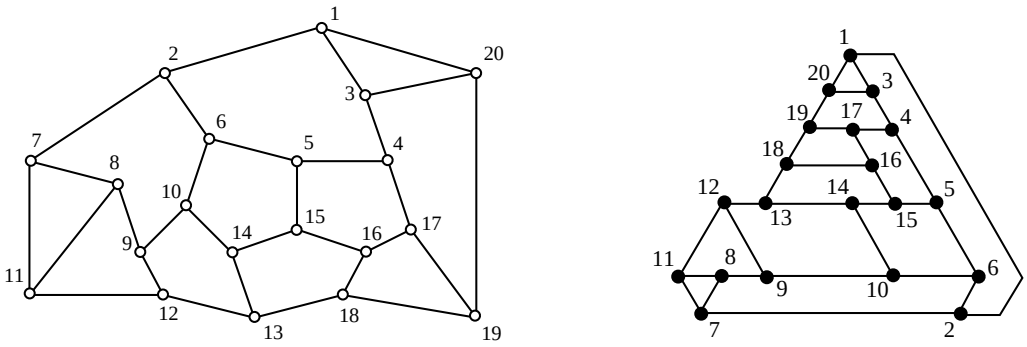


Figure 12.9: Optimizing the drawing of the graph in Figure 12.3.

$(\frac{n}{2}+1)\times(\frac{n}{4}+1)$ grid for a straight-line embedding ($n = 8k$, for some integer $k > 0$), but it is not triconnected. In Figure 12.10(b) a planar graph of degree at most 4 is shown, for which every straight-line drawing requires an $\frac{2}{3}(n-1)\times\frac{2}{3}(n-1)$ grid, if this embedding is used, and $\frac{n+1}{2}\times\frac{n-1}{3}$ otherwise ($n = 6k+1$, for some integer $k > 0$). This gives some indication of the tightness of our algorithm. The triconnected 3-planar graph of Figure 12.10(c) requires an $(\frac{n-1}{3}+2)\times(\frac{n-1}{3}+2)$ grid, if this embedding is used, and $(\frac{n-1}{4}+4)\times(\frac{n-1}{6}+3)$ grid otherwise ($n = 12k+1$, for some integer $k > 0$).

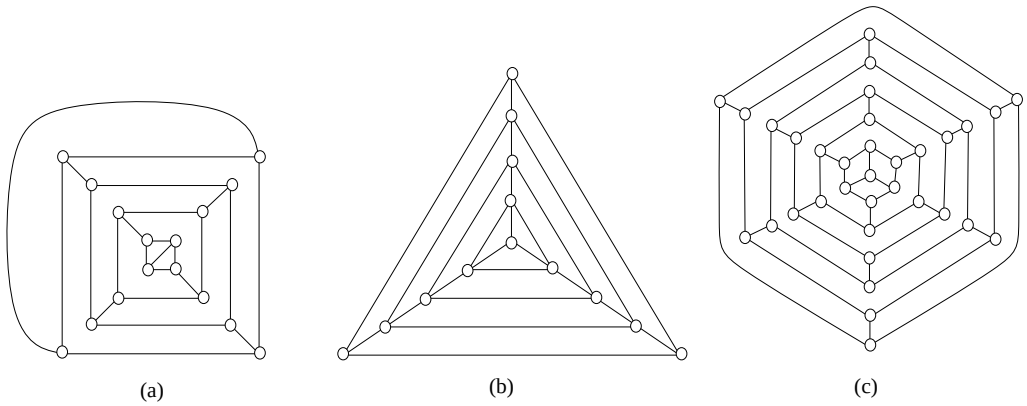


Figure 12.10: Examples of planar graphs for grid size lower bounds.

Chapter 13

Rectangular Duals

13.1 Introduction

In this chapter we consider the problem of representing a graph G by a *rectangular dual*. This is applied in the design of floor planning of electronic chips and in architectural design. A rectangular dual is defined as follows. A *rectangular subdivision system* of a rectangle R is a partition of R into a set , $= \{R_1, R_2, \ldots, R_n\}$ of non-overlapping rectangles such that no four rectangles in , meet at the same point. A *rectangular dual* of a planar graph G is a rectangular subdivision system , and a one-to-one correspondence $R : V \rightarrow$, such that two vertices u and v are adjacent in G if and only if their corresponding rectangles $R(u)$ and $R(v)$ share a common boundary. In the application of this representation, the vertices of G represent circuit modules and the edges represent module adjacencies. A rectangular dual provides a placement of the circuit modules that preserves the required adjacencies. Figure 13.1 shows an example of a planar graph and its rectangular dual.

This problem was first studied by Bhasker & Sahni [6, 7] and Koźmiński & Kinnen [73]. Bhasker & Sahni gave a linear time algorithm to construct rectangular duals [7]. The algorithm is fairly complicated and requires many intriguing procedures. The coordinates of the rectangular dual constructed by it are real numbers and bear no meaningful relationship to the structure of the graph. This algorithm consists of two major steps: (1) constructing a so-called *regular edge labeling* (REL) of G; and (2) constructing the rectangular dual using this labeling. A simplification of step (2) is given in [47]. The coordinates of the rectangular dual constructed by the algorithm in [47] are integers and carry clear combinatorial meaning. However, step (1) still relies on the complicated algorithm in [7]. A parallel implementation of this algorithm, working in $O(\log n \log^* n)$ time with $O(n)$ processors, is given by He [46].

In this paper we present a linear time algorithm for step (1): finding a regular edge labeling. (In [71] another algorithm is presented.) This algorithm extends the canonical ordering of triconnected planar graphs, defined in Section 2.5 to 4-

connected triangular planar graphs. It turns out that the canonical ordering also gives a reduction of a factor 2 in the width of the visibility representation of 4-connected planar graphs. Moreover, using this ordering it is shown that a visibility representation of any planar graph can be constructed on a grid of size at most $(\lfloor \frac{3}{2}n - 1) \times (n - 1)$ grid.

This chapter is organized as follows: in Section 13.2 we present the definition of the regular edge labeling (REL) and we review the algorithm in [47] that computes a rectangular dual from a REL. In Section 13.3, we present the REL algorithm based on the canonical ordering. Section 13.4 discusses the algorithm for the visibility representation of 4-connected planar graphs. Section 13.5 briefly outlines the visibility representation algorithm for general planar graphs.

13.2 The Rectangular Dual Algorithm

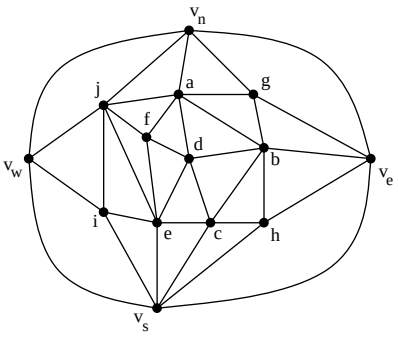
Consider a plane graph H. Let u_0, u_1, u_2, u_3 be four vertices on the exterior face in counterclockwise order. Let P_i $(i = 0, 1, 2, 3)$ be the path on the exterior face consisting of the vertices between u_i and u_{i+1} (addition is mod 4). We seek a rectangular dual R_H of H such that u_0, u_1, u_2, u_3 correspond to the four corner rectangles of R_H and the vertices on P_0 (P_1, P_2, P_3, respectively) correspond to the rectangles located on the north (west, south, east, respectively) boundary of R_H. In order to simplify the problem, we modify H as follows: Add four new vertices v_N, v_W, v_S, v_E. Connect v_N (v_W, v_S, v_E, respectively) to every vertex on P_0 (P_1, P_2, P_3, respectively) and add four new edges $(v_S, v_W), (v_W, v_N), (v_N, v_E), (v_E, v_S)$. Let G be the resulting graph. It is easy to see that H has a rectangular dual R_H if and only if G has a rectangular dual R_G with exactly four rectangles on the boundary of R_G (see Figure 13.1(a) and (b)). Let a *quadrangle* be a cycle of length 4. The following theorem was proved in [6, 73]:

Theorem 13.2.1 *A planar graph G has a rectangular dual R with four rectangles on the boundary of R if and only if (1) every interior face is a triangle and the exterior face is a quadrangle; (2) G has no separating triangles.*

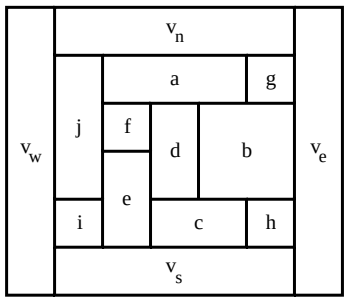
A graph satisfying the conditions in Theorem 13.2.1 is called a *proper triangular planar* (PTP) graph. From now on, we will discuss only such graphs. Note that condition (2) of Theorem 13.2.1 implies that G is 4-connected. Since G has no separating triangles, the degree of any interior vertex v of G is at least 4. (If $deg(v) = 3$, then the triangle induced on the neighbors of v would be a separating triangle.)

The rectangular dual algorithm in [47] heavily depends on the concept of *regular edge labeling* (REL) defined as follows [7, 47]:

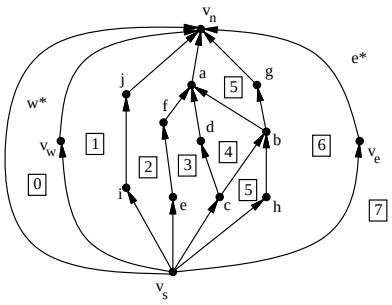
Definition 13.2.1 *A regular edge labeling of a PTP graph G is a partition of the interior edges of G into two subsets T_1, T_2 of directed edges such that:*



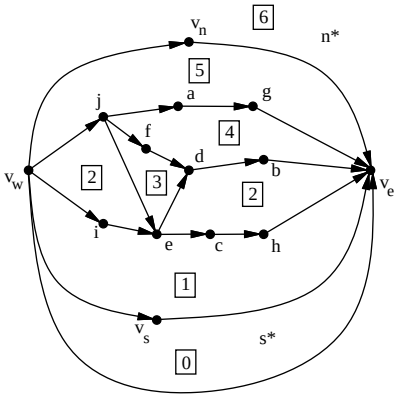
(a) The initial graph G.



(b) A rectangular dual of G.



(c) The graph G_1.



(d) The graph G_2.

Figure 13.1: A PTP graph, its rectangular dual, and the st-graphs G_1 and G_2

1. *For each interior vertex v, the edges incident to v appear in counterclockwise order around v as follows: a set of edges in T_1 leaving v; a set of edges in T_2 entering v; a set of edges in T_1 entering v; a set of edges in T_2 leaving v.*

2. *Let v_N, v_W, v_S, v_E be the four exterior vertices in counterclockwise order. All interior edges incident to v_N are in T_1 and entering v_N. All interior edges incident to v_W are in T_2 and leaving v_W. All interior edges incident to v_S are in T_1 and leaving v_S. All interior edges incident to v_E are in T_2 and entering v_E.*

The regular edge labeling is closely related to *planar st-graphs*, described in Section 9.3.

Let G be a PTP graph and $\{T_1, T_2\}$ be a REL of G. From $\{T_1, T_2\}$ we construct two planar st-graphs as follows. Let G_1 be the graph consisting of the edges of T_1

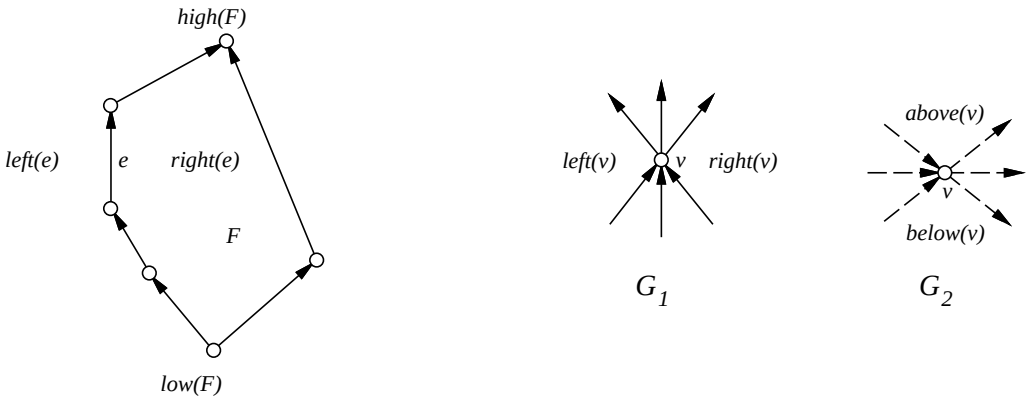


Figure 13.2: Properties of planar st-graphs.

plus the four exterior edges (directed as $v_S \to v_W$, $v_W \to v_N$, $v_S \to v_E$, $v_E \to v_N$), and a new edge (v_S, v_N). G_1 is a planar st-graph with source v_S and sink v_N. For each vertex v, the face of G_1 that separates the incoming edges of v from the outgoing edges of v in the clockwise direction is denoted by $left(v)$. The other face of G_1 that separates the incoming and the outgoing edges of v is denoted by $right(v)$. (See Figure 13.2.)

Let G_2 be the graph consisting of the edges of T_2 plus the four exterior edges (directed as $v_W \to v_S$, $v_S \to v_E$, $v_W \to v_N$, $v_N \to v_E$), and a new edge (v_W, v_E). G_2 is a planar st-graph with source v_W and sink v_E. For each vertex v, the face of G_2 that separates the incoming edges of v from the outgoing edges of v in the clockwise direction is denoted by $above(v)$. The other face of G_2 that separates the incoming and the outgoing edges of v is denoted by $below(v)$. (See Figure 13.2.)

The dual graph G_1^* of G_1 is defined as follows. Every face F_k of G_1 is a node v_{F_k} in G_1^*, and there exists an edge (v_{F_i}, v_{F_k}) in G_1^* if and only if F_i and F_k share a common edge in G_1. We direct the edges of G_1^* as follows: if F_l and F_r are the left and the right face of an edge (v, w) of G_1, direct the dual edge from F_l to F_r if $(v, w) \neq (v_S, v_N)$ and from F_r to F_l if $(v, w) = (v_S, v_N)$. G_1^* is a planar st-graph whose source and sink are the right face (denoted by w^*) and the left face (denoted by e^*) of (v_S, v_N), respectively. For each node F of G_1^*, let $d_1(F)$ denote the length of the longest path from w^* to F. Let $D_1 = d_1(e^*)$. For each interior vertex v of G, define: $x_{left}(v) = d_1(left(v))$, and $x_{right}(v) = d_1(right(v))$. For the four exterior vertices, define: $x_{left}(v_W) = 0$; $x_{right}(v_W) = 1$; $x_{left}(v_E) = D_1 - 1$; $x_{right}(v_E) = D_1$; $x_{left}(v_S) = x_{left}(v_N) = 1$; $x_{right}(v_S) = x_{right}(v_N) = D_1 - 1$.

The dual graph G_2^* of G_2 is defined similarly. For each node F of G_2^*, let $d_2(F)$ denote the length of the longest path from the source node of G_2^* to F. Let D_2 be the length of the longest path from the source node to the sink node of G_2^*. For each interior vertex v of G, define: $y_{low}(v) = d_2(below(v))$, and $y_{high}(v) = d_2(above(v))$. For

the four exterior vertices, define: $y_{low}(v_W) = y_{low}(v_E) = 0$; $y_{high}(v_W) = y_{high}(v_E) = D_2$; $y_{low}(v_S) = 0$; $y_{high}(v_S) = 1$; $y_{low}(v_N) = D_2 - 1$; $y_{high}(v_N) = D_2$.

The rectangular dual algorithm relies on the following theorem from He.

Theorem 13.2.2 ([47]) *Let G be a PTP graph and $\{T_1, T_2\}$ be a REL of G. For each vertex v of G, assign v the rectangle $R(v)$ bounded by the four lines $x = x_{left}(v)$, $x = x_{right}(v)$, $y = y_{low}(v)$, $y = y_{high}(v)$. Then the set $\{R(v)|v \in V\}$ form a rectangular dual of G.*

Figure 13.1 shows an example of the theorem. Figure 13.1(c) shows the st-graph G_1. The small squares in the Figure represent the nodes of G_1^* and the integers in the squares represent their d_1 values. Figure 13.1(d) shows the graph G_2. Figure 13.1(b) shows the rectangular dual constructed as in Theorem 13.2.2. The algorithm for computing a rectangular dual is as follows [47]:

RectangularDual(G);
  construct a regular edge labeling $\{T_1, T_2\}$ of G;
  construct from $\{T_1, T_2\}$ the planar st-graphs G_1 and G_2;
  construct the dual graph G_1^* from G_1 and G_2^* from G_2;
  compute $d_1(F)$ for nodes in G_1^* and $d_2(F)$ for nodes in G_2^*;
  assign each vertex v of G a rectangle $R(v)$ as in Theorem 13.2.2;
End RectangularDual

If we have a REL of a PTP graph, then the rectangular dual can easily be constructed in linear time by this algorithm. In the next section we show how to compute a REL of a PTP graph.

13.3 Computing a REL Using a Canonical Ordering

In this section we consider 4-connected planar triangular graphs. Note that adding an edge connecting two non-adjacent exterior vertices of a PTP-graph G leads to a 4-connected planar triangular graph. So we assume that G is a 4-connected planar triangular graph. Let the exterior vertices of G be u, v, w.

Theorem 13.3.1 *There exists a labeling of the vertices $v_1 = u, v_2 = v, v_3, \ldots, v_n = w$ of G meeting the following requirements for every $4 \leq k \leq n$:*

1. *The subgraph G_{k-1} of G induced by $v_1, v_2, \ldots, v_{k-1}$ is biconnected and the boundary of its exterior face is a cycle C_{k-1} containing the edge (u, v).*

2. *v_k is in the exterior face of G_{k-1}, and its neighbors in G_{k-1} form a (at least 2-element) subinterval of the path $C_{k-1} - \{(u, v)\}$. If $k \leq n-2$, v_k has at least 2 neighbors in $G - G_{k-1}$.*

Proof: The vertices $v_n, v_{n-1}, \ldots, v_3$ are defined by reverse induction. Number the three exterior vertices u, v, w by v_1, v_2 and v_n. Let G_{n-1} be the subgraph of G after deleting v_n. By 4-connectivity of G, G_{n-1} is triconnected, and its exterior face C_{n-1} is a cycle and, hence, admits the constraints of the theorem. Let $v_{n-1} \neq v_1$ be the vertex of C_{n-1} adjacent to both v_2 and v_n in G. By the 4-connectivity, $G - \{v_n, v_{n-1}\}$ is biconnected and its exterior face C_{n-1} is a cycle and, hence, admits the constraints.

Let $k < n-1$ be fixed and assume that v_i has been determined for every $i > k$ such that the subgraph G_i induced by $V - \{v_{i+1}, \ldots, v_n\}$ satisfies the constraints of the theorem. Let C_k denote the boundary of the exterior face of G_k. Assume first that C_k has no interior chords. Suppose $v_1, c_{k_1}, \ldots, c_{k_p}, v_2$ are the vertices of C_k in this order between v_1 and v_2. Then it follows by the 4-connectivity of G that $p \geq 2$. If all vertices $c_{k_1}, \ldots, c_{k_p}$ have only one edge to the vertices in $G - G_k$, then since G is a planar triangular graph, they are adjacent to the same vertex v_j for some $k < j < n$. In this case we also have $(v_1, v_j), (v_2, v_j) \in G$. But then $\{(v_1, v_j), (v_j, v_2), (v_2, v_1)\}$ would be a separating triangle. Hence at least one vertex, say c_{k_α}, has at least 2 neighbors in $G - G_k$. c_{k_α} is the next vertex v_k in our ordering.

Next assume C_k has interior chords. Let (c_a, c_b) $(b > a+1)$ be a chord such that $b-a$ is minimal. Let also (c_d, c_e) be a chord with $e > d \geq b$ such that $e-d$ is minimal. (If there is no such a chord, let $(c_a, c_b) = (c_d, c_e)$ and number the vertices in clockwise order around C_k such that $a = 1 < b = d$ and $e = 1$.) Assume, without loss of generality, that $v_1, v_2 \notin \{c_{a+1}, \ldots, c_{b-1}\}$. If all vertices $c_{a+1}, \ldots, c_{b-1}$ have only one edge to the vertices in $G - G_k$, then since G is a triangular graph, they are adjacent to the same vertex v_j, and we also have $(v_a, v_j), (v_b, v_j) \in G$. But then $\{(v_a, v_j), (v_j, v_b), (v_b, v_a)\}$ would be a separating triangle. Hence there is at least one vertex $c_\alpha, a < \alpha < b$, having at least two neighbors in $G - G_k$ and having no incident chords. c_α is the next vertex v_k in our ordering. □

Theorem 13.3.2 *The canonical ordering can be computed in linear time.*

Proof: We add two labels to each vertex v: *visited*(v), denoting the number of visited and extracted neighbors of v, and *chords*(v), denoting the number of incident chords of v. The algorithm follows the structure of the proof of Theorem 13.3.1.

We start with v_n and v_{n-1} and initialize the labels *visited* and *chords* of their neighbors, after deleting the vertices v_n and v_{n-1}. We compute the ordering in reverse order and update the labels after choosing a vertex v_k as follows: we visit each neighbor v of v_k. Let $c_i, \ldots, c_j$ $(j > i)$ be the neighbors (in this order) of v_k in G_{k-1}. We increase *visited*(c_l) by one, for $i \leq l \leq j$. If $j = i+1$, then there was a chord (c_i, c_j) in G_{k-1}, hence we decrease *chords*(c_i) and *chords*(c_j) by one, since (c_i, c_j) becomes part of C_{k-1}. If $j > i+1$, then for each c_l $(i < l < j)$, we compute *chords*(c_l). If there is a chord (c_l, v), then we also increase *chords*(v) by one. This is done by marking the vertices that are part of the current exterior face.

By Theorem 13.3.1 it follows that, if $k \geq 3$, then there is a vertex v with *visited*(v) ≥ 2 and *chords*(v) $= 0$, and this can be chosen as the next vertex v_k in our ordering. We mark v as being visited. Since there are only a linear number of edges, the canonical ordering is obtained in linear time. □

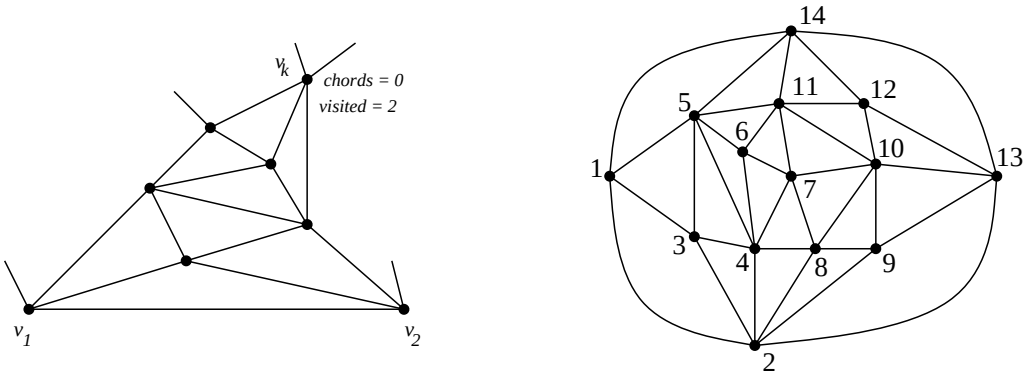


Figure 13.3: Computing the canonical ordering of the graph from Figure 13.1.

To compute a REL of a PTP graph G, we first add an edge connecting two non-adjacent exterior vertices of G. This gives a 4-connected planar triangular graph G'. We compute a canonical numbering of G' and then delete the added edge. The four exterior vertices of G are now numbered as v_1, v_2, v_{n-1}, v_n, respectively. Next we show that a REL of G can be easily derived from the canonical ordering.

First, for each edge (v_i, v_j) of G, direct it from v_i to v_j, if $i < j$. Define the *base-edge* of a vertex v_k to be the edge (v_l, v_k) for which $l < k$ is minimal. The vertex v_k has incoming edges from $c_i, \ldots, c_j$ belonging to C_{k-1} (the exterior face of G_{k-1}), assuming in this order from left to right. We call c_i the *leftvertex* of v_k and c_j the *rightvertex* of v_k. Let $v_{k_1}, \ldots, v_{k_l}$ be the higher-numbered neighbors of v_k, in this order from left to right. We call (v_k, v_{k_1}) the *leftedge* and (v_k, v_{k_l}) the *rightedge* of v_k. Notice that following the definitions of Section 6, $v_{k_1} = \mathit{leftup}(v_k)$ and $v_{k_l} = \mathit{rightup}(v_k)$.

Lemma 13.3.3 *A base-edge cannot be a leftedge or a rightedge.*

Proof: Assume the lemma is false. Suppose the leftedge (v_k, v_{k_1}) of v_k is the base-edge of v_{k_1}. Thus v_k is the lowest-numbered neighbor of v_{k_1}. Since G is triangular, there is an edge between the leftvertex of v_k, say v_i with $i < k$, and v_{k_1}. But this contradicts the fact that (v_k, v_{k_1}) is the base-edge of v_{k_1}. The argument is similar for the rightedges. □

Lemma 13.3.4 *An edge is either a leftedge, a rightedge or a base-edge.*

Proof: The "exclusive or" follows from the previous lemma. We only need to prove that every edge is a leftedge, a rightedge or a base-edge. Let v_k $(3 \leq k \leq n-2)$ be a vertex with incoming edges coming from $c_i, \ldots, c_j$, in this order from left to right. Let (v_k, c_α) be the base-edge of v_k. All vertices c_l $(i < l < j)$ have at least two higher-numbered neighbors, one of them is v_k, the other one is adjacent to (c_l, v_k), hence it is either (c_{l-1}, c_l) or (c_l, c_{l+1}). Thus between c_i and c_α it follows that c_{l+1} is the rightvertex of c_l $(1 \leq l < \alpha)$. Between c_α and c_j vertex c_l is the leftvertex of c_{l+1} $(\alpha \leq l < j)$. Hence the edges (c_l, v_k) are rightedges for $i \leq l < \alpha$ and leftedges for $\alpha < l \leq j$. The edge (c_α, v_k) is a base-edge. Similarly, we can show that the lemma holds for the incoming edges of v_{n-1} and v_n. □

We construct a REL for G as follows: all leftedges belong to T_1, all rightedges belong to T_2. The base-edge (c_α, v_k) of v_k is added to T_1, if $\alpha = j$, to T_2, if $\alpha = i$, and otherwise arbitrarily to either T_1 or T_2. (The four exterior edges belong to neither T_1 nor T_2.)

Lemma 13.3.5 *$\{T_1, T_2\}$ forms a regular edge labeling for G.*

Proof: Let $v_{k_1}, \ldots, v_{k_d}$ be the outgoing edges of the vertex v_k $(3 \leq k \leq n-2)$. It follows from Theorem 13.3.1 that $d \geq 2$. Then (v_k, v_{k_1}) is the leftedge of v_k and is in T_1. (v_k, v_{k_d}) is the rightedge of v_k and is in T_2. The edges $(v_k, v_{k_2}), \ldots, (v_k, v_{k_{d-1}})$ are the base-edges of $v_{k_2}, \ldots, v_{k_{d-1}}$, respectively. Let the vertex v_{k_β} $(1 \leq \beta \leq d)$ be the highest-numbered neighbor of v_k. Then all vertices from v_{k_1} to v_{k_β} have a monotone increasing number, as well as the vertices from v_{k_d} to v_{k_β}. Otherwise there was a vertex v_{k_l} such that $v_{k_{l-1}}$ and $v_{k_{l+1}}$ are numbered higher than v_{k_l}. But this implies that v_k is the only lower-numbered neighbor of v_{k_l}, which is a contradiction with the canonical ordering of G. Hence for every v_{k_l} $(1 < l < d,\ l \neq \beta)$, either $k_{l-1} < k_l < k_{l+1}$ or $k_{l-1} > k_l > k_{l+1}$. Thus, by the construction of T_1 and T_2, the edges (v_k, v_{k_l}) are added to T_1, if $1 \leq l < \beta$, and to T_2, if $\beta < l \leq d$. The edge (v_k, v_{k_β}) is arbitrarily added to either T_1 or T_2. This completes the proof that the edges appear in counterclockwise order around v_k as follows: a set of edges in T_2 entering v_k; a set of edges in T_1 entering v_k; a set of edges in T_2 leaving v_k; a set of edges in T_1 leaving v_k.

Let $v_{1_1}, \ldots, v_{1_d}$ be the higher numbered neighbors of v_1 from left to right. Then $v_{1_1} = v_n$ and $v_{1_d} = v_2$, and by the argument described above, $(v_1, v_{1_2}), \ldots, (v_1, v_{1_{d-1}})$ belong to T_2. Similarly, all outgoing edges of v_2 belong to T_1. All incoming edges of v_{n-1} belong to T_2, and all incoming edges of v_n belong to T_1. This completes the proof. □

Since the construction of $\{T_1, T_2\}$ from the canonical numbering can be easily done in $O(n)$ time, Theorem 13.3.2 and Lemma 13.3.5 constitute our linear time REL algorithm. See Figure 13.3 for the construction of a REL from a canonical ordering.

13.4 Algorithm for Visibility Representation

The *visibility representation* of a planar graph G maps the vertices of G to horizontal line segments and edges of G to vertical line segments. In Section 10.4 we gave a new algorithm for constructing a visibility representation of a triconnected planar graph, using the *lmc*-ordering. Indeed, we observed that if $out(v_k) \geq 2$ for every vertex $v_k, k < n-1$, then the grid size is at most $(n-1) \times (n-1)$. Since this holds for the canonical ordering for 4-connected triangular planar graphs, this leads to the desired grid size for 4-connected planar graphs. (If the 4-connected planar graph is not triangulated, then we can apply any triangulation algorithm, as described in Chapter 6.) In Section 9.3.1 we presented the algorithm VISIBILITY(G) of [96, 104], which constructs in linear time a visibility representation of a graph G. VISIBILITY (G) starts with constructing an st-numbering. If G is a 4-connected planar triangular graph, then we can use the described canonical ordering, since this ordering is also an st-numbering. Figure 13.4 shows an example of applying VISIBILITY(G) of a 4-connected planar triangular graph, using the canonical ordering. We will now show that also the algorithm VISIBILITIY(G) leads to the same grid size.

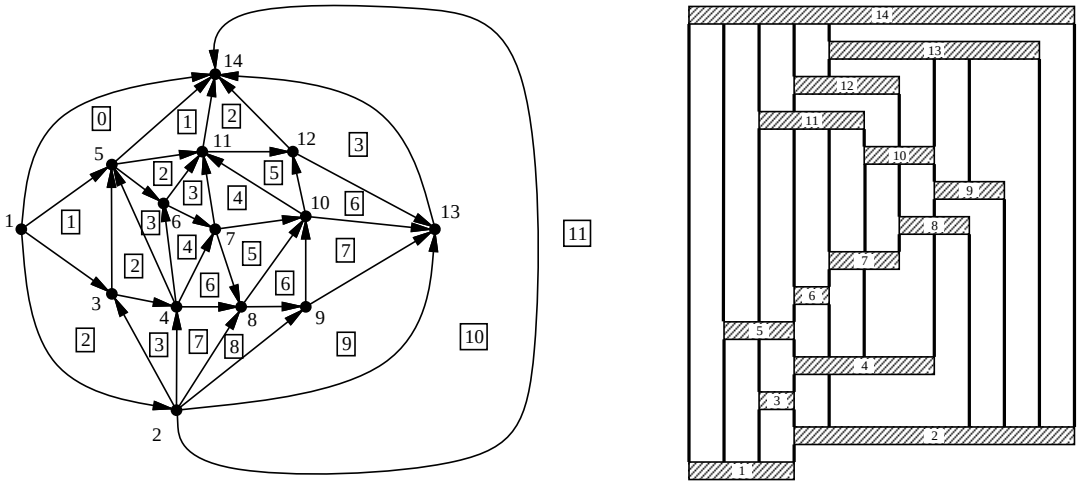


Figure 13.4: The canonical ordering leads to a compact visibility representation.

Theorem 13.4.1 VISIBILITY(G) *constructs a visibility representation of G on a grid of size at most $(n-1) \times (n-1)$.*

Proof: The correctness of VISIBILITY(G) is shown in [96, 104]. We show that the grid size is at most $(n-1) \times (n-1)$. This follows directly for the height, since the length of the longest path from v_1 to v_n is at most $n-1$.

Let s^* be the source node of G^* and t^* be the sink node of G^*. Every vertex v of G corresponds to a face F_v of G^*. If $v \neq v_1, v_2, v_{n-1}, v_n$, then v has ≥ 2 incoming

and ≥ 2 outgoing edges, hence the two directed paths from $low(F_v)$ to $high(F_v)$ both have length ≥ 2. Let $G^{*'}$ be the graph obtained from G^* by removing the sink node t^* and its incident edges. (In Figure 13.4, t^* is the node represented by the square labeled by 11.) This merges the faces F_{v_1}, F_{v_2} and F_{v_n} of G^* into one face F'. Note that for every face $F \neq F_{v_{n-1}}$ of $G^{*'}$, the two directed paths of F between $low(F)$ and $high(F)$ in $G^{*'}$ have length ≥ 2.

Let $s^{*'}$ be the source of $G^{*'}$ and let $t^{*'}$ be the sink of $G^{*'}$. Notice that $s^{*'} = s^* = low(F')$ and $t^{*'} = left((v_2, v_n)) = high(F')$. (In Figure 13.4, $t^{*'}$ is the node represented by the square labeled by 10.) Clearly, there are at least two edges e with $F_{v_{n-1}} = left(e)$, and the only edge e with $right(e) = F_{v_{n-1}}$ has endpoint $t^{*'}$. Let P_{long} be any longest path from $s^{*'}$ to $t^{*'}$. Then the length of any longest path from s^* to t^* in G^* is 1 plus the length of P_{long}.

We claim that P_{long} has at most one consecutive sequence of edges in common with any face F of $G^{*'}$. Toward a contradiction assume the claim is not true. Suppose that P_{long} visits some nodes of F, assume that w_1 is the last one, then $l \geq 1$ nodes $u_1, \ldots, u_l \notin F$, then some nodes of F again, let w_d be the first one. Let $w_2, \ldots, w_{d-1}$ be the nodes, in this order, of F, which are not visited by P_{long} (see Figure 13.5.) Suppose $F = right((w_1, w_2))$. (If $F = left((w_1, w_2))$, the proof is similar.) Let $F_1 = left((w_1, w_2))$. Notice that $w_1 = low(F_1)$. The directed path of F_1, starting with edge (w_1, w_2), has length ≥ 2. Hence w_2 has an outgoing edge to a node of F_1, and an outgoing edge to w_3. Thus $w_2 = low(F_2)$, with $F_2 = left((w_2, w_3))$. Repeating this argument it follows that $w_{d-1} = low(F_{d-1})$, with $F_{d-1} = left((w_{d-1}, w_d))$. However it is easy to see that $w_d = high(F_{d-1})$. This means that one of the two directed paths of F_{d-1} has length 1. This contradiction proves the claim.

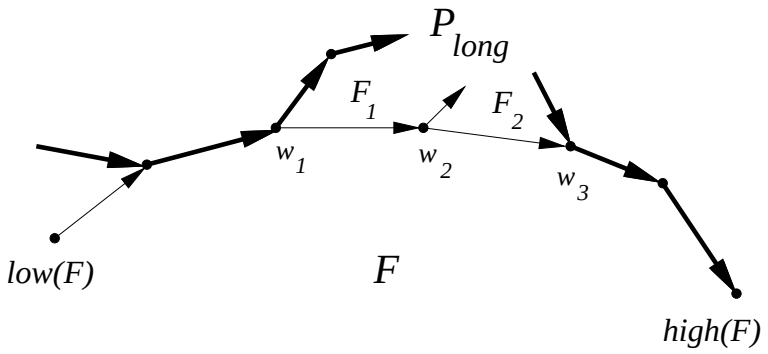


Figure 13.5: Example of the proof of Theorem 5.1.

When traversing an edge e of P_{long}, we visit either $left(e)$ or $right(e)$ (or both) for the first time. We *assign* each edge e to the face F, with $e \in F$, which we visit for the first time now. $G^{*'}$ has $n-2$ faces. To every face F of $G^{*'}$, by the claim, at

most one edge $e \in P_{long}$ is assigned. Hence the longest path from s^* to t^* in G^* has length $\leq n-1$. □

VISIBILITY(G) can be applied to a general 4-connected planar graph by first triangulating it. (The triangulation of a 4-connected planar graph is clearly still 4-connected.) Since the worst-case bounds for visibility representation by applying an arbitrary st-numbering is $(2n-5) \times (n-1)$ [96, 104], our algorithm reduces the width of the visibility representation by a factor 2 in the case of 4-connected planar graphs. In Chapter 14 we show that this algorithm can be used to construct more compact visibility representations of general planar graphs.

Chapter 14

A More Compact Visibility Representation

14.1 Introduction

In this chapter we consider the problem of drawing a general compact visibility representatation. As defined in Chapter 9, in a visibility representation every vertex is mapped to a horizontal segment, and every edge is mapped to a vertical line, only touching the two vertex segments of its endpoints. It is clear that this leads to a nice and readable picture, and it therefore gains a lot of interest (see also Section 9.3.1 and 9.3.2 and Figure 14.1).

We show that every planar graph can be represented by a visibility representation on a grid of size at most $(\lfloor \frac{3}{2}n \rfloor - 3) \times (n-1)$. This improves all previous bounds considerably. An outline of the algorithm to achieve this is as follows. Assume the input graph G is triangulated (otherwise apply a triangulation algorithm, described in Chapter 6). Then we split G into its 4-connected components, and construct the 4-block tree of G. We show that we can do this in linear time for triangulated planar graphs, thereby improving the $O(n \cdot \alpha(m,n) + m)$ time algorithm of Kanevsky et al. [63] for this special case. To each 4-connected component we compute the canonical ordering, as presented in Theorem 13.3.1, leading to a visibility representation of that component on a grid of size at most $(n-1) \times (n-1)$ (Theorem 13.4.1). The representations of the 4-connected components are combined into one entire drawing, leading to the desired width.

To this end, the following lemma is important.

Lemma 14.1.1 *Let $v_1, v_2, \ldots, v_n$ be a canonical 4-ordering of a 4-connected triangular planar graph G, such that v_{n-1} is a neighbor of both v_1 and v_n. Then the numbering $u_i = v_{n-i+1}$ (with $1 \leq i \leq n$) is also a correct canonical 4-ordering $u_1, \ldots, u_n$ of G.*

Proof: By 4-connectivity of G, v_1, v_{n-1} and v_n form one face in G, hence the vertices u_1, u_2 and u_n are forming one face. Every vertex u_i $(2 < i < n-1)$ has at

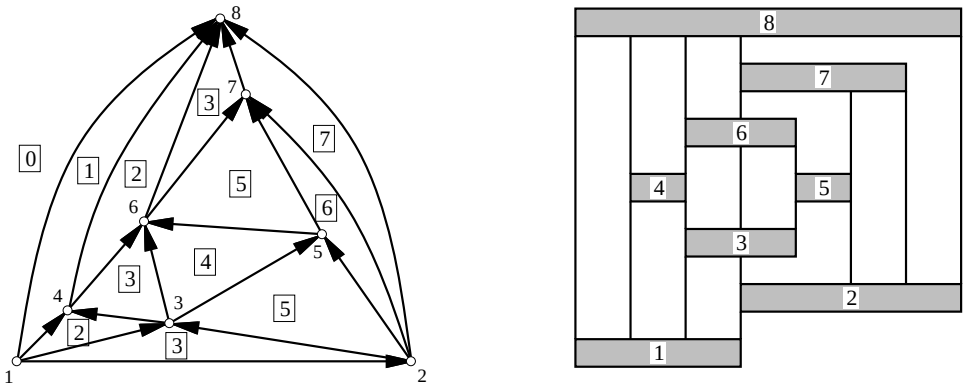


Figure 14.1: The canonical 4-ordering and corresponding visibility representation.

least 2 incoming and 2 outgoing edges, since vertex v_{n-i+1} has at least 2 outgoing and 2 incoming edges. From this observation also the 2-connectivity of G_i, the induced subgraph on $u_1, \ldots, u_i$, follows, which completes the proof. □

A canonical ordering of a 4-connected triangular planar graph G such that v_1, v_{n-1} and v_n are forming one face can be constructed in linear time. In the remaining part of this chapter, , (G) denotes the drawing, obtained by applying VISIBILITY(G). $y(v)$ denotes the y-coordinate of the segment of vertex v, and $x(u, v)$ denotes the x-coordinate of edge (u, v) in , (G). Notice that $x(v_1, v_n) < x(v_1, v_2) < x(v_2, v_n)$ in , (G). The size of the drawing is the size of the smallest rectangle with sides parallel to the x- and y-axis that covers the drawing.

14.2 A General Compact Visibility Representation

In this section we show how we can construct a visibility representation of a planar graph G on a grid, yielding a width smaller than $2n - 5$. We assume that G is triangulated (otherwise apply an arbitrary linear time and space triangulation algorithm (see Chapter 6). To apply the result of Theorem 2.3 we split G into its 4-connected components. Since G is triangulated, a 4-connected component of G is a 4-connected triangulated planar subgraph of G. From this we construct the *4-block tree* T of G: every 4-connected component G_b of G is represented by a node b in T. There is an edge between two nodes b and b' in T, if the separating triplet belongs to both G_b and $G_{b'}$. By planarity every triplet of three vertices is a separating triangle and belongs to precisely two 4-connected components. The separating triplet is an interior face in G_b and the exterior face of $G_{b'}$. We show in Section 4 that T can be computed in linear time and space for triangulated planar graphs. See Figure 14.2

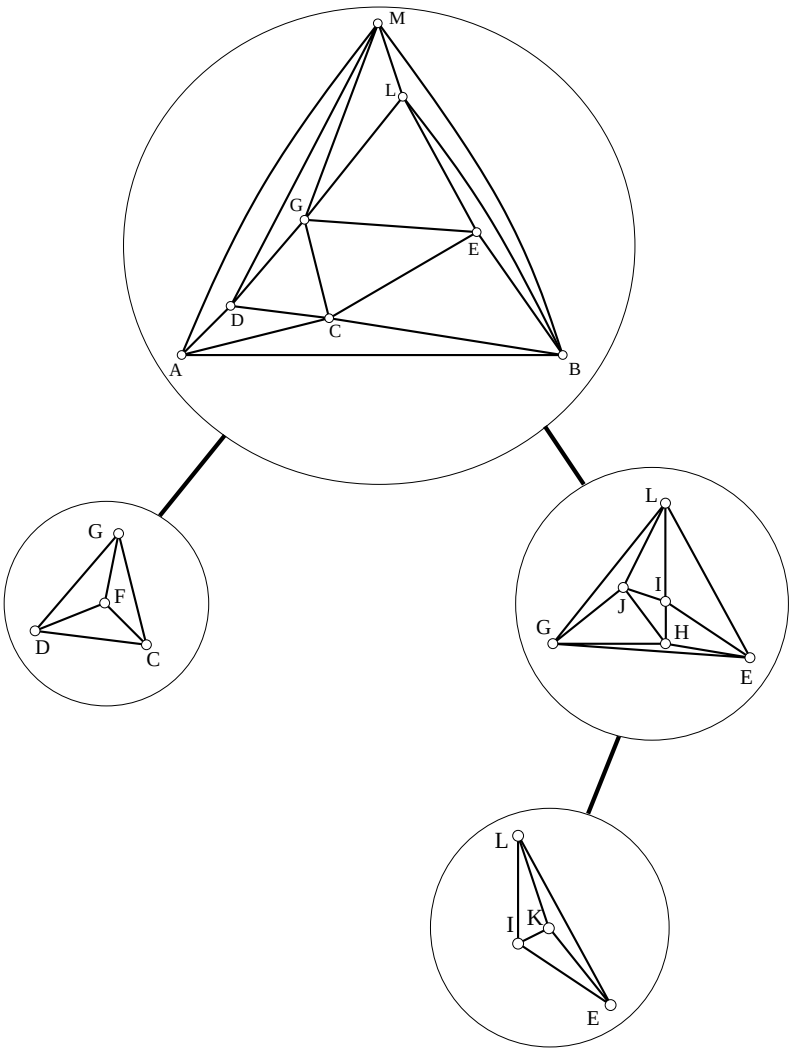


Figure 14.2: The 4-block tree of the graph in Figure 1.

for the 4-block tree of the graph in Figure 1.

We root T at an arbitrary node b. We compute a canonical ordering for G_b, as defined in Theorem 2.1, and direct the edges accordingly. In the algorithm, we traverse T top-down and visit the corresponding 4-connected components. Let b' be a child of b in T, and let $V(G_{b'})$ denote the set of vertices of $G_{b'}$. Let u, v, w be the separating triplet (triangle) of $G_{b'}$. Assume the edges are directed $u \rightarrow v$ and $v \rightarrow w$ in G_b. We define $c(G_{b'}) = v$. Using Lemma 14.1.1, we have two possibilities for computing a canonical ordering in $G_{b'}$ (let $n' = |V(G_{b'})|$):

NORMAL($G_{b'}$) We set $u = v_1, v = v_2$ and $w = v_{n'}$ and compute a canonical ordering $v_1, v_2, \ldots, v_n$ for $G_{b'}$,

REVERSE($G_{b'}$) We set $u = u_{n'}, v = u_2$ and $w = u_1$, and compute the canonical ordering $u_1, u_2, \ldots, u_{n'}$ to $G_{b'}$. Then the ordering is reversed: we set $v_i := u_{n'-i+1}$, for all i with $1 \leq i \leq n'$.

In NORMAL($G_{b'}$), v has number v_2, in REVERSE($G_{b'}$), v has number $v_{n'-1}$. In both orderings, $u = v_1$ and $w = v_{n'}$. See also Figure 14.3.

Both numberings will be applied in the algorithm to achieve a more compact visibility representation. We introduce a label $l(v)$ for each vertex v, which can have the value *up*, *down* or *unmarked*. If $l(v) = unmarked$, then v is called unmarked, otherwise v is called marked. Assume we visit node b' in T, and we have to compute a canonical ordering for $G_{b'}$. Let $v = c(G_{b'})$. The value of $l(v)$ implies whether we use NORMAL($G_{b'}$) or REVERSE($G_{b'}$): if $l(v) = up$, we apply NORMAL($G_{b'}$), if $l(v) = down$, we apply REVERSE($G_{b'}$), otherwise we can do both. We will show later that using these marks, the increase of the width when drawing $G_{b'}$ "inside" G_b is at most $n' - 3$ instead of $n' - 2$, when $l(v) = up$ or *down* (b the parent-node of b' in T).

This method is applied to all 4-connected components of G. After directing the edges, this yields a directed acyclic graph, and applying a topological ordering yields an st-numbering of the vertices. Applying the algorithm VISIBILITY(G) now gives the entire drawing. The complete algorithm can now be described more precisely as follows:

COMPACTVISIBILITY (input: a planar graph G)
  triangulate G;
  construct the 4-block tree T of G, and root T at arbitrary node b;
  compute the canonical 4-ordering for G_b and direct the edges of G_b;
  let $n' = |V(G_b)|$; $l(v_2) = down$, $l(v_{n'-1}) := up$;
  $l(v_i) := unmarked$ for all $v_i \in G_b, i \neq 2, i \neq n' - 1$;
  for every child b' of b **do** DRAWCOMPONENT($G_{b'}$) **rof**;
  compute an st-numbering in the directed graph G;
  apply VISIBILITY to G;
END COMPACTVISIBILITY

procedure DRAWCOMPONENT(G');
  begin
    Case $l(c(G'))$ **of**
      unmarked : NORMAL(G'); $l(c(G')) := down$; $l(v_{|V(G')|-1}) := up$;
      up : NORMAL(G'); $l(c(G')) := unmarked$; $l(v_{|V(G')|-1}) := up$;
      down : REVERSE(G'); $l(c(G')) := unmarked$; $l(v_2) := down$;
    for every $v_i \in G'$ with $2 < i < |V(G')| - 1$ **do** set $l(v_i) := unmarked$ **rof**;
    direct the edges of G' $v_i \to v_j$ iff $j > i$;
    for every child b'' of current node b' in T **do** DRAWCOMPONENT($G_{b''}$) **rof**;
  end;

Since G is triangulated, the following lemma can easily be verified:

Lemma 14.2.1 *Let G be a triangular planar graph, and let $u = rightvertex(v)$ and $w = rightup(v)$. Setting $x(u,v) := x(v,w) := \max\{x(u,v), x(v,w)\}$ does not increase the width of , (G).*

Let now b' be a (non-root) node in T with parent b in T. Let G' be the subgraph of G, consisting of all visited 4-connected components in CompactVisibility. Let u, v, w be the outerface of $G_{b'}$, with $u \to v$ and $v \to w$ in G_b. The following lemma follows ($n' = |V(G_{b'})|$) :

Lemma 14.2.2 *If $x(u,w) < x(u,v) < x(v,w)$ in , (G'), then applying* Normal$(G_{b'})$ *has the result that the width of , $(G' \cup G_{b'})$ is at most the width of , $(G') + n' - 3$.*

Proof: In , $(G_{b'})$, $x(u,v) - x(u,w) \leq n' - 2$ and $x(v,w) - x(u,w) \leq n' - 1$ by Theorem 13.4.1. In , (G'), $x(u,v) - x(u,w) \geq 1$ and $x(v,w) - x(u,w) \geq 2$. Hence the width of , $(G' \cup G_{b'})$ is at most the width of , $(G') + (n' - 1) - 2$. □

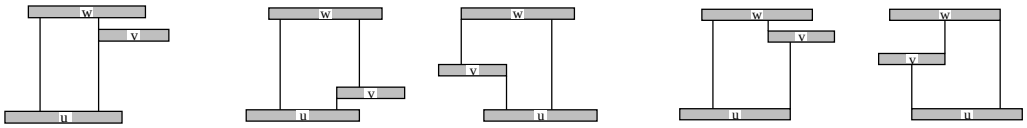


Figure 14.3: The shape of the faces with respect to $l(v)$, Normal$(G_{b'})$ and Reverse$(G_{b'})$.

The same can be proved for $x(v,w) < x(u,v) < x(u,w)$ and applying Normal, or when $x(u,w) < x(v,w) < x(u,v)$ or $x(u,v) < x(v,w) < x(u,w)$ and applying Reverse. See Figure 14.3 for an illustration of this. Assume now that G', b and b' are defined as in the previous lemma. The following lemma can now be proved. Let $v = c(G_{b'})$.

Lemma 14.2.3 *If v is marked, then after applying* Normal$(G_{b'})$ *if $l(v) = up$ and* Reverse$(G_{b'})$ *if $l(v) = down$, the width of , $(G' \cup G_{b'})$ is at most the width of , $(G') + n' - 3$,*

Proof: Let u, v, w be the separating triplet of $G_{b'}$, with $u \to v$ and $v \to w$ in G_b, thus $v = c(G_{b'})$. If $out(v) = 1$ in G_b, then $l(v) = up$. Hence either $x(v,w) < x(u,v) < x(u,w)$ in , (G') or we can change , (G') without increasing the width (by Lemma 14.2.1) such that $x(u,w) < x(u,v) < x(v,w)$ in , (G'). Applying Lemma 14.2.2 yields the desired result. The same follows when $in(v) = 1$ in G_b, thus when $l(v) = down$.

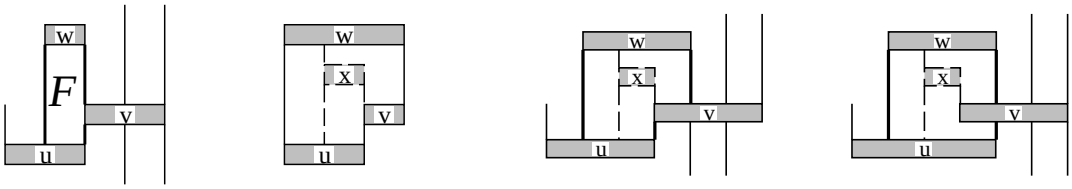
The remaining case is when v was $c(G_{b''})$ for some $b'' \neq b'$, and at the moment of visiting b'', $l(v) = unmarked$. Let the separating triplet of $G_{b''}$ be u', v, w', with

$u' \rightarrow v$ and $v \rightarrow w'$. Let G'' be the subgraph of G, consisting of the visited 4-connected components at the moment of visiting $G_{b''}$. By Lemma 14.2.1 we may assume that $x(u', v) = x(v, w')$ in , (G'').

Consider the case $x(u', w') < x(u', v)$ (the case $x(u', v) < x(u', w')$ goes similar). In CompactVisbility Normal$(G_{b''})$ is applied. Since $in(v) = 1$ in $G_{b''}$, we can set $x(u', v)$ to $x(v, w')$ in , $(G_{b''})$ without increasing the width (see Lemma 14.2.1). This has the result that $x(v,leftup(v)) < x(u', v)$ in , (G'). If $w = leftup(v)$ then the proof is completed by observing that $x(u, w) < x(v, w) < x(u, v)$ in , (G') and applying Lemma 14.2.2.

If $w \neq leftup(v)$, then $w = rightup(v)$ and $u = rightvertex(v)$. By Lemma 14.2.1 we may assume that both $x(u', v) = x(v, w')$ and $x(rightvertex(v), v) = x(v,rightup(v))$ holds in , (G''). But since $x(u', v) < x(v, w')$ holds in , $(G_{b''})$ it directly follows that $x(rightvertex(v), v) < x(v,rightup(v))$ in , $(G'' \cup G_{b''})$. Again the proof is completed by observing that $x(u, v) < x(v, w) < x(u, w)$ in , (G') and applying Lemma 14.2.2. □

See Figure 14.4 for an illustration of the proof of Lemma 14.2.3.



(a) The face and the 4-connected component.

(b) The 2 different possibilities for drawing inside the face.

Figure 14.4: Illustration of Lemma 14.2.3.

Lemma 14.2.4 *The width of the visibility representation of G is at most $\lfloor \frac{3}{2}n \rfloor - 3$.*

Proof: Let $b_1, \ldots, b_p$ be the nodes of T in visiting order. Let K_i be the number of marked vertices after visiting b_i $(1 \leq i \leq p)$. Let K_0 be the initial number of marked vertices. $K_0 = 2$, since initially only v_2 and $v_{n'-1}$ are marked. When we visit G_{b_i}, then vertex v_2 or $v_{n'-1}$ is added to the current graph and is unmarkded. If $l(c(G_{b_i})) \neq unmarked$, then the increase in width is at most $|V(G_{b_i})|-3$ and $l(c(G_{b_i}))$ becomes *unmarked*, i.e., $K_i = K_{i-1}$. If $l(c(G_{b_i})) = unmarked$, then the increase in width is at most $|V(G_{b_i})|-2$ and $l(c(G_{b'}))$ becomes *down*, i.e, $K_i = K_{i-1}+2$. Hence in both cases when visiting G_{b_i}, the width of the drawing increases by at most $|V(G_{b_i})| - 3 + \frac{K_i - K_{i-1}}{2}$. $|V(G_{b'}| - 3$ is also precisely the number of added vertices of $G_{b'}$.

Since the width of , (G_{b_1}) is at most $|V(G_{b_1})|-1$ and K_p is even and $K_p \leq n-2$ (the source and the sink of G never get marked), it follows that the total width of , (G) is at most $n-1+\lfloor\frac{K_p-K_0}{2}\rfloor = n-1+\lfloor\frac{n-2-2}{2}\rfloor \leq \lfloor\frac{3}{2}n\rfloor - 3$. □

Regarding the time complexity we show in Section 4 that the 4-block tree can be computed in linear time. Computing a canonical 4-ordering also requires linear time. We maintain the direction of the edges of the visited 4-connected components, and from this $c(G_{b'})$ can be determined directly in $O(1)$ time. Finally VISIBILITY(G) is applied, which requires linear time [96, 104]. This completes the following theorem.

Theorem 14.2.5 *There is a linear time and space algorithm for computing a visibility representation of a planar graph G on a grid of size at most $(\lfloor\frac{3}{2}n\rfloor-3)\times(n-1)$.*

Consider the graph of Figure 1. In Figure 14.2 the 4-block tree is given. The visibility representation of the root-block is given in Figure 14.1. Drawing the other 4-connected components inside and applying VISIBILITY leads to the drawing as given in Figure 1. Notice that $l(D), l(G)$ and $l(I)$ are *down*, $l(F), l(J)$ and $l(K)$ are *up*, all the other vertices of the graph are unmarked. Hence 6 vertices are marked, and the total width is at most $n-1+\frac{6}{2} = 15$. (The width in the drawing in Figure 1 is 12.)

14.3 Constructing the 4-block tree

In this section we show a method for constructing the 4-block tree of a triangulated planar graph. Since this class of graphs has some special properties, there is no need to use the complicated algorithm of Kanevsky et al., which builds a 4-block tree of a general graph in $O(n \cdot \alpha(m,n)+m)$ time [63]. In our case a separating triplet is a separating triangle, which forms the basis for the algorithm.

For determining the separating triangles, we use the algorithm of Chiba & Nishizeki [11] for determining triangles in a graph. (In [94], Richards describes another linear time algorithm.) Chiba & Nishizeki first sort the vertices in $v_1, \ldots, v_n$ in such a way that $deg(v_1) \geq deg(v_2) \geq \ldots \geq deg(v_n)$. Observing that each triangle containing vertex v_i corresponds to an edge joining two neighbors of v_i, they first mark all vertices u adjacent to v_i (for the current index i). For each marked u and each vertex w adjacent to u they test whether w is marked. If so, a triangle v_i, u, w is listed. After this test is completed for each marked vertex u, they delete v_i from G and repeat the procedure with v_{i+1}. Starting with v_1, this algorithm lists all triangles without duplication in $n-2$ steps.

In our case, we are looking for *separating* triangles. If a triangle is not separating, then it is a face in the triangulated planar graph. To test this, we store the embedding of the original graph also in adjacency lists, say in adjacency lists $adj(v)$ for all $v \in G$. If a triangle u, v, w is not separating, (i.e., a face), then u and w appear consecutively in $adj(v)$, which can be tested in $O(1)$ time by maintaining

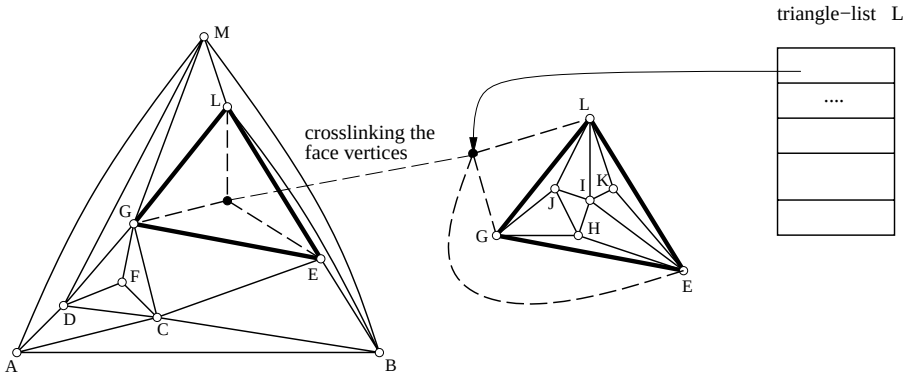


Figure 14.5: The data structure for constructing the 4-block tree.

crosspointers. To compute the time complexity of this algorithm, Chiba & Nishizeki use the *arboricity* of G, defined as the minimum number of edge-disjoint forests into which G can be decomposed, and denoted by $a(G)$ [45].

Lemma 14.3.1 ([11]) $\sum_{(u,v)\in E} \min\{deg(u), deg(v)\} \leq 2 \cdot a(G) \cdot m.$

Using this lemma Chiba & Nishizeki show that the time complexity of the algorithm is $O(a(G) \cdot m)$. If G is planar then $a(G) \leq 3$ ([45]), so the algorithm runs in $O(n)$ time in case G is planar.

To obtain the 4-block tree we introduce now the following data structure: Let L be the list of separating triangles. $L(u, v, w)$ denotes the record in L, containing separating triangle u, v, w. $L(u, v, w)$ contains the edges $(u, v), (v, w)$ and (w, u), and there are crosspointers between L and the edges and vertices in G.

We now want to "sort" the separating triangles, containing edge (u, v). Hereto we do the following: Let $adj(v) = w_0, w_1, \ldots, w_{d-1}$ (in clockwise order around v with respect to a planar embedding). We sort the separating triangles v, w_i, w_j stored at v in order with respect to $w_0, \ldots, w_{d-1}$. If there are separating triangles v, w_i, w_j and v, w_i, w_k then we set a pointer from edge (v, w_i) in $L(v, w_i, w_j)$ to edge (v, w_i) in $L(v, w_i, w_k)$, if $k > j$ (addition modulo d). We do this for every vertex $v \in G$. Of course, when visiting vertex w_i and considering separating triangles v, w_i, w_j and v, w_i, w_k, there is no need to place another directed edge between $L(v, w_i, w_j)$ and $L(v, w_i, w_k)$.

Observe now that when edge (u, v) in $L(u, v, w)$ has no outgoing edge, then this means that when we split G at $(u, v), (v, w), (w, u)$ into two subgraphs, say G_1 and G_2, then all other separating triangles, containing (u, v), belong to either G_1 or G_2. We start at an arbitrary separating triangle in L, say u, v, w, where each edge in $L(u, v, w)$ has either an incoming or an outgoing edge. We split the graph at $(u, v), (v, w)$ and (w, u) into two subgraphs, say G_1 and G_2.

Let $adj(v) = w_0, \ldots, w_{d-1}$ and let $u = w_0$ and $w = w_i$, $0 < i < d$. To obtain G_1 and G_2, we split $adj(v)$ into two adjacency lists, say $adj_1(v)$ and $adj_2(v)$ with $adj_1(v) = w_0, \ldots, w_i$ and $adj_2(v) = w_i, \ldots, w_{d-1}, w_0$ (similar for $adj(u)$ and $adj(w)$). Let $adj_1(v)$ correspond to G_1. If all other separating triangles, containing (u, v), belong to G_1, then we introduce w_0 in $adj_2(v)$. This yields that all other separating triangles in L, containing (u, v), still point to the right edge in the data structure, viz. in the adjacency list of G_1. Testing whether the other separating triangles, containing (u, v), belong to G_1 or G_2 can be tested by checking whether (u, v) in $L(u, v, w)$ has an incoming or an outgoing edge. we introduce w_0 in $adj_1(v)$. Similar is done for the edges (v, w) and (w, u) with respect to $adj(w)$ and $adj(u)$. We mark $L(u, v, w)$ as visited and delete the incoming and outgoing edges of $L(u, v, w)$, and we continue until all separating triangles in L are visited.

To construct the 4-block tree, we apply a simple traversal through the datastructure for determining the connected components. The connections via the face vertices give the connections in the 4-block tree. For every 4-connected component we add pointers to its three vertices on the outerface. The complete algorithm can now be described as follows:

CONSTRUCT 4-BLOCK TREE
  enumerate all separating triangles and store them in L;
  sort the separating triangles and add directed edges in L;
  while not every triangle in L is visited **do**
    Let $L(u, v, w)$ be a record in L such that each edge $(u, v), (v, w)$ and (w, u) in $L(u, v, w)$ has either an incoming or outgoing edge;
    split the graph at edges $(u, v), (u, w), (v, w)$;
    set a pointer between the two corresponding faces;
  od;
  determine the connected components and construct the 4-block tree;
END CONSTRUCT 4-BLOCK TREE

Theorem 14.3.2 *The 4-block tree of a triangulated planar graph can be constructed in linear time and space.*

Proof: Determining and storing the separating cycles requires $O(n)$ time, because every planar graph has at most $n - 4$ separating triangles. Sorting the separating triangles at vertex v can be done in $O(deg(v))$ time by using (double) bucket-sort, since vertex v belongs to at most $deg(v) - 2$ separating triangles. Hence the total sorting time is $O(n)$. By maintaining a sublist of L where we store all separating triangles, which can be visited next, we can find the next separating triangle in $O(1)$ time. Since there are crosspointers between the edges and vertices in G and the corresponding entry in L, we can split the graph at the separating triangle in $O(1)$ time. Determining the connected components and building up the 4-block tree is achieved by a simple traversal through the graph, which completes the proof. □

Chapter 15

Conclusions

In Part C we studied the problem of drawing a planar graph, such that there is no pair of crossing edges and the vertices and bends placed on grid points. To this end we defined a new ordering, called a *canonical ordering*, on the vertices of a triconnected planar graph. Using this ordering, a lot of improvements and new results are obtained for several representation models of planar graphs. To obtain a linear time algorithm we do not need advanced and sophisticated data structures. After computing a planar embedding of the graph, and storing it in the adjacency lists $adj(v)$, we can compute the canonical ordering by only assigning some additional counters to each vertex, like $shift(v)$, $interval(v)$ and $chords(v)$. The various drawing algorithms can also be implemented by using a small number of variables for each vertex, and by maintaining the outerface C_{k-1} in a doubly linked list.

It is the first time that one new characterization of a planar graph leads to such a broad area of applications with respect to drawing requirements. If the edges must be straight lines, then we can draw every triconnected planar graph on a grid of size at most $(n-2) \times (n-2)$ such that every face is convex, i.e., all interior angles have size $\leq \pi$. This problem was open for several years in the graph drawing community. Moreover, this matches the best known grid bounds for drawing a planar graph planar on a grid (see Schnyder [98]). We also used the canonical ordering to construct orthogonal drawings of 3-planar and triconnected 4-planar graphs. In the case of triconnected 4-planar graphs, the number of total bends decreases from $2n+4$ to $\lceil \frac{3n}{2} \rceil + 4$ bends with the property that every edge has at most 2 bends (if $n > 6$). We also proved an existential lower bound of $\frac{4}{3}(n-1)+2$ bends for a class of 4-planar graphs. In the case of 3-planar graphs we presented an algorithm that draws a 3-planar graph orthogonal with at most $\lfloor \frac{n}{2} \rfloor + 1$ bends totally on a grid of size at most $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$. This matches both lower bounds with respect to number of bends and size of the grid. Moreover, we showed that if $n > 4$, then there is a spanning tree, using only straight-line edges, and all remaining edges have at most one bend.

Using a hexagonal grid as drawing model, we showed that every triconnected 3-planar graph can be drawn hexagonal on an $\frac{n}{2} \times \frac{n}{2}$ grid such that one edge has at

most 3 bends, and all other edges have no bends. More important, using this result we showed that every 3-planar graph can be drawn with minimum angle $\geq \frac{\pi}{3}$, if the graph is not triconnected, and with minimum angle $\geq \frac{\pi}{4}$, otherwise. This solves an open problem of Formann et al. [32]. As a side result we showed that every triconnected 6-planar graph can be drawn planar on an $O(n) \times O(n)$ hexagonal grid such that each edge has at most 4 bends. Other applications of the canonical ordering are in the area of rectangular duals and computing visibility representations. For both representations new and fairly simple algorithms are presented. Using the canonical ordering for 4-connected triangular planar graphs, we can construct a visibility representation of a 4-connected planar graph on a grid of size at most $(n-1) \times (n-1)$, thereby decreasing the best known grid size by a factor 2. This result is also used to construct a visibility representation of a planar graph on a grid of size at most $(\lfloor \frac{3}{2}n \rfloor - 3) \times (n-1)$ grid, improving the best known grid size of $(2n-4) \times (n-1)$ considerably. This algorithm also uses a new result, namely that a 4-block tree of a triangulated planar graph can be constructed in linear time and space.

Core areas in the field of graph algorithms are the parallel and dynamic algorithms. Let us be more precise about this field with respect to the problems described in this thesis. Very recently, He announced an $O(\log^4 n)$ algorithm for computing a canonical ordering for triconnected planar graphs [48]. This gives a parallel algorithm for several drawing algorithms, described in Part C. In particular, given a canonical ordering, He proved that a convex planar drawing on an $(2n-2) \times (n-1)$ can be obtained in $O(\log n)$ time [48]. Jou, Suzuki & Nishizeki [61] presented an $O(\log n \log^* n)$ time parallel algorithm for the hexagonal grid drawing algorithm for triconnected planar graphs, presented in Chapter 12. This algorithm is based on the $O(\log n \log^* n)$ parallel algorithm for Schnyder's grid drawing algorithm, described by Fürer et al. [37]. In [46], He describes an $O(\log n)$ parallel algorithm for a variant of the rectangular dual algorithm, described in Chapter 13. The parallel computation model in [37] is the EREW PRAM with $O(n)$ processors. In the other algorithms, described here, the parallel computation model is the CRCW PRAM with $O(n)$ processors. In [16], Cohen et al. describe various dynamic algorithms for updating the figure after adding (or deleting) an edge or vertex. Unfortunately, this framework seems not to work for our drawing algorithms.

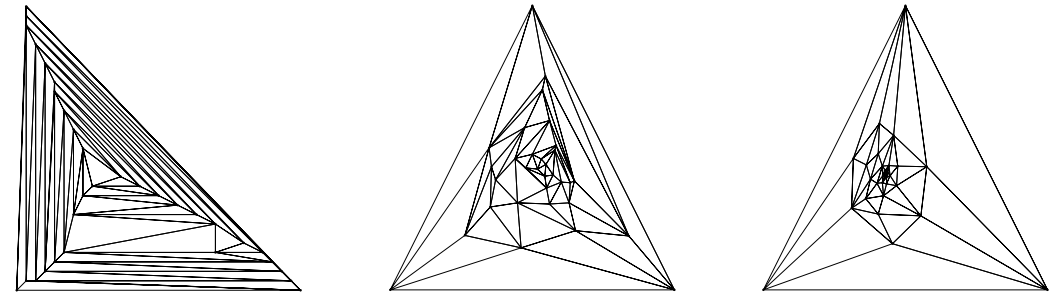
We believe that more combinatorial observations with respect to the *lmc*-ordering can be given than those, given in this thesis. Very recently, a new characterization of planar graphs has been made, based on the *st*-ordering of biconnected planar graphs [95]. The generalization to triconnected planar graphs, using the *lmc*-ordering, may yield new combinatorial and practical results for planar graph drawings.

Let us end Part C with some words about the more practical aspects of the introduced graph drawing problems. In this context the aesthetic aspects and readability is more important than the theoretical derived bounds on grid size, angles, number of bends, and other criteria. A deep investigation of drawing triangular planar graphs planar with vertices represented by points and edges by straight-line edges (also alled a *Fáry drawing*) is presented by Jones et al. [60]. They study the

algorithm of Chiba et al. [14], de Fraysseix et al. [34], Read [92] and Tutte [111]. They used the following metrics to compare the different drawings:

- the standard deviation in angle size,
- the standard deviation in edge length as a percentage of the mean edge length,
- and the standard deviation in face area as a percentage of the mean face area.

They concluded that the algorithm of de Fraysseix et al. [34] is undoubtably the best of the four examined algorithms. The most important disadvantage of the other algorithms is the fact of *clustering*: in a relative small area too many vertices are placed. This leads to very short edge lengths, small face areas and a crowded and messed drawing. The algorithm of de Fraysseix et al. (and its linear time implementation, due to Chrobak & Payne [15]) does not have this problem, since vertices are placed on grid points. However, the problem of this algorithm is that the size of the angles can become very small, which also makes the drawing unattractive. See Figure 15.1 for the output of applying the algorithm of de Fraysseix et al., Read and Tutte on a triangular planar graph with 32 vertices. To inspect these



(a) ConvexSmallGrid-Drawing (b) Modified Read (c) Tutte

Figure 15.1: Drawing a triangular planar graph.

problems, we implemented our $(n-2) \times (n-2)$ grid convex drawing algorithm ConvexSmallGridDrawing(G), as presented in Chapter 10. We also modified the algorithm of Read [92], presented in Chapter 10: we use a sophisticated way for chosing the next vertex v to delete, and when placing v back, we try to optimize the involved angle sizes and edge lengths. More precisely, instead of deleting one vertex in every step, we remove an independent set of vertices from the drawing. Since a planar graph can be colored easily with 6 colors in linear time, this yields a simple method to extract an independent set with size at least $n/6$. The idea is that when the independent set is more or less distributed over G, then the probability decreases that vertices are clustered in the entire drawing. The algorithm can be described as follows (let $N(v)$ be the set of neighbors of vertex v):

```
ModifiedRead(G);
  compute a planar embedding of the triangular planar graph G;
  let u, v, w be the vertices on the outerface of G;
  n' := n;
  while n' > 3 do
    compute an independent set I in G, with u, v, w ∉ I;
    let v_{n'-|I|+1}, v_{n'-|I|+2}, ..., v_{n'} := vertices of I;
    for i := n' - |I| + 1 to n' do
      let v'_i be a neighbor of v_i in G, with |N(v'_i) ∩ N(v_i)| = 2;
      delete v_i from G and add edges from v'_i to neighbors of v_i;
    od;
    n' := n' - |I|;
  od;
  place the three vertices v_1, v_2, v_3 as a triangle;
  for i := 4 to n do OptimalPlacement(v_i) rof;
End ModifiedRead
```

The remaining point is to compute the optimal place of v_i with respect to its neighbors. What is the best place for vertex v_i? Consider for this face F_i, the face in which v_i has to be placed (thus after removing the added edges from v'_i). Then F is a *star-shaped polygon*, that is, there is an nonempty area inside F from which all vertices from F_i are visible. This nonempty area is called the *kernel* of F. Let $u_1, u_2, \ldots, u_d$ be the neighbors of v_i.

For placing v_i, the edgelength of (u_j, v_i), $(1 \leq j \leq d)$, and the size of the incident angles of v_i and u_j inside F_i are important. This problem has a lot to do with several linear programming problems in computational geometry. Matoušek, Sharir & Welzl [81] considered the problem of computing a point p_i inside the kernel of F_i, such that after connecting p_i to all the vertices of F_i by straight-line edges, the minimal angle between two adjacent edges is maximized. They called this problem the *angle-optimal placement of point in polygon* and showed that using generalized linear programming, this problem can be solved in linear time [81].

In our model, we want to deal with both the minimum angle and the minimum length of the incident edges of v_i. Therefore we introduce two variables: an angle α and a distance δ. We place v_i such that the minimum angle inside F_i is greater than α and each incident edge of v_i has length greater than δ. Using binary search, the optimal values of the parameters α and δ can be achieved. Computing the area is done by determining the set of halfspaces, to which vertex v must belong, and to inspect whether its intersection is empty or not. See Preparato & Shamos [91] for a description of this method. A triangular planar graph has $3n - 6$ edges, hence the average degree of every vertex is 6. This yields an $O(1)$ work amortized to compute the optimal place of v_i. Hence the total algorithm can be implemented to run in $O(n \log n)$ time.

We compared the algorithm SmallGridConvex with ModifiedRead with

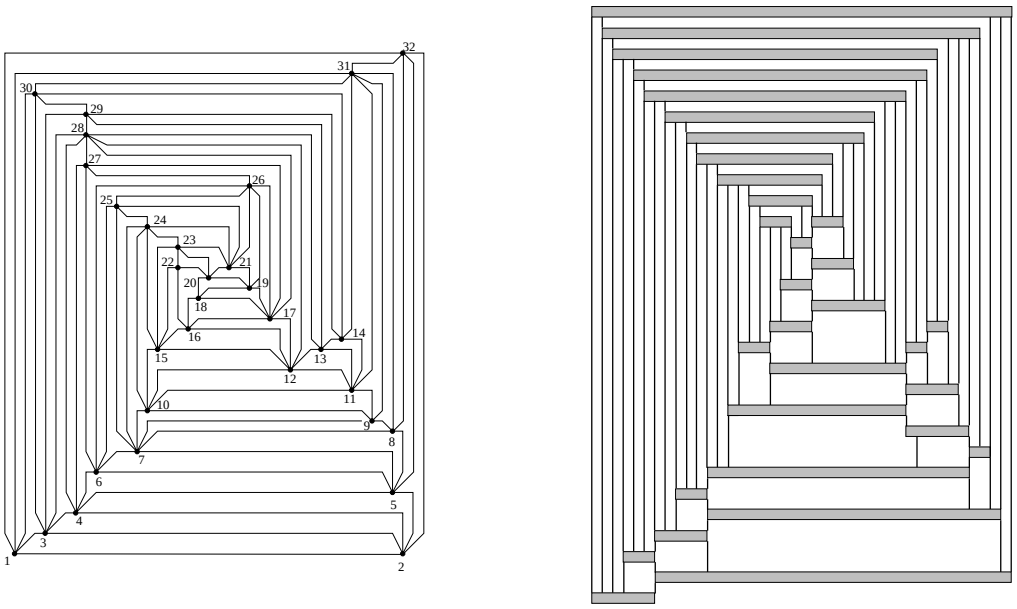
respect to length of the edges, the sizes of the angles and the area of the faces. This leads to the table, as given in Figure 15.2.

			vertices									
		Alg.	50	100	150	200	250	300	350	400	450	500
Edges	$\overline{\lvert e \rvert}$	Read	22.39	17.58	15.10	14.87	13.79	12.34	12.78	13.42	12.24	11.26
		Convex	33.71	25.89	25.85	22.26	21.72	20.28	17.99	17.04	19.75	18.30
	σ	Read	19.30	17.94	17.08	17.76	16.78	15.84	18.23	18.61	17.77	16.15
		Convex	30.06	26.66	28.39	25.95	27.01	26.52	23.78	22.46	26.51	24.94
	$\leq \frac{100}{n}$	Read	27.00	64.20	110.20	197.40	284.00	382.20	475.00	476.80	576.20	673.00
		Convex	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	19.60
	$\leq \frac{200}{n}$	Read	39.60	75.60	121.80	209.40	292.60	392.20	486.00	491.20	590.00	695.40
		Convex	33.40	65.80	92.20	121.40	148.80	189.00	214.60	239.40	277.40	301.40
Angles	$\overline{\alpha}$	Read	2.63	3.01	3.21	3.85	4.09	4.33	4.48	4.27	4.44	4.45
		Convex	1.24	1.23	1.23	1.23	1.23	1.23	1.23	1.23	1.24	1.23
	σ	Read	2.16	2.21	2.34	2.47	2.47	2.43	2.49	2.52	2.46	2.55
		Convex	1.04	1.06	1.07	1.08	1.07	1.08	1.08	1.08	1.09	1.08
	$\leq \frac{5}{n}$	Read	28.80	38.20	48.80	41.80	38.40	42.60	42.20	63.20	51.40	63.20
		Convex	78.80	127.40	191.20	236.60	274.80	312.60	333.40	391.00	465.40	478.60
	$\leq \frac{10}{n}$	Read	54.40	70.60	85.40	76.40	68.20	74.40	102.40	99.20	89.80	119.80
		Convex	108.20	171.60	240.80	291.60	346.00	394.60	422.00	489.80	551.60	584.80
Faces	$\overline{\lvert F \rvert}$	Read	67.03	43.63	24.59	19.93	17.59	15.79	11.76	13.31	10.14	8.16
		Convex	62.75	28.29	20.26	14.47	11.53	9.87	8.51	7.47	6.74	5.70
	σ	Read	72.44	63.19	47.70	40.72	48.13	38.36	37.03	39.81	41.84	31.16
		Convex	76.21	52.98	44.70	45.73	36.78	34.74	25.51	34.16	15.83	22.48
	$\leq \frac{25}{n}$	Read	34.80	86.40	158.80	283.00	359.40	457.60	593.60	642.20	753.60	842.20
		Convex	0.00	0.00	0.00	0.40	58.60	67.40	77.40	89.60	210.60	224.20
	$\leq \frac{50}{n}$	Read	35.00	88.60	163.60	288.80	365.20	461.40	603.40	647.80	763.80	855.60
		Convex	0.00	0.80	36.60	49.20	111.60	137.20	225.20	245.60	350.80	370.80

Figure 15.2: Output of the different drawing algorithms.

$|\overline{e}|$ means the average length of the edges; σ denotes the standard deviation, $\leq \frac{100}{n}$ defines the number of edges, with length at most $\frac{100}{n}$, etc. Hence in the triangular planar graphs with 350 vertices there were on average 189 edges with length at most $\frac{200}{n}$. Notice that in the convex drawing algorithm, there are only a few short edges (with length at most $\frac{100}{n}$), there are a lot of small angles (size at most $\frac{5}{n}$), and there are only a few faces with small area (area at most $\frac{25}{n}$).

The experimental results assent the conclusions of Jones et al. [60], stated above (see Figure 15.1): the modified algorithm of Read has the big problem of clustering, and the algorithm ConvexSmallGridDrawing has the problem of introducing a lot of angles of small size. More important, we state (as well as Jones et al. did) that the straight-line drawing is not the ideal model for representing triangular planar graphs. The resulting pictures do not satisfy the aesthetic criteria for obtaining a readable drawing, as stated in Chapter 1 (see also Figure 15.1). In all investigated cases we observed problems with respect to clustering, face area, angle size and edge length. We believe that the d-planar drawing algorithm, described in Section 10.3 and the visibility representation, described in Section 10.4 give a more convenient insight in the structure of the planar graph. The drawings of the triangular planar graph of Figure 15.1, using this representation models, are given in Figure 15.3.



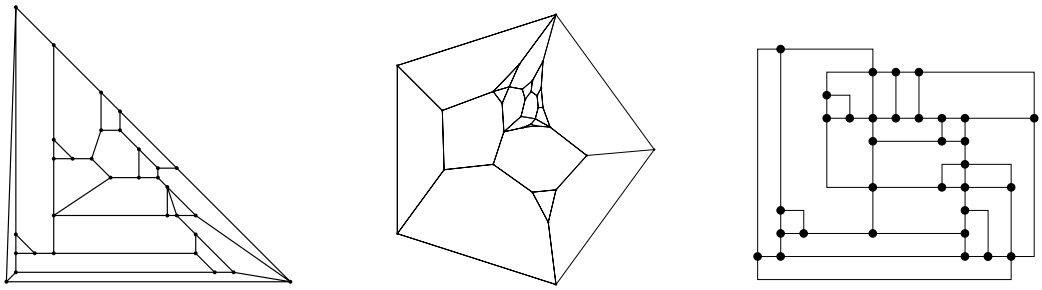
(a) Drawing G as a d-planar graph.

(b) Drawing G as a visibility representation.

Figure 15.3: Drawing a triangular planar graph.

When the input graph is triconnected and has a relative small number of edges, then both the convex drawing algorithm of Tutte (see Chapter 10) and ConvexSmallGridDrawing give somewhat more readable pictures. (Since the graph is not triangular, the algorithm of Read can not be used now.) In Figure 15.4 the output is given when applying these algorithms on a triconnected planar graph with 32 vertices. In this setting, the algorithm ConvexSmallGridDrawing has the advantage that the vertices are placed on grid coordinates. The output of Tutte's algorithm contains several faces with relatively small area, but here all interior angles have size smaller than π and, hence, are strictly convex. Since this graph has maximum degree 4, we can also represent this graph by an orthogonal drawing, as presented in Chapter 11. Though this algorithm might imply bends in the edges, every edge has length at least 1, every face has area at least 1, and the size of the angles is at least $\pi/2$. The required area is at most $n \times n$. Hence this representation is a good candidate when the planar graph has maximum degree 4.

We also notice that several algorithms for straight-line drawings, based on local replacements or simulated annealing, are described in the literature. Though the obtained drawings are more beautiful than several models presented here, the corresponding price with respect to the required time is very high. Hence before chosing



(a) ConvexSmallGrid-Drawing (b) Tutte (c) Orthogonal

Figure 15.4: Drawing a triconnected planar graph.

a drawing algorithm, the question how much time the algorithm may use should be answered. It is clear that if the drawings must be given immediately, the presented algorithms outperforms the existing corresponding algorithms with respect to time and delivered results. We hope that the presented algorithms help you to represent different kinds of networks and diagrams in a convenient way....

Bibliography

[1] Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publ. Comp., Reading, Mass., 1974.

[2] Auslander, L., and S.V. Parter, On embedding graphs in the plane, *J. Math. Mech.* 11 (1961), pp. 517–523.

[3] D. Barnette. Trees in polyhedral graphs. *Canad. J. Math.* (18):731–736, 1966.

[4] Bertolazzi, P., and G. Di Battista, On upward drawing testing of triconnected digraphs, in: *Proc. 7th Annual ACM Symp. on Computational Geometry*, North Conway, 1991, pp. 272–280.

[5] Bertolazzi, P., G. Di Battista, C. Mannino and R. Tamassia, *Optimal Upward Planarity Testing of Single-Source Digraphs*, Manuscript, Dip. di Informatica e Sistemistica, Università degli Studi di Roma La Sapienza, 1992.

[6] Bhasker, J., and S. Sahni, A linear algorithm to check for the existence of a rectangular dual of a planar triangulated graph, *Networks* 7 (1987), pp. 307-317.

[7] Bhasker, J., and S. Sahni, A linear algorithm to find a rectangular dual of a planar triangulated graph, *Algorithmica* 3 (1988), pp. 147–178.

[8] Biedl, T., and G. Kant, A better heuristic for planar orthogonal drawings, in: *Proc. 2nd European Symposium on Algorithms (ESA'94)*, Lecture Notes in Comp. Science 855, Springer-Verlag, 1994, pp. 24–36.

[9] Booth, K.S., and G.S. Lueker, Testing for the consecutive ones property, interval graphs and graph planarity testing using PQ-tree algorithms, *J. of Computer and System Sciences* 13 (1976), pp. 335–379.

[10] Cai, J., X. Han and R.E. Tarjan, *New Solutions to Four Planar Graph Problems*, Tech. Report, Dept. of Computer Science, New York University/Courant Institute, 1989.

[11] Chiba, N., and T. Nishizeki, Arboricity and subgraph listing algorithms, *SIAM J. Comput.* 14 (1985), pp. 210–223.

[12] Chiba, N., T. Nishizeki, S. Abe and T. Ozawa, A linear algorithm for embedding planar graphs using PQ-trees, *J. of Computer and System Sciences* 30 (1985), pp. 54–76.

[13] Chiba, N., K. Onoguchi, and T. Nishizeki, Drawing plane graphs nicely, *Acta Informatica* 22 (1985), pp. 187–201.

[14] Chiba, N., T. Yamanouchi, and T. Nishizeki, Linear algorithms for convex drawings of planar graphs, in: J.A. Bondy and U.S.R. Murty (Eds.), *Progress in Graph Theory*, Academic Press, New York, 1984, pp. 153–173.

[15] Chrobak, M., and T.H. Payne, *A Linear Time Algorithm for Drawing Planar Graphs on the Grid*, Tech. Rep. UCR-CS-90-2, Dept. of Math. and Comp. Science, University of California at Riverside, 1990.

[16] Cohen, R.F., G. Di Battista, R. Tamassia, I.G. Tollis and P. Bertolazzi, A framework for dynamic graph drawing, in: *Proc. 8th Annual ACM Symp. on Computational Geometry*, Berlin, 1992, pp. 261–270.

[17] Cormen, T.H., C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990.

[18] Di Battista, G., Eades, P., and R. Tamassia, I.G. Tollis, *Algorithms for Automatic Graph Drawing: An Annotated Bibliography*, Tech. Report, Dept. of Comp. Science, Brown Univ., 1993.

[19] Di Battista, G., G. Liotta and F. Vargiu, Spirality of orthogonal representations and optimal drawings of series-parallel graphs and 3-planar graphs, in: *Proc. 3nd Workshop on Algorithms and Data Structures*, Lecture Notes in Comp. Science, Springer-Verlag, 1993, to appear.

[20] Di Battista, G., and R. Tamassia, Algorithms for plane representations of acyclic digraphs, *Theoret. Comp. Science* 61 (1988), pp. 436–441.

[21] Di Battista, G., and R. Tamassia, Incremental planarity testing, in: *Proc. 30th Annual IEEE Symp. on Found. of Comp. Science*, North Carolina, 1989, pp. 436–441.

[22] Di Battista, G., R. Tamassia and I.G. Tollis, Area requirement and symmetry display in drawing graphs, *Discrete and Comp. Geometry* 7 (1992), pp. 381–401.

[23] Di Battista, G., R. Tamassia and I.G. Tollis, Constrained visibility representations of graphs, *Inf. Proc. Letters* 41 (1992), pp. 1–7.

[24] Edelsbrunner, H., *Triangulations*, Tech. Report CS 497, Dept. of Comp. Science, University of Illinois at Urbana Champaign, 1991.

[25] Edelsbrunner, H., and T.S. Tan, A quadratic time algorithm for the minmax length triangulation, in: *Proc. 32th Annual IEEE Symp. on Found. of Comp. Science*, Puerto Rico, 1991, pp. 548–559.

[26] Edelsbrunner, H., T.S. Tan and R. Waupopitsch, An $O(n^2 \log n)$ time algorithm for the minmax angle triangulation, in: *Proc. 6th Annual ACM Symp. on Computational Geometry*, Berkeley, 1990, pp. 44-52.

[27] Eswaran, K.P., and R.E. Tarjan, Augmentation problems, *SIAM J. Comput.* 5 (1976), pp. 653–665.

[28] Even, S., *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[29] S. Even, and G. Granot, *Rectilinear Planar Drawings with Few Bends in Each Edge*, Manuscript, Faculty of Comp. Science, the Technion, Haifa (Israel), 1993.

[30] Even, S., and R.E. Tarjan, Computing an st-numbering, *Theoret. Comp. Science* 2 (1976), pp. 436–441.

[31] Fáry, I., On straight lines representations of planar graphs, *Acta Sci. Math. Szeged* 11 (1948), pp. 229–233.

[32] Formann, M., T. Hagerup, J. Haralambides, M. Kaufmann, F.T. Leighton, A. Simvonis, E. Welzl and G. Woeginger, Drawing graphs in the plane with high resolution, in: *Proc. 31th Annual IEEE Symp. on Found. of Comp. Science*, St. Louis, 1990, pp. 86–95.

[33] Frank, A., Augmenting graphs to meet edge-connectivity requirements, in: *Proc. 31th Annual IEEE Symp. on Found. of Comp. Science*, St. Louis, 1990, pp. 708–718.

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

[35] Fraysseix, H. de, and P. Rosenstiehl, A depth first characterization of planarity, *Annals of Discrete Math.* 13 (1982), pp. 75–80.

[36] Frederickson, G.N., and J. Ja'Ja, Approximation algorithms for several graph augmentation problems, *SIAM J. Comput.* 10 (1981), pp. 270–283.

[37] Fürer, M., X. He, M.-Y. Kao and B. Raghavachari, $O(n \log \log n)$-work parallel algorithms for straight-line grid embeddings of planar graphs, in: *Proc. 4th Annual IEEE Symp. on Parallel Algorithms and Architectures*, San Diego, 1992, pp. 410–419.

[38] Garey, M.R., and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman & Co., San Francisco, 1979.

[39] Garey, M.R., D.S. Johnson and L. Stockmeyer, Some simplified NP-complete graph problems, *Theoret. Comp. Science* 1 (1976), pp. 237–267.

[40] Garg, A., M.T. Goodrich and R. Tamassia, Area-efficient upward tree drawings, in: *Proc. 9th Annual ACM Symp. on Computational Geometry*, San Diego, 1993, to appear.

[41] Goldschmidt, O., and A. Takvorian, *An Efficient Graph Planarization Two-Phase Heuristic*, Tech. Report ORP91-01, Dept. of Mechanical Engineering, Univ. of Texas at Austin, 1991.

[42] Granot, G., *Planar Drawings of Graphs on the Rectlinear Grid with few Bends in each Edge*, M.Sc. Thesis (in Hebrew), Faculty of Comp. Science, the Technion, Israel, 1993 (in preparation).

[43] Haandel, F. van, *Straight Line Embeddings on the Grid*, M.Sc. Thesis, INF/SCR-91-19, Dept. of Comp. Science, Utrecht University, 1991.

[44] Hagerup, T., and C. Uhrig, Triangulating a Planar Graph, in: *Library of Efficient Datatypes and Algorithms (LEDA)*, software package, Max-Plank Institüt für Informatik, Saarbrücken, 1991.

[45] Harary, F., *Graph Theory*, Addison–Wesley Publ. Comp., Reading, Mass., 1969.

[46] He, X., *Efficient Parallel Algorithms for Two Graph Layout Problems*, Tech. Report 91-05, Dept. of Comp. Science, State Univ. of New York at Buffalo, 1991.

[47] He, X., On finding the rectangular duals of planar triangulated graphs, *SIAM J. Comput.*, 1993, to appear.

[48] He, X., and M.-Y. Kao, *Parallel Construction of Canonical Ordering and Convex Drawing of Triconnected Planar Graphs*, Manuscript, Dept. of Comp. Science, State Univ. of New York at Buffalo, 1993.

[49] Hopcroft, J., and R.E. Tarjan, Dividing a graph into triconnected components, *SIAM J. Comput.* 2 (1973), pp. 135–158.

[50] Hopcroft, J., and R.E. Tarjan, Efficient planarity testing, *J. ACM* 21 (1974), pp. 549–568.

[51] Hsu, T.-S., On four-connecting a triconnected graph, in: *Proc. 33st Annual IEEE Symp. on Found. of Comp. Science*, Pittsburgh, 1992, pp. 70–79.

[52] Hsu, T.-S., and V. Ramachandran, A linear time algorithm for triconnectivity augmentation, in: *Proc. 32th Annual IEEE Symp. on Found. of Comp. Science*, Puerto Rico, 1991, pp. 548–559.

[53] Hsu, T.-S., and V. Ramachandran, On finding a smallest augmentation to biconnect a graph, in: W.L. Hsu and R.C.T. Lee (Eds.), *Proc. of the Second Annual Int. Symp. on Algorithms*, Lecture Notes in Comp. Science 557, Springer-Verlag, 1992, pp. 326–335.

[54] Hsu, W.L., and S. Shih, *A simple test for planar graphs*, Manuscript, Institute of Information Science, Academia Sinica Taipei, Taiwan, 1992.

[55] Jansen, K., *One Strike Against the Min-max Degree Triangulation Problem*, Manuscript, Fachbereich IV, Mathematik und Informatik, Universität Trier, Germany, 1992.

[56] Jayakumar, R., K. Thulasiraman and M.N.S. Swamy, On maximal planarization of non-planar graphs, *IEEE Trans. on Circuits and System Sciences* 23 (1986), pp. 843–844.

[57] Jayakumar, R., K. Thulasiraman, and M.N.S. Swamy, Planar embedding: linear-time algorithms for vertex placement and edge ordering, *IEEE Trans. on Circuits and Systems* 35 (1988), pp. 334–344.

[58] Jayakumar, R., K. Thulasiraman and M.N.S. Swamy, $O(n^2)$ algorithms for graph planarization, *IEEE Trans. on Computer-aided Design* 8 (1989), pp. 257–267.

[59] Johnson, D.S., The NP-completeness column: an ongoing guide, *J. of Alg.* 3 (1982), pp. 89–99.

[60] Jones, S., P. Eades, A. Moran, N. Ward, G. Delott and R. Tamassia, *A Note on Planar Graph Drawing Algorithms*, Tech. Report No. 216, Key Centre for Software Technology, Dept. of Comp. Science, The University of Queensland, 1991.

[61] Jou, L.-j., H. Suzuki, and T. Nishizeki, *A Parallel Algorithm for Drawing Planar Graphs on the Grids*, Manuscript, Faculty of Information Engineering, Tohoku Univ., Japan, 1992.

[62] Jünger, M., and P. Mutzel, *Solving the Maximum Weight Planar Subgraph Problem by Branch and Cut*, Manuscript, Institüt für Informatik, Universität zu Köln, 1992.

[63] Kanevsky, A., R. Tamassia, G. Di Battista and J. Chen, On-line maintenance of the four-connected components of a graph, in: *Proc. 32th Annual IEEE Symp. on Found. of Comp. Science*, Puerto Rico, 1991, pp. 793–801.

[64] Kant, G., *Implementation Aspects of Graph Planarization Using PQ-trees*, M.Sc. Thesis, Dept. of Computer Science, Utrecht University, 1989.

[65] Kant, G., *An $O(n^2)$ Maximal Planarization Algorithm based on PQ-trees*, Tech. Report, RUU-CS-92-03, Dept. of Comp. Science, Utrecht University, 1992.

[66] Kant, G., Drawing planar graphs using the *lmc*-ordering, Extended Abstract in: *Proc. 33th Ann. IEEE Symp. on Found. of Comp. Science*, Pittsburgh, 1992, pp. 101-110.

[67] Kant, G., Hexagonal grid drawings, in: E.W. Mayr (Ed.), *Proc. 18th Intern. Workshop on Graph-Theoretic Concepts in Comp. Science (WG'92)*, Lecture Notes in Comp. Science 657, Springer-Verlag, 1993, pp. 263–276.

[68] Kant, G., A more compact visibility representation, in: J. van Leeuwen (Ed.), *Proc. 19th Intern. Workshop on Graph-Theoretic Concepts in Comp. Science (WG'93)*, Lecture Notes in Comp. Science, Springer-Verlag, 1993, to appear.

[69] Kant, G., and H.L. Bodlaender, Planar graph augmentation problems, Extended Abstract in: F. Dehne, J.-R. Sack and N. Santoro (Eds.), *Proc. 2nd Workshop on Algorithms and Data Structures*, Lecture Notes in Comp. Science 519, Springer-Verlag, 1991, pp. 286–298.

[70] Kant, G., and H.L. Bodlaender, Triangulating planar graphs while minimizing the maximum degree, in: O. Nurmi and E. Ukkonen (Eds.), *Proc. 3rd Scand. Workshop on Algorithm Theory*, Lecture Notes in Comp. Science 621, Springer-Verlag, 1992, pp. 258–271.

[71] Kant, G., and X. He, *Two algorithms for finding rectangular duals of planar graphs*, in: J. van Leeuwen (Ed.), *Proc. 19th Intern. Workshop on Graph-Theoretic Concepts in Comp. Science (WG'93)*, Lecture Notes in Comp. Science, Springer-Verlag, 1993, to appear.

[72] Khuller, S., and R. Thurimella, Approximation algorithms for graph augmentation, in: W. Kuich (Ed.), *Proc. 19th Int. Colloquium on Automata, Languages and Programming (ICALP'92)*, Lecture Notes in Comp. Science 623, Springer-Verlag, 1992, pp. 330–341.

[73] Koźmiński, K., and E. Kinnen, Rectangular dual of planar graphs, *Networks* 5 (1985), pp. 145–157.

[74] Kuratowski, K., Sur le problème des courbes gauches en topologie, *Fund. Math.* 15 (1930), pp. 271–283.

[75] La Poutré, J.A., *On-line Planarity Testing*, Tech. Report, Dept. of Comp. Science, Utrecht University, 1993, to appear.

[76] Lempel, A., S. Even and I. Cederbaum, An algorithm for planarity testing of graphs, *Theory of Graphs, Int. Symp. Rome* (1966), pp. 215–232.

[77] Lengauer, Th., *Combinatorial Algorithms for Integrated Circuit Layout*, Teubner/Wiley & Sons, Stuttgart/Chichester, 1990.

[78] Lin, Y.-L., and S.S. Skiena, *Complexity Aspects of Visibility Graphs*, Report 92-08, Dept. of Comp. Science, State Univ. of New York, Stony Brook, 1992.

[79] Lipton, R.J., D.J. Rose, and R.E. Tarjan, Generalized nested dissection, *SIAM J. Numer. Anal.* 16 (1979), pp. 346–358.

[80] Malitz, S., and A. Papakostas, On the angular resolution of planar graphs, in: *Proc. 24th Annual ACM Symp. Theory of Computing*, Victoria, 1992, pp. 527–538.

[81] Matoušek, J., M. Sharir and E. Welzl, *A Subexponential Bound for Linear Programming*, Tech. Report B-92-17, Fachbereich Mathematik, Freie Universität Berlin, Germany, 1992.

[82] Mitchell, S.L., Linear algorithms to recognize outerplanar and maximal outerplanar graphs, *Inf. Proc. Letters* 9 (1979), pp. 229–232.

[83] Micali, S., and V.V. Vazirani, An $O(\sqrt{V} \cdot E)$ algorithm for finding maximum matching in general graphs, in: *Proc. 21st Annual IEEE Symp. Found. of Comp. Science*, Syracuse, 1980, pp. 17–27.

[84] Mutzel, P., *A Fast Linear Time Embedding Algorithm Based on the Hopcroft-Tarjan Planarity Test*, Tech. Report 92.107, Institüt für Informatik, Universität zu Köln, 1992.

[85] Naor, D., D. Gusfield and C. Martel, A fast algorithm for optimally increasing the edge-connectivity, in: *Proc. 31st Annual IEEE Symp. on Found. of Comp. Science*, St. Louis, 1990, pp. 698–707.

[86] Nishizeki, T., and N. Chiba, Planar Graphs: Theory and Algorithms, *Annals of Discrete Mathematics* 32, North-Holland, 1988.

[87] Nummenmaa, J., Constructing compact rectilinear planar layouts using canonical representation of planar graphs, *Theoret. Comp. Science* 99 (1992), pp. 213–230.

[88] O'Rourke, J., *Art Gallery Theorems and Algorithms*, Oxford Univ. Press, New York, 1987.

[89] Otten, R.H.J.M., and J.G. van Wijk, Graph representation in interactive layout design, in: *Proc. IEEE Int. Symp. on Circuits and Systems*, 1978, pp. 914–918.

[90] Ozawa, T., and H. Takahashi, A graph-planarization algorithm and its applications to random graphs, in: *Graph Theory and Algorithms*, Lecture Notes in Computer Science 108, Springer Verlag, 1981, pp. 95–107.

[91] Preparata, F.P., and M.I. Shamos, *Computational Geometry, An Introduction*, Springer-Verlag, New York, 1985.

[92] Read, R.C., A new method for drawing a graph given the cyclic order of the edges at each vertex, *Congr. Numer.* 56 (1987), pp. 31–44.

[93] Reingold, E., and J. Tilford, Tidier drawing of trees, *IEEE Trans. on Software Engineering* 7 (1981), pp. 223–228.

[94] Richards, D., Finding short cycles in planar graphs using separators, *J. Alg.* 7 (1986), pp. 382–394.

[95] Rosenstiehl, P., H. de Fraysseix, and P. de Mendez, personal communication, 1992.

[96] Rosenstiehl, P., and R.E. Tarjan, Rectilinear planar layouts and bipolar orientations of planar graphs, *Discrete and Comp. Geometry* 1 (1986), pp. 343–353.

[97] Rosenthal, A., and A. Goldner, Smallest augmentations to biconnect a graph, *SIAM J. Comput.* 6 (1977), pp. 55–66.

[98] Schnyder, W., Embedding planar graphs on the grid, in: *Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms*, San Francisco, 1990, pp. 138–147.

[99] Stein, S.K., Convex maps, *Proc. Amer. Math. Soc.* 2 (1951), pp. 464–466.

[100] Storer, J.A., On minimal node-cost planar embeddings, *Networks* 14 (1984), pp. 181–212.

[101] Supowit, K., and E. Reingold, The complexity of drawing trees nicely, *Acta Informatica* 18 (1983), pp. 377–392.

[102] Tamassia, R., On embedding a graph in the grid with the minimum number of bends, *SIAM J. Comput.* 16 (1987), pp. 421-444.

[103] Tamassia, R., G. Di Battista and C. Batini, Automatic graph drawing and readability of diagrams, *IEEE Trans. on Systems, Man and Cybernetics* 18 (1988), pp. 61–79.

[104] Tamassia, R., and I.G. Tollis, A unified approach to visibility representations of planar graphs, *Discr. and Comp. Geometry* 1 (1986), pp. 321–341.

[105] Tamassia, R., and I.G. Tollis, Efficient embedding of planar graphs in linear time, in: *Proc. IEEE Int. Symp. on Circuits and Systems*, Philadelphia, pp. 495–498, 1987.

[106] Tamassia, R., I.G. Tollis and J.S. Vitter, Lower bounds for planar orthogonal drawings of graphs, *Inf. Proc. Letters* 39 (1991), pp. 35–40.

[107] Tarjan, R.E., Depth-first search and linear graph algorithms, *SIAM J. Comput.* 1 (1972), pp. 146–160.

[108] Tarjan, R.E., A note on finding the bridges of a graph, *Inf. Proc. Letters* 2 (1974), pp. 160–161.

[109] Thomassen, C., Planarity and duality of finite and infinite planar graphs, *J. Comb. Theory, Series B* 29 (1980), pp. 244-271.

[110] Tutte, W.T., Convex representations of graphs, *Proc. London Math. Soc.* 10 (1960), pp. 302–320.

[111] Tutte, W.T., How to draw a graph, *Proc. London Math. Soc.* 13 (1963), pp. 743–768.

[112] Vauchter, J., Pretty printing of trees, *Software Practice and Experience* 10 (1980), pp. 553–561.

[113] Wagner, K., Bemerkungen zum Vierfarbenproblem, *Jber. Deutsch. Math.–Verein* 46 (1936), pp. 26–32.

[114] Watanabe, T., Y. Higashi, and A. Nakamura, Graph augmentation problems for a specified set of vertices, in: T. Asano, T. Ibaraki, H. Imai and T. Nishizeki (Eds.), *Proceedings 1st Annual Int. Symp. on Algorithms*, Lecture Notes in Comp. Science 450, Springer-Verlag, 1990, pp. 378–387.

[115] Watanabe, T., and A. Nakamura, Edge-connectivity augmentation problems, *J. of Computer and System Sciences* 35 (1987), pp. 96–144.

[116] Watanabe, T., and A. Nakamura, 3-connectivity augmentation problems, in: *Proc. 1988 IEEE Int. Symp. on Circuits and Systems*, 1988, pp. 1847–1850.

[117] Watanabe, T., T. Narita, and A. Nakamura, 3-edge-connectivity augmentation problems, in: *Proc. 1989 IEEE Int. Symp. on Circuits and Systems*, 1989, pp. 335–338.

[118] Watanabe, T., M. Yamakado, and K. Onaga, A linear time augmenting algorithm for 3-edge-connectivity augmentation problems, in: *Proc. 1991 IEEE Int. Symp. on Circuits and Systems*, 1991, pp. 1168–1171.

[119] Westbrook, J., Fast incremental planarity testing, in: W. Kuich (Ed.), *Proc. 19th Int. Colloquium on Automata, Languages and Programming (ICALP'92)*, Lecture Notes in Comp. Science 623, Springer-Verlag, 1992, pp. 342–353.

[120] Wetherell, C., and A. Shannon, Tidy drawing of trees, *IEEE Trans. on Software Engineering* 5 (1979), pp. 514–520.

[121] Williamson, S.G., Embedding graphs in the plane – algorithmic aspects, *Ann. Discrete Math.* 6 (1980), pp. 349–384.

[122] Woods, D., *Drawing Planar Graphs*, Ph.D. Dissertation, Tech. Rep. STAN-CS-82-943, Computer Science Dept., Stanford University, 1982.

Index

Samenvatting

Computers raken meer en meer ingeburgerd in de samenleving. Ze worden gebruikt om informatie uit te rekenen, op te slaan en snel weer te geven. Deze weergave kan gebeuren in tekst, tabellen of in allerlei andere schema's. Een plaatje zegt vaak meer dan 1000 woorden, mits het plaatje duidelijk en overzichtelijk is. Een schema kan bestaan uit rechthoeken met informatie en verbindingslijnen tussen deze rechthoeken. Denk maar aan een schematische weergave van de organisatie structuur van een bedrijf. Of beschouw een schematische weergave van alle relaties en links in een database of een ander software programma. Ook een plan voor een uit te voeren project moet duidelijk laten zien welke onderdelen afhankelijk van elkaar zijn en tegelijk of na elkaar uitgevoerd moeten worden. Uit een schema moeten alle onderlinge relaties direct blijken.

Ook op het gebied van electrische schakelingen zijn er vaak vereenvoudigde schema's die alle verbindingen tussen de componenten weergeven. Denk maar aan de bijlagen van een televisietoestel. Een schema wordt hier veelal gebruikt om later reparaties of uitbreidingen aan de electrische schakelingen uit te voeren. De electrische schakelingen kunnen uit duizenden componenten bestaan. Als er zeer veel van deze schakelingen grafisch weergegeven moeten worden, is het belangrijk dat tekeningen van deze netwerken snel gemaakt kunnen worden, en het resultaat moet duidelijk en overzichtelijk zijn. In meer algemene zin bestaat een netwerk uit een aantal componenten, met verbindingen tussen deze componenten. In de wiskunde worden deze netwerken ook wel *grafen* genoemd. De componenten worden *knopen* genoemd en de verbindingen *lijnen*.

Dit proefschrift is gewijd aan het automatisch tekenen en grafisch representeren van grafen. De hierboven vermelde voorbeelden geven een goed inzicht in de betrokken vragen bij de methoden, ook wel *algoritmen* genoemd, om een layout van een graaf te maken. Helaas zijn esthetische criteria zoals "leesbaarheid" of een "mooie tekening" niet direct te vertalen tot wiskundige formules. Anderzijds kan een wiskundig optimaliseringcriterium een goede keus zijn voor een bepaalde graaf, maar leiden tot een onoverzichtelijke tekening in andere gevallen. Heel vaak voldoet een goede tekening aan een combinatie van optimaliseringscriteria. Een belangrijk criterium is ofdat de graaf zonder kruisende lijnen getekend kan worden. Als dit het geval is dan wordt de graaf *planair* genoemd.

We bestuderen in dit proefschrift het automatisch tekenen en representeren van

planaire grafen in het platte vlak en op roosters (dus alle coördinaten zijn gehele getallen). We tekenen de planaire grafen ook zonder kruisende lijnen. Belangrijke criteria voor de representatie van planaire grafen, genoemd in de literatuur, zijn de volgende:

- Het minimaliseren van het aantal bochten in de verbindingen (of het tekenen van de graaf met alle verbindingen als rechte lijnen weergegeven).
- Het minimaliseren van het totaal gebruikte gebied waarbinnen de representatie "mooi" kan worden weergegeven.
- Het plaatsen van de knopen, lijnen en bochten op roostercoordinaten.
- Het maximaliseren van de hoeken tussen elke twee opeenvolgende uitgaande verbindingen van een knoop.
- Het maximaliseren van de totale afstand tussen de knopen.
- De interne gebieden moeten convex getekend worden.

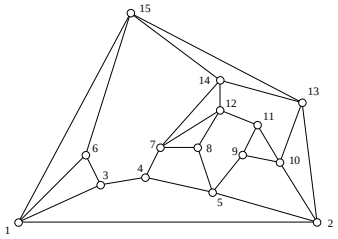
Kwantitatieve uitspraken over de kwaliteit van een tekenalgoritme worden steeds gedaan in termen van het aantal knopen van een graaf.

Het proefschrift is onderverdeeld in drie delen:

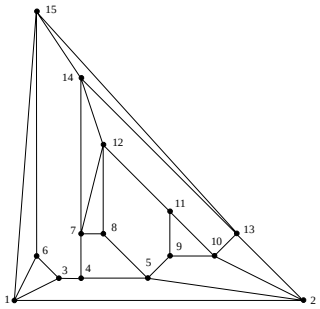
Deel A presenteert een inleiding tot het gebied van planaire grafen. Het geeft een uitgebreid overzicht ven de belangrijkste basistechnieken en algoritmen, die vooraf-gaan aan de algoritmen, beschreven in de andere delen.

Deel B beschouwt het probleem van het uitbreiden van planaire grafen zodat een bepaalde graad van samenhangendheid wordt bereikt. Een graaf heet *k-samen-hangend* als na het weglaten van $k-1$ willekeurige knopen de resulterende graaf nog steeds *niet* in meerdere stukken uiteen valt. Vele tekenmethoden voor planaire grafen stellen 2- of 3-samenhangendheid als voorwaarde voor de graaf, die getekend moet worden. In sommige andere gevallen moet de planaire graaf *getrianguleerd* zijn, dat wil zeggen, elk gebied moet een driehoek zijn. In deel B worden diverse algoritmen besproken om een planaire graaf uit te breiden met zo weinig mogelijk extra (dummy) verbindingen zodat aan de 2- of 3-samenhangendheidseis of aan de triangulatie voldaan wordt zonder de planariteit te verliezen. Hierna kan de gewenste tekenmethode toegepast worden, waarbij de toegevoegde verbindingen natuurlijk niet getoond worden in het uiteindelijke plaatje.

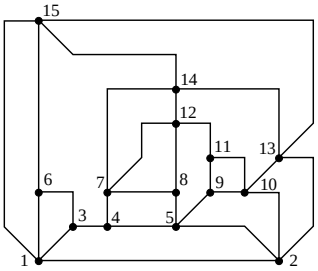
Deel C betreft het belangrijkste onderwerp van dit proefschrift, namelijk het tekenen en representeren van planaire grafen. Hiervoor presenteren we een totaal nieuwe, algemene methode. Deze is gebaseerd op een speciale ordening van de knopen van een 3-samenhangende planaire graaf, de *canonical ordening* genoemd. Door gebruik te maken van deze ordening kunnen vele representaties in lineaire tijd geconstrueerd worden, waarbij de graaf *altijd* planair getekend wordt. Zo worden de volgende resultaten verkregen. (Een bijbehorende illustratie van enkele resultaten is gegeven in de figuur op pagina 225.) n geeft het aantal knopen aan.



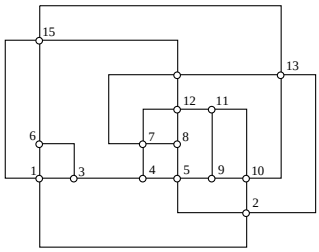
(a) De canonical ordening



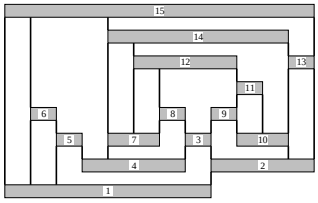
(b) Convexe gebieden



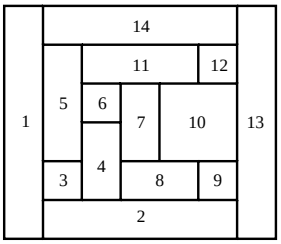
(c) Grote hoeken



(d) Orthogonale tekening



(e) Visibility representatie



(f) Rectangular dual

1. Elke 3-samenhangende planaire graaf kan getekend worden met rechte lijnen op een $(n-2) \times (n-2)$ rooster zodat elk interne gebied convex is.

2. Elke planaire graaf kan getekend worden op een rooster van grootte ten hoogste $(2n-6) \times (3n-6)$ met ten hoogste $5n-15$ bochten en minimum hoek $> \frac{2}{3d+1}$, waarbij elke lijn maximaal 3 bochten en lengte ten hoogste $2n$ heeft. d is de maximale graad.

3. Elke 3-samenhangende planaire graaf met maximale graad 4 kan getekend worden op een $n \times n$ rooster met knopen gerepresenteerd als punten en verbindingen als horizontale en vertikale lijnen met ten hoogste $\lceil \frac{3}{2}n \rceil + 4$ bochten waarbij elke lijn ten hoogste 2 bochten heeft voor $n > 6$ (*orthogonale tekening*).

4. Elke planaire graaf met maximale graad 3 kan getekend worden op een $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$ rooster met ten hoogste $\lfloor \frac{n}{2} \rfloor + 1$ bochten, waarbij er een opspannende boom is waarvan alle verbindingen rechte lijnen zijn en alle overige verbindingen ten hoogste 1 bocht hebben, voor $n > 4$.

5. Elke planaire graaf met maximale graad 3 kan getekend worden met knopen als punten en verbindingen als rechte lijnen en wel zodanig dat de kleinste hoek tussen twee opeenvolgende verbindingen van een knoop minimaal 45 graden is als de graaf 3-samenhangend is, en minimaal 60 graden anders.

6. Elke 4-samenhangende getrianguleerde planaire graaf kan getekend worden met knopen als rechthoeken zodanig dat twee rechthoeken een stuk grens

gemeenschappelijk hebben dan en slechts dan als er een verbinding is tussen de twee betreffende knopen (*rectangular dual*).

7. Elke planaire graaf kan getekend worden met de knopen als horizontale balken en alle verbindingen verticaal, alleen de betreffende eindknopen aanrakend, op een $(n-1) \times (n-1)$ rooster als de planaire graaf 4-samenhangend is, en op een $(\lfloor \frac{3}{2}n \rfloor - 2) \times (n-1)$ rooster anders (*visibility representatie*).

Het proefschrift bevat verder allerlei kleinere resultaten en voorbeelden. Diverse algoritmen zijn ook geïmplementeerd. Hierdoor zijn ook tabellen met resultaten van de diverse programma's opgenomen en geanalyseerd. We besluiten het proefschrift dan ook met diverse conclusies aan de hand van de theoretische en praktisch behaalde grenzen. We hopen dat u tevreden bent over de leesbaarheid en overzichtelijkheid van de getoonde graaftekeningen.

Curriculum Vitae

Goossen Kant

3 januari 1967	geboren te Rijswijk (N.Br.)
1979 – 1984	middelbare school: Chr. Scholen Gemeenschap "De Wegwijzer", Sleeuwijk
mei 1984	HAVO diploma behaald
1984 – 1985	HTS Den Bosch: sept. 1984 – febr. 1985: Weg- en Waterbouwkunde maart 1985 – juni 1985: Electrotechniek
1985 – 1989	studie informatica aan de Rijksuniversiteit Utrecht
augustus 1989	doctoraal examen informatica titel afstudeerscriptie: "Implementation Aspects of Graph Planarization Using PQ-trees"
1989 – 1993	Assistent in Opleiding aan de Vakgroep Informatica van de Rijksuniversiteit Utrecht