Karlsruhe Institute of Technology

Rectilinear Crossing Minimization

Master Thesis of

Klara Reichard

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Dr. Ignaz Rutter
	Marcel Radermacher, M.Sc.

Time Period: 1st May 2016 – 30th November 2016

KIT – University of the State of Baden-Wuerttemberg and National Laboratory of the Helmholtz Association

www.kit.edu

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 25th August 2017

Abstract

This thesis deals with the rectilinear crossing minimization problem, which is NP-hard [BD93]. More precisely, we propose a heuristic for computing a straight-line drawing of a general graph G which realizes a small rectilinear crossing number. Inspired by Gutwenger et al. [GMW05], we pursue an approach which extracts a planar subgraph that includes as many edges of G as possible and iteratively reinserts the missing edges into G. After each edge insertion nodes are moved in order to reduce the number of crossings in the current drawing. We fix the position of all nodes but one node v and move v to a position, that minimizes the number of crossings in the drawing. This position can be computed in an $\mathcal{O}((m \cdot d_{max})^2)$ time-bound, with d_{max} denoting the maximum degree of a node in G. We evaluate several configurations of this algorithm, which use different strategies to avoid local optima. We compare these configurations to the spring embedder of Fruchterman and Reingold [FR91] on a variety of different graph classes and observe that each of our configurations yields drawings with a significantly lower rectilinear crossing number than the commonly used spring embedder. Furthermore, our algorithm finds solutions which are close to optimal on the complete graphs $\mathcal{K}_n$ for $n \leq 30$.

Deutsche Zusammenfassung

Diese Arbeit beschäftigt sich mit dem rectilinear crossing minimization problem, das NP-schwer ist [BD93]. Wir schlagen eine Heuristik für das Berechnen einer geradlinigen Zeichnung eines beliebigen Graphen G vor, sodass diese Zeichnung eine kleine geradlinige Kreuzungszahl realisiert. Wir verfolgen einen Ansatz, der von Gutwenger et al. [GMW05] inspiriert ist. Unser Algorithmus extrahiert einen planaren Subgraph, der möglichst viele Kanten von G enthält. Die übrigen Kanten fügt der Algorithmus iterativ wieder hinzu. Nach jedem Einfügen einer Kante werden Knoten verschoben, um die Kreuzungszahl der Zeichnung zu reduzieren. Wir fixieren die Position von allen Knoten, bis auf einen Knoten v und positionieren v sodass die Anzahl an Kreuzungen in der Zeichnung minimiert wird. Diese Position kann in einer Laufzeit von $\mathcal{O}((m \cdot d_{max})^2)$ gefunden werden. Wir evaluieren einige Konfigurationen dieses Algorithmus mit unterschiedlichen Strategien zum Vermeiden lokaler Optima. Diese Konfigurationen vergleichen wir mit dem Spring-Embedder von Fruchterman und Reingold [FR91] auf unterschiedlichen Graph-Klassen und stellen fest, dass unsere Konfigurationen Zeichnungen mit signifikant weniger Kreuzungen erzeugen als der üblicherweise benutzte Spring-Embedder. Außerdem findet unser Algorithmus Lösungen, die auf den vollständigen Graphen $\mathcal{K}_n$ für $n \leq 30$ fast optimal sind.

Contents

1 Introduction **1**
1.1 Related Work . . . 2
1.2 Contribution and Outline . . . 6

2 Preliminaries **9**
2.1 Geometric Definitions . . . 9
2.2 Graph Drawing . . . 9
2.3 Statistical Tests . . . 11

3 A Framework for Rectilinear Crossing Minimization **13**
3.1 Finding the locally Optimal Position of a Node . . . 14
3.2 Running Time . . . 17

4 Finding a Rectilinear Drawing with a Low Crossing Number **19**
4.1 Node Order . . . 19
4.2 Moving a Subgraph . . . 21
4.3 PrEd . . . 27
4.4 Edge Order Post-processing . . . 27

5 Evaluation and Experiments **29**
5.1 Experimental Setup . . . 29
5.1.1 Graph Classes . . . 29
5.1.2 Comparison Algorithm . . . 32
5.1.3 Parameters . . . 32
5.1.4 Configurations . . . 33
5.2 Evaluation . . . 34
5.2.1 Statistical Tests . . . 34
5.2.2 Pairwise Comparison of Configurations . . . 35
5.2.3 Comparison to Edge Insertion . . . 37
5.2.4 Correlation . . . 39
5.2.5 Complete Graphs . . . 40
5.3 Running Time . . . 41

6 Conclusion **45**
6.1 Outlook and Future Work . . . 46

Bibliography **47**

1. Introduction

Graph drawing has applications in many scientific fields like chemistry, biology, computer-science and social sciences. Whenever data and the relations between data points need to be visualized and interpreted by a human, graph drawing is useful. A few examples where graph drawing is used are: protein-protein interaction graphs and phylogenetic trees in biology, social networks, UML-diagrams in software engineering and integrated circuits in VLSI-design. Furthermore, there are some applications which are more relevant in daily life, for example drawing metro maps. In most of these cases the main objective of drawing a graph is to draw it well arranged such that a human can easily percept its structure and conclude the important information from it. Nevertheless, there are situations in which the purpose of drawing a graph is not only facilitating human perception. For example in VLSI-design drawing a graph in the plane corresponds directly to placing transistors on a chip, which are connected by wires. For the purpose of designing a chip, it is important that wires do not cross. But not every design is planar. This problem is usually solved by using multiple layers for one chip. One routing strategy is the following: all transistors are placed on one layer. If two wires cross, one wire is routed to the second layer immediatly before the crossing and routed back to the first layer directly after the crossing. This way crossings between wires are avoided. Nevertheless, this work around for crossings increases the overall wire length and the number of wires between different layers needs to be minimized due to technical reasons. Therefore, minimizing the crossings of a drawing is benificial in VLSI-design. In summary the quality criteria for drawing a graph differ from application to application.

Nevertheless, there are some metrics for graph-drawing aesthetics like *crossing minimization*, *bend minimization* and *symmetry* which have proved themselves. Purchase found in her study, that minimizing the number of crossings has a great positive effect on human perception and that bend minimization is also benificial for this purpose [Pur97]. In this thesis we focus on the problem of finding a straight-line drawing of a graph G, which has the smallest rectilinear crossing number among all possible drawings of G. We refer to this problem as *rectilinear crossing minimization* throughout this thesis and the respective rectilinear crossing number is denoted with $\overline{\mathrm{cr}}(G)$.

This problem can also be interpreted as optimizing a drawing of a graph according to the following two aesthetic criteria:

1. minimize the number of edge bends, i.e. fix this number to zero.

2. minimize the rectilinear crossing number.

Gutwenger et al. designed a heuristic for finding a drawing of G, which has the smallest possible crossing number. Note, that in this case the drawing which realizes this smallest crossing number can have edges with bends and thus does not need to be straight-line. Such a drawing is called *topological*. This problem is named *crossing minimization* and the respective crossing number is denoted with $\text{cr}(G)$. Both of these problems, rectilinear crossing minimization and crossing minimization, are NP-hard according to Garey and Johnson [GJ83]. Moreover, the crossing number $\text{cr}(G)$ is not an approximation for $\overline{cr}(G)$, which means that the heuristic by Gutwenger et al. does not necessarily minimize $\overline{\text{cr}}(G)$. In fact Bienstock and Dean showed, that for each $k \in \mathbb{N}, k \geq 4$ there is a graph G with $\text{cr}(G) = 4$ and $\overline{\text{cr}}(G) = k$. In other words, the difference $\overline{\text{cr}}(G) - \text{cr}(G)$ can be arbitrarily high [BD93].

In fact the heuristic by Gutwenger et al. does not necessarily minimize the rectilinear crossing number of a graph G. This motivates our intention to design a heuristic for rectilinear crossing minimization.

1.1 Related Work

As already mentioned, in this thesis we focus on the problem of rectilinear crossing minimization, which means finding a drawing $\mathcal{D}^*$ of a graph G, such that $\mathcal{D}^*$ has the smallest rectilinear crossing number $\overline{\text{cr}}(\mathcal{D}^*)$ among all possible drawings of G. The crossing number $\overline{\text{cr}}(\mathcal{D}^*)$ associated to this problem is called the rectilinear crossing number of G and is denoted with $\overline{\text{cr}}(G)$. Note, that we use the same notation $\overline{\text{cr}}$ both for the rectilinear crossing number of a graph, which is NP-hard to find [GJ83] and the rectilinear crossing number of a drawing, which can be found in polynomial-time [SH76].

Pach gives a summary on the different crossing-number problems [PT00]. We mainly make the same assumptions on a graph G of which we want to determine the crossing number or the rectilinear crossing number as Pach in his summary [PT00]:

1. No two edges cross more than once in their interior.
2. There are at most two edges that cross at a specific intersection point.
3. There are no edges, which pass a node of G.
4. The nodes are in *general position*, which means that no three nodes of G are collinear.

Not all of these properties are necessary for every result on the topic. But to avoid confusion, we make these presumptions.

Before we give a brief overview on the different research directions related to rectilinear crossing minimization, we shortly describe a problem related to our approach of minimizing the crossings of a straight-line drawing. As explained before, rectilinear crossing minimization can also be described as successively optimizing a drawing according to two metrics. The first metric is the number of bends, whereas the second metric is the number of crossings. A related approach is to first compute a topological drawing with a minimum number of crossings and subsequently minimize the number of bends. This approach addresses the same two metrics but swaps their priorities and is referred to as *drawing a planarization*. Minimizing the number of edge bends of a topological drawing is related to the problem *simple stretchability* by Mnëv and Shor which is NP-hard [Sho]. Radermacher finds a heuristic for drawing a planarization which works well in practice [Rad15].

In the following we summarize some findings related to rectilinear crossing minimization and crossing minimization. These include the complexity of the problem, upper- and

lower-bounds for the rectilinear crossing number of general graphs and interesting graph classes like the complete graphs. Furthermore, we summarize the findings of force directed graph drawing. Moreover, we describe some heuristics and approximation algorithms for crossing minimization and rectilinear crossing minimization.

Complexity

Computing the crossing number $\mathrm{cr}(G)$ of a graph G was proven to be *NP-complete* by Garey and Johnson [GJ83]. Bienstock proved that computing the rectilinear crossing number is *NP-hard* [Bie91]. Furthermore, Bienstock proved that there exists an infinite family of graphs $\{G^n\}_{n\in\mathbb{N}}$ such that for each drawing which realizes the rectilinear crossing number $\overline{\mathrm{cr}}(G^n)$, the coordinates of the nodes require more than polynomially many bits [Bie91, Theorem 3]. Until now it is not decided whether computing the rectilinear crossing number lies in NP or not.

Schäfer showed that computing the rectilinear crossing number is $\exists\mathbb{R}$-complete [Sch09]. This means computing the rectilinear crossing number is as hard to solve as the existential theory of the reals. More precisely, Schäfer recognized that Bienstock already proved $\exists\mathbb{R}$-hardness of computing the rectilinear crossing number by reducing simple stretchability which is $\exists\mathbb{R}$-complete to computing the rectilinear crossing number. Furthermore, Schäfer proved that computing the rectilinear crossing number lies in $\exists\mathbb{R}$.

Computing the rectilinear crossing number differs from rectilinear crossing minimization in that the latter involves computing a straight-line drawing of a graph a minimum number of crossings, whereas the first only computes this minimum number of crossings but does not necessarily compute a respective drawing. Nevertheless, like computing the rectilinear crossing number, rectilinear crossing minimization is NP-hard. This follows from an easy reduction.

Because of the complexity of computing the rectilinear crossing number, there are three main directions of research concerning this problem. There are many proven *upper and lower bounds* for the rectilinear crossing number of specific graph classes, but also some results for general graphs. Further, there are several *heuristics* for computing a low rectilinear crossing number, for example there are many variants of *force directed graph drawing*. Moreover, there is a polynomial-time *approximation algorithm*. Nevertheless, this algorithm is only an asymptotic approximation and we are not aware of an approximation algorithm which differs from an optimal solution only by a constant factor.

General Bounds

It is obvious, that $\overline{\mathrm{cr}}(G) \geq \mathrm{cr}(G)$. Subsequently lower-bounds for $\mathrm{cr}(G)$ also apply to $\overline{\mathrm{cr}}(G)$. One of the most important lower-bounds for $\mathrm{cr}(G)$ and hence also for $\overline{\mathrm{cr}}(G)$ is the Crossing Lemma, which was discovered independently by Ajtai, Chvátal, Newborn and Szemerédi [ACNS82] and Leighton [Lei83]:

$$\mathrm{cr}(G) \geq \frac{1}{33.75} \cdot \frac{e^3}{n^2},$$

where e is the number of edges of G and n denotes the number of nodes of G. Nevertheless, $\mathrm{cr}(G)$ is not a good lower-bound for $\overline{\mathrm{cr}}(G)$. In particular, Bienstock and Dean showed, that for each $k \in \mathbb{N}, k \geq 4$ there is a graph G with $\mathrm{cr}(G) = 4$ and $\overline{\mathrm{cr}}(G) = k$. In other words, the difference $\overline{\mathrm{cr}}(G) - \mathrm{cr}(G)$ can be arbitrarily high [BD93].

Approximation Algorithms for the Crossing Number

Bhatt and Leighton found an $\mathcal{O}((\mathrm{cr}(G) + n) \cdot B^2(n) \cdot \log^2(n))$ approximation algorithm for the crossing minimization problem, where $B(n)$ is the *bisection width* of the respective graph [BL84]. The bisection width of a graph is the minimum number of edges which have to be removed in order to divide the graph into two connected components of equal size. The idea of Bhatt and Leigton was to recursively bisect G into two equally sized subsets V_1 and V_2, such that a minimum number of edges proceed between V_1 and V_2 until a subset includes at most one node. With their work on balanced cuts Leighton and Rao could improve Bhatt and Leighton's algorithm to a $\mathcal{O}((\mathrm{cr}(G) + n) \cdot \log^4(n))$ approximation algorithm for a bounded-degree graph. The currently best known approximation algorithm for bounded degree graphs is the $\mathcal{O}((\mathrm{cr}(G) + n) \cdot \log^2(n))$ approximation algorithm by Arora, Vazirani and Rao [ARV09].

An Approximation Algorithm for the Rectilinear Crossing Number

Recently Fox et al. used findings on order types to develop an approximation algorithm for the rectilinear crossing number problem. A survey on order types can be found in the book "New trends in discrete and computational geometry" [Pac12, chapter 5]. The important property that makes order types so useful for applications in rectilinear crossing number minimization can be found in Theorem 5.2 of the book, which is adopted from the book "The Banach-Tarski Paradox" by Wagon [Wag93]. Based on this property Fox et al. remark, that $\overline{\mathrm{cr}}(G)$ can be computed in an $2^{\mathcal{O}(n^3)}$ time-bound by brute-forcing over all order-types [FPS16, Lemma1]. Furthermore, Fox et al. give the first polynomial-time approximation-algorithm for computing the rectilinear crossing number.

Lemma 1.1 (Theorem 2 in [FPS16])**.** *There is a deterministic $n^{2+\mathcal{O}(1)}$-time algorithm for constructing a straight-line drawing of any n-vertex graph G in the plane with*

$$\overline{\mathrm{cr}}(G) + \mathcal{O}\left(n^4/(\log\log n)^\delta\right)$$

crossing pairs of edges, where $\delta > 0$ is an absolute constant.

Moreover Fox, Pach and Suk [FPS16] state, that for the dense graphs $\mathcal{K}_n$ their algorithm approximates $\overline{\mathrm{cr}}(\mathcal{K}_n)$ by a factor of $(1 + \mathcal{O}(1))$ [FPS16, Corollary 1].

Bounds for Complete Graphs

There has been much effort to compute or at least approximate the rectilinear crossing number $\overline{\mathrm{cr}}(\mathcal{K}_n)$ of the graphs $\mathcal{K}_n$. For a detailed overview of the research on the rectilinear crossing number of complete graphs, see [ÁFMS13]. The search for bounds on $\overline{\mathrm{cr}}(\mathcal{K}_n)$ is devided into estimating $\overline{\mathrm{cr}}(\mathcal{K}_n)$ from above and from below. The currently known best upper-bound for $\overline{\mathrm{cr}}(\mathcal{K}_n)$ was discovered by Fabila-Monroy and López [FML14].

Theorem 1.2 (Theorem 1.1 in [FML14])**.**

$$\overline{\mathrm{cr}}(\mathcal{K}_n) < 0.38047223873 \cdot \binom{n}{4} + \mathcal{O}(n^3)$$

Fabila-Monroy and López use an approach, that tries to find a drawing with a low rectilinear crossing number for a small graph and expand the result to arbitrary n with a recursive formula. Fabila-Monroy and López use the following reccurence.

Theorem 1.3 (Theorem 4 in [ÁCFM+10])**.** *Let $m < n$ be a natural number.*

$$\overline{\text{cr}}(K_n) \leq \frac{24\,\overline{\text{cr}}(\mathcal{K}_m) + 3m^3 - 7m^2 + (30/7)m}{m^4} \cdot \binom{n}{4} + \mathcal{O}(n^3).$$

For the purpose of finding a low crossing-number for complete graphs $\mathcal{K}_n$ with small n, Fabila-Monroy and López took the currently best known drawing for $\mathcal{K}_n$ from the web-page of the rectilinear crossing number project [Aic13] and improved its rectilinear crossing number by choosing vertices randomly and moving them to points in proximity, if this yields a lower rectilinear crossing number. To this end Fabila-Monroy designed an $\mathcal{O}(n^2)$ algorithm for computing the rectilinear crossing number of a drawing.

The best currently known lower-bound for $\overline{\text{cr}}(\mathcal{K}_n)$ was discovered by Ábrego and Férnandez-Merchant [ÁFMLS08].

Theorem 1.4 (Corollary 3.5 in [ÁFMLS08])**.**

$$\overline{\text{cr}}(\mathcal{K}_n) > 0.379972 \cdot \binom{n}{4} + \mathcal{O}(n^3)$$

In summary the ratio between the best known lower-bound and the best known upper-bound for $\mathcal{K}_n$ is currently above 0.9986. Moreover, $\overline{\text{cr}}(\mathcal{K}_n)$ up to a size of $n = 28$ have been computed exactly.

Planar Subgraphs

There are two problems concerning planar subgraphs, which we are interested in. *Maximum planar subgraph* consists of finding the lowest number of edges e^{del} which have to be deleted from a graph G such that the resulting subgraph G^* is planar. This problem is *NP-complete* [LG77] and more precisely *MAX SNP-hard* [CFFK98].
MAX SNP-hardness implies, that the optimal solution OPT(G) of maximum planar subgraph cannot be approximated in polynomial time arbitrarily good, with absolute guarantees, unless $P = NP$. The concept of MAX SNP-hardness was established by Arora et al. [ALM+98]. The best currently known polynomial-time approximation algorithm for the maximum planar subgraph problem is a 4/9-approximation algorithm discovered by Călinescu [CFFK98] with a running time of $\mathcal{O}(m^{\frac{3}{2}} n \cdot \log^6(n))$.

Nevertheless, the problem *maximal planar subgraph* can be solved in polynomial time. This problem only requires to compute a maximal planar subgraph G^* of G, which means adding an edge $e \in E \backslash E^*$ to G^* results in a non-planar graph. There are several known polynomial-time algorithms, which compute a maximal planar subgraph. The theoretically fastest currently known such algorithm was discovered by Djidjev [Dji06]. It computes a maximal planar subgraph in $\mathcal{O}(m + n)$ time.

The first correct polynomial-time algorithm, which solves maximal planar subgraph was discovered by Chiba, Nishioka and Shirakawa [CHI79]. It terminates in an $\mathcal{O}(mn)$ time-bound and is based on the planarity-testing algorithm by Hopcroft and Tarjan [HT74]. Jayakumar et al. proposed an $\mathcal{O}(n^2)$-time algorithm [JTS89], but Kant recognized that the algorithm by Jayakumar et al. does not necessarily compute a maximal planar subgraph and proposed some modifications which seemed to fix Jayakumar's algorithm. Subsequently Mutzel et al. recognized that both Kant's modified algorithm as well as Jayakumar's original approach do not necessarily compute a maximal planar subgraph [JLM98]. Nevertheless, according to Chimani et al. the $\mathcal{PQ}$-tree based approach by Jayakumar et al. is "one of the best heuristics for the NP-hard maximum planar subgraph problem" [CGJ+11, p.549].

Edge Insertion

Gutwenger's edge insertion algorithm [GMW05] is a heuristic for computing the crossing number of a graph $G = (V, E)$. The approach of Gutwenger consists of two steps.

1. Computing a planar subgraph $G^* = (V^*, E^*)$ of G preferably with a high number of edges.
2. Iteratively reinserting the missing edges in $E \backslash E^*$ into G^* optimally.

With this heuristic Gutwenger simultaneously solved the *optimal edge insertion problem* in polynomial time. Gutwenger et al. use $\mathcal{SPQR}$-trees to represent the different combinatorial embeddings of a graph G. Each node of an $\mathcal{SPQR}$-tree is associated to a biconnected graph. Gutwenger et al. observed, that determining the embeddings of these biconnected graphs already fixes the embedding of G. Furthermore, Gutwenger et al. show, that inserting an edge $e = (s, t)$ into an embedded graph G optimally can be achieved by computing the shortest path $e_1, ..., e_k$ in the extended dual-graph $\mathcal{G}_D^e$ of G. The edge e can then be drawn, such that it crosses exactly $e_1, ..., e_k$.

With these tools at hand, Gutwenger et al. discovered a linear-time algorithm for solving the optimal edge insertion problem and the currently best known heuristic for crossing minimization.

Force Directed Graph Drawing

Force directed drawing algorithms are commonly used for computing rectilinear drawings of a graph. The concept of force directed graph drawing goes back to Eades [Ead84]. According to the idea of Eades, the nodes of a graph are modeled as steel rings, whereas edges are springs. Initially the nodes are placed somehow in the plane. The adjacent springs then pull the nodes to different positions until the system reaches a stable state. More precisely the approach of Eades is to iteratively calculate attractive and repellent forces between certain pairs of nodes and to move them according to these forces until only minor changes occur. Attractive are computed between all pairs of nodes which are connected by edges, whereas repellent forces are computed between all pairs of nodes in the approach of Eades.

The spring embedder by Fruchterman and Reingold is a refined version of this algorithm [FR91]. Unlike Eades algorithm, the spring embedder by Fruchterman and Reingold uses forces which depend linearly on the distance of two nodes. This is more physically accurate but the nodes may not reach a stable state in this scenario. Therefore, Fruchterman and Reingold introduce a temperature that decreases over time. This temperature defines the maximal distance d a node can be moved. This temperature decreases alongside with d in each iteration. A detailed summary on force directed graph drawing was given by Di Battista et al. [DBETT94].

1.2 Contribution and Outline

Due to the complexity of rectilinear crossing minimization we have no hope to find a polynomial-time algorithm which produces a drawing that realizes the minimum rectilinear crossing number of G. Therefore, we develop a heuristic which computes a straight-line drawing of G with a low crossing number in polynomial time. in polynomial time. Our algorithm is inspired by the heuristic of Gutwenger et al. for crossing minimization [GMW05]. Our algorithm consists of two phases:

1. Find a planar subgraph G^* of G, which includes as many edges of G as possible.
2. Draw G^* and iteratively reinsert all edges not included in G^* into G^* while keeping the rectilinear crossing number low.

The task of the first phase is commonly known as maximum planar subgraph problem. As explained in section 1.1, this problem has been investigated by many researchers and there are several good heuristics for solving the maximum planar subgraph problem. In this thesis we mainly concentrate on the second phase of the algorithm and restrict ourselves on already existing heuristics for the first phase.

Our idea is to move nodes to a good position after each edge reinsertion to reduce the rectilinear crossing number. In section 3 we find a way to compute a good position for a single node v. More precisely, we show how to compute the rectilinear crossing minimal position of v given that all other nodes have a fixed position. This is a purely geometric operation. In section 4 we discuss strategies to escape from local optima and different orders in which nodes are moved. Furthermore, we connect these different components and propose an algorithm with several variants. In section 4.1 we discuss which nodes shall be moved in which order. We develop several ordering strategies and see that a finite number of node movements according to each strategy results in a locally optimal drawing. This means no single node can be moved such that the rectilinear crossing number of the drawing is improved. Thereupon, we develop a strategy to escape local optima in section 4.2. In particular we propose to move a whole subgraph to a good position at once. Moving these subgraphs can result in nodes being very close together. This is why we use a force directed algorithm after each edge insertion step, which pulls apart nodes that are too close but does not affect the rectilinear crossing number. This algorithm by Bertault is called `PrEd` [Ber99] and it is shortly described in section 4.3. Furthermore, we propose some post-processing strategies, which delete and reinsert some edges in a certain order in 4.4. In chapter 5 we choose some open parameters like the heuristic for finding a planar subgraph and the initial drawing of this planar subgraph. Subsequently we experimentally evaluate the configurations of our algorithm and compare them to one another and to the commonly known spring embedder by Fruchterman and Reingold on several different graph classes [FR91]. In the following chapter 2 we will explain and determine some fundamental terms in graph drawing, geometry and statistical testing.

2. Preliminaries

In the following chapter we introduce some preliminaries which are needed throughout this thesis.

2.1 Geometric Definitions

Let a, b, c be points in the euclidean plane $\mathbb{R}^2$. A *line segment* $\mathcal{S}(a, b)$ is a set of points $\mathcal{S}(a, b) = \{t(b - a) + a \mid t \in [0, 1]\}$. We call a and b endpoints of the line segment. A *line* $\mathcal{L}(a, b)$ is a set of points $\mathcal{L}(a, b) = \{t(b - a) + a \mid t \in (-\infty, \infty)\}$. A *ray* $\mathcal{R}(c, a)$ is a set of points $\mathcal{R}(c, a) = \{t \cdot c + a \mid t \in (0, \infty)\}$. We call c direction and a startpoint of $\mathcal{R}(c, a)$. The generic term for line segments, lines and rays is *linear curve*. A *half-plane* $\mathcal{H}(a, b)$ with respect to a line $\mathcal{L}(a, b)$ is a set of points, such that all vectors $v \in \mathcal{H}$ lie to the left of $\mathcal{L}(a, b)$, i.e. the vector $b - a$ forms a left-turn with v.

The points a, b, c in the euclidean plane are said to be *collinear*, if and only if there is a line that passes through all three points a, b and c. A set of points $P \subset \mathbb{R}^2$ is said to be *in general position* if and only if no three points $a, b, c \in P$ are collinear.

2.2 Graph Drawing

An undirected *graph* is a tuple (V, E) where V denotes a set of nodes and E a set of edges with $E \subseteq V \times V$. We call an edge $e = (v_1, v_2) \in E$ *adjacent* to $v_1 \in V$ and $v_2 \in V$. We say v_1 and v_2 are the *endpoints* of e.

A *drawing* $\mathcal{D}$ of a graph $G = (V, E)$ is an injective mapping of every node $v \in V$ to a point in $\mathbb{R}^2$ and each edge $e \in E$ to an open Jordan-Curve C_e, such that $\mathcal{D}$ fulfills the *drawing properties* and such that the adjacent nodes u, v of edge e are mapped to the endpoints of the Jordan-Curve C_e. In the following, we will identify a Jordan-Curve with its associated edge and a point with its associated node. Throughout this thesis, we will use the terms interchangeably in the context of a drawing. We say two edges e_1, e_2 *cross* if they intersect in a point which is not an end-point of e_1 or e_2. With this notation at hand, we can now state the drawing properties:

1. for all nodes v there exists no edge e such that e passes through v.
2. the nodes are in general position, which means no three nodes have collinear positions.
3. at each intersection of edges, there are at most two edges which cross.

We also say a graph is *drawn* in the plane. An *intersection point* is a point belonging to two different edges which is not an end-point of an edge. The *crossing number* of a drawing is the number of intersection points. A drawing is called *straight-line*, if all edges are mapped to line segments. Note that a straight-line drawing of G is fully determined by the positions of all nodes $v \in V$. In the following we will always consider straight-line drawings, when we talk about drawings unless otherwise stated. A drawing is called *planar* if no two edges cross. A graph is called *planar* if it has a planar drawing. A planar drawing parts the euclidean plane $\mathbb{R}^2$ into regions, which we call *faces*. Each of these faces f is bordered by a set of edges. These edges are called *incident* to the face f. We also say that e borders f. Moreover, two faces f and g are incident to each other, if there is an edge e such that both faces f and g are incident to e. A node v is called incident to a face f, if v is incident to an edge e which borders f.

Let $\mathcal{D}$ be a planar drawing of G_D. The planar drawing $\mathcal{D}$ parts the plane into regions, which we call faces. The *dual graph* G_D of $\mathcal{D}$ contains a node v_f for each face f induced by $\mathcal{D}$. Each face f is said to be the *dual face* of v_f. Two nodes v_f, v_g of G_D are connected by an edge $e = (v_f, v_g)$ if and only if f and g are adjacent faces in $\mathcal{D}$.
The *extended dual graph* $G_D^{s,t}$ of the dual graph G_D with two nodes s and t is obtained from G_D by adding the nodes s and t to G_D and adding an edge $e = (s, v_f)$ for each $v_f \in V_D$ such that the dual face of v_f is incident to s or incident to t.

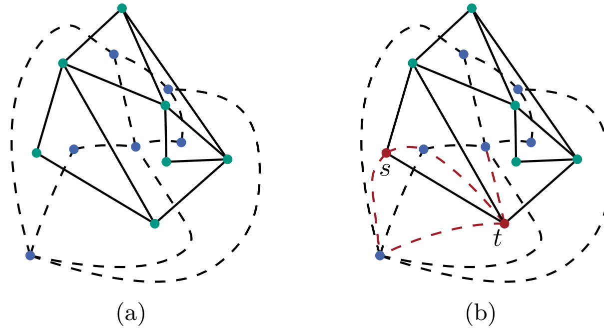


Figure 2.1: The left figure shows the dual graph G_D and the graph G. The dual graph G_D is drawn with blue nodes and dashed edges, whereas G is drawn with green nodes and normal edges. The right figure shows the extended dual graph $G_D^{s,t}$. The red nodes s and t belong to G and to $G_D^{s,t}$. The edges which make the difference between G_D and $G_D^{s,t}$ are drawn red and dashed.

A *planarization* G_P of a drawn graph $G = (V, E)$ is a graph defined as follows. Let $\mathcal{D}$ be the drawing of G. The graph G_P contains one node for each node of $v \in V$. Furthermore, each edge $e \in E$ which has no crossing in the drawing $\mathcal{D}$ is also contained in G_P. Moreover, the graph G_P contains a node v_c for each crossing c of two edges $e \in E$ in the drawing $\mathcal{D}$ and an edge $e = (v_c, v_a)$ between each such node v_c and each node v_a that is incident to an edge involved in the crossing c. Intuitively, G_P is obtained by placing a node at each crossing of $\mathcal{D}$ and each edge passing such crossings is split at these points.

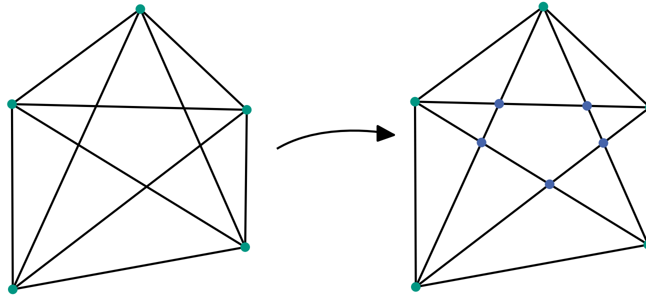


Figure 2.2: On the left side a drawing $\mathcal{D}$ of the graph G is shown. The drawing on the right side shows a drawing of the planarization G_P of $\mathcal{D}$. The nodes in which G_P differs from G are drawn blue.

An *arrangement* $\mathcal{A}(\mathcal{S})$ is the planarization of a drawn graph $G_{\mathcal{A}}$ induced by a set of linear curves $\mathcal{S}$. The drawn graph $G_{\mathcal{A}}$ is obtained from the set of linear curves $\mathcal{S}$ as follows.

1. Place a node at each intersection point of two linear curves $l_1, l_2 \in \mathcal{S}$.
2. Construct a bounding rectangle $\mathcal{R}$, which includes each intersection point of two linear curves $l_1, l_2 \in \mathcal{S}$. Add the line segments which are contained in the boundary of $\mathcal{R}$ to $\mathcal{S}$. Place a node at each intersection point of a linear curve $l \in \mathcal{S}$ with $\mathcal{R}$.
3. Add an edge (u, v) to $G_{\mathcal{A}}$ for each two nodes such that there is a linear curve l which contains u and v.

2.3 Statistical Tests

In this section we describe the binomial sign test, the adapted binomial sign test and spearman's rank-order correlation coefficient. The Adapted Binomial Sign Test was first used in a very similar form by Radermacher [Rad15].

Binomial Sign Test for Two Dependant Samples

In the following we shortly describe the Binomial Sign Test on a graph-class $\mathcal{G}$. Let A_1, A_2 be the two algorithms, we want to compare. Let $\mathcal{G}_S = \{G_1, ..., G_n\}$ be the set of sample-graphs chosen from the graph-class $\mathcal{G}$.

We denote with $\overline{\text{cr}}(A, G_i)$ the crossing-number of the final drawing of algorithm A on graph G_i. The Binomial Sign Test requires two assumptions to be applicable:

1. the graphs $G_i \in \mathcal{G}_S$ are chosen uniformly at random from $\mathcal{G}$
2. $\overline{\text{cr}}(A_1, G)$ and $\overline{\text{cr}}(A_2, G)$ can be rank-ordered for each $G \in \mathcal{G}$, i.e. it can be decided whether $\overline{\text{cr}}(A_1, G) < \overline{\text{cr}}(A_2, G)$ or not.

The second assumption always applies, since $\overline{\text{cr}}(A_1, G)$ and $\overline{\text{cr}}(A_2, G)$ can easily be rank-ordered, because they are both positive natural numbers. The first assumption has to be tested before applying the Binomial Signed Rank Test to a sample set $\mathcal{G}_S$ of a graph class $\mathcal{G}$.

The Binomial Signed Rank Test $\text{Bin}(\mathcal{G}_S, p, \alpha)$ with probability p and significance-level α on the graph-set $\mathcal{G}_S$ involves the following two steps.

1. Deploy a hypothesis H_0^p.
   H_0^p : *For a ratio of at least p of the graphs $G \in \mathcal{G}$, $\overline{\text{cr}}(A_1, G) < \overline{\text{cr}}(A_2, G)$ holds.*

2. Test the hypothesis H_0^p on $\mathcal{G}_S$. For this purpose π^+ is computed, which is the number of sample-graphs $G \in \mathcal{G}_S$ such that $\overline{\text{cr}}(A_1, G) < \overline{\text{cr}}(A_2, G)$.

   Compute π^+ the number of samples $G \in \mathcal{G}$ such that $\overline{\text{cr}}(A_1, G) < \overline{\text{cr}}(A_2, G)$. Further compute π^+_{rel} the proportion of samples $G \in \mathcal{G}$ such that $\overline{\text{cr}}(A_1, G) < \overline{\text{cr}}(A_2, G)$. The hypothesis H_0^p is accepted if $\pi^+_{rel} > p$ and if $P(x \geq \pi^+) \leq \alpha$. The probability $P(x \geq \pi^+)$, that the number of positively signed difference is greater than π^+ can be easily computed because the positively signed differences are binomial distributed. Otherwise the hypothesis H_0^p is rejected.

The hypothesis H_0^p means, that A_1 outperforms A_2 on a ratio of at least p of the graph class $\mathcal{G}$.

Adapted Binomial Sign Test

In the following we will adapt the Binomial Sign Test such that we can compare two algorithms more accurately. The hypothesis H_0^p can be expanded by a multiplicative factor.

H_δ^p: *For a ratio of at least* p *of the graphs* $G \in \mathcal{G}$, $(1+\overline{\text{cr}}(A_1, G)) \cdot \delta < (1+\overline{\text{cr}}(A_2, G))$ *holds.*

The hypothesis H_δ^p means, that A_2 outperforms A_1 by a factor δ on a ratio of at least p of the graphs $G \in \mathcal{G}$.

Spearman's Rank-Order Correlation Coefficient

In the following we describe Spearman's rank order correlation coefficient on a graph class $\mathcal{G}$ and a sample set $\mathcal{G}_S$ chosen uniformly at random from $\mathcal{G}$. The test computes a correlation coefficient $r \in [-1, +1]$ between two random variables X and Y on $\mathcal{G}$. The coefficient r indicates the degree of correlation between both variables. The sign of r indicates the kind of correlation between X and Y. A negative sign indicates a negative correlation, which means each increase in variable X happens alongside with a decrease in variable Y. Accordingly a positive sign indicates a positive correlation, which means each increase in variable X is accompanied with an increase in variable Y.

The rank-order correlation coefficient can be devided into the steps.

1. Deploy the hypothesis H_0 stating, that there is a positive correlation between X and Y or more precisely $H_0 : \rho > 0$.
2. Each sample G is given a rank R_x, which is the position of $X(G)$ in a sequence of all values of X on the sample set ordered from low to high. Similarly G is given a rank R_y. Thus, we obtain a difference in ranks $d^G = R_x - R_y$. The approximative correlation coefficient r can now be computed by

$$r = 1 - \frac{6 \sum_G d_G^2}{n \cdot (n^2 - 1)}. \tag{2.1}$$

   Whether or not the hypothesis is rejected or accepted can be looked up in table A18 in [She07, p.707]. Moreover, r gives further inside in the kind of correlation between X and Y.

3. A Framework for Rectilinear Crossing Minimization

We are concerned with rectilinear crossing minimization, which is the problem of finding a straight-line drawing of a graph G that has the smallest rectilinear crossing number among all possible drawings of G.

This problem is NP-hard and more precisely it is $\exists\mathbb{R}$-complete as already seen in section 1.1. Therefore, we have no hope to solve the problem in an exact way in polynomial time. Nevertheless, we want to find a heuristic for rectilinear crossing minimization.
In other words, we want to find a straight-line drawing of G with a small crossing number.

Our heuristic for the rectilinear crossing minimization problem is a 2-phase approach:

- First as few edges as possible are removed from G in order to obtain a planar subgraph G^* of G. Then G^* can be drawn in the plane with a common drawing algorithm for planar graphs.
- In a second step the deleted edges are iteratively reinserted into G^*. After each iteration we want to find a drawing $\mathcal{D}$ of G^*, such that the rectilinear crossing number $\overline{\text{cr}}(\mathcal{D})$ of $\mathcal{D}$ is low. We call this step *edge-insertion step*

Our approach is closely related to Gutwenger's edge insertion algorithm, which is concerned with the problem crossing minimization. Crossing minimization was already explained in section 1.1 and means finding a drawing of a graph G, which has the smallest crossing number among all possible drawings of G. Note that this drawing is not necessarily straight-line. Its edges are arbitrary open Jordan-Curves.

Gutwenger's edge insertion algorithm is a heuristic for crossing minimization and uses a way to insert an edge e into $\mathcal{D}$ causing a minimum amount of new crossings. When regarding straight lines, there is only one way to insert e into G, which is connecting both endpoints of e by a straight line. Accordingly, we have to use a different strategy in our approach, in order to decrease the rectilinear crossing number of $\mathcal{D}$. In our approach, we want to move nodes separately after each edge insertion such that the crossing number is reduced. There are several questions like which nodes we want to move and where we want to move them.

The former question is difficult to answer. The most obvious idea would be to move one or both of the endpoints of e. But there are other strategies that come to mind. For example ordering the nodes according to the number of crossings they are involved in and moving

a certain amount of nodes in this order, could be a promising strategy. We will discuss different node-ordering strategies in section 4.1 and now focus on where we should move a given node v.

In our heuristic, we want to fix the position of all nodes but one node v and move v to a position that minimizes the number of crossings in the drawing. We call this position *crossing minimal* or *locally optimal*. We will use the remainder of this chapter to describe an algorithm which finds a locally optimal position of a node v, given a drawn graph $G = (V, E)$.

3.1 Finding the locally Optimal Position of a Node

Let $G = (V, E)$ be a graph with a fixed straight-line drawing and $s \in V$. Let $\phi_s^G : \mathbb{R}^2 \to \mathbb{N}$ be a function, that maps each point $p \in \mathbb{R}^2$ to the crossing-number of G, if we move s to p.

Theorem 3.1. *A crossing-minimum position* $\arg\min_{p \in \mathbb{R}^2} \phi_s^G(p)$ *of s in G can be computed in $\mathcal{O}((\deg(s) \cdot m)^2)$ time.*

We want to find the locally optimal position for only one node, the *active* node $s \in V$. Because there is an infinite number of positions s can be moved to, we try to somehow restrict the search space.

The first thing we notice is, that moving s a small amount in any direction does not change which pairs of edges cross in G. This is why we hope to find regions inside of that s can be moved without changing these pairs of edges and hence the crossing number of the drawing. Then we would only have to consider one point inside each of these regions to find the locally optimal position of s.

In order to find these regions, we observe that each new crossing after a movement of s is caused by an edge adjacent to s, since these are the only edges we move.

We call each of these edges *active*. We call non-adjacent edges, which do not move when moving s, *inactive*. Furthermore, we call each node, that is adjacent to the active node *co-active*. We also note, that each new crossing is caused by an inactive edge and an active edge. Besides, we observe that two adjacent edges can never cross. To summarize these thoughts we make the following observation.

Remark 3.2. *Each new crossing after a movement of s is caused by an active edge e and an inactive edge f not adjacent to e.*

Let $e \in E$ be an active edge and let $f \in E$ be an inactive edge, that is not adjacent to f. We want to determine a region $\mathcal{C}^s(e, f)$ such that, if we place s inside of $\mathcal{C}(e, f)$, e and f cross and if we place s outside of $\mathcal{C}^s(e, f)$ they do not cross.

Figure 3.1 depicts the construction of this region $\mathcal{C}^s(e, f)$. Let s be the active node and $e = (s, t)$ an active edge. Let f be an inactive edge. Let o_j with $j \in \{1, 2\}$ be the adjacent nodes of f. We draw a ray r_j^t starting in o_j in direction of $o_j - t$. We call $R_e^f := \{r_1^t, r_2^t\}$ the set of rays induced by e and f. The edge f together with the rays r_j divides the plane into two regions. We define $\mathcal{C}^s(e, f)$ to be the region of those two, which does not contain t. This region is drawn shaded in figure 3.1.

Lemma 3.3. *The segment $[t, p]$ intersects f, if and only if $p \in \mathcal{C}^s(e, f)$.*

Proof. The region $\mathcal{C}^s(e, f)$ is closely related to the concept of visible regions. The *visibility region* $\mathcal{V}(t, f)$ is a set of points, such that for each point $p \in \mathcal{V}(t, f)$, the point p is

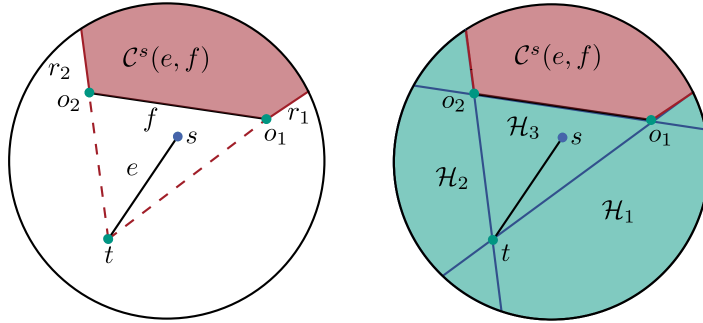


Figure 3.1: On the left side the region $\mathcal{C}^s(e,f)$ of the edges $e = (s,t)$ and f is shown. On the right side the visibility region $\mathcal{V}(t,f)$ which is the intersection of the half-spaces $\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3$, is depicted.

visible from t. This means the segment $\mathcal{S}(p,t)$ does not intersect f. We observe, that $\mathcal{V}(t,f) = \mathbb{R}^2 \setminus \mathcal{C}^s(e,f)$. It is obvious, that $\mathcal{V}(t,f)$ is exactly the union of three visible halfspaces. In particular

$$\mathcal{V}(t,f) = \mathcal{H}_1(t,o_1) \cup \mathcal{H}_2(o_2,t) \cup \mathcal{H}_3(o_1,o_2). \tag{3.1}$$

Figure 3.1 depicts the three half-spaces $\mathcal{H}_1, \mathcal{H}_2$ and $\mathcal{H}_3$.

□

Let $\mathcal{A}(G,s) = (E_{\mathcal{A}}, F)$ be the arrangement induced by all rays r, such that $r \in R_e^f$ for an active edge e and an inactive edge f, that is not adjacent to e and all edges $e \in E$ of graph G. Figure 3.2 shows an example of finding $\mathcal{A}(G,s)$. On the left side we see the graph G. The node s is drawn purple and all neighbors $a_1, ..., a_3$ of s are drawn in different colors. In contrast to that all other nodes are drawn green. On the right we see the same graph G with all regions, that are relevant to the arrangement $\mathcal{A}(G,s)$. Let $1 \leq i \leq 4$. For each neighbor $a_i \in \{a_1, ..., a_3\}$, there is an active edge $e_i = (s, a_i)$. The region $\mathcal{C}^s(e_i, f)$ for each i and each inactive edge f is drawn in the same color as a_i. The insides of these regions are drawn in a shaded color, whereas the inducing rays $R_{e_i}^f$ are drawn opaquely in the same color. Note, that the overlap of two colored regions $\mathcal{C}^s(e_1, f_1)$ and $\mathcal{C}^s(e_2, f_2)$ is a region, such that e_1 crosses f_1 and e_2 crosses f_2, when s is moved to a point inside of the region. The arrangement $\mathcal{A}(G,s)$ is induced by all colored rays and all edges $e \in G$, which are drawn black.

We call a set of points $P \subseteq \mathbb{R}^2$ *crossing-invariant* if and only if

$$\phi_s^G(p_1) = \phi_s^G(p_2) \text{ for all } p_1, p_2 \in P.$$

Lemma 3.4. *The faces of $\mathcal{A}(G,s)$ are crossing-invariant.*

Proof. Let f be a face of $\mathcal{A}(G,s)$. Assume there are two points $p_1, p_2 \in f$ such that moving s to p_1 results in a different crossing number than moving s to p_2, or more formally $\phi_s^G(p_1) < \phi_s^G(p_2)$. This means if s is moved to location p_2 there are at least two edges e_1 and e_2, that cross when s is moved to p_2 and do not cross when s is moved to p_1. Because of Remark 3.2 we can assume, that e_1 is active, e_2 is inactive and not adjacent to e_1. Because e_1 and e_2 do not cross, when s is moved to p_1 and they do cross when s is moved to p_2, we know that $p_2 \in \mathcal{C}^s(e_1, e_2)$ and $p_1 \notin \mathcal{C}^s(e_1, e_2)$. But this means p_1 and p_2 are separated by e_2 or one of the rays $r \in R_e^f$. This is a contradiction to the assumption that f is a face. □

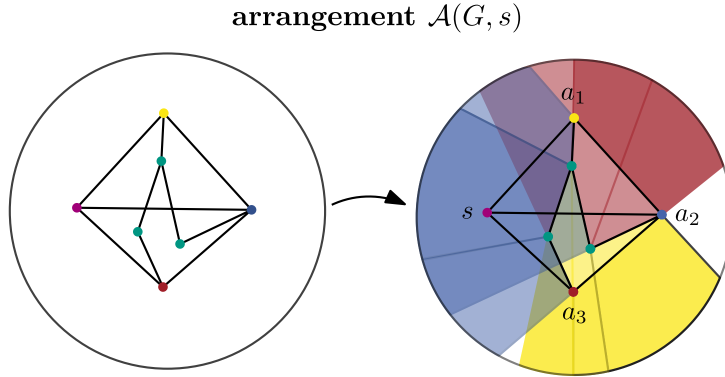


Figure 3.2: On the left side we see the graph G. The node s is drawn purple and all neighbors $a_1, ..., a_3$ of s are drawn in different colors. In contrast to that all other nodes are drawn green. On the right we see the same graph G with all regions, that are relevant to the arrangement $\mathcal{A}(G, s)$.

Solving the Discretized Problem

We have already seen that all faces $f \in F$ are crossing-invariant. Therefore, we have reduced the problem of finding a crossing-minimal position $p^* \in \mathbb{R}^2$ for s to finding the face $f^* \in F$, such that moving s into f^* yields a minimal crossing number. To put it more formally, for each face $f \in F$ it holds, that $\phi_s^G(p^*) = \phi_s^G(p)$ for all $p \in f$. Therefore, we can define a function $\psi_s^G : F \to \mathbb{N}$, with $\psi_s^G(f) := \phi_s^G(p)$ for a point $p \in f$. Note, that ψ maps each face $f \in F$ to the crossing number of G, if s is moved to a point $p \in f$. The function ψ is well-defined, because of Lemma 3.4. We have now reduced computing $\min_{p \in \mathbb{R}^2} \phi_s^G(p)$, to computing $\min_{f \in F} \psi_s^G(f)$, where F is a finite set of faces. In the following sections we will discuss how to compute $f^* = \arg\min_{f \in F} \psi_s^G(f)$, given the arrangement $\mathcal{A}(G, s)$.

The first idea to compute $\arg\min_{f \in F} \psi_s^G(f)$, that comes to mind is to compute $\psi_s^G(f)$ for each $f \in F$ separately. We can do this by choosing a point $p_f \in f$ for every face $f \in F$. Then we can iterate over all points $\{p_f\}_{f \in F}$. In each iteration, we move s to p_f and compute the crossing number of G, while keeping track of the minimum crossing number and the corresponding p_f. Computing the crossing number of G for every face $f \in F$ is very time-consuming. In particular finding the crossing number of G can be solved in an $\mathcal{O}(m \cdot \log(m) + k)$ time bound by [Cha92], where k is the number of crossings. Note that $k \in \mathcal{O}(m^2)$. Finding the crossing number when moving s to p_f can also be solved by only computing the crossings between active and inactive edges. This is possible in an $\mathcal{O}(\deg(s) \cdot m)$ time-bound. But by computing the arrangement, we already have computed information about the relation between adjacent faces. We can therefore try to propagate these information instead of computing them again for each face.

Assume we had information about how the crossing number of G changes, when moving s from a face $f_1 \in F$ to an adjacent face $f_2 \in F$. Let n_f be the number of faces of $\mathcal{A}(G, s)$. Let $d : F \times F \to \mathbb{N}$ be the function that maps a tuple of faces (f, g) to the difference in crossing number, when moving s from f to g. In particular $d(f, g) := \psi_s^G(g) - \psi_s^G(f)$.

Lemma 3.5. *Computing the optimal face $f^* = \arg\min_{f \in F} \psi_s^G(f)$ requires time linear in n_f, if $\psi(h)$ is known for one fixed face $h \in F$ and the differences $d(f, g)$ are known for adjacent faces f and g. This computation can be done by a breadth-first search on the dual-graph of $\mathcal{A}(G, s)$ starting at face h.*

Proof. We observe that the function d has the following property:

$$d(f, g) = d(f, h) + d(h, g) \text{ for all } f, g \in F.$$

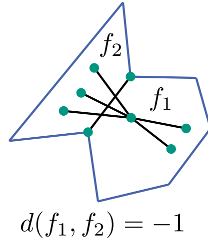


Figure 3.3: Computing $d(f_1, f_2)$ if f_1 and f_2 are adjacent by an edge of G.

This is easy to check from the definition of d. From the definition of $d : F \times F \to \mathbb{N}$, we know that

$$\min_{f \in F} \psi_s^G(f) = \min_{f \in F} \left(\psi_s^G(h) + d(h, f)\right) = \psi_s^G(h) + \min_{f \in F} d(h, f)$$

Accordingly we only have to figure out a way to compute $d(h, f)$ for all $f \in F$ and we do not need to compute $\psi_s^G(h)$. We can compute these values in a breadth-first search on the dual-graph of $\mathcal{A}(G, s)$ starting at h. Let $f_1, ..., f_p \in F$ be the order of faces in which a breadth-first search from h iterates over the faces of $\mathcal{A}(G, s)$. Note that $f_1 = h$. For $i = 1$ we can compute $d(h, f_i) = d(h, f_1) = d(h, h) = 0$. In each iteration $i \in \{2, ..., p\}$ we can compute $d(h, f_i)$ by adding $d(h, f_{i-1})$ and $d(f_{i-1}, f_i)$. A breadth-first search over all faces can be performed in an $\mathcal{O}(n_f)$ time-bound. □

Now we are left with computing $d(f_1, f_2)$ for adjacent faces $f_1, f_2 \in F$. We assume that no three nodes of G have collinear positions. Let $e \in E_{\mathcal{A}}$ be an edge, that connects f_1 and f_2 and let c be a segment that induces e. Note that we can access the inducing segments of each edge e in time $\mathcal{O}(1)$, if we store these segments for each edge e in the data structure for an arrangement. There are two cases we have to consider.

1. The segment c is a ray r_t^e induced by $e \in E$ for $t \in V$. Since every three nodes of G are non-collinear, r_t^e is the only ray , that induces e. Moving s across r_t^e can only cause a difference in crossing number of $+1$ or -1, since s either enters or leaves a crossing region when moved across r_t^e. Accordingly, $d(f_1, f_2) = +1$ or $d(f_1, f_2) = -1$. To check which of these two possibilities is true, we only have to check whether $\overline{e} = (s, t)$ and e cross, if $s \in f_1$. If the answer is yes $d(f_1, f_2) = 1$, otherwise $d(f_1, f_2) = -1$. Note, that in this case $d(f_1, f_2)$ can be computed in $\mathcal{O}(1)$ time, if we store t and the inducing edge e for each ray r_t^e.

2. The segment c is an edge $e \in E$. Let $p \in f_1, q \in f_2$. In this case we can compute $d(f_1, f_2)$ by iterating over all neighbors of s. Let $a_1, ..., a_{\deg(v)}$ be the neighbors of s. Let n_{f_1} be the number of neighbors a_i, such that (a_i, p) crosses e^* and let n_{f_2} be the number of neighbors a_i, such that (a_i, q) crosses e^*. We observe, that $n_{f_2} - n_{f_1} = d(f_1, f_2)$. Note that $d(f_1, f_2)$ can be computed in $\mathcal{O}(\deg(s))$ in this case. We further note, that this computation has to be done only once per edge $e \in E$, if we store $n_{f_2} - n_{f_1}$ at edge e after its first computation. Figure 3.3 shows an example of this computation of $d(f_1, f_2)$.

3.2 Running Time

We can now compound all parts of our algorithm and estimate their running time to proof Theorem 3.1 from the beginning.

Proof of Theorem 3.1. Let $f^* = \arg\min_{f \in F} \psi_s^G(f)$. The following algorithm finds $f^* \in F$ in time $\mathcal{O}((\deg(s) \cdot m))^2$:

1. Compute the arrangement $\mathcal{A}(G, s)$.
2. Perform a breadth-first search on the dual-graph G_D of $\mathcal{A}(G, s)$ starting at the face h with the weight-function $d : F \times F \to \mathbb{N}$. Let $f_1, ..., f_{n_f} \in F$ be the order of faces in which the breadth-first search iterates over the faces of $\mathcal{A}(G, s)$. In each iteration $i \in \{2, ..., n_f\}$ we compute $d(f_{i-1}, f_i)$ as described in the last section in cases 1 and 2. Then we compute $d(h, f_i)$ by adding $d(h, f_{i-1})$ and $d(f_{i-1}, f_i)$ while keeping track of the face f^* with minimal $d(h, f_i)$. The fact that this breadth-first search computes the minimal face f^* given that the distances $d(f_i, f_{i-1})$ are computed correctly for $i \in \{2, ..., n_f\}$, follows from Lemma 3.5.

Let n_s be the number of inducing segments of $\mathcal{A}(G, s)$. Let m be the number of edges of G. Computing the arrangement is dominated in running time by computing the intersections of all segments inducing $\mathcal{A}(G, s)$. This problem can be solved in an $\mathcal{O}(n_s \cdot \log(n_s) + k)$ time-bound by [Cha92], where k is the number of crossings and n_s is the number of segments inducing $\mathcal{A}(G, s)$. Note that $k \in \mathcal{O}(n_s^2)$. We further observe, that $\mathcal{O}(n_s) \in \mathcal{O}(\deg(s) \cdot m)$ since there are three inducing segments for each pair of an adjacent node of s and an inactive edge. Accordingly $k \in \mathcal{O}((\deg(s) \cdot m)^2)$ and computing $\mathcal{A}(G, s)$ can be done in an $\mathcal{O}((\deg(s) \cdot m) \cdot \log(\deg(s) \cdot m) + (\deg(s) \cdot m)^2) \in \mathcal{O}((\deg(s) \cdot m)^2)$ time-bound.

The second step of our algorithm can be computed in an $\mathcal{O}(n_f + \deg(s) \cdot m)$ time-bound. This is because for all but m iterations of the breadth-first search we can bound the running time of the iteration by $\mathcal{O}(1)$. For the remaining m iterations we can bound the running time by $\mathcal{O}(\deg(s))$ as observed in the previous section in case 2. Because $\mathcal{A}(G, s)$ is a planar graph, Euler's formula implies that $n_f \in \mathcal{O}(m_{\mathcal{A}})$. Furthermore, similar to the previous argument we know that $\mathcal{O}(m_{\mathcal{A}}) \in \mathcal{O}(n_s^2)$ and hence $n_f \in \mathcal{O}((\deg(s) \cdot m)^2))$. It follows, that the algorithm has a time-bound of $\mathcal{O}((\deg(s) \cdot m)^2 + m \cdot \deg(s)) = \mathcal{O}((\deg(s) \cdot m)^2)$. □

4. Finding a Rectilinear Drawing with a Low Crossing Number

In this chapter we refine our algorithm for finding a straight-line drawing with a low crossing number and discuss open parameters. We will discuss which nodes should be moved in which order and how we can avoid getting stuck in local optima.

As motivated in more detail in Section 4.2 an idea to escape local optima is to move a subgraph instead of a single node. As already mentioned our approach is related to Gutwenger's edge-insertion-algorithm. Likewise we use a two-phase approach. In each iteration we decide between two possibilities.

1. In contribution to the first question we compute a *node order*. In this order each node is moved to its locally optimal position, with our node-moving approach discussed in Section 3.1 until a certain *stopping criterion* holds or all nodes are in their locally optimal position. At the end of each iteration the node order is updated according to a geometric criterion. We allow nodes to be moved multiple times in one iteration, but not in a row. In section 4.1 we discuss several strategies for computing a good node order.
2. Regarding the second question we move a subgraph to a good position and repair the local structure inside of the subgraph afterwards. The choice of the subgraph and further details are explained in Section 4.2.

Algorithm 4.1 gives an outline of our algorithm, which we will call Geometric Crossing Minimization or GCM from now on. Subsequently we discuss further free parameters regarding our two-phase approach like in which order edges are reinserted into the graph in section 4.4 and how we prevent nodes from getting too close to one another with a force-directed algorithm in section 4.3.

4.1 Node Order

The Geometric Crossing Minimization algorithm moves several nodes to its locally optimal position in each iteration. More precisely GCM computes an order in which nodes are moved. We restrict ourselves to node orders that depend on the geometric position of each node. This means we have to update the node order after each movement of a node. In particular we want a node v to be moved before a node w if the new position of v helps to improve the locally optimal position of w. In this chapter we will present several node

Algorithm 4.1: GEOMETRIC CROSSING MINIMIZATION

Input: Graph $G = (V, E)$
Output: Straight-line Drawing of G

```
1  Find maximum planar subgraph G* = (V*, E*) of G.
2  Remove all edges e ∈ E\E* from G.
3  forall e ∈ E\E* do
4      reinsert e into G
5      if contract then
6          move subgraph to a good position.
7      else
8          choose NodeOrder O
9          while O.stopping-criterion does not hold do
10             choose v as first node in O
11             move v to locally optimal position
12             update O
```

order strategies to achieve this goal. Since we allow each node to appear in the node order multiple times we are interested in the question, whether moving nodes to its locally optimal position in an arbitrary order does converge and results in a stable local optimum. We call a drawing $\mathcal{D}$ of G *stable*, if no movement of a single node can improve the crossing number of $\mathcal{D}$. Let n_{cr} be the number of crossings of the drawing of G and let n be the number of nodes. We make only one restriction for our node order strategies. A node v is not considered again until n_{cr} decreases. Note that this means, the crossing number of the drawing of G was improved by moving a node w. With this restriction at hand we can easily see, that after at least $n \cdot n_{cr}$ tries to move a node in a node order, the drawing of G is stable. A justification for this statement is, that when trying to move all nodes in the node order O at least one has to decrease the crossing number when being moved. But this means the drawing of G is stable. For practical reasons and because moving a node to its locally optimal position is time-consuming, we fix the number of iterations. In the evaluation we use a static number of 10 iterations. This means GCM either starts a new edge-insertion step if every node is placed at its locally optimal position or if the number of iterations exceeds 10. In the following we will discuss some node-ordering strategies. Let e be the currently inserted edge and $s, t \in V$ its endpoints.

Endpoints Strategy

The first node-ordering strategy, that comes to mind is to move both endpoints of e. To explain the idea of this heuristic we subdivide the crossings of G into two categories. The *new crossings* are the crossings, that are caused by the currently inserted edge e in contrast to the *old crossings*, which were already there before the insertion of e.

With this heuristic we try to diminish the new crossings in each iteration. Because we already tried to reduce the old crossings in former iterations. Because e is involved in all new crossings, we hope that the endpoints of e have the highest influence on these crossings. Note that this heuristic does not consider crossings, that are caused by moving the endpoints of e in following iterations. Certainly this node-ordering strategy is not perfect. Nevertheless, it involves only two iterations and can be computed in an $\mathcal{O}(1)$-time bound.

Low-to-High Strategy

The low-to-high strategy orders all nodes respective of their *adjacent crossing number* from low to high. The adjacent crossing number of a node v is the number of crossings, that are caused by an edge incident to v. We only try to move nodes with a strictly positive adjacent crossing-number. The idea of this node-ordering strategy is, that there may be many edges around a node v^* with a high adjacent crossing number. Possibly these edges restrict the locally optimal position of v. Moving the nodes with a lower crossing number might clear the space for v^* to be moved to a better position. Computing the low-to-high order can be done by maintaining a priority queue $\mathcal{Q}$, which contains each node $v \in V$ with its adjacent crossing number as key in $\mathcal{Q}$. We denote the key of v in $\mathcal{Q}$ as rank of v. The initial computation of $\mathcal{Q}$ takes $\mathcal{O}(m)$ time, because for each node $v \in V$ we have to check for all its adjacent edges, whether they cross with e or not. If we use a suitable priority queue, like a binary heap [Tar83], inserting a node can be done in $\mathcal{O}(\log(n))$ time.

In each node iteration we have to update $\mathcal{Q}$. Each update takes $\mathcal{O}(\log(n))$ time in a binary heap. Let v be the currently moved node in this node iteration. In particular we only have to update $\deg(v)$ nodes, the adjacent nodes of v. These are obviously the only nodes which can change their adjacent crossing number when v is moved. The difference between the old and the new rank of $u \in N(v)$ can easily be computed by

$$\text{rank}_{\text{new}} = \begin{cases} \text{rank}_{\text{old}} - 1, & \text{if } (u,v) \text{ crosses } e \text{ before moving } v \text{, but not after moving } v \\ \text{rank}_{\text{old}} + 1, & \text{if } (u,v) \text{ crosses } e \text{ after moving } v \text{, but not before moving } v \\ 0, & \text{else} \end{cases}$$

In conclusion we can update $\mathcal{Q}$ in an $\mathcal{O}(\deg(v) \cdot \log(n))$-time bound.

High-to-Low Strategy

The high-to-low strategy is the exactly opposite node order as the low-to-high strategy. According to this strategy all nodes are ordered respective of their adjacent crossing-number from high to low. Obviously computing this order also requires computing and maintaining the same priority queue $\mathcal{Q}$ as ordering the nodes from low to high. Furthermore the same time-bounds for the initialization of the node-ordering strategy, $\mathcal{O}(m)$, and for each iteration, $\mathcal{O}(\deg(v) \cdot \log(n))$ hold. The idea of this strategy is that nodes with a high adjacent crossing number have a higher potential to decrease the overall crossing number of the drawing of G, when being moved. Because moving nodes is time-consuming for the Geometric Crossing Minimization algorithm, we want to move these nodes first.

Random Strategy

We compare these node-ordering strategies with the random strategy. This means we choose a random node $v \in G$ instead of choosing the first node in an order in each node-moving iteration.

4.2 Moving a Subgraph

Our node-moving approach can easily get stuck in local optima. We can imagine situations, in which it may be good to move a node to a position that increases the overall crossing-number, because this would enable another node to be moved to a better position in a following iteration. This may lead to a globally better solution. In order to approach the global optimum, it may be helpful to move several nodes at once.

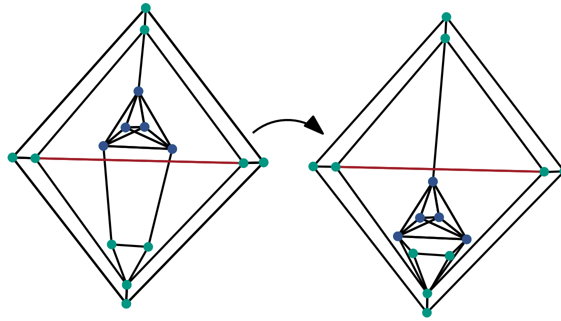


Figure 4.1: This figure depicts how the blue subgraph can be moved at once in order to improve the crossing-number of the drawing.

Figure 4.1 shows a situation, where moving a whole subgraph (the subgraph induced by all blue nodes) at the same time, results in a better crossing number. The red edge is the edge, that is currently reinserted into the graph. Moving all nodes of the blue subgraph to their locally optimal position one by one, would not change anything. This is because all nodes in the blue subgraph are in a locally optimal position. In particular moving any of them across the red edge results in a higher crossing number. Further it can be noticed, that in the example of figure 4.1, moving an arbitrary node to its locally optimal position, does not decrease the crossing number. This shows, that moving a whole subgraph can sometimes improve the node-moving approach. In the following we describe a heuristic for finding and moving a subgraph to a good position after the insertion of an edge.

Moving a Subgraph as Geometric Operation

Let $G_S = (V_S, E_S)$ be a subgraph of G. We call nodes $v \in V$, such that $v \in V_S$, active. We further call edges $e \in E$, such that $e \in E_S$ active. The neighbors $N(G_S)$ in G are all nodes $v \in V$, such that v is adjacent to an active node. We call each node $v \in N(G_S)$ co-active. We call each edge e, that is adjacent to an active node and adjacent to a co-active node, adjacent to the subgraph G_S.

As mentioned before we want to find a heuristic for moving a subgraph to a good position. With a good position of G_S we mean a position, such that the active and co-active edges of G_S have few crossings. The main purpose of this heuristic is to escape from local optima as in figure 4.1. For the purpose of finding a good position for G_S we first need to define the term *contraction*.

The graph G' is the graph resulting from a contraction of G_S to a node s in G. We also call G' *collapsed graph*. A contraction of G_S to a node s in G is defined by the following procedure. Each neighbor $v \in N(G_S)$ is connected with s by adding an edge (s, v) to G. Subsequently each node $v \in V_S$, s.t. $v \neq s$ and its incident edges are removed from G. Let $\mathcal{A}(G', s) = (E_\mathcal{A}, F_\mathcal{A})$ be the arrangement as constructed in section 3.1. Let $f^* \in \mathcal{A}(G', s)$ be the face such that moving s into f^* yields a minimal crossing number.

The idea of our heuristic is to move G_S into f^* by *shifting* and *scaling*. More precisely our heuristic for moving a subgraph G_S can be described in two steps.

- Let $u \in V_S$ be a node of G_S and $p_u \in \mathbb{R}^2$ be the position of u. Let $p_{f^*} \in f^*$ be a point in f^*. The first step consists of moving each node $v \in V_S$ with position p_v to the new position $p_v^{new} = p_v + p_{f^*} - p_u$. Note that the new position of u lies in f^*, because $p_u^{new} = p_{f^*}$. This step is also called shifting G_S.

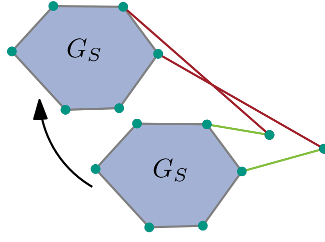


Figure 4.2: This figure shows that moving a subgraph can cause that co-active edges cross each other.

- In a second step we *scale* G_S towards u by the maximum factor $c_{max} \in \mathbb{R}$, such that all nodes $v \in V_S$ are contained in f^* after the scaling. Scaling G_S towards u by a factor $c \in \mathbb{R}$ is defined by the following procedure. The new position p_v^{new} of a node $v \in V_S$ with position p_v is computed by $p_v^{new} = c \cdot (p_v - p_u) + p_u$. Then each node $v \in V_S$ is moved to p_v^{new}.

In the following we discuss if our heuristic moves G_S to an optimal face. This means G_S cannot be moved to a different face by scaling and shifting such that the resulting drawing has a lower number of crossings. The co-active and active edges of G also exist in the collapsed graph G' in contrast to the active edges which are contracted in G'. We observe, that after moving G_S into f^* each co-active edge crosses exactly the same inactive edges as in the collapsed graph G'. If we can move G_S into f^* such that each co-active edge crosses exactly the same edges in G as in G', f^* would be the optimal face for G_S. But this property is not guaranteed when moving G_S into f^* by scaling and shifting as described before. In particular crossings between two co-active edges as in figure 4.3 and crossings between a co-active and an inactive edge as in figure 4.2 can occur. Obviously in the collapsed graph the co-active edges cannot cross each other, because they are all incident to s. Furthermore, a co-active edge in G' can never cross an active edge in G', because there are no active edges in G'.

We try to diminish these crossings in our heuristic GCM by removing and iteratively reinserting all edges $e \in E_S$ and moving their endpoints to their crossing-minimal positions after moving a subgraph.

Crossing-minimal Subgraphs

Until now we have only seen how to move a given subgraph. In this section we propose a heuristic for finding a suitable subgraph that should be moved in order to minimize the crossing number of a straight-line drawing. Let $G = (V, E)$ be a graph with a fixed straight-line drawing. Let $s, t \in V$. We want to find a heuristic for computing a subgraph G_S of G, that we move after inserting an edge $e = (s, t)$. Our heuristic for finding the

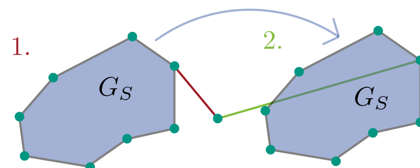


Figure 4.3: This figure shows that moving a subgraph G_S can cause that co-active edges pass through G_S.

subgraph G_S, is related to the *optimal edge insertion problem with a fixed drawing*. The optimal edge insertion problem with a fixed drawing is the problem of finding an open Jordan-curve c from s to t, such that inserting e drawn as c into a drawn planar graph causes a minimum number of crossings.

Gutwenger et al. designed a linear-time algorithm which solves the optimal edge-insertion problem with a planar graph G^* in $\mathcal{O}(|V|+|E|)$ time [GMW05]. Furthermore, they showed that finding an optimal solution to the edge-insertion problem corresponds to computing a shortest path $P = (e_1^d, ..., e_k^d)$ in the extended dual graph $G_D^{s,t}$ of G^*. More precisely, they showed that there is an optimal solution of the edge-insertion problem which crosses exactly the dual edges of $e_1^d, ..., e_n^d$ that are contained in the shortest path P. We denote this solution with $c^*(G^*, e)$. The graph G is not necessarily planar. Therefore, we compute the planarization G_P of G and denote Gutwenger's solution of the edge-insertion problem $c^*(G_P, e)$ with c^* from now on.

Our idea is to move the subgraph G_S of G, that is enclosed by the curve c^* and the line segment $[s,t]$. The thought behind this heuristic is that possibly G_S can be moved across e. This way we hope, that the edge e, which we inserted as the straight line $[s,t]$, crosses exactly the edges , that c^* crossed before moving G_S. This would be optimal, because c^* causes a minimum number of crossings. To put it differently we want e to be drawn as c^*. But because e is a straight-line, we want to move the subgraph G_S enclosed between c^* and the straight-line $e = [s,t]$ such that it might clear the space for $[s,t]$.

The closed curve $c^* \cup e$ parts $\mathbb{R}^2$ into two subsets. The points inside of $c^* \cup e$ form a subset A_1. The curve $c^* \cup e$ is not necessarily a simple curve, see figure 4.5 for an example. Nevertheless, its inside is the set of all points p, such that an arbitrary ray starting at p crosses $c^* \cup e$ an odd number of times. The points outside of $c^* \cup e$ form the other subset A_2. Furthermore, the closed curve $c^* \cup e$ parts $V \backslash \{s,t\}$ into two sets of nodes, the set of nodes N_1 with a position in A_1 and the set of nodes N_2 with a position in A_2.

We want to compute N_1 and N_2. In the following we describe an algorithm which computes N_1 and N_2 in three steps.

1. Compute the planarization G_P of G.
2. Find the *cut edges* of G. A cut edge of graph G is an edge which crosses $c^* \cup e$ an odd number of times.
3. Perform an adapted breadth-first search in order to identify N_1 and N_2.

moving the subgraph G_S enclosed by c^* and e

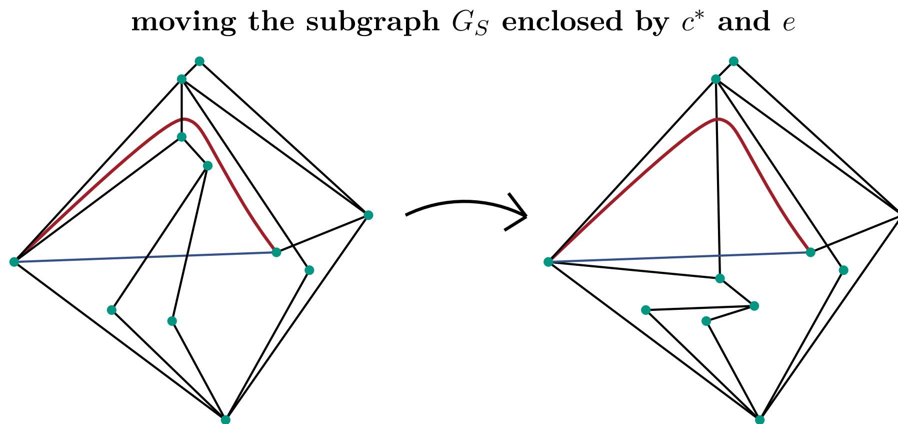


Figure 4.4: The optimal Jordan-Curve c^* for e is drawn red. The newly inserted edge e is drawn blue. Note that in this case after moving G_S the co-active edges of G_S cross exactly the same edges as c^*.

moving the subgraph G_S enclosed by c^* and e

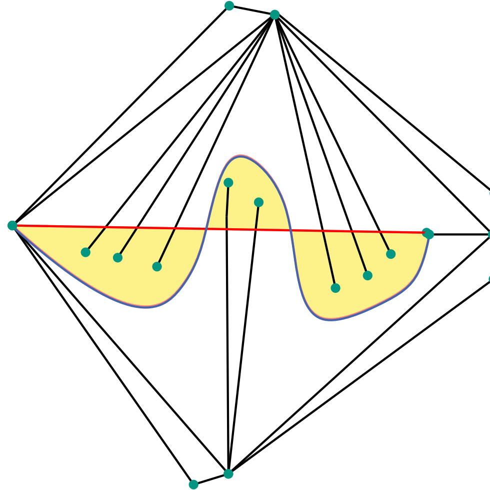


Figure 4.5: The optimal Jordan-curve c^* for e is drawn red. The newly inserted edge e is drawn blue. Note that in this case after moving G_S the co-active edges of G_S cross exactly the same edges as c^*.

Computing the planarization G_P of the drawn graph G is bounded in running time by $\mathcal{O}(m \cdot \log(m) + k)$ where k is the number of crossings in the drawing of G.

Finding the Cut Edges

For the purpose of computing N_1 and N_2 it is helpful to identify the cut edges. As described earlier a cut edge of graph G is an edge which crosses $c^* \cup e$ an odd number of times. Note that the cut edges of G can be found in $\mathcal{O}(m^* + n^*)$-time, where m^* is the number of edges and n^* the number of nodes in the planarization G_P of G. As described earlier Gutwenger et al. showed that the edges e_P of G_P which cross c^* can be found in linear time $\mathcal{O}(n^* + m^*)$. In order to determine the cut edges of G, we store a bit $\texttt{state}(\overline{e}) \in \{0, 1\}$ for each edge $\overline{e}$ of G. $\texttt{state}(\overline{e})$ is initialized to 0 for all edges $\overline{e}$ of G. We iterate over all edges e_P of G_P and flip $\texttt{state}(\overline{e})$ each time e_P is contained in the corresponding path of $\overline{e}$ in G_P and crosses $c^* \cup e$, i.e. is contained in the shortest path P. Subsequently we iterate over all edges $\overline{e}$ of G and flip $\texttt{state}(\overline{e})$ if $\overline{e}$ crosses e. As a result all edges $\overline{e}$ of G with $\texttt{state}(\overline{e}) = 1$ are cut edges.

But there is another possibility how the closed curve $c^* \cup e$ can be passed by traversing edges of G. $c^* \cup e$ can also be passed by a path of length two which passes s or t. In order to be able to identify these cases in a breadth-first search we add some additional edges to G. Let f_s be the face which corresponds to the first node $v \neq s$ on the shortest path P in the extended dual graph. Let f_t be the face which corresponds to the last node $v \neq t$ on the shortest path P in the extended dual graph. We choose two points $v_s \in f_s, v_t \in f_t$ and add the edges (s, v_s) and (t, v_t) to G. We mark these edges as *rim edges*. Furthermore we mark the edge e as rim edge.

Adapted Breadth-First Search

Given that we have identified the cut edges we can describe an adapted breadth-first search which determines N_1 and N_2. In particular the following breadth-first search iteratively computes a function $\mu : V \backslash \{s, t\} \to \{0, 1\}$ with the property $\mu(v) = \mu(w)$ if and only if

$u \in N_1$ and $w \in N_1$ or $u \in N_2$ and $v \in N_2$. The breadth-first search starts at a node $v \in V \setminus \{s, t\}$ but traverses only edges that are not rim edges. Let x be the current node in the breadth-first search and let $p[x]$ be the parent node of x.

Case 1: *$p[x] \notin \{s, t\}$ and $(p[x], x)$ is not a cut edge*
In this case, we set $\mu(x) := \mu(p[x])$ since either x and $p(x)$ are outside of $c^* \cup e$ or x and $p[x]$ are inside of $c^* \cup e$.

Case 2: *$p[x] \notin \{s, t\}$ and $(p[x], x)$ is a cut edge*
In this case, we set $\mu(x) := 1 - \mu(p[x])$, because x and $p[x]$ are on different sides of $c^* \cup e$.

Case 3: $p[x] \in \{s, t\}$
Without loss of generality we assume that $p[x] = s$. This case is a bit more difficult. We have to distinguish several sub-cases which depend on whether (s, x) and $(p[s], s)$ are both not a cut edge, both not a cut edge or exactly one of them is a cut edge. Furthermore, they depend on if the path $(p[s], s, x)$ only touches or crosses $c^* \cup e$. Figure 4.6 shows a generic case for each case in which we set $\mu(x) := \mu(p[s])$ and figure 4.7 shows a generic case for each case in which we set $\mu(x) := 1 - \mu(p[s])$.

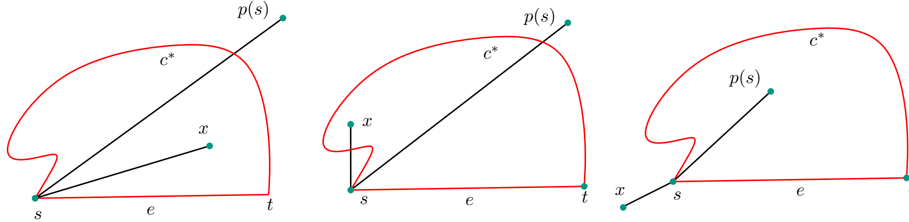


Figure 4.6: Cases in which we set $\mu(x) := \mu(p[s])$ in the adapted breadth-first search for computing N_1 and N_2

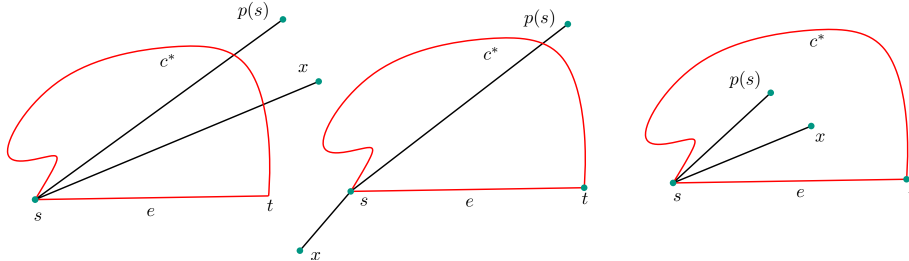


Figure 4.7: Cases in which we set $\mu(x) := 1 - \mu(p[s])$ in the adapted breadth-first search for computing N_1 and N_2

This adapted breadth-first search has a running time of $\mathcal{O}(m)$ if the cut edges are already determined and all cases can be distinguished in a time of $\mathcal{O}(1)$. This is because we can decide whether the path $(p[s], s, x)$ touches or crosses $c^* \cup e$ in the course of the breadth-first search in $\mathcal{O}(1)$ time. A breadth-first search discovers the adjacent edges of the current node in every iteration. We assume that our breadth-first search discovers these edges in a cyclic order around the current node starting with the parent edge. Therefore, we can store a bit `edgestate` which is flipped every time the breadth-first search discovers a rim edge. The bit `edgestate` then indicates whether there is a rim edge between the current edge and the parent edge. In summary computing μ is possible in an $\mathcal{O}(m \cdot \log(m) + m^*)$ time-bound.

The function μ parts $V \backslash \{s,t\}$ into two components N_1 and N_2. But we do not know which of these components is N_1 and which is N_2. Nevertheless, we want to move the subgraph induced by N_1 or the subgraph induced by $N_2 \cup \{s,t\}$. Accordingly we choose one of the components N_i and try to move the subgraph induced by N_i and the subgraph induced by $N_i \cup \{s,t\}$.

4.3 PrEd

After a few edge iterations of the Gcm algorithm, the ratio between the different edge lengths in G becomes high and some nodes are positioned very close to each other. This effect is even strengthened if Gcm moves subgraphs, because in this case whole subgraphs are moved into the face of an arrangement. Due to this problem we use a force directed method, called `PrEd`, in each edge iteration of Gcm if contraction of subgraphs is used or many nodes are moved in one edge iteration. This is the case if one of the node orders High-to-Low, Low-to-High or Random is employed.

In the following we shortly explain the algorithm `PrEd` by Bertault which is a force-directed drawing algorithm that preserves crossings [Ber99]. More precisely if two edges cross in the initial drawing, they also cross in the final drawing produced by `PrEd`. This means `PrEd` does not improve the rectilinear crossing number of the final drawing produced by Gcm. It only improves the drawing in symmetry and uniformity of edge lengths.

`PrEd` is a force-directed drawing algorithm, which means forces are iteratively applied to the position of nodes. As it is common among force directed algorithms, `PrEd` applies attractive forces between nodes which are connected by edges, repellent forces between nodes and repellent forces between nodes and edges. In order to preserve the crossings of the drawing the amplitudes of a node movement are restricted. In order to realize this restriction, `PrEd` identifies eight zones for each node $v \in V$, such that v can be moved inside of the zone without changing the crossings of the drawing. The zones $Z_v^1, ..., Z_v^8$ are sectors of circles with midpoint v and the same central angle, but different radii $R_v^1, ..., R_v^8$ emanating from v. When forces are applied the direction of the force and, therefore, the sector Z_v^i is determined which includes this direction. The amplitude of the force is then bounded by the radius R_v^i. A detailed description on how the different radii are computed can be found in [Ber99].

4.4 Edge Order Post-processing

The idea of the edge-order post-processing is to iteratively delete and reinsert a set of edges subsequent to the Gcm algorithm. A post-processing strategy varies in the number and order of edges, which are deleted and reinserted into the graph. Similarly to the node-order strategies, we want to compute an edge order based on geometric criteria. One idea is to order the edges according to the number of crossings they cause from low to high. We call this order *Low-to-High* order. Similar to the node-order strategies, another idea is to order edges from high to low. This strategy is called *High-to-Low*. For practical reasons we delete and reinsert only up to 10 edges.

5. Evaluation and Experiments

Rectilinear crossing minimization is a very difficult problem. In fact rectilinear crossing minimization is NP-hard and $\exists\mathbb{R}$-complete. We are not able to show any theoretical performance guarantees for our heuristic GCM. However, we evaluate GCM on a variety of different graph classes with several statistical methods to be able to rank its performance in comparison to other rectilinear drawing algorithms.

In section 5.1 we describe the experimental setup. Thereupon, in section 5.2 we precisely describe the conducted statistical experiments and evaluate their results.

5.1 Experimental Setup

In this section we describe the experimental setup. More precisely, we characterize the considered graph classes, the comparison algorithm and determine some fixed parameters of GCM. Subsequently we describe the considered configurations which emanate from the remaining open parameters of GCM.

All experiments were conducted on an AMD Opteron Processor 6172 with four 12-cores clocked at 2.1 GHz. All algorithms were compiled with `g++` version 4.8.5 with compile flags `-std=c++11 -DNDEBUG -O3`.

5.1.1 Graph Classes

We conduct our experiments on three sets of graphs in order to cover a variety of non-planar graphs, with different properties. GCM reinserts a number of edges into a maximally planar subgraph G_P of a graph G. This number ε of edges is a good approximation for the degree of non-planarity of G. In order to evaluate GCM, we want to consider graphs with different degrees of non-planarity. We will now first present the different graph classes and some of their theoretical properties. Subsequently we will describe the concrete benchmark sets obtained from these graph classes.

Our first idea is to generate planar graphs and add edges uniformly at random. Since GCM minimizes crossings, we want these edges to cause crossings. For this reason we want to generate *triangulated* graphs. A triangulated graph is a planar graph, if adding an arbitrary additional edge results in a non-planar graph. Therefore, these graphs are also called maximal planar. In particular we generate all planar triangulated graphs with 64 nodes with the tool-kit `plantri` [BM+07]. Each of these graphs has the same edge number,

because $m = 3 \cdot n - 6$ holds for triangulated graphs, where n is the node number and m the edge number of G. We add 10 edges uniformly at random to each graph, thus they all have the same density. We refer to these graphs as *Plantri graphs* throughout this chapter.

The *Rome graphs*[1] data set is a set of 11.534 graphs with a node number between 10 and 100. *Gutwenger et al.* used this data set to evaluate their edge insertion algorithm [GMW05].

The *NetworKit graphs* data-set is motivated by the aim to evaluate GCM on graphs with a *community structure*. The term community structure is not a mathematically precise notion. But a graph with a community structure is said to have similar properties as a real-world network. More precisely, a graph with a community structure has communities with different sizes and a heterogeneous distribution of node degrees. A community is a set of nodes which is internally densely connected by edges. In order to generate the NetworKit graphs, we use the LFR-GENERATOR according to [LFR08] implemented in NETWORKIT [SSM14]. For each of the NetworKit graphs the number n of nodes is 100.

From all these sets of graphs, we choose 100 graphs uniformly at random and refer to each of these random subsets as *Plantri-100*, *Rome-100* and *NetworKit-100*.

Plantri graphs

Figure 5.1 shows the distribution of the number ε of deleted edges on the graphs Plantri-100. Because the Plantri graphs are triangulated graphs, to which we added random edges, none of these graphs is planar. Moreover, in comparison to the other graph classes the number of deleted edges ε is rather high. Note, that as explained in Section 5.1.3 we use a method that does not necessarily identify a maximum planar subgraph, otherwise ε would be exactly 10 for each Plantri graph. The graph size $m + n$ of each graph contained in the Plantri-100 is 196.

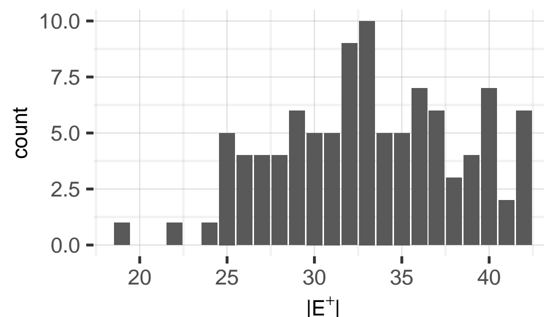


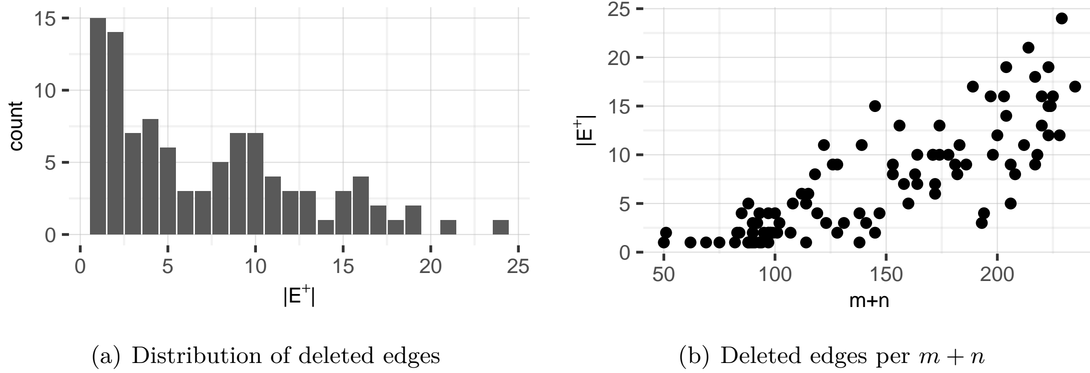
Figure 5.1: Distribution of deleted edges on Plantri-100

Rome graphs

Figure 5.2(a) shows, that there are at least 34 planar graphs in our Rome-100 graphs. Moreover, the highest number of inserted edges is 18. On average the Rome-100 have a much lower number of inserted edges than the Plantri-100. Figure 5.2(b) depicts the size of a graph against its number of deleted edges ε. On the Rome-100 there is an approximately

[1] The Rome graphs data-set is currently available via http://www.dia.uniroma3.it/people/gdb/wp12/undirected-1.tar.gz

linear relationship between the graph size $m + n$ and ε. The Rome-100 contain graphs of a size from 18 to 272. There are 87 different graph sizes among these graphs.



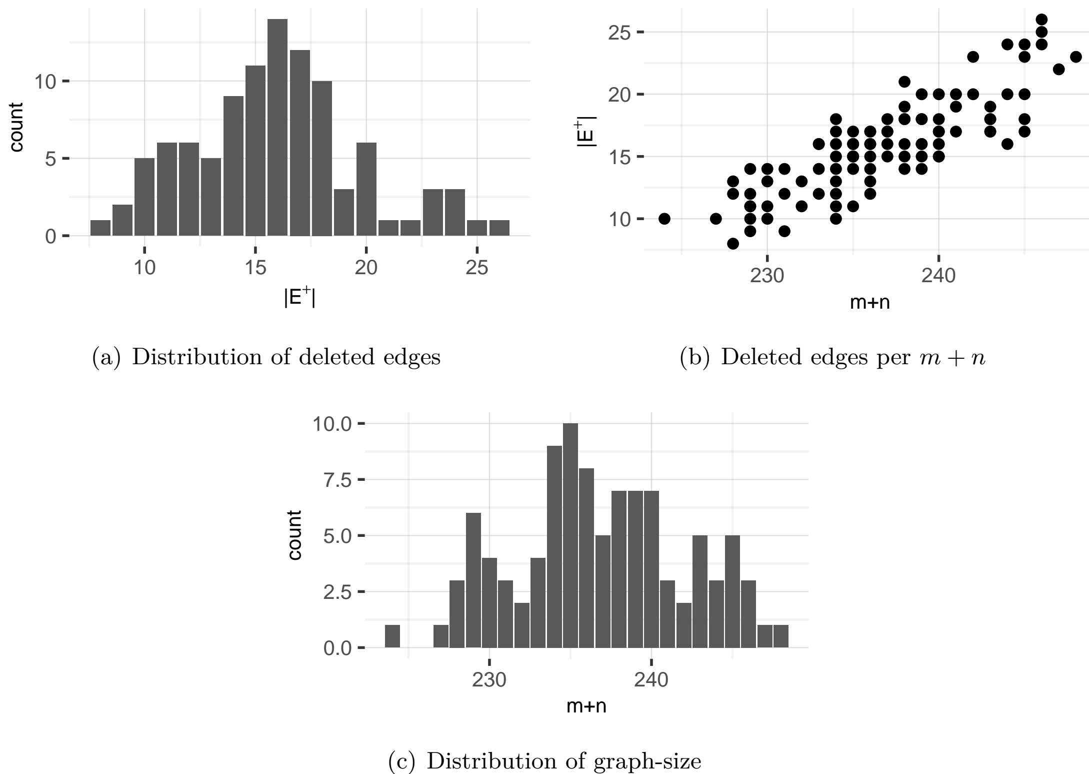
(a) Distribution of deleted edges

(b) Deleted edges per $m + n$

Figure 5.2: Rome-100

NetworKit graphs

Figure 5.3(a) shows the distribution of ε on the NetworKit-100. The range of ε reaches from 8 to 27 and the average ε is higher than on the Rome-100 but lower than on the Plantri-100. The graph size $m + n$ on the NetworKit-100 reaches from 224 to 248 as can be seen in figure 5.3(c). The average graph size of the NetworKit-100 is higher than on the other graph sets even though the average ε is lower than on the Plantri-100. Again the relationship between the graph size $m + n$ and ε can be seen to be linearly increasing in figure 5.3(b).



(a) Distribution of deleted edges

(b) Deleted edges per $m + n$

(c) Distribution of graph-size

Figure 5.3: NetworKit-100

5.1.2 Comparison Algorithm

Our main comparison algorithm is the spring embedder by Fruchterman-Reingold [FR91], which we call SpringEmbedder from now on. The SpringEmbedder belongs to the family of force-directed drawing algorithms. Force-directed drawing algorithms are relatively fast and produce drawings, that are aesthetically pleasing and have a low crossing number. Moreover, they are commonly used in practical graph-drawing applications. This makes these drawing algorithms good comparison algorithms for Gcm.

The most common force-directed drawing algorithms are the spring embedders by Fruchterman and Reingold [FR91], Kamada-Kaway [KK89] and the simulated annealing approaches by Tutte [Tut63], Davidson-Harrel [DH96] and stress minimization by Gansner et al. [GKN05] all of which are implemented in OGDF. We compared all of these implementations against each other on the Rome-100 benchmark set with respect to the produced crossing number. Both spring embedders require an initial drawing and both produce the lowest crossing numbers with stress minimization as their initial drawing. By comparing all the force-directed methods implemented in OGDF, we find that SpringEmbedder produces the lowest amount of crossings on average. This justifies, that we choose SpringEmbedder as our main comparison algorithm.

5.1.3 Parameters

In this section we describe the configurations of Gcm, which we consider in our experiments. As explained in section 3, Gcm uses a two-phase approach. In the first phase a planar subgraph is extracted and this subgraph is drawn. Subsequently the edge-insertion step is performed. Our major contribution to this approach is the edge-insertion step. For the first step we use common algorithms for finding a planar subgraph and draw it in the plane. In particular there are two algorithms we have to choose for this step:

1. a heuristic for computing a maximum planar subgraph $G^* = (V^*, E^*)$.
2. a planar drawing-algorithm for finding an initial drawing of G^* in the plane.

Fur the purpose of computing the planar subgraph G^*, we restrict ourselves to algorithms implemented in OGDF [CGJ+11]. There are three algorithms for computing G^* implemented in OGDF.

1. FastPlanarSubgraph
2. MaximumPlanarSubgraph
3. MaximalPlanarSubgraphSimple

We choose the FastPlanarSubgraph algorithm, since it produces rectilinear drawings with less crossings than MaximalPlanarSubgraphSimple and equally many crossings as MaximumPlanarSubgraph on average on the Rome-100, but needs less time. Note, that FastPlanarSubgraph is only a heuristic and does not necessarily identify the maximum planar subgraph of a graph G.

We choose PlanarStraightLayout as initial layout, because the SpringEmbedder achieves the best results on the Rome-100 with this initial layout. As already mentioned we want to compare our algorithm Gcm to SpringEmbedder.

There are three free parameters of Gcm:

1. the *node order* in which nodes are moved after each edge-reinsertion. The different node-ordering strategies were already discussed in Section 4.1. The node order can be chosen from the following four node ordering strategies: *High-to-Low*, *Low-to-High*, *Endpoints* and *Random*.

2. whether or not a *subgraph* is moved to a good position after each edge-reinsertion. A detailed description of this strategy to find and move a subgraph can be found in 4.2. This means subgraph strategy is a boolean parameter and can only take the values *true* and *false* which indicates whether the strategy is used or not.
3. the different *edge-order post-processing strategies* as discussed in Section 4.4. Subsequently, the edge-order post-processing strategy can be chosen from: *High-to-Low*, *Low-to-High* or *None*.

5.1.4 Configurations

Because of the high number of free parameters we want to find the most promising node order before our final evaluation.

In order to choose a suitable node order we compare all node-ordering strategies on the Rome-100 benchmark set. For this purpose, we choose the following set of configurations and compare the average crossing number of their resulting drawings on the Rome-100 graph set:

	NodeOrder	**Subgraph**	**EdgeOrder**
Gcm-Low-NodeOrder	Low-to-High+EP	false	None
Gcm-High-NodeOrder	High-to-Low+EP	false	None
Gcm-Random-NodeOrder	Random+EP	false	None
Gcm-Base-NodeOrder	Endpoints	false	None

Table 5.1: NodeOrder Configurations

.

We shortly recapitulate the different node-ordering strategies as explained in section 4.1. The strategy *Endpoints* only moves the endpoints of a newly inserted edge e. Whereas *High-to-Low+EP* and *Low-to-High+EP* first move both endpoints of e. Afterwards both strategies order the nodes according to the number of adjacent edges that cross e from high to low and respectively from low to high. Subsequently nodes are moved according to this order until there are no more nodes with adjacent edges crossing e, or 10 nodes have been moved. *Random+EP* first moves both endpoints and subsequently moves 10 nodes uniformly at random.

Table 5.1.4 shows, that Gcm-High-NodeOrder performs best in terms of the average crossing number on Rome-100. Given that the Gcm-Base-NodeOrder strategy moves only two nodes, whereas the other node-ordering strategies move up to 12 nodes, it still performs suprisingly well on the Rome-100 graph set. Moreover, we expect Gcm-Base-NodeOrder to have the best running time among the node-ordering strategies. Therefore, we restrict ourselves to the node order High-to-Low+EP and the basic node order Endpoints.

Configuration	**avg. crossing-number**
Gcm-Low-Nodeorder	11.67
Gcm-High-Nodeorder	11.62
Gcm-Random-Nodeorder	13.42
Gcm-Base-Nodeorder	14.74

Table 5.2: NodeOrder Configurations

This leaves us with the following set of configurations:

	NodeOrder	**Subgraph**	**EdgeOrder**
Gcm-Contract	Endpoints	true	None
Gcm-Contract-High	Endpoints	false	None
Gcm-Base	High-to-Low+EP	true	None
Gcm-Base-High	High-to-Low+EP	false	None

Table 5.3: Final Configurations

5.2 Evaluation

In this section we evaluate our configurations of Gcm. In subsection 5.2.1 we choose two statistical tests that fit our main purposes for this evaluation. Subsequently, we compare our configurations of Gcm pairwise against each other and SpringEmbedder on all three graph classes in subsection 5.2.2. To this end we perform the previously chosen statistical test in an adapted form. The chosen statistical procedure was first used by Radermacher [Rad15]. Furthermore, we test the hypothesis that the difference in crossing number between a drawing of SpringEmbedder and Gcm of the graph G is positively correlated with the density of G in section 5.2.4. In the following sections 5.2.3 and 5.2.5 we try to estimate whether the solutions of Gcm are close to optimal. We do not know the rectilinear crossing number for graphs contained in one of our three graph classes. We neither have a non-asymptotic approximation of the rectilinear crossing number. Therefore, we compare the best of our configurations of Gcm to Gutwenger's edge insertion algorithm in 5.2.3. For the complete graphs $\mathcal{K}_n$ with $n \leq 100$ we have very close bounds for the rectilinear crossing number. In section 5.2.5 we, therefore, test our heuristic on the complete graphs $\mathcal{K}_n$ with $n \leq 30$. Finally, we compare the running time of our configurations in 5.3.

5.2.1 Statistical Tests

Our evaluation mainly focuses on comparing the resulting drawings of two algorithms on the same input graph. More precisely, we would like to decide whether one algorithm produces straight-line drawings with a significantly lower crossing number on a class of graphs than a second algorithm. There are different measures, which can be compared when deciding whether one set of straight-line drawings has less crossings than another. The first measures, that come to mind are the mean or the variance of both test sets. However, we want to guarantee that one of the algorithms is significantly better on a large ratio of these input graphs. Both, the mean value and the variance can be strongly influenced by outliers. A more stable measure is whether or not one algorithm outperforms the other algorithm on more than 50% of the samples.

For the purpose of testing whether the resulting drawings of two algorithm differ with statistical significance on a certain graph class, we use the method of hypothesis testing. The book Handbook of parametric and nonparametric statistical procedures by David J. Sheskin [She07] gives an overview on hypothesis testing. The book [She07] recommends the test *The Binomial Sign Test for Two Dependent Samples* for our purpose, because the distribution of the crossing number on arbitrary straight-line drawings is not known.

The second part of our evaluation focuses on measuring the correlation between different random variables. In particular our aim is to measure the correlation between a certain property of the input-graphs and the performance of the Geometric Crossing Minimization algorithm 4.1. In statistical terms we want to measure the correlation between two random variables X and Y on the same set of samples.

The book [She07] recommends three tests for this purpose.

1. *Spearman's Rank-Order Correlation Coefficient*
2. *Kendall's Tau*
3. *Goodman and Kruskal's Gamma*

We choose Spearman's rank-order correlation coefficient for our intention, because it is the most commonly used test among the three. Spearman's rank-order correlation coefficient is designed to detect a significant correlation between both variables. More precisely, it approximates the correlation ρ between X and Y by a coefficient r and decides whether there is a significant correlation between the two variables based on the approximation r.

In the following we will apply the binomial sign test and spearman's correlation coefficient on the graph classes Plantri, NetworKit and Rome with the sample sets Plantri-100, NetworKit-100, Rome-100. The only assumption which has to be fulfilled in order to apply the binomial sign test and spearman's correlation coefficient is, that the sample set is chosen uniformly at random from the underlying class. This assumption is clearly fulfilled, because each of the sets Plantri-100, NetworKit-100 and Rome-100 is chosen uniformly at random from the respective graph class.

5.2.2 Pairwise Comparison of Configurations

In this section we compare our chosen configurations and the SPRINGEMBEDDER pairwise against each other based on the crossing number of their drawings on the graph sets *Plantri-100*, *NetworKit-100* and *Rome-100*. For this purpose we perform the adapted binomial sign test as explained in section 5.2.1 for these pairs of configurations on each graph set respectively. More precisely, in order to find the maximum factor δ_{max}, such that A_1 outperforms A_2 by the factor δ_{max} we proceed as follows.

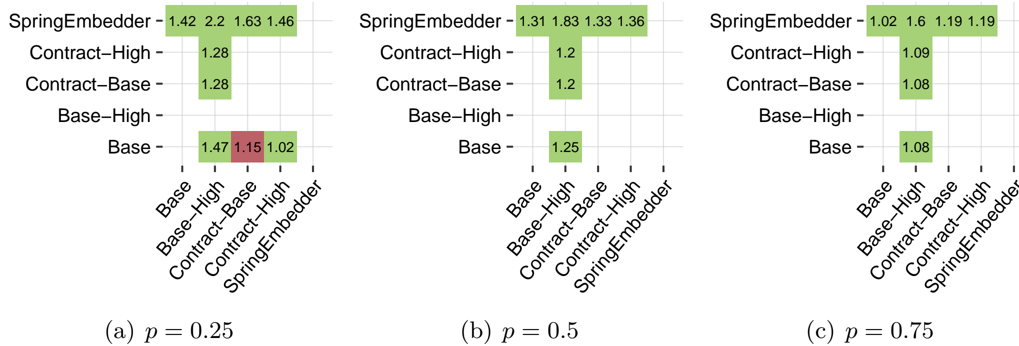
The set of sample graphs $\mathcal{G}_S$ is divided into a set of training graphs $\mathcal{G}_{Training}$ and a set of test graphs $\mathcal{G}_{Test}$. In particular we randomly assign 20% of the sample graphs to $\mathcal{G}_{Test}$ and 80% of the sample graphs to $\mathcal{G}_{Training}$. On the set $\mathcal{G}_{Training}$ we compute the maximum factor δ_{max} such that the hypothesis $H^p_{\delta_{max}}$ holds with a binary search. Subsequently we compute $\delta_{Test} = \frac{3}{4} \cdot \delta_{max}$ and test the hypothesis $H^p_{\delta_{Test}}$ on $\mathcal{G}_{Test}$. If this hypothesis is accepted by the test $\text{Bin}(\mathcal{G}_{Test}, p, 0.05)$, we can be sure that algorithm A_1 outperforms algorithm A_2 on the graph class $\mathcal{G}$ by a multiplicative factor of δ_{Test} with statistical significance. This procedure is done for all $p \in \{0.25, 0.5, 0.75\}$ such that we get a factor δ_{Test} for each of these values of p.

This way we obtain three matrices for each graph set. For example figure 5.4 shows the three matrices associated with the graph set *Plantri-100*. Each of them contains the resulting factors δ_{Test} of an adapted binomial sign test on all pairs of algorithms employed with probability $p = 0.25$, $p = 0.5$ and $p = 0.75$ respectively. More precisely the value in column *algorithm A* and row *algorithm B* of one of these matrices represents the factor δ_{Test} which is the result of a an adapted binomial sign test employed with probability p. A red cell reports, that the hypothesis $H^p_{\delta_{Test}}$ is rejected on the test set $\mathcal{G}_{Test}$ whereas a green cell represents the hypothesis $H^p_{\delta_{Test}}$ being accepted on $\mathcal{G}_{Test}$. An empty cell reports that there cannot be found a factor $\delta_{max} \geq 1$ such that A_1 outperforms A_2 by the factor δ_{max} on the training set $\mathcal{G}_{Training}$.

Plantri

Figure 5.4 shows the resulting matrices on the graph class Plantri. It can be seen, that with an increasing p the factors tend to decrease. We first take a closer look at figure 5.4(a).

The matrix shows, that all chosen configurations outperform the SpringEmbedder by a factor of at least 1.42 and the SpringEmbedder outperforms none of our configurations. Moreover, Gcm-Base-High seems to outperform all other configurations. Furthermore we recognize, that Gcm-Contract and Gcm-Contract-High outperform Gcm-Base but with a lower factor than Gcm-Base-High. This trend is confirmed in 5.4(b) and 5.4(c), although there is no significant difference between Gcm-Base-Contract and Gcm-Base if $p = 0.5$. Nevertheless, if $p = 0.5$ Gcm-Base outperforms SpringEmbedder by a factor of 1.42 and Gcm-Contract-Base outperforms SpringEmbedder by a factor of 1.63. It is remarkable, that Gcm-Base-High outperforms SpringEmbedder on a ratio of at least 0.5 of the Plantri graphs by a factor of 1.83 and by a factor of 1.6 on a ratio of at least 0.75 of the Plantri graphs.

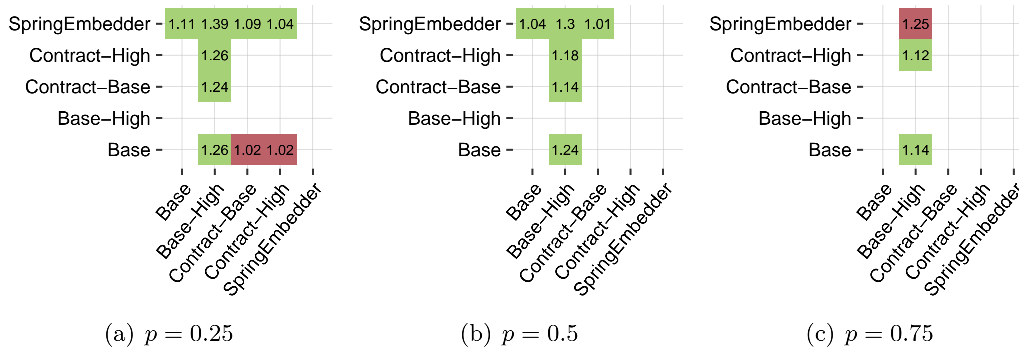


(a) $p = 0.25$ (b) $p = 0.5$ (c) $p = 0.75$

Figure 5.4: Binomial Test on Plantri

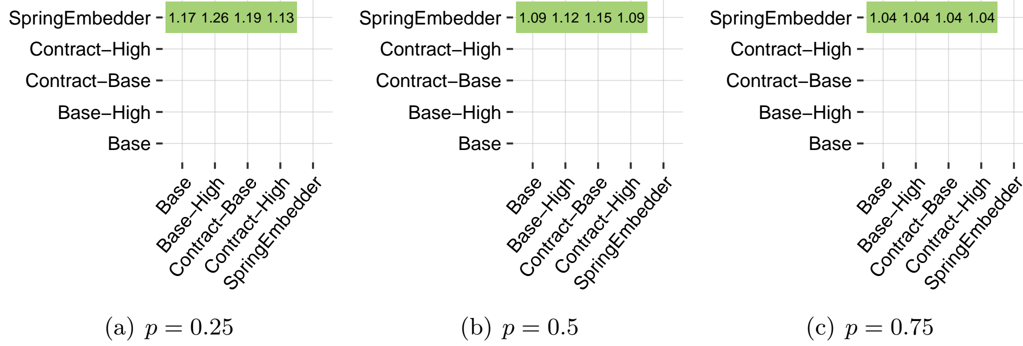
NetworKit

Figure 5.5 shows the matrices for the graph class NetworKit. Overall on the NetworKit graphs our configurations do not perform as good as on the Plantri graphs compared to SpringEmbedder. In figure 5.5(b) we can only find a significant factor of 1.04 for the Gcm-Base configuration and a factor of 1.3 for Gcm-Base-High. Our hypothesis was, that our configurations perform better on dense graphs in comparison to SpringEmbedder. This hypothesis will be evaluated in the next section 5.2.3. Again also on the NetworKit graphs Gcm-Base-High seems to be the configuration, that performs best.



(a) $p = 0.25$ (b) $p = 0.5$ (c) $p = 0.75$

Figure 5.5: Binomial Test on NetworKit



(a) $p = 0.25$ (b) $p = 0.5$ (c) $p = 0.75$

Figure 5.6: Binomial Test on Rome

Rome

Figure 5.6 shows, that on the Rome graphs GCM-BASE-HIGH and GCM-CONTRACT-BASE outperform SPRINGEMBEDDER by factors of 1.12 and 1.15, whereas GCM-BASE and GCM-CONTRACT-HIGH only outperform SPRINGEMBEDDER by lower factors. The configurations among each other show no significant factors. One reason for this could be, that there are 30% planar graphs in our Rome-100 benchmark-set, which are all drawn planar by our configurations and therefore have 0 crossings.

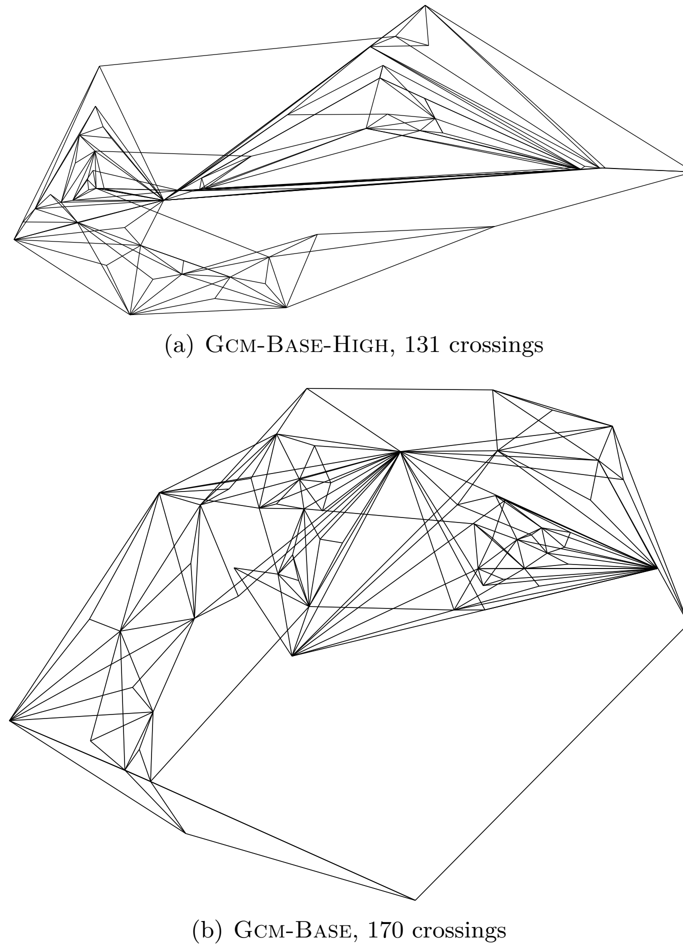
From this pairwise comparison of our configurations we conclude, that GCM-BASE-HIGH is the most promising configuration. Therefore, we conduct the following tests only on the best configuration GCM* = GCM-BASE-HIGH. Figure 5.7 shows the drawings of the same Plantri graph by GCM-BASE-HIGH and GCM-BASE. It can be seen, that there are nodes in the drawing of GCM-BASE which are not in their locally optimal position. On this graph is makes sense to move more nodes to their locally optimal position.

5.2.3 Comparison to Edge Insertion

Gutwenger's edge-insertion algorithm, named GUTWENGER from now on, is a heuristic for finding a crossing-minimal drawing $\mathcal{D}$ of a graph G. The drawing $\mathcal{D}$ is not necessarily straight-line in this case. This means the edges of G can be represented by arbitrary topological Jordan-Curves.

The idea of Gutwenger is an edge-reinsertion strategy. In a first step a planar subgraph G^* is extracted from G. Subsequently the remaining edges are reinserted into G^* such that the crossing number stays low. The step is called edge-reinsertion step. Gutwenger's edge-reinsertion step iterates over all possible drawings $\mathcal{D}^*$ using an $\mathcal{SPQR}$-tree and inserts the current edge e optimally into $\mathcal{D}^*$ by computing the shortest path in the dual graph of G. A more detailed summary on Gutwenger's edge-insertion algorithm can be found in section 1.1.

The study of Gutwenger and Mutzel [GM03] compares different configurations of Gutwenger's edge-insertion algorithm. According to this study the resulting drawing of GUTWENGER can be remarkably improved by post-processing, that deletes and reinserts a part of the edges after each edge-reinsertion. Furthermore, the whole process of deleting and reinserting edges into G^* can be repeated multiple times with different permutations of the edge order. The permutation variant of GUTWENGER keeps the best result of these different permutations. According to the experiments in [GM03] PERM-20 is the best configuration among the different permutation variants.

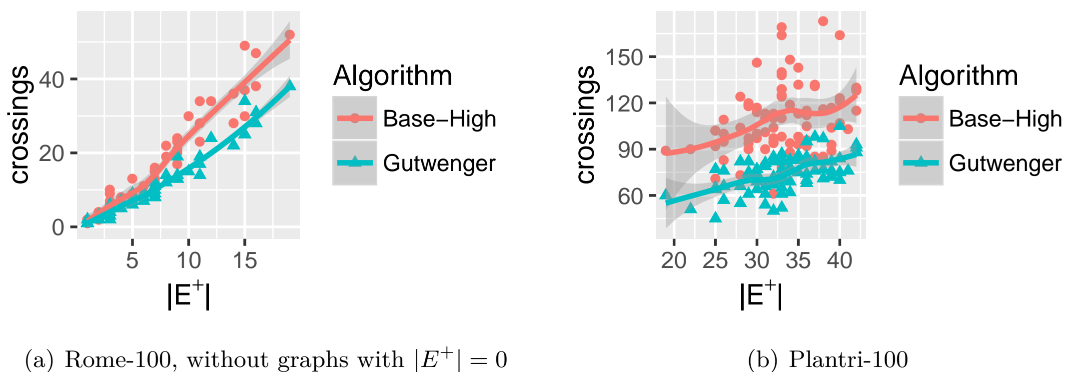


(a) GCM-BASE-HIGH, 131 crossings

(b) GCM-BASE, 170 crossings

Figure 5.7: The drawings GCM-BASE-HIGH and GCM-BASE on the same Plantri graph

As comparison algorithm, we choose GUTWENGER with PERM-20 and with the edge-reinsertion strategy RRMOSTCROSSED, which reinserts edges with the most crossings first. We use the implementation of GUTWENGER in OGDF [CGJ+11]. In particular we use the SUBGRAPHPLANARIZER with 20 permutations together with the VARIABLEEMBEDDINGINSERTER with the REMOVEREINSERT-option RRMOSTCROSSED and FASTPLANARSUBGRAPH.



(a) Rome-100, without graphs with $|E^+| = 0$

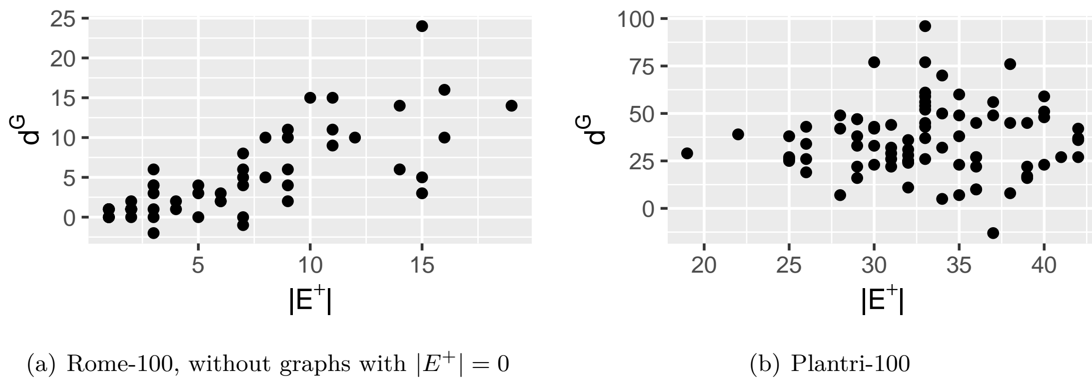
(b) Plantri-100

Figure 5.8: GUTWENGER vs. GCM-HIGH

In this section we compare Gutwenger to Gcm* = Gcm-Base-High. For this purpose we consider the difference $d^G = \overline{\text{cr}}_{\text{Gcm}^*} - \overline{\text{cr}}_{\text{Gutwenger}}$, where $\overline{\text{cr}}^G_{\text{Gcm}^*}$ is the crossing number of the straight-line drawing of G produced by Gcm*. Analogously $\overline{\text{cr}}_{\text{Gutwenger}}$ is the crossing number of the straight-line drawing of G produced by Gutwenger.

Because Gutwenger is a heuristic for topological crossing minimization we expect Gcm* to produce more crossings. We consider Gutwenger more as a baseline which we would like to reach. Figure 5.8(a) shows, that on the Rome-100 Gutwenger and Gcm* produce drawings with almost the same amount of crossings and figure 5.9 shows, that for most of the graphs the difference d^G is below 10 and for a few graphs d^G is negative.

On the Plantri-100 the average difference d^G is approximately 30, as can be seen in 5.8(b). Figure 5.9(b) shows that for the majority of graphs the difference d^G is below 50 and for one graph d^G is negative.



(a) Rome-100, without graphs with $|E^+| = 0$

(b) Plantri-100

Figure 5.9: difference in crossing number between Gcm-High and Gutwenger

5.2.4 Correlation

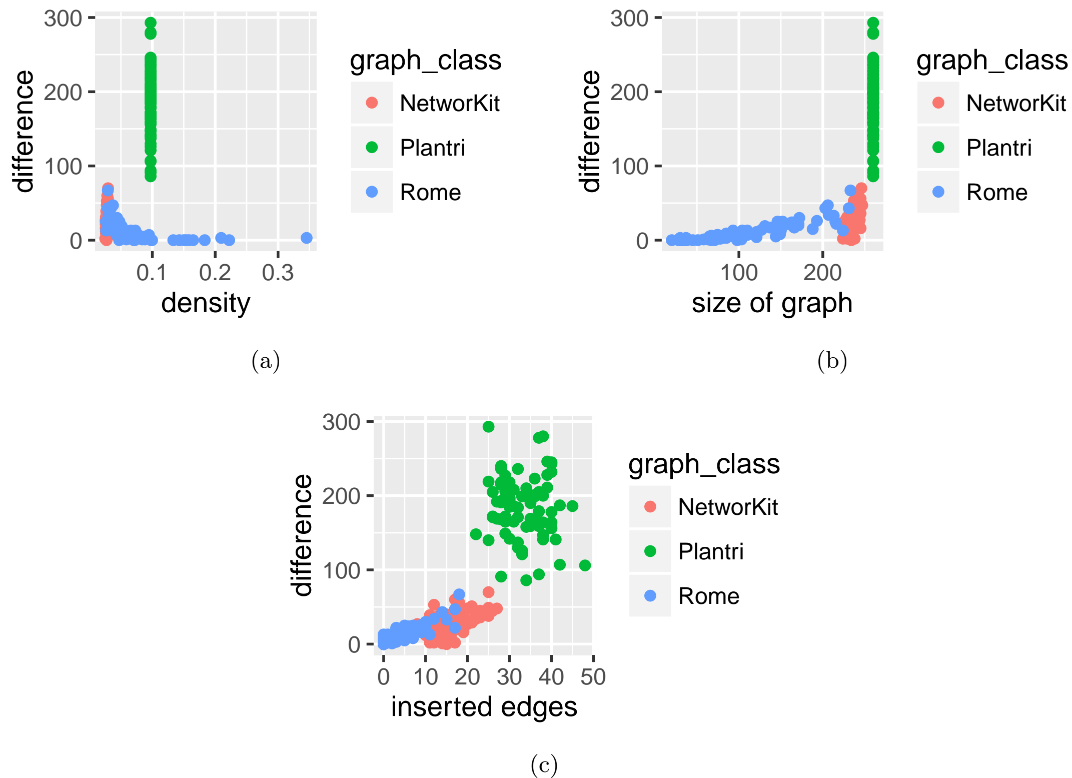
We have seen in section 5.2.2, that Gcm-Base-High = Gcm* is our best configuration concerning quality. Moreover, we have seen, that Gcm* performs better on the Plantri graphs than on the NetworKit graphs or on the Rome graphs. The Plantri graphs have a higher density, than most NetworKit graphs. This is why we suspect, that the density $\rho = 2|E|/(|V| \cdot |V| - 1)$ of a graph G correlates positively with the difference $d^G = \overline{\text{cr}}^G_{\text{Spring}} - \overline{\text{cr}}^G_{\text{Gcm}^*}$, where $\overline{\text{cr}}^G_{\text{Spring}}$ is the crossing number of the final straight-line drawing of G by the SpringEmbedder. Analogously $\overline{\text{cr}}^G_{\text{Gcm}^*}$ is the crossing number of the final straight-line drawing of G by Gcm*. Based on this presumption, we deploy the following hypothesis:

H_0 : *The density ρ of a graph G is positively correlated with the difference d^G.*

But the spearman's rank-order correlation coefficient described in section 5.2.1 rejects this hypothesis on all three graph classes. Figure 5.10 suggests, that the difference d^G may be positively correlated with the number of inserted edges $|E^+|$ rather than with the density. We therefore deploy the following new hypothesis.

H_0 : *The number of inserted edges $|E^+|$ is positively correlated with the difference d^G.*

This hypothesis is accepted by spearman's rank-order correlation coefficient on the NetworKit graphs and on the Rome graphs, but rejected on the Plantri graphs.



(a)

(b)

(c)

Figure 5.10:

5.2.5 Complete Graphs

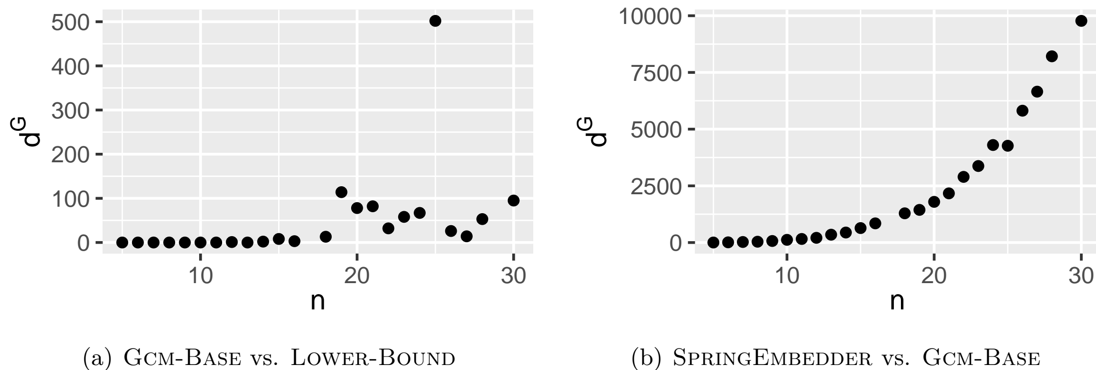
There are no known lower or upper bounds for the rectilinear crossing number of arbitrary graphs. Therefore, it is impossible to estimate how close our configurations are to an optimal solution. Nevertheless, for the family of complete graphs $\mathcal{K}_n$ there is a collection of proven lower and upper bounds for the rectilinear crossing number of $\mathcal{K}_n$. These upper and lower bounds are summarized on the website of the *rectilinear crossing number project* [Aic13]. The lower and upper bounds of $\mathcal{K}_n$ for $n \in \{1,..,100\}$ are collected on this website. In this section we want to evaluate GCM on the graphs $\mathcal{K}_1,...,\mathcal{K}_{100}$ and compare the resulting crossing numbers to the highest currently known theoretical lower bound.

For the family of complete graphs, the edge number increases quadratic with the node number n of $\mathcal{K}_n$. Subsequently even our basic configuration GCM-BASE needs a running time of 226 minutes to draw $\mathcal{K}_{30}$. Our best configuration GCM-HIGH has an even higher running time as analyzed in Section 5.3. Therefore, we restrict our evaluations to GCM-BASE and the complete graphs $\mathcal{K}_1,...,\mathcal{K}_{30}$. These graphs are called *Complete-30* from now on. For completeness we also compare GCM-BASE to SPRINGEMBEDDER. For most cases of $\mathcal{K}_n$ with $n \in \{1,...,100\}$, the upper bound u and the lower bound l are tight and $u = l = c^*$, where c^* is the actual minimum rectilinear crossing number of $\mathcal{K}_n$. The only exceptions are $\mathcal{K}_{28}$ and $\mathcal{K}_{29}$, where there is a difference between the currently known upper bound and the currently known lower bound.

Figure 5.11 compares four pairs of algorithms
$A_1, A_2 \in \{\text{GCM-BASE}, \text{SPRINGEMBEDDER}, \text{LOWER-BOUND}, \text{GUTWENGER}\}$. All sub-figures in Figure 5.11 show the difference $d^G = \overline{\text{cr}}^G_{A_1} - \overline{\text{cr}}^G_{A_2}$ on graph G between algorithm A_1 and algorithm A_2 on the y-axis and the number of nodes n on the x-axis.

Figure 5.11(a) shows, that GCM-BASE is very close to the optimal lower bound. In particular up to $n = 11$ GCM-BASE finds the optimal drawing of $\mathcal{K}_n$. Moreover, except for 4 outliers the crossing number produced GCM-BASE is only up to 2.5% higher than the optimal lower bound and on all graphs GCM-BASE is only up to 11% higher, than the optimal lower bound.

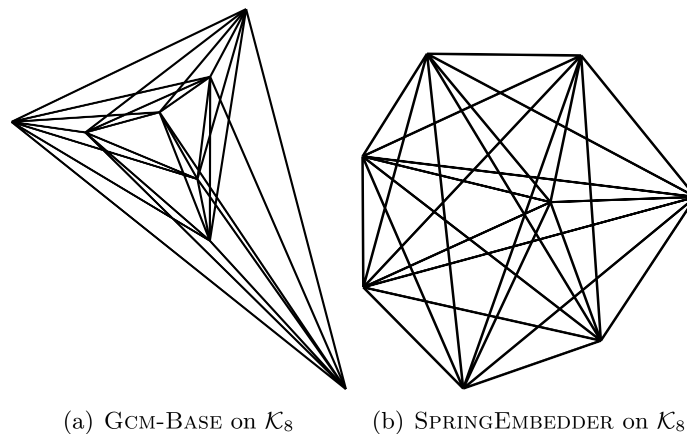
Figure 5.11(b) shows, that the difference d^G between SPRINGEMBEDDER and GCM-BASE increases monotonic with n. Furthermore, on all graphs the crossing number of the straight-line drawing produced by SPRINGEMBEDDER is at least 1.9 times as high as the crossing number of the straight-line drawing produced by GCM-BASE.



(a) GCM-BASE vs. LOWER-BOUND (b) SPRINGEMBEDDER vs. GCM-BASE

Figure 5.11:

One reason for the bad performance of the SPRINGEMBEDDER on complete graphs is, that the SPRINGEMBEDDER draws complete graphs very symmetric. Because each two nodes are connected by an edge in a complete graph, the nodes tend to be placed in a circle such that each two nodes have approximately the same distance towards one another. Figure 5.11 shows an example-drawing.



(a) GCM-BASE on $\mathcal{K}_8$ (b) SPRINGEMBEDDER on $\mathcal{K}_8$

Figure 5.12: The resulting drawing of GCM-BASE on $\mathcal{K}_8$ has 19 crossings, whereas the drawing by SPRINGEMBEDDER on $\mathcal{K}_8$ has 58 crossings.

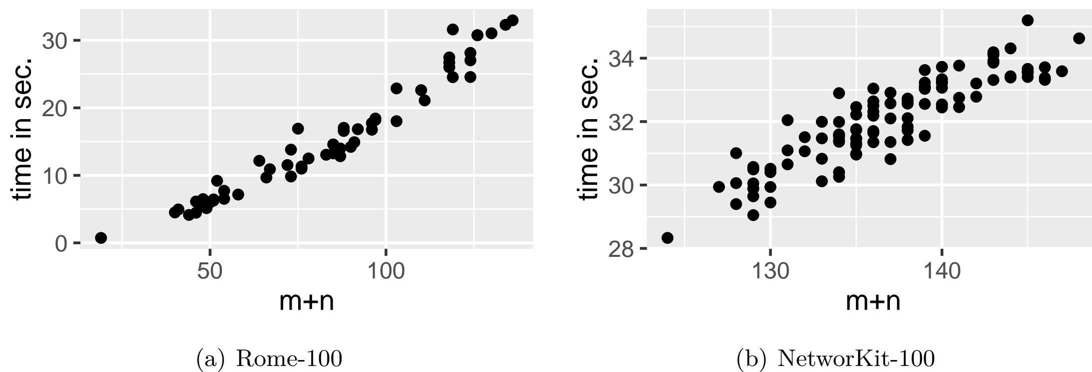
5.3 Running Time

In this section we compare the running time of GCM-BASE and GCM-BASE-HIGH. Both of them have a theoretical running time of $\mathcal{O}(m \cdot (\deg_{max} \cdot m)^2)$ with $\deg_{max}$ denoting the

maximal degree of a node of G. But in practice we expect Gcm-Base to be faster than Gcm-Base-High, because it passes less iterations of moving a node to its locally optimal position.

Running Time per Node Movement

Figure 5.13 shows the running time of Gcm-Base per node movement. Each point in the plane represents a graph. The x-coordinate of this point represents the graph's size $m + n$, whereas the y-axis represents the average running time of one node movement of Gcm-Base. Since a node movement has a theoretical worst-case running time of $\mathcal{O}((\deg_{max} \cdot m)^2)$, we would expect the graph size to have a quadratic relationship to the running time. We observe from figure 5.13 that on our instances the relationship looks rather sub-quadratic. Nevertheless, figure 5.13 shows, that there is a monotonic relationship between the graph size $m + n$ and the average running time per node movement. Since the graphs of Plantri-100 all have the same graph size $m + n$ there are only plots for NetworKit-100 and Rome-100.



(a) Rome-100 (b) NetworKit-100

Figure 5.13: Running time per node movement of Gcm-Base

Running Time per Edge Insertion

Figure 5.14 consists of three figures 5.14(a), 5.14(b) and 5.14(c) and compares the average running time per edge insertion of Gcm-Base and Gcm-Base-High on the Rome-100, Plantri-100 and the NetworKit-100 respectively. In each of these figures a point represents a graph and the y-axis represents the running time in seconds. In figure 5.14(a) and figure 5.14(b) the x-axis represents the graph size $m+n$. Due to the fact, that all graphs contained in the Plantri-100 have the same graph size, we choose the number of edge insertions $|E^+|$ as x-axis for figure 5.14(c).

On the Rome-100 Gcm-Base has nearly the same running time per edge insertion as Gcm-Base-High. This makes sense, because on the Rome-100 Gcm-Base-High has no significant advantage concerning the quality over Gcm-Base. We assume, that Gcm-Base-High performs nearly the same number of node movements per edge insertion as Gcm-Base, because this suffices to reduce the crossings with the newly inserted edge. Again there is a positive monotonic relationship between the graph size $m + n$ and the running time per edge insertion.

Furthermore, 5.14 shows, that on the NetworKit-100 the average running time per edge insertion of Gcm-Base is higher than the respective running time of Gcm-Base-High.

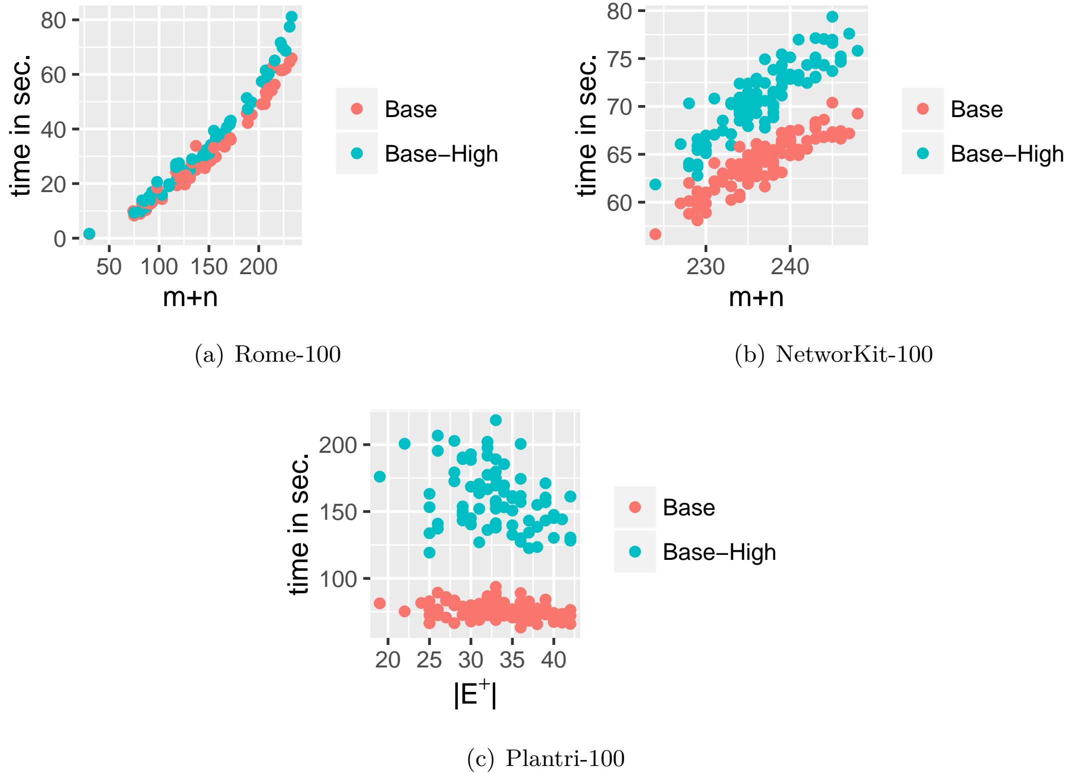


Figure 5.14: Running time per Edge-Iteration, Gcm-Base vs. Gcm-Base-High

Moreover, the difference of these running times seems to be constant. On the Plantri-100 Gcm-Base-High lies between 120 and 230, whereas the running time of Gcm-Base is smaller than 100 for each graph.

Running Time per Graph

Figure 5.15 compares the running time of Gcm-Base and Gcm-Base-High per graph. Because the theoretical running time of Gcm-Base is $\mathcal{O}(m \cdot (\deg_{\max} \cdot m)^2)$ we would expect the running time to be a polynomial of degree 3 in the graph size $m + n$. In figures 5.15(a) and 5.15(b) we can at least observe a positively monotonic relationship between the graph size $m + n$ and the running time. Moreover, Figure 5.15(c) shows an approximately linear relationship between the number of inserted edges $|E^+|$ and the running time per graph.

Expectedly on all three graph classes Gcm-Base-High has a higher running time than Gcm-Base. On the Plantri-100 graphs the running time of Gcm-Base-High is approximately twice as high as as on Gcm-Base.

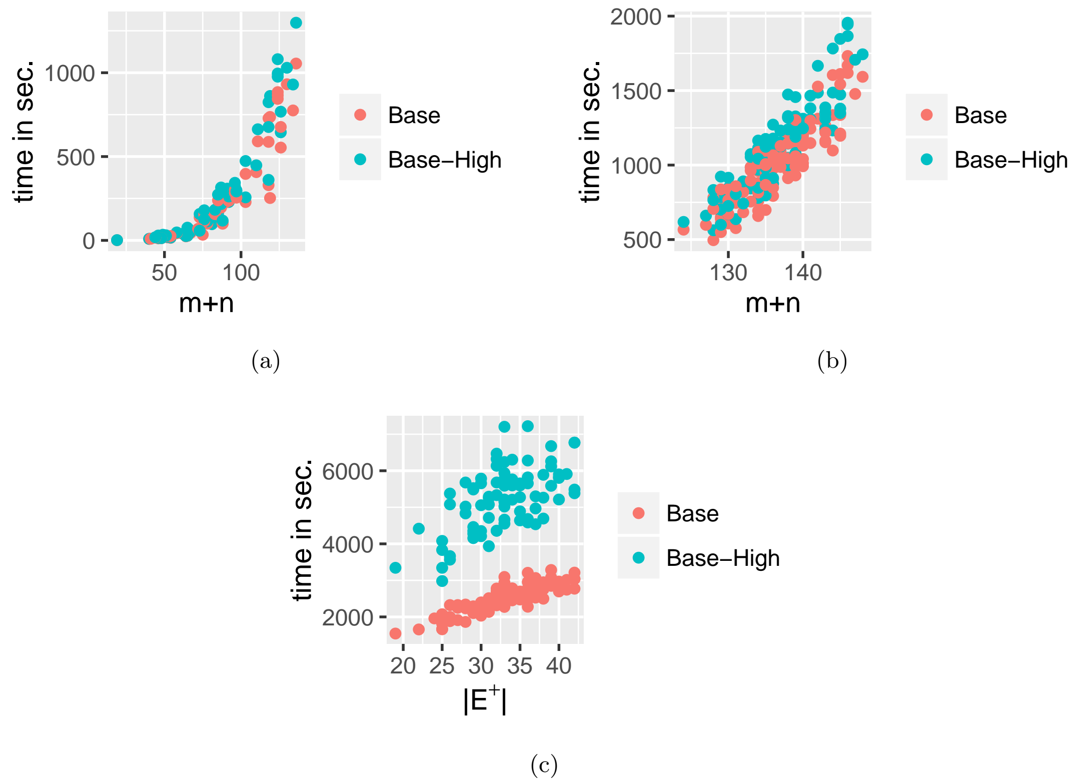


Figure 5.15: Running time per graph, Gcm-Base vs. Gcm-Base-High

6. Conclusion

Rectilinear crossing minimization is an NP-hard [Bie91] and further $\exists\mathbb{R}$-complete [Sch09]. For this reason, there is only a small chance to find a polynomial-time algorithm which computes a straight-line drawing of a graph G that realizes the rectilinear crossing number $\overline{\mathrm{cr}}(G)$. Gutwenger et al. have put much effort into designing a heuristic for computing a topological drawing which realizes the crossing number $\overline{\mathrm{cr}}(G)$ of a graph G [GMW05]. We are not aware of any similar heuristic for rectilinear crossing minimization which focuses on minimizing the crossings only. There is one approximation-algorithm for rectilinear crossing minimization which has been proposed recently by Fox et al. More precisely, they describe a polynomial-time algorithm which computes a drawing with $\overline{\mathrm{cr}}(G) + \mathcal{O}(n^4/(\log\log n)^\delta)$ crossings [FPS16, Thm. 1]. This result describes an asymptotic approximation but is not shown to minimize crossings in practice.

We developed a heuristic called GCM for rectilinear crossing minimization, we pursued an approach inspired by Gutwenger et al. [GMW05]. The idea is to first extract a maximum planar subgraph G^* of G, draw G^* without crossings and iteratively reinsert the missing edges into G^*. After each edge reinsertion we try to reduce the crossings by moving nodes to their crossing-minimal position. We identified several node orders in which these nodes can be moved. We found that, the best variant among these is to order nodes according to the number of crossings which are caused by an incident edge from high to low. Furthermore we proposed a way to move whole subgraphs at once in order to escape from locally optimal node placements.

We evaluated these configurations of GCM against some comparison algorithms. For the choice of this comparison algorithm, we restricted ourselves to rectilinear drawing algorithms implemented in the commonly known graph drawing framework OGDF. More precisely we chose the spring embedder by Fruchterman and Reingold with stress minimization as initial layout, which produces the lowest number of crossings on average on the Rome-100 benchmark-set. For the evaluation of our configurations of GCM against the spring embedder, we took a statistical approach. We were able to show with statistical testing, that all our configurations produce drawings with significantly less crossings than the spring embedder. Furthermore, we found that the best configuration does not move whole subgraphs. Moving whole subgraphs does improve the results, but only if few nodes are moved in each iteration. It has a positive effect to apply a force-directed algorithm which preserves the crossings in each iteration.

6.1 Outlook and Future Work

One drawback of our heuristic GCM is certainly its running time of $\mathcal{O}((m \cdot \deg_{max})^2))$ for one node movement. In order to reduce this running time, one approach is to select a small set of inactive edges to build $\mathcal{A}(G, s)$. The theory of ε-nets tells us that we still find a point with a small number of crossings with high probability. It would be interesting to evaluate this idea in practice.

Furthermore, moving whole subgraphs in a drawing did not improve the crossing number as much as we expected. One reason for this result is that if GCM determines a subgraph G^* with many nodes and edges, some of these edges tend to cross G^*. This issue is fixed with local improvements but, nevertheless, can affect the number of crossings in the final drawing negatively. We presume that moving smaller subgraphs could help to solve this problem. The critical parameter for running time of GCM is the number of edges. We cannot compute drawings with GCM in reasonable time for graphs with more than 1000 edges. One idea to handle greater graphs is to employ a multilevel approach. More precisely a multilevel partition could be used. The algorithm starts with the coarsest partition and contracts all nodes in a cell into one node. In each iteration the nodes are recontracted to the next finer level of the multilevel partition.

Bibliography

[ÁCFM+10] Bernardo M Ábrego, Mario Cetina, Silvia Fernández-Merchant, Jesús Leaños, and Gelasio Salazar. 3-symmetric and 3-decomposable geometric drawings of kn. *Discrete Applied Mathematics*, 158(12):1240–1258, 2010.

[ACNS82] Miklós Ajtai, Vašek Chvátal, Monroe M Newborn, and Endre Szemerédi. Crossing-free subgraphs. *North-Holland Mathematics Studies*, 60:9–12, 1982.

[ÁFMLS08] Bernardo M Ábrego, Silvia Fernández-Merchant, Jesús Leanos, and Gelasio Salazar. A central approach to bound the number of crossings in a generalized configuration. *Electronic Notes in Discrete Mathematics*, 30:273–278, 2008.

[ÁFMS13] Bernardo M Ábrego, Silvia Fernández-Merchant, and Gelasio Salazar. The rectilinear crossing number of k n: Closing in (or are we?). In *Thirty Essays on Geometric Graph Theory*, pages 5–18. Springer, 2013.

[Aic13] Oswin Aichholzer. On the rectilinear crossing number. *Dynamic web page. URL: http://www. ist. tugraz. at/staff/aichholzer/research/rp/triangulations/crossing/. Web page related to the Rectilinear crossing number project (URL: http://dist. ist. tugraz. at/cape5/)*, 2013.

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM (JACM)*, 45(3):501–555, 1998.

[ARV09] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM (JACM)*, 56(2):5, 2009.

[BD93] Daniel Bienstock and Nathaniel Dean. Bounds for rectilinear crossing numbers. *Journal of Graph Theory*, 17(3):333–348, 1993.

[Ber99] François Bertault. A force-directed algorithm that preserves edge crossing properties. In *International Symposium on Graph Drawing*, pages 351–358. Springer, 1999.

[Bie91] Daniel Bienstock. Some provably hard crossing number problems. *Discrete & Computational Geometry*, 6(3):443–459, 1991.

[BL84] Sandeep N Bhatt and Frank Thomson Leighton. A framework for solving vlsi graph layout problems. *Journal of Computer and System Sciences*, 28(2):300–343, 1984.

[BM+07] Gunnar Brinkmann, Brendan D McKay, et al. Fast generation of planar graphs. *MATCH Commun. Math. Comput. Chem*, 58(2):323–357, 2007.

[CFFK98] Gruia Călinescu, Cristina G Fernandes, Ulrich Finkler, and Howard Karloff. A better approximation algorithm for finding planar subgraphs. *Journal of Algorithms*, 27(2):269–302, 1998.

[CGJ+11] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). *Handbook of Graph Drawing and Visualization*, pages 543–569, 2011.

[Cha92] Herbert Chazelle, Bernard; Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, pages 1–54, 1992.

[CHI79] T. CHIBA. An algorithm of maximal planarization of graphs. *Proc. IEEE Symposium on Circuits and Systems, 1979*, 1979.

[DBETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[DH96] Ron Davidson and David Harel. Drawing Graphs Nicely using Simulated Annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[Dji06] Hristo N Djidjev. A linear-time algorithm for finding a maximal planar subgraph. *SIAM journal on discrete mathematics*, 20(2):444–462, 2006.

[Ead84] Peter Eades. A heuristics for graph drawing. *Congressus numerantium*, 42:146–160, 1984.

[FML14] Ruy Fabila-Monroy and Jorge López. Computational search of small point sets with small rectilinear crossing number. *arXiv preprint arXiv:1403.1288*, 2014.

[FPS16] Jacob Fox, Janos Pach, and Andrew Suk. Approximating the rectilinear crossing number. *arXiv preprint arXiv:1606.03753*, 2016.

[FR91] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[GJ83] Michael R Garey and David S Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983.

[GKN05] Emden R. Gansner, Yehuda Koren, and Stephen C. North. Graph Drawing by Stress Majorization. In *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *Lecture Notes in Computer Science*, pages 239–250. Springer, January 2005.

[GM03] Carsten Gutwenger and Petra Mutzel. An experimental study of crossing minimization heuristics. In *International Symposium on Graph Drawing*, pages 13–24. Springer, 2003.

[GMW05] Carsten Gutwenger, Petra Mutzel, and René Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005.

[HT74] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[JLM98] M Junger, Sebastian Leipert, and Petra Mutzel. A note on computing a maximal planar subgraph using pq-trees. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(7):609–612, 1998.

[JTS89] R Jayakumar, Krishnaiyan Thulasiraman, and Madisetti NS Swamy. o (n 2) algorithms for graph planarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(3):257–267, 1989.

[KK89] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[Lei83] Frank Thomson Leighton. *Complexity issues in VLSI: optimal layouts for the shuffle-exchange graph and other networks.* MIT press, 1983.

[LFR08] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110, 2008.

[LG77] PC Liu and RC Geldmacher. On the deletion of nonplanar edges of a graph. In *Proc. 10th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 727–738, 1977.

[Pac12] János Pach. *New trends in discrete and computational geometry*, volume 10. Springer Science & Business Media, 2012.

[PT00] János Pach and Géza Tóth. Thirteen problems on crossing numbers. *Geombinatorics*, 9(DCG-ARTICLE-2000-005):194–207, 2000.

[Pur97] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

[Rad15] Marcel Radermacher. *How to Draw a Planarization.* PhD thesis, Informatics Institute, 2015.

[Sch09] Marcus Schaefer. Complexity of some geometric and topological problems. In *International Symposium on Graph Drawing*, pages 334–344. Springer, 2009.

[SH76] Michael Ian Shamos and Dan Hoey. Geometric intersection problems. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 208–215. IEEE, 1976.

[She07] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman & Hall/CRC, 4 edition, 2007.

[Sho] PW Shor. Stretchability of pseudolines is np-hard. applied geometry and discrete mathematics, 531–554. *DIMACS Ser. Discrete Math. Theoret. Comput. Sci*, 4.

[SSM14] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. *arXiv preprint arXiv:1403.3005*, 2014.

[Tar83] Robert Endre Tarjan. *Data structures and network algorithms*, volume 44. Siam, 1983.

[Tut63] William T. Tutte. How to Draw a Graph. *Proceedings of the London Mathematical Society*, s3-13(1):743–767, 1963.

[Wag93] Stan Wagon. *The Banach-Tarski Paradox*, volume 24. Cambridge University Press, 1993.