Technische Universität Wien

Vienna University of Technology

Diplomarbeit

Optimal Crossing Minimization using Integer Linear Programming

Ausgeführt am
Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

Unter der Anleitung von
Univ.Prof. Dr. Petra Mutzel

und

Univ.Ass. Dr. Gunnar W. Klau sowie
Univ.Ass. Dr. René Weiskircher

durch

Dietmar Ebner
Schlagergasse 6/15
1090 Wien

Februar 2005 _____

Acknowledgments

First of all I want to thank my advisor Petra Mutzel and her group at the department of Algorithms and Data Structures at the Technical University of Vienna for the possibility to write my thesis about this interesting topic and for the outstanding support during the last years. Her great experience and expert knowledge in the field of Graph Drawing was a great help and her enthusiasm for this topic was always a great motivation. This work originated from ideas developed by her group and a number of contributing people and the results could never have been achieved without her support.

My special thanks also belong to Gunnar Klau and René Weisskircher. In addition to this thesis they provided me with a number of interesting topics for practical classes and their courses have always been interesting and instructive. Especially I want to thank Gunnar, who took upon him to proofread this thesis and some other papers, and for the large number of fruitful discussions.

The major part of the practical work of this thesis was done during a six month stay at the University of La Laguna in Tenerife. I want to thank Prof. Dr. María Belén Melián Batista and her group at the institute for Estadística, Investigación Operativa y Computación for providing me with the necessary facilities, their support and for the possibility to spend a great time on this wonderful island.

Furthermore I want to thank Christoph Buchheim, who provided me with a proof-of-concept implementation of our algorithm and for his large number of useful suggestions and ideas.

Thanks also to Prof. Barth for representing Prof. Mutzel and to Prof. Raidl and Prof. Krall for holding the final exam.

Last but not least I want to thank all my friends for the wonderful time we spent together and my parents for their support during all those years.

Short Abstract

In this thesis we study the Crossing Minimization Problem. We are looking for a drawing for a given graph in the plane, such that a minimum number of edges cross. Although this problem was shown to be *NP* hard, algorithms are needed in practice. Application areas occur, *e.g.*, in the area of Graph Drawing and *VLSI* design.

While the problem mostly is attacked using heuristic approaches in practice, we present an algorithm that is able to solve medium sized instances to provable optimality in reasonable computation time.

We show how we can solve the "general" Crossing Minimization Problem by "reducing" the problem to simple drawings using a transformation on the input graphs. We call a drawing simple, if every edge crosses at most one other edge. The latter problem is attacked using a Branch-and-Cut algorithm in combination with Integer Linear Programming. This technique could be successfully applied to attack many prominent *NP* hard combinatorial optimization problems in the past.

The performance of our new algorithm in terms of solution quality and runtime is tested on a large benchmark set derived from real world graphs and the results are compared to existing heuristic methods.

Deutsche Zusammenfassung

Diese Diplomarbeit behandelt das sogenannte Kreuzungsminimierungsproblem. Gesucht ist eine Zeichnung eines gegebenen Graphen in der Ebene, sodass sich möglichst wenige der Kanten kreuzen. Durch zahlreiche Anwendungsgebiete in der Praxis, z.b. im automatischen Zeichnen von Graphen oder im Bereich des *VLSI* Entwurfs, ist der Bedarf an entsprechenden Algorithmen hoch.

Aufgrund der Komplexität des Problems verwenden die meisten praktischen Anwendungen heuristische Ansätze. Im Gegensatz dazu präsentieren wir einen exakten Algorithmus, der in der Lage ist, Instanzen mittlerer Grösse innerhalb weniger Minuten beweisbar optimal zu lösen.

Das Kreuzungsminimierungsproblem wird vorerst auf einfache Zeichnungen reduziert. Einfache Zeichnungen sind solche, in denen jede Kante maximal eine andere Kante kreuzt. Für einfache Zeichnungen präsentieren wir einen Algorithmus basierend auf Ganzzahliger Linearer Programmierung und Branch-and-Cut. Diese Technik wurde in den letzten Jahren bereits erfolgreich zur Lösung zahlreicher prominenter kombinatorischer Optimierungsprobleme angewendet.

Wir untersuchen die Leistungsfähigkeit des Algorithmus in Bezug auf Lösungsqualität und Laufzeit mittels Testinstanzen, die von Echtdaten abgeleitet wurden und vergleichen die Ergebnisse mit bestehenden heuristischen Verfahren.

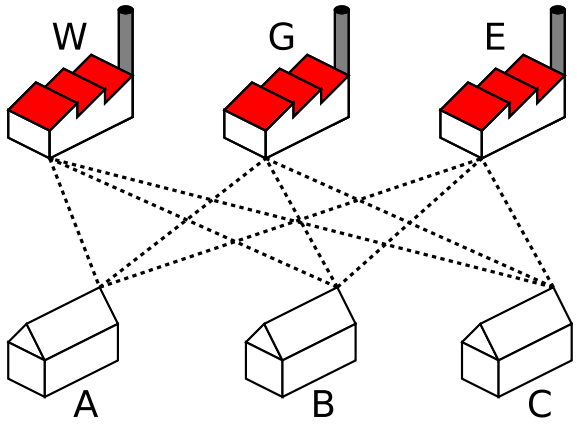
Contents

List of Figures

1 Introduction

Three Utilities

Suppose you are an architect in a small village. Your task is to supply the three houses A, B and C with water, gas and electricity (W, G, E) such that the pipes do not cross each other. How can every house be fully equipped?



Looking for an answer is not worth the trouble since there is no solution to this puzzle unless we allow some "tricks" such as pipes that pass through one of the buildings or drawings on the surface of a torus.

It is obvious to translate the puzzle into the framework of graph theory. A *graph* is a collection of nodes and edges connecting a pair of nodes. Usually graphs are visualized by representing nodes with points in the plane and edges are drawn as curves connecting the nodes.

In graph theoretic terms our sample corresponds to the complete bipartite graph $K_{3,3}$. We can partition the nodes into two distinct sets F and H (factories and houses) such that for each pair of vertices $u \in F$, $v \in H$ there is an edge (u, v). F and H have both cardinality three. What is remarkable about this graph is that it is impossible to embed it into the plane without any edge crossings. A graph that can be drawn in this way is called a *planar* graph.

Another interesting example is the complete graph K_5. It consists of five nodes and there is an edge e for every pair of distinct nodes. As we will see later, the Polish

mathematician Kazimierz Kuratowski provided a full characterization of planar graphs based on this two simple structures. He proved that a graph is planar if and only if it contains no subdivision of K_5 or $K_{3,3}$. This important result is widely known as *Kuratowski's Theorem*.

Even if a graph is not planar we often need to answer the following question. *What is the minimum number of edge crossings in any drawing of a given graph in the plane?* This problem is also known as the *Crossing Number Problem* while the *Crossing Minimization Problem* asks for a drawing in the plane with a minimum number of crossings.

P. Turàn posed this problem for the first time in his "Notes of Welcome" in the first issue of the *Journal of Graph Theory*. In following quotation he describes his experiences in a labor camp during the Second World War (see [46]).

> We worked near Budapest, in a brick factory. There where some kilns where the bricks were made and some open storage yards where the bricks where stored. All the kilns were connected by rail with all the storage yards. The bricks were carried on small wheeled trucks to the storage yards. All we had to do was to put the bricks on the trucks at the kilns, push the trucks to the storage yards and unload them there. We had a reasonable piece rate for the trucks and the work itself was not difficult; the trouble was only at the 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 which was rather precious to all of us (for reasons not to be discussed here). We were all sweating and cursing at such occasions, I too; but *nolens-volens* the idea occurred 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? I realized after several days that the actual situation could have been improved, but the exact solution of the general problem with m kilns and n storage yards seemed to be very difficult.

In other words, Turàn was looking for the Crossing Number of the complete bipartite graph $K_{n,m}$. In 1953, K. Zarankiewicz and K. Urbaník independently claimed a solution for this problem.

$$\mathrm{cr}(K_{m,n}) = \lfloor \frac{m}{2} \rfloor \lfloor \frac{m-1}{2} \rfloor \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \qquad \text{(conjecture)}$$

In 1965, 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 complete bipartite graph. Further details are given in Section 3.3.

This work concentrates on the Crossing Minimization Problem for general graphs. Although it was shown to be *NP* hard, algorithms are needed in practice. While the

problem typically is attacked using heuristic approaches we give an optimal algorithm based on Integer Linear Programming (*ILP*) and branch-and-cut.

To complete this section we present two important application areas, automatic graph drawing and *VLSI* design. Furthermore we give an overview of this thesis in Section 1.2.

1.1 Practical Applications

1.1.1 Automatic Graph Drawing

Graphs are a widely used technique to model and visualize relations between objects. The growth of complexity and the large number of applications leads to an increasing need for automatic layout tools, *e.g.*, in business process modelling, software engineering and database design. Therefore this relatively new research area in computer sciences received more and more attention in the last years.

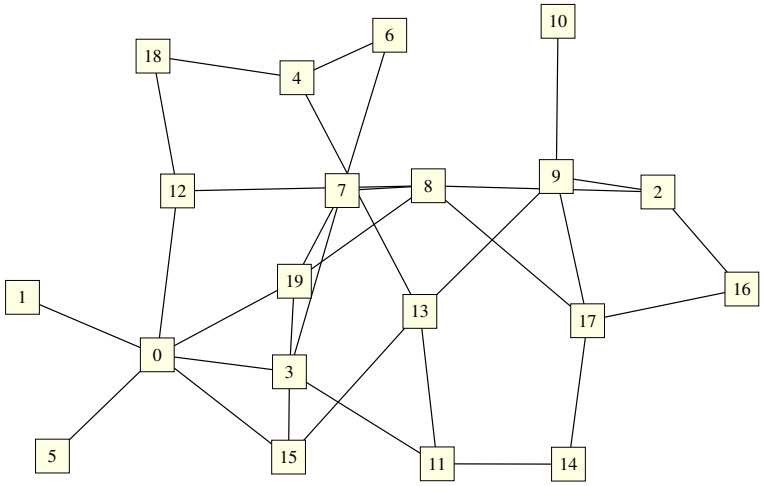
The main goal is to draw graphs such that they are easy to read and understand. Thus the quality of a layout often depends on the particular use case and it is difficult to model the "niceness" of a layout. However, there is a number of widely accepted aesthetic criteria to measure the quality of a given drawing. Vertices should be even distributed over the available space, symmetries should become visible, overlaps between nodes and other objects should be avoided and the length of edges should be small. Furthermore a small drawing area is desired in many cases. Often those criterias are contradictory and preferences have to be defined depending on the particular application.

H. Purchase published a study about the effects of different aesthetic criteria on human understanding (see [40]). It points out that crossing minimization is the most important criterion. Crossings decrease the readability of a drawing since it is often difficult to follow the edges. The complexity of a drawing strongly grows with its number of crossings. Figure 1.1 shows two sample drawings of the same graph drawn with many 1.1(a) and few 1.1(b) crossings.

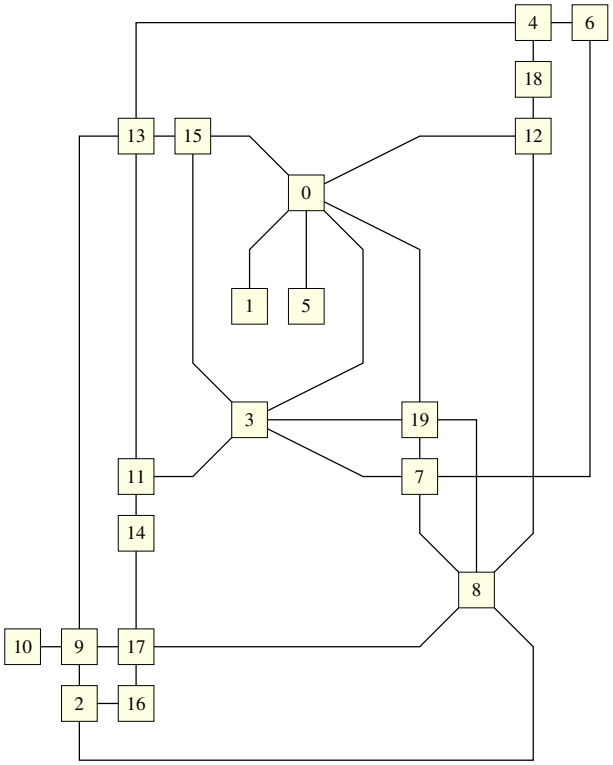
Algorithms for Planar Graphs

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 (see Section 2.1). 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 (see [48]). Tutte presented in [47] an algorithm that produces a straight line drawing for planar graphs. A sample layout produced with Tutte's algorithm is shown in Figure 1.2(a). Furthermore there are algorithms by De Fraysseix, Pach and Pollack (see [11]) and by Schnyder (see [42]) that draw graphs with n vertices on a grid of size $O(n^2)$. We show sample drawings of their algorithms in Figure 1.2(b) and Figure 1.2(c).



(a)



(b)

Figure 1.1: Two different drawings of the same graph with many crossings (a) and an optimum of two crossings (b).

If we allow bends for edges we can use an algorithm proposed by Tamassia (see [44]). 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. Figure 1.2(d) shows the sample graph drawn with a pseudo orthogonal layout. Further information concerning this extensions can be found in [26].

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. We discuss this approach in detail in Section 3.4.

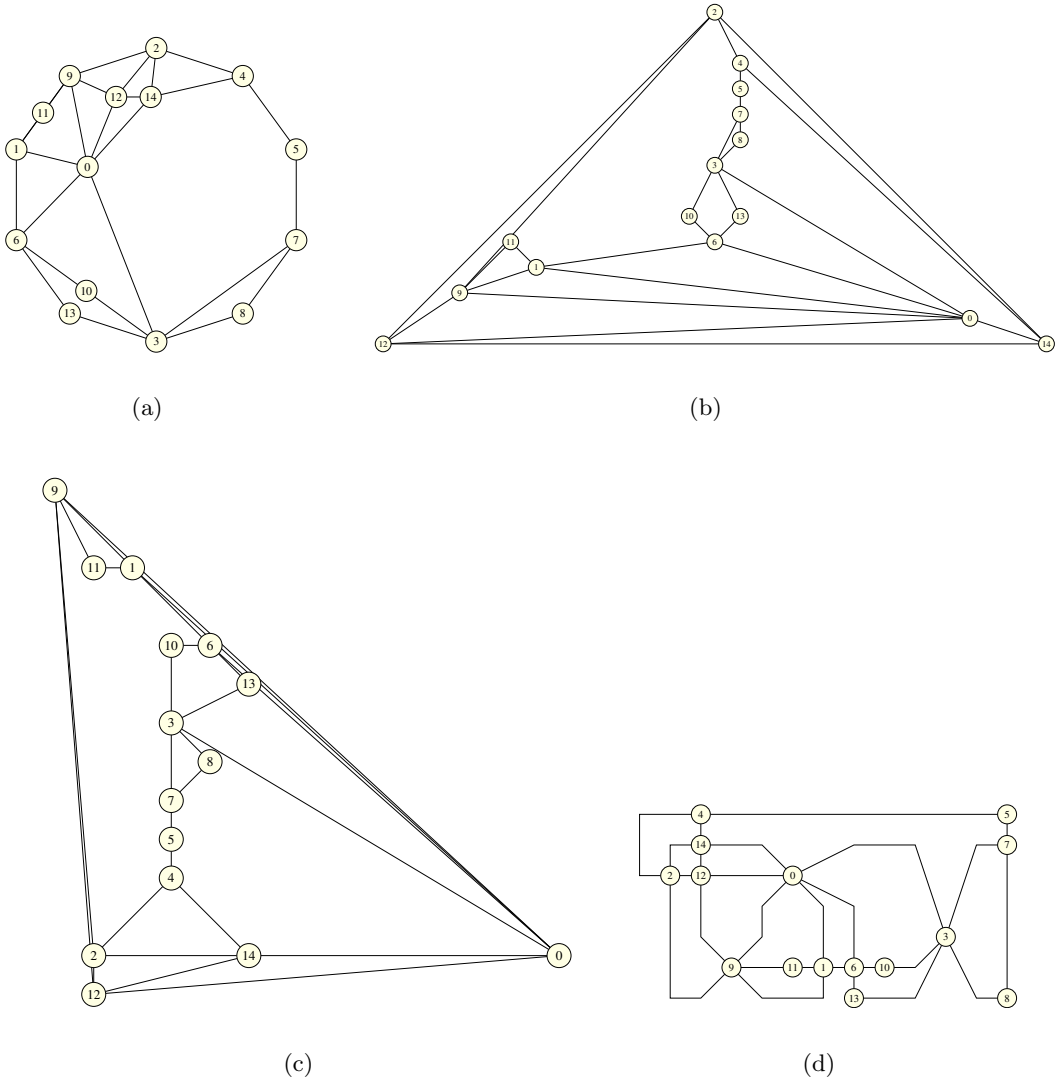
In addition to this approach there are two more widespread techniques, the so called *spring embedder method* and the *hierarchical method*.

Spring embedders were introduced by Eades in [13]. 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. Usually this method yields to an even distribution of the vertices on the available space and even edge lengths. 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. We show a sample drawing produced by a spring embedder in Figure 1.3(a).

The hierarchical method goes back to Sugiyama, Tagawa and Toda (see [43]). 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.

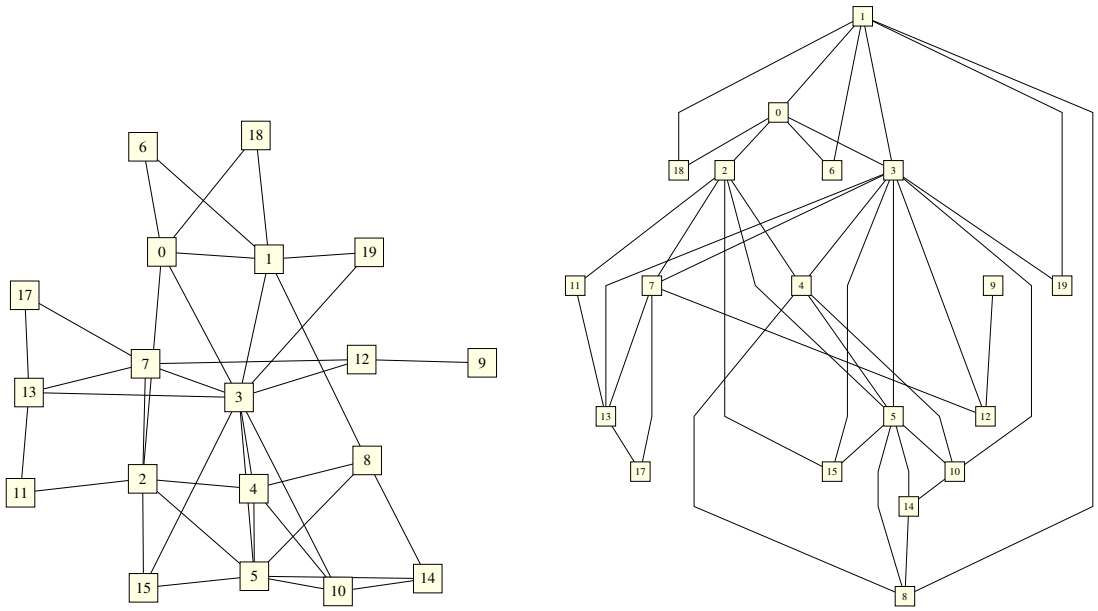
We show a drawing using the hierarchical layout in Figure 1.3(b). The algorithm works well if the input graph has a "natural" layering or a hierarchical drawing is preferred, *e.g.*, in flow- or sequence diagrams.



(a) (b)

(c) (d)

Figure 1.2: A planar graph drawn with different layout algorithms: (a) Tutte, (b) De Fraysseix et al., (c) Schnyder, (d) extended version of Tamasia's algorithm.



(a) (b)

Figure 1.3: The same nonplanar graph drawn with a spring embedder (a) and a hierarchical layout using the Barycenter heuristic for the crossing minimization step (b).

1.1.2 VLSI Design

VLSI Design (Very Large Scale Integration) deals with the layout of integrated circuits on a single chip. There is no exact definition but the number of transistors usually exceeds 10000 items. We can understand a chip layout as a collection of components that are connected by wires. Since chips are usually produced in a high number of items the layout of the circuit on the board plays a major role in the design of integrated circuits. There are several quality criteria that have to be met, such as the minimization of the layout areas, the total edge length or the number of bends per wire.

One of the most important 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.

Those 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. Furthermore 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 crossing is one of the most important steps in the layout phase.

Due to the large number of components the method to minimize the number of crossings described in this paper is not yet suitable for this application area. Furthermore there are a number of conditional constraints such as a maximum edge length or a minimal distance to the neighboring wire that complicate this task. Therefore practical layout tools rely on fast and specialized heuristics to accomplish this task. However, new insights concerning the Crossing Minimization Problem could lead to better working heuristics.

1.2 Guide to this Thesis

In Chapter 2.1 we introduce some basic definitions concerning graphs and define the terminology that is used in the rest of the paper. We give a description of Kuratowski's Theorem and mention some linear time algorithms to test the planarity of a graph. Furthermore Section 2.2 gives an introduction to (Integer) Linear Programming and points out some important results like weak and strong duality. We also summarize the column generation approach to deal with a large number of variables. At last we describe the cutting plane approach and show how we can combine it with the branch-and-bound method to solve *ILP*s without an optimal algorithm that solves the associated separation problem.

In Chapter 3 we give an overview of the "state of the art" in crossing minimization. We sketch the *NP*-completeness proof by Garey and Johnson and describe several know variants of the Crossing Number Problem. As far as they are known we furthermore present relations between the different definitions. A important "ingredient" for the

branch-and-cut approach are tight lower and upper bounds. We summarize known results for special classes of graphs and general bounds on the Crossing Number in Section 3.3. Moreover we describe the most important heuristic method for the crossing minimization of nonplanar graphs, the so called planarization approach.

The main algorithm is addressed in Chapter 4. We present an *ILP* formulation for the "general" Crossing Minimization Problem by reducing the problem to the Crossing Minimization Problem restricted to drawings such that each edge crosses at most one other edge. We call such drawings *simple drawings* and present some related theoretical results in Section 4.1. In order to decrease the number of variables in the given formulation we neglect crossings between adjacent edges or edges in different biconnected components and show that this step does not affect the optimality of our results. The last Section describes how the particular components can be combined to develop a branch-and-cut based algorithm that solves the Crossing Minimization Problem to optimality.

Clearly this algorithm can take exponential time. We test its performance on a widely used benchmark set derived from real world graphs and compare the solution quality and runtime to the best known heuristic methods in Chapter 5.

2 Preliminaries

This chapter presents the basic definitions and concepts which will be used in the rest of the thesis and introduces some basic results in graph theory and linear programming. Furthermore, Section 2.3.2 gives an overview of the branch-and-cut approach. Some of the definitions in this chapter are taken from [50].

2.1 Graph Theory

A graph is a mathematical structure often used to define relationships between objects. It consists of a set of vertices V and pairs of vertices connecting them (edges).

Definition 2.1.1 (Graph). *A tuple $G = (V, E)$ is considered a graph if V is a finite set of vertices (nodes) and E is a finite multiset of edges (arcs). An edge is a tuple $v, w \in V$. A graph is called* directed *if its edges are ordered tuples of two nodes and* undirected *otherwise.*

If an edge $e = (v, w)$ is directed, the node v (w) is called the *source* (*target*) of e. Two edges with a common end vertex or two vertices $v, w \in V$ which are connected by an edge $e = (v, w) \in E$ are called *adjacent*. The vertices v and w are *incident* to e.

We denote with $\delta(v)$ the set of edges incident to vertex $v \in V$, $|\delta(v)|$ is called the *degree* of v. If G is directed $|\delta^+(v)|$ and $|\delta^-(v)|$ denotes the *out-* respective *indegree* of v. A graph is called *r-regular* if all vertices $v \in V$ have degree r.

A *path* in a graph G is a sequence of vertices such that from each of the vertices there is an edge to the successor vertex. A path is called *simple* if none of the vertices in the path are repeated. A *cycle* is a path starting and ending at the same node.

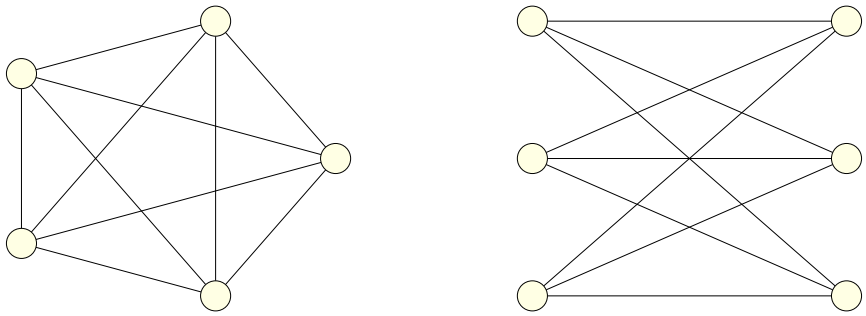
Definition 2.1.2 (k-connectivity). *A graph $G = (V, E)$ is k-connected (k-edge-connected) if at least k vertices (edges) must be deleted to disconnect G. A graph that is 2-connected (3-connected) is also called* biconnected *(*triconnected*).*

A graph $G' = (V', E')$ is called a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. We call G the *supergraph* of G'. An edge $e = (v, v)$ is called a *loop*. A graph without loops and parallel edges is called a *simple graph*.

Let $G = (V, E)$ and $G' = (V', E')$ be two simple graphs. We call G and G' *isomorphic* if there exists a one-to-one function $\varphi : V \to V'$ with the property that $e = (u, v) \in E$ if and only if $e' = (\varphi(u), \varphi(v)) \in E'$.

A simple directed graph is called *bidirected* if each edge $e = (v, w)$ has a unique reversal edge $\hat{e} = (w, v)$.

The *complete graph* K_n is defined as a simple graph G with $|V| = k$ and the property that for every pair of nodes $u, v \in V$ there is an edge $e = (u, v) \in E$. A graph G is



(a) (b)

Figure 2.1: Sample drawings of the complete graph K_5 (a) and the complete bipartite graph $K_{3,3}$ (b).

bipartite if its set of vertices V can be partitioned into two sets A and B such that all edges $e \in E$ have one incident vertex in A and one in B. A *complete bibartite graph* $K_{n,m}$ is a bipartite graph such that $|A| = n$, $|B| = m$ and for each pair of vertices $a \in A$ and $b \in B$ there is and edge $e = (a, b) \in E$. Figure 2.1 shows a sample drawing of K_5 and $K_{3,3}$.

Drawings of Graphs A Drawing of a graph $G = (V, E)$ in the plane is a mapping of each vertex $v \in V$ to a distinct point and each edge $e = (v, w) \in E$ to an arc connecting the incident vertices v and w without passing through any other vertex. A drawing is called a *good drawing*, if it satisfied the following conditions

i no edge crosses itself

ii adjacent edges do not cross

iii non-adjacent edges cross at most once

A common point of two edges in a drawing that is not an incident vertex is called a *crossing*. The *crossing number* $\mathrm{cr}(G)$ is defined as the minimum number of crossings in any drawing of G. Graphs that can be drawn without any crossings ($\mathrm{cr}(G) = 0$) are also called *planar graphs*. There is an infinite number of different drawings of a planar graph, nevertheless they can be divided into a finite set of equivalence classes of drawings called the *embeddings* of a graph. Drawings that realize the same embedding are in a topological sense equivalent. Figure 2.2 shows a simple example of two different embeddings of a planar graph.

Each planar embedding partitions the plane into connected components, called *faces*. There is one face with unbouded area, called *outer face*. A planar embedding induces for each node $v \in V$ a counterclockwise ordering of the edges incident to v. All planar

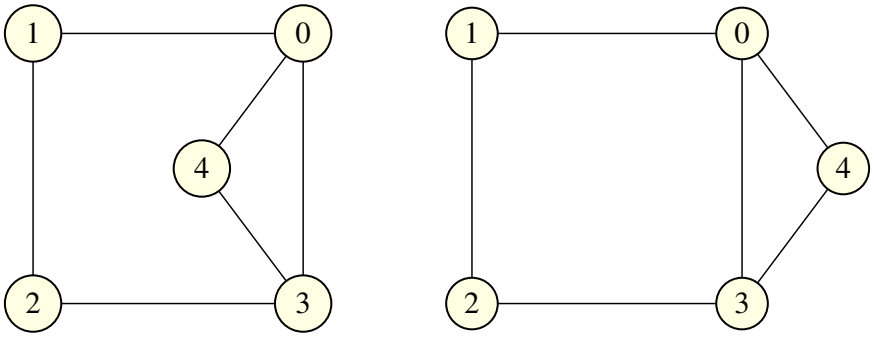


Figure 2.2: Different embeddings of the same planar graph

embeddings with the same cyclic ordering for each vertex form an equivalence class which is called a *combinatorial embedding* $\Pi(G)$. Each face in a combinatorial embedding can be the outer face of a corresponding planar embedding. An alternative and equivalent way to define a combinatorial embedding is to specify the set of circuits which bound the faces.

Euler has shown a close connection between the number of edges, nodes and faces in a connected planar graph, known as Euler's formula.

Theorem 2.1.3 (Euler's formula). *Let G be a simple connected planar graph with n vertices and m edges, then in every combinatorial embedding $\Pi(G)$ of G the number of faces is*

$$f = m - n + 2 \tag{2.1}$$

Kuratowski's Theorem Already in 1930, the Polish mathematician Kazimierz Kuratowski provided a characterization of planar graphs, now known as Kuratowski's theorem. It follows from Euler's formula that the number of edges m in a simple connected planar graph with n nodes is bound by

$$m \leq 3n - 6 \tag{2.2}$$

If in addition the graphs contains no cycles of length 3, then

$$m \leq 2n - 4 \tag{2.3}$$

It follows from inequality 2.2 that K_5 is not planar and from inequality 2.3 that $K_{3,3}$ is not planar. Therefore no graph containing a subdivision of K_5 or $K_{3,3}$ can be planar. A subdivision S of a graph G is obtained by repeatedly replacing edges e by a path of length two.

Kuratowski proved that also the converse is true which leads to

Theorem 2.1.4 (Kuratowski's theorem). *A finite graph is planar if and only if it contains no subgraph that is isomorphic to or is a subdivision of K_5 or $K_{3,3}$.*

A proof of Euler's formula and Kuratowski's theorem can be found in [36].

Planarity Testing This section shortly summarizes two important results concerning planarity tests and planar embeddings: the algorithm by Hopcroft and Tarjan (see [22]) and the algorithm due to Booth and Lueker (see [8]).

The algorithm of Hopcroft and Tarjan is a simplified version of a linear time planarity testing algorithm presented in 1971 by Tarjan in his thesis. The algorithm is based on depth-first search and a divide-and-conquer strategy by Auslander and Parter (see [2]).

Booth and Lueker's algorithm is based on the vertex addition algorithm proposed by Lempel, Even and Cederbaum in [29] and runs also in linear time. After starting with a single vertex, a sequence of induced subgraphs of G is constructed. In every step the current subgraph is tested for planarity. The runtime of the original algorithm of $O(|V|^2)$ could be improved to a linear time algorithm by Booth and Lueker using a new data structure: PQ-trees.

Both algorithms can be extended to compute a combinatorial embedding without increasing their computational complexity.

2.2 (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:

$$\begin{aligned} &\text{maximize } c^T x \\ &\text{subject to } Ax \leq b \\ &x \geq 0 \end{aligned}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$. The linear function $c^T x : \mathbb{R}^n \to \mathbb{R}$ is called the *objective function* and the inequalities in the system $Ax \leq b$ are called *constraints*.

We can easily transform inequalities of the form $a^T x \geq b$ by multiplying them with -1, which leads to $-a^T x \leq -b$ and equations ($a^T x = b$) can be expressed by the two inequalities $a^T x \leq b$ and $a^T x \geq b$. Furthermore, we can transform minimization problems to maximization problems by multiplying the objective function $c^T x$ by -1.

Definition 2.2.1 (Linear Programming Problem). *Given a matrix $A \in \mathbb{R}^{m \times n}$, and vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, find a vector $\hat{x} \in \mathbb{R}^n$ with*

$$c^T \hat{x} = \max \{c^T x \mid Ax \leq b\} \tag{2.4}$$

Linear Programs can be solved in polynomial time. The most widespread algorithms are the simplex- (see [9]), the ellipsoid- and the interior point method (see [41]). There are a number of well developed and optimized implementations that efficiently solve linear programs, even on a very large scale.

Definition 2.2.2. *For every Linear Program* $P = \max\{c^T x \mid Ax \leq b\}$ *(*primal problem*) the* dual problem D *is defined as* $D = \min\{y^T b \mid A^T y = c^T, y \geq 0\}$.

Primal and their dual problems are closely connected. If D is the dual problem of a primal problem P, then the dual of D is equal to P. Duality can be used to prove the optimality of linear programs in an elegant way.

Theorem 2.2.3 (Weak duality). *If* $\hat{x}$ *is a feasible solution of* P *and* $\hat{y}$ *is a feasible solution of* D*, then* $c^T \hat{x} \leq \hat{y}^T b$.

In other words: every feasible solution of D gives us a bound on the optimum value of D.

Theorem 2.2.4 (Strong duality). *If* P *has an optimum solution* $\hat{x}$*, then* D *has an optimum solution* $\hat{y}$ *and* $c^T \hat{x} = \hat{y}^T b$.

Integer Linear Programming When we try to formulate combinatorial optimization problems using linear programs, some or all of the variables often have to take integer values. Adding so called *integrality constraints* leads to a *mixed integer linear program* (*MILP*), if we require all variables to be integer we call the program *integer linear program* (*ILP*). A special but very common case occurs when all variables have to be zero or one (decision variables). Such programs are also referred as *(mixed) zero-one integer linear programs*.

Unfortunately, the integrality constraints render the problem *NP*-complete in the general case (see [15] for a proof). Assuming $P \neq NP$, there is no polynomial time algorithm that is able to find a vector $\hat{x}$ such that $c^T \hat{x} = \max\{c^T x \mid Ax \leq b, x_i \text{ integer } \forall i \in I\}$ for a subset $I \subseteq \{1, ..., n\}$.

2.3 Dealing with 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. 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.

2.3.1 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* $L(J) = \max\{\sum_{j \in J} c_j x_j \mid \sum_{j \in J} A_{i,j} x_j \leq b\}$ for some $J \subseteq \{1, ..., n\}$. Using the dual variables $\hat{y}$ 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.

2.3.2 Branch-and-Cut

Cutting Planes

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:

Definition 2.3.1 (General Separation Problem). *Given a class of valid inequalities and a vector $y \in \mathbb{R}^n$, either prove that y satisfies all inequalities in the class, or find an inequality which is violated by y.*

We call an algorithm for the general separation problem an *exact separation algorithm*. If the algorithm can not guarantee that there are no violated inequalities if none was found, we call it *heuristic separation algorithm*.

When we want to apply the cutting plane approach to zero-one *ILP*s (as we will do), we can transform the ILP $I = \max\{c^T x \mid Ax \leq b, x \in \{0,1\}^n\}$ to a linear program by dropping the integrality constraints and adding for each variable x_i an inequality $x_i \leq 1$. The resulting linear program $L = \max\{c^T x \mid Ax \leq b, 0 \leq x_i \leq 1, \forall i \in \{1, ..., n\}\}$ is called the *linear relaxation* of I.

Every optimum solution $\hat{x}$ 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* (see [17]). 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).

Branch-and-Bound

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.

Branch-and-Cut

The *branch-and-cut* approach was first used by Grötschel, Jünger and Reinelt in [31] 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 [25].

3 The Crossing Minimization Problem

This chapter gives an overview of the "state of art" in crossing minimization. Section 3.1 outlines the *NP*-completeness proof by Garey and Johnson. We furthermore describe several variants of the "general" Crossing Number, such as the *t-Polygonal Crossing Number*, the *Linear Crossing Number* and the *Pairwise and Odd Crossing Number* defined by Pach and Tóth. Moreover Section 3.2.4 contains some results concerning drawings with a limited number of crossings per edge.

An important ingredient for a successful branch-and-cut algorithm are tight lower and upper bounds. We summarize known results for particular families of graphs such as for *complete graphs* and *complete bipartite graphs* and give some known general lower bounds for the crossing number of graphs in Section 3.3.

Due to the complexity of the Crossing Minimization Problem practical applications rely on good heuristics. Section 3.4 describes the so-called *planarization approach* which is the most widespread technique to attack the problem in practice. We furthermore summarize a number of improvements for the basic heuristic and discuss computational studies to compare the solution quality of different strategies for the planarization- and the edge re-insertion step.

3.1 Problem Definition and Computational Complexity

In Section 2.1 we already defined $\mathrm{cr}(G)$ to be the minimum number of crossings in any drawing of a graph G. Using this notation we define the *Crossing Number Decision Problem* as follows:

Definition 3.1.1 (Crossing Number Decision Problem). *Let $G = (V, E)$ be a graph and K a positive integer. Is $\mathrm{cr}(G) \leq K$?*

Garey and Johnson proved the *NP*-completeness of the Crossing Number Decision Problem in [16]. They provided a reduction of the *Optimal Linear Arrangement Problem*, which is already known to be *NP*-complete, to the crossing number problem in polynomial time. Optimal Linear Arrangement is defined as follows:

Definition 3.1.2 (Optimal Linear Arrangement Problem). *Let $G = (V, E)$ be a simple graph and K a positive integer. Is there a one-to-one function $f : V \to \{1, ..., |V|\}$ such that*

$$\sum_{e=(v,w)\in E} |f(v) - f(w)| \leq K \tag{3.1}$$

The reduction is done in two steps. First, Optimal Linear Arrangement is reduced to the *Bipartite Crossing Number*, which is defined as follows:

Definition 3.1.3 (Bipartite Crossing Number). *Let $G = (V_1, V_2, E)$ be a bipartite graph and K a positive integer. Can G be drawn in a unit square such that all vertices in V_1 are on the northern boundary, all vertices in V_2 are on the southern boundary, all edges are within the square, and there are at most K crosssings?*

Then, Bipartite Crossing Number is reduced to Crossing Number.

Lemma 3.1.4. *Optimal Linear Arrangement can be reduced to Bipartite Crossing Number in polynomial time.*

Proof. Given a pair $[G, K]$ with $G = (V, E)$ and $V = \{v_1, v_2, ..., v_n\}$ of an instance of Optimal Linear Arrangement, Garey and Johnson define a corresponding instance of Bipartite Crossing Number $[G', K']$ with $G'(V_1, V_2, E_1 \cup E_2)$ as follows:

$$
\begin{aligned}
&V_1 = \{u_1, u_2, ..., u_n\} \\
&V_2 = \{w_1, w_2, ..., w_n\} \\
&E_1 = \{|E|^2 \text{ copies of } (u_i, w_i)\ \forall i \in \{1, 2, ..., n\}\} \\
&E_2 = \{(u_i, w_j) \mid i, j \in \{1, 2, ..., n\}, i < j, (v_i, v_j) \in E\} \\
&K' = |E|^2(K - |E|) + (|E|^2 - 1)
\end{aligned}
$$

To show that an Instance of Optimal Linear Arrangement $[G, K]$ and its corresponding instance of Bipartite Crossing Number $[G', K']$ are equivalent, we need to prove that the answer for $[G, K]$ is "yes" if and only if the answer for $[G', K']$ is "yes"

Given an ordering function f for G such that $\sum_{e=(u,v)\in E} |f(u) - f(v)| \leq K$, we can construct a drawing of G' as follows: We place the vertices $u_i \in V_1$ uniformly distributed at the upper boundary of the unit square in the order given by $f(u_i)$, *e.g.*, if $f(u_i) = n$ then u_i is drawn as the n-th node from left to right. In the same way nodes $w_i \in V_2$ are placed at lower boundary. We can easily insert the $|E|^2$ edges $e \in E_1$ without any crossings as shown in Figure 3.1. When inserting the edges $e = (u_i, w_j) \in E_2$, every edge crosses exactly $|f(u_i) - f(w_j)| - 1$ "bundles" of $|E|^2$ edges from E_1. The total number of crossings between edges from E_1 and edges from E_2 is at most

$$\sum_{e=(u,v)\in E} (|f(u) - f(v)| - 1)|E|^2 \leq |E|^2(K - |E|) \tag{3.2}$$

The total number of crossings among E_2 is at most $\binom{|E|}{2} \leq |E|^2 - 1$. Thus the total number of crossings is at most $|E|^2(K - |E|) + (|E|^2 - 1) = K'$.

To prove the remaining direction, suppose the answer to the Bipartite Crossing Number problem $[G', K']$ is "yes", which means there is a drawing of G' with less than K' crossings. We can construct a bijection $f_1 : V_1 \to \{1, ..., n\}$ from the ordering of the vertices from V_1 on the upper boundary in a natural way. In the same way we get a one-to-one function $f_2 : V_2 \to \{1, ..., n\}$ from the ordering on the lower boundary. It is easy to prove that f_1 and f_2 must be identical. If we can find $i, j \in \{1, 2, ..., n\}$ such that without loss of generality $f_1(v_i) < f_1(v_j)$ and $f_2(v_i) > f_2(v_j)$, then the drawing would

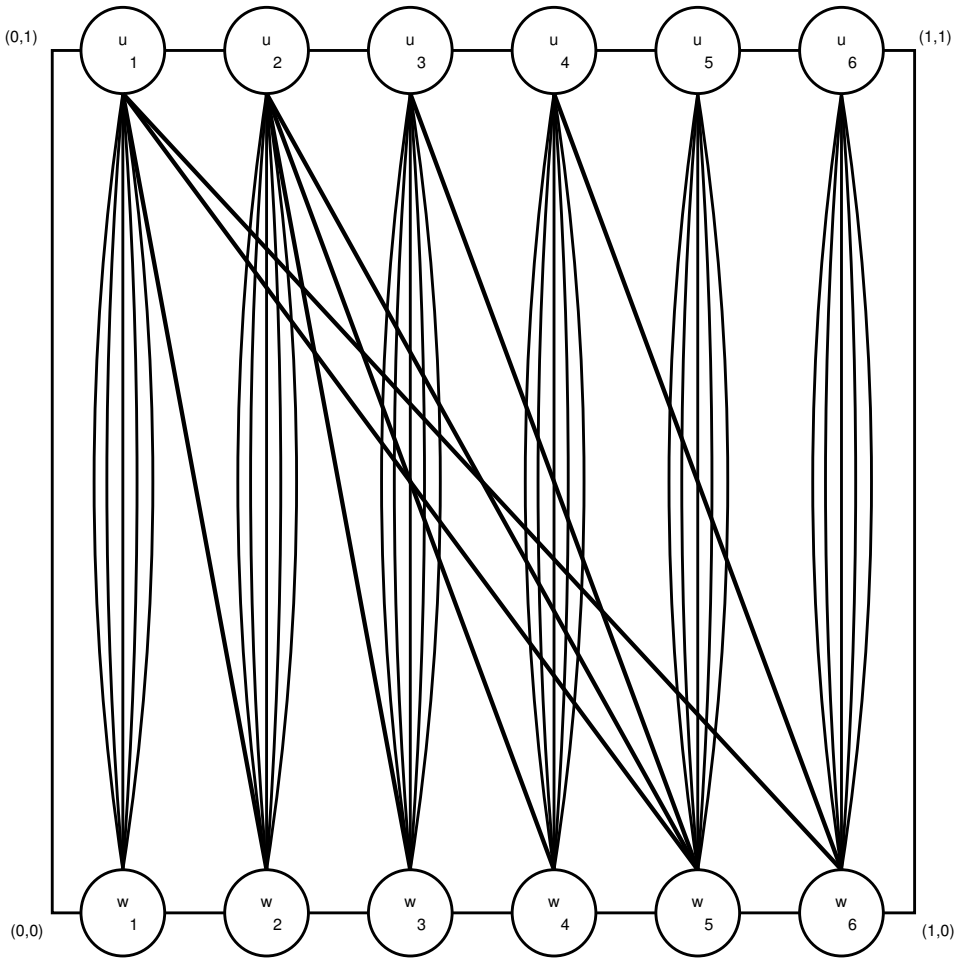


Figure 3.1: Instance of a Bipartite Crossing Number Problem corresponding to an instance of Optimal Linear Arrangement

contain at least $|E|^4$ crossings (since at least two "bundles" of $|E|^2$ edges cross). This is a contradiction to our bound on K' limiting the number of crossings in the drawing.

Since all nodes on the upper and lower boundary are ordered as outlined in Figure 3.1 each edge $e = (u_i, w_j) \in E_2$ is involved in at least $(|f_1(v_i) - f(v_j)| - 1)|E|^2$ crossings. Then

$$\sum_{e=(u,v)\in E} (|f(u) - f(v)| - 1)|E|^2 \leq K' = |E|^2(K - |E|) + (|E|^2 - 1) \tag{3.3}$$

If we divide the whole inequality by $|E|^2$ and take into account that $\frac{|E|^2-1}{|E|^2} < 1$ we get

$$\sum_{e=(u,v)\in E} (|f(u) - f(v)| - 1) \leq K - |E| + 1 \leq K \tag{3.4}$$

and f_1 satisfies the Linear Arrangement Problem. □

The next step is to reduce Bipartite Crossing Number to Crossing Number.

Lemma 3.1.5. *Bipartite Crossing Number can be reduced to Crossing Number in Polynomial time*

Proof. Similar to the proof of Lemma 3.1.4, Garey and Johnson define for every instance of Bipartite Crossing Number $[G, K]$ with $G = (V_1, V_2, E)$ an instance of Crossing Number $[G', K]$ with $G' = (V', E \cup E_1 \cup E_2 \cup E_3)$ as follows:

$$V' = V_1 \cup V_2 \cup \{u_0, w_0\}$$
$$E_1 = \{3K + 1 \text{ copies of } (u_0, u)\ \forall u \in V_1\}$$
$$E_1 = \{3K + 1 \text{ copies of } (w_0, w)\ \forall w \in V_2\}$$
$$E_3 = \{3K + 1 \text{ copies of } (u_0, w_0)\}$$

Again, G' can easily be constructed in polynomial time. It should be noted that K is identical for both problem instances. We need to show that G has a drawing in the unit square as required in Definition 3.1.3 with at most K crossings if and only if G' has a drawing in the plane with at most K crossings.

Given a drawing of G in the unit square, we can easily add the additional vertices and edges of G such that the crossing number is not increased as outlined in Figure 3.2.

It remains to show that, given a drawing of G' in the plane with K crossings, there is also a drawing of G with no more than K crossings. The proof is done using four normalization steps. Since it is rather technical it is not included here. Details can be found in [16]. □

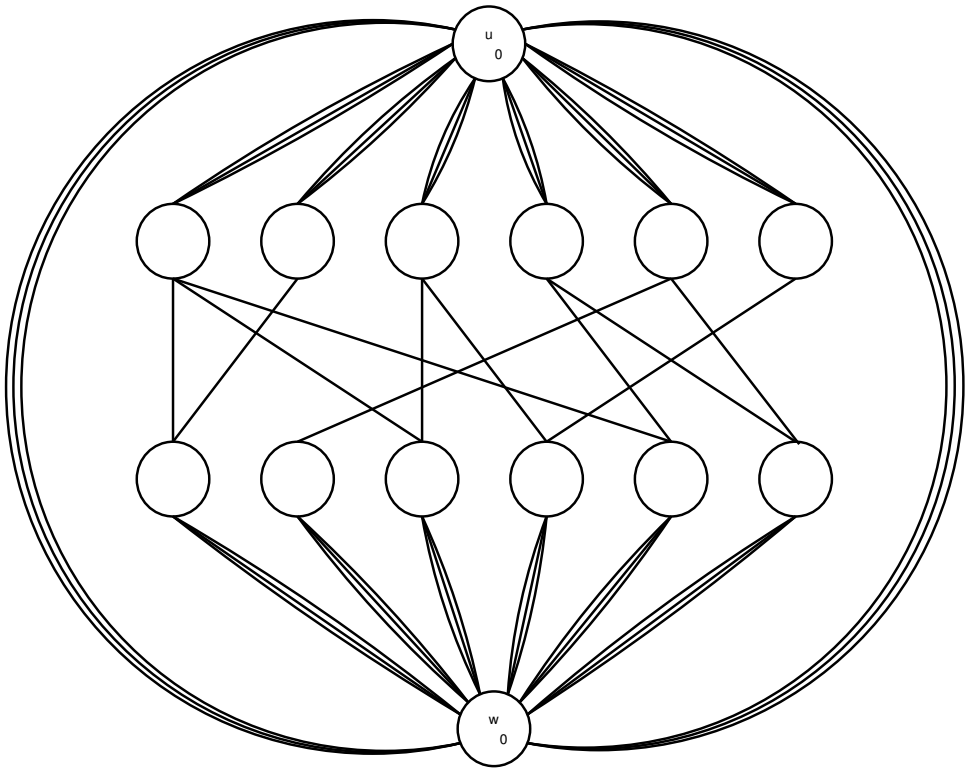


Figure 3.2: Instance of a Crossing Number Problem corresponding to an instance of Bipartite Crossing Number

3.2 Variants of Crossing Number

In contrast to the "general" Crossing Minimization Problems, a number of variants restricted to special types of graphs or additional properties can be found in the literature.

3.2.1 t-Polygonal Crossing Number

A well studied variant of the Crossing Number Problem is the *Rectilinear Crossing Number*, defined as follows.

Definition 3.2.1 (Rectilinear Crossing Number). *Let $G = (V, E)$ be a non-planar, simple graph. The* Rectilinear Crossing Number $\mathrm{cr}_1(G)$ *is the minimum number of crossings in a drawing of G where all edges are drawn as straight lines.*

Since any solution of the Rectilinear Crossing Number Problem is also a solution for the "general" Crossing Number Problem, it is easy to see that $\mathrm{cr}(G) \leq \mathrm{cr}_1(G)$ for any graph G.

Bienstock and Dean proved in [6] that for graphs with crossing number at most three, the Rectilinear Crossing Number and the Crossing Number are equal.

Theorem 3.2.2. *If* $\mathrm{cr}(G) \leq 3$ *then* $\mathrm{cr}_1(G) = \mathrm{cr}(G)$.

They could further show that there are graphs G such that $\mathrm{cr}_1(G)$ is arbitrary large, even if $\mathrm{cr}(G)$ is only four.

Theorem 3.2.3. *For every $m > k \geq 4$ there is a graph G with* $\mathrm{cr}(G) = 4$*, but* $\mathrm{cr}_1(G) \geq m$.

As a generalization to the Rectilinear Crossing Number, Bienstock introduced in [5] the concept of the *t-Polygonal Crossing Number*.

Definition 3.2.4 (t-Polygonal Crossing Number). *Let $G = (V, E)$ be a graph. A t-polygonal drawing of G, for $t \geq 1$, is a good drawing where every edge is drawn as a t-polygonal line,* i.e.*, a polygonal line with at most t segments. The* t-Polygonal Crossing Number $\mathrm{cr}_t(G)$ *is defined as the minimum number of crossings in any t-polygonal drawing of G.*

For $t = 1$ the t-Polygonal Crossing Number is equal to the Rectilinear Crossing Number. Bienstock also showed that there can be no polynomial time algorithm for producing optimal t-polygonal drawings of G unless $P = NP$ and that there is no fixed t such that $\mathrm{cr}(G) = \mathrm{cr}_t(G)$ for any graph G.

3.2.2 Linear Crossing Number

Given a simple graph G, a drawing of G is called a *linear drawing* if all vertices lie on a straight line and edges are drawn as semi-ellipses above and below this line.

Figure 3.3 shows an example with one crossing of a linear drawing. The Crossing Minimization Problem restricted to linear drawings is known as *Linear Crossing Minimization Problem* and is defined as follows.

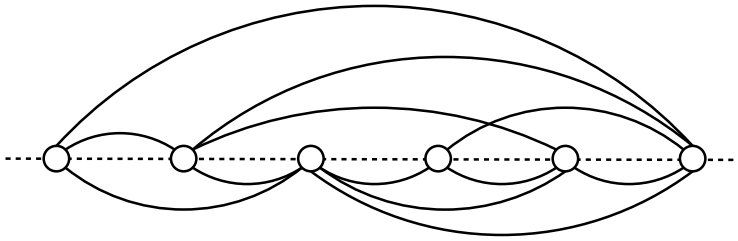


Figure 3.3: Linear Drawing of a graph G. Vertices are placed on a horizontal line and edges are drawn as semi-ellipses.

Definition 3.2.5 (Linear Crossing Minimization Problem). *Let $G = (V, E)$ be a simple graph. Find a linear drawing of G such that the number of crossings is minimal among all linear drawings of the given graph. The number of crossings of a minimum linear drawing is denoted as $\mu(G)$.*

A special variant of the Linear Crossing Number Problem arises, when the arcs are only allowed to be drawn on one side of the line. We denote the number of crossings in an optimum drawing in this case with $\mu^+(G)$. We can state that $\mathrm{cr}(G) \leq \mu(G) \leq \mu^+(G)$ since any linear embedding (with only one degree of freedom) cannot have less crossings than any embedding in the two dimensional space.

There is a connection to the "general" Crossing Minimization Problem proved by Nicholson in [35]:

Theorem 3.2.6. *Any graph drawn in the plane with a minimum number of crossings can be redrawn with an equivalent crossing structure such that the resulting drawing has the following properties:*

i all vertices are placed on a horizontal line

ii all edges are drawn as a series of semi-ellipses such that successive semi-ellipses lie on different sides of the horizontal line.

It is interesting to see that the complexity of the problem stays the same, even if we fix the ordering of the vertices of V. This variant is known as the *Fixed Linear Crossing Minimization Problem* and is formally defined as follows.

Definition 3.2.7 (Fixed Linear Crossing Minimization Decision Problem). *Given a graph $G = (V, E)$, a fixed ordering on the vertices and an integer K, find a linear drawing of G with the specified ordering of the vertices such that $\mu(G) \leq K$.*

Masuda et al. proved in [32] that even this variant is *NP*-complete.

3.2.3 Pairwise and Odd Crossing Number

In their paper "Which Crossing Number is it Anyway?" Pach and Tóth define two further possibilities how to count the number of crossings in a graph (see [38]).

Definition 3.2.8. *Let $G = (V, E)$ be a simple graph.*

i The pairwise crossing number *of G* $\mathrm{pcr}(G)$ *is the minimum number of pairs of edges $(e_1, e_2) \in E^2$, $e_1 \neq e_2$ such that e_1 and e_2 determine at least one crossing, over all drawings of G.*

ii The odd-crossing number *of G* $\mathrm{ocr}(G)$ *is the minimum number of pairs of edges $(e_1, e_2) \in E^2$, $e_1 \neq e_2$ such that e_1 and e_2 cross an odd number of times.*

The authors also prove that $\mathrm{ocr}(G) \leq \mathrm{pcr}(G) \leq \mathrm{cr}(G) \leq \mathrm{cr}_1(G)$ and show the following theorems.

Theorem 3.2.9. *For every graph G, we have*

$$\mathrm{cr}(G) \leq 2(\mathrm{ocr}(G))^2 \tag{3.5}$$

Theorem 3.2.10. *Given a graph G and a positive integer K, it is NP-complete to decide whether* $\mathrm{pcr}(G) \leq K$*, or whether* $\mathrm{ocr}(G) \leq K$*.*

3.2.4 Restrictions on the Number of Crossings per Edge

In Chapter 4 we concentrate on a special variant of the Crossing Minimization Problem. We only consider drawings of a graph $G = (V, E)$ where every edge $e \in E$ crosses at most one other edge.

As a result of Euler's Formula, the number of edges m in any planar graph with n nodes is bounded by $3n - 6$ (see Equation 2.2). Denoting the maximum number of crossings per edge with k, Pach and Tóth generalize this bound in [37] for $k \leq 4$:

Theorem 3.2.11. *Let $G = (V, E)$ be a simple graph drawn in the plane so that every edge is crossed by at most k others. If $0 \leq k \leq 4$, then we have*

$$|E| \leq (k + 3)(|V| - 2) \tag{3.6}$$

They could further prove that this bound cannot be improved for $0 \leq k \leq 2$ and that for *any* $k \geq 1$ the following inequality holds:

$$|E| \leq \sqrt{16.875k}|V| \approx 4.108\sqrt{k}|V| \tag{3.7}$$

Another interesting result by Bodlaender and Grigoriev can be found in [7]. The authors prove the following theorem:

Theorem 3.2.12. *The problem to determine if a given graph G can be embedded on the plane with crossing parameter 1 is NP-complete.*

The proof is done by a reduction of the well known strongly *NP*-complete problem *3-PARTITION* (see [7] for details).

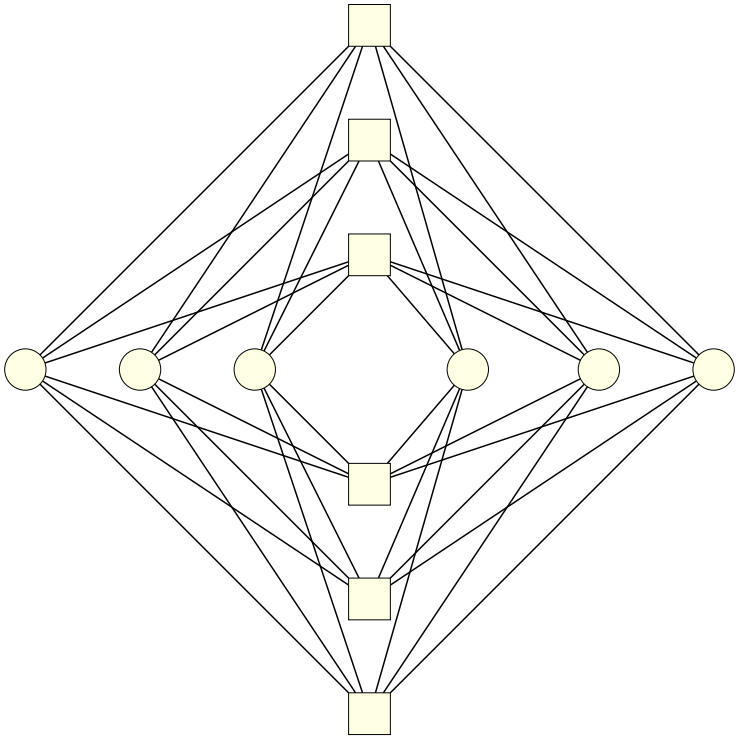


Figure 3.4: Drawing of $K_{6,6}$ with a minimum number of 36 crossings using Zarankiewicz's rule.

3.3 Known Bounds

3.3.1 Bounds for particular Families of Graphs

Complete Bipartite Graphs

Already in 1953, K. Zarankiewicz and K. Urbaník claimed a solution for the Crossing Number Problem on Complete Bipartite Graphs:

$$\mathrm{cr}(K_{m,n}) = \lfloor \frac{m}{2} \rfloor \lfloor \frac{m-1}{2} \rfloor \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \qquad \text{(conjecture)} \tag{3.8}$$

The right hand side of Equation 3.8 is denoted in the following as $Z(m,n)$. Any complete bipartite graph $K_{m,n}$ can be drawn with $Z(m,n)$ crossings by placing the vertices in vertex set A at coordinates $(i(-1)^i, 0)$ for all $i = 1, ..., m$ and the vertices of vertex set B at coordinates $(0, j(-1)^j)$ for all $j = 1, ..., n$. All edges are drawn as straight lines. Figure 3.4 shows a sample drawing of $K_{6,6}$ with 36 crossings.

In 1965, P. Kainen and G. Ringel found an error in the induction argument of Zarankiewicz's proof, hence Equality 3.8 is only a conjecture so far. However, the provided drawing rule gives us an upper bound for $\mathrm{cr}(K_{m,n})$.

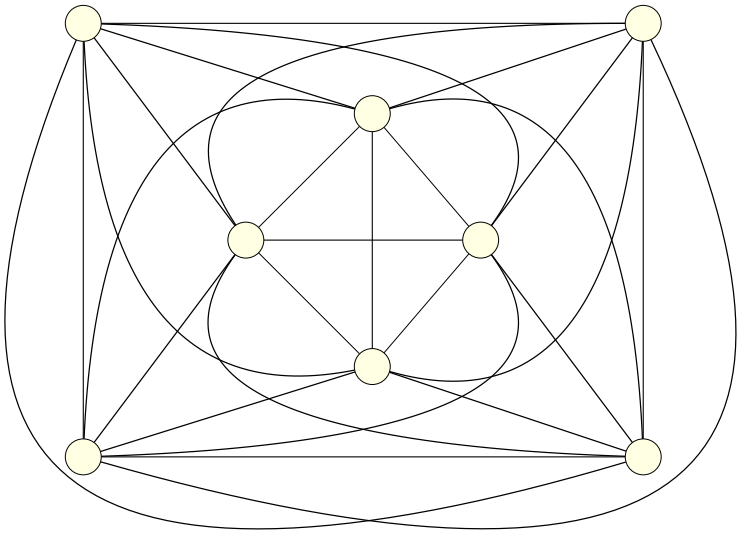


Figure 3.5: Drawing of K_8 with a minimum number of 18 crossings.

Complete Graphs

As for complete bipartite graphs there is also a conjecture for the number of crossings of a complete graph K_n with n nodes.

$$\mathrm{cr}(K_n) = \frac{1}{4}\lfloor\frac{n}{2}\rfloor\lfloor\frac{n-1}{2}\rfloor\lfloor\frac{n-2}{2}\rfloor\lfloor\frac{n-3}{2}\rfloor \qquad \text{(conjecture)} \tag{3.9}$$

From constructions of drawings (see [21]) we know that Equation 3.9 is an upper bound for $\mathrm{cr}(K_n)$. For complete graphs up to ten nodes, the correctness of 3.9 has been verified by Guy in [20]. Figure 3.5 shows a sample drawing of the complete graph K_8 with a minimum number of 18 crossings.

In [14], Erdös and Guy furthermore prove the following lower bound for $\mathrm{cr}(K_n)$:

$$\mathrm{cr}(K_n) \geq \frac{1}{80}\, n\, (n-1)\, (n-2)\, (n-3) \tag{3.10}$$

3.3.2 General Bounds

Unfortunately there are nearly no known general upper bounds for $\mathrm{cr}(G)$. 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

$$\mathrm{cr}(G) \leq \mathrm{cr}(K_n) \leq \frac{1}{4}\lfloor\frac{n}{2}\rfloor\lfloor\frac{n-1}{2}\rfloor\lfloor\frac{n-2}{2}\rfloor\lfloor\frac{n-3}{2}\rfloor \tag{3.11}$$

A simple lower bound can be obtained from Euler's formula. Since any planar simple connected graph G with n nodes cannot have more than $3n-6$ edges, clearly $\mathrm{cr}(G) \geq m - 3n + 6$. If, in addition, G contains no triangle then $\mathrm{cr}(G) \geq m - 2n + 4$.

In 1983, Leighton used induction on the number of nodes to show the following theorem (see [28]). We denote with $G(n, m)$ the family of simple graphs with n vertices and m edges.

Theorem 3.3.1. *Let $G \in G(n, m)$. If $m \geq 4n$, we have*

$$\mathrm{cr}(G) \geq \frac{1}{100} \frac{m^3}{n^2} \tag{3.12}$$

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

We already discussed an important paper presented by Pach and Tóth in Section 3.2.4. As a consequence of Theorem 3.2.11 the authors derive a general lower bound on the Crossing Number of a graph (see [37]).

Theorem 3.3.2. *Let $G \in G(n, m)$, then the Crossing Number $\mathrm{cr}(G)$ satisfies*

$$\mathrm{cr}(G) \geq \frac{1}{33.75} \frac{m^3}{n^2} - 0.9n \tag{3.13}$$

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*.

Definition 3.3.3 (Skewness). *Let $G \in G(n, m)$. The* skewness $\mathrm{sk}(G)$ *is the minimum number of edges that must be deleted from G to obtain a planar subgraph.*

Each of the removed edges produces at least one crossing, hence it is not difficult to see that

$$\mathrm{cr}(G) \geq \mathrm{sk}(G) \tag{3.14}$$

for any graph G. Cimikowski showed in [10] that a planar graph can have skewness one, but an arbitrary high Crossing Number (see Figure 3.6). Computing the skewness is equivalent to the *maximum planar subgraph problem*. While we can easily compute a *maximal planar subgraph* in polynomial time, the maximum planar subgraph problem was shown to be *NP*-hard by Liu and Geldmacher in [30]. While the maximum planar subgraph problem asks for planar subgraph with maximum cardinality among all subgraphs, a planar subgraph $G' = (V', E')$ of $G = (V, E)$ is called maximal, if the addition of any edge $e \in V \setminus V'$ destroys the planarity of G'. Any maximum planar subgraph is clearly also a maximal planar subgraph while there are simple examples that disprove the opposite.

For the complete and complete bipartite graph, the skewness is known.

$$\mathrm{sk}(K_n) = \frac{n(n-1)}{2} - 3n + 6 \tag{3.15}$$

$$\mathrm{sk}(K_{m,n}) = mn - 2(m+n) + 4 \tag{3.16}$$

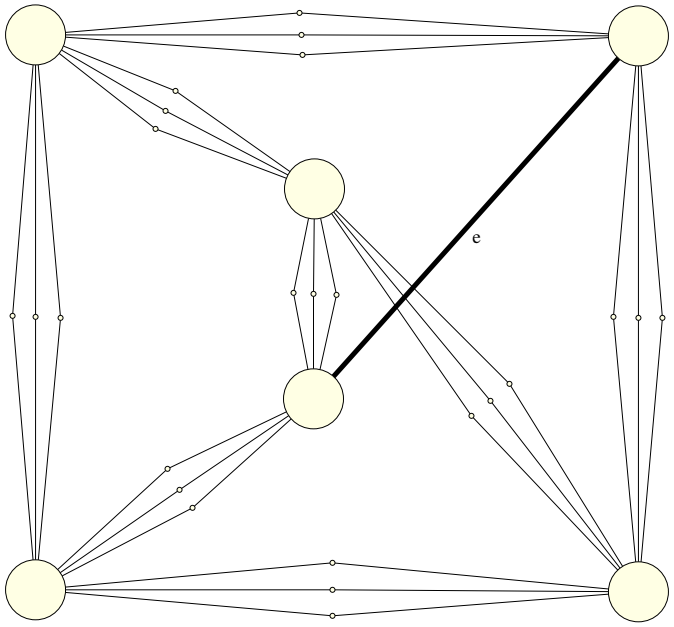


Figure 3.6: A sample graph with skewness one and arbitrary high Crossing Number.

3.4 Crossing Minimization in Practice

Due to its complexity, the crossing minimization problem is usually solved heuristically using a 2-step approach. The first step is to compute a maximum (or maximal) planar subgraph $G_P \subseteq G$. After determining a combinatorial embedding $\Pi(G_P)$ the deleted edges are reinserted such that the number of crossings is minimized. Since there is a large number of algorithms for drawing planar graphs, the crossings are usually replaced by artificial vertices, which are removed afterwards. The following procedure summarizes these steps to compute a combinatorial embedding for a given graph $G = (V, E)$:

1: Compute a maximum (maximal) planar subgraph $G_P = (V', E') \subseteq G$. Let $F = E \setminus E'$.
2: Determine a combinatorial embedding $\Pi(G_P)$
3: Reinsert all edges in F preserving the combinatorial embedding $\Pi(G_P)$ while trying to keep the number of crossings low.
4: Replace crossings by new artificial vertices
5: Draw the resulting graph using a planar graph drawing algorithm.
6: Remove the artificial vertices inserted in Step 5

As already mentioned above, the maximum planar subgraph problem is *NP*-hard. However, for medium sized instances the branch-and-cut approach suggested in [24] can be used to compute optimal subgraphs in an acceptable amount of time. The runtime depends heavily on the number of edges that have to be deleted. Besides this there is a large number of heuristics for the computation of maximal planar subgraphs. There is an algorithm presented by Djidjev in [12] that uses *SPQR*-trees and runs in time $O(n+m)$. Another widespread $O(n^2)$-time algorithm by Jayakumar et al. can be found in [23].

A very simple but widely used heuristic for the computation of a maximal planar subgraph is to start with an empty set of edges and iteratively try to add the edges of G. In every step a planarity testing algorithm is used to determine if the insertion of the edge would lead to a non-planar graph. If so, we disregard the new edge, otherwise it is added permanently. Since we can perform a planarity test in time $O(|V|)$ (see Section 2.1) the overall runtime of this heuristic is $O(|V||E|)$.

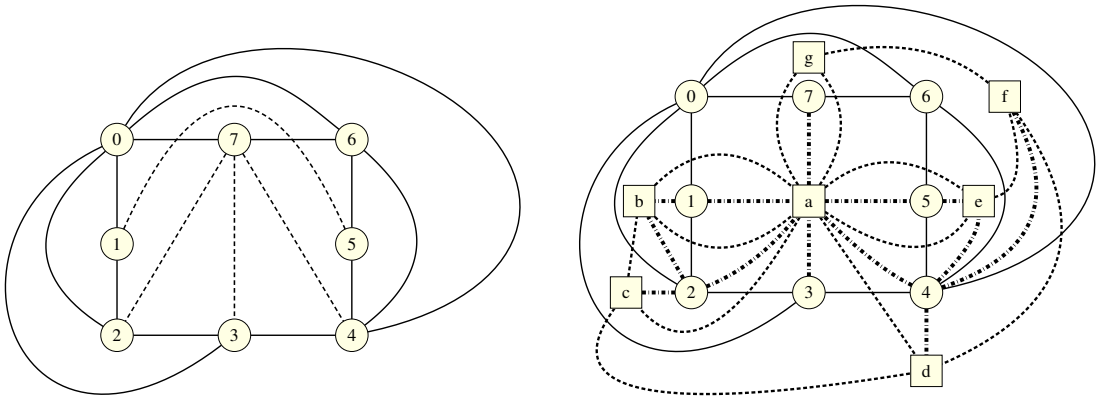
Improvements can be achieved by the use of *incremental planarity testing algorithms*. Instead of testing a whole graph for planarity, those algorithms test if an additional edge can be added to a planar graph without loosing planarity. Di Battista and Tamassia propose such an algorithm in [4]. It runs in time $O(\log(|V|))$ (worst case) and leads to a $O(|E|\log(|V|))$ time algorithm for the maximal planar subgraph problem. Also Westbrook proposes in [49] a datastructure, that allows to perform an incremental planarity test for biconnected graphs in amortized time $O(\alpha(|E|,|V|))$. We denote with $\alpha(|E|,|V|)$ the inverse of the Ackermann function. La Poutré improves this result in [39] and presents an $O(|V|+|E|\alpha(|E|,|V|))$ time algorithm for the maximal planar subgraph problem.

A more complete overview as well as a number of related results are given by Mutzel in [33].

The second major part of the planarization method is also known as the *Constrained Crossing Minimization Problem*. Given a planar graph $G = (V, E)$, a combinatorial embedding $\Pi(G)$ and a set of additional edges F connecting vertices in V, we try to find a crossing configuration of $G' = (V, E \cup F)$ that preserves the combinatorial embedding $\Pi(G)$ and minimizes the number of crossings. Also this step can be solved to optimality, *i.e.*, using the branch-and-cut algorithm presented by Ziegler in [50], for small instances, although it is shown to be *NP*-hard.

The best known heuristic methods for the constrained crossing minimization problem rely on the so-called *extended dual graph*. If we denote the combinatorial dual graph of a graph $G = (V, E)$ with respect to the combinatorial embedding $\Pi(G)$ with $G^* = (V^*, E^*)$, we can derive the extended dual graph as follows ($\Pi(G^*)$ denotes the corresponding combinatorial embedding of G^*).

For every vertex v that is an end vertex of an edge $e \in F$ we add a vertex v^* to V^* and place those vertices inside the face of $\Pi(G^*)$ that correspond to v. Furthermore, we connect each inserted node v^* to every node that appears at the boundary of the corresponding face $f^* \in \Pi(G^*)$. The combinatorial embedding $\Pi(G^*)$ is updated in a natural way as outlined in Figure 3.7. The left figure shows a graph G, its combina-



(a) (b)

Figure 3.7: Example of a graph (a) and its extended dual graph (b).

torial embedding $\Pi(G)$ and the set of edges F denoted as dashed lines while the right figure shows G together with its extended dual graph and the associated combinatorial embedding. The figure is taken from [50].

One of the best known heuristic methods based on the extended dual graph relies on the observation that we can insert *one* edge $e = (u, v)$ into the combinatorial embedding $\Pi(G)$ of G with a minimum number of crossings as follows. Let G^* be the extended dual graph of G with respect to $\Pi(G)$ and $F = \{e\}$. The graph G^* contains only two additional vertices u^* and v^* corresponding to the end vertices u and v of e. It can be shown that e can be inserted with a minimum number of crossings by computing a shortest-path from u^* to v^* in the extended dual graph. Each edge e^* on the shortest from u^* to v^* indicates a crossing of e with the corresponding edge in G.

We can use this method iteratively to reinsert all edges $e \in F$ in the following way:

1: $G' = G$, $\Pi(G') = \Pi(G)$
2: **while** $F \neq \emptyset$ **do**
3: Choose $e = (u, v) \in F$, $F = F \setminus \{e\}$
4: Compute the extended dual graph G^* of G' with respect to $\Pi(G')$ and $\{e\}$
5: Insert e with a minimum number of crossings in $\Pi(G')$ by computing a shortest path in G^* between u^* and v^*.
6: Replace crossings in G' by artificial vertices
7: **end while**
8: Replace all artificial vertices by crossings

The total number of crossings produced by the above procedure strongly depends on the insertion order of the edges in F. There are exponentially many permutations of the edges in F and for each edge there can also be exponentially many shortest paths in the extended dual graph. Unfortunately, even if we check all possible insertion orders and all possible shortest paths for each edge, we cannot guarantee to get an optimum solution. Ziegler presents in [50] an example graph with an optimum of 8 crossings such that the optimum solution is never found by the heuristic.

However, there are a number of modifications that try to improve the quality of the basic heuristic.

- *Permutation Heuristics* Instead of using a fixed random insertion order, the whole procedure is called n times with a different randomly generated permutation of the edges in F. The result of the algorithm is the solution of the permutation with the smallest number of crossings

- *Shortest First Heuristic* This variant computes the length of the shortest path in the extended dual graph for all edges in F and inserts the edge with the smallest number of crossings. This step is repeated until all edges have been inserted.

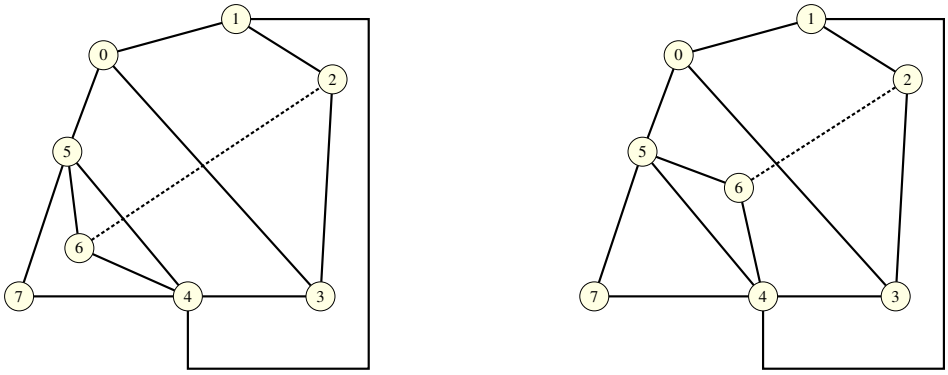
  The idea is that a large number of crossings extend the length of the shortest path for subsequent insertions and thus produces a larger number of overall crossings.

- *Remove and Reinsert Heuristic* Whenever we insert an edge into the combinatorial embedding $\Pi(G')$ we change the situation for previous inserted edges. If the edge is crossed by some subsequent inserted edges of F it could be "cheaper" to use another path in the extended dual graph.

  One possibility to improve this drawback is a postprocessing procedure that removes each edge and tries to reinsert it again by a shortest-path computation. Since any edge can always be inserted exactly as it was routed before, the number of crossings never increases after each step. After all edges have been reinserted, the situation could have changed again, so the whole procedure is repeated until no further improvement can be achieved.

In addition to the insertion order, the quality of the produced solution also highly depends on the chosen combinatorial embedding $\Pi(G)$. Figure 3.8 shows the optimum solution for the insertion of the dashed edge in two different embeddings. The left figure requires at least two crossings while the edge can be inserted in the embedding on the right with only one crossing. Gutwenger et al. give in [18] a linear time algorithm based on *SPQR*-trees which is able to insert one edge optimally into a planar graph G over all combinatorial embeddings of G.

The algorithm is based on the observation that only R-nodes of the *SPQR*-tree of G are crossing-critical. After determining a unique path in the *SPQR*-tree, the algorithm computes for each R-node u a shortest path in the skeleton of u. The total number of crossings can be computed by the sum of the lengths of the shortest paths.



(a) (b)

Figure 3.8: Optimum solution for the insertion of the dashed edge for a fixed embedding (a) and for a optimal embedding (b)

Gutwenger and Mutzel present in [19] an extensive experimental study of heuristics for crossing minimization based on the planarization approach. The authors compare the effects of various methods for the computation of a maximal planar subgraph and for the edge re-insertion as well as several postprocessing algorithms. The algorithm proposed in [18] performs outstandingly well, especially in conjunction with further modifications of the basic planarization approach.

4 Solving the Crossing Minimization Problem to Optimality

While many *NP*-hard combinatorial optimization problems could be attacked with mathematical programming in combination with the branch-and-cut technique for practical instances in the last years, there is no such approach for the Crossing Minimization Problem.

We present in this chapter an *ILP* formulation for the "general" Crossing Minimization Problem by reducing the problem to the Crossing Minimization Problem restricted to drawings such that each edge crosses at most one other edge. We call such drawings *simple drawings* and present the transformation as well as some related theoretical results in Section 4.1.

In Section 4.2 we describe our *ILP* formulation for simple drawings and prove its correcteness. Moreover we show in Section 4.3 how we can save variables in order to improve the given *ILP* and describe an algorithm to decompose a graph in its biconnected components in linear time.

We describe how we can simplify graphs in many cases to improve the runtime in Section 4.4 and outline the branch-and-cut approach based on the given *ILP* formulation in Section 4.5.

4.1 Simple Drawings of Graphs

In Section 3.2.4 we already considered drawings of graphs such that the number of crossings for each edge is at most k. It can be shown that the number of edges m for a graph $G = (V, E)$ with n nodes that is drawable in the plane such that each edge crosses at most k other edges is restricted to $(k+3)(n-2)$ for $0 \leq k \leq 4$ (see Theorem 3.2.11).

We will consider the special case $k = 1$ and denote a drawing that satisfies this restriction as a *simple drawing*.

Definition 4.1.1 (Simple Drawing). *Given a graph $G = (V, E)$. A* good drawing *of G is called a* simple drawing *if every edge $e \in E$ crosses at most one other edge.*

In fact there cannot be a simple drawing if the number of edges exceeds $4n - 8$. Since any complete graph K_n has a number of $\frac{n(n-1)}{2}$ edges, there is no such drawing if $\frac{n(n-1)}{2} > 4n - 8$ which is true for all $n \geq 7$. Figure 4.1 shows a sample simple drawing of K_6 while there is no such drawing for K_7. We can obtain similar results for the complete bipartite graph $K_{n,m}$.

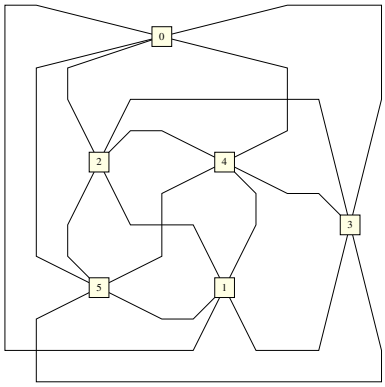


Figure 4.1: Optimal simple drawing of K_6 with three crossings.

Even if the number of edges is less than $4n - 8$, Bodlaender and Grigoriev prove in [7] that it is *NP*-complete to decide whether a given graph G has a simple drawing. If there is a simple drawing of G we denote the minimum number of crossings among all simple drawings of G by $\mathrm{crs}(G)$.

Definition 4.1.2. *Let G be a graph that is drawable in the plane such that each edge crosses at most one other edge. We define* $\mathrm{crs}(G)$ *to be the minimum number of crossings in any simple drawing of G.*

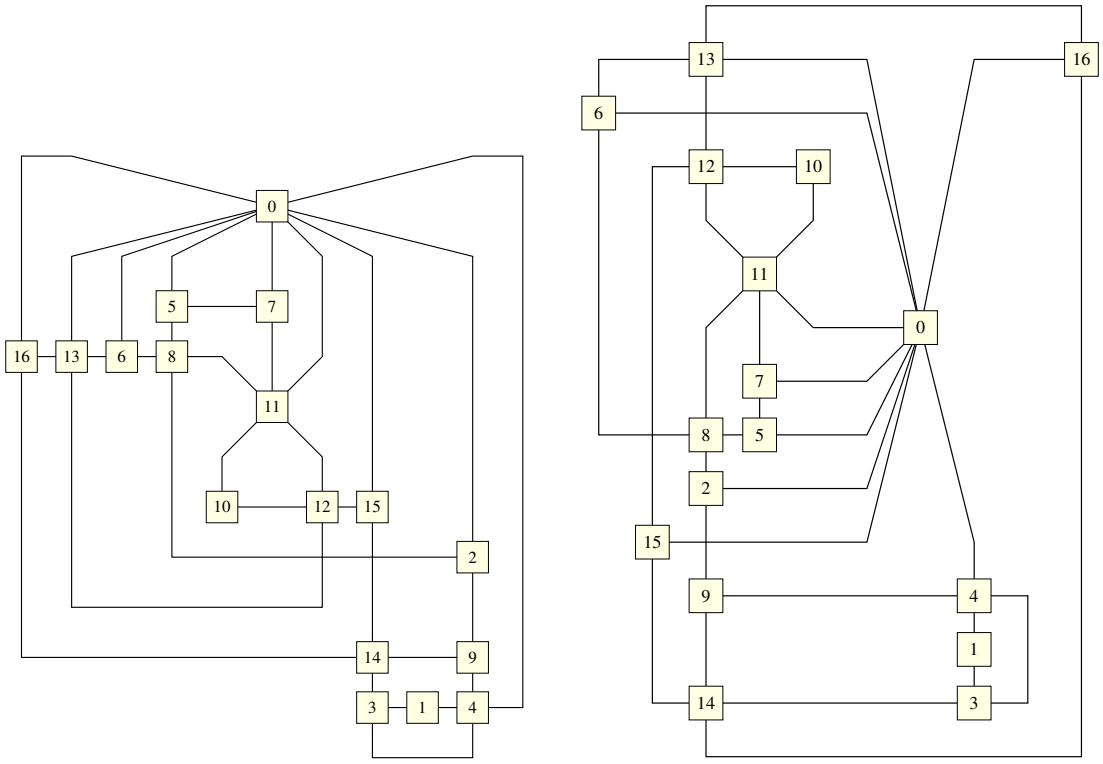
It is easy to see that $\mathrm{cr}(G) \leq \mathrm{crs}(G)$ since any simple drawing is also a good drawing of G. We cannot state equality because there are graphs G such that $\mathrm{crs}(G) > \mathrm{cr}(G)$. Consider the sample graph in Figure 4.2. The left drawing shows an optimum drawing with two crossings while the right drawing shows an optimum drawing among all simple drawings. Note that edge $(2, 8)$ crosses edges $(12, 13)$ and $(14, 15)$.

Given an integer l and a graph $G = (V, E)$ such that $l \geq |E|$ we can create a graph $G^* = (V^*, E^*)$ by replacing every edge $e \in E$ with a path of length l. Figure 4.3 shows an example that illustrates this transformation. The graph G^* contains a total number of $|V| + (l - 1)|E|$ nodes and $l|E|$ edges.

Lemma 4.1.3. *G can be drawn with n crossings if and only if there is a* simple drawing *of G^* with n crossings.*

Proof. Given a simple drawing of G^* we can easily create a drawing of G with the same number of crossings as follows. For every dummy node $d_i \in V^* \setminus V$ let $e = (u, d_i)$ and $f = (d_i, v)$ be the only two adjacent edges. We successively delete all dummy nodes and replace the original edges e and f with the edge (u, v). After all dummy nodes have been deleted the resulting graph is isomorphic to G and contains the same number of crossings as the simple drawing for G^*.

On the other hand we can construct a simple drawing of G^* preserving the number of edges in a good drawing of G by introducing at least one of the dummy nodes between



(a) (b)

Figure 4.2: Optimal drawing of a graph with two crossings (a) and an optimum simple drawing of the same graph with three crossings (b). Both drawings were produced with our exact algorithm presented in Section 4.2.

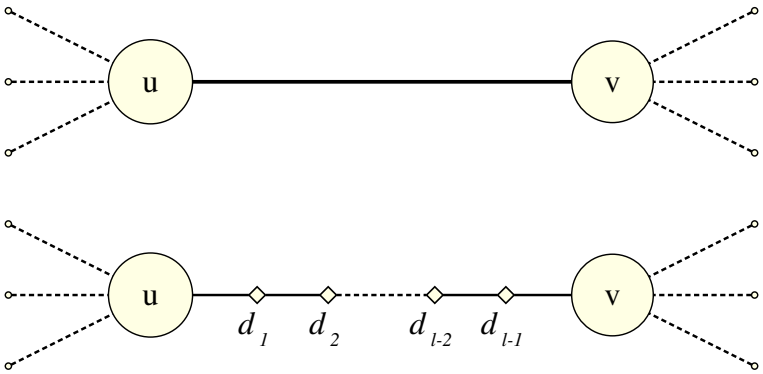


Figure 4.3: Edges are replaced with a path of length l by inserting $l - 1$ dummy nodes.

two consecutive crossings of an edge e. Since the number of dummy nodes is defined to be greater than the number of original edges this is always possible. □

Since an edge $e = (u, v)$ never crosses itself or any adjacent edge it is sufficient to replace e with a path of length $|E| - |\delta(u)| - |\delta(v)| - 1$. Lemma 4.1.3 also shows that it is "sufficient" to solve the Crossing Minimization Problem restricted to simple drawings in order to solve the "general" Crossing Minimization Problem and the *NP*-completeness follows immediately from the proof by Garey and Johnson ([16]).

4.2 An ILP Formulation for simple Drawings

A basic approach to attack the Crossing Minimization Problem using Mathematical Programming for a given graph $G = (V, E)$ is to introduce for each pair of edges $(e, f) \in E^2$ a zero-one decision variable $x_{e,f}$ such that $x_{e,f} = 1$ if and only if e and f crosses in a consistent drawing. For each subdivision of K_5 or $K_{3,3}$ we can add constraints that force at least one of the involved variables to 1.

Mutzel and Jünger point out the problems with this formulation in [34]. To our knowledge there is no known polynomial time separation algorithm to identify these constraints. Moreover those constraints are not strong enough since it is not guaranted that there is a consistent drawing if at least one of the involved crossing variables is one in every Kuratowski subdivision.

Another problem of this formulation is the so-called *Realizability Problem* which is defined as follows.

Definition 4.2.1 (Realizability Problem). *Given a simple graph $G = (V, E)$ and a vector $x \in \{0, 1\}^{\binom{E}{2}}$. Is there a drawing of G such that edges e and f cross if and only if $x_{e,f} = 1$?*

Kratochvíl proved in [27] the *NP*-completeness of the Realizability Problem. To compute a consistent drawing we also need the *order* of the crossings for a particular edge e. This would enable us to insert dummy nodes on each crossing in the given order and use a linear-time planarity testing algorithm to solve the problem.

To avoid the Realizablity Problem we concentrate on simple drawings. Section 4.1 already describes how to solve the general Crossing Minimization Problem if there is an algorithm for crossing minimization restricted to simple drawings.

Given a graph $G = (V, E)$ and a set of unordered pairs of edges $D \subseteq E^2$. 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 we denote with G_D the graph that can be obtained by introducing a dummy node $d_{e,f}$ for each pair of edges $(e,f) \in D$. The edges $e = (u_e, v_e)$ and $f = (u_f, v_f)$ are replaced by the four edges $(u_e, d_{e,f})$, $(d_{e,f}, v_e)$, $(u_f, d_{e,f})$ and $(d_{e,f}, v_f)$. G_D contains a total number of $|V| + |D|$ nodes and $E + 2|D|$ edges.

For a subgraph $H = (V', E') \subseteq G_D$ we denote with $\hat{H} \subseteq E$ the subset of edges in G such that $e = (u, v) \in \hat{H}$ if and only if $e \in E$ and $u \in V'$ or $v \in V'$.

Corollary 4.2.2. *Let D be simple. D is realizable if and only if G_D is planar.*

Using a linear time planarity testing algorithm we can test in time $O(|V| + |D|)$ if D is realizable and compute a consistent drawing if so.

Definition 4.2.3. *For a set of pairs of edges $D \subseteq E^2$ we define*

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

Proposition 4.2.4. *Let D be simple and realizable. For an arbitrary set of pairs of edges $D' \subseteq E^2$ of $G = (V, E)$ and any subdivision H of K_5 or $K_{3,3}$ in $G_{D'}$ the following inequality holds:*

$$C_{D',H} : \sum_{(e,f) \in \hat{H}^2 \setminus D'} x^D_{ef} \geq 1 - \sum_{(e,f) \in \hat{H}^2 \cap D'} (1 - x^D_{ef}) \tag{4.1}$$

Proof. Suppose Inequality 4.1 is violated. Since every $x^D_{e,f} \in \{0,1\}$ the left side of the inequality must be zero and the right hand side must be one which means that

$$x^D_{e,f} = 0 \qquad \forall\ (e,f) \in \hat{H}^2 \setminus D'$$

and

$$x^D_{e,f} = 1 \qquad \forall\ (e,f) \in \hat{H}^2 \cap D'$$

If we only consider the subgraph induced by $\hat{H}$, it follows that $\hat{H}^2 \cap D' = \hat{H}^2 \cap D$ (see Definition 4.2.3). This means that the edges $(e,f) \in \hat{H}^2$ 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. It follows from Kuratowski's theorem (see Theorem 2.1.4) that G_D is not planar. This contradicts the realizability of D (Corollary 4.2.2). □

Theorem 4.2.5. *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 set of inequalities holds:*

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

Proof. It is easy to see that the constraints from the second row are satisfied if and only if D is simple. It remains to show that a simple D is realizable if and only if the conditions $C_{D',H}$ from the last row hold. For a realizable D every $C_{D',H}$ is satisfied according to the proof of Proposition 4.2.4.

We have to show that any set of pairs of edges D that is not realizable violates at least one of the constraints $C_{D',H}$. It follows from Corollary 4.2.2 that G_D is not planar if D is not realizable and we know from Theorem 2.1.4 that there exists a subdivision H of K_5 or $K_{3,3}$ of G_D. Let $D' = D$ and consider the constraint $C_{D,H}$.

$$C_{D,H}: \sum_{(e,f)\in\hat{H}^2\setminus D} x^D_{ef} \geq 1 - \sum_{(e,f)\in\hat{H}^2\cap D} (1 - x^D_{ef}) \tag{4.2}$$

It follows from the definition of x^D that every $x^D_{e,f} \in \hat{H}^2 \setminus D$ is zero, hence the left hand side of Inequality 4.2 is also zero. Since $\hat{H}^2 \cap D \subseteq D$ we also know that $\sum_{(e,f)\in\hat{H}^2\cap D}(1 - x^D_{ef})$ is zero and the right hand side of $C_{D,H}$ is one. □

Since it is easy to compute a corresponding drawing for a simple and realizable D we can reformulate the Crossing Minimization Problem for simple drawings as "*Given a graph $G = (V, E)$. Find a simple realizable subset $D \subseteq E^2$ of minimum cardinality*". This leads to the following *ILP*-Formulation. We use $x(F)$ as an abbreviation for the term $\sum_{(e,f)\in F} x_{e,f}$.

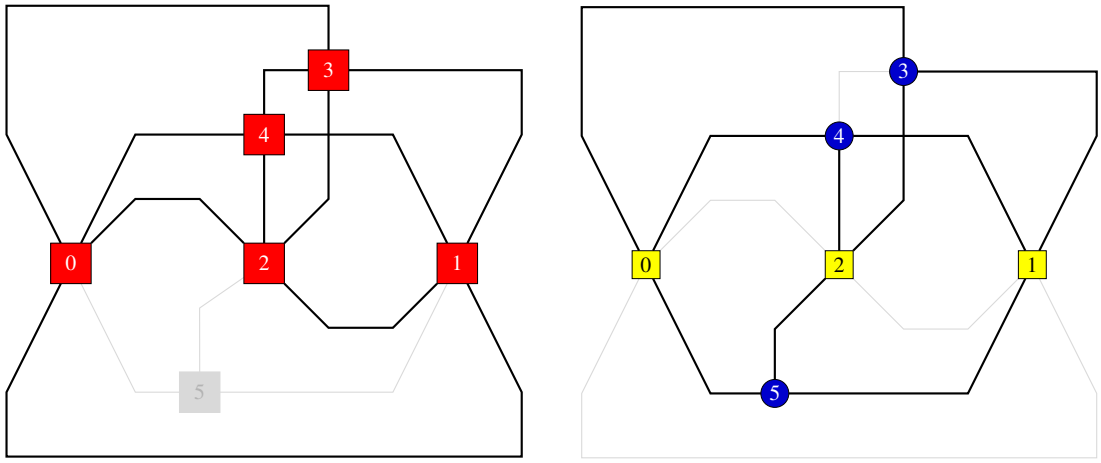
minimize $x(E^2)$

subject to

$\sum_{f\in E} x_{e,f} \leq 1$	$\forall\ e \in E$
$x(\hat{H}^2 \setminus D') - x(\hat{H}^2 \cap D') \geq 1 - \lvert\hat{H}^2 \cap D'\rvert$	for every simple D' and every forbidden subgraph H in $G_{D'}$
$x_{e,f} \in \{0, 1\}$	$\forall\ e, f \in E$

Given a simple set of crossings D we can easily check if D is realizable by applying a planarity testing algorithm to G_D. If the answer is "no" we also get a forbidden subdivision H of G_D and we can separate an additional constraint $C_{D,H}$ according to the proof of Theorem 4.2.5 that excludes D.

It should be noted that the number of Kuratowski subdivisions does not correspond to the number of crossings. Consider the sample drawing in Figure 4.4. The graph can be drawn with only one crossing while it includes a subdivision of K_5 as well as a subdivision of $K_{3,3}$. The figure also shows that we do not necessarily have to separate *all* Kuratowski subdivisions in order to obtain a realizable crossing configuration.



(a) (b)

Figure 4.4: A sample graph drawn drawn with only one crossing. Figure (a) shows a subdivision of K_5 while Figure (b) outlines a subdivision of $K_{3,3}$ in the same drawing.

4.3 Saving Variables

The *ILP* given in Section 4.2 contains a zero-one variable for every pair of edges $(e, f) \in E^2$ that encodes the crossing information between e and f. Edges e and f cross in a consistent drawing if and only if $x_{e,f} = 1$.

Due to the encoding there can only be one crossing between two edges. We show in the following that any optimum drawing satisfies this restriction and how we can save variables to increase the performance of an algorithm implementing the given *ILP*.

4.3.1 Adjacent Edges and Self-Crossings

Lemma 4.3.1. *Any optimum drawing of a graph $G = (V, E)$ satisfies the following conditions:*

(I) no edge crosses itself

(II) adjacent edges do not cross

(III) non-adjacent edges cross at most once

Proof. For the proof of Proposition I suppose there is an optimum drawing of G such that there is an edge $e \in E$ that crosses itself at point p. We can transform the drawing according to Figure 4.5 by deleting the line segment starting and ending at point p

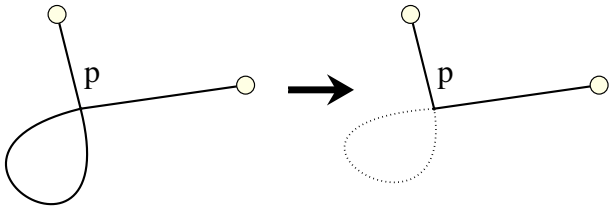


Figure 4.5: Reducing the number of crossings by avoiding self crossings

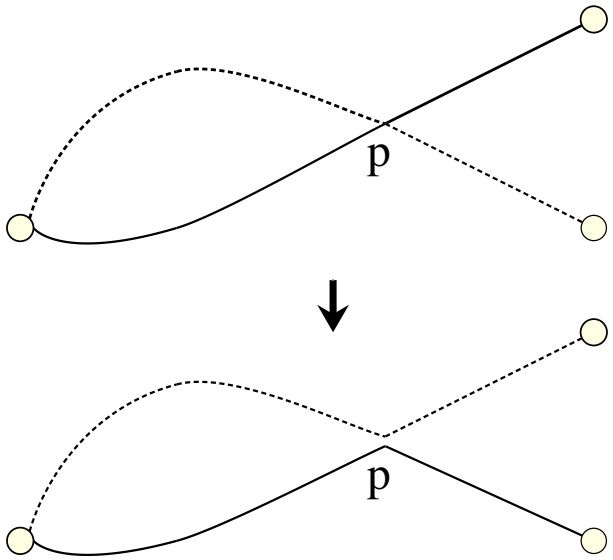


Figure 4.6: Reducing the number of crossings by avoiding crossings of adjacent edges

which immediately reduces the number of crossings by at least one. This contradicts the assumed optimality of the drawing.

To prove that adjacent edges do not cross (Proposition II) let p be the first crossing of edge $e = (u, v) \in E$ and $f = (u, w) \in E$. We can transform the given drawing by replacing the line segment starting at node u and ending at point p of edge e with the corresponding segment of edge f. Moving the line segments in point p slightly apart reduces the number of crossings by exactly one and again contradicts the optimality. Figure 4.6 illustrates this transformation.

It remains to show that non-adjacent edges cross at most once (Proposition III). We can sort the crossings of edge $e \in E$ and edge $f \in E$ in order of their appearance on e. Let p and q be two consecutive crossings between e and f. As in the proof of Proposition II we replace the line segments between p and q and move the two edges slightly apart such that they do not touch anymore (see Figure 4.7). We can repeat this transformation to

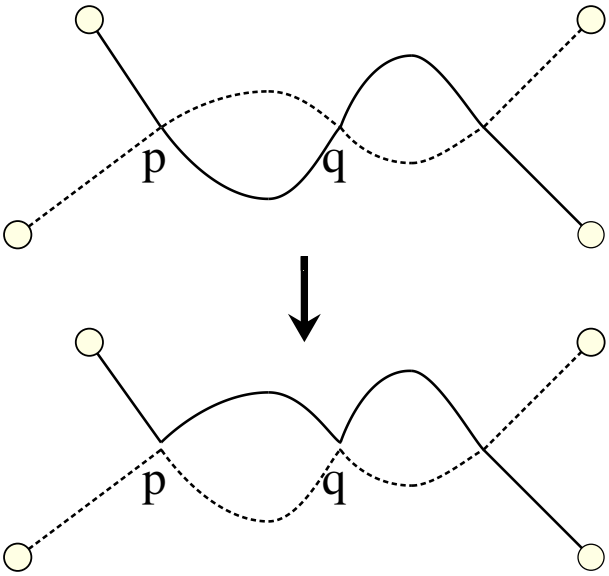


Figure 4.7: Reducing multiple crossings of non-adjacent edges

"remove" any even number of crossings between two non-adjacent edges. □

A drawing that satisfies the conditions of Lemma 4.3.1 is called a *good drawing*. Since the value of any variable $x_{e,f}$ of an optimum solution is zero if $e = f$ or e and f are adjacent edges we can neglect them in the given *ILP*.

4.3.2 Biconnected Components

Consider a connected graph $G = (V, E)$. A graph is called *biconnected* if we have to delete at least two edges in order to disconnect G (see also Section 2.1). A vertex (edge) whose removal disconnects G is called a *separation vertex (edge)*. The following three definitions of a biconnected graph are equivalent.

Definition 4.3.2 (Biconnected Graph). *Given a connected graph $G = (V, E)$. The following three definitions are equivalent:*

I) G contains no separation edges and no separation vertices

II) For any two vertices $u, v \in V$ $u \neq v$ there are at least two disjoint simple paths between u and v.

III) For any two vertices $u, v \in V$ $u \neq v$ there is a simple cycle containing u and v.

A *biconnected component* of G is a maximal biconnected subgraph or a subgraph consisting of a separation edge $e_s = (u, v)$ and its end vertices u and v. Every edge $e \in E$

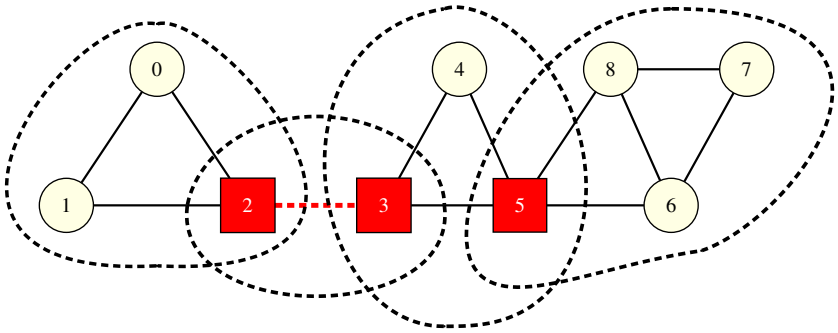


Figure 4.8: A sample graph and its biconnected components

and every nonseparation vertex $v \in V$ belongs to exactly one biconnected component of G while a separation vertex belongs to two or more biconnected components. Figure 4.8 shows a sample graph with its biconnected components. Separation vertices are drawn as rectangles and separation edges as dashed straight lines. Note that any pair of biconnected components has at most one vertex in common and this vertex is a separation vertex.

It is easy to show that the partition of G into its biconnected components forms a *equivalence relation* on the edges of G. Two edges e and f are equivalent (denoted by $e \equiv f$) if and only if they are in the same biconnected component. The symmetry ($e_1 \equiv e_2 \rightarrow e_2 \equiv e_1$) and the reflexivity ($e \equiv e$) are obvious. It remains to show the transitivity:

$$e_1 \equiv e_2 \text{ and } e_2 \equiv e_3 \rightarrow e_1 \equiv e_3$$

Proof. Since $e_1 \equiv e_2$ and $e_2 \equiv e_3$ there are simple cycles C_1 and C_2 such that $e_1, e_2 \in C_1$ and $e_2, e_3 \in C_2$. We need to show that there is a simple cycle that contains both e_1 and e_3. If $e_3 \in C_1$ we are done. Otherwise we can search for the longest path in C_2 that contains e_3 such that only its end points u, v are also part of C_1. The union of this path with the part of C_1 between u and v that contains e_1 forms a simple cycle that contains e_1 as well as e_3. □

We will now present a linear time algorithm to compute the biconnected components based on *Depth-first search* (*DFS*), that goes back to Tarjan (see [45]).

Algorithm 4.3.1 outlines the basic structure of a *DFS*-traversal using a stack to keep track of the order in which edges are processed. S is the set of *visited* nodes. The order $p : V \rightarrow \{1, ..., |V|\}$ in which the nodes are added to S is called *preorder*. Edges that lead to an addition of nodes to S form a tree with root node s (the start node) and are also called *tree edges* while "unused" edges are referred as *back edges*.

We can derive an algorithm to compute the biconnected components from the following observation.

Algorithm 4.3.1: depth_first_search(G, s)

```
1: S = {s}
2: for all e ∈ δ(s) do
3:    push(e) onto stack
4: end for
5: while stack not empty do
6:    pop edge e = (u, v) from stack
7:    if v ∉ S then
8:       S = S ∪ {v}
9:       for all f ∈ δ(v) do
10:         push(f) onto stack
11:      end for
12:   end if
13: end while
```

Lemma 4.3.3. *Let (u, v) and (v, w) be consecutive tree edges of a DFS tree for a graph $G = (V, E)$, $(u, v) \equiv (v, w)$ if and only if there is a back edge from a descendent of w (or w itself) to an ancestor of u (or u itself).*

Proof. Assume there is a back edge (a, b) such that a is an ancestor of u and b is a descendent of w in the *DFS* tree. Then $u \to v \to w \rightsquigarrow b \to a \rightsquigarrow u$ is a simple cycle containing (u, v) and (v, w). Thus the edges (u, v) and (v, w) must be part of the same biconnected component.

On the other hand assume $(u, v) \equiv (v, w)$. We know from Definition 4.3.2 that there must be a simple cycle containing (u, v) and (v, w). If we consider the subgraph of w in the *DFS* tree we can always find an edge (a, b) that is not part of the subtree and leads to an ancestor of v or v itself. Such an edge must exist because the simple cycle "goes back" to u at some point. Since a tree is defined to be a connected graph without cycles the edge (a, b) must be a back edge. □

For every node v we can compute the smallest number $p(u)$ over all nodes u that lie on some simple cycle with v in linear time. We denote this number with $\text{low}(v)$. There is a simple cycle from v to u if we can reach u from v using a (possibly empty) tree path to a descendent of v and a single back edge. We can easily compute $\text{low}(v)$ using the following recursive definition.

Definition 4.3.4 ($\text{low} : V \to \{1, ..., |V|\}$)**.** *For any node v, $\text{low}(v)$ is defined as*

$$\text{low}(v) = \min \begin{cases} \min \text{low}(u) & u \text{ is a child of } v \\ \min p(u) & (v, u) \text{ is a back edge} \\ p(v) & \end{cases}$$

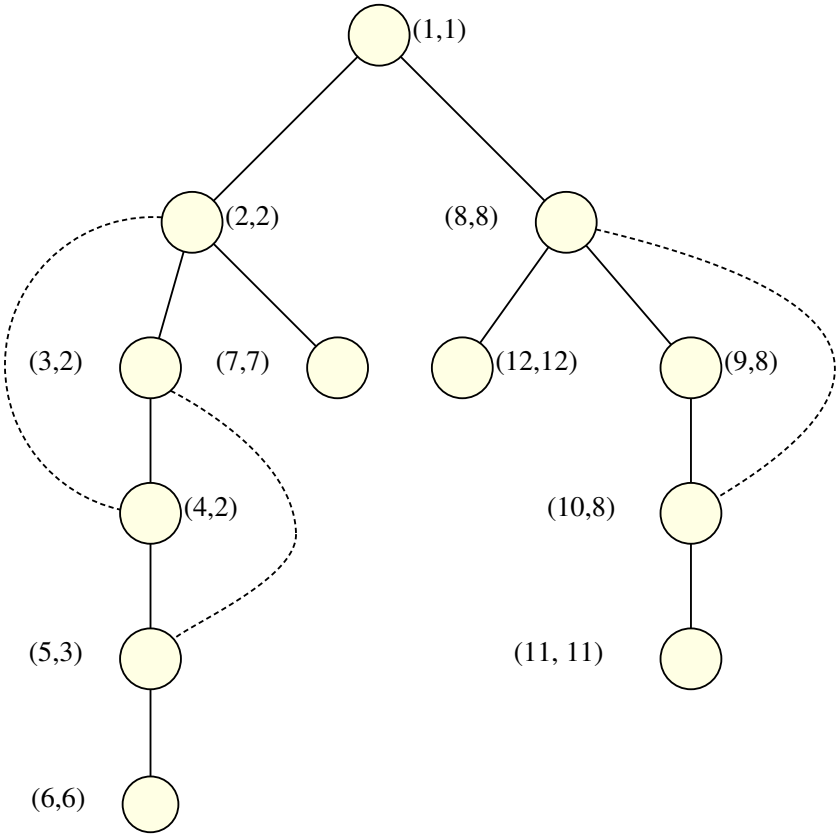


Figure 4.9: A sample graph and the corresponding *DFS* tree. Solid edges denote tree edges while dashed edges are back edges. Every node is labeled with the pair (p, l) denoting the order in the preorder traversal p and its low-number l.

We can easily compute $\text{low}(v)$ for every node v using a preorder traversal of the *DFS* tree. We initialize $\text{low}(v)$ with $p(v)$ for every node v and update the value with the minimum of the current value and $\text{low}(w)$ whenever returning from a child node w. If we discover a backward edge (v, u) we keep the minimum of $\text{low}(v)$ and $p(u)$. This can clearly be done in linear time.

Figure 4.9 shows a sample graph with a *DFS* tree and the corresponding preorder p. Every node v is further labeled with its value $\text{low}(v)$. Tree edges are drawn as solid straight lines and back edges are dashed arcs.

Given the low-numbers we can determine if a vertex v is a separation vertex by checking if one of the following conditions holds.

i) v is the root of the *DFS* tree and is incident to at least two tree edges

ii) v is not the root and there is a child w of v such that $\text{low}(w) \geq p(v)$

Algorithm 4.3.2 outlines in pseudocode how we can modify a recursion based version of

the *DFS* algorithm in order to label the nodes in preorder, compute the low-values and identify the separation nodes in a single pass.

Algorithm 4.3.2: identify_separation_nodes(v)

```
1:  visit[v] = true
2:  low[v] = p[v] = ++time
3:  for all w ∈ δ(v) do
4:    if visit[w] == false then
5:      pred[w] = v
6:      identify_separation_nodes(w)
7:      low[v] = min{low[v], w}
8:      if pred[v] == unset then
9:        {v is a root node}
10:       if v has more than one child then
11:         is_separation_point[v] = true
12:       end if
13:     else if low[w] ≥ p[v] then
14:       is_separation_point[v] = true
15:     end if
16:   else if w ≠ pred[v] then
17:     {(v, w) is a back edge}
18:     low[v] = min{low[v], p(w)}
19:   end if
20: end for
```

To actually separate the set of edges into the biconnected components we can make use of a *stack* to trace back the recursive calls of the algorithm. An edge (u, x) is either processed by a recursive call on vertex x, or (u, x) is identified as a back edge in line 16. Whenever we do so we push that edge onto the stack.

Later, if we identify u as an articulation point, all the edges from the top of the stack down to (u, x) form the edges of one biconnected component, and we pop them from the stack.

It is easy to see that the algorithm runs in time $O(|E|)$.

Many practical graphs are not biconnected and can be divided into different biconnected components. We can save many variables in the *ILP* given in Section 4.2 by employing the following lemma.

Lemma 4.3.5. *In an optimum drawing of a graph $G = (V, E)$, edges belonging to different biconnected components do not cross.*

Proof. Let $C_1, C_2, ..., C_n$ be the biconnected components of a graph $G = (V, E)$. Any two components C_i and C_j $1 \leq i < j \leq n$ have at most one vertex v in common and this vertex is a separation vertex.

Given two optimal drawings of C_i and C_j we can create a drawing for $C_i \cup C_j$ such that $\text{cr}(C_i \cup C_j) = \text{cr}(C_i) + \text{cr}(C_j)$. If C_i and C_j have no node in common we are done. Otherwise let C_i^* respective C_j^* be the graph that can be obtained by introducing a dummy node for each crossing in the usual way. The cyclic ordering of the edges for each vertex defines a combinatorial embedding $\Pi(C_i^*)$ respective $\Pi(C_j^*)$. We can obtain a planar embedding with the same number of crossing by choosing an adjacent face of node v as the outer face. By combining this planar embeddings at node v we obtain a planar embedding for $C_i \cup C_j$ without introducing new crossings.

The same procedure can be repeated to add further biconnected components since the number of common vertices is again at most one. We can use this procedure to merge all biconnected components to obtain a crossing minimal drawing for G such that the number of crossings is the sum of the crossings of its biconnected components.

Now, suppose there is a crossing minimal drawing for G such that two edges e and f belonging to different biconnected components cross. We can obtain a drawing for each component by deleting all vertices (and nodes v with $|\delta(v)| = 0$) that do not belong to this component. Using the construction for a crossing minimal drawing given above we can obtain a drawing for G with a smaller number of crossings. This clearly contradicts the optimality of the original drawing. □

Since every edge belongs to exactly one biconnected component we can remove variables $x_{e,f}$ if $e \in C_i$, $f \in C_j$ and $i \neq j$.

4.4 Preprocessing

Another possibility to improve the runtime of a branch-and-cut algorithm implementing the *ILP* in Section 4.2 is to remove parts of the input graph that do not influence the number of crossings. Those subgraphs are, *e.g.*, trees starting at a single node v or pairs of adjacent edges connected by a node w with degree 2. The aim is to transform a given graph G to a graph G' such that we can obtain a combinatorial embedding for G from a combinatorial embedding for G'.

Figure 4.10 shows some sample situations that can be easily simplified without changing the number of crossings. We describe the particular samples in the following in more detail.

Removing Degree-One Nodes Let $v \in V$ be a node of a connected graph G such that $|\delta(v)| = 1$ and let $\Pi(G')$ denote a combinatorial embedding for a planarized graph $G' = (V \setminus \{v\}, E \setminus \delta(v))$.

Node v is incident to a single edge $e = (v, w)$. We can easily transform the combinatorial embedding $\Pi(G')$ to a combinatorial embedding $\Pi(G)$ by inserting e at an arbitrary position in the counterclockwise ordering of the incident edges of w (see Figure 4.10(a) for an example).

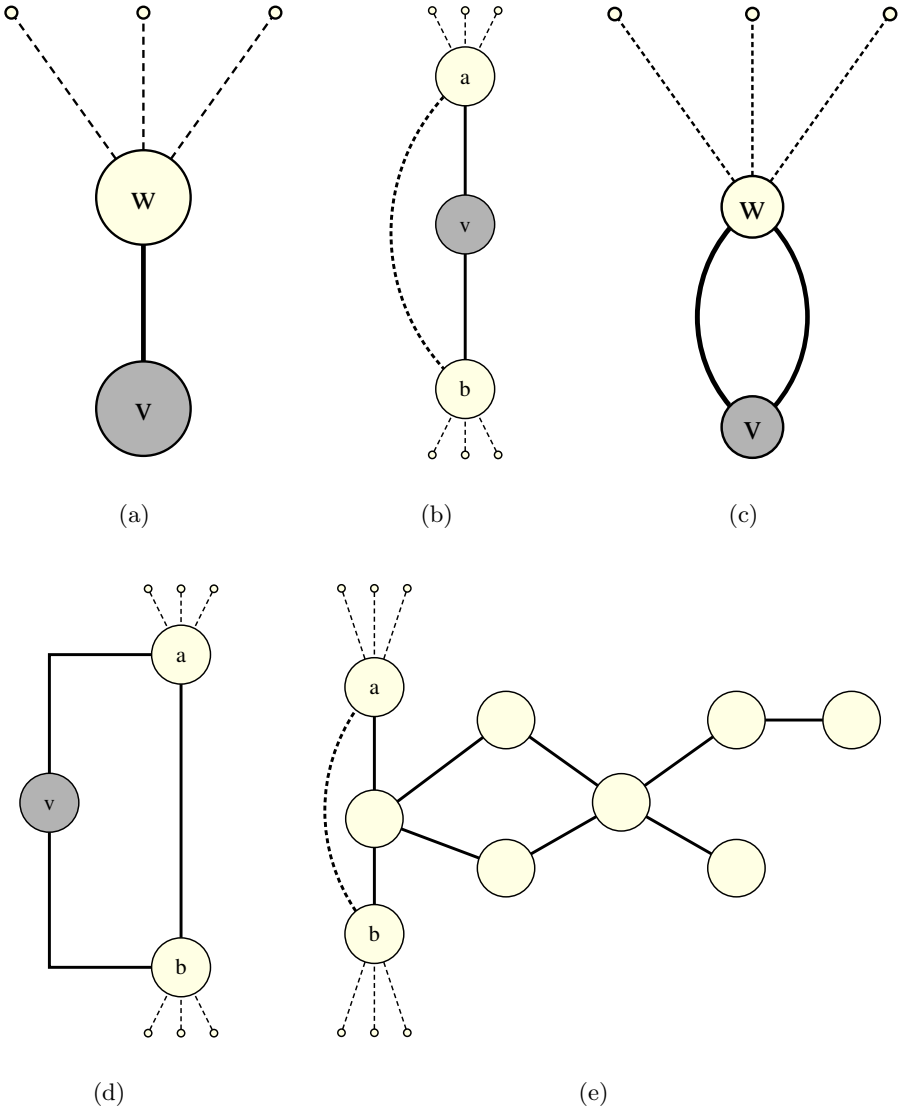


Figure 4.10: Sample graphs that can be simplified using a preprocessing procedure.

Removing Degree-Two Nodes Given a node v with degree two we can use a similar approach. Let (a, v) and (v, b) be the only two incident edges of v and let $G' = (V \setminus \{v\}, E \setminus \{(a, v), (v, b)\} \cup \{(a, b)\})$. We can easily split edge (a, b) in G' by inserting the original vertex v at an arbitrary position (that is not a crossing) on the arc associated with (a, b) in any drawing for G'. This leads to a drawing for G. Figure 4.10(b) outlines this case. The dashed edge represents the dummy edge replacing (a, v) and (v, b).

A special situation occurs if v is connected to a single node w by two edges. Since parallel edges – as well as self loops – do not impact on the number of crossings, we can remove them as in the case of degree-one nodes (Figure 4.10(c)).

Since we do not consider multigraphs in our implementation we must further check if there is already an edge (a, b). If we remove the edges (a, v) and (v, b) in this case, a crossing of (a, b) would "cost" two crossings in a corresponding drawing of G and a drawing for G' cannot be transformed to a drawing for G without increasing the number of crossings (Figure 4.10(d)).

After the removal of each node we have to check again if any of its neighbours can be removed, even it was not possible before. Consider the example in Figure 4.10(e). After repeatedly applying the transformations given above the whole subgraph can be replaced by the edge (a, b).

4.5 Putting it All Together

We will now use the *ILP* given in Section 4.2 to develop a branch-and-cut algorithm. In addition to the basic concepts of this method presented in Section 2.3.2, Algorithm 4.5.1 outlines the general structure of a branch-and-cut algorithm. We will understand this method as a "recipe" and describe the particular "ingredients" in order to solve the Crossing Minimization Problem to provable optimality in this section.

In the case of zero-one integer linear programs, the set of unsolved subproblems L is organized as a binary tree, called the *branch-and-bound tree*. Each subproblem corresponds to a node in the tree and the list of unsolved problems L is represented by its leaves. If we need to split a problem Π into subproblems we choose a fractional *branching variable* and create two new subproblems by setting the branching variable to zero, respectively one.

As in the branch-and-bound approach we store a global upper bound of the best found feasible solution. We can obtain a initial upper bound by applying the heuristic method described in Section 3.4. Another 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.

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

If we find a feasible solution during the branch-and-cut process we can update the

Algorithm 4.5.1: Branch-and-Cut "Recipe"

```
1: L = initial Problem                    {L denotes the list of unsolved problems}
2: repeat
3:    Choose a subproblem Π and let L = L \ Π
4:    repeat
5:       Let x̂ be an optimal solution for the linear relaxation of Π
6:       if x̂ is not feasible for Π then
7:          Separate violated inequalities and add them to the LP
8:       end if
9:    until no more violated inequalities can be found
10:   if  no feasible solution for Π could be found then
11:      Split Π into subproblems and add them to L
12:   end if
13: until L = {}
14: Print the best found feasible solution
```

global upper bound if its value exceeds the objective value of the feasible solution of the subproblem.

Whenever we split a problem into two subproblems by setting the branching variable to zero respective one we can compute a local lower bound. This is the best value for the objective function that can be obtained subject to the assignments of values for the branching variables up to the root node. If this value is greater than the global upper bound we can discard all descendents of the current subproblem since they can never improve the current feasible solution.

Processing Given an input graph G_{in} we first eliminate all degree-one and degree-two nodes to simplify the instance as described in Section 4.4. For each dummy edge replacing the two edges (a, v) and (v, b) we store one of the original edges to be able to reconstruct a drawing for G_{in} from a drawing for the simplified graph. We call the graph that is obtained after the preprocessing procedure $G = (V, E)$.

The next step is to compute a first heuristic solution H that serves two purposes:

- The number of crossings in H can be used as a initial global upper bound for the number of crossings and we can easily compute a first feasible solution vector for the *ILP*.
- As described in Section 4.1 we replace every edge by a path of length l since our *ILP* formulation allows at most one crossing per edge. Clearly the parameter l has a strong influence on the number of variables and should be chosen as tight as possible. Let h denote the number of crossings in H. Any optimal solution contains at most h crossings per edge and thus we can use h as an upper bound for the edge transformation.

To continue with the notation of Section 4.1 we call the graph after transforming each edge by a path of length l $G^* = (V^*, E^*)$. We call nodes in $V^* \setminus V$ and edges in $E^* \setminus E$ *dummy nodes (edges)*.

During the preprocessing phase we eliminate all degree-two nodes except those that are adjacent to nodes u and v which are already connected by an edge $e = (u, v)$ (see Figure 4.10(d) for an example). Furthermore every dummy node $v \in V^* \setminus V$ has degree two and is incident to exactly two dummy edges $e \in E^* \setminus E$. It is easy to show that edges belonging to a common path P such that each node alongside P has degree two do not cross. The proof is according to the proof of Proposition I of Lemma 4.3.1 if we remove the dummy nodes and reinsert them after computing an optimal drawing.

We can easily identify these paths using the procedure given in Algorithm 4.5.2. We compute a function path : $E \to \{1, ..., n\}$ such that path(e) = path(f) if and only if e and f belong to a common path alongside degree-two nodes.

Algorithm 4.5.2: path_decomposition$(G = (V, E))$

```
1:  for all e ∈ E do
2:      path[e] = −1                                   {Initialization}
3:  end for
4:  n = 0
5:  for all e = (u, v) ∈ E do
6:      if path[e] == −1 then
7:          for all w ∈ {u, v} do
8:              while |δ(w)| == 2 do
9:                  Let f = (w, w′) be the incident edge of w such that path[f] == −1
10:                 path[f] = n
11:                 w = w′
12:             end while
13:         end for
14:         path[e] = n++
15:     end if
16: end for
```

Furthermore we compute the biconnected components $C_1, ..., C_c$ for G^* as described in Section 4.3.2. Every edge $e \in E^*$ belongs to exactly one biconnected component.

Initial ILP We create a zero-one variable $x_{e,f}$ for every unordered pair of edges $(e, f) \in E^* \times E^*$ that satisfies the following conditions

- $e \neq f$
- path$(e) \neq$ path(f)
- e and f belong to the same biconnected component

- e and f are non-adjacent edges

The objective function is defined to be

$$\text{minimize} \sum_{\forall x_{e,f}} x_{e,f}$$

We furthermore create starting constraints that ensure at most one crossing per edge.

$$\sum_{f \in E^*} x_{e,f} \leq 1 \qquad \forall\ e \in E^*$$

Given a lower bound l for the number of crossings in G we can create a special constraint that forces at least l variables to be one.

$$\sum_{\forall x_{e,f}} x_{e,f} \geq l$$

As already described in Section 3.3 we can obtain a general lower bound by the number of nodes n of a graph G (see Theorem 3.3.2).

$$\text{cr}(G) \geq \frac{1}{33.75} \frac{n^3}{n^2} - 0.9n$$

Unfortunately this bound is not very useful in practice since the term often becomes negative. A better bound can be obtained from the skewness of the graph G. Although the computation of the skewness of a graph is known to be *NP* complete (see [30]), there are exact approaches that are able to solve medium-sized instances of the problem in reasonable time. We therefore used a branch-and-cut algorithm based on *ABACUS* that is able to solve practical instances within a few minutes (see [24]).

Feasibility Test Let x be an integral solution vector. We can test in linear time if x represents a feasible solution in the following way. Let $D = \{(e, f) \mid x_{e,f} = 1\}$ be the set of pairs of edges that cross each other. As described in Section 4.2 we can compute the graph G^*_D by inserting a dummy node for each $(e, f) \in D$. We can use one of the linear time planarity test algorithms mentioned in Section 2.1 to test if G^*_D contains a subdivision of K_5 or $K_{3,3}$ and compute a corresponding planar embedding if not.

Separation of Inequalities Suppose the fact that the answer of the planarity test is "no", we also get a Kuratowski subdivision $H \subseteq E^*$. In this case it is easy to separate an additional constraint $C_{D,H}$ that excludes the current infeasible solution (see the proof of Theorem 4.2.5). In many cases it is possible to separate additional constraints by removing a random edge $e \in H$ from G^* and searching for further Kuratowski Subdivisions.

The main problem is that the solution vector x for the linear relaxation of the *ILP* contains fractional values that cannot be used directly to compute a set $D \subseteq E \times E$. We

have to round the current fractional solution in order to decide if we introduce a dummy node for (e, f). Therefore we experimented with different strategies and compared their performance against each other. We present those results in Chapter 5.

We can not guarantee that there is no violated inequality if G_D^* is planar. In this case we have to select a branching variable and split the current problem Π into subproblems Π_0 and Π_1 by setting the branching variable to zero, respectively one. Due to this fact we "only" present a heuristic solution for the Separation Problem.

5 Computational Results

This chapter presents the computational results of our algorithm on a widely used benchmark set descrived from real world data. All experiments were done on an an Intel P4 2.4 GHz with 512KB of cache memory and 1 GB of main memory using LINUX 2.4.

In Section 5.1 we give some implementation details of our algorithm and shortly describe the used libraries. Furthermore we give an overview of *AGD* (Algorithms for Graph Drawing) and show how we can integrate our algorithm into this software library in order to combine it with different layout algorithms.

The rest of this Chapter describes the used benchmark set and presents the results obtained from an extensive computational study. We close this work by discussing those results and further improvements that are not part of this work in Chapter 6

5.1 Implementation Details

The algorithm presented in Chapter 4 has been implemented using *C++* and the class library *LEDA*. We used the commercial optimization library *CPLEX* in order to solve the Integer Linear Program and implemented the branch-and-cut strategy using its built in features. The whole module has been integrated into *AGD*, a library of Algorithms for Graph Drawing. We now shortly describe those libraries and sketch out the design of *AGD*. Further we describe how our new algorithm fits into the design of this library and how it can be combined with existing modules to compute layouts with a minimum number of crossings.

LEDA is an abbreviation for Library of Efficient Data Structures and Algorithms. It is a *C++* library that offers combinatorial and geometric data types and a number of highly optimized basic algorithms. While the project has been started by Kurt Mehlhorn at the "Max-Plank-Institut für Informatik" it is now maintained by Algorithmic Solutions Software GmbH (`http://www.algorithmic-solutions.de`).

The package includes excellent data types for graphs and a number of graph algorithms that were used in our implementation. Due to the close relationship between *LEDA* and *AGD* it was easy to use those implementations, *e.g.*, for the computation of biconnected components and linear time planarity testing.

CPLEX Many *NP*-hard optimization problems could be attacked in the last years using Linear Programming in conjunction with the branch-and-cut approach. Due to the high flexibility of mathematical programming this approach became very popular and is often

used to solve practical (combinatorial) optimization problems. The need for practical implementations led to a large number of highly optimized *LP* solvers.

Our implementation is based on *CPLEX* (version 8), which is a commercial optimization tool that is available from *ILOG* (`http://www.ilog.com`). It offers interfaces to a large number of programming languages and can be extended using so called callbacks to develop a branch-and-cut style algorithm.

AGD As already mentioned, *AGD* is a powerful library that offers a broad range of existing algorithms for two-dimensional graph drawing. It originated from a DFG-funded project in 1996 and is now developed in a cooperations of groups of the Technical University of Vienna, Cologne and the Max Planck Institute for Computer Science.

One of the most notable design feature is the representation of algorithms as objects that provide a common method for calling the algorithm. All Algorithms that implement the same functionality belong to a common base class and can easily be exchanged by each other. Algorithms of the same "*type*" implement a common call interface, which allows the development of generic algorithms that do not need to be aware of the details of an exchangeable module.

In addition, each algorithm can define *pre-* and *postconditions* that are maintained by the framework. An instance of an algorithm together with this conditions is called a *module*. Those conditions can either be properties of the input graph, *e.g.*, planar or biconnected, or properties of the produced layout (*i.e.*, orthogonal or straight line).

Figure 5.1 is taken from the *AGD* user manual and shows an overview of the available modules. Some of the modules make use of other modules to perform particular subtasks. Those modules can be exchanged at runtime by different modules of the same type. Arrows in Figure 5.1 indicate the module options for each module and the required type.

The class `LayoutModule` is a common base class for drawing algorithms. It defines a common interface to those algorithms. Every layout algorithm maps the nodes to points in the plane and edges are represented by a a list of bend points with a source and target anchor point. A specialization of this class is the module `GridLayoutModule`, which serves as a common base class for layout algorithms that place nodes at integer points.

A widespread technique to draw nonplanar graphs is to replace crossings by artificial vertices. Those graphs can be drawn using algorithms for planar graphs and the temporary vertices are removed afterwards. We already described this technique in Section 3.4. *AGD* therefore provides the class `PlanarizationLayout`. A `PlanarizerModule` is applied in order to transform a given graph into a planar representation which is drawn afterwards. Both, the planarizer module and the planar layout algorithms are exchangeable modules.

In order to match the design of *AGD*, our new algorithm is implemented as a *PlanarizerModule* and can be used as a replacement to the planarization heuristic, which is

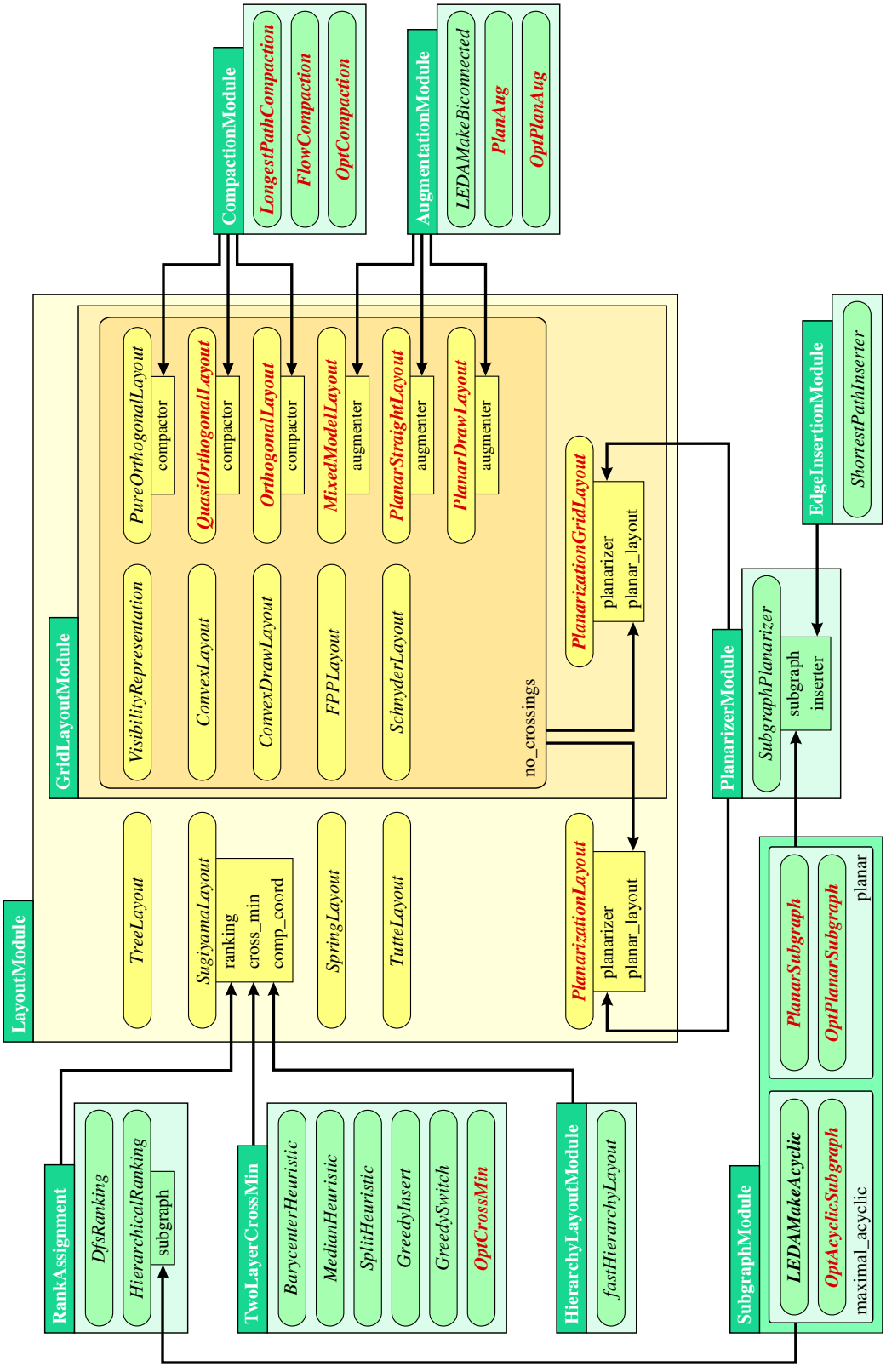
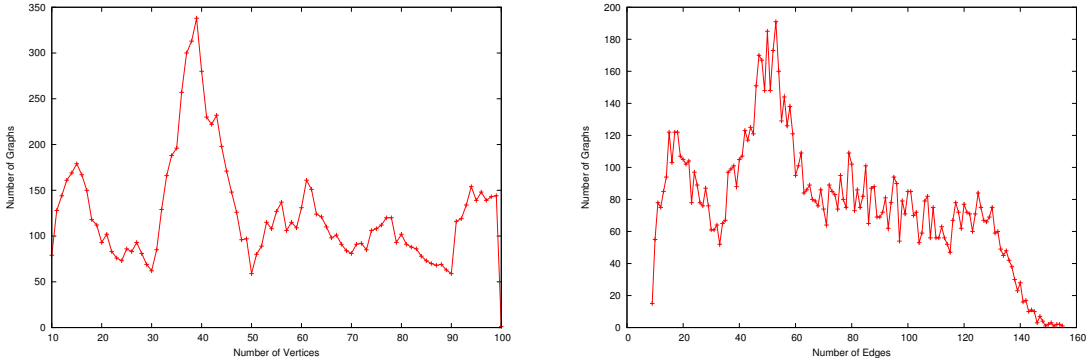


Figure 5.1: Overview of the classes in *AGD*. The figure is taken from the *AGD* user manual



(a) (b)

Figure 5.2: Number of graphs included in the benchmark set sorted by (a) number of nodes and (b) number of edges

implemented in `SubgraphPlanarizer`. The result of our algorithm is a planar embedding $\Pi(G')$ while G' is the graph that is obtained by introducing a dummy node at each crossing point in G. This enables us to use any of the existing planar layout algorithms to produce a drawing for G with $\text{cr}(G)$ crossings. Most of the examples in this thesis are computed using the `MixedModelLayout` module.

5.2 The Benchmark Suite

In order to test the performance of our new algorithm and compare its solution quality to existing heuristics, we used a benchmark set of graphs of the University of Rome III, used by Di Battista et al. in [3].

The set contains 11389 graphs that consist of 10 to 100 vertices and 9 to 158 edges. Those graphs were generated from a core set of 112 "real life" graphs used in database design and software engineering applications. Di Battista et al. developed a software tool that emulates typical operations on graphs in practice. Generated graphs were tested for suitability according to objective and subjective criteria (see [3]). Figure 5.2 shows the number of graphs included in the benchmark set that contain a certain number of edges respective nodes. A number of 3280 graphs are already planar. Figure 5.3 shows the number of planar graphs sorted by the number of vertices. As expected their number decreases with increasing size of the graphs.

Most of the graphs are sparse, which is a usual property in most application areas in automatic graph drawing. The average ratio between the number of edges and the number of nodes of the graphs from the benchmark set is about 1.35. The minimum, maximum and average ratio depending on the number of nodes is given in Figure 5.4.

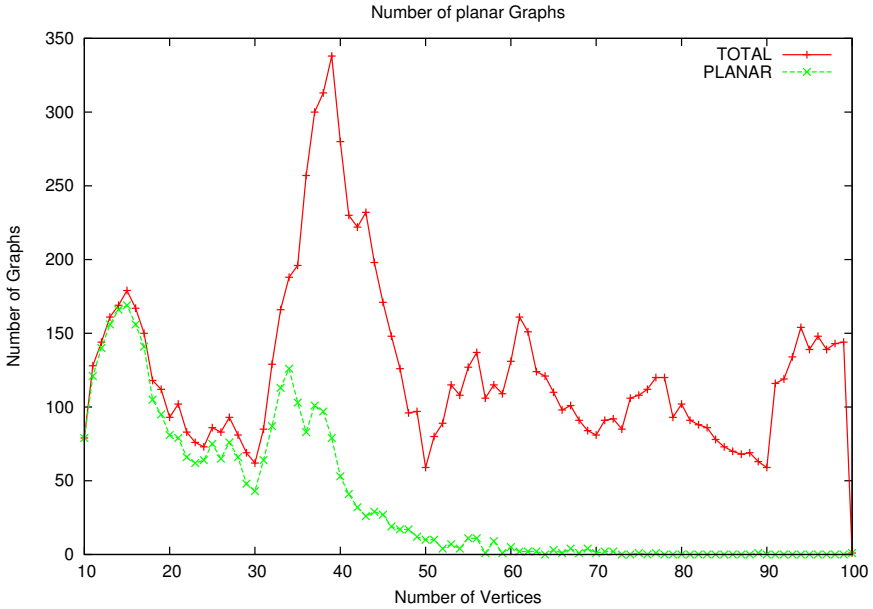


Figure 5.3: Total number of graphs and number of planar graphs sorted by the number of nodes

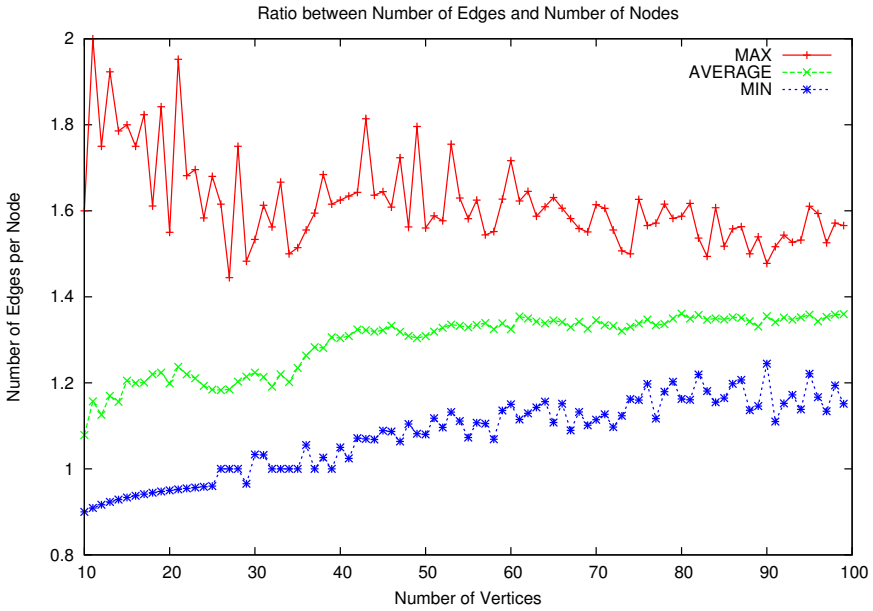


Figure 5.4: Minimum, maximum and average number of edges per node sorted by the number of nodes

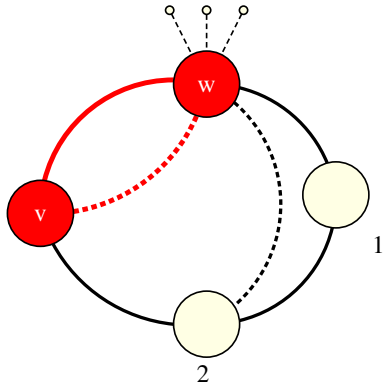


Figure 5.5: Consecutive removal of degree-two nodes in the preprocessing procedure leads to multiple edges between two nodes v and w.

5.3 The Effects of Preprocessing

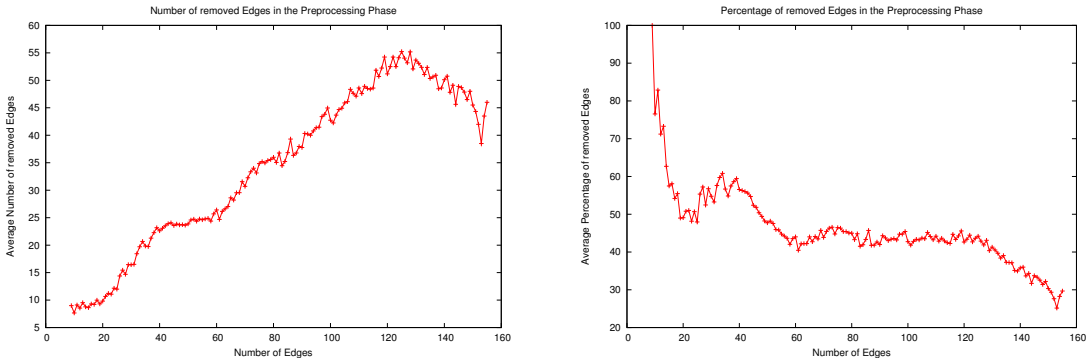
We implemented the preprocessing procedure described in Section 4.4 and applied it to the input graphs of our benchmark set. Since the size of the graphs is relatively small, the runtime of the algorithm is only about some milliseconds and can be neglected.

Whenever we remove a node of degree-one or two, the resulting graph is also reduced by exactly one edge. The only exception occurs if there is a node v of degree two that is connected to a different node w by two edges (v, w). None of the input graphs is a multigraph, but those situations can be the result of previous preprocessing steps. Consider the sample in Figure 5.5. After each of the degree-two nodes is replaced by a dummy edge in the denoted order, we get a circle of length two that can be removed without influencing the crossing number as in the case of degree-one nodes. Those situations only occurred in 131 of the 11389 graphs, thus we only consider the number of removed edges in Figure 5.6. For very small instances we could remove all of the edges and the average percentage over the whole benchmark set is 47.83%, which corresponds to an average of 30.6 removed edges. This leads to a major improvement of the runtime of our branch-and-cut algorithm. Due to the effectiveness and simplicity of the preprocessing procedure it is also suitable for other algorithms for the Crossing Minimization Problem. Figure 5.7 shows the average number of edges with and without preprocessing sorted by the number of nodes.

5.4 Determining Biconnecting Components

We described a linear time algorithm based on *DFS* to compute the biconnected components of a graph in detail in Section 4.3.2. Edges belonging to different biconnected components do not cross, which allows us to neglect a large number of variables in our *ILP* formulation if the graph is not already biconnected.

We used an implementation contained in *LEDA* and present the results for our bench-



(a) (b)

Figure 5.6: Absolute number of reduced edges (a) and the percentage of reduced edges (b) during the preprocessing phase.

mark set in this section. The runtime of the algorithm can be neglected since the time needed to read the input files and write the output almost exceeds the runtime of only some milliseconds.

Figure 5.8(a) shows the minimum, average and maximum number of biconnected components sorted by the number of nodes in our benchmark set. The number of biconnected components is surprisingly high. Figure 5.8(b) shows the number of graphs with a certain number of biconnected components. The average number of biconnected components over the whole benchmark set is about 14.37 and there are only 58 graphs that are already biconnected.

In Figure 5.9 we consider the number of nodes per biconnected component. While the average number of edges per component is relatively small, the maximum number of edges grows nearly linear with the number of edges. This shows that a bulk of our instances consists of one biconnected component covering nearly all the edges and a relatively high number of very small components.

5.5 Computing the Skewness

In order to compute tight lower bounds for the Crossing Number of an input graph G, we used an exact branch-and-cut algorithm for the maximum planar subgraph problem to compute the skewness of G. We defined the skewness in Section 3.3 to be the minimum number of edges that must be deleted from G in order to obtain a planar subgraph G_P. Although the maximum planar subgraph problem was shown to be *NP* hard (see [30]), we could successfully solve instances of medium size using the algorithm proposed by Jünger and Mutzel ([24]) and Mutzel ([33]).

The algorithm is based on *ABACUS* (A Branch-And CUt System), which is a frame-

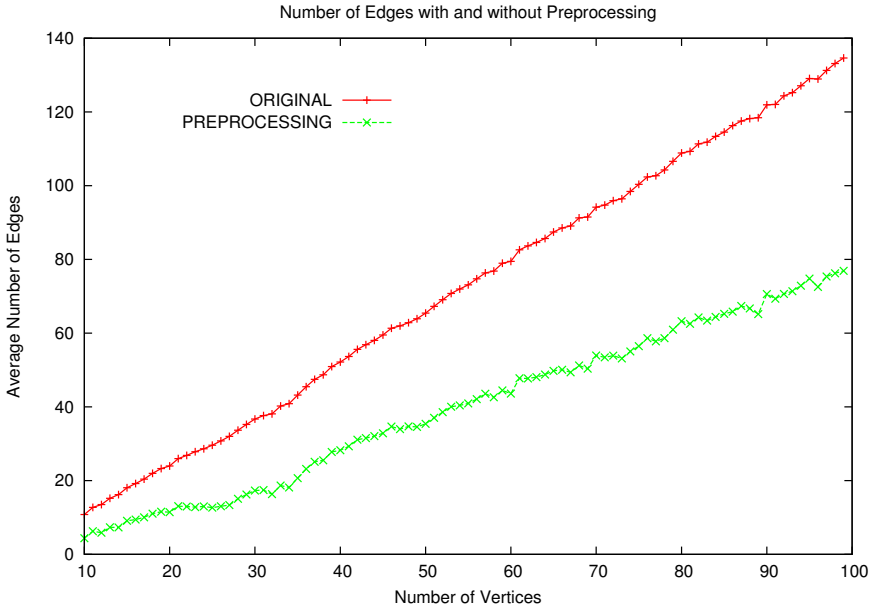


Figure 5.7: Average number of edges with and without preprocessing sorted by the number of nodes

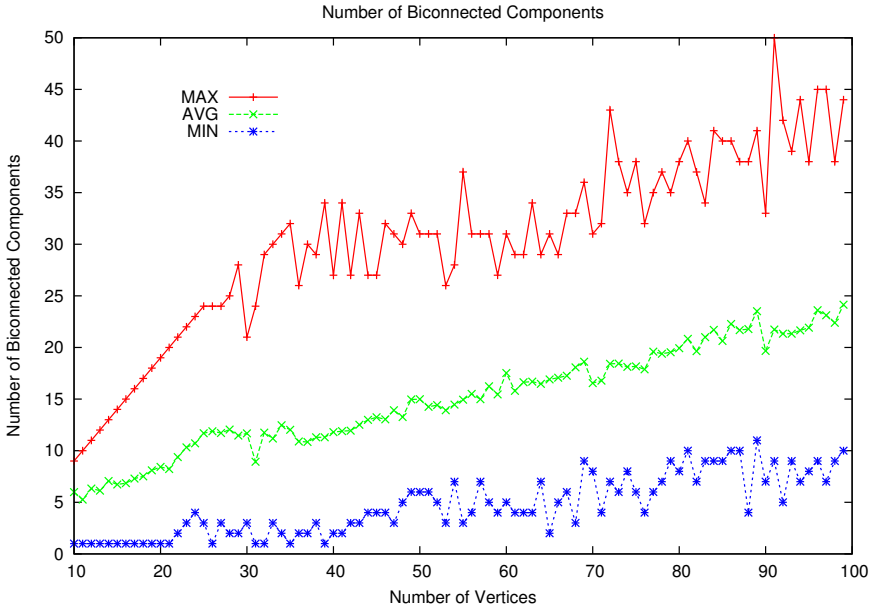
work for the implementation of branch-and-cut algorithms. Details can be found in [25]. There is already an implementation of this algorithm in *AGD*, which was used to perform the computational results. The corresponding module is named `OptPlanarSubgraph`.

Due to the complexity of the Crossing Minimization Problem we only considered instances up to 40 nodes. Figure 5.10(a) shows the maximal, minimal and average skewness of graphs from our benchmark set up to 40 nodes. The computation time strongly depends on the number of edges that have to be removed in order to obtain a maximal planar subgraph. While instances with skewness up to four take only some milliseconds, the computation of more complex graphs takes some minutes. Therefore we show the number of graphs sorted by their skewness in Figure 5.10(b) and give the average computation time in Figure 5.10(c). Since the skewness of planar graphs is always zero we only consider nonplanar instances.

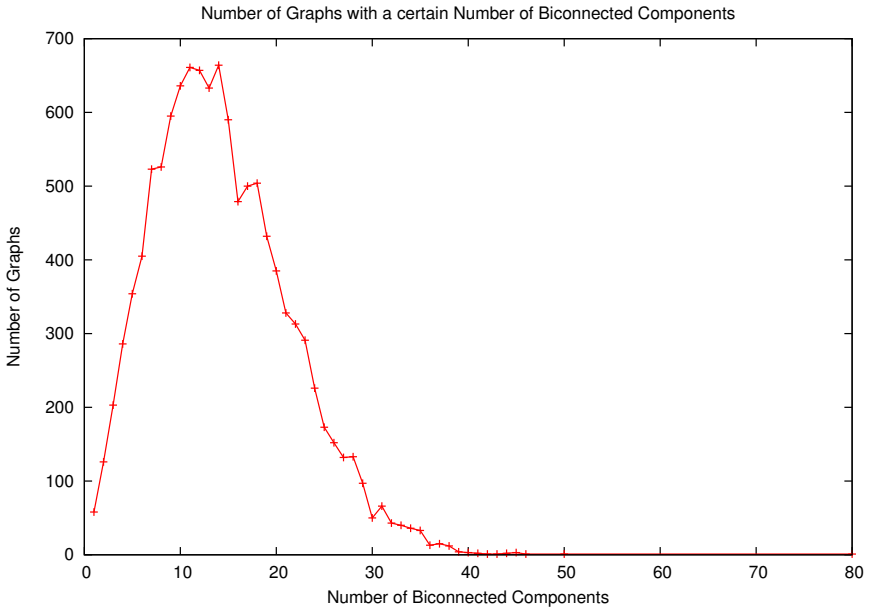
5.6 Results of our exact Approach

We now present the results of our exact algorithm for the Crossing Minimization Problem.

We need to round the current fractional solution to integer values in order to separate violated inequalities. Therefore we experimented with different strategies and compared their performance against each other.



(a)



(b)

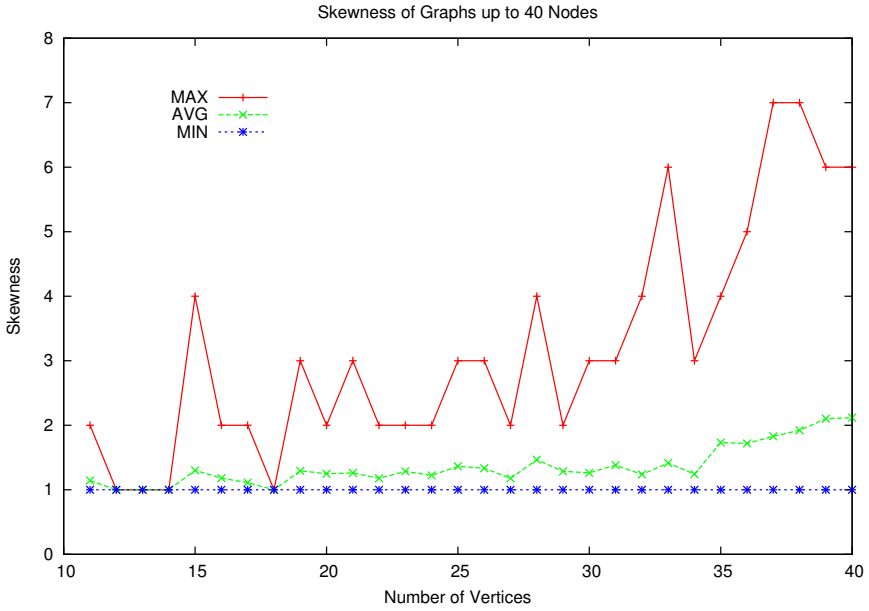
Figure 5.8: Number of biconnected components sorted by the number of nodes (a) and the number of graphs with a certain number of biconnected components (b)

Figure 5.9: Average and maximum number of edges per biconnected component sorted by the number of edges

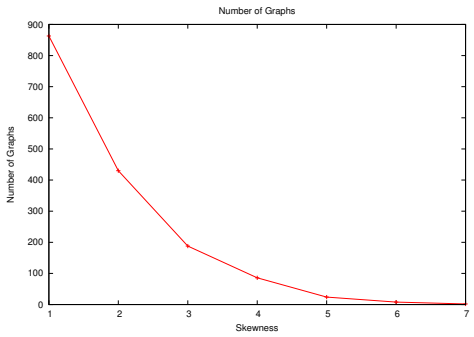
- *R1* We round every value that is greater than $1 - \varepsilon$ to one. All other variables are mapped to zero.
- *R05* Every variable with a value greater or equal than 0.5 is rounded to one.
- *R0208* If the value of a variable is less than 0.2 or greater than 0.8 it is mapped to zero respective one. In the interval $[0.2,\ 0.8]$ we flip a coin with equal probability between zero and one.

Once we obtained an integer solution vector, we start to look for a Kuratowski subdivision H and add the constraint $C_{D,H}$ to the linear relaxation if it is violated by the fractional solution vector. As a heuristic we remove one of the edges from H and try to find further forbidden subdivisions in order to separate more than one constraint in each cut phase.

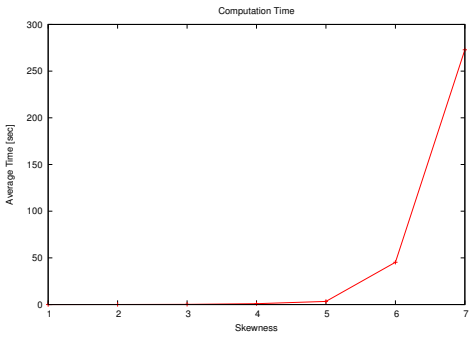
We considered graphs up to 40 nodes and compared the number of solved instances of the different strategies against each other. All computations were done with and without transforming the graph by replacing the edges by a path of length u, while u denotes an upper bound for the crossing number of the particular graph. In the latter case we can not guarantee to find the crossing number $\mathrm{cr}(G)$ of a graph G since any optimum drawing can contain at least one edge that crosses more than one other edge. We denote the minimum number of crossings in any simple drawing of G as in Chapter 4 with $\mathrm{crs}(G)$. This constraint strongly reduces the number of required variables and enables



(a)



(b)



(c)

Figure 5.10: Average skewness (a) for graphs up to 40 nodes. Figure (b) shows the number of graphs with a certain skewness and Figure (c) shows the corresponding average computation time.

us to solve more instances in the same amount of time. There are only 24 graphs of all nonplanar instances up to 40 nodes in our benchmark set, such that $\mathrm{crs}(G) > \mathrm{cr}(G)$ and the difference is at most one. The average number of variables in our *ILP* sorted by the number of edges is shown in Figure 5.11. *SMON* denotes the number of variables with enabled edge transformation and *SMOFF* shows their number for the computation of simple drawings. Note the logarithmic scale for the average number of variables.

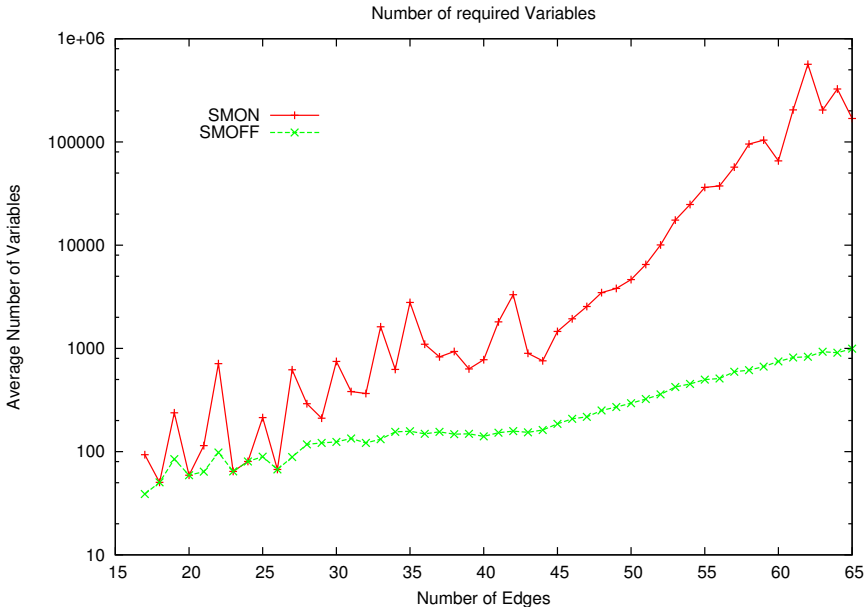
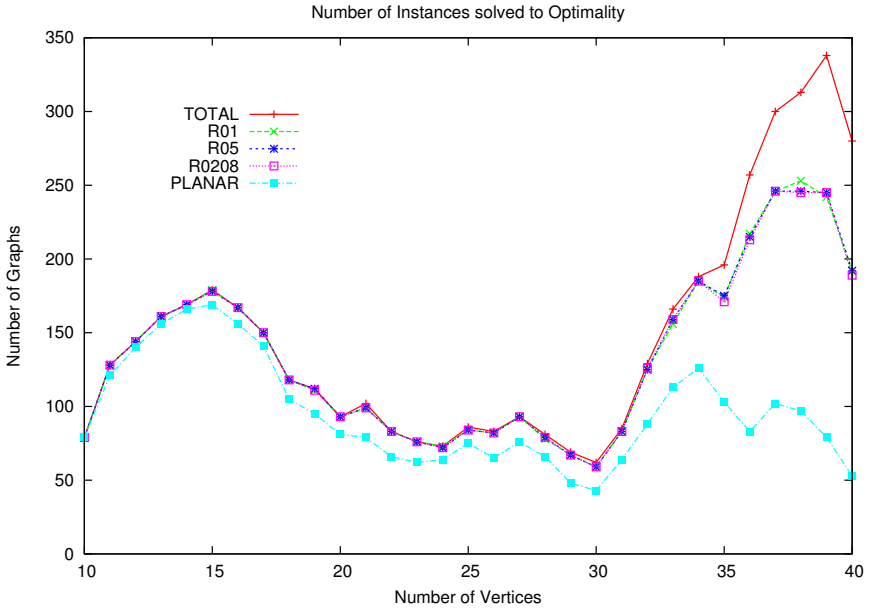
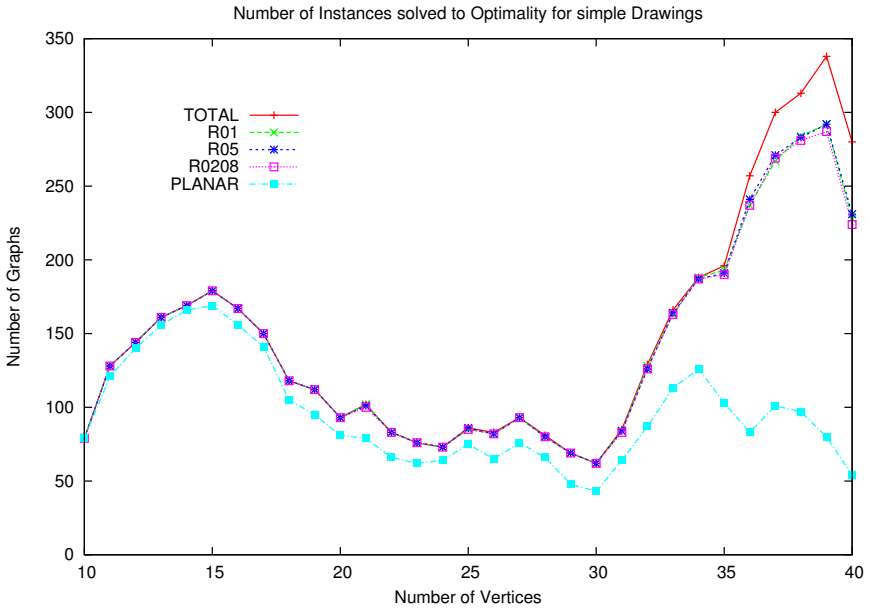


Figure 5.11: Number of required Variables in our *ILP* with and without edge transformation

Number of solved Instances Figure 5.12 shows the number of graphs that could be solved within a time limit of 5 minutes on an Intel Pentium 4 with 2.4 GHz and 1 GB of main memory. As a baseline we further included the number of planar graphs and the total number of graphs with a certain number of nodes. As expected, the difference between our implementation for $\mathrm{cr}(G)$ and $\mathrm{crs}(G)$ grows with the size of the graphs. Another interesting aspect is that the particular strategy to round the fractional values does not seem to have great impact on the solution quality. Figure 5.13 shows the percentage of solved instances sorted by the number of nodes. The smaller amount of variables without using the edge decomposition leads to a significantly higher number of instances that could be solved within the time limit. While the percentage of solved graphs goes down to about 65% for the general crossing number for instances with 40 nodes, we can still solve about 80% of those instances without supporting multiple edge crossings.

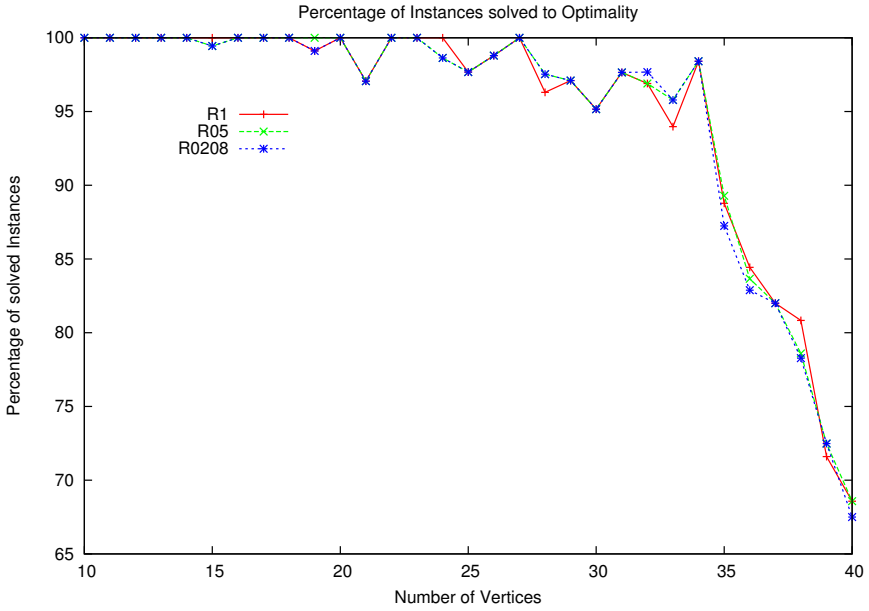


(a)

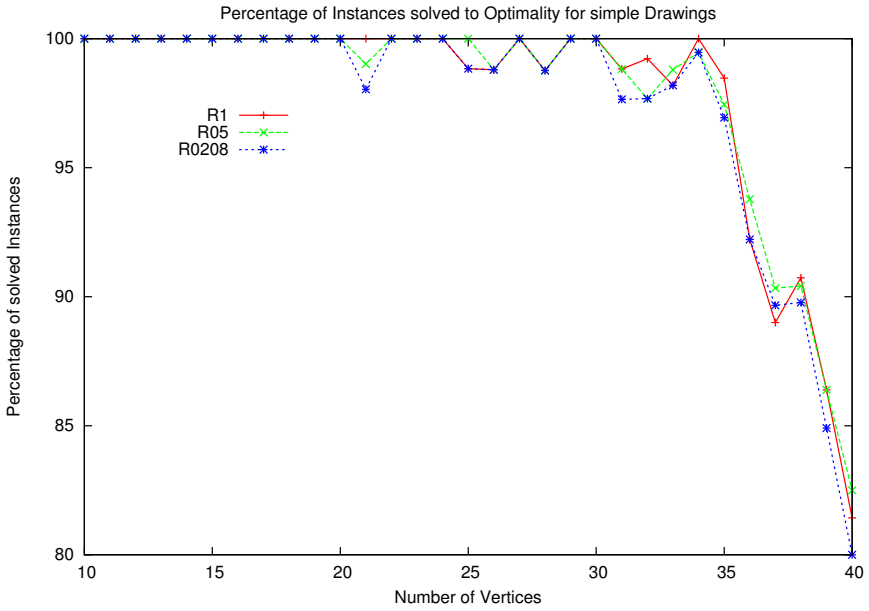


(b)

Figure 5.12: Number of graphs solved by our exact algorithm for graphs up to 40 nodes with (a) and without (b) supporting multiple crossings per edge



(a)



(b)

Figure 5.13: Percentage of graphs solved by our exact algorithm for graphs up to 40 nodes with (a) and without (b) supporting multiple crossings per edge

Average Computation Time Figure 5.14 shows the average computation time for instances that could be solved within 5 minutes by all of the considered strategies to round the fractional solution. While there was nearly no notable difference in respect to the number of solved instances between the different strategies, the comparison of the runtime shows a surprising results. There neither is a clear winner nor a clear loser in our study. While each of the strategies performs well on a certain range of graphs, there are also examples where the same strategy leads to a larger computation time.

As for the maximum planar subgraph problem, the required time to solve a particular instance strongly depends on its crossing number. To illustrate this fact we plot the average computation time of graphs with a particular crossing number in Figure 5.15 with and without supporting multiple edge crossings. Again we only considered instances that could be solved by all strategies within 5 minutes.

Comparison with heuristic Results Clearly we are interested in the quality of our results in comparison to heuristic approaches. For the computation of heuristic values we used the planarization approach described in Section 3.4. The algorithm is already implemented in *AGD* (the module is named `SubgraphPlanarizer`) and serves furthermore for the computation of upper bounds for our *ILP*.

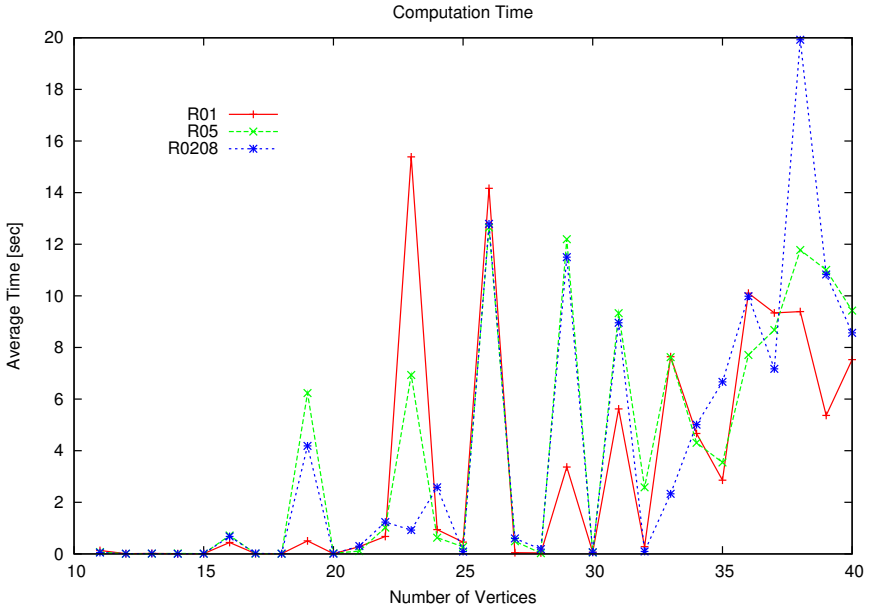
Gutwenger and Mutzel present in [19] an extensive computational study of crossing minimization heuristics. The authors investigate the effects of various methods for the computation of a maximal planar subgraph and different edge re-insertion strategies for the planarization approach. Furthermore they study the impact of post processing heuristics.

In addition to the iterative algorithm for the computation of a maximal planar subgraph described in Section 3.4 (*MAXIMAL*), the authors also use an algorithm based on *PQ*-trees that is suggested in [23]. The first step of this algorithm is to compute a *st*-numbering. The particular choice for s and t has an impact on the solution quality and the authors study the effects of randomly choosing an edge $e = (s, t) \in E$ for up to 100 calls. This method is denoted by *PQ1*, *PQ10*, *PQ50* and *PQ100* for 1, 10, 50 and 100 iterations.

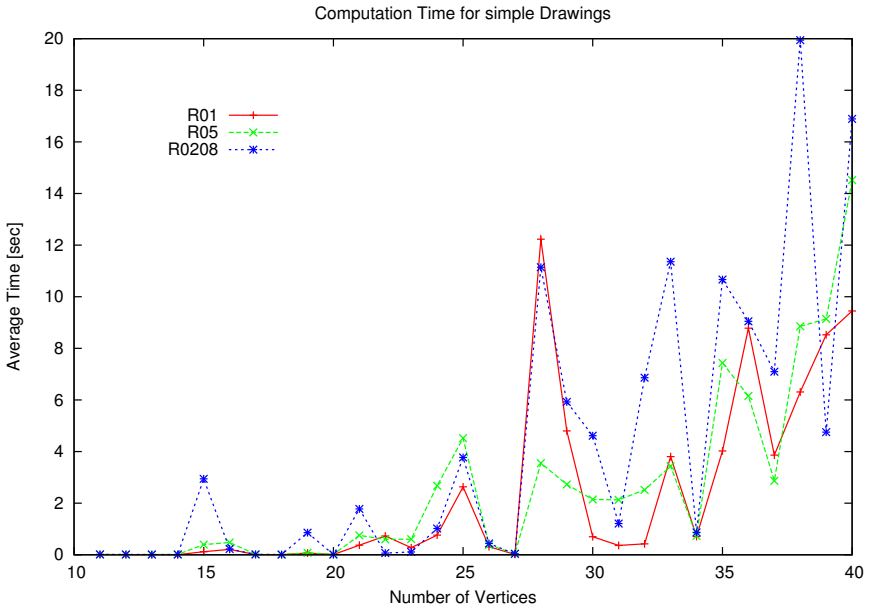
The edge re-insertion step is done for a fixed embedding (*FIX*) and over the set of all possible planar embeddings (*VAR*). As a post processing procedure the authors iteratively delete an edge and re-insert it into the planar embedding. Therefore they consider different selection strategies

- *INS*: Only edges that are not part of the maximal planar subgraph are considered for the re-insertion procedure.
- *ALL*: The whole set of edges is used for the post processing procedure
- *MOST* x%: The authors use only x percent of the total set of edges for the post processing method. The selected edges are chosen according to the number of crossings they are involved in.

Furthermore the authors propose a permutation variant that repeats the whole procedure for a different initial ordering of the edges that are not part of the maximal planar

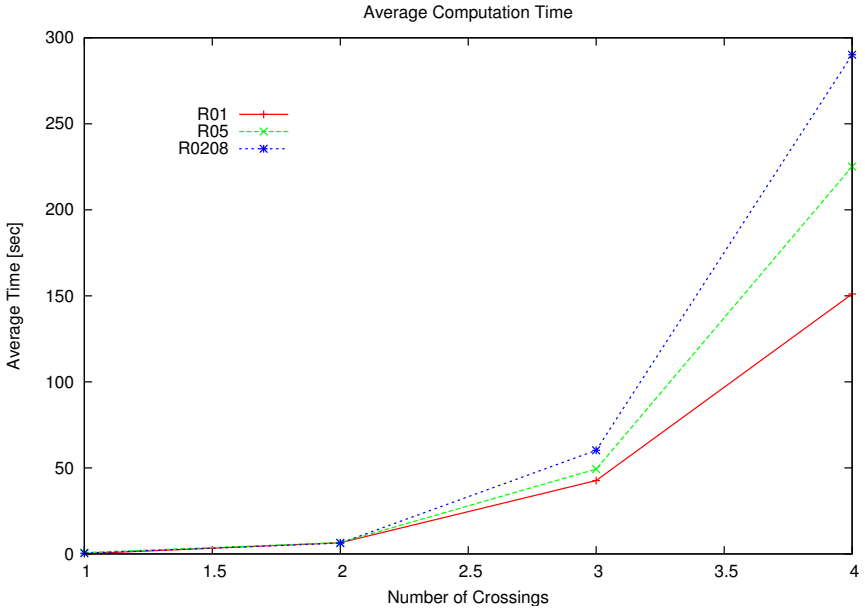


(a)

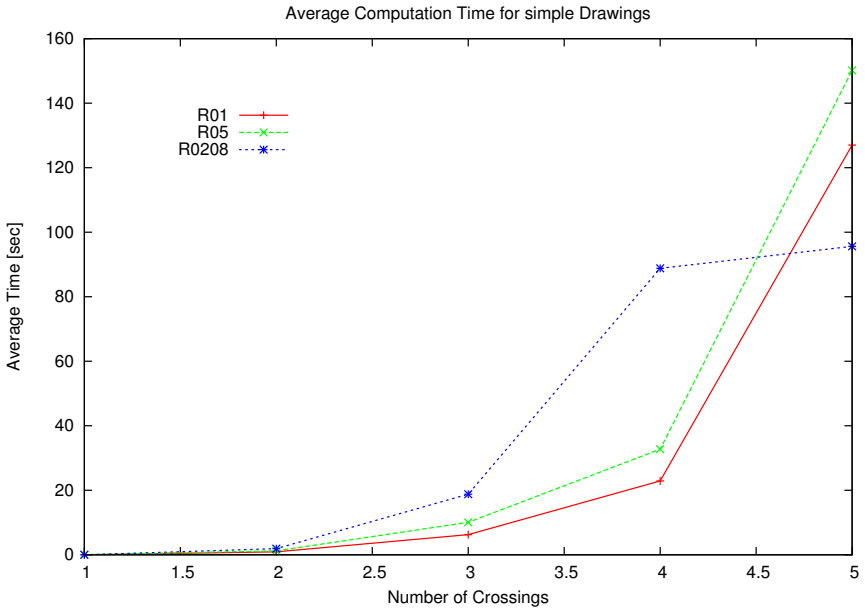


(b)

Figure 5.14: Computation time for graphs up to 40 nodes with (a) and without (b) supporting multiple crossings per edge



(a)



(b)

Figure 5.15: Average computation time for graphs G sorted by $\mathrm{cr}(G)$ (a) respective $\mathrm{crs}(G)$ (b)

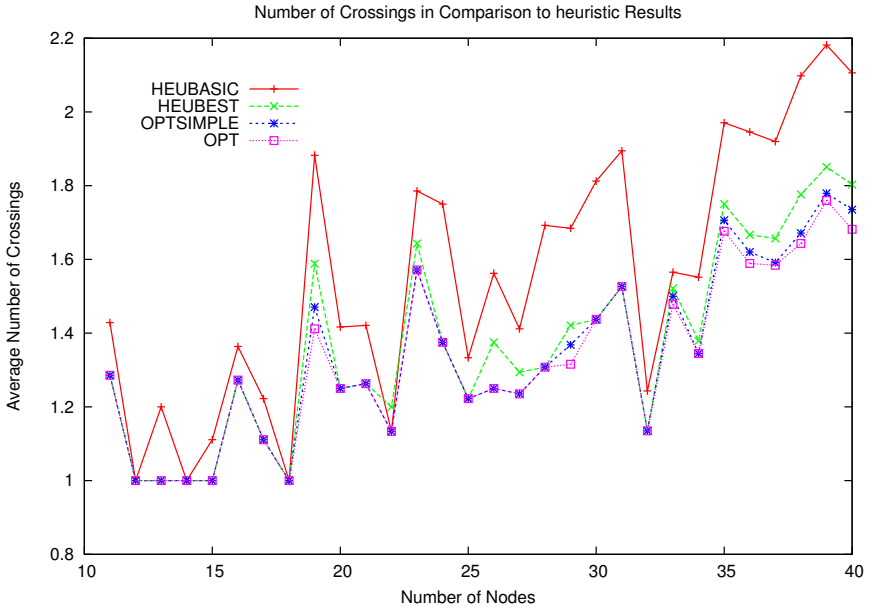


Figure 5.16: Comparison between heuristic results and the crossing numbers computed with our exact algorithm.

subgraph. This method is denoted by *PERM1*, *PERM2*, *PERM10* and *PERM20* for 1, 2, 10 and 20 repetition rounds.

It points out that one of the best strategies is the combination of *PQ100* together with *VAR* for the edge re-insertion and *ALL* for the post processing step. Furthermore *PERM20* is the best choice for the permutation procedure. We denote this combination by *HEUBEST*.

Figure 5.16 shows the average number of crossings sorted by the number of nodes for the basic planarization approach (*HEUBASIC*) and the improved version *HEUBEST*. Furthermore we compare this results to those of our exact approach with (*OPT*) and without (*OPTSIMPLE*) supporting multiple edge crossings. To highlight the improvements in more detail we only considered graphs that could be solved to optimality within the time limit.

We can clearly improve the heuristic results for the basic approach, even for the relatively small instances considered in our computational study. Also upon the best known heuristic methods we achieve a notable improvement for some larger instances. The average improvement over the whole considered benchmark set is about 15.8% for the basic heuristic and 4.7% for the best known strategy. Figure 5.17 shows the average improvement in percent of the exact algorithm in respect to the heuristic values.

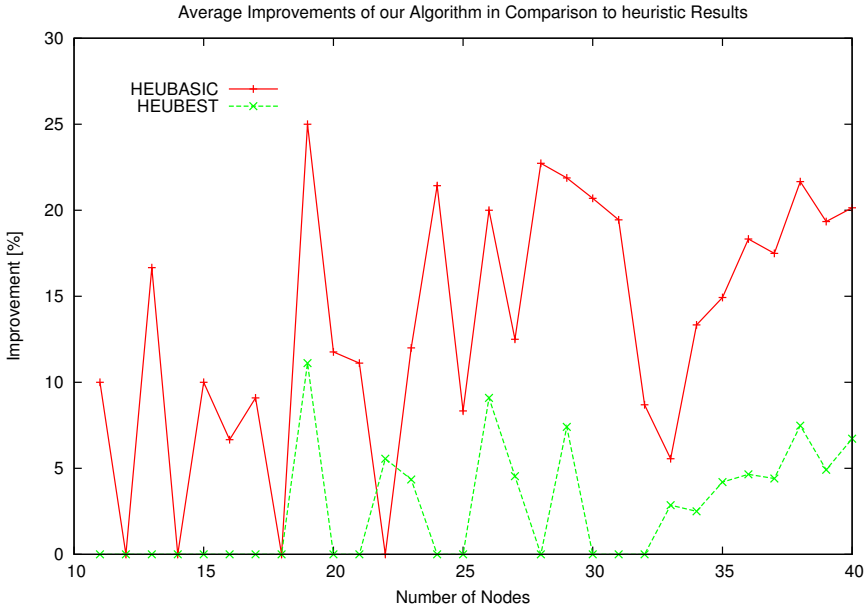


Figure 5.17: Relative Improvement of the exact algorithm in respect to the heuristic solutions.

Further Statistics To close this chapter we present interesting parameters of the branch-and-cut process for some "hard" instances in more detail. Therefore we select some typically graphs that could be solved by our algorithm but took at least one minute of computation time. All those computations were done with enabled edge decomposition and the round strategy *R05*. Table 5.1 lists a selection of those instances sorted by the total runtime t_{total}. The columns have the following meaning.

- *Instance* Filename of the particular instance G
- t_{total} Total runtime of our algorithm without the time for in- and output and the computation of the skewness
- t_{sep} Total runtime of the separation procedure
- *Variables* Number of variables in the *ILP*
- *Satisfied* Number of constraints $C_{D,H}$ that were computed for the rounded solution vector of the *LP* relaxation but that were already satisfied by the current solution.
- *Treesize* Number of nodes in the Branch-and-Cut tree
- cr(G) Crossing Number of G

Instance	t_{total}	t_{sep}	Variables	Cuts	Satisfied	Treesize	cr(G)
grafo11376.38	60.3	32.8	4338	501	974	334	3
grafo1368.31	68.0	35.4	4088	645	1473	400	3
grafo7347.40	72.2	56.4	10557	490	333	74	2
grafo11644.38	76.7	39.5	4338	513	1253	437	3
grafo1711.33	83.6	44.5	3523	1274	2786	1251	3
grafo6003.37	91.7	76.6	14494	288	319	26	3
grafo9852.38	92.7	49.1	4338	664	1536	573	3
grafo9999.39	101.7	53.4	4344	379	1715	514	3
grafo10401.38	105.8	61.7	4338	452	2149	609	3
grafo6984.39	117.1	50.1	6401	247	1463	185	4
grafo11661.38	120.6	58.8	4338	854	1720	673	3
grafo6049.36	134.0	46.9	2019	1984	2575	1189	2
grafo3813.39	161.8	88.7	4708	1095	2682	775	3
grafo5525.40	189.9	102.6	10495	1202	2295	687	3
grafo5857.38	192.0	106.1	5540	720	1601	353	3
grafo10342.37	198.2	63.0	2846	1586	2791	970	3
grafo5536.38	200.6	135.3	7848	576	1403	224	3
grafo3954.37	212.4	115.4	7319	1155	1764	871	3
grafo10770.35	218.3	135.0	5948	1012	2584	419	3
grafo10888.35	252.5	149.4	5948	1223	2670	431	3
grafo3281.34	270.0	143.6	7768	1570	5483	1066	3
grafo10087.39	299.4	181.8	7332	846	2817	710	3
grafo3225.37	313.8	178.6	10492	1309	1864	513	3
grafo7111.40	333.4	206.7	5116	864	6537	1807	3
grafo3019.39	374.9	269.3	12265	1128	2691	525	3
grafo11240.36	395.1	144.2	3969	2161	3406	1322	3
grafo3974.37	430.1	259.0	6798	2185	8457	1645	3
grafo1456.29	435.8	288.9	13479	1771	1755	388	3
grafo6455.39	453.7	261.1	5975	331	6896	1286	3
grafo11284.37	468.9	263.1	7345	1547	2738	646	3
grafo5657.33	497.9	287.2	4337	1211	7307	2291	3
grafo3646.37	549.8	414.5	8674	267	6008	1083	3
grafo2878.19	558.9	432.2	10026	923	6298	1203	3
grafo7222.40	591.9	372.9	11539	413	3866	520	3
grafo3722.40	597.3	385.8	17442	530	2261	240	3
grafo2742.39	624.7	372.6	5950	1453	6427	1514	3
grafo5763.39	1154.9	943.0	14754	454	5493	741	3
grafo7475.40	1337.9	999.6	10795	572	6436	991	3
grafo6593.38	2355.6	864.0	4354	3785	17226	4504	3

Table 5.1: Branch-and-Cut parameters for some instances that could be solved in more than 100 seconds.

The average percentage of the required time for the separation procedure in respect to the total runtime t_{total} is about 60%. This is quite a lot and can be explained by the large number of constraints that are computed but not added to the current *ILP* since they are already satisfied by the last solution vector. While the average number of added cuts in our selection is about 1030, we rejected 3590 computed constraints on average. The average number of subproblems that were computed in the branch-and-cut tree is about 870.

6 Discussion

Despite the complexity of the Crossing Minimization Problem, the results obtained so far are promising. Our implementation is able to solve sparse instances up to about 30 nodes within a few seconds to provable optimality on average hardware. Even for such small graphs we can improve upon a widley used heuristic by 15% and upon the best known heuristic strategies by about 5% on average. We can expect to achieve even better results on larger graphs.

Unfortunately the required computation time strongly depends on the number of crossings and quickly exceeds practical limits for larger graphs. However, there is still much room for further improvements.

We can try to find further or stronger constraints by the investigation of the polytope associated to the underlying combinatorial optimization problem. We can represent those polytopes either as a convex hull of the extreme points or by a system of linear equations and inequalities. There is a software package called *PORTA* – available from the University of Heidelberg (`www.uni-heidelberg.de`) – that is able to enumerate all integral solution vectors for a given set of inequalities and transform this representation back to a system of inequalities.

First experiments with this software led to promising results, but even the computation of the smallest interesting examples blew the available computation time. However, this approach can be useful for the search for stronger constraints.

Another problem that prevents the use in practical applications until now is the strongly increasing number of variables in our *ILP* formulation. We can try to work around this by the use of column generation. A brief overview of this approach is given in Section 2.3.1. However, until now it is not clear how we efficiently insert new variables without violating the existing constraints.

A straight-forward way to improve the runtime of our algorithm is probably the combination with good heuristics during the Branch-and-Cut process. We can strongly decrease the required computation time by finding tight upper bounds early in the optimization process. This allows us to discard whole subtrees whose lower bound exceeds our global limit, and explore more promising branches early. Important improvements may also be achieved by better strategies for the node selection phase.

Even when we are not able to find an optimum solution within a certain time limit, we can try to use the fractional values in order to compute good heuristic solutions. Variables with a relatively large value may indicate good candidates for a crossing in a

corresponding drawing.

Bibliography

[1] M. Ajtai, V. Chvátal, M.M. Newborn, and E. Szemerédi. Crossing-free subgraphs. *Annals of Descrete Mathematics*, 12:9–12, 1982.

[2] L. Auslander and S. Parter. On embedding graphs in the sphere. *Journal of Mathematics and Mechanics*, 12(3):517–523, 1961.

[3] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7(5-6):303–325, 1997.

[4] G. Di Battista and R. Tamassia. Incremental planarity testing. In *30th annual Symposium on Foundations of Computer Science, October 30–November 1, 1989, Research Triangle Park, North Carolina*, pages 436–441. IEEE Computer Society Press, 1989.

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

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

[7] H. Bodlaender and A. Grigoriev. Algorithms for graphs embeddable with few crossings per edge. Research Memoranda 036, Maastricht : METEOR, Maastricht Research School of Economics of Technology and Organization, 2004. available at http://ideas.repec.org/p/dgr/umamet/2004036.html.

[8] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976.

[9] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, 1983.

[10] R.J. Cimikowski. Graph planarization and skewness. *Congressus Numerantium*, 88:21–32, 1992.

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

[12] H.N. Djidjev. A linear algorithm for the maximal planar subgraph problem, 1995.

[13] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[14] P. Erdös and R.K. Guy. Crossing number problems. *The American Mathematical Monthly*, 80:52–58, 1973.

[15] M. Garey and D. Johnson. Computers and interactability: A guide to the theory of NP-completeness, 1979.

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

[17] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Society*, 64:275–278, 1958.

[18] 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. Society for Industrial and Applied Mathematics, 2001.

[19] C. Guwenger and P. Mutzel. An experimental study of crossing minimization heuristics. In *Graph Drawing*, volume 2912 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2004.

[20] R.K. Guy. Crossing numbers of graphs. In *Graph Theory and Applications (Proceedings)*, Lecture Notes in Mathematics, pages 111–124. Springer, 1972.

[21] R.K. Guy, T.A. Jenkyns, and J.Schaer. The toroidal crossing number of the complete graph. *Journal of Combinatorial Theory*, 4:376–390, 1968.

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

[23] R. Jayakumar, K. Thulasiramans, and M.N.S. Swamy. On $O(n^2)$ algorithms for graph planarization. *IEEE Transactions on Computer Aided Design*, 8:257–267, 1989.

[24] M. Jünger and P. Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16(1):33–59, 1996.

[25] M. Jünger and S. Thienel. Introduction to ABACUS—a branch-and-cut system. *Operations Research Letters*, 22:83–95, 1998.

[26] G. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Research Report MPI-I-98-1-013, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, May 1998.

[27] J. Kratochvíl. String graphs. II.: Recognizing string graphs is NP-hard. *J. Comb. Theory Ser. B*, 52(1):67–78, 1991.

[28] F.T. Leighton. *Complexity issues in VLSI: optimal layouts for the shuffle-exchange graph and other networks.* MIT Press, 1983.

[29] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs: International Symposium*, pages 215–232, 1967.

[30] P. Liu and R. Geldmacher. On the deletion of nonplanar edges of a graph. In *Proceedings of the 10th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 727–738, Boca Raton, FL, 1977.

[31] M. Jünger M. Grötschel and G. Reinelt. A cutting plane approach for the linear ordering problem. *Operations Research*, 32:1195–1220, 1984.

[32] S. Masuda, K. Nakajima, T. Kashiwabara, and T. Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Trans. Comput.*, 39(1):124–127, 1990.

[33] P. Mutzel. *The Maximum Planar Subgraph Problem.* PhD thesis, Universität zu Köln, 1994.

[34] Petra Mutzel and Michael Jünger. Graph drawing: Exact optimization helps! In M. Grötschel, editor, *The Sharpest Cut*, Series on Optimization. MPS - SIAM, 2001. Festschrift zum 60. Geburtstag von Manfred Padberg.

[35] T.A.J. Nicholson. Permutation procedure for minimising the number of crossings in a network. *IEE Proceedings*, 115:21–26, 1968.

[36] T. Nishizeki and N. Chiba. Planar graphs: Theory and applications, 1988.

[37] J. Pach and G. Tóth. Graphs drawn with few crossings per edge. *Combinatorica*, 17(3):427–439, 1997.

[38] J. Pach and G. Tóth. Which crossing number is it anyway? *J. Comb. Theory Ser. B*, 80(2):225–246, 2000.

[39] J.A. La 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. ACM Press, 1994.

[40] 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. Springer-Verlag, 1997.

[41] C. Roos and T. Terlaky. Advances in linear optimization, 1997.

[42] 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. Society for Industrial and Applied Mathematics, 1990.

[43] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.

[44] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

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

[46] P. Turán. A note of welcome. *Journal of Graph Theory*, 1:7–9, 1977.

[47] W.T. Tutte. How to draw a graph. *Proc. London Mathematical Society*, 13:743–768, 1963.

[48] K. Wagner. Bemerkungen zum vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 46:26–32, 1936.

[49] 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.

[50] T. Ziegler. *Crossing Minimization in Automatic Graph Drawing*. PhD thesis, Max-Planck-Instiut für Informatik, 2000.