Department of Computer Science
Faculty of Mathematics, Physics and
Informatics
Comenius University

Analysis of algorithms for computing the crossing number

(Master's Thesis)

Helena Kocúrová

Advisor: doc. RNDr. Rastislav Královič, PhD.

Bratislava, 2007

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

.....

Acknowledgements

I am very grateful to my advisor Rastislav Královič for his patience, care, and a number of inspiring discussions during my work on this thesis.

I am also grateful to Imrich Vrťo for introducing me to the intriguing subject of crossing number and providing me with the initial volume of literature.

I am indebted to Peter Čech for helping me to acquire a large part of additional literature. My special thanks belong to Miloš Černák for his positive attitude and many helpful ideas in the implementation parts of this work.

Last but not least, I wish to thank my friends and family for their love and support during all those years.

Abstract

The Crossing Number Problem is the problem to decide, for a given integer $k > 0$, whether a graph G can be embedded in the plane with k or fewer pairwise edge crossings. Its optimization version, the Crossing Minimization Problem, looks for an embedding in the plane with a minumum number of edge crossings, and occurs as one of the major tasks in application areas like automatic graph drawing and VLSI design.

In this thesis, we survey and analyse exact, approximate, and heuristic algorithms for computing the crossing number of a graph, and some of its variants, *e.g.*, the rectilinear crossing number and the (fixed) linear crossing number. Since the Crossing Number Problem is NP-complete, no exact polynomial algorithms are known for the general problem. However, algorithms based on a branch-and-cut approach have been reported to solve medium sized instances to provable optimality in a reasonable computation time.

For graphs of bounded degree, approximate algorithms based on a bisection width concept have been developed, yielding an $O(\alpha^2(n)log\ n)$-approximation of the minimum drawing size, *i.e*, the number of crossings plus the number of vertices of a graph.

In practice, the problem is mostly attacked heuristically using a planarization approach consisting of two NP-hard steps, *i.e.*, the Maximum Planar Subgraph Problem and the Constrained Crossing Minimization Problem. In the first step, we obtain a planar subgraph and its embedding by deleting a possibly small number of edges from the graph. In the second step, the deleted edges are re-inserted back into the graph trying to keep the number of crossings small. Various strategies have been proposed optimizing over the set of all embeddings or iterating the whole edge-reinsertion process.

Finally, we propose two new heuristics for the (fixed) linear crossing number, respectively, based on the Simulated Annealing scheme and compare their performance with the best heuristics known from the literature.

Abstrakt

Problém priesečníkového čísla je problém určiť, pre dané celé číslo k, či graf G môžeme rozložiť v rovine tak, aby sa jeho hrany krížili navzájom najviac k-krát. Jeho optimalizačná verzia, hľadanie rozloženia grafu v rovine s najmenším počtom hranových prekrížení, tvorí jeden z hlavných problémov pri automatickom kreslení grafov a návrhu VLSI obvodov.

Táto práca sa zaoberá zhrnutím a analýzou exaktných, aproximatívnych a heuristických algoritmov na počítanie priesečníkového čísla grafu a jeho variantov, napr. rektilineárneho priesečníkového čísla a (fixného) lineárneho priesečníkového čísla. Vzhľadom k NP-úplnosti tohto problému, nie sú známe žiadne exaktné polynomiálne algoritmy na riešenie všeobecného problému. Napriek tomu, algoritmy založené na „branch-and-cut“ prístupe sú schopné efektívne a optimálne riešiť inštancie strednej veľkosti.

Pre grafy s ohraničeným stupňom sú známe aproximatívne algoritmy založené na „bisection width“ koncepte, dosahujúce $O(\alpha^2(n)log\ n)$-aproximáciu najmenšej kresliacej plochy, t.j. počet krížení plus počet vrcholov grafu.

V praxi sa tento problém najčastejšie rieši heuristicky tzv. planarizačným pristupom, ktorý pozostáva z dvoch NP-ťažkých krokov: Problém najväčšieho planárneho podgrafu a Problém obmedzeného minimalizovania krížení. V prvom kroku získame, zmazaním čo najmenšieho počtu hrán z pôvodného grafu, planárny podgraf a jeho rozloženie. V druhom kroku vkladáme zmazané hrany späť do grafu tak, aby výsledný počet krížení bol čo najmenší. Predstavíme rôzne stratégie založené napr. na optimalizovaní vkladania cez všetky rozloženia planárneho podgrafu alebo iterácii celého reinzertovacieho procesu.

Na záver navrhneme dve nové heuristiky na (fixné) lineárne priesečníkové číslo, respektívne, založené na schéme simulovaného žíhania a porovnáme ich úspešnosť s najlepšími heuristikami známymi z literatúry.

Contents

List of Figures

List of Tables

1 Introduction

For the first time, the crossing number problem was introduced by Pál Turán [200], who posed it, while in a forced labor camp during the WW2, in his Note of Welcome as follows:

"There were some kilns where the bricks were made and some open storage yards where the bricks were stored. All the kilns were connected by rail with all storage yards. ... the trouble was only at crossings. The trucks generally jumped the rails there, and the bricks fell out of them; in short this caused a lot of trouble and loss of time ... the idea occured to me that this loss of time could have been minimized if the number of crossings of the rails had been minimized. But what is the minimum number of crossings?"

Put in technical terms, the Turán's Brick Factory Problem is: *"What is the crossing number $cr(K_{n,m})$ of the complete bipartite graph $K_{n,m}$?"*

Garey and Johnson [80] have shown that the general Crossing Number decision problem: *"Given G and an integer k is $cr(G) \leq k$"* is NP-complete [79].

Due to the complexity of the Crossing Number Problem, many restricted variants have been considered in the literature. However, in most cases, *e.g.,* for bipartite, linear, circular, and book drawings, the problem remains NP-hard [64, 140, 141].

Although no reasonable polynomial algorithms are known for the general graph, algorithms are needed in practice. This work concentrates on gathering and analysing algorithmic solutions available for computing the crossing number of graphs today. Moreover, we conduct experimental studies evaluating the performance of the heuristics known from literature [39, 152] in comparison with our new proposed heuristics based on the simulated annealing approach for some restricted variants of the crossing number.

In the rest of this chapter, we present two important application areas of this problem, automatic graph drawing and VLSI design. Furthermore, we give an overview of this thesis in section 1.2.

1.1 Practical Applications

The crossing number represents a fundamental measure of nonplanarity of graphs but is also attractive from practical point of view.

Crossing minimization is one of the oldest and most fundamental problems arising in the area of automatic graph drawing and VLSI design. We can simply formulate it as follows: *"Given a graph $G = (V, E)$, draw it in the plane with a minimum number of edge crossings."*

Once the crossing numbers for complete graphs have been determined, the problem can be generalized to apply to less than complete graphs. Examples of such graphs are circuit diagrams, communication networks, railroad track and highway networks, and other interconnection diagrams.

The aesthetics and readability of graphlike structures (circuit diagrams, information diagrams, class hierarchies, flowcharts...) heavily depends on the number of crossings [114, 150], when the structures are visualized on a 2-dimensional medium.

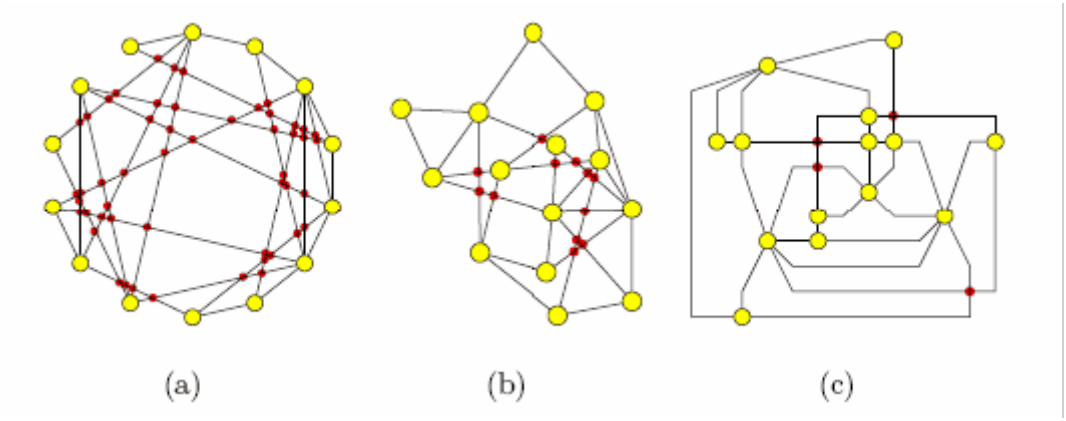

Figure 1: [20] Three drawings of the same graph with 51 (a), 12 (b), and 4 crossings (c).

1.1.1 Automated Graph Drawing

The main goal in automatic graph drawing is to obtain a layout that is easy to read and understand. The definition of layout quality depends on the particular application and is therefore hard to measure. However, the number of edge crossings is among the most important criteria [166].

Figure 1 shows a comparison of different drawings for the same graph preferring different aesthetic criteria. Most aesthetic criteria, for example, number of bends, uniformity of edge lengths, or drawing area, favor the first two drawings, while the last drawing is preferable with respect to the number of edge crossings.

A broad range of exixting algorithms for graph drawing favouring different criteria for both planar and nonplanar graphs can be experienced in practice using the Library of Algorithms for Graph Drawing (AGD) [4]. It also offers tools for implemening new algorithms.

Algorithms for Planar Graphs

Planar graphs are a relatively well studied class of graphs in respect to automatic layouts. There are linear time algorithms for testing the planarity of a given graph. Following [65], we can divide algorithms for the layout of planar graphs in those that use only straight lines and those that allow bends on the edges.

Wagner was upon the first who showed that every planar graph can be drawn without crossings using only straight lines to represent the edges [205]. Tutte presented in [201] an algorithm that produces a straight line drawing for planar graphs. Other algorithms were presented by De Fraysseix, Pach and Pollack [49] and by Schnyder [177] that draw graphs with n vertices on a grid of size $O(n^2)$.

If we allow bends for edges we can use an algorithm proposed by Tamassia [49]. The algorithm produces an orthogonal drawing if the maximum degree of any node of the given graph is at most four. Orthogonal drawings use horizontal and vertical straight line segments to represent the edges. Tamassia's algorithm minimizes the number of bends for a fixed combinatorial embedding by transforming the problem to a network flow problem.

There are extensions of the basic algorithm to graphs with maximum degree greater than four. Further information concerning this extensions can be found in [119].

Algorithms for Nonplanar Graphs

Nonplanar graphs are usually solved heuristically using a planarization approach. After computing a maximum or maximal planar subgraph we can use an algorithm for planar graphs to compute a combinatorial embedding. Afterwards the remaining edges are reinserted while trying to keep the number of crossings low.

In addition to this approach there are two more widespread techniques, the so called *spring embedder method* and the *hierarchical method.*

Spring embedders were introduced by Eades in [61]. The graph is interpreted as a physical system. Vertices are balls that repel each other and edges are modeled as springs between the balls. The preferred edge length can be influenced by changing the virtual spring constant of each edge. The algorithm tries to reach a state of minimum energy by moving the edges alongside their resulting force vector. Since crossings do not influence the overall energy of the system, the quality in respect to the number of crossings is usually poor, even for planar graphs.

The hierarchical method goes back to Sugiyama, Tagawa and Toda (see [121]). Their method works in three steps.

1. Assign vertices to layers such that no two adjacent nodes are placed at the same layer
2. Find a permutation of the nodes for each layer such that the number of crossings is minimized.
3. Compute the actual coordinates of the vertices. The nodes of a single layer are usually drawn on a straight line.

1.1.2 VLSI Design

VLSI Design (Very Large Scale Integration) deals with the layout of integrated circuits on a single chip. We can understand a chip layout as a collection of components that are connected by wires. One of the major problems in the design phase are crossings of wires. A widely used method is based on a two-layer approach.

Components are placed on one layer and crossings between wires are resolved by routing one of the wires to the second layer immediately before the crossing. After the wire has passed the crossing point it can be routed back to the primary layer.

The changes between the two layers are realized by using contact cuts. They occupy a large area and thus increase the total size of the layout. Moreover, the total edge length grows usually with the number of crossings and the wires tend to cross-talk at these points. This means that a change of the signal at one wire influences the voltage on the second wire which decreases the reliability of chips. Thus minimizing the number of crossings is one of the most important steps in the layout phase.

The work by Leighton [126] has shown that the crossing number of a graph can be used to obtain a lower bound on the amount of chip area required by that graph in a VLSI circuit layout. Chang [31] proposed an algorithm for minimizing *vias* in multi-layer circuits, which is a transformation of the minimal crossing problem.

1.2 Guide to this thesis

In chapter 2, we define basic terms from graph theory necessary for further reading of this thesis. We introduce the Crossing Number Problem and investigate its properties such as complexity and approximation bounds. We also present other variants of the crossing number which, though still NP-hard, can be used to obtain approximate values for the general crossing number satisfying specific restrictions.

Chapter 3 serves as an introduction to algorithmic solutions of the general Crossing Number Problem.

In chapter 4, we present algorithms for computing an exact value of the general crossing number. Some are simple branch-and-bound techniques, others combine the branch-and-bound method with a cutting plane approach in order to solve mathematical programming formulations of the problem.

In chapter 5, we describe approximation algorithms for computing the crossing number for bounded degree graphs, since there is no known polynomial time algorithm for general graphs. This approach is based on the bisection width concept using estimators in every node of the decomposition tree to obtain an $O(\alpha^2(n)log\ n)$-approximation of the minimum drawing size, *i.e.* $cr(G) + n$, where n is the number of vertices of the graph G and $cr(G)$ its crossing number.

Heuristic algorithms are introduced in chapter 6. One of the most important methods for solving the Crossing Number Problem heuristically represents the planarization approach. It consists of two NP-hard problems which are thoroughly discussed in chapters 7, and 8, respectively. In chapter 6, we also present another heuristic algorithm for computing the crossing number on complete graphs based on the scheme of Simulated Annealing.

Computing the maximum planar subgraph, *i.e.*, the first phase of the planarization approach, is discussed in chapter 7. We present numerous exact, approximation, and heuristic algorithms with two computational studies evaluating the performance of the heuristics.

In chapter 8, we discuss the second phase of the planarization approach, *i.e.*, reinserting edges deleted in order to obtain a maximum planar subgraph bach into the graph.

In chapter 9, we present algorithms for computing the rectilinear crossing number where edges are shall be drawn as straight lines.

Various exact and heuristic algorithms for computing the (fixed) linear crossing number are analysed in chapter 10. Moreover, we present an empirical evaluation of the heuristics known from literature in comparison with our two proposed heuristics based on the simulated annealing scheme.

In appendix A, we present some background knowledge from the theory of linear programming. Appendix B provides some basic definitions and theoretical results concerning two interesting data structures used in several algorithms discussed in this thesis, *i.e.*, the SPQR-trees and the PQ-trees.

2 Graphs and crossing number

This chapter provides a short introduction to the graph theory necessary for understanding the graph theoretical aspects of this thesis. Some basic notions, definitions and properties of graphs are given.

Next, we define the Crossing Number Problem and its connection to other NP-hard problems serving as a measure of planarity, and present some basic results concerning the complexity of the problem and its approximation bounds. This chapter ends with an overview of other variants of the Crossing Number Problem.

2.1 Preliminaries

A graph $G(V, E)$ consists of a vertex set V and an edge set E. The *order* of G is $|V| = n$ and the *size* is $|E| = m$. An edge is a pair of vertices, drawn as a line between them. Two vertices $u, v \in V$ are *adjacent* if and only if there is an edge $e = \{u, v\} \in E$. A graph has *multiple edges* if there exist edges $e_1, e_2, \ldots, e_i \in E, i \geq 2$, where $e_1 = e_2 = \ldots = e_i = \{u, v\}$ for some vertices u, v. An edge $\{u, u\}$ is called a *loop*. See Figure 2.

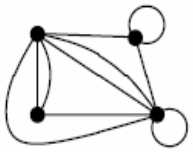

Figure 2: [204] Graph with multiple edges and loops.

A graph is *finite* if both vertex- and edge-sets are finite. A graph is *simple* if there are no loops or multiple edges in the graph. Only finite, simple graphs will he considered in this thesis.

If the graph G is *directed*, the edges are ordered pairs, written (u, v) where the direction is from u to v. A directed edge is drawn as an arrow, u being the tail and v the head, (u, v) is an *outgoing* edge of u and an *incoming* edge of v. The *degree* of a vertex is the number of edges adjacent to it. For directed graphs, a vertex has *out-degree* and *in-degree* equal to the number of its outgoing and incoming edges, respectively.

A *path* from u to v, $\{u, e_1, u_1, e_2, ..., u_{k-1}, e_k, v\}$, is a sequence of vertices and edges in which all vertices are distinct and all edges $e_i \in E$. For directed graphs, the edges must have the same direction as the path. That is, $e_i = (u_{i-1}, u_i)$, for all $i = 1, ..., k, u = u_0, v = u_k$. If $v = u$, we have a *cycle*.

A graph is *connected* if there is a path between every pair of vertices. A graph is *biconnected* if there are two vertex disjoint paths between every pair of vertices. A vertex is a *cut vertex* if its removal disconnects the graph.

A connected graph is a *tree* if every vertex of degree higher than one is a cut vertex (Figure 3). The vertices of a tree are called *nodes*.

Rooted trees are drawn as the middle tree in Figure 3, with the *root* on top. Rooted trees are considered to be directed in direction *from* the root. Thus, the root is the single node in a tree with no incoming edges.

Nodes with no outgoing edges are called *leaves*, and nodes with both incoming and outgoing edges are called *internal*. The root is the *ancestor* of all nodes below it, and they are the *descendants* of the root. Leaves are their own descendants.

A node, not root, and all its descendants define a *subtree*, with the node as root of the subtree, (Figure 3). The immediate ancestor and descendants of a node are often referred to as *parent* and *children*. Nodes that are children of the same parent are *siblings*.

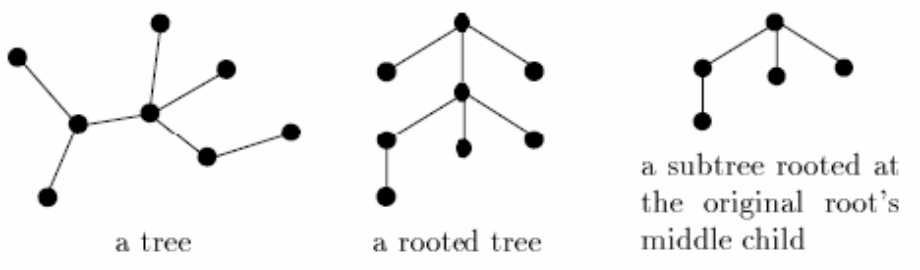

Figure 3: [204] The two first trees are the same graph, but the latter is rooted.

A graph is *complete* on n vertices, denoted K_n, if there is an edge between every pair of vertices.

A *bipartite* graph is a graph $G(V_1 \cup V_2, E)$, with $V_1 \cap V_2 = \varnothing$, where for every edge $(u, v) \in E$, $u \in V_1$ and $v \in V_2$ or vice versa.

A bipartite graph is complete if every vertex in V_1 is adjacent to every vertex in V_2. If $|V_1| = n_1$, and $|V_2| = n_2$, the graph is denoted K_{n_1,n_2}.

A $H(V', E') \subseteq G(V, E)$ have $V' \subseteq V$ and $E' \subseteq E$. A connected subgraph with exactly $n - 1$ edges is called a *spanning subtree* of G. For a non planar graph G, if H is planar, it is called a *planar subgraph*.

If no edge from $E - E'$ can be added to H without destroying planarity, it is a *maximal planar subgraph*. The largest of all such subgraphs of G with respect to E', is the *maximum planar subgraph* of G. A subgraph $H(V', E')$ where $V' \subset V$ and E' contains all edges $(u, v) \in E$ where $u, v \in V$, is called an *induced subgraph on* V.

The maximal biconnected subgraphs of a graph is called its *biconnected components* or *blocks*. A biconnected graph has exactly one biconnected component.

A subdivision of an edge $e = (u, v)$ is the insertion of a new vertex w on e, dividing e into $e_1 = (u, w)$ and $e_1 = (w, v)$.

One graph is a *homeomorph* of another, if the first can be obtained from the second by a sequence of subdivisions of edges.

2.1.1 Drawing of a graph

A *drawing* D of a graph G on a surface S consists of placing the vertices of G on S and drawing the edges of G using the continuous curves of S between the corresponding vertices, such that no curve has a vertex as an internal point and no point is an internal point of 3 curves.

We also speak about the images of vertices as vertices, and about the curves as edges. We say that two edges in a drawing *cross* in a certain point of the plane, or the point is a *crossing point* of the two edges, if this point belongs to the interiors of the curves representing the edges.

The *number of crossings* $cr(D)$ in the drawing D is the sum of the number of crossing points for all unordered pairs of edges.

A drawing is *good* when the following conditions hold:

(i) an edge does not cross itself,

(ii) any intersection of two edges is a crossing rather than tangential,

(iii) edges with common endpoints do not cross,

(iv) no three edges have a common crossing, and

(v) any pair of edges cross at most once.

Notice that if (iv) fails and some k curves cross each other in an otherwise normal drawing in a single point, then this situation can easily be transformed locally into a normal drawing where any two of the k curves cross each other locally once, and the number of crossings in the drawing does not change.

It is a easy to show, for any graph G, there is a good drawing of G having the minimum number of crossings. Two drawings are *isomorphic* if there is a homeomorphism from one to the other such that vertices are mapped to vertices.

The *crossing number* $cr(G)$ of the graph G is the minimum of $cr(D)$ over all drawings of G. We call a drawing D *optimal* (for cr) if it realizes $cr(D) = cr(G)$. Thus, we have an equivalent definition of $cr(G)$: the minimum of $cr(D)$ over all good drawings of G.

2.1.2 Embedding of a graph

For our purposes, a *compact-orientable 2-manifold* or simply a surface, may be thought of as a sphere or a sphere with handles. The *genus* of the surface is the number of handles.

An *embedding* of a graph G on a surface S is a drawing of G on S in such a manner that edges intersect only at a vertex to which they are both incident.

A region in an embedding is called a 2-*cell* if any simple closed curve in that region can be continuously deformed or contracted in that region to a single point.

An embedding is called a 2-*cell embedding* if all the regions in the embedding are 2-cell. An algebraic description of a 2-cell embedding was given by Dyck [59] and Heffter [99]. This description is referred to as a Rotational Embedding Scheme.

Definition 1 (**Good embedding**). *We call an embedding of graph G on a surface S of genus g a good embedding if it satisfies the following conditions:*

(i) all vertices of the graph are given as distinct points in S;

(ii) no two edge crossings happen in the same point in S;

(iii) for any edge no vertex of the graph, except the endpoints of the edge, is situated on the edge.

The relationship between the number of regions of a graph and the surface on which it is embedded is described by the well-known *generalized Euler's Formula* [32]:

Theorem 1. *Let G be a connected graph with n vertices and m edges with a 2-cell embedding on the surface of genus g having f regions. Then: $n - m + f = 2 - 2g$.*

2.1.3 Combinatorial embedding

In the current definition, we allow drawings with arbitrary curves for edges. How would these be stored in a computer, *i.e.*, represented in a discrete way? One option is to store vertices as points and declare edges as straight lines.

Another is to represent planar graphs via what is called a *combinatorial embedding*. This concept actually exists for all graphs, whether planar or not, but is equivalent to planar graphs in a special case.

Definition 2 (Combinatorial embedding). *Let G be a graph. A combinatorial embedding of G is a set of orderings Π_v for each vertex $v \in V$, where Π_v specifies a cyclic ordering of edges incident to v.*

If we are given a drawing of a graph (with crossings or without), then this always implies a combinatorial embedding, by taking the clockwise order of the edges incident to each vertex. On the other hand, if we are given a combinatorial embedding, then it is easy to create a drawing (with crossings, possibly) such that the combinatorial embedding exactly corresponds to the clockwise order of edges at each vertex. Figure 4 shows an example of a combinatorial embedding.

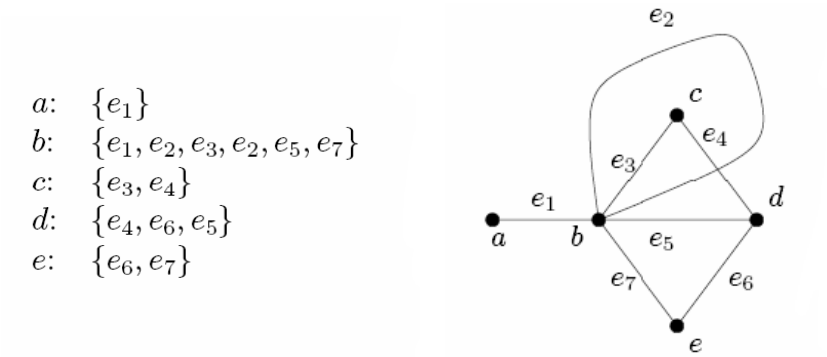

Figure 4: [1] A combinatorial embedding of a graph, and a drawing that respects it.

A graph is called *planar* when it admits a drawing into the plane without edge-crossings. There are infinitely many different drawings for every planar graph, but they can be divided into a finite number of equivalence classes. We call two planar drawings of the same graph *equivalent* when the sequence of the edges in clockwise order around each vertex is the same in both drawings. In this case they realize the same *combinatorial embedding*.

In general, a planar graph can have an exponential number of combinatorial embeddings, see Figure 5. A combinatorial embedding also defines the set of cycles in the graph that bound faces in a planar drawing.

An alternative definition of a *combinatorial embedding* is defined as a clockwise ordered list of adjacent neighbors for each vertex $v \in V$. When, in addition, the outer face is fixed, the combinatorial embedding is also called a *planar embedding* of G.

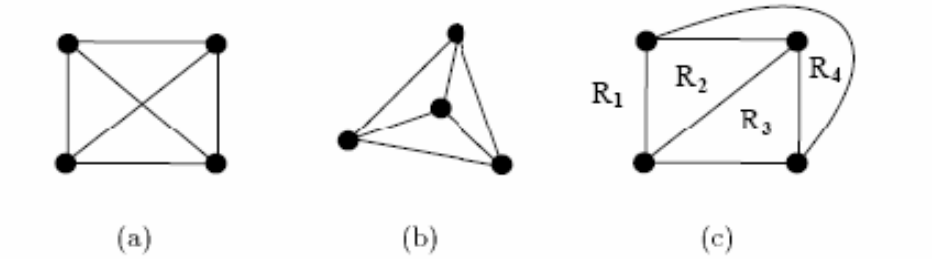

Figure 5: Three embeddings of K_4, (c) with the four faces marked.

The complexity of embedding planar graphs has been studied by various authors in the literature [16, 17, 24]. Given a planar graph, a combinatorial embedding can be computed in linear time [33, 143].

2.1.4 Properties of planarity and non-planarity

As stated earlier, a planar graph, by definition, has a planar embedding, and any planar embedding shows the number of faces of the graph. Then, by the *Euler's polyhedral formula*, we have: $n - m + f = 2$.

In a planar embedding of a maximal planar graph every face must be a triangle, the number of edges in such a graph is thus $m = 3n - 6$. Hence any graph on n vertices with $m > 3n - 6$ edges is non-planar.

Another famous and important result, this time on non-planarity, was published in 1930 by Kuratowski [123]. It is known as the *Kuratowski Theorem*.

Theorem 2. *A graph G is nonplanar if and only if there is a subgraph of G which is homeomorphic to either $K_{3,3}$, or K_5.*

This is why K_5 and $K_{3,3}$ often are called *Kuratowski graphs*. K_5 and $K_{3,3}$ are the smallest non planar graphs.

2.2 Degrees of non planarity

There are several NP-hard problems that relate to the degree of non-planarity of a given graph. Although the primary scope of this work are methods for solving the Crossing Number Problem, we define and give some theoretical results for four such problems: Crossing Number, Skewness, Thickness and Splitting Number.

All four can be viewed as *measures of non-planarity* of a graph G and are somewhat related in the way they may be heuristicly solved, as a heuristic solution for one problem may provide a heuristic solution for the other [204].

2.2.1 The crossing number

The *crossing number* $cr(G)$ is the minimum number of edge crossings in any possible embedding of G.

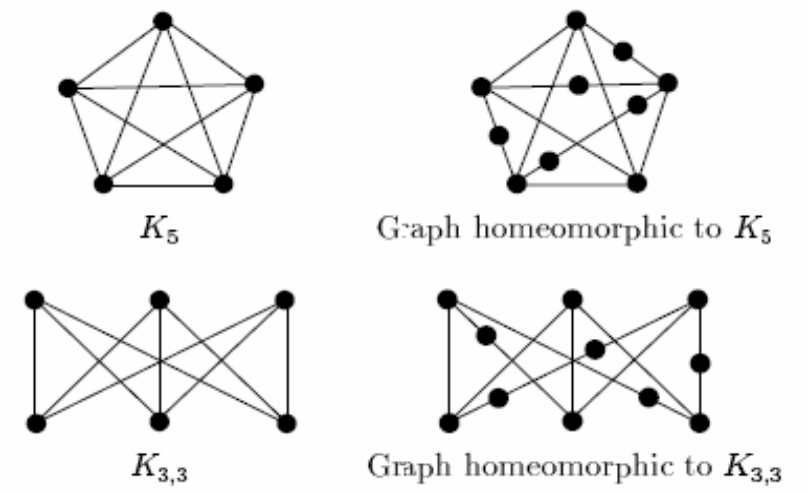

Figure 6: [204] K_5 and $K_{3,3}$ with their respective homeomorphs.

If the drawing is required to have only straight lines, we get the *rectilinear crossing number* $\overline{cr}(G)$. Naturally, $\overline{cr}(G) \geq cr(G)$. Since any planar graph can be embedded using only straight lines [71], $\overline{cr}(G) = cr(G) = 0$ for G planar.

The *Crossing Number Problem* [79, 80] is the problem of determining, for a given integer k, whether a graph G can be embedded in the plane with k or fewer pairwise crossings of the edges (not including the intersections of the edges at their common endpoints). Formally, the Crossing Number Problem is the decision problem:

Problem 1 (Crossing Number). *Given a graph G and an integer k, is $cr(G) \leq k$?*

Garey and Johnson [80] proved it NP-complete, which makes the optimization version of crossing number NP-hard, as we will show in section 2.3, where we discuss the complexity of the Crossing Number Problem in further detail.

The *Crossing Minimization Problem* is the problem of finding an embedding of a graph in the plane with the minimum number of edge crossings.

2.2.2 The skewness

The *skewness number* or just *skewness* $sk(G)$ of a graph $G(V, E)$ is the minimum number of edges whose removal makes G planar. Thus, if E' is a set of such edges, $sk(G) = |E'|$. If E' is given, the resulting subgraph $H(V, E - E')$ is a maximum planar subgraph or G. For G planar, $sk(G) = 0$ and $H = G$.

Finding the Skewness of a graph was shown NP-hard by Liu and Geldmacher in [136].

2.2.3 The thickness

The *thickness* of a graph, $th(G)$, is the smallest number of planar subgraphs of G whose union is G. For G planar, $th(G) = 1$, since it is its own planar subgraph.

The Thickness Problem is NP-hard, for a proof see [139]. A more complete historical review and further references can be found *e.g.* in [193].

2.2.4 The splitting number

The splitting number of a graph G is the smallest integer $k \leq 0$, such that a planar graph can be obtained from G by k splitting operations. Such operation replaces a vertex v by two nonadjacent vertices v_1 and v_2, and attaches the neighbors of v either to v_1 or to v_2.

In [69], Faria, de Figueiredo and Mendonça proved that the Splitting Number decision problem is NP-complete. As a consequence, [69] obtained that Maximum Planar Subgraph Problem remains NP-complete when restricted to graphs with maximum degree 3, to graphs with no subdivision of K_5, or when restricted to cubic graphs.

2.2.5 Some relationships between these problems

The problems discussed above all try to give a numerical value for how far a non planar graph is from planarity. Since all these problems are NP-hard, no exact, polynomial algorithms are known for the general graph. Vollen [204] proposed an intuitive heuristic for skewness is proposed, based on a solution to or estimate of the crossing number.

Given a heuristic or accurate algorithm for the maximum planar subgraph problem, a straight forward heuristic for the thickness problem is to repeatedly apply the planar subgraph algorithm and remove this subgraph, until the graph is empty [204].

The thickness estimate is then the number of maximum or maximal planar subgraphs extracted [42]. The connection between the thickness $th(G)$ and crossing number $cr(G)$ is given by the relation $th(G) \leq cr(G) + 1$.

As shown in [204], an algorithm for crossing number that also determines the edges involved in each crossing can give a heuristic result for the skewness number.

Assume that each edge has received a set of (pointers to) crossing edges by the crossing number algorithm. Edges are then given a priority according to the number of crossings they are involved in. Edges with priority > 0 are put in a priority queue, the rest are left in the planar subgraph.

Each edge that is removed from the front of the queue, is counted in the skewness number. When no longer part of the graph, the priority of the edges it crosses with, is decremented by 1. Edges fall out of the queue and into the planar subgraph when their priority reaches zero.

2.3 Complexity of the crossing number problem

NP-complete problems are a class of decision problems that are considered to be intractable. In other words, solutions to these problems probably will not be found by using a polynomial time algorithm. Although it has not been proven whether or not NP-complete problems are truly intractable, it would appear that a major breakthrough will be necessary to solve them in polynomial time, for more information on NP-completness see *e.g.* [79, 109]. In this section, we show that the crossing number problem is NP-hard.

2.3.1 Crossing number is NP-complete

As defined, the Crossing Number Problem is in NP. One need only guess the k or fewer crossings (and the order in which they occur along edges involved in more than one

crossing), create a new "crosspoint" vertex for each, replace each edge involved in one or more crossings by a path that contains all the crosspoint vertices associated with that edge in the appropriate order, and then test the resulting graph for planarity.

Note that the above approach also allows us, for any fixed value of k to test whether $cr(G) \leq k$ in polynomial time (the degree of the polynomial depending on k. The algorithm guesses $l \leq k$ pairs of edges that cross and tests if the graph obtained from G by adding a new vertex at each of these edge crossings is planar. The running time of this algorithm is $n^{\Theta(k)}$[88].

Garey and Johnson have proved that the general Crossing Number decision problem: *"Given G and an integer k is $cr(G) \leq k$?"*, is NP-complete [79] and hence likely to be intractable.

To prove that the Crossing Number Problem is NP-complete, one must show that a known NP-complete problem can be transformed to it. The "known" NP-complete problem for Garey and Johnson [81] was the Optimal Linear Arrangement Problem: *"Given a graph $G = (V, E)$ and an integer k, is there a one-to-one function $f : V \to \{1, 2, \ldots, |V|\}$ such that $\sum_{(u,v)\in E} |f(u) - f(v)| \leq k$?"*

Theorem 3 (Garey, Johnson [80]). *There is a polynomial reduction of OLA to the Crossing Number Problem.*

A quick look at the reduction in [80] reveals that the constructions used involve (many) parallel edges. Since crossing number of a graph is not changed under subdivision of edges, the natural workaround is to subdivide the parallel edges. However, that yields large collections of nontrivial 2 cuts. This gives rise to a very natural question: is the Crossing Number Problem *NP*-complete even for 3 connected simple graphs? Hliněný proved the following result:

Theorem 4 (Hliněný [102]). *The Crossing Number Problem is NP-complete for 3-connected (simple) cubic graphs.*

2.3.2 Crossing number is fixed-parameter tractable

Downey, Fellows, Niedermeier and Rossmanith[56] raised the question if the crossing-number problem is *fixed parameter-tractable,* that is, if there is a constant $c \geq 1$ such that for every fixed k the problem can be solved in time $O(n^c)$.

Grohe [88] answered this question positively with $c = 2$. In other words, he has shown that for every fixed k there is a quadratic time algorithm deciding whether a given graph G has crossing number at most k. However, the constant factor of this algorithm is double-exponential in k, which makes it useless for practical purposes.

2.3.3 Crossing number on graphs with bounded tree-width

Generally speaking, many known results show that hard algorithmic problems become easy for graphs of bounded tree-width. So, *how difficult is it to determine crossing number of a given graph of bounded, tree-width (or path-width, or even bandwidth)?*

The pessimistic point of view is supported by two observations. The first considers OLA, which is reducible to the Crossing Number Problem and for which there has been

a long and so far infructuous quest for a polynomial algorithm for graphs with bounded tree-width (of path-width).

The other considers the reduction of the Bandwidth Problem, NP-complete even on graphs with bounded tree-width, to Unsaturated Drawing Problem [103], where the latter is naturally related to the Crossing Number Problem.

Not surprisingly, exact crossing numbers are in general very difficult to compute. As a consequence, future research into crossing numbers will be justified in focusing on inexact methods that only *estimate* crossing numbers, and the quest for exact values of $cr(G)$ will have to be restricted to promising special cases.

Dealing with such a difficult-to-compute parameter, structural theorems are most valuable, but such results have been proved hard to come by. Those include, among others, investigations on crossing-critical graphs (see [101, 168]) and general bounds for crossing numbers of graphs (see for instance [159]).

2.4 Known bounds

Much of the literature falls into one of two categories: the first investigates exact values of crossing numbers or makes lower bounds on crossing numbers based on information on the crossing number of a certain small graph, the second tries to prove bounds based on structural properties of the graph [186]. The first is called the theory of small graphs, the second is called the theory of large graphs.

2.4.1 Theory of small graphs

During the early history of crossing numbers the theory of small graphs existed only. For more information on the early history and the theory of small graphs, see White and Beineke [209], for the modern history and the theory of large graphs, see Shahrokhi, Sýkora, Székely and Vrťo [178], and for the most recent results see Pach [157].

Zarankiewicz's conjecture

The Turán's Brick Factory Problem is: *what is the crossing number* $cr(K_{n,m})$ *of the complete bipartite graph* $K_{n,m}$ *?*

Place $\lfloor \frac{n}{2} \rfloor$ vertices to negative positions on the x-axis, $\lfloor \frac{n}{2} \rfloor$ vertices to positive positions on the x-axis, $\lfloor \frac{m}{2} \rfloor$ vertices to negative positions on the y-axis, $\lfloor \frac{m}{2} \rfloor$ vertices to positive positions on the y-axis, and draw nm edges by straight line segments to obtain a drawing of $K_{n,m}$. It is not hard to check that the following formula gives the number of crossings in this particular drawing: $\lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{m-2}{2} \rfloor \lfloor \frac{m-3}{2} \rfloor$.

Zarankiewicz's Crossing Number Conjecture is that the drawing described above is optimal. In 1963, P. Kainen and G. Ringel found an error in the induction argument of the proof. Nevertheless it can be used as an upper bound for the crossing number of a comptete bipartite graph. Kleitman [127] verified this conjecture in the special case $\min\{m, n\} \leq 6$, and Woodall [160] for $\min\{m, n\} \leq 7$.

Guy [95] conjectured that the crossing number $\mathrm{cr}(K_n)$ of the complete graph K_n is equal to $\mathrm{Z}(n) = \frac{1}{4} \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{n-2}{2} \rfloor \lfloor \frac{n-3}{2} \rfloor$. He proved this for $n \leq 10$ and also determined that,

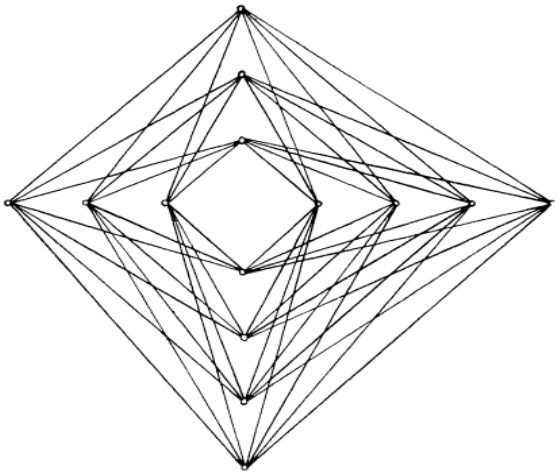

Figure 7: [67] Drawing of $K_{7,7}$ with a minimum number of 81 crossings using Zarankiewicz's rule.

for $n = 4, 5, 6, 7, 8$, the number of optimal drawings of K_n is $1, 1, 1, 1, 5, 3$, respectively [169].

Pan and Richter [163] constructed an algorithm by which they have shown that $cr(K_{11}) = Z(11)$. In particular, they determined that K_9 and K_{10} have 3080 and 5679 optimal drawings, respectively. Along the way, they showed that every good drawing of K_n induces a 3-connected planar graph. The main theoretical result they obtained is the following:

Theorem 5 (Pan and Richter [163]).

1. *For $n \leq 8$, any optimal drawing of K_n contains an optimal drawing of $Kn-1$.*
2. *Any optimal drawing of K_9 contains a good drawing of K_8 with at most 20 crossings. Any good drawing of K_8 with at most 20 crossings contains an optimal drawing of K_7.*
3. *Any good drawing of $K_1 1$ with fewer than 100 crossings contains a good drawing of $K_1 0$ with at most 62 crossings. Any good drawing of $K_1 0$ with at most 62 crossings contains an optimal drawing K_9.*

It is usually not hard to come up with drawings of graph whose optimality is intuitively clear. The difficulty lies in proving matching lower bounds for the crossing numbers.

General bounds

The simplest lower bound for the crossing number of a simple graph with $n \geq 3$ vertices and m edges is $cr(G) \geq m-3n+6$. Since any simple planar graph cannot have more that $3n-6$ edges, this immediately follows from Euler's polyhedral formula, and already gives $cr(K_5) \geq 1$. A counterpart of this formula for triangle-free graphs $cr(G) \geq m-2n+4$, which proves $cr(K_{3,3}) \geq 1$. Both these formulas can give interesting lower bounds for small graphs only, since the magnitude of the crossing number can be as large as m^2.

Leighton [127] used induction on the number of nodes to show the following lower bound for $m \geq 4n$, where n denotes the number of vertices and m the number of edges:

$$cr(G) \geq \frac{1}{100}\frac{m^3}{n^2}$$

. In [3], Ajtai et al. obtained the same result independently with a smaller constant of $\frac{1}{375}$.

In [67], Erdös and Guy have shown the following lower bound for $cr(K_n)$:

$$cr(K_n) \geq \frac{1}{80}n(n-1)(n-2)(n-3).$$

Apart from bounds with respect to the number of vertices and edges we can try to obtain lower bounds from other properties of a graph, *e.g.,* the *skewness, bisection width* or *cutwidth.*

Let $sk(G)$ be the skewness of graph G. Each of the removed edges produces at least one crossing, hence it is not difficult to see that $cr(G) \geq sk(G)$ for any graph G. Cimikowski showed in [41] that a planar graph can have skewness one, but an arbitrary high Crossing Number.

Efficient lower bounds are obtained using the bisection width concept. The bisection width of a graph $G = (V, E)$ is the minimum number of edges whose removal divides G into two parts having at most $2|V|/3$ vertices each. Leighton [127] proved that in any n-vertex graph G of bounded degree, the crossing number satisfies

$$cr(G) + n = \Omega(bw^2(G)).$$

This bound was extended to

$$cr(G) + \frac{1}{16}\sum_{v \in V} d_v^2 \geq \frac{1}{40}bw^2(G)$$

in [158, 185], where d_v is the degree of any vertex $v \in V$.

In [55], Djidjev and Vrťo improved this bound by replacing the bisection width with a larger parameter - the cutwidth of the graph.

Unfortunately there are nearly no known general upper bounds for $cr(G)$. The only bound can be obtained from the observation that the crossing number of a graph G with n nodes cannot exceed the crossing number of the complete graph K_n. Hence we have

$$cr(G) \leq cr(K_n) \leq \lfloor\frac{n}{2}\rfloor\lfloor\frac{n-1}{2}\rfloor\lfloor\frac{n-2}{2}\rfloor\lfloor\frac{n-3}{2}\rfloor.$$

The standard counting method

A basic technique to obtain a lower bound for the crossing number of a larger graph from that of a sample graph is the *standard counting method.*

Take a hypothetical (good, optimal) drawing of the large graph, find many copies of the sample graph in it, each exhibiting as many crossings as its crossing number. Add

up those numbers, and divide by the largest multiplicity with which a crossing may have been counted in different copies of the sample graph [186]

Graph minors

The graph minor community also has an interest in crossing numbers. Their usual approach is characterization in terms of excluded minors.

Robertson and Seymour [172] calls a graph H *singly crossing* provided H is a minor of a graph that can be drawn on the sphere with at most one crossing. They show that a graph is singly crossing if and only if it does not have one of 41 explicitly given graphs as a minor.

2.4.2 Theory of large graphs

The modern history started with Leighton's thesis [127]. Leighton introduced methods to set lower bounds for crossing numbers which instead of crossing numbers of small graphs, depended on certain parameters of the large graphs. He introduced three methods that become classic: lower bounds in terms of number of edges, bisection width, and graph embedding.

For more on the results concerning the number of edges, see, for example [2, 3, 67, 127, 162, 159], a survey of results on the bisection width and graph embedding can be found in [178], for the results concerning random graphs, see [161, 182].

2.5 Other crossing numbers

Considering the way, how a graph is embedded on a surface, we recognise several variants of the crossing number problem. The most significant are the *rectilinear crossing number, (fixed) linear crossing number, book crossing number and k-layer crossing number.*

In [160], Pach and Tóth introduced two new variants of the crossing number problem:

The pairwise crossing number $cr_{pair}(G)$ is equal to the minimum number of unordered pairs of edges that cross each other at least once, (*i.e.*, they are counted once instead of as many times they cross), over all normal drawings of G; and

The odd crossing number $cr_{odd}(G)$ is equal to the minimum number of unordered pairs of edges that cross each other odd times, over all normal drawings of G.

In Tutte's work [202], another kind of crossing number is implicit:

The independent-odd crossing number $cr_{iodd}(G)$ is equal to the minimum number of unordered pairs of non-adjacent edges that cross each other odd times, over all normal drawings of G.

The following chain of inequalities is obvious from the definitions [157]:
$cr_{iodd}(G) \leq cr_{odd}(G) \leq cr_{pair}(G) \leq cr(G)$

2.5.1 Rectilinear crossing number

The *rectilinear drawing* $D(G)$ of a graph G is a configuration of G on the plane in such a way that every edge is composed of horizontal and vertical line segments. Often the plane is seen as a rectangular grid.

Rectilinear crossing minimization looks for an embedding of a graph G with the minimum number of edge crossings, where the edges are represented as straight lines. This minimum number of edge crossing is called the *rectilinear crossing number* [66, 108, 160, 182].

Problem 2 (Rectilinear Crossing Number). *Given a graph $G = (V, E)$, integer $k \geq 0$, decide whether $\overline{cr}(G) \leq k$?*

Fáry's theorem [71] telling that planar graphs can be drawn using straight line segments for edges and Zarankicwicz's Crossing Number Conjecture may suggest that optimal drawings can be done using straight line segments for edges. This is not the case.

Guy showed that first for K_9 [95], and later Bienstock and Dean [15] exhibited a series of graphs with crossing number 4, whose rectilinear crossing numbers are arbitrary large. On the other hand, they proved that any graph G for which $cr(G) \leq 3$ must also satisfy $cr(G) = \overline{cr}(G)$, as an extension of the Fáry's theorem. This yields that $cr(G) \leq \overline{cr}(G)$.

Using the proof in [80], it can be shown that computing the rectilinear crossing number is NP-hard. This problem is not yet known to be in NP, for the (cartesian) coordinates of the vertices in a drawing can be assumed to be rational, and thus, integral.

We remark that practically every paper on crossing numbers has in fact also dealt with rectilinear crossing numbers (the latter sometimes used to approximate the former).

2.5.2 Linear crossing minimization

We call an embedding of a given graph *linear embedding*, when it satisfies the following conditions:

(i) the vertices are placed on a horizontal line l, and

(ii) the edges are drawn by semicircles (see Figure 8).

We call this type of embedding a *linear embedding.*

We can consider the linear crossing minimization problem with an additional constraint,

(iii) the positions of the vertices on l are predetermined.

The *Fixed Linear Crossing Minimization Problem* is the problem of finding a linear embedding of a graph with the minimum number of edge crossings under a specified vertex ordering. We call the problem of finding such an embedding with no vertex ordering specified the *Free Linear Crossing Minimization Problem.*

Furthermore, we define the *Free Linear Crossing Number Problem* to be that of determining, for a given integer k, whether there is a linear embedding of a graph with k or fewer edge-crossings. When a vertex ordering is specified, we call it the *Fixed Linear Crossing Number Problem.*

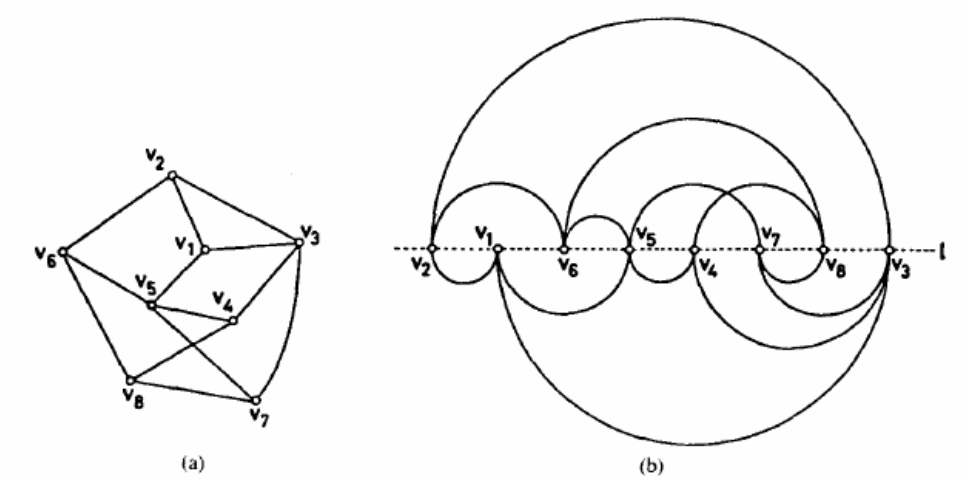

Figure 8: [141] (a) A simple graph G. (b) A linear embedding of G.

2.5.2.1 Fixed linear crossing number

Let $f : V \to \{1, 2, \ldots, |V|\}$ be a one-to-one function. We call an embedding G' of G in the plane an *f-fixed linear embedding*, or simply an *f-linear embedding*, if

(i) each vertex $v \in V$ is placed on the x-axis l with x-coordinate $f(v)$,

(ii) the edges in E are drawn by semi-ellipses, and

(iii) the semi-ellipses for nonparallel edges intersect in at most one point.

Problem 3 (Fixed Linear Crossing Number). *Given a graph $G = (V, E)$, integer $k \geq 0$ and one-to-one function $f : V \to \{1, 2, \ldots, |V|\}$, decide whether $cr_f(G) \leq k$?*

The special case of this problem in which $k = 0$ can easily be solved. Suppose $V = \{v_1, v_2, \ldots, v_n\}$ and $f(v_i) = i$ for $i = 1, 2, \ldots, n$. It is obvious that $cr_f(G) = 0$ if and only if graph $(V, E \cup \{(v_i, v_{i+1}) | i = l, 2, \ldots, n - l\} \cup \{(v_n, v_1)\})$ is planar. Thus, one can solve the problem in linear time by using one of the existing graph planarity testing algorithms [19, 104]. In the general case the Fixed Linear Crossing Number Problem is NP-complete.

The Linear Crossing Number Problems are related to the *Book Embedding Problems* [11, 35] which have recently attracted considerable attention.

2.5.3 Book crossing number

Informally, a drawing of an undirected graph G = (V, E) in the book consists of an ordering of the vertices on the spine and then drawing each edge of the graph in one page of the book with a curve, such that any curve has only its two end-points on the spine and no three curves intersect in one point unless it is an end-point in common.

The spine of the book is a line. Each page is a half-plane that has the spine as its boundary. One can assume with no loss of generality that the curves for drawing the edges are half-circles.

For a drawing D of G on a k-page book, let $cr_k(D)$ denote the number of edge crossings in D. The book crossing number $cr_k(G)$ is the minimum number of crossings among all k-page book drawings of G.

Problem 4 (Book Crossing Number). *Given a graph G, a k-page book and an integer $K>0$, decide whether there is a k-page book drawing of graph G with less than or equal to K crossings?*

A *book embedding* of a graph is an embedding in a book with the vertices placed on the spine and the edges on the pages such that no two edges drawn on the same page cross each other.

The *Book Embedding Problem* is the decision problem: whether a given planar graph can be embedded in a k-page book, so that no crossings occur.

The problem of embedding a graph in a book may be solved intuitively in the following way:

1. Embed the graph so that its vertices lie on a circle and its edges are chords of the circle, see Figure 9.
2. Assign the chords to layers so that edges on the same layer do not cross.
3. Cut the circle between two vertices and open it to form a line of vertices.

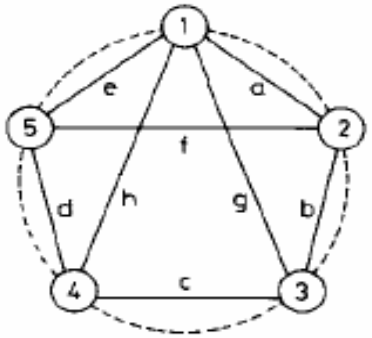

Figure 9: [18] Graph G embedded on a circle.

To embed the graph G in a book we have to solve two main problems [35]:

(i) Finding a Hamiltonian cycle in G or in some edge augmentation of G (that is, a graph obtained by adding new temporary edges) such that the embedding will be optimal. The order of the vertices in the cycle is equal to the order along the spine.

(ii) Assigning the edges of G to a minimum number of pages in some noncrossing manner.

We call the Book Embedding and Crossing Number Problems *fixed or free* depending on whether the vertex ordering on the spine is specified or not.

The *page number* or *book thickness* of G, denoted by $p(G)$ is the smallest k so that G can be drawn on a k-page book with no edge crossings.

The *fixed book thickness* of a graph G is the least integer k such that G can be embedded into k pages under a specified vertex ordering. For example, the fixed book thickness of the graph of Figure 8(a) is 4 if the vertices must be placed on the spine in ascending order of their subscripts (see Figure 10).

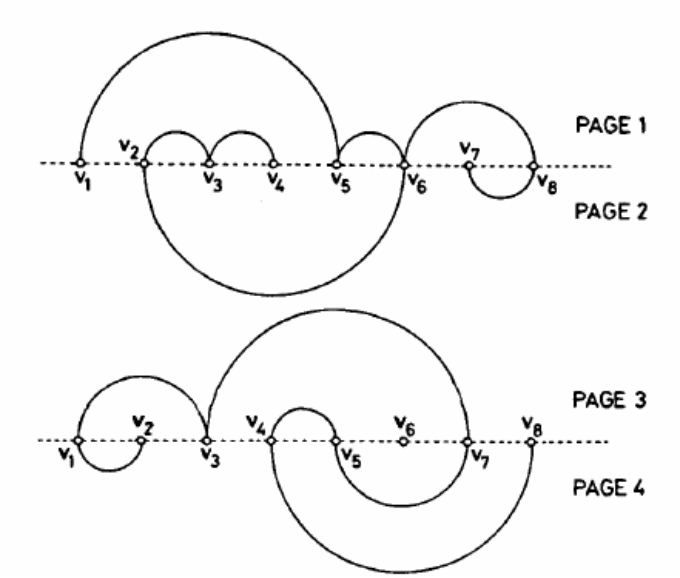

Figure 10: [141] A four page embedding of the graph shown in Figure 8(a).

Graphs with page number one are exactly the outerplanar graphs. Graphs with pagenumber two are the subhamiltonian planar graphs: these are the subgraphs of planar Hamiltonian graphs. As there are triangulated (maximal) planar graphs which are not Hamiltonian, this implies that there are planar graphs which require at least three pages [213].

Berhart and Kainen conjectured that planar graphs have unbounded pagenumber, but this was disproved in [23] and [98]. Yannakakis has shown that the right number is four. That is, he gave an algorithm which embeds all planar graphs in four pages; the algorithm runs in linear time. And conversely, he has shown that there are planar graphs which cannot be embedded in three pages. [213]

The Book Crossing Number Problem is NP-complete [210, 35]. It deals with two subproblems both of which are NP-complete, at least for general graphs [78], the problem of finding a good vertex order and the problem of embedding the edges optimally for a fixed node-embedding.

The Book Embedding Problem is also NP-complete already for a 1-page book. Note that in this case the crux of the problem is in the node-embedding part, as once this is fixed, it is easy to tell whether the edges can be embedded in two pages [210, 35].

Determining the Fixed Book Thickness of graphs is NP-hard, since it is equivalent to Coloring Circle Graphs [35], which was proven to be NP-hard in [78]. Both of the Book Crossing Number and Graph Thickness Problems are NP-complete for the general case [80, 109], and they have the planarity testing problem, which is solvable in linear time [19, 104], as their common subproblem.

2.5.3.1 1-page crossing number

In [35], Chung, Leighton and Rosenberg proved that a graph can be embedded in a *one-page book* if and only if it is *outerplanar*. A graph G is *outerplanar* if its vertices can be placed on a circle so that its edges become noncrossing chords of the circle. If G is *outerplanar* and is laid out on a circle, then cutting the circle between any two vertices and opening it out to form a line yields a *one-page embedding* of G.

A *convex drawing* of G is a rectilinear drawing in which the vertices are placed in the corners of a convex n-gon, see Figure 11. *Convex crossing numbers* (also called *outerplanar crossing numbers*) were first introduced by Kainen [115] in connection with the Book Thickness Problem.

Let $\overline{cr}(G)$ and $cr^*(G)$ denote the rectilinear crossing number and the convex crossing numbers of G, respectively. Clearly $cr(G) \leq \overline{cr}(G) \leq cr^*(G)$. In particular, it is well known that $cr(K_8) = 18 < \overline{cr}(K_8) = 19$. (Note that $cr^*(K_8) = \binom{8}{4} = 70$). In terms of the k-page crossing number cr_k[178, 180], it is obvious that $cr^*(G) = cr_1(G)$ for every graph G.

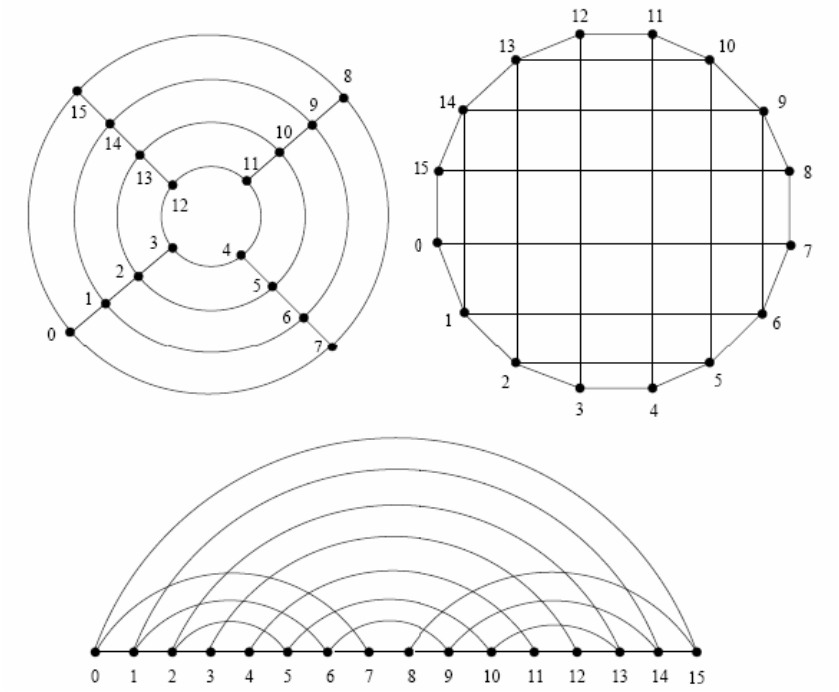

Figure 11: [179] The 4x4 cylindrical grid and its convex and one-page drawings.

Also, in [180] the following result and a greedy algorithm are given for constructing a k-page drawing of G from a 1-page drawing with the indicated number of crossings:

Theorem 6.

$$cr_k(G) \leq \frac{cr_1(G)}{k}$$

2.5.3.2 2-page crossing number

A graph is sub-Hamiltonian if it is embeddable in the plane so that:

1. Its vertices lie on a circle.
2. Each of its edges lies either totally within the circle or totally outside it.
3. No edges cross in the layout.

A graph G admits a two-page embedding if and only if it is sub-Hamiltonian, *i.e.*, a subgraph of a planar Hamiltonian graph [35]. If G is sub-Hamiltonian and is laid out on a circle, then cutting the circle between any two of G' vertices and opening it out yields a planar embedding of G in a line, with each edge lying either totally above the line (on page 1) or totally below it (on page 2).

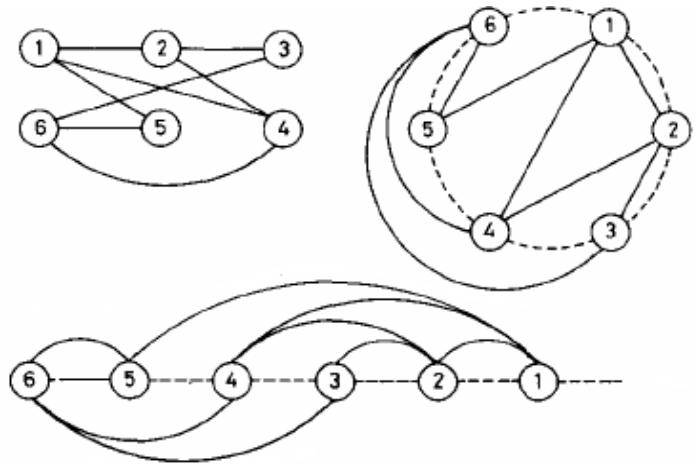

Figure 12: [18] Sub-Hamiltonian graph, its circle representation and book embedding.

Further, a graph G is 2-page free embeddable if and only if there exists a *free linear embedding* of G with no edge crossings. Since testing the 2-page free embeddability of graphs is NP-complete [35], the free linear crossing number problem is NP-complete.

On the other hand, it is easy to show that the 2-page fixed embeddability problem, which is equivalent to the *fixed linear crossing number* problem with $K = 0$, is solvable in linear time.

2.5.3.3 Theoretical bounds

Let $G = (V, E)$, $n = |V|$, and $m = |E|$. Shahrokhi, Székely, Sýkora and Vrťo have shown the following results:

Theorem 7 (**Shahrokhi *et al.***[180]).

$$cr_1(G) \geq m - 2n + 3 \quad \text{for } n \geq 2$$

Theorem 8 (**Shahrokhi *et al.*** [180]).

$$cr_1(G) \geq \frac{m^3}{37n^2} \quad \text{for } n \geq 4, m \geq 3n$$

Theorem 9 (**Shahrokhi *et al.*** [180]).

$$cr_k(G) \geq \frac{m^3}{37n^2} - \frac{27kn}{37}$$

Also, the following result can be deduced:

Theorem 10 (**Cimikowski** [39]).

$$cr_1(K_n) = \frac{n(n-1)(n-2)(n-3)}{24}$$

The Following result for K_n was previously shown in [171] (see also [95, 94]):

Theorem 11 (**Guy, Jenkyns and Schaer** [171]).

$$cr_2(K_n) \leq \frac{1}{4}\lfloor\frac{n}{2}\rfloor\lfloor\frac{n-1}{2}\rfloor\lfloor\frac{n-2}{2}\rfloor\lfloor\frac{n-3}{2}\rfloor$$

Actually, equality has been shown for $n \leq 10$ in the above formula.

In [46], an alternate upper bound based on the adjacency matrix is given for $cr(K_n)$ when drawn on k pages, and tables of results for different n and k values are given. The results for $k = 2$ coincide with those of Theorem 11.

Theorem 12 (**Cimikowski** [39]).

$$\nu_2(G) \leq (\lceil\frac{m}{2}\rceil^2 + \lfloor\frac{m}{2}\rfloor^2 - m)/2$$

2.5.4 Crossing minimization on hierarchical embeddings

A common method for drawing directed acyclic graphs is to produce *layered drawings* or *hierarchical drawings* as introduced by Tomii *et al.* [199], Carpano [30], and Sugiyama *et al.* [121].

In these drawings, the vertices are arranged on two or more "layers", *i.e.*, on parallel horizontal lines, and edges are drawn straight between vertices on adjacent layers. Edges between vertices on the same layer are not permitted, and no point between layers may lie on more than two edges.

Layouts of this kind have applications, for example, in visualization [7], in DNA mapping [207], and in row-based VLSI layout [131]. The readability of layered drawings is believed to depend crucially on the number of edge crossings. Once vertices have been assigned to layers, this number is determined by the orderings of the vertices within the layers.

Unfortunately, the problem of choosing vertex orderings that minimize the number of edge crossings in layered drawings is in fact an NP-complete problem [80] even if there are only two layers [64]. The problem of choosing vertex orderings that minimize the number of edges whose removal leaves the graph planar is also NP-complete, even for two layers [63].

2.5.4.1 k-layer crossing number

A *k-layered graph* or a *k-layer hierarchy* is defined as a graph $G = (V, E) = (V_1, V_2, \ldots, V_k, E)$ with vertex sets $V_1, V_2, \ldots, V_k$, $V = V_1 \cup V_2 \ldots \cup V_k$, $V_i \cap V_j = \emptyset$ for $i \neq j$, and an edge set E connecting vertices in levels V_i and V_j with $i \neq j$ $(1 \leq i, j \leq k)$. V_i is called the *i-th layer*.

In a geometric representation of a *k-layered* graph, the vertices in each level V_i are drawn on a horizontal line L_i, with y-coordinate $k-i$, and the edges are drawn strictly monotone, *i.e.*, an edge $(v_i, v_j) \in E, v_i \in V_i, v_j \in V_j, i < j$, is drawn with decreasing y-coordinates. Essentially, a *k-leveled* graph is a *k-partite* graph that is drawn in a special way.

A *proper k-layered graph* is a *k-leveled graph* $G = (V_1, V_2, \ldots, V_k, E)$ in which any edge in E connects vertices in two consecutive levels V_i and V_{i+1} for $i \in \{1, 2, \ldots, k-1\}$.

The following figure shows a proper layered graph on $k = 4$ layers. This graph represents the face lattice of the Cuboctahedron [147].

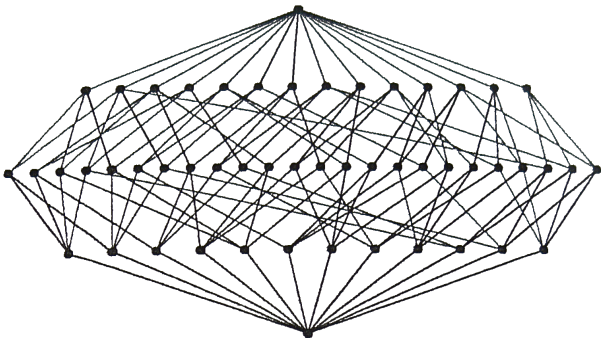

Figure 13: [147] Cuboctahedron

Most techniques for producing layered drawings first assign vertices to layers (sometimes this is determined by the context), and then do a *layer-by-layer sweep.* A permutation Π_1 for the vertices in the top layer L_1 is chosen and fixed. Then for each succeeding layer L_i, a permutation Π_i is sought that keeps to a minimum the number of edge crossings among the edges between $L_{i\text{-}1}$ and L_i.

The k-layer crossing minimization problem as well as the related k-Layer Planarization Problem have been studied from the fixed parameter tractability point of view in [58]. Here k represents the number of layers and planarization means to remove some number h of edges so that the remaining graph can be drawn without crossings (see [63]).

According to [58], bounded pathwidth techniques prove both these general problems are in the class FPT. Unfortunately, the pathwidth-based approach is only of theoretical interest, since the running time of the algorithms is $O(2^{32(k+2h)^3}n)$. In [57], other FPT techniques are used to derive an $O(h6^h + |G|)$ time algorithm for 2-Layer Planarization of a graph G, and an $O(3^h|G|)$ time algorithm for 1-Layer Planarization.

2.5.4.2 Bipartite crossing number

Let $G = (V, E)$, $V = V_0 \cup V_1$ be a bipartite graph with vertex partitions V_0 and V_1. A *bipartite drawing* of G is obtained by placing the vertices of V_0 and V_1 into distinct points on two horizontal lines y_0 and y_1, respectively, and drawing each edge with one straight line segment. Any bipartite drawing of G is identified by two permutations Π_0 and Π_1 of the vertices on y_0 and y_1.

The problem of the *two-sided bipartite* drawing of G is to find permutations Π_0 and Π_1 that minimise the number of pairwise edge crossings in the corresponding bipartite drawing.

Let $bcr(G, \Pi_0, \Pi_1)$ denote the total number of crossings in the bipartite drawing represented by the permutations Π_0 and Π_1. The *bipartite crossing number* of G, denoted by $bcr(G)$, is the minimum number of crossings over all Π_0 and Π_1. Clearly, $bcr(G) = min_{\Pi_0,\Pi_1} bcr(G, \Pi_0, \Pi_1)$.

The problem of the *one-sided bipartite* drawing of G is the same, except the permutation Π_1 is fixed.

Gary and Johnson [80] proved that the two-layer edge crossing minimization problem is NP-hard. More recently, Eades, McKay, and Wormald [64] proved that the problem is NP-hard even if the permutation of nodes on one layer is fixed.

The problem of determining the minimum cardinality set of edges whose removal allows G to be drawn with no crossings is also NP-hard, whether or not the order on one of the layers is fixed [63].

Junger and Mutzel [113] gave an exact integer linear programming algorithm for the 1-Sided Crossing Minimization problem. They also surveyed heuristics and made performance comparisons with optimal solutions generated by their methods. They reported that the *iterated barycentre* method of Sugiyama *et al* [121] performs best in practice.

However, from a theoretical point of view the *median heuristic* of Eades and Wormald [64] is a linear 3-approximation algorithm, whereas the barycentre heuristic is a $\Theta(n^{1/2})$-approximation algorithm. The most recent heuristic based on the computation of feedback arc sets and experimental results has been reported in [51].

When only a small number, c, of edge crossings is acceptable, then an algorithm for 1-Sided Crossing Minimization whose running time is exponential in c but polynomial in the size of the graph may be useful.

The crossing minimization on hierarchical embeddings has been extensively studied and surveyed in the literature, see for example [57, 58, 64, 184, 183, 151, 147]. Hence, it is not the purpose of this work to study the algorithmic approaches for computing the crossing number for hierarchical embeddings of graphs any further.

3 Algorithms for the general crossing number

It is well known that the general crossing minimization problem is *NP*-hard [80]. More precisely, it is shown that the *crossing number problem, i.e., "given a graph G and a non-negative integer k, decide whether there is a drawing of G with at most k edge crossings"*, is NP-complete.

However, for every k, there is a simple polynomial time algorithm deciding whether a given graph G has crossing number at most k: It guesses $l \leq k$ pairs of edges that cross and tests if the graph obtained from G by adding a new vertex at each of these edge crossings is planar. This can be implemented by exhaustive search of the space of m^{2k} k-tuples of edge pairs, where m denotes the number of edges of the input graph [88]. The running time of this algorithm is $n^{\Theta(k)}$. Clearly, this algorithm is not appropriate in practical applications for larger values of k.

Grohe [88] has given an exact algorithm that works in time $O(|V|^2)$ if the crossing number is fixed. Even though the exponent is independent of k, the constant factor of this algorithm grows doubly exponentially in k. Also this algorithm is rather of theoretical nature and has so far not been useful for solving practical instances.

More promising are branch-and-cut algorithms using mathematical programming formulations for computing the crossing number. In the next chapter, we discuss the exact algorithms in further detail.

While there is no known polynomial time approximation algorithm with any type of quality guarantee for the general problem, Bhatt and Leighton could derive an algorithm for graphs with *bounded degree* that approximates the number of crossings *plus the number of nodes* in polynomial time [13]. This was later improved by Even, Guha and Schieber [68], see chapter 5.

Heuristic approaches for solving the general crossing number problem are attended in the chapter 6. The most prominent and practically successful approach for solving the general crossing minimization problem heuristically is the *planarization approach* [5], which addresses the problem by a two step strategy.

In a first step, a preferably small number of edges is deleted from $G = (V, E)$ in order to obtain a planar graph P. In a second step, the edges are re-inserted into the planar graph P while trying to keep the number of crossings small. Both of these steps are NP-hard.

For each step, various algorithms can be applied. For the second step, pre- and post-processing procedures have been developed to improve the solution quality. A computational study on "state-of-the-art" of these heuristics can be found in [91]. We study the two steps of the planarization method in chapters 7 and 8.

In chapters 9 and 10, we survey algorithms for solving the rectilinear and (fixed) linear crossing number problems, respectively, including approaches based on genetic and neural network models of computation.

In chapter 10, we also give an account of our experimental study, which we conducted in order to compare our proposed heuristics based on a general simulated annealing scheme with the existing heuristics for the (fixed) linear crossing number.

4 Exact algorithms

Grohe designed an exact quadratic time algorithm for computing the general crossing number for a fixed k. Although it is not suited for practical use, we give a brief account of his attempt in the beginning of this section.

In the next sections 4.2 and 4.3, we explore some more promising approaches based on the branch-and-bound techniques combined with depth first search and/or mathematical programming formulations.

4.1 Quadratic time algorithm

Grohe [88] proved, that the crossing number problem was fixed-parameter tractable and constructed a quadratic time algorithm deciding whether a given graph G has crossing number at most k. He also showed that if this is the case, a drawing of G in the plane with at most k crossings can also be computed in quadratic time.

As for the crossing number, it is NP-complete to decide if the *genus* of a given graph is less than or equal to a given k [192]. However, the fact that the *genus* problem is fixed-parameter tractable was known earlier as a direct consequence of a strong general theorem due to Robertson and Seymour [174] stating that all minor closed classes of graphs are recognizable in cubic time.

Unfortunately, while the class of graphs of genus at most k is closed under taking minors, the class of all graphs of crossing number at most k is not. So although Grohe could not apply Robertson and Seymour's result directly, the overall strategy of his algorithm is inspired by their ideas: The algorithm first iteratively reduces the size of the input graph until it reaches a graph of bounded tree-width, and then solves the problem on this graph.

For the reduction step, Grohe used Robertson and Seymour's Excluded Grid Theorem [173] together with a nice observation due to Thomassen [194] that in a graph of bounded genus (and thus in a graph of bounded crossing number) every large grid contains a subgrid that, in some precise sense, lies "flat" in the graph. Such a flat grid does not essentially contribute to the crossing number and can therefore be contracted.

For the remaining problem on graphs of bounded tree-width Grohe applied a theorem due to Courcelle [43] stating that all properties of graphs that are expressible in monadic second-order logic are decidable in linear time on graphs of bounded tree-width.

Of course, it is not surprising that Grohe's algorithm is double-exponential in k, which makes it unsuitable for practical use.

4.2 Depth First Search with Branch-and-Bound

If the solution space of a problem can be mapped to a tree, where each interior vertex is a partial solution, edges toward the leaves are options that refine the partial solution, and the leaves are complete solutions, then there are various algorithms that can search the tree to find the optimal solution.

A Depth First Search (DFS) algorithm is one such algorithm which, as its name implies, searches more deeply into the tree for a solution whenever possible. Once a path is found from the root to a leaf representing a solution, the search backtracks to explore

the nearest unsearched portion of the tree. This continues until the entire tree has been traversed.

The Branch and Bound portion allows us to change one simple part of the DFS algorithm. When the cost to get to a vertex v exceeds the current optimal solution, we then tell the DFS algorithm not to traverse the subtree having v as its root.

This method exhaustively covers the entire search space even after finding an initial solution. However, it does not cover those sections of the search space that lead to solutions that are guaranteed to cost more than the current optimal solution. When the entire tree is covered the current optimal solution is the globally optimal solution.

F. and C. Harris [97] created a branch-and-bound algorithm for finding the minimum crossing number of a graph. The algorithm begins with the vertex set and adds edges by selecting every legal option for creating a crossing or not.

After each edge is added a subroutine called the Rotational Embedding Scheme counts the regions of the resulting embedding and using Euler's formula, determines if the given graph has a planar embedding. A code for this routine can be found in [137].

The algorithm continues adding edges until either all edges have been added or it reaches a point where the graph cannot be completed as started, because the number of crossings would exceed the current optimum. At this point it backtracks to see if the graph can be drawn with fewer crossings by selecting other options when adding edges.

Tadijev and F. Harris [187] constructed a preliminary parallel version of the above sequential algorithm. In order to obtain some initial results they did a basic static partitioning of the search tree among the p processors in their parallel machine. This method, along with its benefits and drawbacks, is discussed in detail in [122].

As a next step they intended to modify the implementation to have dynamic partitioning of the search space.

4.3 Mathematical programming formulations

Mathematical programming is a powerful tool to address NP-hard combinatorial optimization problems. Starting from an integer linear program (ILP) modeling the problem under consideration, *i.e.*, a linear program with integer variables, sophisticated techniques like branch-and-cut can be applied.

In [21], Buchheim, Ebner, Jünger, Klau, Mutzel and Weiskircher present the first algorithm able to compute the crossing number of general sparse graphs of moderate size. They state computational results on a popular benchmark set of graphs, the so-called Rome library [6].

The approach uses a new integer linear programming formulation of the problem combined with efficient heuristics and problem reduction techniques. Thus, they managed to compute the crossing number for 91% of all graphs on up to 40 nodes in the Rome library within a time limit of five minutes per graph.

In graph drawing it is often desirable to optimize some cost function over all possible embeddings in a planar graph. In general these optimization problems are NP-hard [82]. In the planarization method, the number of crossings highly depends on the chosen embedding when the deleted edges are reinserted into a planar drawing of the rest-graph.

The problem can be formulated as a flow problem in the geometric dual graph [148]. A flow between vertices in the geometric dual graph corresponds to a flow between adjacent face cycles in the primal graph. Once we have characterized the set of all feasible embeddings (via an integer linear formulation on the variables associated with each cycle), we can use it in an ILP-formulation for the corresponding flow problem. Here, the variables consist of "flow variables" and "embedding variables".

In [148], Mutzel and Weiskircher introduced an integer linear program whose set of feasible solutions corresponds to the set of all possible combinatorial embeddings of a given biconnected planar graph.

One way of constructing such an integer linear program is by using the fact that every combinatorial embedding corresponds to a 2-fold complete set of circuits (see MacLane [138]). The variables in such a program are all simple cycles in the graph; the constraints guarantee that the chosen subset of all simple cycles is complete and that no edge of the graph appears in more than two simple cycles of the subset.

[148] have chosen another way of formulating the problem. They only introduce variables for those simple cycles that form the boundary of a face in at least one combinatorial embedding of the graph, thus reducing the number of variables significantly.

Furhtermore, [148] derive the constraints of their ILP using the structure of the graph by constructing the program recursively using an SPQR-tree. The SPQR-tree data structure can be used to code and enumerate all possible combinatorial embeddings of a biconnected planar graph [10].

4.3.1 Integer Linear Program for simple drawings

Let $G = (V, E)$ be a graph and let D be a set of unordered pairs of edges of G. We call D *simple* if for every $e \in E$ there is at most one $f \in E$ such that $(e, f) \in D$. Furthermore, D is called *realizable* if there is a drawing of G such that there is a crossing between edges e and f if and only if $(e, f) \in D$.

For every graph G and every simple D, Buchheim, Ebner, Jünger, Klau, Mutzel and Weiskircher [21] denote with G_D the graph that is obtained by introducing a dummy node $d_{e,f}$ for each pair of edges $(e, f) \in D$. Note that G_D is only well-defined if D is simple, as otherwise it would not be clear where to place the dummy nodes. For both edges e_1 and e_2 resulting from splitting e, we set $\overline{e}_1 = \overline{e}_2 = e$, analogously for f.

Corollary 1 (Buchheim *et al.* [21]). *[21] Let D be simple. Then D is realizable if and only if G_D is planar.*

Using a linear time planarity testing and embedding algorithm, we can thus test in time $O(|V| + |D|)$ whether D is realizable, and compute a realizing drawing in the affirmative case.

Definition 3. *For a set of pairs of edges $D \subseteq E^2$, define*

$$x^D_{e,f} = \begin{cases} 1 & \text{if} (e, f) \in D \\ 0 & \text{otherwise} \end{cases}.$$

Next, for every subgraph $H = (V', E')$ of G_D, let $\overline{H} = \{e | e \in E'\} \subseteq E$. Less formally, $\overline{H}$ contains all edges of G involved in the subgraph H of G_D.

Proposition 1 (**Buchheim *et al.*** [21]). *Let D be simple and realizable. For an arbitrary simple set of pairs of edges $D' \subset E^2$ of $G = (V, E)$ and any subdivision H of K_5 or $K_{3,3}$ in $G_{D'}$, the following inequality holds:*

$$C_{D',H} : \Sigma_{(e,f)\in\overline{H}^2/D'} x^D_{e,f} \geq 1 - \Sigma_{(e,f)\in H^2\cap D'}(1 - x^D_{e,f}).$$

If we only consider the subgraph induced by H, it follows that $\overline{H}^2 \cap D' = \overline{H}^2 \cap D$(see Definition 3). This means that the edges $(e, f) \in H^2$ cross in respect of D' if and only if they cross in respect of D and H is also a "forbidden" subgraph in G_D, i.e., a subdivision of K_5 or $K_{3,3}$. It follows from Kuratowski's Theorem that G_D is not planar. This contradicts the realizability of D by Corollary 1.

Theorem 13 (**Buchheim *et al.*** [21]). *Let $G = (V, E)$ be a simple graph. A set of pairs of edges $D \subseteq E^2$ is simple and realizable if and only if the following conditions hold:*

$x^D_{e,f} \in \{0, 1\}$	$\forall e, f \in E,\ e \neq f$
$\Sigma x^D_{e,f} \leq 1$	$\forall e \in E$
$C_{D',H}$	*for every simple $D' \subseteq E^2$ and every forbidden subgraph H in $G_{D'}$*

For every simple and realizable set $D \subseteq E^2$, we can compute a corresponding drawing in polynomial time. Thus we can reformulate the crossing minimization problem for simple drawings as: *"Given a graph $G = (V, E)$, find a simple and realizable subset $D \subseteq E^2$ of minimum cardinality."* This immediately leads to the following ILP-formulation, where we use $x(F)$ as an abbreviation for the term $\Sigma_{(e,f)\in F} x_{e,f}$:

min $x(E^2)$ s.t.

$\Sigma x^D_{e,f} \leq 1$	$\forall (e, f) \in E$
$x(H^2 \backslash D') - x(H^2 \cap D') \geq 1 - \lvert H^2 \cap D' \rvert$	for every simple D' and every forbidden subgraph H in $C_{D'}$
$x^D_{e,f} \in \{0, 1\}$	$\forall (e, f) \in E$

It is clearly impractical to generate all constraints $\mathrm{C_{D,H}}$ in advance and solve the *ILP* in a single step. Instead, Buchheim *et al.* embed the given formulation into a branch-and-cut framework, separating violated inequalities dynamically during runtime.

A crucial factor in this approach is the *separation problem: "Given a class of valid inequalities and a vector $y \in R^n$, either prove that y satisfies all inequalities in the class, or find an inequality which is violated by y."* Although, it is easy to separate violated inequalities for integral solution vectors, the problem is more complex within the branch-and-cut framework since we have to deal with fractional values.

A heuristic for separating the inequalities is to round variables to either zero or one, and then check for violated inequalities. The problem is that the inequalities produced by this heuristic might not be violated by the current fractional solution. In this case Buchheim *et al.* select a branching variable and split the current problem into two subproblems by setting the branching variable to zero, respectively one. The same is done if no inequalities at all are produced by the separation heuristic.

In some cases, variables can be omitted from the ILP. For instance, when splitting the graph into its *blocks* (two-connected components) and solving these blocks independently - the crossing number of a graph is equal to the sum of the crossing numbers of its blocks.

Furthermore, it is obvious that adjacent edges do not cross in an optimal drawing and no edge crosses itself, which makes it possible to restrict only to good drawings.

The results of their computational study show that Buchheim *et al.* [21] have improved the heuristic results for the basic planarization approach, even for the relatively small instances considered in their study. Compared to the best known heuristic methods, they achieved a notable improvement for some larger instances. The average improvement over the whole considered benchmark set was about 19.6% for the basic heuristic and 4.1% for the best known strategy.

4.3.2 Integer Linear Program for SPQR-trees

The SPQR-tree represents the decomposition of a biconnected planar graph with respect to its triconnected components. All embeddings of the graph can be enumerated by enumerating all possible embeddings of these components in respect to the rest of the graph.

The approach of Mutzel and Weiskircher [148] uses the fact that each skeleton of a node in the SPQR-tree represents a simplified version of the original graph. By computing the integer linear programs (ILP) for these simple graphs and using a lifting procedure, it is possible to compute an ILP for the original graph.

The skeleton of a node in the SPQR-tree, can be constructed from the original graph by replacing one or several subgraphs by single edges, which are called *split edges*. Such an edge is representing the set of all the simple paths in the original graph, that connect the two nodes of the split edge. So every circle in a skeleton that includes a split edge represents a set of circles in the original graph that we get by replacing the split edge with the paths represented by it.

The variables of the program correspond to directed circles in the graph that are face cycles in at least one planar embedding, because the recursive construction computes the set of variables for the original problem using the sets of variables from subproblems for which the ILP has already been computed. So circles in the original graph are constructed from circles in the subproblems by replacing split edges with paths in the graph.

According to Mutzel and Weiskircher [148], every feasible solution of the generated ILP corresponds to a combinatorial embedding of the given biconnected planar graph G and vice versa: every combinatorial embedding of G corresponds to a feasible solution for the generated ILP. For the full details on the recursive construction of the integer linear program see [148].

The computational results on two benchmark sets of graphs have been quite surprising. Despite expected that the size of the linear system will grow exponentially with the size of the graph, the number of constraints and variables grew only linearly.

However, the time for generating the system grew subexponentially: but for practical instances it was still reasonable. For a graph with 500 vertices and 10^{19} different combinatorial embeddings the construction of the ILP took about 10 minutes. Very surprising was the fact that the solution of the generated ILPs took only up to 2 seconds using CPLEX.

5 Approximation algorithms

Concerning crossing numbers of standard graphs, there are only a few infinite classes of graphs for which exact or tight bounds are known [133]. The main problem is the lack of efficient lower bound methods for estimating the crossing numbers of explicitly given graphs [DV03]. A survey on known methods is given in [88, 39, 178].

One of the powerful methods is based on the bisection width concept. The bisection width of a graph G is the minimum number of edges whose removal divides G into two parts having at most $\frac{2}{3}|V|$ vertices each. Leighton [127] proved that in any n-vertex graph G of bounded degree, the crossing number satisfies $cr(G) + n = \Omega(bw^2(G))$, where $cr(G) + n$ defines the minimum drawing size of the graph.

Bhatt and Leighton, in [13], apply a $B(n)$-approximate bisection procedure recursively to *decompose* a bounded degree graph with n vertices. They prove that this recursive decomposition induces a drawing of size $O(B^2(n)log^2n)$ times the minimum size drawing. Shahrokhi *et at.* [178] considered straight line drawings induced by decomposition trees and presented a simpler construction of a drawing of the same size as the one achieved in [13].

Leighton and Rao [128] showed that the above result can be realized with a $(\frac{1}{3}, \frac{2}{3})$ separator. Leighton and Rao also showed how to find such a separator with size bounded by $\alpha(n) = O(log\ n)$ times the optimal bisector. This implied an $O(log^4n)$ or $O(\alpha^2(n)log^2n)$ approximation algorithm for the drawing size of a bounded degree graph. The bound on the approximation factor in this algorithm relies only on the fact that an optimal drawing induces a planar graph which admits small vertex separators.

Even, Guha, and Schieber [68] improved the approximation factor to log^3n. There is known nothing that would exclude the possibility of approximation within a constant multiplicative factor [186].

The reason that the approximation algorithm applies only to bounded degree graphs is that, in bounded degree graphs, the vertex separators and the edge separators have roughly the same size. No non-trivial approximation algorithms are known for the general case of arbitrary degrees. From now on we consider only bounded degree graphs for the drawing problem.

5.1 Approximation algorithm with estimators

A *decomposition tree* of a graph G is a tree together with a one-to-one correspondence between its leaves and the vertices of G. Each internal tree node is associated with the cut separating the vertex sets mapped to the sets of leaves of the subtrees rooted at its children.

Bhatt and Leighton, in [13], proposed a special type of decomposition trees, called *bifurcators*. These decomposition trees are binary trees, and the cut sizes associated with their nodes decrease exponentially relative to the cut size of the root. Using the approximation of drawing size, Bhatt and Leighton provided an $O(log^{2.5}n)$ approximation for the optimal $\sqrt{2}$-bifurcator.

Even, Guha and Schieber [68] first provide an $O(log^3n)$ approximation for the drawing size problem. As a consequence of this result, based on Bhatt and Leighton's results,

Even *et al.* obtain an $O(log^2 n)$ approximation for the optimal $\sqrt{2}$-bifurcator problem and a corresponding improvement for all its applications.

Let $\alpha(n)$ denote the smallest known ratio of the separator size (that can be computed efficiently) to the optimal bisector size. Hence, the algorithm by Even *et al.* can be stated as an $O(\alpha^2(n) log\ n)$-approximation of the minimum drawing size.

Even *et al.*'s approximation algorithm [68] constructs a decomposition tree T that can be viewed as an approximation of a decomposition tree $\overline{T}$ obtained by recursively bisecting the planar graph induced by an optimal drawing. Such a decomposition tree $\overline{T}$ induces a drawing of size $O(log\ n)$ times the optimal drawing size.

The decomposition tree computed by Even *et al.* mimics the useful properties of $\overline{T}$. In the problem of drawing a graph on the plane, Even *et al.* attach an estimator $\phi(t)$ to every tree node that estimates the optimal drawing size of the corresponding subgraph. The estimator quality is one-sided; it may not surpass (twice) the drawing size, but might be much smaller than the drawing size.

The estimators have the following two properties that follow the properties of the drawing sizes of the subgraphs in $\overline{T}$: (a) the cut sizes are bounded by the square root of the corresponding estimators upto an logarithmic error term; and (b) the estimators decrease exponentially as one goes down the tree.

The main difficulty in constructing such an "approximated" decomposition tree with estimators is that the only tool available are approximate separators [68].

A bottom-up approach fails because merging can take place only when the subgraphs have comparable drawing sizes. Estimates on drawing sizes are based on lower bounds derived from separators. Such a lower bound would need to be based on the subgraph with the largest separator; however, finding such a subgraph is computationally prohibitive. A naive top-down approach of recursive separators fails since the partitioning is not guaranteed to divide the drawing sizes in a balanced fashion, and hence, the estimators may not decay exponentially.

Therefore, Even *et al.* applied a top-down approach with re-balancing to guarantee that the estimators decrease exponentially.

6 Heuristic algorithms

A practically very successful and important approach for solving the general crossing minimization problem heuristically is the *planarization approach* [5], which addresses the problem by a two step strategy.

In the first step, a preferably small number of edges is deleted from $G = (V, E)$ in order to obtain a planar graph P. In the second step, the edges are re-inserted into the planar graph P while trying to keep the number of crossings small.

Both of these steps are NP-hard. For each step, various algorithms can be applied. Pre- and post-processing procedures have been developed to improve the solution quality. The planarization approach will be studied thoroughly in chapters 7 and 8.

Simulated annealing (SA) is a flexible optimization method, suited for large-scale combinatorial optimization problems. It has been applied successfully to classical combinatorial optimization problems, such as the traveling salesman problem, and problems concerning the design and layout of VLSI.

SA differs from standard iterative improvement methods by allowing "uphill" moves-moves that spoil, rather than improve, the temporary solution. The rest of this chapter discusses the application of SA to the crossing number problem.

6.1 Simulated annealing

The problems for which SA is useful are characterized by a very large discrete configuration space, too large for an exhaustive search, over which an objective cost function is to be minimized (or maximized) [47].

After picking some initial configuration, most iterative methods continue by choosing a new configuration at each step, evaluating it and possibly replacing the previous one with it. This action is repeated until some termination condition is satisfied (e.g., no move reduces the objective function). The procedure ends in a minimum configuration, but generally it is a local minimum, rather than the desired global minimum. The SA method tries to escape from these local minima by using rules that are derived from an analogy to the process in which liquids are cooled to a crystalline form, a process called *annealing.*

Metropolis *et al.* [144] devised an algorithm for simulating this annealing procedure by a series of sequential moves. The basic rule is that the probability with which the system changes its state from one with energy E_1 to one with energy E_2 is: $e^{-\frac{E_2-E_1}{kT}}$. This rule implies that whenever the energy $E2$ of the new candidate state is smaller than the current energy El the system will take the move, and if it is larger the state change is probabilistic.

Algorithm 6.1. Schematic form of the SA method

1. get an initial configuration σ and an initial temperature T;
2. **while** *stop criterion* is not satisfied **do**
    1. **while** *inner loop criterion* is not satisfied **do**
        1. choose a new configuration σ' from the neighborhood of σ;
        2. let E and E' be the values of the cost function at σ and σ' respectively;
        3. **if** $E' < E$, **set** $\sigma \leftarrow \sigma'$

4. **if** $E' \geq E$, **set** $\sigma \leftarrow \sigma'$ with probability $e^{\frac{(E-E')}{T}}$

2. decrease the temperature T;

3. **return** σ

Kirkpatrick *et al.* [118] were apparently the first to realize that the above procedure could be used for general optimization problems.

In [47], Davidson and Harel pointed out SA is not always suitable for a given optimization problem. A basic requirement from the cost function is that it should not have overly steep descents, that is, the width of the valleys (around the minima points) should be roughly proportional to their depth. A very narrow valley that contains the desired minimum point has a low probability of being found.

To be amenable to solution by SA, the problem must also admit near optimal solutions that are acceptable. SA is very successful in finding minima values that are close to the global minimum, but seldom does it detect the global minimum itself.

Another important requirement is that the cost function should be easily calculated for a new configuration (possibly using the value for the old one). This is due to the fact that these calculations are carried out repeatedly and constitute the major computational task in implementing SA.

6.1.1 Simulated annealing for complete graphs

In [170], Ringenburg described a method for attempting to find optimal or nearly optimal crossing numbers of complete graphs using simulated annealing. He has shown that this method has polynomial time and space requirements, and yet still achieves very close to optimal results on this NP-Complete problem.

Ringenburg suggested that for the purposes of finding the optimal crossing number, any drawing of a complete graph can be represented simply by the set of edge pairs which cross. Since any crossing between two edges sharing a vertex can be trivially removed, we need only consider pairs of edges which share no vertices.

Any such pair will clearly contain four distinct vertices. There are $\binom{n}{4}$ such sets of vertices in an n vertex complete graph, and there are 3 distinct ways to pair them off: $\{(1,2),(3,4)\}$, $\{(1,3),(2,4)\}$ and $\{(1,4),(2,3)\}$. Thus one needs to consider $B = 3\binom{n}{4}$ pairs of edges. Since each pair either crosses or does not cross (multiple crossings can be trivially removed), we can easily represent a drawing of a graph as a B-bit vector.

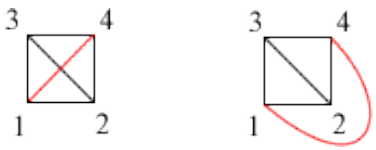

Figure 14: [170] Wrapping edge (2, 3) around vertex 4.

We can transform one drawing of a graph to another by a series of operations involving wrapping an edge of the graph around a non-incident vertex, as shown in figure 14 above. This will have the effect of reversing whether or not the edge in question crosses each edge

incident on the vertex being wrapped around and not incident on either of the vertices of the wrapped edge.

In the bit vector representation, this is equivalent to a bitwise XOR with a bit vector which has l's in the bits representing whether or not the wrapped edge crosses each of the edges incident on the wrap around vertex. Ringenburg referred to these bit vectors representing the wrapping of an edge as "perturbation vectors". Note that for any of the n vertices, there are $\binom{n-1}{2}$ non-incident edges which can be wrapped around it. Thus there are $n\binom{n-1}{2}$ perturbation vectors.

This representation allows to search the space of drawings of a complete graph by starting with a bit vector representation of some drawing of the graph, and then XOR-ing it with various sequences of perturbation vectors. Note that the crossing number problem has essentially been reduced to a coding theory problem. Searching for the lowest crossing number is equivalent to searching for the minimum weight vector. For implementation details refer to [170].

The biggest limiting factor to the algorithm [170] seemed to be the space requirements of the table of perturbation vectors. There are $n\binom{n-1}{2}$ vectors, each of which is $3\binom{n}{4}$ bits. This works out to $\theta(n^7)$. The running time of the algorithm is essentially the number of inner loop iterations times the number ot outer loop iterations. The inner loop iterations are proportional to the neighborhood size, which is $n\binom{n-1}{2}$. Thus the inner loop is $\theta(n^3)$. The outer loop stops iterating when the results stabilize. Intuitively it would seem that this should be at most polynomial, and the time trials performed by Ringenburg seem to confirm this. Thus supposedly the total running time is polynomial.

The experimental results performed by [170] show that optimal or nearly optimal results to the crossing number problem can be found relatively quickly using simulated annealing. Interestingly, Ringenburg observed that the algorithm almost always finds a drawing with the predicted number of crossings if the number of vertices is odd. However when the number of vertices is even the algonihm tends to find results in a narrow range above the predicted value.

7 Maximum planar subgraph problem

The heuristic planarization approach for solving the crossing number problem consists of two steps. In the first step, a maximum planar subgraph is computed by deleting the minimum number of edges of the original graph untill a planar subgraph is obtained. In the second step, we try to reinsert all the edges deleted in the first step so that the resulting number of crossings is small. Both these steps are NP-hard [135, 214].

In the beginning of this chapter, we define the *Maximum Planar Subgraph Problem* and present some theoretical results concerning its complexity. Thereafter, we give a brief account of various methods for computing the Maximum Planar Subgraph Problem in sections 7.2 to 7.4. Another survey of algorithms for graph planarization through edge deletion can be found in, *e.g.*, Liebers [133] or Mutzel [146].

7.1 Introduction

If a graph $G = (V, E)$ with an edge $e \in E$ is transformed into a graph $G' = (V, E \backslash \{e\})$ then we say that G' was obtained from G by edge deletion. By repeatedly deleting edges from a given nonplanar graph G, G can be transformed into a planar graph G'. In this section, we are interested in planarizing G by deleting as few edges as possible [7].

Definition 4 (Maximum Planar Subgraph, Skewness). *If a graph $G' = (V, E')$ is a planar subgraph of a graph $G = (V, E)$ such that there is no planar subgraph $G'' = (V, E'')$ of G with $|E''| > |E'|$, then G' is called a maximum planar subgraph of G, and the number of deleted edges, $|E| - |E'|$, is called the skewness of G.*

So the skewness of a graph G is 0 if and only if G is planar. For some graph classes, the skewness is known: The complete graph K_n has $\frac{n(n-1)}{2}$ edges. For $n \geq 3$, it has a planar subgraph with $3n - 6$ edges. Since a planar graph with $n \geq 3$ vertices cannot have more than $3n-6$ edges, the skewness of the complete graph K_n is $\frac{n(n-1)}{2} - (3n-6) = \frac{(n-3)(n-4)}{2}$ for $n \geq 3$. A similar argument shows that the skewness of the complete bipartite graph K_{n_1,n_2} is $n_1 n_2 - 2(n_1 + n_2) + 4$ for $n_1 \geq 2$ and $n_2 \geq 2$ [41].

Definition 5 (Maximal Planar Subgraph). *If a graph $G' = (V, E')$ is a planar subgraph of a graph $G = (V, E)$ such that every graph $G'' \in \{(V, E' \cup \{e\}) | e \in E \backslash E'\}$ is nonplanar, then G' is called a maximal planar subgraph of G.*

In other words a maximal planar subgraph is maximal with respect to inclusion of its edge set, whereas a maximum planar subgraph is maximal with respect to the cardinality of its edge set. Observe that every maximum planar subgraph is also a maximal planar subgraph, but not vice versa. Figure 15 illustrates maximal and maximum planar subgraphs.

Finding a maximum planar subgraph is an NP-hard problem, and Section 7.1.1 discusses this result. But a maximal planar subgraph can be found in polynomial time, as we will see in Section 7.3. Sections 7.2, 7.3, and 7.4 discuss exact, approximative, and heuristic approaches for finding a large planar subgraph. We also consider the weighted version where edges are assigned nonnegative edge weights, and where the goal is to find a planar subgraph with total edge weight as large as possible.

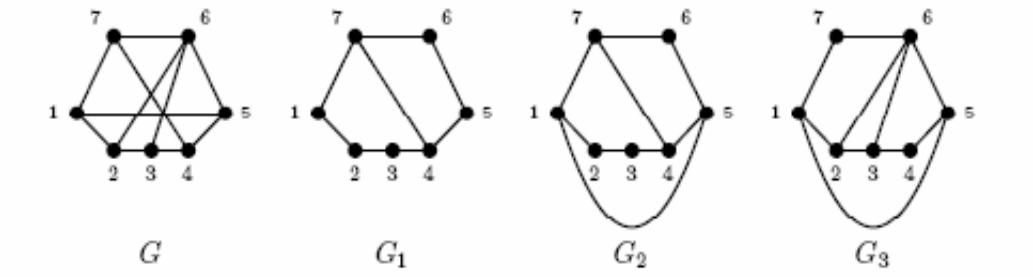

Figure 15: [133] G is a nonplanar graph. Note that G contains $K_{3,3}$ as a minor (contract edge $(2,3)$). G_1 is a planar subgraph of G, but it is not a maximal planar subgraph: Edge $(1,5)$ can be added to G_1 without destroying planarity. The result is G_2. Another maximal planar subgraph of G is G_3. G_3 is also a maximum planar subgraph.

7.1.1 Complexity of the maximum planar subgraph problem

Problem 5 (**Maximum Planar Subgraph** [79]). *Given a graph $G = (V, E)$ and a positive integer $k \leq |E|$, is there a subset $E' \subseteq E$ with $|E'| \geq k$ such that the graph $G' = (V, E')$ is planar?*

Liu and Geldmacher [136], and, independently, Yannakakis [212], and, also independently, Watanabe, Ae and Nakamura [206] showed that this problem is NP-complete. The proof of Liu and Geldmacher is a two step reduction using the following problems:

Problem 6 (**Vertex Cover** [79]). *Given a graph $G = (V, E)$ and a positive integer $k \leq |V|$, is there a vertex cover of size k or less for G, i.e. is there a subset $V' \subseteq V$ of vertices with $|V'| \leq k$ such that for each edge $uv \in E$ at least one of its end vertices u and v belongs to V'?*

Problem 7 (**Hamilton Path in Graphs Without Triangles**). *Given a graph $G = (V, E)$ that does not contain a cycle of length 3, and given two vertices $u \in V$ and $v \in V$, does G contain a Hamilton path from u to v?*

Karp [117] shows Vertex Cover to be NP-complete. Liu and Geldmacher [136] first reduces Vertex Cover to Hamilton Path in Graphs Without Triangles, and then reduces this problem to Maximum Planar Subgraph. Recently, Faria, Figueiredo, and Mendonça [69, 70] have shown that Maximum Planar Subgraph is even NP-complete for cubic graphs.

We should also mention a simplified proof of the NP–Completeness of the Maximum Planar Subgraph Problem (MPS). The proof presented in [29] is considerably shorter and simpler, proceeding through a straight reduction from the MPS to the Bipartite Hamiltonian Circuit Problem [79].

The problem can be solved in polynomial time if G is already planar, since planarity testing can be done in linear time [104]. If $G = K_n$, the complete graph on n nodes, or $G = K_{m,n}$, the complete bipartite graph on $n+m$ nodes, it is easy to construct a solution which contains $3n-6$, resp. $2n-4$, edges. Since Euler showed that the number of edges in a planar graph on n nodes cannot exceed $3n-6$, resp. $2n-4$, [111] have solved the unweighted problem in linear time.

Problem 8 (Weighted Maximum Planar Subgraph). *Given a graph $G = (V, E)$ with a nonnegative edge weight $w(e)$ for each edge e, and a positive number k, is there a subset $E' \subseteq E$ with $\sum_{e \in E'} w(e) \geq k$ such that the graph $G' = (V, E')$ is planar?*

In other words, given a nonplanar weighted graph with edge weights $w(e)$ for $e \in E$, we want to delete a set of edges F to obtain a planar subgraph $G' = (V', E \backslash F)$ such that the sum of all edge weights $\sum_{e \in E \backslash F} w_e$ of G' is maximum. In the unweighted case, where $w(e) = 1$ for all edges $e \in E$, the problem consists of finding the minimum number of edges whose deletion from a nonplanar graph gives a planar subgraph.

Being a generalization of the Maximum Planar Subgraph Problem, Weighted Maximum Planar Subgraph is NP-complete as well [79].

7.2 Exact algorithms

For finding the maximum planar subgraph, exact solution methods like *Branch and Bound* [75] and *Branch and Cut* [112] have been proposed in the literature. Branch-and-bound algorithms only have a chance on small dense graphs [75].

Jünger and Mutzel [111] presented a branch-and-cut algorithm using facet-defining inequalities for $PLS(G)$ (the *planar subgraph polytope* of graph G) as cutting planes. In a cutting plane algorithm, a sequence of relaxations is solved by linear programming.

After the solution x of some relaxation is found, we must be able to check whether x is the incidence vector of a planar subgraph (in which case we have solved the problem) or whether any of the known facet-defining inequalities are violated by x. If no such inequalities can be found, we cannot tighten the relaxation and have to resort to branching, otherwise we tighten the relaxation by all facet-defining inequalities violated by x which we can find. Then the new relaxation is solved, *etc.* The process of finding violated inequalities (if possible) is called "separation" or "cutting plane generation".

If the number of edges to be deleted is small, this approach gives quite good, and in many cases provably optimal, solutions for sparse graphs and very dense graphs. However, the method is quite complicated to understand and to implement. Moreover, if the number of deleted edges exceeds 10, the algorithm usually needs far too long to be acceptable for practical computation. Interested readers are refered to the study of Ziegler [214] concerning the number of deleted edges in the *Rome library* benchmark set.

7.2.1 A branch-and-cut algorithm for MPS based on LP

Jünger and Mutzel [111] designed a branch-and-cut algorithm based on polyhedral combinatorics [165], a subfield of combinatorial optimization which aims at describing combinatorial optimization problems as linear programs and solving these with special purpose methods.

Jünger and Mutzel [111] define a polytope of all planar subgraphs of a graph G. All subgraphs of a graph G, which are subdivisions of K_5 or $K_{3,3}$, turn out to define facets of this polytope. For cliques contained in G, the Euler inequalities turn out to be facet-defining for the planar subgraph polytope. Moreover, they introduce the subdivision inequalities, V_{2k} inequalities, and the flower inequalities, all of which are facet-defining for the polytope.

The approach of Jünger and Mutzel [111] is based on an algorithm which searches for forbidden substructures in a graph that contains a subdivision of K_5 or $K_{3,3}$. These structures give inequalities which are used as cutting planes.

A *polytope* in R^n is the convex hull of finitely many points, or, equivalently, a polytope is a bounded subset of R^n that is the intersection of finitely many half-spaces. Those points of a polytope P which are not representable as a convex combination of other points in P are the vertices of P.

Suppose a graph $G = (V, E)$ with edge weights w_e for all $e \in E$ is given. Let P_G be the set of all planar edge-induced subgraphs of G. For each planar subgraph $P = G[F] \in P_G$, we define its *incidence vector* $\chi^P \in R^E$ by setting $\chi^P_e = 1$ if $e \in F$ and $\chi^P_e = 0$ if $e \notin F$. This yields a 1-1-correspondence of the planar subgraphs with certain $\{0,1\}$-vectors in R^E. The *planar subgraph polytope* $PLS(G)$ of G is defined as the convex hull over all incidence vectors of planar subgraphs of G : $PLS(G) := conv\{\chi^P \in R^E | P \in P_G\}$.

The problem of finding a planar subgraph P of G with weight $w(P)$ as large as possible can be written as the linear program $max\{w^T x | x \in PLS(G)\}$, since the vertices of the polytope $PLS(G)$ are exactly the incidence vectors of the planar subgraphs of G. In order to apply linear programming techniques to solve this LP, $PLS(G)$ has to be represented as the solution of an inequality system.

Due to the NP-hardness of this problem, it cannot be expected to find a full description of $PLS(G)$ by linear inequalities. Nevertheless, a partial description of the facial structure of $PLS(G)$ by linear inequalities is useful for the design of a "branch-and-cut" algorithm, because such a description defines a relaxation of the original problem. Such relaxations can be solved within a branch-and-bound framework via cutting-plane techniques and linear programming in order to produce tight bounds.

The Planarity-Testing Algorithm of Hopcroft and Tarjan [104]. At the beginning a depth-first-search procedure is called in order to divide the edge set of the graph $G = (V, E)$ into back edges and tree edges. First, a cycle C is identified. When this cycle is removed from G, the graph falls apart into several pieces. The algorithm is called recursively to embed each piece in the plane together with the original cycle. Then the embeddings of the pieces are combined, if possible, to give an embedding of the entire graph.

One may think of successively adding paths consisting of tree edges and one back edge at the end to a previously obtained partial embedding. For more details, see [145] or [104]. In the following we describe some details of the branch-and-cut algorithm of Junger and Mutzel [111, 112].

Cutting-Plane Generation. The trivial inequalities are handled implicitly by the LP-solver via lower and upper bounds. At the beginning also the inequality $x(E) \leq 3|V|-6$ is added, if it is violated (resp. $x(E) \leq 2|V| - 4$ in case G contains no triangles, if it is violated).

Let x be an LP-solution produced in the cutting-plane procedure applied in some node of the enumeration tree. For $0 \leq \varepsilon \leq 1$ we define $E_\varepsilon = \{\varepsilon \in E | x_\varepsilon \geq 1 - \varepsilon\}$ and consider $G_\varepsilon = (V, E_\varepsilon)$. For the unweighted graph G_ε the linear planarity-testing algorithm of Hopcroft and Tarjan is called. The algorithm stops if it finds an edge set F which is not planar. In case the inequality $x(F) \leq |F| - 1$ is violated, we add the inequality to the constraints of the current LP. Jünger and Mutzel [112] remove the back edge of the path, which proved the nonplanarity of F after it was added and proceed with the

planarity-testing algorithm. This way Jünger and Mutzel usually find several forbidden subgraphs of the graph G_ε in one run of the planarity-testing algorithm. Of course, these forbidden subgraphs do not necessarily define facets of the PLS-polytope.

However, these subgraphs must contain subgraphs which define facets. Jünger and Mutzel [112] try to reduce them to facet-defining inequalities in the following way. Once an edge set F is found, where the inequality $x(F) \leq |F| - 1$ is violated, successively one edge $f \in F$ is deleted from it, and the planarity testing algorithm starts again. If $F \backslash \{f\}$ is planar, it is added to F again. In either case a different edge $f \in F$ is chosen. In at most $|F|$ steps F is reduced to a set of edges, which induces a minimal nonplanar subgraph.

So an inequality $x(F) \leq |F| - 1$ is obtained which is facet-defining for $PLS(G)$ and still violated by the current LP-solution. Additionaly, a simple heuristic is used which searches for violated Euler inequalities.

Lower-Bound Heuristic. After an LP has been solved, Jünger and Mutzel [112] try to exploit the solution to produce a feasible solution, again, by applying the planarity-testing algorithm. This way they obtain lower bounds which are useful not only for fathoming nodes in the branch-and-cut tree but also for fixing variables due to their reduced costs during a cutting-plane phase.

After discovering a forbidden substructure, the back edge of the last added path is removed, so that the remaining substructure becomes planar. Since different depth-first-search trees yield different paths and thus different lower bounds, in every call of the planarity-testing algorithm the depth-first-search tree is changed.

Jünger and Mutzel also implemented a simple random heuristic, where the edges are subsequently added to the graph, if they do not destroy planarity. Their experimental results confirm the results of Cimikowski [36], who reported that simple random heuristics lead to better results on random graphs than the above-described method.

Jünger and Mutzel [111] report computational results where the heuristic based on the branch-and-cut algorithm was applied to various graphs known from the literature with 10 to 100 vertices. In many cases, a provably optimal solution, or at least a solution that is better than the previously known one, could be found. But the running time needed is usually significantly larger than the running time of other algorithms. In fact, Junger and Mutzel interrupt their algorithm when a time limit of 1000 CPU seconds is reached. They find that the easiest problem instances are sparse graphs and very dense graphs having up to 200 vertices or 700 edges, and that for weighted graphs the performance of their branch and cut heuristic is much worse than for unweighted graphs.

7.3 Approximation algorithms

Numerous approximation algorithms for Maximum Planar Subgraph Problem appear in the literature, the simplest ones being Spanning Tree (output any spanning tree of G, assuming G is connected) and Maximal Planar Subgraph (output any planar subgraph to which the addition of any new edge would violate planarity).

First consider a trivial approximation for finding a maximum planar subgraph by observing that for a given connected graph G with n vertices, any spanning tree of G is a planar subgraph with $n - 1$ edges, and that a spanning tree can be found in linear time.

Furthermore, a planar subgraph of G cannot have more than $3n-6$ edges. So if E' is the edge set of a spanning tree for a given graph G, and if E^* is the edge set of a maximum planar subgraph of G, then the ratio $\frac{|E'|}{|E^*|}$ is bounded (see also [41]):

$$\frac{|E'|}{|E^*|} = \frac{n-1}{E^*} \geq \frac{n-1}{3n-6} > \frac{1}{3}$$

For a graph G, we define $Opt(G)$ to be the maximum size of a planar subgraph of G. Given an algorithm A that takes representations of graphs G as input and outputs subgraphs of G, define $A(G')$ to be the size of the planar graph A produces when G is the input. Now let us define A's *performance* or *approximation ratio* $r(A)$ to be the infimum, over all graphs G, of $\frac{A(G)}{Opt(G)}$, if $Opt(G) > 0$, and 1 otherwise.

Dyer, Foulds and Frieze [60] proved that Maximal Planar Subgraph has performance ratio 1/3. Cimikowski [36] proved that a path embedding heuristic of Chiba, Nishioka and Shirakawa [34] and an edge-embedding heuristic of Cai, Han and Tarjan [25] have performance ratios not exceeding 1/3. In the same paper, Cimikowski studied two other polynomial-time heuristics: the "vertex-addition heuristic" and the "cycle-packing heuristic". The performance ratio of the former, to the authors' knowledge, is not known, whereas for the cycle-packing algorithm, it is 0. Dyer, Foulds and Frieze [60] studied two other algorithms and proved that each has performance ratio at most 2/9. What makes the problem more tantalizing is that achieving a performance ratio of 1/3 is trivial.

The trivial bound was improved for the first time by Calinescu, Fernandes, Finkler and Karloff [26, 27, 28] who present two new approximation algorithms for Maximum Planar Subgraph Problem. Each achieves a performance ratio exceeding 1/3. The higher performance ratio is $4/9 = 0.444...$ and is achieved by an algorithm which (surprisingly) invokes an algorithm for the graphic matroid parity problem as a subroutine and which runs in time $O(m^{3/2}nlog^6n)$. A greedy variant still has performance ratio $7/18 = 0.3888...$ and runs in linear time on graphs of bounded degree.

Given a (connected) graph G, an algorithm which outputs a spanning tree of G achieves a performance ratio of 1/3. A graph whose cycles all have length three, *i.e.* are triangles, is planar, as it cannot contain a subdivision of K_5 or $K_{3,3}$Moreover, a connected spanning subgraph of G whose cycles are triangles, besides being planar, has one more edge per triangle than a spanning tree of G.

Calinescu *et al.*'s better algorithm [27] produces a subgraph of G whose cycles are triangles and, among these subgraphs, has the maximum number of edges. It can be implemented in time $O(m^{3/2}nlog^6n)$, where m is the number of edges in G and n is the number of vertices in G, using a graphic matroid parity algorithm.

Planarity testing

Since finding a maximum planar subgraph is NP-complete [79], a maximal planar subgraph seems to be a reasonable approximation, as it is solvable in polynomial time. The maximal planar subgraph problem is closely related to the planarity-testing problem. In fact, a graph is planar iff it is the maximal planar subgraph of itself.

Given an undirected graph, the *planarity testing problem* is to determine whether there exists a clockwise edge ordering around each vertex such that the graph can be drawn in the plane without any crossing edges.

Linear time planarity testing algorithm was first established by Hopcroft and Tarjan [104] based on a "*path addition* approach". A "*vertex addition* approach", originally developed by Lempel, Even and Cederbaum [130], was later improved by Booth and Lueker [19] to run in linear time using a data structure called a PQ-tree. These algorithms are quite complicated to implement.

Di Battista and Tamassia [52] introduced first on-line planarity testing algorithms based on a recursive SPQR-tree decomposition of a biconnected graph into its triconnected components. This structure allows to test whether two vertices are on the same face of the embedding and to add vertices and edges to the embedding in $O(\log n)$ time.

Subsequently, other algorithms for incremental planarity testing were designed, *e.g.* by La Poutre [164] and Hsu [107].

7.3.1 The $O(nm)$ algorithms

A straightforward way of finding a maximal planar subgraph is the Greedy Algorithm [133]: The input is a graph $G = (V, E)$ with n vertices and m edges. The output is a maximal planar subgraph $G' = (V, E')$ of G. We start with $G' = (V, E')$ and build up E' by considering one edge e of E after the other. For each $e \in E$, e is added to E' if $G' = (V, E')$ remains planar, and discarded otherwise. We stop either after all edges of E have been considered, or when $|E'|$ becomes equal to $3n - 6$ (since a planar graph cannot have more than $3n - 6$ edges).

For each edge of E that is considered we need to perform a planarity test for a graph with n vertices and at most $3n-6$ edges. Each planarity test takes linear time, *i.e.* $O(n)$ in the worst case. The remaining operations like updating E' take $O(1)$ time per edge. Thus the worst case time complexity is in $O(nm)$. Therefore, algorithms for finding a maximal planar subgraph are sought that not only have a better worst case time complexity than the algorithm described above, but that are also less involved.

Chiba, Nishioka, and Shirakawa [34] propose an algorithm using the planarity testing algorithm of Hopcroft and Tarjan [104]. However, they achieve a worst case time complexity of $O(nm)$, the same as that of the Greedy Algorithm.

Ozawa and Takahashi [156] have presented an $O(nm)$ algorithm based on the vertex addition algorithm for planarity testing of Booth and Lueker [19]. Jayakumar, Thulasiraman and Swamy [196] showed that in general this algorithm does not determine a maximal planar subgraph. Moreover, the resulting planar subgraph may not even contain all vertices.

7.3.2 The $O(mlog\ n)$ algorithms

In a static environment, where an n-vertex graph G is entirely known in advance, we can test the planarity of G and compute a planar embedding in optimal $O(n)$ time [104]. In a dynamic environment, where a planar graph G is assembled on-line by insertions of vertices and edges, we would like to determine quickly whether an update causes G to become nonplanar. A first step in this direction has been the dynamic technique presented in [191] for the restricted problem of maintaining a planar embedding of a planar graph.

Di Battista and Tamassia [53, 10, 8, 9] define and use SPQR-trees to describe the recursive decomposition of a 2-connected graph into its 3-connected components. Their

incremental planarity testing problem consists of performing the following operations on a planar graph G: (i) for two vertices v_1 and v_2 in G with $v_1v_2 \notin E$, determine whether G stays planar if the edge v_1v_2 is added to G; (ii) If $v_1 \in V$, $v_2 \in V$, $v_1v_2 \notin E$, add the edge v_1v_2 to G (assuming the corresponding request of type a yields a positive answer); (iii) add a new vertex to G.

This data structure uses $O(n)$ space, and allows to test whether two vertices are on the same face of the embedding and to add vertices and edges to the embedding in $O(log\ n)$ time. Hence, Di Battista and Tamassia [53] obtained an $O(mlog\ n)$ time algorithm for finding a maximal planar subgraph as a byproduct of an algorithm for incremental planarity testing.

Independently, Cai, Han and Tarjan [25] give an $O(mlog\ n)$-time and $O(m)$-space solution to the maximal planar subgraph problem. For sparse graphs (*i.e.* graphs with $m = O(n^{1+\varepsilon})$, where $\varepsilon < 1$), it beats the algorithm of Jayakumar *et al.* even in the special case when a biconnected spanning planar subgraph is given. The method of [25] is much less complicated than that of [53], however, as it is designed to solve a less general problem.

Cai *et al.*'s algorithm [25] is based on a new version of the Hopcroft and Tarjan planarity testing algorithm [104]. The main difference is that it admits a more general ordering than the original H-T algorithm does in processing the successors of each tree edge. Also, the H-T algorithm processes one path at a time, while that of [25] processes one edge at a time. In this sense, the new algorithm is a more recursive version of the H-T algorithm.

In [164], La Poutre presented algorithms for incremental planarity testing that yield an $O(n + m\alpha(m, n))$ time algorithm for the maximal planar subgraph problem (where $\alpha(m, n)$ is the functional inverse of the Ackermann function). This result was improved to linear time complexity by Djidjev [54], and, independently, by Hsu [107].

7.3.3 The $O(n^2)$ algorithms

Jayakumar, Thulasiraman and Swamy [197] presented a two-phase algorithm for solving the maximal planar subgraph problem in time $O(n^2)$ based on the Lempel-Even-Cederbaum algorithm for testing planarity. In the first phase, an algorithm called Planarize computes a spanning planar subgraph G_s of G in $O(n^2)$ time. Furthermore, they present an algorithm called MaxPlanarize that augments G_s to a subgraph G' of G by adding additional edges in $O(n^2)$ time. They claim that G' is a maximal planar subgraph of G if G_s turns out to be biconnected.

Kant [116] shows that this algorithm is incorrect, and suggests a modification of the second phase of the algorithm that augments G_s to a maximal planar subgraph of G, even if G_s is not biconnected, maintaining $O(n^2)$ time requirement.

Jünger, Leipert and Mutzel [110] showed that the algorithm of Jayakumar *et al.* [197] to solve the maximal planar subgraph problem with PQ-trees is not correct even with the ideas described by Kant. The source of the problems is that the PQ-trees in MaxPlanarize are constructed according to the *st*-numbering that was computed for the nonplanar input graph G. As a matter of fact, the *st*-numbering of G does not imply an *st*-numbering of any maximal planar subgraph G_p even if the subgraph G_p is biconnected. Thus, MaxPlanarize does not obey the following invariant for planarity: Given a planar graph $G = (V, E)$ with an *st*-numbering, $1 \leq k \leq n$.

If the edge (t, s) is drawn on the boundary of the outer face, then all edges and vertices that have not yet been introduced into the current subgraph G_k are always embedded into the outer face of G_k. Since the numbering that is used to determine the order in which the vertices are reduced does not correspond to an st-numbering of G_p in general, the algorithm of Jayakumar *et al.* ignores edges that can be added into an inner face of the embedding of a current graph G_k without destroying planarity, and only considers edges for reintroduction into the planar subgraph G_p that are on its outer face.

Jünger *et al.*[110] have further noted that even a corrected version of the two-phase algorithm applied in the best possible case, where the st-numbering of a graph G is as well an st-numbering of the planar subgraph G_p, is not correct. Since this best case is a very rare case and since the modifications for the solved problems (see [129]) are far beyond any reasonable implementation, Jünger *et al.* doubt that a useful algorithm based on the strategy presented by Jayakumar *et al.* [197] can be found.

7.3.4 The $O(n)$ algorithms

Djidjev [54] constructed an optimal linear time algorithm for the maximal planar subgraph problem. His solution is based on a dynamic graph search procedure and a fast data structure for on-line planarity testing of triconnected graphs.

Given a graph $G = (V, E)$, Djidjev first computes a depth first search tree of G. This spanning tree of G is the initial planar subgraph $G' = (V, E')$ of G. Then for each edge $e \in E \backslash E'$ it is determined whether the graph $(V, E' \cup \{e\})$ is still planar. If so, e is added to E'. The order in which the edges in $E \backslash E'$ are considered is chosen in a sophisticated way so that, with $O(1)$ amortized time per test and insert operation for each edge $e \in E \backslash E'$, the overall time complexity is linear. Many intricate data structures are needed to achieve the $O(1)$ amortized time per test and insert operation. Two of them are BC-trees to describe the decomposition of a connected planar graph into its 2-connected components and SPQR-trees to describe the decomposition of a 2-connected graph into its 3-connected components [9].

Djidjev's [54] algorithm for the maximal planar subgraph problem can be transformed into a linear algorithm for planarity testing based on an approach completely different from the existing ones. The previous algorithms of Hopcroft and Tarjan [104] and Booth and Lueker [19] are based on the Jordan Curve Theorem while Djidjev's algorithm is based on the uniqueness of the planar embedding of any triconnected planar graph. Djidjev's algorithm is linear and therefore asymptotically best possible. However, it is so involved that a linear implementation seems difficult to achieve.

Compared to the standard algorithms for planarity testing of Hopcroft and Tarjan [104] or Booth and Lueker [19], Hsu and Shih [181] developed a very simple linear time algorithm for testing planarity based only on a depth-first search tree. The key to their approach is to add vertices according to a postordering obtained from a depth-first-search tree. The postordering is a labeling $l : V \rightarrow \{1, \ldots, n\}$ so that if u is an ancestor of v in the depth first search tree, then $l(u) > l(v)$.

By emulating its steps, Hsu [107] extended the planarity testing algorithm of [181] for finding a maximal planar subgraph. The new algorithm also starts with a depth first search tree of the given graph $G = (V, E)$, and then determines a postordering of the vertices of G. The initial planar subgraph G' of G is empty, and the vertices are added in

ascending order of their labels. So in step i of the algorithm, the vertex with label i (and the edges incident to it) are added to G'. Note that G' is not necessarily connected at all times.

Hsu pointed out the the way, in which the vertices are added and in which for each edge it is decided whether the edge can be added to G' without destroying planarity ensures the construction of a maximal planar subgraph in linear time. Moreover this algorithm appears to be less complicated than that of Djidjev [54].

7.3.4.1 A linear algorithm for finding a maximal planar subgraph 1

Djidjev [54] described the first linear $O(n+m)$ time algorithm for the maximal planar subgraph problem. The algorithm uses a tree-represented decomposition of a biconnected graph into triconnected components, a common feature of the incremental planarity testing algorithms [53, 8, 208, 164].

The algorithm has the following structure: (i) it initially constructs a depth-first spanning tree of G (we can assume that w.l.o.g. that G is connected) and uses it as an initial approximation of the maximal planar subgraph; (ii) it adds the edges one by one, making an on-line choice of the next edge to be added so that the testing time be appropriately small. The ability to make a choice of the order in which to insert, while possible, the edges into the subgraph so that planarity is preserved is essential for achieving $O(1)$ amortized time per test and insert operation.

Djidjev maintains in each bicomponent a special dynamic path of nodes of the decomposition tree such that all testing and updating operations are performed on nodes of that path. This makes it possible to implement data structures such as SPQR- or BC-trees supporting set union and set split operations in a constant amortized time. Furthermore, Djidjev developed a new efficient data structure used for incremental planarity testing of triconnected graphs which works in $O(1)$ amortized time per operation.

Recall that a SPQR-tree for a biconnected graph G is a recursively defined tree T closely related to the decomposition of G with respect to its split pairs [53]. T has four types of nodes S, P, Q, and R and there is an st-graph, $skeleton(\mu)$, associated with each node μ of T. The skeletons of the internal nodes of T are in one-to-one correspondence with the tricomponents of G and hence their number is $O(m)$. The endpoints of each edge t in the skeleton of the root of T correspond to a maximal split pair of G and t represents the set of split components of that split pair.

A property of the SPQR-trees that is relevant to planarity testing is that the skeleton of any internal node μ of a SPQR-tree has either a unique planar embedding (if μ is an R node), or *any* two edges can be placed on the same face (if μ is a P, Q, or S node.) For a more detailed discussion of SPQR-trees see [53, 8]. Our next goal is to show how to reduce a planarity testing in a graph to planarity testing in skeletons of nodes of its SPQR-tree.

In order to handle connected graphs that are not necessarily biconnected we define the BC trees introduced in [8] which are extensions of the SPQR-trees. To construct a BC tree of a connected graph G first find all bicomponents of G. Then construct a tree that contains a node of type B for any bicomponent b and a node of type C for any cut vertex c of GÅssociate with each B node b an SPQR-tree representing b. Connect a C node c and a B node b iff c belongs to b. Finally root the tree at an arbitrary B node.

Djidjev's algorithm [54] uses the decomposition tree described above to represent the decomposition of the current planar subgraph. For maintaining the embeddings of skeletons and for answering queries at each node of a SPQR-tree, Djidjev uses an algorithm that chooses a new edge and checks if it is possible to add it to the subgraph so that planarity is preserved at each iteration. The order in which edges are tested for insertion into the subgraph is critical for the efficiency of Djidjev's algorithm.

Another feature of the algorithm is that it maintains a dynamic set *Upaths* of paths in the decomposition tree called *update paths* which will be the "working" paths, *i.e.* all information it currently could need will be associated with nodes in these paths and all updates will be done on nodes in paths from *Upaths*. By using properties of these paths it is possible to make queries and do updates more efficiently.

Djidjev's algorithm is linear and therefore asymptotically best possible. However, it is so involved that a linear implementation seems difficult to achieve.

7.3.4.2 A linear algorithm for finding a maximal planar subgraph 2

In the two previous approaches [104, 130] of planarity test, the partial subgraph constructed at each iteration is always connected. The *st*-numbering of Lempel *et al.*'s approach further requires that those vertices not added induce a connected subgraph.

Hsu [107] adopted a „vertex addition" approach which only requires that those vertices „not added" induce a connected subgraph. Thus, a simple postordering of a depth-first search tree of G suffices. Let G_i be the subgraph at the i-th iteration consisting of the first i vertices and those edges among them. In the approach of Hsu, G_i may be disconnected, but the embedding for each biconnected component of G_i, once determined, is never changed.

Assume the given graph G is biconnected and the degree of each vertex is at least 3. Hsu [107] uses a ***generalized forest representation*** $\boldsymbol{F_i}$ for each G_i. Each node in the forest F_i represents either (a) an original vertex of G (denoted by a ***v-node***) adjacent to vertices not in G_i. or (b) a biconnected component of G_i whose planar embedding has already been determined (denoted by a ***c-node***).

Further Hsu defines the ***external degree*** of a vertex at iteration i to be the the number of its neighbors among $\{i+1, \ldots, n\}$. Since some of those vertices in G_{i-1} with external degree 0 do not have to be examined for future embedding, we shall apply a **vertex contraction** procedure to eliminate them. The contraction procedure replaces a connected subgraph by a contracted edge, which makes it possible to recognize and embed the planar graph more efficiently. Let T_j' denote the tree after the vertex contraction from T_j.

At the beginning of the i-th iteration, Hsu [107] reduces the external degrees of all neighbors of vertex i in F_i by 1. The contraction procedure starts by marking all vertices of T_j that are adjacent to i. Note that a vertex which is not marked at this stage can become marked later through a contracted edge. There are two types of contraction: contracting vertices on a path and contracting vertices on a cycle.

The main idea of the MPS (Maximum Planar Subgraph) algorithm presented in [107] is to traverse the tree towards the root from those vertices adjacent to the new vertex. To accommodate for slot reservation, it is required to label the vertices and biconnected

components traversed in the algorithm so that each edge will be traversed a constant number of times.

Each tree in the current forest that contains vertices adjacent to the new vertex will produce at most one biconnected component containing that new vertex. The MPS algorithm will assume such a component is already formed (which is actually pending on the determination of terminal nodes) and each vertex of the boundary cycle is labeled with the new vertex. Thus, the next time any vertex, say u, of this component, say C, is traversed two things will happen: (i) vertex u will be identified to be lying on the boundary of the biconnected component formed by the current new vertex; (ii) the traversal will continue with the new vertex of C, thus skipping all other vertices of component C.

Let i be the new vertex in the current iteration. Consider a tree with root x in the current forest that contains at least two vertices adjacent to i. Such a tree will create a biconnected component for i at this iteration. We shall associate the edge (i, x) with this component. The MPS algorithm traverses the vertices and edges of the tree as follows. It first picks the neighbor of i with the lowest order, say u, in the tree and traverses the unique path from u to x. Each vertex v on that path will be labeled with (i, x, d), where d is the distance between v and u measured by the number of edges along the path.

Since the postordering guarantees that vertices with lower order will be considered before those with higher orders, those neighbors of i along the path from u to i will be labeled and they will no longer be considered at this iteration. After the unique path from u has been traversed the algorithm will pick the next lowerest unlabeled vertex, say u', adjacent to i and traverse the unique path from u' to i. The traversal will terminate when it encounters a vertex already labeled. The algorithm then picks the next lowerest unmarked vertex and continues until all neighbors of i in this tree have been considered. Everytime a new edge is added to the subgraph some other edges could be deleted.

The argument for the linear complexity is mainly based on the fact that the number of times an edge is traversed is a constant. Hence, the time complexity of the MPS algorithm is linear. Full details on this approach can be found in [107].

7.4 Heuristic algorithms

The Greedy Algorithm of Section 7.3.1 finds a maximal planar subgraph, which will be at least as good as just taking a spanning tree. [120, 60, 74] use the following greedy heuristic for the Weighted Maximum Planar Subgraph problem: Instead of considering the edges in arbitrary order, they consider them in an order of nonincreasing weight. This Greedy Heuristic does involve repeated planarity testing, and even though planarity testing can be done in linear time, the algorithms are rather complicated. The following heuristics avoid planarity testing.

The Deltahedron Heuristic [76, 74] of Foulds and Robinson starts with a tetrahedron (K_4) as the initial planar subgraph and then adds one vertex at a time, placing each new vertex in one of the faces of the current planar subgraph. The sequence in which the vertices are added is determined by a vertex weight W that can be defined in various ways, as discussed below. Note that in contrast to the Greedy Heuristic, the Deltahedron Heuristic does not necessarily yield a maximal planar subgraph of the input graph.

Leung [132] generalizes the Deltahedron Heuristic. Starting with a tetrahedron (K_4), a planar subgraph is built such that in each step, the current planar subgraph has only

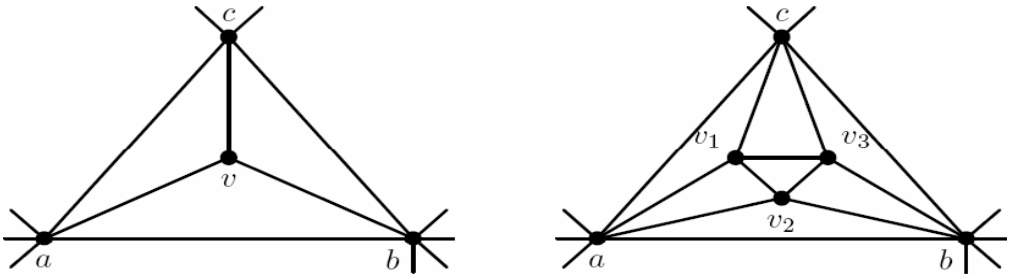

Figure 16: [133] A step in the Deltahedron Heuristic for finding a planar subgraph with large edge weights [76, 74, 60](left), or in its generalization [132](left or right). In the operation on the left, vertex v and 3 incident edges are inserted into face abc. In the operation on the right, vertices v_1, v_2, v_3 and 9 incident edges are inserted into face abc.

triangular faces. In each step, a single vertex and three incident edges (as in the Deltahedron Heuristic) or a set of three vertices and nine incident edges are placed in one of the faces of the current planar subgraph as illustrated in 16.

Unlike in the Deltahedron Heuristic, the vertices to be inserted are not chosen in any predetermined ordering, but in each step the vertex or the set of three vertices, and the face into which to insert them, is determined so that the gain in edge weights per inserted vertex in this step is best possible. The worst case time complexity of this approach is $O(n^4 log\ n)$. Computational results are carried out, generating the test base in much the same way as [74]. They suggest that the results of the generalized Deltahedron Heuristic are better than the ones achieved by the original Deltahedron approach discussed in [76, 74].

For the unweighted case (*i.e.* all edge weights are 1) there are still other approaches. Cimikowski [41] suggests a heuristic based on finding, for each 2-connected component of a non-planar graph, a pair of edge-disjoint spanning trees whose union is planar. Then this union forms a planar subgraph and has $2n - 2$ edges.

Although no computational results are given by the author, the main interest of this approach is due to the fact that, under certain conditions, the number of edges of the generated planar subgraph is at least 2/3 of the optimum. If the graph does not have two such spanning trees, some heuristic edge manipulations are performed, so that the output is still a spanning planar subgraph, but without a guaranteed number of edges. If two spanning trees exist, they can be found in $O(m^2)$ [175].

Takefuji and Lee [188, 189] and Goldschmidt and Takvorian [86] each propose a two-phase heuristic for finding a planar subgraph with as many edges as possible. In the first phase, a linear ordering of the vertices is determined. The vertices are placed on a line according to that ordering. In the second phase edges are placed above or below the line. The resulting planar subgraph is thus embedded in a book with two pages. The techniques used for each phase are very different in [188] and [86]. [188] places the vertices in a random order in the first phase and uses a neural network technique for the second phase.

Goldschmidt and Takvorian [86] argue that it is useful to attempt to order the vertices of the input graph $G = (V, E)$ according to a Hamiltonian cycle in the first phase. Given

an ordering of the vertices on a line, in the second phase a partition of E into three sets A, B, and C must be determined so that $|A| + |B|$ is as large as possible, and so that no two edges of A (B) intersect if all edges of A (B) are placed above (below) the line of vertices. The edges in C are not part of the planar subgraph. If we imagine the vertices of G to lie on the real line, then each edge $e \in E$ can be regarded as an interval defined by its two end vertices [133]. Let $H = (E, F)$ be a graph such that each edge of G is a vertex of H. Let e_1, e_2 be two edges of G and thus two vertices of H, and let i_1 and i_2 be the intervals corresponding to the edges e_1 and e_2 in G, e_1 and e_2 are connected by an edge in H if and only if the intervals i_1 and i_2 intersect but none is contained in the other. Thus H is an overlap graph (also called circle graph). Finding the sets A, B and C as described above is now equivalent to finding a maximum induced bipartite subgraph of the overlap graph H. Finding a maximum induced bipartite subgraph of an overlap graph is NP-complete [176].

Goldschmidt and Takvorian [86] now use the following greedy algorithm to construct a maximal induced bipartite subgraph of an overlap graph: Find a maximum independent vertex set in H (the vertices of this set are then the edges in A), delete it from H, and find a maximum independent set in the remaining graph (the vertices of this set are then the edges in B). Since a maximum independent set of an overlap graph can be found in polynomial time [83], this algorithm runs in polynomial time also. [86] shows that the number of vertices in the maximal induced bipartite subgraph is at least 0.75 times the number of vertices of a maximum bipartite subgraph.

Computational results reported by Goldschmidt and Takvorian [86] compare their implementation of their heuristic with their implementation of [188] on a set of 19 graphs with 10 to 150 vertices and two larger graphs with 300 and 1000 vertices, respectively. For each instance, their heuristic finds at least as good a solution as [188]. For the graphs with 50 or more vertices, the solution of [86] is even dramatically better than that of [188]. But note that the test base is small, that it is unclear how representative it is, and that even the results of [86] might still be very far away from an optimal solution.

The approach of [86] is further refined by Resende and Ribeiro [167]. They apply a greedy randomized adaptive search procedure (GRASP), a metaheuristic for combinatorial optimization [72, 142], to the problem of planarizing a graph through edge deletion and review basic concepts of GRASP: construction and local search algorithms.

Experimental results using most graphs from the test base in [86] as well as graphs with up to 300 vertices collected by Cimikowski are discussed in [167]. They indicate that the GRASP compares favorably with the results of [86]. In comparison with the branch and cut heuristic [146, 111], however, the situation is not so clear: On some instances the branch and cut heuristic is clearly better, on others the GRASP outperforms the branch and cut heuristic. The latter happens in particular when the time limit set for the branch and cut heuristic is reached so that the computation is halted and the best solution found until then is reported.

7.4.1 GRASP for Graph Planarization

Resende and Ribeiro [167] apply the concepts of GRASP to the graph planarization problem. A GRASP [72] is an iterative process, where each GRASP iteration consists of two phases: construction and local search. The construction phase builds a feasible

solution, whose neighborhood is explored by local search. The best solution over all GRASP iterations is returned as the result.

In the construction phase, a feasible solution is built, one element at a time. At each construction iteration, the next element to be added is determined by ordering all elements in a candidate list with respect to a greedy function that measures the actual benefit of selecting each element. The adaptive component of the heuristic arises from the fact that the benefits associated with every element are updated at each iteration of the construction phase to reflect the changes brought on by the selection of the previous elements. The probabilistic component of a GRASP is characterized by randomly choosing one of the best candidates in the list, but usually not the top candidate. This way of making the choice allows for different solutions to be obtained at each GRASP iteration, but does not necessarily jeopardize the power of GRASP's adaptive greedy component.

The solutions generated by a GRASP construction are not guaranteed to be locally optimal with respect to simple neighborhood definitions. Hence, it is almost always beneficial to apply a local search to attempt to improve each constructed solution. A local search algorithm works in an iterative fashion by successively replacing the current solution by a better solution from its neighborhood. It terminates when there is no better solution found in the neighborhood with respect to some cost function. Success for a local search algorithm depends on the suitable choice of a neighborhood structure, efficient neighborhood search techniques, and the starting solution.

The GRASP construction phase plays an important role with respect to this last point, since it produces good starting solutions for local search. Normally, a local optimization procedure, such as a two-exchange, is employed. While such procedures can require exponential time from an arbitrary starting point, empirically their efficiency significantly improves as the initial solutions improve [167]. Through the use of customized data structures and careful implementation, an efficient construction phase that produces good initial solutions for efficient local search can be created. The result is that often many GRASP solutions are generated in the same amount of time required for the local optimization procedure to converge from a single random start. Furthermore, the best of these GRASP solutions is generally significantly better than the solution obtained from a random starting point.

Algorithm 7.1.

```
procedure GRASP(ListSize, MaxIter, RandomSeed)
1    InputInstance();
2    for k = 1, ..., MaxIter do
3        ConstructGreedyRandomizedSolution(ListSize,RandomSeed);
4        LocalSearch(BestSolutionFound);
5        UpdateSolution(BestSolutionFound);
6    end
7    return BestSolutionFound
end
```

Algorithm 7.1 illustrates a generic GRASP implementation in pseudo-code. The GRASP takes as input parameters for setting the candidate list size, maximum number of GRASP iterations and the seed for the random number generator. After reading the instance data (line 1), the GRASP iterations are carried out in lines 2-6. Each GRASP

iteration consists of the construction phase (line 3), the local search phase (line 4) and, if necessary, the incumbent solution update (line 5).

As outlined above, a GRASP possesses four basic components: a greedy function, an adaptive search strategy, a probabilistic selection procedure, and a local search technique. These components are linked together into an iterative method that constructs a feasible solution one element at a time and then feeds the solution to the local search procedure.

The computational complexity of phases 1 and 2 of this GRASP [167] is discussed next. The construction phase takes $|V| - 1$ iterations, each of which has complexity $O(|V|)$, resulting in an $O(|V|^2)$ procedure. To analyze the local search phase, we need an upper bound on the number of edge crossings for a given vertex sequence. The number of edge crossings in a complete graph is bounded above by $O(|V|^4)$. Each GRASP iteration reduces the number of edge crossings by at least one. Consequently, the number of iterations is bounded above by $O(|V|^4)$. Since each iteration has time complexity $O(|E||V|^2)$, the time complexity for the local search is $O(|E||V|^6)$.

7.4.2 Performance of the heuristics

For algorithmic results, and in particular for approximations and heuristics, computational results are an important performance measure, both regarding the quality of the result of the algorithm and the running time needed. But a fair comparison of algorithms with each other on the basis of computational results is usually difficult, if not impossible, since the implementation of an algorithm and the graphs used for the test strongly influence the computational results. Hence, the comparisons of algorithms made in this section have to be considered with caution.

In this section, we present two computational studies for evaluating the performance of heuristics for computing the maximal planar subgraph, each of them comparing a different set of heuristics. The first study was conducted by Cimikowski (see [37, 36, 38, 41]), the latter was conveyed by Silva Carmo and Wakabayashi (see [45]).

7.4.2.1 Computational study 1

Cimikowski [37, 36, 38, 41] performed an empirical evaluation of heuristics for the graph planarization problem. Several heuristics were tested on a large and comprehensive set of test problems.

These included random graphs with unknown maximum planar subgraph size, non-planar graphs containing a maximum planar subgraph of size $3|V| - 6$, random Hamiltonian non-planar graphs, and a few special graphs already considered in the literature or possessing interesting structures or relevant applications. Cimikowski tested the following heuristic methods:

7.4.2.1.1 The Path-Embedding Heuristic.

The path embedding heuristic CNS (after Chiba, Nishioka, and Shirakawa [34]) is based on the linear time planarity algorithm of Hopcroft and Tarjan [104].

Using depth-first search (*dfs*), an initial cycle is found in a graph G, deleted, and then embedded in the plane. The remainder of G is then decomposed into edge-disjoint paths and an attempt is made to embed each path inside or outside the cycle. If all paths can be embedded, the graph is planar; otherwise it is nonplanar. Whereas the original

planarity algorithm halts when a path cannot be embedded, the heuristic CNS deletes a special edge called a "frond" from the path and continues.

7.4.2.1.2 The Edge-Embedding Heuristic.

Cai, Han, and Tarjan [25] proposed a variant of CNS which processes an *edge* rather than a path at a time. The heuristic CHT begins by finding a *dfs* tree with tree edges E_T and fronds E_F. Then it constructs a maximal planar subgraph of input graph G by first embedding an initial cycle C, then recursively embedding any remaining tree edges and as many fronds as it can while maintaining planarity, in an order determined by a *successor* relationship between edges.

For any edge $e = (a, b)$, let b be the *head* of e. If (a, b) is a tree edge and (b, c) is any edge, then (b, c) is a *successor* of (a, b). By definition, fronds have no successors. $low_1(c)$ is the lowest-numbered vertex reachable from vertex a or from any of its descendants (in the *dfs* tree) by a frond. Initially, the ordered list $succ(e)$ is computed for each edge $e \in E$. $succ(e)$ gives the successor edges of e in increasing low_1 order.

CHT first performs a *dfs* of G, partitioning E into tree edges E_T and fronds E_F. The edge-embedding process then begins, starting with the first tree edge generated by *dfs*. Then the next successor edge of the last tree edge is embedded. The embedding step is recursively applied, where the next edge chosen is the next successor of the last edge embedded, if any; otherwise, the next successor of the previous edge embedded is chosen, etc. Nonembeddable edges are discarded. The time complexity of CHT is $O(mlog\ n)$.

7.4.2.1.3 The Incremental Heuristic.

La Poutre's incremental heuristic INC starts with an "empty" graph and adds edges one at a time, discarding an edge if it causes nonplanarity. After each edge addition, a planarity test is performed. Using "incremental" planarity testing, i.e. [164], the time complexity is $O(n + m\alpha(m, n))$, which is roughly $O(n + m)$.

7.4.2.1.4 The Vertex-Addition Heuristic.

Booth and Lueker [19] developed a planarity testing algorithm based on vertex addition using PQ-trees. It embeds one vertex of a graph at each step in an order given by an "st-numbering" of the vertices. After each vertex addition, it may be necessary to reverse or permute pieces of the graph to preserve planarity.

Ozawa and Takahashi [156] extended this method, here we call it PQ, to find a maximal planar subgraph. Later, Kant [116] introduced some corrections to the method, though as shown in [110] it still does not guarantee to find a maximal planar subgraph.

The theoretical worst case running time is $O(|V|^2)$. In practice it is much faster. The quality of the results can be improved by introducing random events and calling them several times. The random event can be simply to choose a random edge of E in order to get s and t. Gutwenger and Mutzel [91] studied the effects of up to 100 calls.

7.4.2.1.5 The Cycle-Packing Heuristic.

This method by Goldschmidt and Takvorian ([85]) finds an optimal ordering of vertices along a horizontal "node line" in the plane and tries to embed as many edges as possible

Table 1: Time and space complexities of the heuristics († with incremental planarity testing [164]).

Heuristic	Time	Space
CNS	$O(mn)$	$O(mn)$
CHT	$O(m\ log\ n)$	$O(m)$
PQ	$O(n^2)$	$O(n^2)$
INC	$O(n + m\alpha(m,n))^{\dagger}$	$O(m)$
GT	$O(nm^2)$	$O(m)$

above and below the line. In the first phase of the heuristic *GT,* an attempt is made to find a hamiltonian cycle of a graph $G = (V, E)$, using a probabilistic algorithm. The cycle determines a vertex ordering O along the node line. The second phase embeds edges above and below the node line.

The ordering O influences the number of edges embedded. If G is hamiltonian, the optimal O corresponds to a hamiltonian cycle and a 3/4-approximation is guaranteed. Otherwise, a greedy ordering is used, and there is no performance bound.

7.4.2.1.6 The Branch-and-Cut Heuristic.

The branch-and-cut heuristic developed by Jünger and Mutzel [111, 112] is an exhaustive search algorithm based on linear programming and cutting plane generation, with feasibility bounding performed by a planarity testing algorithm. As planar obstructions are detected during the search process, fronds are deleted, until a maximal planar subgraph is found.

Since heuristic JM can require exponential time, a time limit is imposed on the computation and the current bound value is output. If run until completion, optimality is guaranteed.

7.4.2.1.7 Time and space complexities

As displayed in Figure 1, the $O(n^2)$ PQ implementation of Kant [116] is currently the fastest method for dense graphs. For sparse graphs, the incremental heuristic [164] is $O(n + m\alpha(n,m))$, while the method of [25] is $O(nlog\ n)$.

Since Cimikowski [37, 36, 38, 41] didn't include any of the linear time algorithms [107, 54] into his tests, these statements shall be considered with restraint.

7.4.2.1.8 Experimental results

Cimikowski's experiments show that the branch-and-cut heuristic of Jünger and Mutzel (JM [111, 112]) consistently outperformed the other heuristics for the graphs tested.

The two-phase heuristic of Goldschmidt and Takvorian (GT [86]) markedly outperforms the remaining in terms of solution quality, although its running time makes it

prohibitive for very large graphs. It is the method of choice when the input graphs are sufficiently dense and hamiltonian with high probability.

If the computation time is critical, then the approaches based on planarity testing (PQ [197]) and edge embedding (CHT [25]) are recommended. If cpu time is very critical then PQ and CHT is recommended.

The superiority of the incremental heuristic (INC [164]) over the path-embedding heuristic (CNS [34]) and the edge-embedding heuristic CHT, is difficult to explain. On random graphs, sparse graphs, and hamiltonian graphs it consistently outperformed CHT and CNS. Thus, INC is a good choice providing some care is taken to avoid occasional bad orderings of edges. This can be achieved by trying several different orderings.

7.4.2.2 Computational study 2

Foulds, Gibbons and Giffin [74] and Leung [132] performed computational comparison of the heuristics divided into two classes:

1. The *greedy* heuristic, which starts with the empty subgraph and tries to add edges one at a time, accepting an edge only if its addition results in a planar subgraph [84, 74]. Notice that all the heuristics mentioned in the first case are. in some sense, greedy approaches, but we use this term here to refer to the second case only which we denote GDY.

2. Those which start with a planar subgraph (actually a triangle) of the given graph and appropriately add vertices and edges so that planarity is preserved from the start to the end of the processing [75, 73, 62, 74, 132]. We call these the *planarity preserving* heuristics.

The *planarity preserving* heuristics may be thought of as variations on two main themes, with respect to the way the vertices and edges are added at each step: the *deltahedron* (DH) approach [76] and the *wheel expansion* (WE) approach [62].

In [74] an experimental comparison of them and their variants is presented: Some improvements to DH are proposed, which are referred to as IDH (for "improved DH"); also, a variation on the choice of the starting triangle gives different results, making up a total of 6 "different" heuristics which will be denoted as WE, WE', DH, DH', IDH and IDH'; finally, the *simplicial decomposition* approach, a last variation of DH, is presented in [132]. We will denote this by SD.

Further, Silva Carmo and Wakabayashi [45] proposed a nonnaive greedy algorithm GRD, where they tried to reduce the overhead imposed by the HT planarity test which is needed at each step of the algorithm. In doing so, they introduce the concept of *hierarchies,* a partial order with special properties which, when induced on the vertex set of a graph, serves as the formal counterpart of the well known technique of *Depth First Search* (DFS), and which can be easily "translated" into a data structure which represents this DFS. However, the perfomance ratio of GRD doesn't exceed 1/3, just as that of GDY.

7.4.2.2.1 Experimental results

Foulds, Gibbons and Giffin [74] and Leung [132] present experimental results on a set of 102 complete graphs whose edge weights obey a normal distribution with controlled

variance. Some measures for the performances of the mentioned heuristics on instances of various sizes are given in Figure 17.

Table (a) shows the average running times (I/O operations not computed) of each heuristics; Table (b) shows, the proportion in what each heuristics yielded the best of the solutions (compared to the solutions given by the others); Table (c) shows the average performance of each of them; the values indicate the ratio obtained solution/optimal solution the case of the size 10 instances and obtained solution/optimal solution in the others, where upper bound is the sum of the $3n-6$ highest weight edges of the instance; each column in the last line shows the average of that column; Table (d) shows the value of the worst performance ratio obtained for each heuristics. At each table, n indicates the number of vertices of the instances in question.

(a) Average Running Time (CPU seconds)

n	GDY	WE(′)	DH(′)	IDH(′)	SD
10	2,0	0,27	0,15	0,17	—
20	20,2	3,67	0,38	1,82	0,90
30	71,3	13,41	0,72	7,37	7,60
40	165,8	39,49	1,13	19,29	37,05

(b) Number of Instances with Best Performance (%)

n	GDY	WE	WE′	DH	DH′	IDH	IDH′
10	46,67	0,00	0,00	3,33	6,66	26,67	36,67
20	53,33	0,00	0,00	0,00	3,33	13,33	36,67
30	50,00	0,00	0,00	0,00	0,00	10,00	40,00
40	66,67	0,00	0,00	0,00	0,00	25,00	8,33
	52,94	0,00	0,00	**0,98**	**2,94**	**17,65**	**34,31**

(c) Average Performance (%)

n	10	20	30	40
GDY	99,588	96,042	95,585	94,730
WE	98,219	94,887	93,986	92,459
WE′	98,101	94,726	93,785	92,387
DH	98,270	94,607	94,025	93,280
DH′	99,090	95,376	94,703	93,567
IDH	99,101	95,784	94,984	94,338
IDH′	99,421	96,244	95,613	94,303
SD	—	96,423	95,507	94,985

(d) Worst Performance (%)

GDY	91,060
WE	91,002
WE′	89,481
DH	87,335
DH′	90,894
IDH	89,740
IDH′	90,977

Figure 17: [45] Comparative Performance of various Heuristics for the MPSP.

These tables were based on data published in [74] and [132]. From these data we can see that all the proposed heuristics give quite good results and, among them, GDY and IDH show the best performances.

If we are to chose between these two, we can see that GDY has a more stable behavior, slightly outperforming IDH in most of the tests. On the other hand, GDY has a serious drawback: it is too time consuming, and this is due to the overhead of performing a planarity test at each step of the construction, which is avoided by the *planarity preserving* heuristics-Based on such remarks, [74] concludes that IDH is the best practical choice, arguing that GDY, besides requiring the implementation of a linear planarity testing algorithm (a laborious task) would become impractical for large instances of the problem.

Silva Carmo and Wakabayashi [45] performed a computational comparison of the above heuristics with their algorithm GRD given in [45] on the same set of benchmark graphs. The average running times obtained for a series of random generated instances are presented in 18, each instance consists of a complete graph, given by a list of its edges in random order. The running times presented were averaged over a series of 10 runs for each instance size; Figure 18 shows, for each instance G, the number of vertices and edges

and the time in seconds spent by the algorithm to find a greedy maximum weight planar subgraph.

$\|V_G\|$	$\|E_G\|$	GRD	GDY	IDH	DH	WE	SD
10	45	0.06	2.0	0.17	0.15	0.27	—
20	190	0.66	20.2	1.82	0.38	3.67	0.89
30	435	2.53	71.3	7.37	0.72	13.41	7.59
40	780	6.65	165.8	19.29	1.13	39.49	37.05
50	1225	13.09	—	—	—	—	—
80	3160	60.39	—	—	—	—	—
100	4950	127.58	—	—	—	—	—

Figure 18: [45] Running times (CPU seconds).

8 Edge inserting strategies

In practice, the crossing minimization problem is usually solved heuristically using a 2-step planarization approach. After obtaining a maximum planar subgraph, now, in the second step of the planarization approach, we need to reinsert all edges that were removed for violating planarity in the first phase back into the graph. The aim is to obtain a smallest number of crossings in the resulting drawing of the graph.

8.1 Edge Re-insertion Strategies

According to Gutwenger and Mutzel [91] there are three main types of the edge re-inserting strategies, all of which are NP-hard [214]. In addition to that, they mention several post-processing strategies helping to reduce the number of crossings in the resulting drawing.

8.1.1 Fixed embedding.

The *edge insertion problem for a fixed embedding* can be stated as follows [91]: given a planar graph $G = (V, E)$ and a pair of vertices (v_1, v_2) in V, find a drawing of $G' = (V, E \cup \{(v_1, v_2)\})$ that has the minimum number of crossings among all drawings of G' in which every crossing is a crossing between an edge in E and the edge (v_1, v_2). Note that such a drawing of G' is not necessarily crossing minimal [93].

The standard algorithm used in practice re-inserts the edges $e_1, e_2, \ldots, e_k$ iteratively starting with a given planar embedding of G . The approach is based on the observation that an edge e_i crosses an edge in P if and only if it uses an edge in the geometric dual graph of P. Hence, the problem of reinserting only one edge into P with a given planar embedding can be solved via a simple shortest-path computation in the extended dual graph of P. (We need to extend the dual graph in order to connect the end-vertices of e_i with the dual graph.) After each insertion step i, the crossings generated by edge e_i are substituted by artificial vertices so that the resulting graph $G \cup \{e_1, \ldots, e_i\}$ becomes planar again $(i = 1, \ldots, k)$.

The theoretical worst case running time for inserting k edges of the Gutwenger and Mutzel's implementation [91] is $O(\Sigma_{i=1}^{k}(|V| + \Sigma_{j=0}^{i-1} c_j)) = O(k(|V| + |C|))$, where c_j is the number of crossings introduced in step j, $c_0 = 0$, and C the number of crossings in the final drawing. In practice, it is much faster, since the updates of the dual graphs are implemented efficiently. Gutwenger and Mutzel denote this re-insertion method as FIX.

8.1.2 Variable Embedding.

Problem 9 (Edge Insertion Optimizing over All Embeddings [91]). *Given a planar graph $G = (V, E)$ and a pair of vertices $(v_1, v_2$ in G, find an embedding Π of G such that we can add the edge $e = (v_1, v_2)$ to Π with the minimum possible number of crossings among all embeddings of G.*

When inserting an edge into a planar graph P, the quality of the resulting drawing highly depends on the chosen embedding of P. In [93], Gutwenger *et al.* present a linear time algorithm based on the SPQR-tree data structure for inserting one edge into a planar

graph P so that the number of crossings in $P \cup \{e\}$ over the set of all possible planar embeddings of P is minimized.

[91] denote this re-insertion method as VAR. Their implementation has the same theoretical running time as the variant FIX. Note that an optimal solution of the edge insertion problem does not necessarily lead to a drawing of the graph $G' = (V, E \cup \{e\})$ with the minimum number of crossings. This is due to the fact that there may not always be a drawing with the minimum number of crossings such that $G = (V, E)$ is drawn without crossings.

8.1.2.1 Edge inserting into a planar graph using SPQR-trees

One criticism of the planarization method was that when choosing a "bad" embedding in the edge re-insertion phase, the number of crossings may get much higher than necessary [96]. Hence, the question arose if there is a polynomial time algorithm for inserting an edge into the planar subgraph P so that the number of crossings is minimized. Therefore, the task is to optimize over the set of all possible combinatorial embeddings of P.

While it is possible to compute an arbitrary combinatorial embedding for a planar graph in linear time [143, 33], it is often NP-hard to optimize over the set of all possible combinatorial embeddings. Figure 19 shows a simple case in which the choice of the combinatorial embedding of the planar subgraph has an impact on the number of crossings produced when inserting the dashed edge. When choosing the embedding of Figure 19(a) for the planar subgraph (without the dashed edge), we get two crossings, while the optimal crossing number over the set of all combinatorial embeddings is one (see Figure 19(b)).

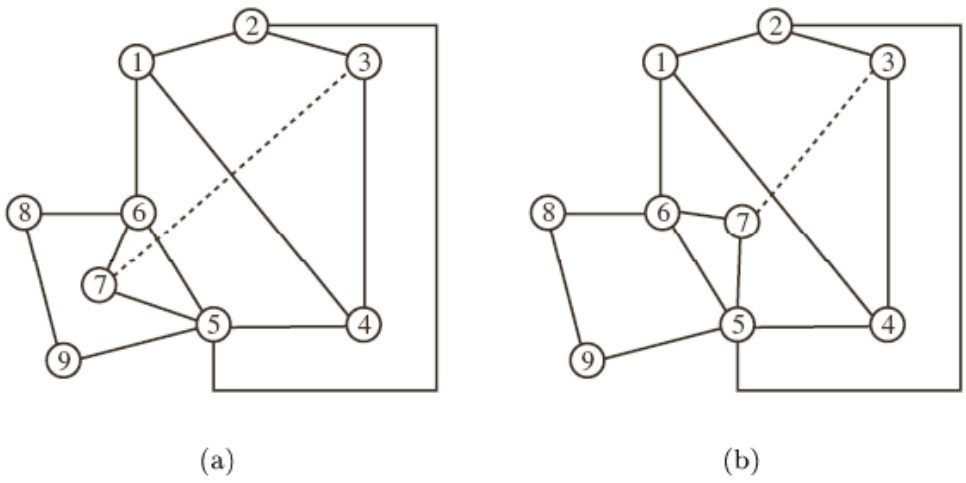

Figure 19: [93] The number of crossings required when inserting an edge highly depends on the chosen embedding.

Gutwenger, Mutzel and Weiskircher [93] show that the edge insertion problem can be solved in polynomial time, thus solving a long standing open problem in graph drawing. They present a conceptually simple linear time algorithm based on SPQR-trees which is able to solve the edge insertion problem to optimality. The time complexity of the

algorithm for computing an optimal edge insertion path for two vertices in graph $G = (V, E)$ is $O(|V| + |E|)$.

Definition 6 (**Edge Insertion Path** [93]). *Let $G = (V, E)$ be a connected planar graph, and let n be an embedding of G. Let v_1 and v_2 be two non-adjacent vertices in G. Then $e_1, \ldots, e_k$ is an edge insertion path for v_1 and v_2 in G with respect to Π if either $k = 0$ and v_1 and v_2 are contained in a common face in Π or the following conditions are satisfied:*

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

If $p = e_1, \ldots, e_k$ is an edge insertion path for v_1 and v_2 with respect to Π, then it is possible to insert the edge (v_1, v_2) into Π with k crossings, where the ith crossing involves edge (v_1, v_2) and edge e_i for $1 \leq i \leq k$. The length of p, denoted by $|p|$, is k. We call p an *optimal edge insertion path* for v_1 and v_2 in G, if there is no shorter edge insertion path for v_1 and v_2 in G with respect to any embedding of G. Figure 20 shows three different edge insertion paths for v_1 and v_2 with respect to the embedding realized by the drawing. The three paths are the empty path, the path e_1, e_2, e_3, and the path e_4, e_5, e_6. In this case the empty path is the optimal edge insertion path for v_1 and v_2.

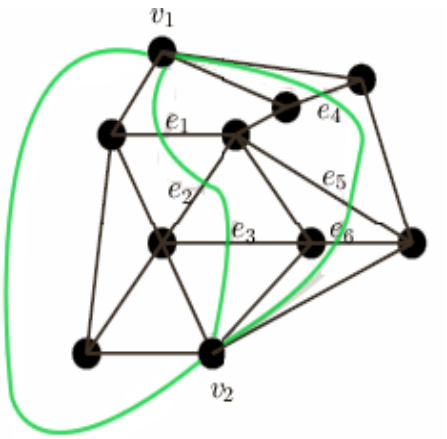

Figure 20: [93] Three different edge insertion paths for v_1 and v_2.

The *traversing costs* $c(e)$ of a skeleton edge e are defined as follows. Consider an arbitrary embedding Π of the graph $expansion^+(e)$ and its dual graph Π^*. Let f_1 and f_2 be the two faces in Π that are separated by e and let f_1^* and f_2^* be the corresponding vertices in the dual graph. Gutwenger *et al.* [93] denote with $P(\Pi^*, e)$ the shortest path in Π^* that connects f_1^* and f_2^* and does not use edge e^*. They also show that the length of this path is independent of the embedding Π chosen for $expansion^+(e)$. Thus they define the *traversing costs* $c(e)$ simply as $c(e) = length\ of\ the\ path\ P(\Pi^*, e)$ for any embedding Π of $expansion^+(e)$.

For the algorithm for inserting an edge into a biconnected planar graph, we need a definition of the *augmented dual graph*, which is used for finding a shortest edge insertion path in case of a fixed embedding.

Definition 7 (**Augmented Dual Graph** [93]). *Let G be a planar graph and let Π be an embedding of G. Let v_1 and v_2 be two vertices in G. For $i = 1, 2$, let F_i be the set $\{f^* | f$ is a face with v_i on its boundary$\}$. The augmented dual graph of Π, v_1, v_2 denotes the graph obtained from the dual graph Π^* by adding the vertices v_1 and v_2 and inserting the edges (v_1, f_1) for all $f_1 \in F_1$ and (v_2, f_2) for all $f_2 \in F_2$.*

Further a skeleton edge e *represents* a vertex v of G if v is contained in $expansion(e)$ and v is not an endpoint of e. If $L_1 = a_1, \ldots, a_k$ and $L_2 = b_1, \ldots, b_k$ are two lists, we denote with $L_1 + L_2$ the list $a_1, \ldots, a_k, b_1, \ldots, b_k$. The algorithm for computing an optimal edge insertion path for a biconnected planar graph G and two non-adjacent vertices v_1 and v_2 of G is shown in 8.1

Algorithm 8.1. [93] *Computes an optimal edge insertion path for a pair of non-adjacent vertices v_1, v_2 in a biconnected planar graph G.*

procedure $OptimalBlockInserter(graphG, vertexv_1, vertexv_2)$
    Compute the SPQR-tree T of G;
    Find the shortest path $\mu_1, \ldots, \mu_k$ in T between an allocation node
        μ_1 of v_1 and μ_k of v_2;
    for $i = 1, \ldots, k$ **do**
        $S_i := skeleton(\mu_i s)$;
        if v_1 is in S_i, **then**
            $x_i^1 := v_1$;
        else
            Split the edge representing v_1 in S_i by inserting a new vertex y_i^1;
            Mark the two edges produced by the split;
            $x_i^1 := y_i^1$;
        end
        if v_2 is in S_i **then**
            $x_i^2 := v_2$:
        else
            Split the edge representing v_2 in S_i by inserting a new vertex y_i^2;
            Mark the two edges produced by the split;
            $x_i^2 := y_i^2$;
        end
        let G_i be the graph obtained from S_i by replacing each unmarked edge
            with its expansion graph;
        if μ_i is not an R-node **then**
            set p_i to the empty path;
        else
            Compute an arbitrary embedding Π_i of G_i;
            let A_i be the augmented dual graph of Π_i, x_i^1, x_i^2;
            Compute the shortest path $e_0^*, \ldots, e_l^* + 1$ in A_i between x_i^1 and x_i^2;
            $p_i := e_1, \ldots, e_l$, where e_j is the primal edge of e_j^*;
        end
    end
    return $p_1 + \ldots + p_k$;
end

Algorithm 8.1 computes only an edge insertion path $p = e_1, \ldots, e_l$ for the vertices v_1 and v_2 in G. but not the corresponding embedding of G. However, there is a simple way for finding an embedding Π such that p is an edge insertion path for v_1 and v_2 in G with respect to Π. Construct a graph G' by splitting each edge e_i in p introducing a new vertex w_i and insert new edges forming a path $v_1, w_1, \ldots, w_l, v_2$. Since p is an edge insertion path, the graph C is planar and an embedding Π' for G' can be computed in linear time (see, e.g. [104] and [143]). Replacing all split edges in Π' by original edges (thus removing the vertices $w_1, \ldots, w_l$ and their adjacent edges again) results in an embedding Π for G such that p is an edge insertion path for v_1 and v_2 in G with respect to Π.

The algorithm for computing an optimal edge insertion path for a connected planar graph G and two non-adjacent vertices v_1 and v_2 is given in Algorithm 2. The algorithm constructs the block-vertex tree B of G and considers only the blocks on the path from v_1 to v_2 in B. For each block B_i, an optimal edge insertion path p_i for the representatives of v_1 and v_2 in B_i is computed using Algorithm 8.1, and these paths are then concatenated.

Algorithm 8.2. [93] *Computes an optimal edge insertion path for a pair of non-adjacent vertices v_1, v_2 in a connected graph G.*
procedure $OptimalInserter(graphG, vertexv_1, vertexv_2)$
Compute the block-vertex tree B of G;
Find the path $v_1, B_1, c_1, \ldots, B_{k-1}, c_{k-1}, B_k, v_2$ from v_1 to v_2 in B;
for $i = 1, \ldots, k$ **do**
Let x_i and y_i be the representatives of v_1 and v_2 in B_i;
$p_i := OptimalBlockInserter(B_i, x_i, y_i)$;
end
return $p_1 + \ldots + p_k$;
end

The algorithm 8.2 can easily be generalized to arbitrary planar graphs. If v_1 and v_2 belong to the same connected component, simply apply Algorithm 8.2. Otherwise, the graph $G \cup \{(v_1, v_2)\}$ is obviously planar and an empty path is the optimal edge insertion path.

8.1.3 Constrained Crossing Minimization.

Problem 10 (Constrained Crossing Minimization [91]). *Given a connected planar graph $G = (V, E)$, a combinatorial embedding $\Pi(G)$ of G, and a set of pairwise distinct edges $F \subseteq V \times V$, find a drawing of $G' = (V, E \cup F)$ such that the combinatorial embedding $\Pi(G)$ of G is preserved and the number of edge crossings is minimized.*

Obviously, re-insertion of all edges at the same time will improve the solution. However, no practically efficient algorithm is known, since the constrained crossing minimization problem is NP-hard. The proof of Mutzel and Ziegler [149] is based on the NP-completeness of Fixed Linear Crossing Number Problem, shown in [141]. The problem has been investigated in [149, 214]. Experiments show that it can only be solved to provable optimality if there are less than 10 re-inserted edges - and even then, the running time is relatively high.

In practice it is attacked by iterative heuristics using the following observation: If only one edge needs to be inserted, the problem can be solved optimally in polynomial time by computing a shortest path in the combinatorial dual graph extended by some vertices and edges. The heuristics iteratively insert the edges using this dual graph approach. However, the result is not always acceptable and it is supposed that the exact solution of the constrained crossing minimization problem will lead to much nicer drawings [149, 150].

In [150], Mutzel and Ziegler have presented the first step towards an algorithm for solving practical instances of the constrained crossing minimization problem to provable optimality. They have shown that the constrained crossing minimization problem can be formulated as an $|F|$-pairs shortest walks problem, where we want to minimize the sum of the lengths of the walks plus the number of crossings between the walks.

8.1.3.1 The constrained crossing minimization problem using ILP

In [150], Mutzel and Ziegler have shown that the constrained crossing minimization problem can be formulated as a shortest crossing walks problem, which is of rather combinatorial than geometric nature. This allows us to solve practical instances of the constrained crossing minimization problem to provable optimality.

A *walk* in a graph G is an alternating sequence of vertices and edges of G, beginning and ending with a vertex, in which each edge is incident to the two vertices immediately preceding and succeeding it. We denote a walk W between two vertices v_0 and v_1 by $W = v_0\ e_1\ v_1 \ldots e_l\ v_l$. If all the edges of a walk are distinct, we call the walk a *trail*. If all the vertices are distinct, we call the walk a *path*.

Mutzel and Ziegler [149] define the *shortest crossing walks problem* as follows. Given a weighted planar connected graph $G = (V, E)$ with embedding $\Pi(G)$ and a set $F \subseteq V \times V$ of distinct pairs of vertices of G, called *commodities*, find a set of walks in G with the following properties. There is one walk between s_k and t_k for each commodity $k = (s_k, t_k) \in F$, no walk uses an end vertex of a commodity as an internal vertex, and the sum of the weighted lengths of the walks plus the number of crossings between walks is minimum.

Figure 21 illustrates the two kinds of crossings between walks that can appear, called simple and distributed crossings. In a simple crossing two walks cross at one vertex and in a distributed crossing they have a common subsequence.

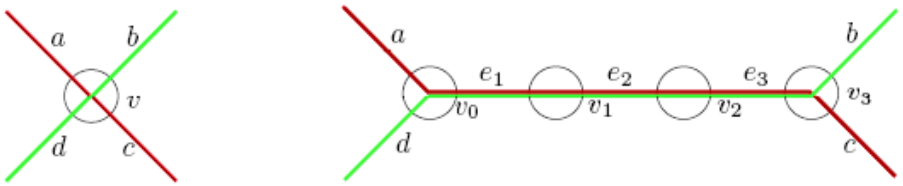

Figure 21: [149] On the left is a simple crossing at vertex $v \in V$, on the right is a distributed crossing with common subsequence $v_0e_1v_1e_2v_2e_3v_3$.

The *combinatorial dual graph* $G^* = (V^*, E^*)$ of a planar graph $G = (V, E)$ with embedding $\Pi(G)$ has a vertex $v^* \in V^*$ for each face f of $\Pi(G)$. Corresponding to each edge $e \in E$ there is an edge $e^* \in E^*$ connecting the two vertices v^* and w^* corresponding

to the faces incident to e. The order of the edges around a vertexv^* in $\Pi(G^*)$ is defined by the order of the edges in G on the boundary of the face corresponding to v^*. This defines the embedding $\Pi(G^*)$ of G^* uniquely.

Given an instance, $(G, \Pi(G), F)$, of the constrained crossing minimization problem we compute the corresponding instance, $(G^*, \Pi(G^*), F^*)$, of the shortest crossing walks problem as follows.

Let G^* be the combinatorial dual graph of G. For every vertex of G that is an end vertex of an edge in F we add a new vertex in the appropriate face of $\Pi(G^*)$. We connect each additional vertex to all vertices on the boundary of the face it was placed in (see Figure 22). The resulting graph G^* is still planar and $\Pi(G^*)$ is uniquely determined by $\Pi(G)$. We call the resulting graph G^* the *extended dual graph*. The set F^* is the set of pairs of additional vertices corresponding to the end vertices of the edges in F.

Mutzel and Ziegler [MZ99.] associate weight 0 with the additional edges, weight 1/2 with the edges replacing a loop, and weight 1 with all the other edges. Using these weights on the edges the value of a solution of the shortest crossing walks problem is the same as the number of crossings in the corresponding solution of the constrained crossing minimization problem.

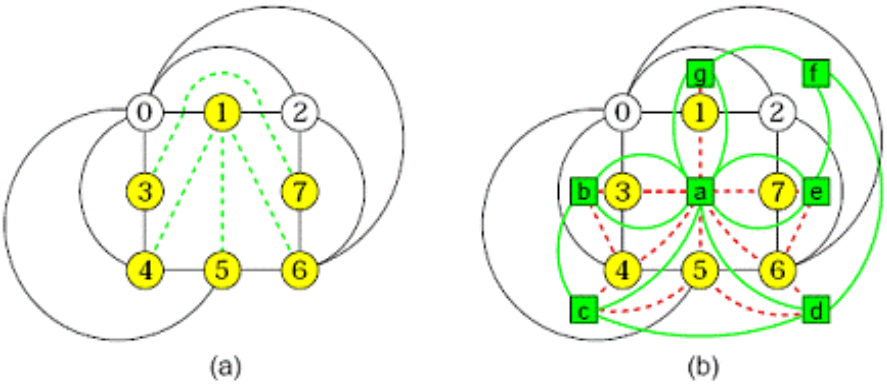

Figure 22: [149] Example of a graph and its extended dual graph.

Figure 22a shows an instance of the constrained crossing minimization problem. Here G is the graph induced by the solid edges and we want to insert the dashed edges with a minimum number of crossings. Figure 22b shows G together with its extended dual graph. The vertices of the combinatorial dual graph are drawn as rectangles and the edges as grey solid lines. The additional vertices are 1,3,4,5,6,7 and the additional edges are drawn as dashed lines.

This example also shows that considering only paths in the extended dual graph is not sufficient to find an optimal solution of the constrained crossing minimization problem. The optimal solution for this example has two crossings (Figure 22a). The best solution using only paths in the extended dual graph needs three crossings.

The basic idea for representing a set of walks in terms of linear inequalities is to use variables for pairs of adjacent edges instead of variables for the edges only. Mutzel and Ziegler [149] start with a few constraints and use the other constraints as cutting planes. They

managed to separate the constraints for simple crossings and cuts exactly in polynomial time. For separation of distributed crossing constraints, they applied a heuristic. (See [149] for more details.)

Although it is much easier to express a set of paths than a set of walks in terms of linear inequalities, an optimal solution to the shortest crossing paths problem is, in general, not an optimal solution of the constrained crossing minimization problem.

However, an ILP for the shortest crossing trails problem can be applied to the shortest crossing walks problem. The only difference is that walks can use edges of the graph more than once. The idea is to replace every regular edge by an appropriate number of parallel copies (without changing the embedding). Then, an optimal solution of the shortest crossing trails problem in the modified graph corresponds directly to an optimal solution of the constrained crossing minimization problem [150]. The full ILP for the shortest crossing trails can be found in [149].

Mutzel and Ziegler [149] performed computational experiments considering 5157 graphs from the benchmark set of graphs given in [6]. By deleting between 2 and 10 edges, for every graph they computed a planar subgraph and a combinatorial embedding of the planar subgraph. Mutzel and Ziegler attempted to compute provable optimal solutions for the shortest crossing paths, the shortest crossing trails, and the shortest crossing walks problem on the extended dual graphs of the benchmark graphs.

The number of crossings computed by the branch and cut algorithm for the trails version improved by about 7% on average compared to the iterative heuristic. Mutzel and Ziegler observed that the number of crossings of the paths version and the walks version differed only slightly from that of the trails version, however, the trails version came out as the most successful.

8.2 Post-Processing Strategies

The idea of the post-processing strategies is to iteratively delete an edge from the drawing and to re-insert it again. The strategies suggested in [91] vary in the set and/or number of edges involved in the deletion and re-insertion process, and the order of processing them.

The variant INS involves exactly those edges which have been deleted already in the planar subgraph step, whereas the variants ALL and MOST involve the whole set of edges E in the original graph G. The variant in which there is no post-processing routine is called NONE.

An iteration takes either the whole set (in variant INS and ALL) or $x\%$ of this edges (variant MOST $x\%$) iteratively (one after the other). The procedure stops only if within one iteration no improvement has been made. In variant MOST $x\%$, after each iteration, the involved edge set is sorted in descending order according to the number of crossings the edges are involved in. Then, only the first $x\%$ edges of this list are taken for re-insertion.

The post-processing procedure can be implemented efficiently by updating the dual graph only at those regions, in which changes did occur. Gutwenger and Mutzel [91] did this for the FIX strategy. In principal, such an update is also possible for the VARIABLE embedding setting [10]. However, there was to the authors' knowledge [91] no implementation of this algorithm. This explains the big running time discrepancy in the post-processing procedure between the FIXED and VARIABLE embedding setting. The results can be found in [91].

8.3 Permutations

After a whole deletion and re-insertion process of the chosen strategy for embedding FIX/ VAR and a strategy for post-processing NONE/ INS/ ALL/ MOST, we get a certain crossing number.

The permutation variant of [91] does nothing else, but repeating the whole edge re-insertion process and keeping the best results. The random effect exists in choosing a different ordering of the edges in $G - P$ for the initial re-insertion step.

The notation PERM; gives the number of these repetition rounds. Gutwenger and Mutzel have experimented with PERM1, PERM2, PERM10, and PERM20. The results are given in [91].

8.4 Computational study

Gutwenger and Mutzel [91] have conducted extensive experimental studies on the crossing minimization problem for a benchmark set of graphs. The main conclusions are:

1. Post-processing always helps. It is recommended not to restrict the postprocessing procedure to the inserted edges. Already re-inserting 25% of all the edges helps a lot.

2. Permutations and random effects help, but not as well as post-processing.

3. It is important to start with a good planar subgraph. A better subgraph leads not only to an improved number of achieved crossings, but also to an improved running time of the algorithm.

4. The re-insertion within a variable embedding setting is still worth doing, even if post-processing is used.

Due to the high running time for the constrained crossing minimization method, Gutwenger and Mutzel did not include this method into their experiments.

9 Algorithms for the rectilinear crossing number

Any drawing where the edges are straight-line segments is called a rectilinear drawing. The goal of rectilinear crossing minimization is to find a rectilinear drawing of G with as few edge crossings as possible. This minimum value is called the rectilinear crossing number $cr(G)$.

The corresponding decision problem is known to be NP-hard. The general crossing number problem is known to be NP-complete [80], but so far no one has determined whether the rectilinear crossing number problem is in NP. This is somewhat surprising, since it is much easier to determine if two straight-line segments intersect than it is to determine if two curved lines do.

In this chapter, we present two algorithms for computing the rectilinear crossing number. The first one is a branch-and-bound algorithm formulating the problem as a mathematical program with a linear objective function and simple quadratic constraints [50]. The other is a genetic algorithm that uses parallel processing in a cooperative fashion to determine mappings for the rectilinear crossing problem [195].

Research in integer and nonlinear programming provides some hope that some useful instances of QCF (Quadratic Constraints Formulation) can be solved. One breakthrough was made by Crowder, Johnson and Padberg [44] who solved binary integer programming (BIP) problems with up to 2,756 variables and no special structure. Roy and Wolsey [203] succeeded in solving mixed BIPs with nearly 1,000 binary variables and an even larger number of real variables.

Dean [50] provides data to show that, for the graphs of most interest, the number of binary and real variables in QCF is evidently still larger than the instances that can be solved by the best solvers. On the other hand, there is hope that experts in the field can somehow take advantage of the special structure or any new algorithmic developments that can be applied to solve these problems.

Many mathematical problems have useful mathematical programming formulations (for example, see [149, 190] for related results). Though there was no such formulation for the rectilinear crossing number problem until Dean [50] presented a new approach to rectilinear crossing minimization including a formulation of the problem as a mathematical program with a linear objective function and simple quadratic constraints.

Genetic algorithms are heuristic search techniques. Hence, the goal of a genetic algorithm is not to necessarily find the optimal solution, but to produce a "satisfactory" solution when searching a complex system [195]. Random choice is used as a tool to guide a genetic algorithm as it searches. As a genetic algorithm produces new generations, better solutions may be discovered. Thus, the power of genetic algorithms lies in their robustness, or ability to adapt, just as in natural systems.

The structure of a genetic algorithm is based on natural selection. First, an initial population of feasible solutions is randomly generated. The initial population consists of chromosomes representing particular encodings of solutions. Reproduction takes place between members of the population, and a child is formed from a combination of the parent chromosomes. For each new child, an evaluation function is used to determine the fitness of that child. Whether or not the child becomes a member of the population depends on its fitness value. Each new child chromosome is compared against the worst member of the population, and the better one is kept in the population.

By producing new generations in this manner, the population improves and the best member of the final population is the solution which is returned by the algorithm.

9.1 Quadratic Constraints Formulation

For any rectilinear drawing of a graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, Dean [50] defined $Area2(i, j, k) = x_i y_j - y_i x_j + y_i x_k - x_i y_k + x_j y_k - x_k y_j$ where (x_i, y_i),(x_j, y_j),(x_k, y_k) are the coordinates for the vertices i, j, k, respectively. Geometrically, $Area2(i, j, k)$ is twice the signed area determined by the triangle ijk [155]; in particular, $Area2(i, j, k)$ is positive if vertex k lies to the left of line ij, negative if it lies to the right, and zero if k lies on the line ij. With a line ij (not a line segment) is meant the two-way infinite directed line determined by the ordered pair of vertices i, j.

For any graph G, let $I(G)$ denote the set of independent (*i.e.*, disjoint) edge sets of size two. Dean [50] formulated the rectilinear crossing minimization as the following problem which he called the *Quadratic Constraints Formulation* QCF(G) for G, or simply QCF when there is no particular graph G under consideration.

Minimize

$$\sum_{1 \le i < j,\ i < k < l,\ \{ij, ki\} \in I(G)} c_{ijkl}$$

subject to

$$\begin{aligned}
Area2(i,j,k) &\le M(1 - c_{ijkl}) + Mt_{ijkl} - 1 && (1)\\
-Area2(i,j,l) &\le M(1 - c_{ijkl}) + Mt_{ijkl} - 1 && (2)\\
-Area2(i,j,k) &\le M(1 - c_{ijkl}) + M(1 - t_{ijkl}) - 1 && (3)\\
Area2(i,j,l) &\le M(1 - c_{ijkl}) + M(1 - t_{ijkl}) - 1 && (4)\\
Area2(k,l,i) &\le M(1 - c_{ijkl}) + Mp_{ijkl} - 1 && (5)\\
-Area2(k,l,j) &\le M(1 - c_{ijkl}) + Mp_{ijkl} - 1 && (6)\\
-Area2(k,l,i) &\le M(1 - c_{ijkl}) + M(1 - p_{ijkl}) - 1 && (7)\\
Area2(k,l,j) &\le M(1 - c_{ijkl}) + M(1 - p_{ijkl}) - 1 && (8)\\
Area2(i,j,k) &\le Mc_{ijkl} + Mt_{ijkl} + Mp_{ijkl} - 1 && (9)\\
Area2(i,j,l) &\le Mc_{ijkl} + Mt_{ijkl} + Mp_{ijkl} - 1 && (10)\\
-Area2(i,j,k) &\le Mc_{ijkl} + M(1 - t_{ijkl}) + Mp_{ijkl} - 1 && (11)\\
-Area2(i,j,l) &\le Mc_{ijkl} + M(1 - t_{ijkl}) + Mp_{ijkl} - 1 && (12)\\
Area2(k,l,i) &\le Mc_{ijkl} + Mt_{ijkl} + M(1 - p_{ijkl}) - 1 && (13)\\
Area2(k,l,j) &\le Mc_{ijkl} + Mt_{ijkl} + M(1 - p_{ijkl}) - 1 && (14)\\
-Area2(k,l,i) &\le Mc_{ijkl} + M(1 - t_{ijkl}) + M(1 - p_{ijkl}) - 1 && (15)\\
-Area2(k,l,j) &\le Mc_{ijkl} + M(1 - t_{ijkl}) + M(1 - p_{ijkl}) - 1 && (16)
\end{aligned}$$

where the variables i, j, k, l satisfy $\{ij, ki\} \in I(G)$, $1 \le i < j, i < k < l$, the functions c, t, p are 0-1 binary variables, and M is a sufficiently large integer constant.

The function c_{ijkl} counts the number of times edges ij and kl cross (*i.e.*, 0 or 1). The objective function counts the total number of crossings for the drawing. The -1 in the constraints insures that $|Area2(i,j,k)| \geq 1$, $|Area2(i,j,l)| \geq 1$, $|Area2(k,l,i)| \geq 1$, and $|Area2(k,l,j)| \geq 1$. Rescaling can be used to make all nonzero areas satisfy this condition, and so the -1 really only ensures that no vertex lies on a line to which it does not belong; in particular, only good drawings are considered.

Suppose $c_{ijkl} = 1$. Then constraints (9)-(16) become irrelevant, and constraints (l)-(4) ensure that points k and l lie on opposite sides of the line ij. This holds because $t_{ijkl} = 1$ implies that k and l lie to the left and right of line ij(respectively), and $t_{ijkl} = 0$ implies that k amd l lie to the right and left of line ij(respectively). Similarly, constraints (5)-(8) ensure that points i and j lie on opposite sides of the line kl. The assignment $p_{ijkl} = 1$ implies i is on the left and j is on the right of kl and the assignment $p_{ijkl} = 0$ implies i on the right and j is on the left. Therefore, the line segments ij and kl must cross.

Now suppose $c_{ijkl} = 0$. Then constraints (l)-(8) are irrelevant, and either $p_{ijkl} = 0$ which ensures that points k and l lie on the same side of the line ij (see constraints (9)-(12)) or $p_{ijkl} = 1$ which ensures that points i and j lie on the same side of the line kl (see constraints (13)-(16)). Thus, ij and kl do not cross.

Fixing the binary variables in a quadratic constraints formulation QCF(G) reduces to the problem of deciding whether, for the given graph G and a given set of crossings (and non-crossings), a set of coordinates exists for a rectilinear drawing of G that is consistent with these constraints [50]. This problem is called the QCF Realization Problem, and turns out to be NP-hard as explained below.

A pseudoline is a homeomorph in R^2 of the closed unit interval. In an *arrangement* of pseudolines every two lines meet at exactly one interior point where they must cross. Bienstock [14] proved that any given arrangement of p pseudolines can be forced to occur in every crossing minimal drawing of an appropriate graph, with $O(p^3)$ edges and $5p(p-1)$ crossings by using some results of Goodman, Pollack, and Sturmfels [87] on arrangements whose straight-line realizations require vertex coordinates with exponentially many bits.

Therefore, assuming that *the coordinates must be stored and that drawing a graph consumes time proportional to the size of the drawing*, this yields that there is no polynomial-time algorithm for producing a rectilinear drawing of a graph which achieves the rectilinear crossing number.

9.2 Genetic algorithms

The algorithm by Thorpe and Harris [195] used for the rectilinear crossing problem is basically a genetic algorithm with some modifications that enhance its use on a parallel processing system. First, eight nodes of a parallel processing machine (iPSC/2) are allocated to run the genetic algorithms. Each node generates its own initial population, and begins executing the genetic algorithm. Each iteration of the genetic algorithm produces two children to evaluate and possibly insert into the node's population. Every so often, a "mutant" is generated and inserted into the population. Mutation is a technique to help prevent stagnation of the population. Once the genetic algorithm meets one of its convergence criteria (time limit, number of iterations, difference in the number of crossings between the best and the worst solutions), the algorithm halts and broadcasts its results

to the host program. If all nodes have converged to the same crossing number, the host stops and reports the results.

Thorpe and Harris [195] use no strategy involved in creating initial populations. The graph vertices are randomly placed in a rectangular area, and the number of crossings for each resulting graph are determined. After the initial population is generated, the genetic algorithm selects two sets of parent "chromosomes" to recombine. One set is chosen via a simple linear bias, the other is chosen from a normal distribution of the best ten percent (10%) of the population.

A pollenation rate, P, is set as a parameter to the program, and once every P iterations, each genetic algorithm sends a solution to the host program. The host then chooses the best solution and broadcasts it back to all the nodes to use in their recombination. According to an previous observation given in [100], this method of cross-pollenation provides better results then simply running eight genetic algorithms independently. Though, when cross-pollenation occurs too frequently, the populations tend to converge very quickly, and rarely do they produce a "good" result.

Thorpe and Harris [195] use a recombination technique based on uniform order-based crossover presented by Davis [48]. According to this recombination, a chromosome is a bit string that represents the contribution of the parents to a child, see Fig.23. A one in the bit string indicates that the vertex corresponding to that bit index will contribute to the construction of child 1, and a zero indicates that the vertex will contribute to child 2. This bit string is generated randomly for each generation with parent 1 receiving the bit string and parent 2 receiving its complement. Hence, child 1 is composed of vertices marked with ones and child 2 is made of vertices marked with zeroes.

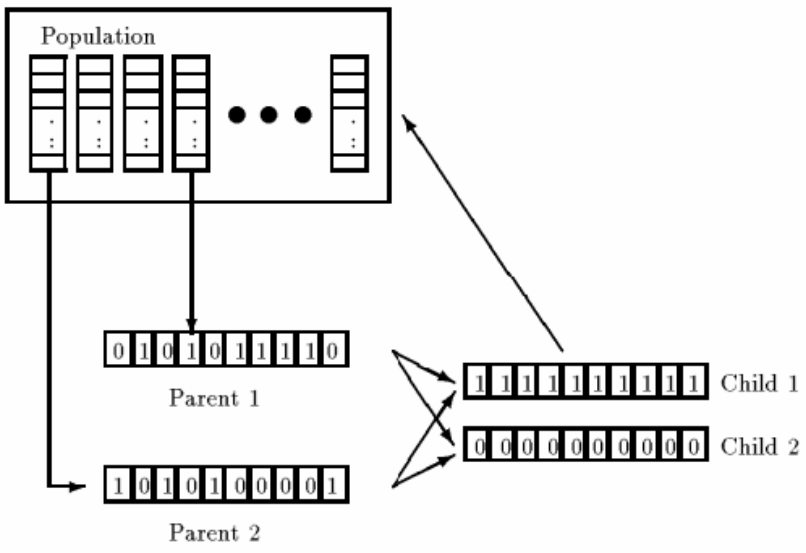

Figure 23: [195] Recombination of Chromosomes.

The cost function of Thorpe and Harris for the rectilinear crossing problem proceeds as follows. All edges are checked in a pairwise fashion such that no edge is compared to itself, and no pair of edges are compared more than once. If a pair of edges cross, then the crossing number is increased by one. Whenever the cost evaluation function detects

colinear edges (something that violate the problem constraints) a large penalty is invoked. Solutions containing large penalty values tend to drop out of the population very quickly.

Thorpe and Harris [195] observed that in the genetic algorithms they used, there appears to be no real correlation between the "goodness" of a solution and how long it takes a genetic algorithm to produce it. Much of the disparity between the results of each execution lied in the use of randomness as a tool to guide the search. Some of the disparity was a result of the error checking and cost evaluation functions required by the minimal crossing problem.

Furthermore, the performance of a traditional exhaustive search method is known to have a speedup linearly proportional to the number of processors working in concert on the problem. Genetic algorithms do not provide any speedup; however, the results of Thorpe and Harris suggest that genetic algorithms are a valuable search tool.

10 Algorithms for the (fixed) linear crossing number

Many real-life scheduling, routing and location problems can be formulated as combinatorial optimization problems whose goal is to find a linear layout of an input graph in such a way that the number of edge crossings is minimized.

In a linear embedding, vertex adjacency is established by either an upper or a lower edge with respect to the horizontal node line. Hence, the routing of edges either above or below the horizontal node line determines the number of crossings in the embedding.

The ordering of vertices along the node line also affects the minimum number of crossings obtainable, that is, once an ordering is fixed, the minimum number of crossings for that ordering is not necessarily the global optimum for the graph. Hence, finding a good vertex ordering is a critical subproblem.

In this chapter, we present algorithms for computing the crossing number of linear embedded graphs for both the free (section (10.1) and the fixed (section 10.2) ordering of vertices. Moreover, we propose two new heuristic algorithms based on the simulated annealing scheme for computing both the free and the fixed linear crossing number, respectively. We also present an experimental study comparing our new heuristics with the existing ones.

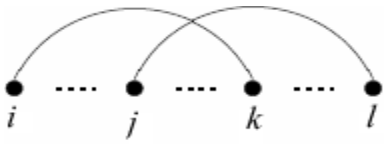

Figure 24: [39] Edge crossing condition $i < j < k < l$.

Counting the number of crossings. Let $n = |V|$ where $V = 1, 2, \ldots, n$. A 2-page drawing of a graph is represented by a pair of binary adjacency matrices $A[]$ and $B[]$. For each edge ij, $A[i, j](B[i, j])$ is 1 if ij is embedded in the upper (lower) page and 0, otherwise. Then any pair of edges ik and jl cross in a drawing iff $1 \leq i < j < k < l \leq n$ and both lie in the same page (see Figure 24). Hence, the following formula [39] counts the number of crossings in a 2-page drawing D :

$$cr_2(D) = \sum_{i=1}^{n-3} \sum_{j=i+2}^{n-1} \{A[i,j] \sum_{k=i+1}^{j-1} \sum_{l=j+1}^{n} A[k,l] + B[i,j] \sum_{k=i+1}^{j-1} \sum_{l=j+1}^{n} B[k,l]\}$$

10.1 Algorithms for the linear crossing number

The following heuristic algorithms try to find a linear embedding of a graph with the minimum number of crossings both by permuting its vertices along the node line and by switching its edges between the two pages.

In addition to the Nicholson's heuristic, in this section, we propose two new algorithms for computing the linear crossing number based on simulated annealing. According to our experiments, only one was able to improve solutions obtained with the Nicholson's heuristic.

10.1.1 Nicholson's heuristic

Nicholson's heuristic is essentially a two-phase "greedy" method for finding a linear embedding of a graph with the minimum number of crossings.

In the first phase, vertices are placed along a horizontal node line in the plane. The first vertex placed is one with maximum degree in the graph. Thereafter, the next vertex selected for placement is one with maximum adjacencies to the vertices already placed. The vertex is placed in the position on the node line for which the increase in the number of crossings is least, with its edges drawn as arcs above or below the node line, according to whichever route offers fewer crossings.

In the second phase, vertices are moved to different positions along the node line, modifying the routing of the edges appropriately. The next vertex selected for trial movement is the one with the most crossings associated with its incident edges, and it is moved to the position which offers the largest reduction in the number of crossings. Vertices are moved until no further improvement is possible.

The time complexity of the method is $O(n^3)$. Further details are given in [152].

10.1.2 Simulated annealing

Simulated annealing is a well-known optimization technique. It is based on the analogy with annealing in physics, where the changes of the system state are seen as essentially random, but changes that reduce the energy are more likely than those that increase it. In addition changes that increase the energy are more likely while the temperature is high than later when the temperature is low.

Proposed algorithms.

Simple algorithm for computing the crossing number on linear embeddings of an arbitrary graph is a customized version of a standard simulated annealing algorithm which appears in [154]. The pseudocode scheme for the algorithm SA may be outlined as follows:

Algorithm 10.1.
procedure *SimulatedAnnealing*
1. start with an initial state and temperature
2. **while** outer loop criterion not satisfied **do**
    1. **while** inner loop criterion not satisfied **do**
        1. select a random neighbor from neighborhood
        2. **if** neighbor has lower cost **then** transition
        3. **if** neighbor has higher cost **then** transition with probability $e^{-\frac{difference}{temperature}}$
    2. Reduce temperature
3. **return** best state seen

In our proposed algorithms for computing the linear crossing number, we specified the main entities as follows:

The initial state is represented by vertices of a graph in a random order with edges randomly placed in the upper or in the lower half-plane. The cost function for a state is the number of crossings. The temperature is reduced to a fraction r of its previous value each time the inner loop finished.

The inner loop criterion is $k * size(neighborhood)$. Neighbouring states are those that differ from the current state in having one of their edges embedded into a opposite half-plane, thus size of the neighborhood is essentially the number of edges, *i.e.* $O(n^2)$.

The outer loop terminates when the cost does not improve for a fixed number $crit_{outer}$ of iterations in a row.

We propose two modified versions of the simple algorithm:

The first algorithm (SA1) has a modified inner loop. It can change the position of a vertex in the vertex ordering, or move an edge to an opposite half-plane. This is set to happen with a probability $1 : n$.

The second algorithm (SA2) works in two phases: In the first phase, it determines the initial vertex ordering by using Nicholson's heuristic. In the second phase, it attempts to find an optimal solution using a simple simulated annealing algorithm switching only edges between the two half-planes.

Time complexity of SA1. First, we shall analyze the complexity of the inner loop.

1. With a probability $\frac{1}{n}$, a vertex is moved to a different position, which forces us to count how many crossings the vertex is responsible for at its old and new position, which takes $O(n^3)$

2. In the remaining cases when an edge is moved to an opposite half-plane, we have to count how many crossings the edge is responsible for in each half-plane, which takes $O(n^2)$.

Thus, the amortized time complexity of the inner loop is $\frac{1}{n}O(n^3) + \frac{n-1}{n}O(n^2)$, which yields $O(n^2)$. The running time of algorithm SA1 is essentially the number of inner loop iterations times the number ot outer loop iterations. The outer loop stops iterating when the results stabilize.

Time complexity of SA2. The first phase takes $O(n^3)$ (Nicholson's heuristic). The inner loop of the second phase takes $O(n^2)$ – this follows from the analysis of the time complexity of SA1. The overall complexity of algorithm SA1 depends on how many times the inner loop is executed, *i.e*, on the outer loop criterion.

10.1.3 Experimental results

Implementation details for SA1 and SA2. The $crit_{inner}$ was assigned to $10 * n^2$ iterations in one iteration of the outer loop. The outer loop criterion was set to 20. Initial system temperature was set to n. The temperature cooling schedule was set to $r = 0.7$.

In our experiments, we ran the two proposed algorithms SA1 and SA2 together with Nicholson's heuristic (NI) for comparison. We conducted the tests on random graphs with different edge densities (see tables 2, 3 and 4). Since our algorithms SA1 and SA2 are random, we ran each of them 10 times on a single graph to obtain a better impression on how they performed.

Our first algoritm, SA1, was not successfull enough, it appears that frequent vertex reorderings interferred unfavourably with the cooling process.

Table 2: Number of crossings for random graphs with edge probability of 0.5.

	NI	SA1			SA2		
		min	max	avg	min	max	avg
rg_{10}	2	1	2	1.2	2	5	2.3
rg_{15}	72	83	102	95.0	68	87	74.9
rg_{20}	210	329	378	359.6	197	199	197.2
rg_{25}	584	897	959	930.7	562	606	568.6
rg_{30}	2148	2933	3047	2984.8	2032	2150	2076.5
rg_{35}	3475	5241	5479	5391.1	3393	3396	3394.1
rg_{40}	6317	9951	10286	10145.5	6222	6243	6233.5
rg_{45}	10604	16739	17029	16863.0	10415	10433	10423.9
rg_{50}	16162	24156	24674	24478.8	15826	15854	15840.3

Table 3: Number of crossings for random graphs with edge probability of 0.3.

	NI	SA1			SA2		
		min	max	avg	min	max	avg
rg_{10}	0	0	0	0.0	0	0	0.0
rg_{15}	11	16	23	19.8	11	28	12.7
rg_{20}	74	118	136	125.1	71	92	79.3
rg_{25}	142	233	267	255.5	130	168	141.3
rg_{30}	566	903	951	932.4	522	621	540.4
rg_{35}	1072	1913	1992	1954.8	1006	1020	1010.4
rg_{40}	1363	2480	2577	2515.3	1276	1494	1302.1
rg_{45}	3792	5686	6178	6001.2	3593	3674	3615.8
rg_{50}	5229	8853	9236	9063.7	5160	5173	5165.6

Table 4: Number of crossings for random graphs with edge probability of 0.8.

	NI	SA1			SA2		
		min	max	avg	min	max	avg
rg_{10}	32	35	40	37.6	32	34	32.3
rg_{15}	200	272	285	277.9	194	228	205.0
rg_{20}	952	1328	1357	1346.8	932	949	939.7
rg_{25}	2458	3503	3607	3565.6	2428	2719	2515.0
rg_{30}	5364	7613	7801	7730.0	5243	5258	5248.4
rg_{35}	9892	14034	14239	14128.3	9754	9770	9757.4
rg_{40}	19320	27517	27774	27646.2	19026	19046	19032.5
rg_{45}	30237	43848	44511	44205.0	29937	32371	30423.4
rg_{50}	47248	67526	68271	67954.7	46769	50017	47096.6

On the other hand, Algorithm SA2 found a better or an equally good solution as NI. For most of the tested graphs, even the average number of crossings was better than the one obtained with NI.

Although the running times of SA1 and SA2 cannot be guaranteed, our experiments suggest that they are polynomial (with the settings used, it is $yO(n^4)$, where y is the number of times the outer loop is executed).

10.2 Algorithms for the fixed linear crossing number

In the restricted version of the Linear Crossing Number Problem called the Fixed Linear Crossing Number Problem (FLCNP), the order of vertices along the node line is predetermined and fixed. FLCNP belongs to the class of NP-hard optimization problems [141].

In section 10.2.1, we present two exact algorithms for solving this NP-hard problem. The first is a simple branch-and-bound algorithm (section 10.2.1.1). The latter approach (section 10.2.1.2) is based on a reduction of FCLNP to the Maximum Cut Problem.

Heuristic algorithms are described in section 10.2.2, including an experimental study evaluating the performance of the heuristics. We propose a simple new heuristic based on simulated annealing, performing even better then the best heuristic from the literature, albeit at the cost of a higher complexity.

10.2.1 Exact algorithms

The exact algorithm used by Cimikowski [39] to solve the problem FLCNP is based on the branch-and-bound technique: basically, all possible solutions of the problem are enumerated.

The set of solutions is given by a binary enumeration tree, where each inner node corresponds to a decision whether a chosen edge is drawn above or below the horizontal

line. In the worst case, an exponential number of solutions has to be enumerated.

However, the basic idea of branch-and-bound is the pruning of branches in this tree: at some node of the tree, a certain set of edges is already fixed. According to this information, one can derive a lower bound on the number of crossings subject to these fixed edges. If this lower bound is at most as good as a feasible solution that has already been found, e.g., by some heuristics, it is clear that the considered subtree cannot contain a better solution, so it does not have to be explored.

Buchheim and Zheng [22] show that this NP-*hard* problem can be reduced to the well-known maximum cut problem. The latter problem was intensively studied in the literature; efficient exact algorithms based on the branch-and-cut technique have been developed.

By an experimental evaluation on a variety of graphs, [22] show that using this reduction for solving FLCNP compares favorably to the earlier branch-and-bound algorithm. Moreover, testing the existence of a planar fixed linear embedding of a given graph can be done in an easy way using this transformation.

10.2.1.1 A branch-and-bound algorithm

The number of fixed linear layouts of a graph is 2^m. Ignoring up to n insignificant edges, this yields at most $2^{m(n-3)/2}$ different layouts. Since any layout has a "mirror image" (symmetric) drawing with the same number of crossings obtained by switching the embeddings between the two pages, only one half of this number need be checked, or $2^{m(n-3)/2-1}$.

Cimikowski [39] developed a branch-and-bonnd algorithm to find optimal solutions by enumerating all possible embeddings of edges subject to optimization bound conditions. Two bounding conditions were applied to prune partial solution paths in the search tree. A path (branch) of the tree is pruned if:

(i) the number of crossings in the partial solution exceeds the current global upper bound, or

(ii) the number of crossings in the partial solution plus the number of extra crossings resulting from adding each remaining edge to the partial embeddding greedily and independently of other remaining edges exceeds the current global upper bound.

Cimikowski developed a backtracking algorithm to enumerate the embeddings and apply the bounding conditions. An initial global upper bound was obtained from the best solution generated by the theoretical bounds and the heuristics. As with the heuristics, the output of the algorithm is the number of crossings obtained and the corresponding embedding.

This algorithm is a practical choice for graphs with up to approximately 50 edges, although its performance is highly dependent on the quality of the initial upper bound value.

10.2.1.2 FCLNP by reduction to the maximum cut problem

Buchheim and Zheng [22] have presented a new exact algorithm for the Fixed Linear Crossing Number Problem, running significantly faster than earlier exact algorithms. The essential part of their approach is the reduction to the maximum cut problem given as follows:

Problem 11 (**Maximum Cut Problem (MAXCUT)**). *Given an undirected graph $G' = (V', E')$, find a partition of V' into disjoint sets V_1 and V_2 such that the number of edges from E that have one endpoint in V_1 and one endpoint in V_2 is maximal.*

For an instance of FLCNP. i.e., a given graph $G = (V, E)$ with a fixed vertex permutation, we construct the associated *conflict graph* $G' = (V', E')$ as follows: the vertices of G' are in one-to-one correspondence to the edges of G, i.e., $V' = E$. Two vertices of G corresponding to edges $e_1, e_2 \in E$ are adjacent if, and only if, e_1 and e_2 are potentially crossing. See Fig. 25 for an illustration.

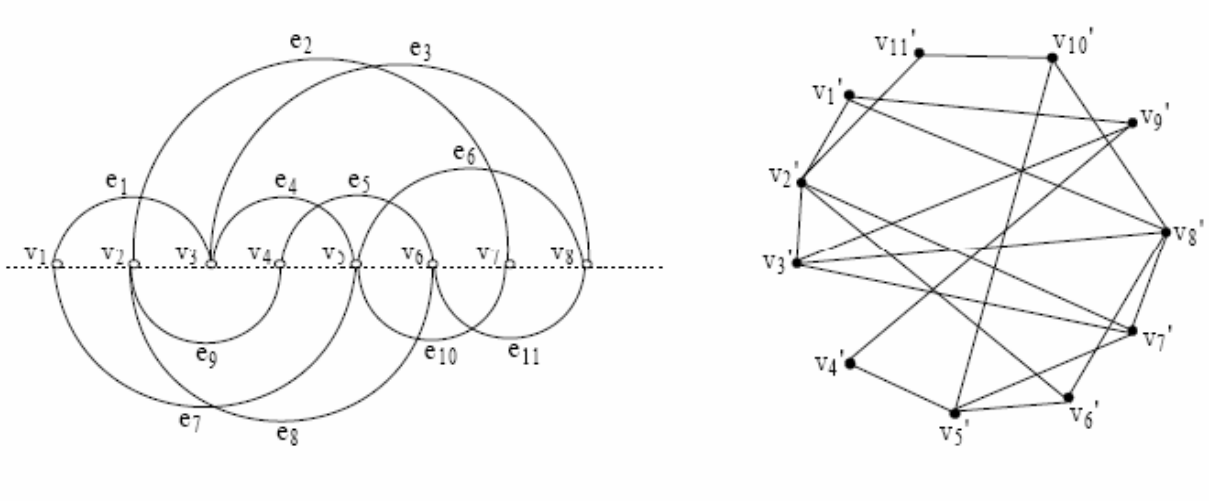

Figure 25: [22] Graph G and its associated conflict graph G'.

Definition 8 (**Cut embedding**). *Let G be a graph with a fired vertex permutation. Given a vertex partition (V_1, V_2) of its conflict graph G', the associated cut embedding is the fixed linear embedding of G where edges corresponding to V_1 and V_2 are embedded to the half spaces above and below the vertex line, respectively.*

Theorem 14 (**Buchheim and Zheng** [22]). *Consider a partition (V_1, V_2) of V'. Then the corresponding cut embedding is a fixed linear embedding of G with a minimum number of crossings if, and only if, (V_1, V_2) is a maximum cut in G'.*

Theorem 15 (**Buchheim and Zheng** [22]). *For a graph $G = (V, E)$ with a fixed vertex permutation, there is a planar fixed linear embedding of G if, and only if, the associated conflict graph G' of G is bipartite.*

Observe that testing whether the graphG' is bipartite can be done in linear time (with respect to G') by two-coloring a DFS-tree.

By Theorem 14, we can use any algorithm for MAXCUT in order to solve FLCNP. One of the most successful approaches for solving MAXCUT to optimality in practice is branch-and-cut.

Buchheim and Zheng [22] first model the problem as an integer linear program (ILP) and solve as a linear program (LP), i.e., the integrality constraints are relaxed. LPs are solved very quickly in practice. If the LP-solution is integer, we can stop. Otherwise, one tries to add cutting planes that are valid for all integer solutions of the ILP but not necessary for (fractional) solutions of the LP. If such cutting planes are found, they are added to the LP and the process is reiterated.

One has to resort to the branching part only if no more cutting planes are found. In general, only a small portion of the enumeration tree has to be explored, as many branches can be pruned. Compared to a pure branch-and-bound approach as presented in [39], the number of subproblems to be considered is very small in general. This, however, depends on the quality of the cutting planes being added.

The latter in turn depend on the specific problem; finding good cutting planes is a sophisticated task. Fortunately, the MAXCUT problem has been investigated intensively, so that many classes of cutting planes are known. More detailed information on algorithms for MAXCUT using cutting plane techniques can be found in [124, 134].

Observe that MAXCUT can also be adressed by semidefinite programming methods; see e.g. [125]. These methods perform well on very dense instances, while being outperformed by ILP approaches on sparse or large graphs. Therefore, practical instances tend to be easy for the ILP approach of [22].

10.2.1.3 Experimental Results

In order to evaluate the practical performance of the new exact approach to FLCNP presented in the previous section, Buchheim and Zheng [BZ06.] performed extensive experiments. In these experiments, they compared the performance of their approach to the results obtained with the branch-and-bound algorithm proposed by Cimikowski [39] on the same set of test instances as used in [39].

These instances mainly arise from network models of computer architectures; in general they are harniltonian. The fixed order of nodes, as part of the input of FLCNP, is then determined by a hamiltonian cycle in the graph, as an ordering of the vertices along a hamiltonian cycle tends to yield a smaller number of crossings in general. In their experiments, [22] always used the same ordering as chosen in [39] for ensuring comparability.

More specifically, the networks considered are the following, see also [39]:

(i) *complete graphs* K_n for $n = 5, ..., 13$

(ii) *hypercubic networks:* this class of graphs includes the *hypercubes* Q_d and several derivatives of hypercubes such as the *cube-connected-cycles* CCC_d, the *twisted cubes* TQ_d, the *crossed cubes* CQ_d, the *folded cubes*FLQ_d, the *hamming cubes* HQ_d, the *binary de Bruijn graphs* DB_d and the *undirected de Bruijn graphs* UDB_d, the *wrapped butterfly graphs* WBF_d and the *shuffle-exchange graphs* SX_d

(iii) other interconnection networks, including the $d \times d$ textit tori $T_{d,d}$, the *star graphs* ST_d, the *pancake graphs* PK_d, and the *pyramid graphs* PM_d

(iv) *circular graphs:* the circular graph $C_n(a_1, \ldots, a_k)$ is regular and hamiltonian.

Figure 26 shows the runtime results of the algorighm of [22] compared with those of the branch-and-bound algorithm presented in [39]. All running times are given in CPU seconds.

instance	B & B	MAXCUT
Q_4	0.01	0.00
CCC_3	0.02	0.00
SX_4	0.01	0.00
FLQ_4	0.13	0.42
UDB_5	0.43	0.07
$C_{26}(1,3)$	0.46	0.00
$T_{6,6}$	1.27	0.04
CCC_4	2.59	0.01
K_{10}	2.27	3.21
SX_5	2.16	1.84
$C_{20}(1,2,3)$	16.69	0.39
$T_{7,7}$	64.89	0.15
$C_{22}(1,2,3)$	73.16	0.39
K_{11}	148.21	24.56
Q_5	612.35	1.67
K_{12}	1925.51	79.15
K_{13}	> 86400.00	2119.12

Figure 26: Running times for exact approaches.

As obvious from Figure 26, the new approach of [22] is much faster than the branch-and-bound algorithm. This is particularly true for sparse instances, e.g., Q_5. However, the new approach outperforms [39] also on the larger complete graphs. [22] remark that many instances can be solved very quickly by their new approach while others cannot even be solved in one CPU day. Allegedly, the border line between easy instances (those solvable within 25 seconds, say) and hard ones (those unsolved even in one day) is very sharp, few instances do not fall into one of these categories.

The results of [22] can also help to evaluate the quality of the heuristic methods. In fact, it turns out that many heuristics proposed by [39] are able to find optimal or near-optimal solutions even for larger instances. In summary, [22] assume that small to medium sized instances should be solved to optimality in general, whereas for larger instances one can be confident that the heuristic solution is not too far away from the optimum.

10.2.2 Heuristics for the fixed linear crossing number

In this section, we present 8 heuristics designed by Cimikowski [39]. Additionally, we compare their performance with our proposed heuristic based on the simulated annealing scheme.

Preprocessing. Assume that vertices are fixed in the order $1, 2, \ldots, n$ along the node line. As a pre-processing step to each algorithm, all insignificant edges can be removed [39].

Observe that edges between consecutive vertices on the node line and the edge $1n$ cannot be involved in crossings according to the constraints of the problem.

Also, if there is a vertex k, such that no edge ij, $i < k < j$ exists, then edges $1k$ and kn cannot cause crossings. Hence, these edges are insignificant and may be ignored without affecting the final solution. At the same time, the problem size is reduced so that larger instances can be solved.

The output of each heuristic is the minimum number of crossings obtained and the corresponding embedding.

10.2.2.1 Greedy Heuristics

The *Greedy* heuristic [39] adds edges to the layout in *row-major order* of the adjacency matrix of the graph, that is, first all edges $1i$ are added in increasing order of i-value, then all edges $2i$ in increasing i-value order, etc. At each step, an edge is embedded in the page (upper or lower) which results in the smallest increase in the number of crossings. Ties are broken by placing the edge in the upper page.

Heuristic *Gr-ran* [39] uses the same approach but adds edges in random order.

10.2.2.2 Maximal Planar Heuristic

Heuristic *Mplan* [39] finds a maximal planar subgraph in each page. In the first phase, edges are added in row-major order of the adjacency matrix to the upper page. If an edge causes a crossing, it is put aside until the second phase. In the second phase, all edges put aside in the first phase are added to the lower page. If an edge causes a crossing, it is once again put aside. In the third phase, any edges put aside in the second phase are added to the page with the smallest increase in crossings.

10.2.2.3 Edge-Length Heuristic

Heuristic *E-len* [39] initially orders all edges non-increasingly by their "length", i.e., $|u-v|$ for edge uv. The intuition here is that longer edges have a greater potential for crossings than shorter edges and hence should be embedded first. Each edge is added one at a time to the page of smallest increase in crossings.

10.2.2.4 One-Page Heuristic

This is essentially the method described in [180] and implied by $cr_k(G) \leq cr_1(G)/k$ for $k = 2$. Heuristic *1-page* [39] initially embeds all edges in the upper page.This is followed by a "local improvement" phase in which each edge is moved to the lower page if it results in fewer crossings. Edges are considered for movement in order of non-increasing *local crossing number*, *i.e.*, the number of crossings involving an edge.

10.2.2.5 Dynamic Programming Heuristic

Unfortunately, FLCNP does not satisfy the *principle of optimality* which says that in an optimal sequence of decisions each subsequence must also be optimal. Subgraphs embedded optimally earlier in the process do not necessarily lead to optimal embeddings of larger subgraphs when edges are added between the smaller subgraphs later on.

However, this does not preclude the potential benefit of a dynamic programming approach to the problem as a heuristic solution. If most crossings are localized within relatively small subgraphs along the node line for a given graph, a dynamic programming method [39] may produce a good solution.

Let $G_{i..j}$ denote the subgraph induced by consecutive vertices $i..j$ along the node line, and let $cr[i,k,j]$ be the number of crossings in the subgraph $H = G_{i..k} \cup G_{k+1..j} \cup E_{i,k,j}$ where $E_{i,k,j}$ is the set of "link edges" between $G_{i..j}$ and $G_{k+1..j}$. We compute $cr[i,k,j]$ greedily by adding each link edge to the page with the smallest increase in crossings. This leads to a recurrence for the number of crossings, $nc[1,n]$, computed by a dynamic programming solution: $nc[i,j] = \begin{cases} 0 & \text{if } j-i \leq 3 \\ min_{i \leq k < j}\{nc[i,k] + nc[k+1,j] + cr[i,k,j]\} & \text{if } i < j-3 \end{cases}$

The base cases for the algorithm are the subgraphs of order 2-4, Optimal embeddings for these are predetermined and shown in Figure 27.

Figure 27: [39] Fixed embeddings for base cases of dynamic programming heuristic.

10.2.2.6 Bisection Heuristic

This heuristic [39] uses a straightforward divide-and-conquer approach. The original graph $G_{i..n}$ is initially bisected into two smaller subgraphs $G_{1..\lfloor \frac{n}{2} \rfloor}$ and $G_{\lfloor \frac{n}{2} \rfloor..n}$ by temporarily removing the link edges between them. Each subgraph is then bisected recursively in the same manner until subgraphs of order 4 or less are obtained. Embeddings for these base cases are the same as those shown in Figure 27.

When combining smaller subgraphs, link edges between the subgraphs are embedded in greedy fashion as before. A similar method is described in [13], although the way in which edges are re-inserted into the embedding after the bisection phase is not clearly specified.

10.2.2.7 Neural Network Heuristic

This heuristic [39] is based on the neural network model of parallel computation [106]. In this model there is a large number of simple processing elements called "neurons". Cimikowski assumes the McCulloch-Pitts binary neuron [211] in which each element has a binary state . For testing purposes, a sequential simulator of the actual parallel algorithm was used.

The model uses $2m$ neurons for a graph with m edges. With each edge is associated an "up" and a "down" neuron, representing the two pages of the plane. Briefly, two kinds of forces, "excitatory" and "inhibitory", are present in the neural network. The presence of an edge uv in a graph encourages the two neurons for the edge to fire as the excitatory force, while neurons of crossing edges are discouraged from firing as the inhibitory force.

At each iteration of the main processing loop, neuron values are recalculated according to specified motion equations.

Eventually, after several iterations, either an "up" or "down" neuron for each edge is in an excitatory state and a final embedding is obtained. Empirical testing shows that the algorithm always converges to a solution within 50 iterations.The parallel time complexity of the algorithm is $O(1)$ for a neural network with n^2 processing elements, where n is the number of vertices of the graph.

It is straightforward to simulate the parallel algorithm with a sequential algorithm [39]. Whereas, in the parallel algorithm, the output values of the neurons are simultaneously updated outside of the motion equation loop, in the sequential simulator, the output value of each neuron is individually computed in sequence as soon as the input of the neuron is evaluated inside of the motion equation loop.

A drawback to neural network algorithms is the possibility of non-convergence. Typically, a constant limit is imposed upon the number of iterations of the motion equation computation loop, and the process is terminated if convergence to the equilibrium state has not occurred by the limit. Full details of the heuristic are given in [40].

An extension of this method was suggested in [198], where Tomaštik presented a neural network algorithm computing a k-page book crossing number and embedding.

10.2.2.8 Simulated annealing

Simulated annealing is a well-known optimization technique. It is based on the analogy with annealing in physics, where the changes of the system state are seen as essentially random, but changes that reduce the energy are more likely than those that increase it. In addition, changes that increase the energy are more likely while the temperature is high than later when the temperature is low.

This heuristic is essentially our algorithm SA1 proposed in section 10.1.2. The only difference lies in the fact that in this version, there are no vertex permutations at all, *i.e.*, every outer loop iteration has the same inner loop exchanging exclusively edges between the two half-planes.

10.2.2.9 Time complexity

The time complexities of the heuristics are given in Table 5. For the greedy, maximal planar, edge-length, and one-page heuristics, the total time is dominated by the time to calculate the number of crossings after each edge is added to the layout, which is $O(n^4)$ if eq. (1) is directly applied. Instead, however, we use a "dynamic" crossing recalculation method which only checks for crossings involving the edge just added. This lowers the recalculation time to $O(m)$ for each edge added.

For the dynamic programming heuristic, there are a total of $O(n^2)$ subgraphs to process, and each subgraph requires $O(m^2)$ time to add link edges and recalculate crossings. The time for the bisection heuristic is given by the recurrence $T(n) = 2T(\frac{n}{2}) + O(n^4)$, where $O(n^4)$ is the time to merge each pair of subgraphs, and this recurrence has the solution $O(n^4)$.

The sequential simulator of the neural network heuristic has a main loop with a number of iterations dependent on the rate of convergence of the system to a stable state,

Table 5: Time Complexities of the Heuristics.

Heuristic	Time Complexity
Greedy	$O(m^2)$
Gr-ran	$O(m^2)$
Mplan	$O(m^2)$
E-len	$O(m^2)$
1-*page*	$O(m^2)$
Dynamic	$O(m^2)$
Bisection	$O(m^2n^2)$
Neural	$O(m)$
Annealing	$rO(n^4)$

which is not bounded by any function of the input size. However, in [39], Cimikowski claimed that the maximum number of loop iterations observed for any test graph was 84. There are $O(1)$ operations performed on each of the m edge neurons per iteration. Hence, the time complexity is $O(m)$.

The simulated annealing uses 2 nested loops. The inner loop is repeated $O(n^2)$ times and takes $O(n^2)$ time. Number of iterations of the outer loop is variable, we shall denote it as y. Hence, the time complexity is $yO(n^4)$. Although, a bound for y cannot be given, experiments suggest that it is polynomial.

10.2.2.10 Experimental results

Inspired by Cimikowski [39, 40], we performed an experimental study of 9 different heuristics for the *fixed linear crossing number problem* (FLCNP).

The tests were conducted on instances of complete graphs (see Table 6) and random graphs with 3 different probabilities for edge occurrence (see Tables 7, 8 and 9). Each instance was allowed to run up to four minutes on every algorithm. In the tables we denote with "-" those instances that did not produce a result in the time limit.

Our results confirmed Cimikowski's conclusion, that the neural network heuristic topped the other heuristics presented in [39] in terms of solution quality. The second best was the 1-page heuristic which gave nearly optimal solutions in very short time.

Our simulated annealing heuristic, which was not a part of the original study, performed even better than the neural network heuristic. However, this was achieved partially at the cost of an increase in the time complexity. Nevertheless, our simulated annealing heuristic is still a feasible option, compared to the neural network heuristic, should a closer-to-optimal solution be required on average.

Table 6: Number of crossings for complete graphs.

	Greedy	Gr–ran		Mplan	E–len	1–page	Dyn.	Bis.	Neural		Annealing	
		min	avg						min	avg	min	avg
K_5	1	1	1.0	1	1	1	1	1	1	1.0	1	1.0
K_6	4	3	3.7	4	3	4	3	3	3	3.6	3	3.1
K_7	11	9	9.2	11	9	9	9	9	9	9.2	9	9.4
K_8	24	19	20.9	24	20	20	18	18	18	19.2	18	20.1
K_9	46	36	39.4	46	36	42	40	42	36	37.6	36	39.6
K_{10}	80	63	68.2	80	62	60	64	64	60	63.3	60	63.3
K_{11}	130	102	105.0	130	100	100	112	120	100	101.4	100	107.2
K_{12}	200	161	169.9	200	156	175	169	188	150	153.0	150	153.8
K_{13}	295	225	253.6	295	231	225	255	277	225	228.0	225	231.2
K_{14}	420	336	351.1	420	328	334	370	393	315	321.2	315	318.0
K_{15}	581	451	477.2	581	449	441	501	543	441	447.0	441	441.6
K_{16}	784	622	653.5	784	604	588	676	733	588	603.9	588	590.4
K_{17}	1036	804	847.8	1036	796	784	892	986	784	790.0	784	785.6
K_{18}	1344	1023	1079.3	1344	1045	1116	1149	1282	1008	1016.3	1008	1010.1
K_{19}	1716	1318	1384.6	1716	1318	1296	1462	1640	1296	1303.6	1296	1297.0
K_{20}	2160	1678	1759.3	2160	1675	1620	1847	2026	1620	1627.3	1620	1620.0

Table 7: Number of crossings for random graphs with edge probability of 0.5.

	Greedy	Gr–ran		Mplan	E–len	1–page	Dyn.	Bis.	Neural		Annealing	
		min	avg						min	avg	min	avg
rg_{10}	10	7	10.1	11	17	7	7	14	7	9.6	7	8.8
rg_{15}	118	95	106.7	105	109	93	99	112	91	99.7	91	94.2
rg_{20}	561	456	481.5	551	473	436	480	555	436	449.1	436	441.5
rg_{25}	1189	911	980.1	1177	898	876	1012	1175	867	884.4	867	879.6
rg_{30}	3115	2515	2572.4	3163	2506	2420	2644	2998	2407	2427.5	2405	2430.1
rg_{35}	5052	4059	4151.3	5055	4306	3921	4298	5046	3910	3968.6	3904	3920.4
rg_{40}	10738	8465	8702.8	10685	8539	8229	9189	10829	8258	8372.7	8228	8270.0
rg_{45}	16871	12892	13205.5	16347	13311	12662	14405	16557	12682	12812.2	12662	12751.6
rg_{50}	29291	22327	22969.7	29482	23035	21870	25997	29242	21888	22222.7	21873	22162.6

Table 8: Number of crossings for random graphs with edge probability of 0.3.

	Greedy	Gr–ran		Mplan	E–len	1–page	Dyn.	Bis.	Neural		Annealing	
		min	avg						min	avg	min	avg
rg_{10}	2	2	2.6	2	3	2	2	3	2	2.4	2	2.1
rg_{15}	21	15	19.5	21	19	13	16	18	13	16.8	13	13.7
rg_{20}	175	139	162.0	177	168	137	147	178	132	142.0	131	138.7
rg_{25}	565	437	481.1	548	455	429	461	557	423	435.1	423	436.3
rg_{30}	1056	841	887.6	1022	902	818	926	1024	810	822.2	808	832.6
rg_{35}	2004	1610	1666.5	2006	1761	1533	1763	2016	1531	1601.0	1532	1588.7
rg_{40}	4338	3516	3630.5	4274	3783	3298	3693	4465	3297	3402.2	3298	3335.9
rg_{45}	6192	4719	4995.3	5945	5012	4644	5382	6124	4616	4723.3	4602	4649.8
rg_{50}	9887	7696	7961.4	9748	8210	7508	8595	9992	7486	7576.7	7487	7590.8

Table 9: Number of crossings for random graphs with edge probability of 0.8.

	Greedy	Gr–ran		Mplan	E–len	1–page	Dyn.	Bis.	Neural		Annealing	
		min	avg						min	avg	min	avg
rg_{10}	49	38	42.2	47	37	34	41	44	34	37.8	34	37.8
rg_{15}	294	225	244.2	291	249	207	255	275	207	219.9	207	214.8
rg_{20}	1300	1050	1117.3	1323	1014	978	1152	1309	978	1011.1	979	1017.6
rg_{25}	3645	2813	3013.0	3653	2823	2723	3243	3638	2723	2775.1	2723	2756.2
rg_{30}	8238	6363	6621.7	8141	6859	6341	7200	7954	6282	6327.3	6282	6311.1
rg_{35}	14886	11601	11976.0	15034	11452	11418	13007	14501	11314	11370.3	11316	11352.4
rg_{40}	27172	20477	21436.6	26834	22082	20643	23767	26289	20246	20750.4	20205	20460.3
rg_{45}	45631	34869	35609.6	45693	37595	34194	40372	45328	-	-	-	-
rg_{50}	-	-	-	-	-	-	-	68093	-	-	-	-

11 Conclusion

The aim of this thesis was to survey and analyse the major algorithmic solution approaches for the NP-complete Crossing Number Problem and some of its variants. We collected and studied as much literature as possible in order to properly cover the subject.

The main areas presented were the general crossing number, the rectilinear crossing number, and the (fixed) linear crossing number. The book crossing number was mentioned only marginally, since there was not enough information available to us to study it in further detail. The problem of the crossing number on hierarchical embeddings was not included in this work, as it has already been thoroughly studied in the literature.

Most sections concerning algorithmic methods contain a paragraph analysing the performance of the particular approaches. We demonstrated computational results of several experimental studies and comparisons.

Moreover, we proposed three new heuristics for computing the (fixed) linear crossing number based on the simulated annealing scheme. We compared their performance with the best known approaches from the literature. One heuristic was unsuccessful, the other two managed to produce slightly better results for the respective variant of crossing number, albeit at the cost of a higher time complexity.

We also attempted to design a heuristic based on simulated annealing for computing the rectilinear crossing number that would shift vertices along their incident edges. Unfortunately, the computational results obtained were not satisfying, hence, we did not include this method into our survey. Supposedly, this movement of vertices interferred intrusively with the cooling process.

Since the problem covered in this thesis is so large-scale, there is a number of areas were the study could continue, *e.g.*, from keeping the survey up-to-date with new results, through proposing new approaches following a more thorough acquaintance with the problem, up to conducting further empirical evaluations.

Bibliography

[1] C. 762. Lecture notes of a graduate course, Fall 1999, Winter 2002, Winter 2004.

[2] M. Aigner and G. M. Ziegler. *Proofs from THE BOOK*. Springer-Verlag, Berlin, 1998.

[3] M. Ajtai, V. Chvátal, M. M. Newborn, and E. Szemerédi. Crossing-free subgraphs, 1982.

[4] D. Alberts, C. Gutwenger, P. Mutzel, and S. Näher. AGD-library: A library of algorithms for graph drawing. Technical Report MPI-I-97-1-019, Im Stadtwald, D-66123 Saarbrücken, Germany, 1997.

[5] C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity relationship diagrams. *J. Syst. Softw.*, 4(2-3):163–173, 1984.

[6] D. Battista, Garg, Liotta, Tamassia, Tassinari, and Vargiu. An experimental comparison of four graph drawing algorithms. *CGTA: Computational Geometry: Theory and Applications*, 7, 1997.

[7] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall; 1ST edition, 1998.

[8] G. D. Battista and R. Tamassia. On-line graph algorithms with spqr-trees. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 598–611, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[9] G. D. Battista and R. Tamassia. On-line maintenance of triconnected components with spqr-trees. *Journal Algorithmica*, 15(4), 1996.

[10] G. D. Battista and R. Tamassia. On-line planarity testing. *SIAM J. Comput.*, 25(5):956–997, 1996.

[11] F. Berhart and P. C. Kainen. The book thickness of a graph, 1979.

[12] P. Bertolazzi, G. D. Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. In *Workshop on Algorithms and Data Structures*, pages 331–344, 1997.

[13] S. N. Bhatt and F. T. Leighton. A FRAMEWORK FOR SOLVING VLSI GRAPH LAYOUT PROBLEMS. Technical Report MIT/LCS/TR-305, 1983.

[14] D. Bienstock. Some provably hard crossing number problems. *Discrete Comput. Geom.*, 6(5):443–459, 1991.

[15] D. Bienstock and N. Dean. Bounds for rectilinear crossing numbers. *J. Graph Theory*, 17(3):333–348, 1993.

[16] D. Bienstock and C. L. Monma. Optimal enclosing regions in planar graphs, 1989.

[17] D. Bienstock and C. L. Monma. On the complexity of embedding planar graphs to minimize certain distance measures, March, 1990.

[18] T. Bilski. Embedding graphs in books: survey. 139(2):134–138, 2005.

[19] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *JCSS*, 13:335–379, 1976.

[20] C. Buchheim, M. Chimani, D. Ebner, C. Gutwenger, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. A branch-and-cut approach to the crossing number problem. Technical report, Zentrum für Angewandte Informatik Köln, Lehrstuhl Jünger, January 2006.

[21] C. Buchheim, D. Ebner, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. Exact crossing minimization. In P. Healy and N. S.Nikolov, editors, *Graph Drawing*, pages 37–48. Springer, Limerick, Ireland, September 2005.

[22] C. Buchheim and L. Zheng. Fixed linear crossing minimization by reduction to the maximum cut problem. In *COCOON 2006*, April 2006.

[23] J. F. Buss and P. W. Shor. On the pagenumber of planar graphs. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 98–100, New York, NY, USA, 1984. ACM Press.

[24] J. Cai. Counting embeddings of planar graphs using DFS trees. *SIAM Journal on Discrete Mathematics*, 6(3):335–352, 1993.

[25] J. Cai, X. Han, and R. E. Tarjan. An o(m log n)-time algorithm for the maximal planar subgraph problem. *SIAM J. Comput.*, 22(6-7):1142–1162, 1993.

[26] G. Calinescu and C. G. Fernandes. Finding large planar subgraphs and large subgraphs of a given genus. pages 152–161, 1996.

[27] G. Calinescu, C. G. Fernandes, U. Finkler, and H. Karloff. A better approximation algorithm for finding planar subgraphs. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1996.

[28] G. Calinescu, C. G. Fernandes, U. Finkler, and H. Karloff. A better approximation algorithm for finding planar subgraphs. volume 27, pages 269–302, 1998.

[29] R. J. S. Carmo. O problema do subgrafo planar ótimo, 1994.

[30] M.-J. Carpano. Automatic display of hierarchized graphs for computer-aided decision analysis, 1980.

[31] K. C. Chang and D. H.-C. Du. Efficient algorithms for layer assignment problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(1):67 – 78, January 1987.

[32] G. Chartrand and L. Lesniak. *Graphs & digraphs (2nd ed.)*. Wadsworth Publ. Co., Belmont, CA, USA, 1986.

[33] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *J. Comput. Syst. Sci.*, 30(1):54–76, 1985.

[34] T. Chiba, I. Nishioka, and I. Shirakawa. An algorithm for maximal planarization of graphs. In *Proceedings on the 1979 IEEE International Symposium on Circuits and Systems*, pages 336–441, 1979.

[35] F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Embedding graphs in books: a layout problem with applications to vlsi design. *SIAM J. Algebraic Discrete Methods*, 8(1):33–58, 1987.

[36] Cimikowski. An analysis of some heuristics for the maximum planar subgraph problem. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1995.

[37] R. Cimikowski. Branch-and-bound techniques for the maximum planar subgraph problem, 1994.

[38] R. Cimikowski. An analysis of heuristics for graph planarization, 1997.

[39] R. Cimikowski. Algorithms for the fixed linear crossing number problem, 2002.

[40] R. Cimikowski and P. Shope. A neural-network algorithm for a graph layout problem. *IEEE Transactions on Neural Networks*, 7(2):341–345, March 1996.

[41] R. J. Cimikowski. Graph planarization and skewness. In *Congressus Numerantium*, volume 88, pages 21–32, 1992.

[42] R. J. Cimikowski. On heuristics for determining the thickness of a graph. *Information Sciences*, 85(1–3):87–98, 1995.

[43] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 193–242. Elsevier, Amsterdam, 1990.

[44] H. Crowder, E. L. Johnson, and M. Padberg. Solving large-scale zero-one linear programming problems, 1983.

[45] R. J. da Silva Carmo and Y. Wakabayashi. The maximum planar subgraph problem (extended abstract), June 1995.

[46] E. Damiani, O. D'Antona, and P. Salemi. An upper bound to the crossing number of the complete graph drawn on the pages of a book. *J. Comb. Inf. Syst. Sci.*, 19(1-2):75–84, 1994.

[47] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[48] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[49] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid, 1990.

[50] N. Dean. Mathematical programming formulation of rectilinear crossing minimization. Technical Report DIMACS TR 2002-12, 2002.

[51] C. Demetrescu and I. Finocchi. Break the "right" cycles and get the "best" drawing. In *Proc. 2nd Work. Algorithm Engineering and Experiments, ALENEX*, 7–8 2000.

[52] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 436–441, 1989.

[53] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 436–441, 1989.

[54] H. N. Djidjev. A linear algorithm for finding a maximal planar subgraph, 1995.

[55] H. N. Djidjev and I. Vrťo. Crossing numbers and cutwidths, 2003.

[56] R. G. Downey, M. R. Fellows, R. Niedermeier, and P. Rossmanith. *Parameterized Complexity*. Springer, 1999.

[57] V. Dujmović, M. R. Fellows, M. T. Hallett, M. Kitching, G. Liotta, C. McCartin, N. Nishimura, P. Ragde, F. A. Rosamond, M. Suderman, S. Whitesides, and D. R. Wood. A fixed-parameter approach to two-layer planarization. In *Proc. Graph Drawing (GD'01)*, pages 1–15, 2001.

[58] V. Dujmović, M. R. Fellows, M. T. Hallett, M. Kitching, G. Liotta, C. McCartin, N. Nishimura, P. Ragde, F. A. Rosamond, M. Suderman, S. Whitesides, and D. R. Wood. On the parameterized complexity of layered graph drawing. In *European Symposium on Algorithms*, pages 488–499, 2001.

[59] W. Dyck. Beiträge zur analysis situs i. aufsatz. ein- und zweidimensionale mannigfaltigkeiten, December, 1888.

[60] M. Dyer, L. Foulds, and A. Frieze. Analysis of heuristics for finding a maximum weight planar subgraph. *European Journal of Operations Research*, 20:102–114, 1985.

[61] P. Eades. A heuristic for graph drawing. In *Congressus Numerantium*, volume 42, pages 149–160, 1984.

[62] P. Eades, L. R. Foulds, and J. W. Giffin. An efficient heuristic for identifying a maximum weight planar graph, 1982.

[63] P. Eades and S. Whitesides. Drawing graphs in two layers. *Theor. Comput. Sci.*, 131(2):361–374, 1994.

[64] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs, 1994.

[65] D. Ebner. Optimal crossing minimization using integer linear programming, February 2005.

[66] R. B. Eggleton. Rectilinear drawings of graphs. *Utilitas Math.*, 29:149–172, 1986.

[67] P. Erdos and R. K. Guy. Crossing number problems.

[68] G. Even, S. Guha, and B. Schieber. Improved approximations of crossings in graph drawings. pages 296–305, 2000.

[69] L. Faria, C. M. H. de Figueiredo, and C. F. X. de Mendonça Neto. Splitting number is NP-complete. In *Workshop on Graph-Theoretic Concepts in Computer Science*, pages 285–297, 1998.

[70] L. Faria, C. M. H. de Figueiredo, and C. F. X. de Mendonça Neto. The splitting number of the 4-cube. In *Latin American Theoretical INformatics*, pages 141–150, 1998.

[71] I. Fáry. On straight line representations of graphs. *Acta Univ. Szeged Sect. Sci. Math.*, 11:229–233, 1948.

[72] T. Feo and M. Resende. Greedy randomized adaptive search procedures, 1995.

[73] L. Foulds and D. Robinson. Construction properties of combinatorial deltahedra, 1979.

[74] L. R. Foulds, P. B. Gibbons, and J. W. Giffin. Facilities layout adjacency determination: An experimental comparison of three graph theoretic heuristics, 1985.

[75] L. R. Foulds and D. F. Robinson. A strategy for solving the plant layout problem, 1976.

[76] L. R. Foulds and D. F. Robinson. Graph theoretic heuristics for the plant layout problem, 1978.

[77] D. Franken, J. Ochs, and K. Ochs. Generation of wave digital structures for connection networks containing ideal transformers. volume 3, pages 240 – 243, May 2003.

[78] M. Garey, D. Johnson, G. Miller, and C. Papadimitriou. The complexity of coloring circular arcs and chords, 1980.

[79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

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

[81] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63, New York, NY, USA, 1974. ACM Press.

[82] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM Journal on Computing*, 31(2):601–625, 2002.

[83] F. Gavril. Algorithms for a maximum clique and a maximum independent set of a circle graph, 1973.

[84] J. W. Giffin and L. R. Foulds. Efficient graph planarity updating tests. *Ars-Combinatoria*, 17:185–202, 1984.

[85] O. Goldschmidt and A. Takvorian. An efficient graph planarization two-phase heuristic. Technical Report ORP91-01, 1992.

[86] O. Goldschmidt and A. Takvorian. An efficient graph planarization two-phase heuristic, 1994.

[87] J. E. Goodman, R. Pollack, and B. Sturmfels. Coordinate representation of order types requires exponential storage. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 405–410, New York, NY, USA, 1989. ACM Press.

[88] M. Grohe. Computing crossing numbers in quadratic time. *J. Comput. Syst. Sci.*, 68(2):285–302, 2004.

[89] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32(6):1195–1220, 1984.

[90] C. Gutwenger and P. Mutzel. A linear time implementation of spqr-trees. In *GD '00: Proceedings of the 8th International Symposium on Graph Drawing*, pages 77–90, London, UK, 2001. Springer-Verlag.

[91] C. Gutwenger and P. Mutzel. An experimental study of crossing minimization heuristics. In G. Liotta, editor, *Graph Drawing 2003, Perugia, September 2003*. Springer, 2003.

[92] C. Gutwenger and P. Mutzel. Graph embedding with minimum depth and maximum external face (extended abstract). In G. Liotta, editor, *Graph Drawing, Perugia, 2003*, pages pp. 259–272. Springer, 2004.

[93] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 246–255, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[94] R. Guy. Latest results on crossing numbers. 1971.

[95] R. Guy. Crossing numbers of graphs. pages 111–124, May 1972.

[96] D. Harel and M. Sardas. Randomized graph drawing with heavy-duty preprocessing. *Journal of Visual Languages and Computing*, 6(3):233–253, 1995.

[97] F. C. Harris, Jr., and C. R. Harris. A proposed algorithm for calculating the minimum crossing number of a graph, 1995.

[98] L. S. Heath. Embedding planar graphs in seven pages. In *Proc. 25th Annual Symp. on Foundations of Comput. Sci.*, pages 74–83, 1984.

[99] L. Heffter. Über das problem der nachbargebiete, 1891.

[100] J. Hines, F. C. Harris, and K. B. Winiecki. Solving quadratic assignment problems with parallel genetic algorithms, May 1992.

[101] P. Hliněný. Crossing-critical graphs and path-width. In *Graph Drawing*, pages 102–114, 2001.

[102] P. Hliněný. Crossing number is hard for cubic graphs. *J. Comb. Theory Ser. B*, 96(4):455–471, 2006.

[103] P. Hliněný and G. Salazar. On difficulty of graph crossing number (extended abstract), 2003.

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

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

[106] J. Hopfield and D. Tank. Neural computiaton of decisions in optimization problems. 52:141–152, 1985.

[107] W.-L. Hsu. A linear time algorithm for finding maximal planar subgraphs. In *ISAAC: 6th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms), Organized by Special Interest Group on Algorithms (SIGAL) of the Information Processing Society of Japan (IPSJ) and the Technical Group on Theoretical Foundation of Computing of the Institute of Electronics, Information and Communication Engineers (IEICE))*, 1995.

[108] H. F. Jensen. An upper bound for the rectilinear crossing number of the complete graph. *J. Combinatorial Theory Ser. B*, 10:212–216, 1971.

[109] D. S. Johnson. The np-completeness column: An ongoing guide. *J. Algorithms*, 7(4):584–601, 1986.

[110] M. Jünger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using pq-trees. Technical Report TR GDEA-19, 1998.

[111] M. Jünger and P. Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. Technical Report TR GDEA-25, 1993.

[112] M. Jünger and P. Mutzel. Solving the maximum weight planar subgraph problem by branch and cut, 1993.

[113] M. Jünger and P. Mutzel. 2–layer straightline crossing minimization: Performance of exact and heuristic algorithms, 1997.

[114] M. Jünger and S. Thienel. Introduction to abacus - a branch-and-cut system, March 1998.

[115] P. C. Kainen. Book thickness of graphs, ii, 1990.

[116] G. Kant. An o(n2) maximal planarization algorithm based on pq-trees. Technical Report RUU-CS-92-03, January 1992.

[117] R. M. Karp, R. E. Miller, and J. W. Thatcher. Reducibility among combinatorial problems, December 1975.

[118] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[119] G. Klau and P. Mutzel. Quasi orthogonal drawing of planar graphs. Technical Report TR MPI-I-98-1-013, 1998.

[120] B. Korte and D. Hausmann. An analysis of the greedy heuristic for independence systems, 1978.

[121] S. T. Kozo Sugiyama and M. Toda. Methods for visual understanding of hierarchical system structures, 1981.

[122] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[123] K. Kuratowski. Sur le probleme des courbes gauches en topologie, 1930.

[124] M. Laurent. The max-cut problem. In *Chapter in Annotated Bibliographies in Combinatorial Optimization*, pages pages 241–259. edited by M. Dell'Amico, F. Maffioli and S. Martello. John Wiley, 1997.

[125] M. Laurent and F. Rendl. Semidefinite programming and integer programming, 2005.

[126] F. T. Leighton. New lower bound techniques for VLSI. *NASA STI/Recon Technical Report N*, 83:19003–+, Aug. 1982.

[127] F. T. Leighton. *Complexity issues in VLSI: optimal layouts for the shuffle-exchange graph and other networks.* MIT Press, Cambridge, MA, USA, 1983.

[128] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.

[129] S. Leipert. Berechnung maximal planarer untergraphen mit hilfe von pq-bäumen, diplom thesis, 1995.

[130] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs, 1966.

[131] T. Lengauer. *Combinatorial algorithms for integrated circuit layout.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[132] J. Leung. A new graph-theoretic heuristic for facility layout. *Manage. Sci.*, 38(4):594–605, 1992.

[133] A. Liebers. Planarizing graphs - a survey and annotated bibliography, 2001.

[134] F. Liers, M. Jünger, G. Reinelt, and G. Rinaldi. Computing exact ground states of hard ising spin glass problems by branch-and-cut, 2003.

[135] P. Liu and R. Geldmacher. On the deletion of nonplanar edges of a graph, 1977.

[136] P. Liu and R. Geldmacher. On the deletion of nonplanar edges of a graph, 1979.

[137] C. Lovegrove. Crossing numbers of permutation graphs, master thesis, May 1988.

[138] S. MacLane. A combinatorial condition for planar graphs, 1936.

[139] A. Mansfield. Determining the thickness of a graph is np-hard, 1983.

[140] S. Masuda, T. Kashiwabara, K. Nakajima, and T. Fujisawa. On the np-completeness of a computer network layout problem, 1987.

[141] S. Masuda, K. Nakajima, T. Kashiwabara, and T. Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Trans. Comput.*, 39(1):124–127, 1990.

[142] C. C. R. Mauricio G. C. Resende. Greedy randomized adaptive search procedures, 2002.

[143] K. Mehlhorn and P. Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[144] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21(6):1087–1092, 1953.

[145] P. Mutzel. A fast 0(n) embedding algorithm, based on the hopcroft-tarjan planary test. Technical report, 1992.

[146] P. Mutzel. The maximum planar subgraph problem, phd thesis, 1994.

[147] P. Mutzel. Optimization in leveled graphs, 1999.

[148] P. Mutzel and R. Weiskircher. Optimizing over all combinatorial embeddings of a planar graph. *Lecture Notes in Computer Science*, 1610:361+, 1999.

[149] P. Mutzel and T. Ziegler. The constrained crossing minimization problem. In *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*, pages 175–185, London, UK, 1999. Springer-Verlag.

[150] P. Mutzel and T. Ziegler. The constrained crossing minimization problem - a first approach, 1999.

[151] M. Newton, O. Sýkora, and I. Vrťo. Two new heuristics for two-sided bipartite graph drawing. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, pages 312–319, London, UK, 2002. Springer-Verlag.

[152] T. Nicholson. Permutation procedure for minimising the number of crossings in a network. *Proc. IEE*, 115(1):21–26, 1968.

[153] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*, volume 140. North-Holland Mathematics Studies.

[154] K. J. Nurmela and P. R. J. Östergard. Constructing covering designs by simulated annealing. Technical Report B10, 1993.

[155] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.

[156] T. Ozawa and H. Takahashi. A graph-planarization algorithm and its application to random graphs. In *Proceedings of the 17th Symposium of Research Institute of Electric Communication on Graph Theory and Algorithms*, pages 95–107, London, UK, 1981. Springer-Verlag.

[157] J. Pach. Crossing numbers, 2000.

[158] J. Pach, F. Shahrokhi, and M. Szegedy. Applications of the crossing numbers. *Algorithmica*, 16:11–117, 1996.

[159] J. Pach, J. Spencer, and G. Tóth. New bounds on crossing numbers. Technical Report 99-37, 21, 1999.

[160] J. Pach and G. Tóth. Which crossing number is it anyway? Technical Report 29, 11, 1998.

[161] J. Pach and G. Tóth. Thirteen problems on crossing numbers, 2000.

[162] J. Pach and G. Tóth. Graphs drawn with few crossings per edge. In S. North, editor, *Graph Drawing, Berkeley, California, USA*, pages pp. 345–354. Springer, September 1996.

[163] S. Pan and R. B. Richter. The crossing number of k(11) is 100, January 2007.

[164] J. A. L. Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 706–715, New York, NY, USA, 1994. ACM Press.

[165] W. R. Pulleyblank. Polyhedral combinatorics. pages 371–446, 1989.

[166] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 248–261, London, UK, 1997. Springer-Verlag.

[167] M. Resende and C. Ribeiro. A GRASP for graph planarization. *NETWORKS: Networks: An International Journal*, 29, 1997.

[168] R. B. Richter and C. Thomassen. Minimal graphs with crossing number at least k. *J. Comb. Theory Ser. B*, 58(2):217–224, 1993.

[169] R. B. Richter and C. Thomassen. Relations between crossing numbers of complete and complete bipartite graphs, February 1997.

[170] M. Ringenburg. Using simulated annealing to find crossing numbers, 2000.

[171] J. S. R.K. Guy, T. Jenkyns. The toroidal crossing number of the complete graph. *J. Combin. Theory*, 4:376–390, 1968.

[172] N. Robertson and P. Seymour. Excluding a graph with one crossing. pages 669–675, 1971.

[173] N. Robertson and P. D. Seymour. Graph minors. v. excluding a planar graph. *J. Comb. Theory Ser. B*, 41(1):92–114, 1986.

[174] N. Robertson and P. D. Seymour. Graph minors. xiii: the disjoint paths problem. *J. Comb. Theory Ser. B*, 63(1):65–110, 1995.

[175] J. Roskind and R. E. Tarjan. A note on finding minimum-cost edge-disjoint. spanning trees, 1985.

[176] M. Sarrafzadeh and D. T. Lee. A new approach to topological via minimization problem, August 1989.

[177] W. Schnyder. Embedding planar graphs on the grid. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 138–148, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[178] F. Shahrokhi, O. Sýkora, L. Székely, and I. Vrťo. Crossing numbers: bounds and applications, 1997.

[179] F. Shahrokhi, O. Sýkora, L. A. Székely, and I. Vrťo. A gap between crossing numbers and convex crossing numbers, September 2003.

[180] F. Shahrokhi, L. A. Székely, O. Sýkora, and I. Vrťo. The book crossing number of a graph. *J. Graph Theory*, 21(4):413–424, 1996.

[181] W.-K. Shih and W.-L. Hsu. A simple test for planar graphs, 1993.

[182] J. Spencer and G. Tóth. Crossing numbers of random graphs. *Random Struct. Algorithms*, 21(3-4):347–358, 2002.

[183] M. Stallmann, F. Brglez, and D. Ghosh. Evaluating iterative improvement heuristics for bigraph crossing minimization, 1999.

[184] M. Stallmann, F. Brglez, and D. Ghosh. Heuristics, experimental subjects, and treatment evaluation in bigraph crossing minimization, 2000.

[185] O. Sýkora and I. Vrťo. On vlsi layout of the star graph and related networks, 1994.

[186] L. A. Székely. A successful concept for measuring non-planarity of graphs: the crossing number, 2004.

[187] U. Tadjiev, F. C. Harris, and Jr. Parallel computation of the minimum crossing number of a graph, 1997.

[188] Y. Takefuji and K. Lee. A near-optimum parallel planarization algorithm, September 1989.

[189] Y. Takefuji, K. Lee, and Y. B. Cho. Comments on o(n2) algorithms for graph planarization, 1991.

[190] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987.

[191] R. Tamassia. A dynamic data structure for planar graph embedding (extended abstract). In *ICALP '88: Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pages 576–590, London, UK, 1988. Springer-Verlag.

[192] C. Thomassen. The graph genus problem is np-complete. *J. Algorithms*, 10(4):568–576, 1989.

[193] C. Thomassen. Embeddings and minors. pages 301–349, 1995.

[194] C. Thomassen. A simpler proof of the excluded minor theorem for higher surfaces. *J. Comb. Theory Ser. B*, 70(2):306–311, 1997.

[195] J. T. Thorpe and F. C. Harris. A parallel stochastic optimization algorithm for finding mappings of the rectilinear minimal crossing problem. *Ars Combin.*, 43:135–148, 1996.

[196] K. Thulasiraman, R. Jayakumar, and M. Swamy. On maximal planarization of nonplanar graphs, August 1986.

[197] K. Thulasiraman, R. Jayakumar, and M. Swamy. O(n2) algorithms for graph planarization, March 1989.

[198] M. Tomaštik. Metóda neurónových sietí v riešení problému lineárneho priesečníkového čísla, práca študentskej vedeckej konferencie, 2006.

[199] N. Tomii, Y. Kambayashi, and Y. Shuzo. On planarization algorithms of 2-level graphs, 1977.

[200] P. Turán. A note of welcome, 1977.

[201] W. T. Tutte. How to draw a graph, 1963.

[202] W. T. Tutte. Toward a theory of crossing numbers. *J. Combinatorial Theory*, 8:45–53, 1970.

[203] T. J. van Roy and L. A. Wolsey. Solving mixed integer programming problems using automatic reformulation, 1987.

[204] G. Vollen. Pq-trees and maximal planarization - an approach to skewness, diplom thesis, February 1998.

[205] K. Wagner. Bemerkungen zum vierfarbenproblem, 1936.

[206] T. Watanabe, T. Ae, and A. Nakamura. Np-hardness of edge-deletion and contraction problems, 1983.

[207] M. Waterman and J. Griggs. Interval graphs and maps of dna, 1986.

[208] J. Westbrook. Fast incremental planarity testing. In *Proc. 19th Int. Colloquium on Automata, Languages and Programming*, pages 342–353. Lecture Notes in Computer Science, Springer-Verlag 623, Berlin, 1992.

[209] A. T. White and L. W. Beineke. Topological graph theory. pages 15–49, 1983.

[210] A. Widgerson. The complexity ot the hamiltonian circuit problem for planar graphs.

[211] W. P. W.S. McCulloch. A logical calculus of ideas imminent in nervous activity. 5:115–133, 1943.

[212] M. Yannakakis. Node-and edge-deletion np-complete problems. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 253–264, New York, NY, USA, 1978. ACM Press.

[213] M. Yannakakis. Four pages are necessary and sufficient for planar graphs (extended abstract). In *STOC*, pages 104–108, 1986.

[214] T. Ziegler. Crossing minimization in automatic graph drawing, phd thesis, 2000.

A (Integer) Linear Programming

A *Linear Program* (*LP*) is an optimization problem consisting of an objective function and several constraints. George B. Dantzig proposed the following standard model:

$$\begin{gathered} \text{maximize } c^T x \\ \text{subject to } Ax \leq b \\ x \geq 0 \end{gathered}$$

where $c \in R^n$, $A \in R^{m \times n}$, $b \in R^m$. The linear function $c^T x : R^n \rightarrow R$ is called the *objective function* and the inequalities in the system $Ax \leq b$ are called *constraints.* We can transfrom minimization problems to maximization problems by multiplying the objective function cTx by -1.

Problem 12 (Linear Programming). *Given a matrix $A \in R^{m \times n}$, and vectors $b \in R^m$, $c \in R^n$, find a vector $\hat{x} \in R^n$ with*

$$c^T \hat{x} = max\{c^T x | Ax \leq b\}$$

Linear Programs can be solved in polynomial time. The most widespread algorithms are the simplex- , the ellipsoid- and the interior point method [65]. There are a number of well developed and optimized implementations that efficiently solve linear programs, even on a very large scale.

Solving large Linear programs

In large linear programs, especially for NP-hard optimization problems, the number of variables and/or constraints often becomes too large to be handled by the LP-solver [65]. In both cases there are techniques that start with a subset of variables (inequalities) and add new ones only if required during the runtime of the algorithm.

Column generation

Column generation is a widely used technique to work around a huge number of variables. Instead of solving the original linear program, we solve a reduced LP $L(J) = max\{\sum_{j \in J} c_j x_j | \sum_{j \in J} A_{i,j} x_j \leq b\}$ for some $J \subseteq \{l, ..., n\}$. Using the dual variables y we can search for a $j \notin J$ such that $c_j - \hat{y} A_j > 0$. If there is such a j we can add it to J and solve the new LP, otherwise we have an optimal solution over all columns.

Branch-and-Cut

The *cutting plane approach* starts with a small subset of constraints and computes an optimum solution. If there are no more constraints that are violated by the current solution, we have an optimum solution for the original linear program. Otherwise we add the violated constraint and resolve the *LP.*

It is not required to have a complete "list" of all constraints, but we need a method to identify constraints that are violated by the current solution but are valid for the original LP. This is called the *general separation problem* and defined as follows:

Problem 13 (General Separation). *Given a class of valid inequalities and a vector $y \in R^n$, either prove that y satisfies all inequalities in the class, or find an inequality which is violated by y.*

When we want to apply the cutting plane approach to zero-one *ILPs*, we can transform the ILP $I = max\{c^T x | Ax \leq b, x \in \{0,1\}^n\}$ to a linear program by dropping the integrality constraints and adding for each variable x, an inequality $x_i \leq 1$. The resulting linear program $L = max\{c^T x | Ax \leq b, 0 \leq x_i \leq l, \forall i \in \{l, ..., n\}\}$ is called the *linear relaxation* of I.

Every optimum solution x of L that is *integral* ($x_i \in \{0,1\} \forall i \in \{0, ..., n\}$) is also an optimum solution for our original problem I. If $\hat{x}$ is fractional we can try to find further problem specific inequalities to "cut off" the fractional solutions. Another possibility is the use of *general purpose cutting planes,* for example, *Gomory Cuts* [65]. They are not specific to the particular linear program and can be used in conjunction with every combinatorial optimization problem.

For a large number of NP-hard optimization problems no efficient exact separation algorithms are known. However, we can use the cutting plane approach in combination with an enumeration procedure called *branch-and-bound* (see next subsection).

The *branch-and-bound* approach is a simple divide-and-conquer approach that tries to solve the original problem by splitting it in smaller subproblems. On each node of the resulting search tree upper and lower bounds are computed. An upper bound is called *local* if it is only valid for the current subproblem and *global* otherwise.

In the case of zero-one integer linear programs the root of the search tree corresponds to the original problem. At each node a local upper bound can be computed by solving the linear relaxation of the particular subproblem. If the solution is feasible and its objective value is higher than the best found feasible solution, it is stored and the global lower bound is increased accordingly. If the local upper bound is smaller than the current global lower bound, we can discard the subproblem. Otherwise (the local upper bound is higher than the best feasible solution known so far), we select a fractional *branching variable* and create two new subproblems by setting the branching variable to zero, respectively one.

The procedure is repeated until the list of unsolved subproblems becomes empty. In this case we can guarantee that the best feasible solution is an optimum solution for the root problem.

The *branch-and-cut* approach was first used by Grötschel, Jünger and Reinelt in [89] for the linear ordering problem. It is a combination of the branch-and-bound method and the cutting plane approach.

In addition to the pure branch-and-bound approach we try to find violated cuts which are added to the LP relaxation and the subproblem is solved again. The branching process continues when no more cuts can be separated. A more detailed description can be found in [65].

B Some tree structures

B.1 SPQR-trees

In this section, we give a brief introduction to the SPQR-tree data structure for biconnected planar graphs. SPQR-trees were introduced by Di Battista and Tamassia [10] and since then have been used in various graph drawing applications like, *e.g.*, minimizing the number of bends in an orthogonal drawing [12, 93] or finding planar embeddings with a minimum depth and a maximum external face [92]. Recently, SPQR-trees have also been applied in the area of circuit design [77].

An undirected graph G is *connected*, if any two vertices of G are connected by a path. The maximal connected subgraphs of G are the *connected components* of G. A vertex v is a *cutvertex* if the removal of v increases the number of components. G is *biconnected.* if G is connected and G has no cut vertices. The maximal biconnected subgraphs of G are called *bicomponents.* A pair u, w of vertices of G is a *separation pair*, if the deletion of v and w disconnects G. G is *triconnected*, if G has no cut vertex and no separation pair.

An essential property of triconnected graphs related to planarity is given in the next lemma. A *subdivision* of a graph H is a graph H' that can be obtained by H by replacing some of the edges of H by paths having at most their endpoints in common.

Lemma 1 (**Nishizeki, Chiba [153]**). *A planar graph G has a unique embedding in the plane if and only if G is a subdivision of a triconnected graph.*

The *triconnected components* (*or tricomponents*) of G are produced by a recursive procedure that, if G has a separation pair v, w, divides G into two subgraphs G_1 and G_2 defined by the separation pair. Each of v and w is included in both G_1 and G_2. For the precise definition and a linear time algorithm that finds the tricomponents of a graph see [105].

The decomposition tree

SPQR-trees represent the decomposition of a planar biconnected graph according to its split pairs. Let G be a planar 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) or a maximal subgraph C of G such that $\{u, v\}$ is not a split pair of C. Let $\{s, t\}$ be a split pair of G. A *maximal split pair* $\{u, v\}$ of G with respect to $[s, t]$ is such that, for any other split pair $\{u', v'\}$, vertices u, v, s, and t are in the same split component.

The *decomposition tree* T of G describes a recursive decomposition of G with respect to its split pairs and will be used to synthetically represent all the embeddings of G with vertices s and t on the external face.

Let $e = (s, t)$ be an edge of G, called the *reference edge.* The SPQR-tree T of G with respect to e is a rooted ordered tree whose nodes are of four types: S, P, Q, and R. Each node μ of T has an associated biconnected multi-graph, called the *skeleton* of μ and denoted by $skeleton(\mu)$. Also, it is associated with an edge of the skeleton of the parent ν of μ, called the *virtual edge* of μ in $skeleton(\nu)$.

Tree T is recursively defined as follows:

Trivial Case. If G consists of exactly two parallel edges between s and t, then T consists of a single Q-node whose skeleton is G itself.

Parallel Case. If the split pair $\{s,t\}$ has k split components $G_1,...,G_k$ with $k \geq 3$, the root of T is a P-node μ, whose skeleton consists of k parallel edges $e = e_1,...,e_k$ between s and t.

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

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

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

Let μ_r be the root of T in the decomposition given above. We add a Q-node representing the reference edge e and make it the parent of μ_r so that it becomes the new root. Figure 2 shows an example of a graph and its SPQR-tree.

Let e be an edge in $skeleton(\mu)$ and let v be the pertinent node of e. Deleting edge $\{\mu, v\}$ in T splits T into two connected components. Let T_v be the connected component containing v. The *expansion graph* of e (denoted with $expansion(e)$) is the graph induced by the edges of G contained in the skeletons of the Q-nodes in T_v. We further introduce the notation $expansion^+(e)$ for the graph $expansion(e) \cup e$. Figure 3 gives an example for the expansion graph of an edge.

The *pertinent graph* of a tree node μ is obtained from the skeleton of μ by replacing each skeleton edge except for the reference edge of μ with its expansion graph. Examples for the pertinent and skeleton graphs of the different node types are shown in Figure 4. If v is a vertex in G, a node in T whose skeleton contains v is called an *allocation node* of v.

Two basic primitives can be used to obtain a new planar embedding from T. A reverse operation consists of flipping a split component around its poles. A *swap* operation consists of exchanging the position of two split components of the same split pair. For example, Fig. 31 shows a planar embedding obtained by means of two swap operations and one flip operation.

SPQR-trees can be constructed in linear time and their size including the skeleton graphs is linear in the size of the original graph [10, 90]. Choosing a different reference edge e' is equivalent to rooting the tree T at the Q-node whose skeleton contains e'. In particular, the unrooted version of the SPQR-tree of a planar biconnected graph (including the skeleton graphs) is unique.

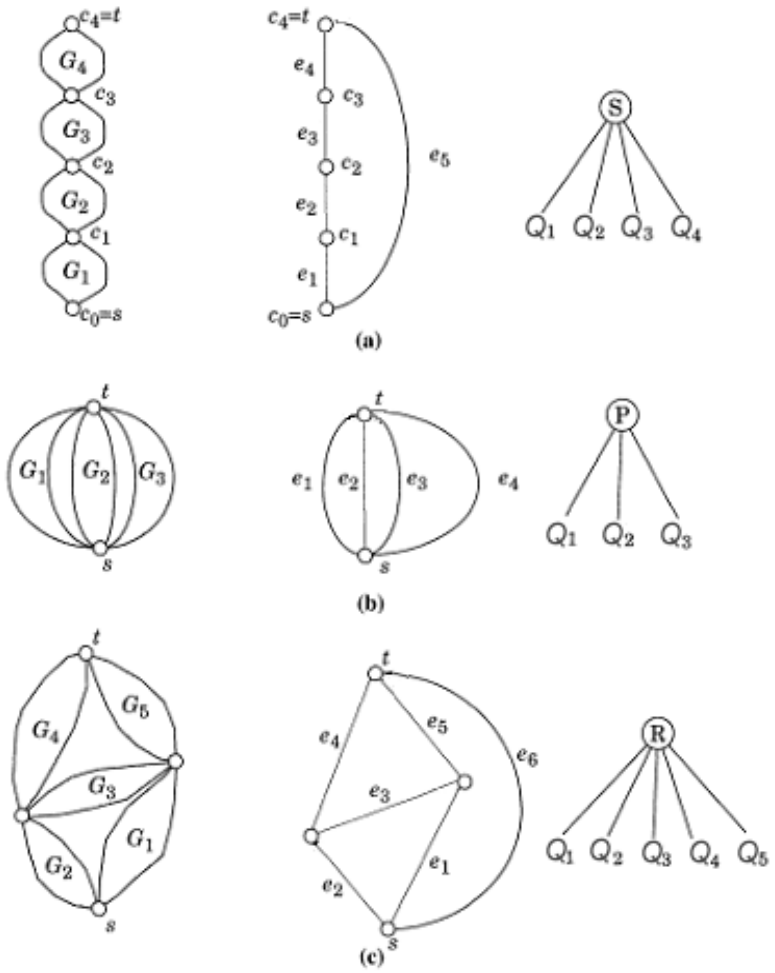

Figure 28: [10] (a) Series decomposition, (b) Parallel decomposition, (c) Rigid decomposition.

As described in [10], SPQR-trees can be used to represent all combinatorial embeddings of a biconnected planar graph. This is done by choosing combinatorial embedd'mgs for the skeletons of the nodes in the tree. The skeletons of S- and Q-nodes are simple cycles, so they have only one embedding. The skeletons of R-nodes are always triconnected graphs. According to the definition of combinatorial embeddings, a triconnected graph has two embeddings which are mirror-images of each other, *i.e.*, the order of the edges around each vertex is reversed in the mirror embedding. The number of embeddings of a P-node skeleton with k edges is $(k-1)!$.

Every embedding of the original graph defines a unique embedding for each skeleton of anode in the SPQR-tree. Conversely, when we define an embedding for each skeleton of a node in the SPQR-tree. we define a unique embedding for the original graph. Thus, if the SPQR-tree of G has r R-nodes and P-nodes P_1 to P_k where the skeleton of P_i has p_i edges, then the total number of combinatorial embeddings of G is

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

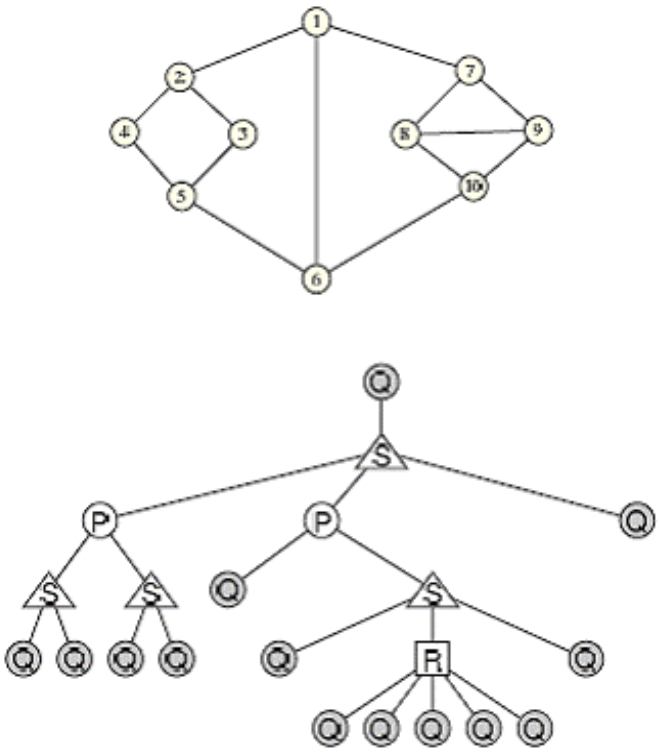

Figure 29: [93] A biconnected planar graph and its SPQR-tree.

In [12] SPQR-trees are used to enumerate all combinatorial embeddings of a biconnected planar graph within a branch-and-bound algorithm for finding a combinatorial embedding and an external face for a graph such that the drawing computed by Tamassia'S algorithm [190] has the minimum number of bends among all possible orthogonal drawings of the graph.

B.2 PQ-trees

The PQ-tree data structure, used for planarity testing of graphs, is the basic building block in this thesis. The PQ-tree data structure was designed by Booth and Lueker [19] for sorting out permissible permutations of a set, where some subsets have to be consecutive. This sorting algorithm is also useful in planarity testing, which were one of the main uses for it described in [19]. The other two were tests for *consecutive ones property* and *interval graphs.* Today, the PQ-tree data structure is used in biology, chemistry, graph theory, graph drawing, circuit layout, matrix manipulation, and other areas where certain types of legal permutations are of interest.

PQ-trees are rooted trees, which have two types of internal nodes, called P-and Q-nodes. A *Q-node* specifies that its children can be placed in forward or reverse order, while a *P-node* specifies that its children can be placed in any order. (Note that if a node has two children, it does not matter if it is a P- or a Q-node.) The permutations stored by the entire tree, then, are the set of all allowed permutations of the leaves.

An example of a PQ-tree is given in Figure 5.3. P-nodes are depicted using a circle,

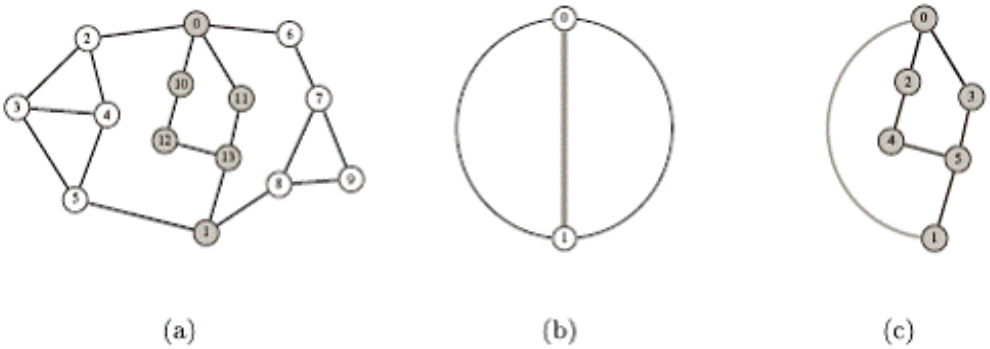

Figure 30: [93] Example for the expansion graph of a skeleton edge: (a) a biconnected planar graph G, (b) the skeleton μ of a P-node in the SPQR-tree of G , and (c) the graph $expansion^{+}(e)$ for the gray edge e in $skeleton(\mu)$.

whereas Q-notes are depicted by a rectangle. The permutations allowed by the PQ-tree in Figure 5.3 include among others

$$\begin{array}{lllll} AB & ODE & F & GH & IJ \\ AB & EDO & F & GH & IJ \\ EDO & BA & F & GH & IJ \\ F & ODE & BA & JI & HG. \end{array}$$

In general, PQ-trees serve to solve the following problem [1]:

Problem 14. *Given a finite set $\mathcal{U}$ and a collection $\mathcal{S}$ of subsets of $\mathcal{U}$, find a permutation $\Pi(\mathcal{U})$ such that for each $S_i \in \mathcal{S}$ the elements of S_i are consecutive in Π.*

This problem is solved with a PQ-tree in an iterative approach. We start with the PQ-tree allowing all possible permutations (*i.e.*, it consists only of all the leaves connected to a P-node). We then add each constraint in $\mathcal{S}$ one at a time.

To add a constraint $S_i \in \mathcal{S}$ we try to modify the tree until the elements of S_i are consecutive. This can be tested in $O(|S_i|)$ amortized time. If this cannot be done, then there is no possible permutation which satisfies the constraints. If it can be done, then we update the tree to reflect the new constraint.

Booth and Lueker showed that this can be done with only a constant number of replacement rules in $O(|S_i|)$ amortized time. Therefore, in total, adding a constraint can be done in $O(|S_i|)$ amortized time. In all, determining whether or not a the set $\mathcal{U}$ of elements has a permutation in which all consecutiveness constraints $S_i \in \mathcal{S}$ are satisfied can be done in time and space $O(|\mathcal{U}|)\sum_{S_i \in \mathcal{S}} |S_i|)$ amortized time. Pull details can be found in Booth and Lueker [19].

For example, let $\mathcal{U} = \{A, B, C, D\}$, and $\mathcal{S} = \{S_1 = \{A, B, C\}, S_2 = \{AD\}\}$. Figure 5.4 shows the steps in adding the constraints to the PQ-tree. The final tree gives all possible orderings which satisfy the constraints: $\{B, C, A, D\}, \{C, B, A, D\}, \{D, A, C, B\}$, and $\{D, A, B, C\}$.

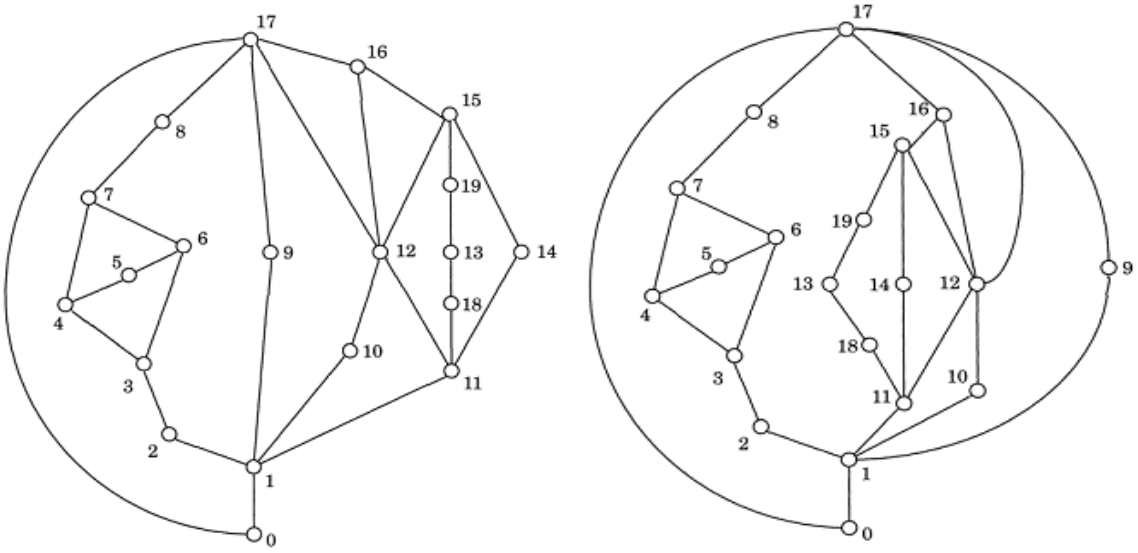

Figure 31: [10]. The second embedding is obtained from the first one by means of two swap operations around the split pairs $(1, 17)$ and $(11, 15)$ and one flip operation around the split pair $(1, 17)$.

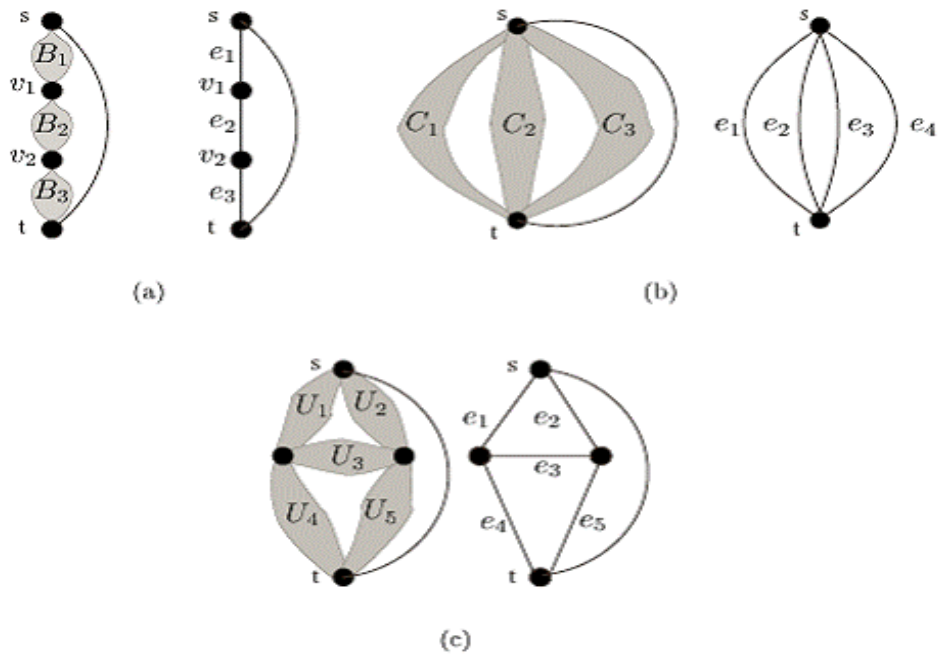

Figure 32: [93] Pertinent and skeleton graphs of the different node types of an SPQR-tree. The shaded regions represent subgraphs, (a) an S-node, (b) a P-node, and (c) an R-node.

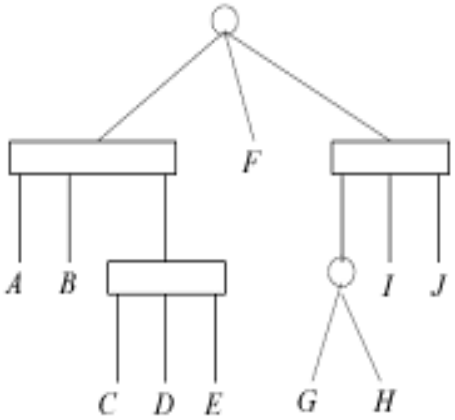

Figure 33: [1] Example of a PQ-tree.

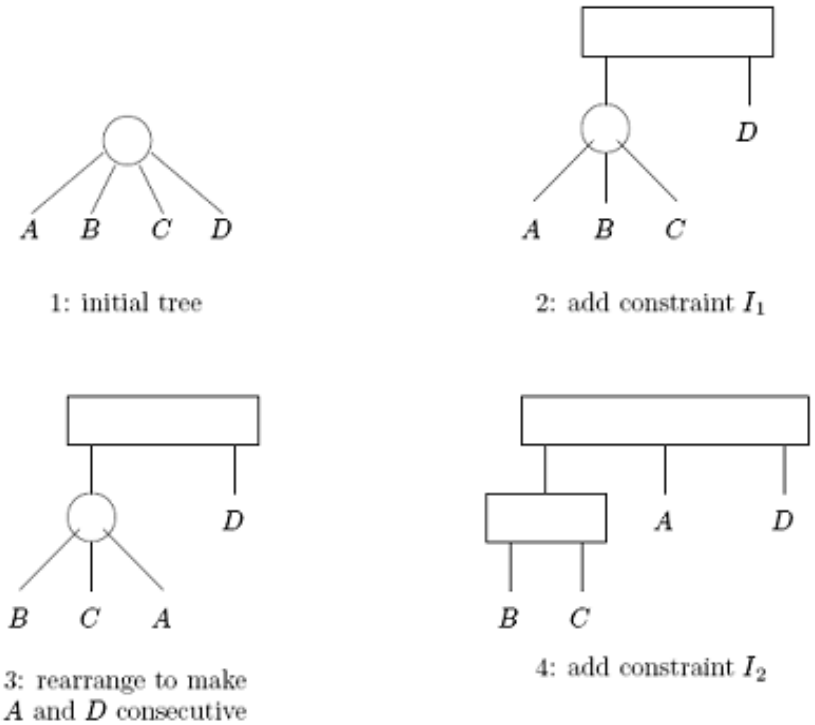

Figure 34: [1] Example of adding constraints to a PQ-tree ($I_1 = S_1, I_2 = S_2$).