Karlsruhe Institute of Technology

How to Draw a Planarization

Master Thesis of

Marcel Radermacher

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers: Prof. Dr. Dorothea Wagner
Prof. Dr. Peter Sanders
Advisors: Dr. Ignaz Rutter
Dipl.-Inform. Thomas Bläsius

Time Period: 1. December 2014 – 31. May 2015

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, 26th May 2015

Abstract

In this thesis we study the problem of computing a straight-line representation of a planarization with a fixed combinatorial structure. By the results of Mnëv [Mnë88] and Shor [Sho91] this problem is as hard as the Existential Theory of the Reals ($\exists\mathbb{R}$-Complete). We present a heuristic called *Geometric Planarization Drawing* to compute a straight-line representation of a planarization. We take an iterative approach inspired by a force-directed method utilizing geometric operations to place a vertex to a locally optimal position. The core of our algorithm is the *planarity region*, in which we can move a vertex without changing the embedding. We take a statistical approach to evaluate the final drawings of different configurations of our Geometric Planarization Drawing approach. The evaluation shows that we improve the initial drawing, and find better results than our implementation of a force-directed method based on *PrEd* [Ber99]. Depending on the number of crossings per edge, a relaxation of the constraints improves the final drawing. To the best of our knowledge, the Geometric Planarization Drawing approach is the first heuristic to tackle the problem of straight-line representations of a planarization. In practice, our algorithms finds close to optimal solutions for drawings with at most 3 crossings per edge.

Deutsche Zusammenfassung

In dieser Arbeit untersuchen wir wie sich geradlinige Zeichnungen von Planarisierung berechnen lassen. Mnëv [Mnë88] und Shor [Sho91] konnten zeigen, dass dieses Problem so schwer ist wie die Existential Theory of the Reals ($\exists\mathbb{R}$-vollständig). Wir stellen eine Heuristik zum geradlinigen Zeichnen von Planarisierung vor. Unser *Geoemtric Planarization Drawing* Ansatz geht iterativ vor, ähnlich zu einem kräftebasierten Verfahren. Wir nutzen geometrische Operationen um nacheinander Knoten zu einer lokal optimalen Position zu verschieben. Der Kern unseres Verfahrens ist die *Planarity Region*. Diese Region beschreibt die einbettungserhaltenen Positionen eines Knotens. Wir werten die finalen Zeichnungen verschiedener Konfigurationen unseres Verfahrens anhand von statistischen Tests aus. Unsere Auswertung zeigt, dass wir eine signifikante Verbesserung der initialen Zeichnungen erreichen und wir bessere Resultate als unsere Implementierung des kräftebasierten Verfahren PrEd erzielen. Abhängig von der Anzahl an Kreuzungen auf einer Kante führt eine Relaxierung unserer Einschränkungen zu einer Verbesserung der Zeichnungen. Nach unserem Wissensstand ist unser Geometric Planarization Drawing Ansatz die erste Heuristik zum Berechnen (möglichst) geradliniger Zeichnungen von Planarisierungen. In der Praxis erreicht unser Verfahren (fast) optimale Lösungen für Zeichnungen mit maximal 3 Kreuzungen auf einer Kante.

Contents

1. Introduction

Data analysis is an important part of many scientific fields. A great part of data in research can be described as a graph. For example, UML-diagrams in software engineering, protein-protein interaction networks in biology, or social interaction in sociology. For most people it is hard to extract useful information out of a graph by looking at pure data. Thus, visualizing data and graphs plays a key role in data analysis and in the understanding of relationships. Planar graphs are a well studied family of graphs. The drawing of a planar graph is intersection-free and thus, it is easy to track the pathways of the edges. In many application the underlying graph is not planar and the significance of minimizing the number of crossings becomes of an important factor. For non-planar graphs, minimizing the number of crossings in the drawing of a graph improves the readability of the layout significantly [Pur97]. Depending on the application, a specific representation of the graph might be preferable. Figure 1.1 depicts different drawings of the complete graph K_5 with five vertices. It might be desirable to find a straight-line drawing of a non-planar graph with a minimum number of crossings. An embedding of a graph with a minimum number of crossings does not have to admit a straight-line drawing with the same number of crossings [Guy72]. For example, the complete graph K_8 with eight vertices has a rectilinear crossing number of 19 and a crossing number of 18. Thus, we can restrict the crossing number problem to straight-line drawings and ask for the minimum number of crossings of a straight-line drawing of a graph G. This problem is known as the *rectilinear crossing number problem*. Both problems, the crossing number problem and the rectilinear crossing number problem are $\mathcal{NP}$-hard [Can88, Bie91].

Thus, we cannot hope for an efficient algorithm that computes a drawing of a graph with a minimum rectilinear crossing number. Nevertheless, there are well known algorithms that compute a *planarization* of a graph with a small number of crossings. These methods

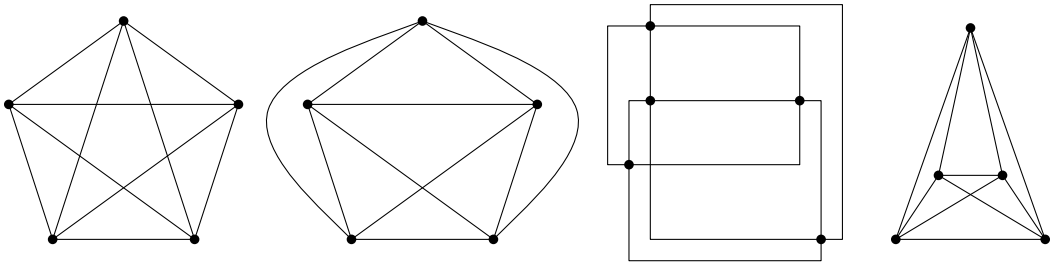


Figure 1.1.: Different drawings of the complete graph K_5 with five vertices.

extract a maximal subgraph from a graph G and reinsert the missing edges into the graph so that these edges have small number of crossings [GMW05]. The result is a planar graph G_p with *dummy vertices* at the crossings. Our idea is to utilize these topological results and compute a geometrical embedding of the planarization with the same combinatorial structure. In the underlying optimization problem we are interested in whether or not there is a straight-line representation of the planarization with the same combinatorial structure of the planarization. Unfortunately, this problem can be reduced to a $\mathcal{NP}$-hard problem called *stretchability* [Mnë88, Sho91].

1.1. Related Work

The field of graph drawing is interested in drawings of graphs which are easy to comprehend. For non-planar graphs, the empirical study of Purchase [Pur97] suggests that minimizing the number of crossings in a drawing has the greatest impact on the readability of a graph. Whereas minimizing the number of bends and maximizing the symmetry plays only a minor role. Thus, finding drawings of graphs with few crossings is an important task of graph drawing. Unfortunately, the problem is known to be $\mathcal{NP}$-hard [GJ83, Bie91].

The *rectilinear crossing number* problem asks if there is a straight-line drawing of a graph with at most $k \in \mathbb{N}$ crossings. This problem is $\mathcal{NP}$-hard as well [Bie91]. According to Schaefer [Sch10] this problem is actually $\exists\mathbb{R}$-complete. Where $\exists\mathbb{R}$ is a class called *the existential theory of the reals* and captures all true sentences of the form $\exists x_1, x_2, \ldots, x_n : \phi(x_1, x_2, \ldots, x_n)$, where ϕ is a quantifier free Boolean formula over the signature $(0, 1, +, \times, <)$ and the real numbers [Sch10]. The class is decidable in PSPACE [Can88].

Planar Graphs

The title of this thesis is inspired by the title of one of the first papers on planar graph drawing: "How to draw a Graph" by Tutte [Tut63]. Fáry's theorem states that every planar graph has a straight-line drawing [Fár48]. Tuttes algorithm computes a planar straight-line drawing of a graph. The algorithm draws the vertices of the external face of a graph as a convex polygon. Every other vertex is assigned the average position of all its neighbors. This can be formulated as a system of linear equations. Unfortunately, this kind of drawing can result in an exponential resolution of the drawing, i.e., there is a family of drawings so that the ratio of the length of the shortest edge to the length of the longest edge is exponential in the number of vertices of the graph.

Kant [Kan96] introduced a straight-line drawing of planar graphs that draws a graph on a $O(n) \times O(n)$ grid. The idea of the algorithm is illustrated in Figure 1.2. The algorithm iteratively places the vertices in a leftmost canonical ordering. The first three vertices are

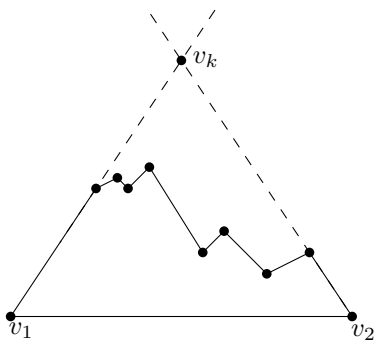


Figure 1.2.: Placement of a new vertex in Kant Drawings.

drawn as a triangle. Every other vertex v is placed in the order of the canonical order. The position of the vertex is the intersection of the lines defined by the left and right most vertex incident to v and their edges on the external face.

Planarization Methods

Badini, Talamo and Tamassia used a three phase concept to draw a planarization of a non-planar graph G [BTT84]. The *topology* phase uses a so-called *planarization method* to ensure that the final drawing has only a small number of crossings. The second phase determines the *shape* of the drawing. The final phase fixes the *metric* of the (orthogonal) drawing. Their planarization approach removes those edges from the graph responsible for crossings. This results in a planar graph and a set of removed edges. They reinsert the removed edges incrementally so that the edges add as few crossings as possible to the fixed embedding of the graph. In order to get a planar graph, the edges are split at crossings and so-called *dummy vertices* are inserted.

A more sophisticated approach computes a maximal planar subgraph of G and inserts the remaining edges in such a way that an edge has a minimum number of crossings and the planar subgraph has a crossing free drawing. This algorithm by Gutwenger et al. [GMW05] is known as the *single edge insertion* algorithm. They proved, that their algorithm requires only linear time to insert one edge optimally. In general, we are interested in inserting a set of edges into a planar graph, so that the edges have a minimum number of crossings and original graph remains planar. This $\mathcal{NP}$-hard [Zie00] problem is known as the *multiple edge insertion algorithm*. A simple heuristic is to iteratively apply the single edge insertion algorithm. The number of crossings depends on the order in which the edges are inserted. Thus, different random permutations of the edges can be computed. The result is the setting with the minimum number of crossings. A further post-processing step removes an edge from the planarization and reinserts this edge to the graph [CG12].

Another planarization method inserts a vertex with all its incident edges to the planar graph with a minimum number of crossings on those edges. Chimani et al. [CGMW09] introduced a rather complex algorithm that solves this problem in polynomial time using SPQR trees and dynamic programming. The SPQR trees can represent all embeddings of a planar graph with linear space in the number of vertices [DBT96, GM01]. Note that there is an exponential number of embeddings of a planar graph.

Stretchability

Unfortunately, there is no known polynomial time algorithm to decide whether or not a planarization has a straight-line representation. A similar problem called *stretchability* was studied by Mnëv [Mnë88] and Shor [Sho91]. In the original problem we are interested in whether or not a set of pseudolines in the projective plane, where each pair of curves intersects at most once, has a straight-line representation with the same combinatorial structure; compare Figure 1.3. This problem is $\exists\mathbb{R}$-complete and according to Schaefer the problem can be restated in the euclidian plane as follows [Sch10]: is there a straight-line representation of pairwise intersecting x-monotone curves? This problem is still $\exists\mathbb{R}$-complete. A reduction from stretchability to the question whether or not there is a straight-line representation of a planarization is straight forward. Thus, finding a straight-line representation of a planarization is $\exists\mathbb{R}$-complete as well.

Corollary 1.1. *The problem of finding a straight-line representation of a planarization is $\exists\mathbb{R}$-complete.*

Fáry's theorem states that every planar graph has a straight-line drawing [Fár48]. Hong et al. [HELP12] studied the problem of finding a straight-line representation of 1-planar graphs,

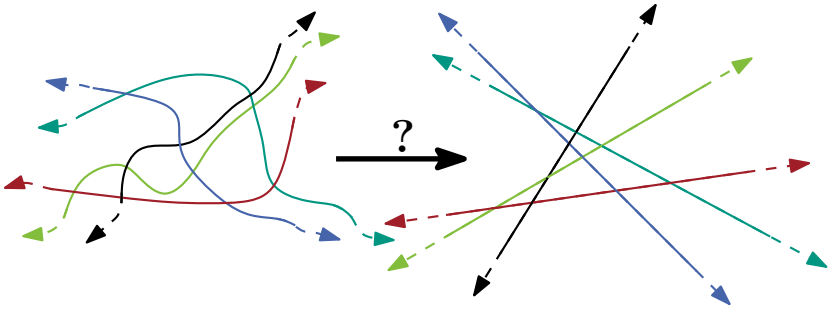


Figure 1.3.: The right drawing depicts a straight-line representation of the pseudoline on the left side.

i.e., graphs with at most one crossing per edge. They introduced a linear time algorithm to decide whether or not a *1-planar* graph has a straight-line representation. A 1-planar graph G has a straight-line representation if neither the BULGARI GRAPH (Figure 1.4(a)) nor the GUCCI GRAPH (Figure 1.4(b)) is a subgraph of G. Further, they provided a linear time algorithm to check if a graph contains one of both subgraphs. If there is a straight-line representation of the graph, their algorithm computes the respective drawing. They observe that the graph depicted in Figure 1.5 has a straight-line representation that requires an exponential area in the number of vertices.

Drawing Planarizations

Tamassia [Tam87] introduced an algorithm to find an orthogonal drawing with a small number of bends. The approach utilizes the three phase concept mentioned before. The first phase fixes the *topology* of the drawing, e.g., computing a planarization and/or fixing the planar embedding of the graph. Orthogonal drawings allow to handle the *shape* and the *metric* in two separate phases. Straight-line drawings coherently determine the shape with the metric in one step. The shape step assigns a minimum number of bends per edge. Finally, the metric phase assigns the length to each (sub)-segment of an edge. Tamassia introduced two network-flow problem to solve the last two phases of his framework.

Force-Directed Methods

Didemo et al.[DLR11] extended the process of drawing an orthogonal drawing of a planarization by a forced-directed method. They take the orthogonal drawing and remove all dummy vertices and insert vertices at every bend. They apply a forced-directed algorithm to this graph. The movement of a vertex is restricted by a certain distance which does not increase the number of crossings in the layout. Nevertheless, the combinatorial structure can change over time and they observe a decrease in the number of crossings.

Bertault [Ber99] introduced a spring-embedder called PrEd which is able to preserve the embedding of a drawings. Simonetto et al. [SAAB11] introduced improvements to PrEd.

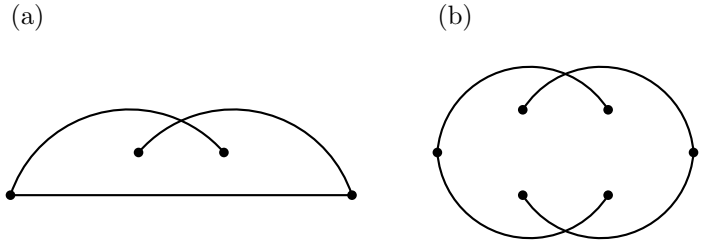


Figure 1.4.: (a) The BULGARI GRAPH. (b) the GUCCI GRAPH.

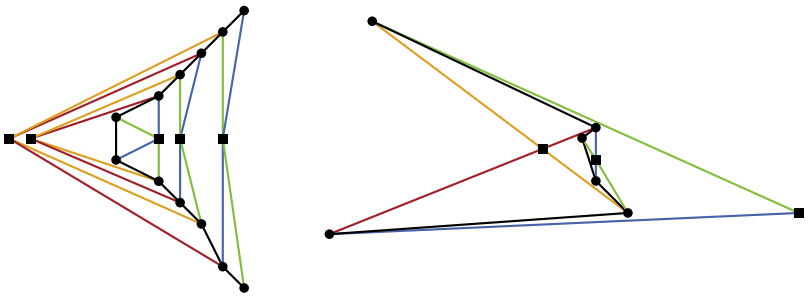


Figure 1.5.: Version of Eades graph with a straight-line representation with an exponential area. On the right, a drawing of the graph with our Geometric Planarization Drawing heuristic.

We use PrEd as a force-directed approach to improve an initial drawing of a planarization. For further details see Section 4.2.

Geometrical Realizations of Topological Graphs

Similar to our problem, the following two results are concerned with the geometrical realization of topological graphs. The problem asks for a simultaneous drawing of two graphs that share a set of vertices and edges. The drawing of both graphs should be planar and the drawing of the intersection graph should coincide. A topological variant of the problem asks for a *simultaneous embedding with fixed edges (SEFE)*, where each edge can be an arbitrary open Jordan curve. Grilli et al. [GHKR14] showed that two graphs with such an embedding admit a drawing with no bend at common edges and at most 9 bends on every other edge. For a (bi-)connected intersection graph the number bends is at most 3.

A related problem is the *planarity of a partially embedded graph* problem. Given a planar graph G and a straight-line drawing of a subgraph H of G, is there a planar drawing of G that contains the drawing of H. Chan et al. [CFG$^+$14] presented an algorithm that extend the drawing of H to G so that every edge has at most a linear number of bends? By Pach and Wenger [PW01] this is worst-case optimal.

1.1.1. Contribution & Outline

There is only a small hope to find an efficient algorithm that computes a straight-line representation of a planarization. We contribute a heuristic to the problem. Our *Geometric Planarization Drawing* approach can be divided into two phases.

1. Find a planarization of a graph G
2. Draw the planarization

For the first phase, there are well known methods that compute a planarization of graph with a small number of crossings and a fixed embedding. Our actual contribution to the problem is the second phase. Given a planar drawing of the planarization, we try to find a straight-line representation of the planarization (with the same embedding as the initial drawing).

The idea of our approach is to successively flatten the planarization paths by carefully pushing the vertices to good position. Fixing one vertex of the planarization path can worsen the adjacent vertices. Figure 1.6 illustrates this behaviour. Thus, we repeat this process until we are not able to further improve the drawing. There are several questions that arise with this approach.

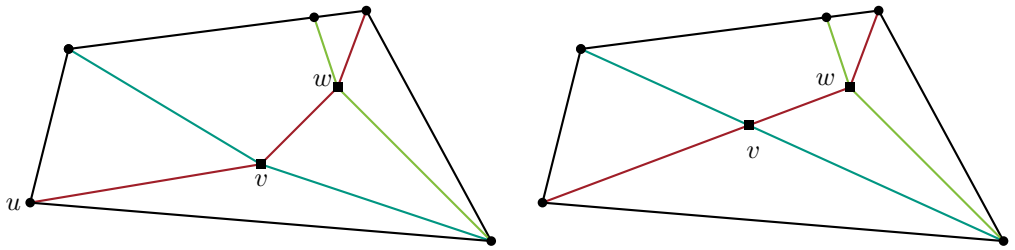


Figure 1.6.: A planarization; The square vertices v and w are dummy vertices. The vertex u is an example for a tail vertex. Improving the planarization path at v, worsens the path at vertex w.

1. How far can we push a vertex without touching any other edges, i.e., how do we preserve the planarity of the drawing?
2. Where should we move the vertices?
3. What is a good strategy? Should we randomly select a vertex or is there a more systematic way to select the next vertex?

We take a geometrical approach to answer the first two questions in our Geometrical Framework in Chapter 3. We select one vertex and introduce the planarity region in Section 3.1.2. The planarity region characterizes the region in which we can place a vertex without changing the embedding of the planar drawing. With this region at hand, we discuss in Section 3.2 how to place a vertex optimally. In Chapter 4 we refine our Geometrical approach and extend a force-directed approach. In Section 4.1, we assemble the operations of Chapter 3 into an algorithm called *Geometric Planarization Drawing*. We specify the open parameters like the initial drawing of the planarization and discuss several vertex orders. As an alternative approach, we introduce an extension of the Springer-Embedder PrEd in Section 4.2. Our approach is a heuristic and we do not have any quality guarantees to the final drawing. Thus in Chapter 5, we experimentally evaluate the Geometric Planarization Drawing approach. We compare the quality of the final drawings of the different configurations of our approach and PrEd.

In the following Chapter 2, we introduce some notations and discuss preliminaries. We close this thesis in Chapter 6 with a short conclusion and thoughts on open problems and potential future work.

2. Notation and Preliminaries

In the following chapter, we introduce some concepts and notations used throughout this thesis.

2.1. Graph

An *(undirected) graph* is a tuple $G = (V, E)$ with a finite set V of vertices and a finite multiset E of edges with $E \subseteq \{\{u, v\} \mid u, v \in V\}$. We denote with $V(G)$ and $E(G)$ the set of vertices and edges of a graph G, respectively. An edge $e = \{u, v\} = \{v, u\}$ *connects* the two vertices $u, v \in V$.

If an edge occurs multiple times in E, then we refer to G as a *multigraph*. We call such edges *parallel*. A *loop* is an edge of the form $\{v, v\}$. A *simple graph* is a graph without loops and parallel edges. The edge $e = \{u, v\}$ is *incident* to the vertices u and v and both vertices are *adjacent* to each other. The neighbors $N(v) \subseteq V$ of a vertex v is the set of adjacent vertices of v. The degree $\deg(v)$ of node v is the number of its incident edges, formally $\deg(v) = |N(v)|$. In context of a graph $G = (V, E)$ we denote with $n = |V|$ the number of vertices and with $m = |E|$ the number of edges. A graph H is a *subgraph* of G, if and only if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

A *path* $P = \langle v_1, v_2, \ldots, v_r \rangle$ is an ordered sequence of adjacent vertices visiting every edge at most once. The *length* of the path is the number of edges it uses. A *simple* path visits every vertex at most once. We call v_1 and v_r the *source* and the *target* of a path. An *interior* vertex of a path is a vertex on the path that is neither the source nor the target vertex. A *cycle* is a path of length at least 1 with $v_1 = v_r$. A *simple cycle* is a cycle with pairwise different vertices $v_1, v_2, \ldots, v_{r-1}$. Throughout this thesis, we assume, if not otherwise stated, that a path and cycle is simple.

A *connected component* of a graph G is a maximal subgraph H for which each pair of vertices $u, v \in V(H)$ is connected by a path from u to v. A graph is *connected* if it has only one connected component. The graph is biconnected if we can remove an arbitrary node and the graph remains connected. If a vertex v separates a graph into two or more connected components, we refer to this vertex as a *cut vertex*.

A *drawing* of an undirected graph $G = (V, E)$ is a mapping of every node $v \in V$ to a point $p \in \mathbb{R}^2$ and each edge $e = \{u, v\} \in E$ to a open Jordan curve in $\mathbb{R}^2$ with the coordinates of u and v as its endpoints. A *straight-line drawing* maps every edge to a line segment.

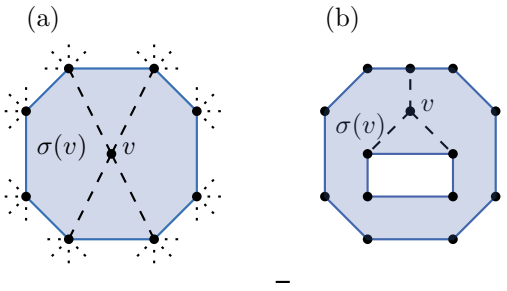


Figure 2.1.: (a) A vertex v with its surrounding $\text{surr}(v)$. (b) If a graph is not biconnected the surrounding of the cut vertex has holes.

Depending on the context we refer with v and e to the vertex and edge, respectively, or to their geometric representation. In a *planar drawing* no two edges intersect except at their common endpoints. A *planar graph* has a planar drawing. A planar drawing divides the plane into connected regions, so-called *faces*. The boundary of a face is a simple cycle in G. There is exactly one unbounded region, we refer to this region as the *external face* or the *outer face*. We refer to every other face as *internal*. The *surrounding* $\text{surr}(v)$ of a vertex v is the face that contains v after removing v and all incident edges of v from the drawing of G. If the graph is not biconnected, the surrounding of a vertex may have holes as Figure 2.1 shows. This is the case for a cut vertex. Throughout this thesis we like to avoid this scenario. Therefore unless otherwise stated, we require the graphs to be simple and biconnected.

Vertices and edges of G are *incident* to a face f if they occur on the boundary of f. Two faces are *adjacent* if they share at least one edge on their boundary. Two faces of two different drawing of G are equal if their boundary is equal. Let $F_1 = \{f_1^1, f_2^1, \ldots, f_s^1\}$ and let $F_2 = \{f_1^2, f_2^2, \ldots f_t^s\}$ be the set of faces of two drawings. The two drawings are *combinatorially equivalent* if there exists a bijective function $\phi : F_1 \to F_2$ so that for every face $f \in F$, f and $\phi(f)$ are equal. Combinatorial equivalence is an equivalence relation. We refer to the equivalence classes as *combinatorial embeddings*. A combinatorial embedding of a graph induces a unique rotational order of the edges incident to a vertex, i.e., two combinatorial embeddings are equal if for every vertex the rotational order of the edges incident to the vertex is equal.

Let $G = (V, E)$ be a (non-planar) graph and let $G_p = (V \cup V_p, E' \cup E_p)$ be a planar graph. We refer to G_p as a *planarization* of G if the following condition apply to G_p.

1. $V \cap V_p = \emptyset$
2. $E' \cap E_p = \emptyset$
3. $E' \subset E$
4. Every vertex in V_p has degree 4
5. If an edge $e = \{u, w\} \in E$ is not element of E', then a path $p_e = \langle v_1, v_2, \ldots, v_r \rangle$ connects u and w with $v_2, v_3, \ldots v_{r-1} \in V_p$ and $r > 2$
6. Two such paths do not share any edges.
7. The paths cover all edges in E_p.
8. The interior vertices of all paths cover all vertices in V_p

Intuitively a planarization of graph G is the graph resulting from placing vertices at the intersection of edges in a drawing of G. Figure 2.2 illustrates the idea of a planarization. Those vertices form the set V_p. We refer to them as *dummy vertices*. We call a vertex $v \in V \cup V_p$ a *tail vertex* if a neighbor of the vertex is a dummy vertex. Note that a vertex can be a dummy vertex and be a tail vertex at the same time. We refer to vertex as *independent* if its neither a dummy nor a tail vertex. Placing the dummy vertices splits some edges of e

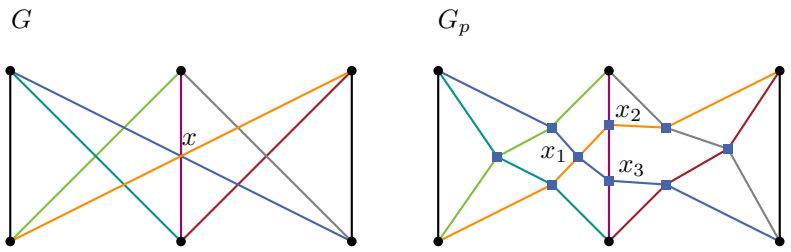


Figure 2.2.: A graph G and its planarization G_p. The blue vertices shows the set V_p. The black edges build the set E' and the colored edges of the right graph are elements of E_p and build the paths p_e for every colored edge e of G. The intersection at point x leads to three vertices in G_p.

into a *planarization path* p_e of e. We call two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ a *dissected pair* if they are incident to the same dummy vertex and belong to the same planarization path. Note that an edge can be in two "relationships". We use the triple (u, v, w) as a shortcut for a dissected pair. In general, we are interested in drawings of a graph with a small number of crossings. Assume, there is an embedding of G_p where two planarization paths only touch but do not cross, then we can remove the dummy vertex v and reduce the number of crossings. If this does not apply for any embedding of G_p, we refer to G_p as a *normalized planarization*. We call the graph G *k-planar* if the longest planarization path in the planarization G_p has at most k dummy vertices, i.e., there are at most k crossings per edge. We refer to the factor k as the *local crossing number* of the planarization. We call a drawing of a (normalized) planarization a *straight-line representation* of the planarization if every planarization path is a straight line, i.e., all vertices on a planarization path are distinct and collinear and no pair of edges covered by the planarization path overlaps.

2.2. Geometric Terminology

We introduce some notations of geometric objects in the plane, i.e., each object is a subset of $\mathbb{R}^2$. Let p_1 and p_2 denote two points and let d be a direction vector. A *line* $\mathcal{L}(p, d)$ with a base p and a direction vector d is the set of points $\mathcal{L}(p, d) = \{p + \lambda d \mid \lambda \in \mathbb{R}\}$. A *ray* $\mathcal{R}(p, d)$ is a subset of the line $\mathcal{L}(p, d)$ with only non-negative λ. A *line segment* $\mathcal{S}(p_1, p_2)$ with the *source point* p_1 and *target point* p_2 is the set of points between p_1 and p_2 and the target point excluded from the set. Two vectors d_1, d_2 form a *left-turn* (*right-turn*) if the determinant of the matrix $(d_1\ d_2)$ is greater (less) than 0. A *half-plane* $\mathcal{HP}(l)$ with respect to a line $l = \mathcal{L}(p, d)$ are all points q to the left of l, i.e., the vectors d and $q - p$ form a left-turn. A bounding box $\mathcal{B}$ of a set of segments is the smallest axis-aligned rectangle that contains all segments. We refer to the distance between two geometric objects O and P as $\text{dist}(O, P) = \min_{o \in P, p \in P} \text{dist}(o, p)$, where $\text{dist}(o, p)$ is the euclidian distance of the two point o and p.

In order to define polylines and polygons, we use the drawing of a simple biconnected planar graph G. A *polyline* is the ordered sequence of coordinates of the vertices of a path. Thus, a *simple polyline* is an ordered sequence of coordinates of a simple path. A polygon $\mathcal{P}$ is a face f in G. The boundary of a polygon $\mathcal{P}$ is the ordered sequence $\langle v_1, v_2, \ldots, v_n \rangle$ of coordinates of the vertices of f. Depending on the context, we refer with $\mathcal{P}$ to the boundary or to the interior of the polygon. Note that vertices can occur multiple times in $\mathcal{P}$. A corner of a polygon is a triple $[v_{i-1}, v_i, v_{i+1 \mod n}]$ with $i = 2, 3, \ldots, n$. A polygon $\mathcal{P}$ is simple if any two edges of $\mathcal{P}$ only intersect at their common endpoints. In our definition of a polygon, two edges or two vertices can touch. We refer to this polygons as *weakly simple*. Chang et al. [CEX14] give a formal definition of weakly simple polygons and introduce an algorithm that detects whether or not a polygon is weakly simple.

Let S be a set of segments. A planar subdivision of the set S is a planar graph so that every vertex of the graph is either an endpoint of a segment or an intersection point of two segments. An edge in the planar subdivision is subsegment of a segment in S. We denote the planar subdivision of a polygon $\mathcal{P}$ with a set of segments S with $\mathcal{P} \sqcap S$, i.e., the planar subdivision of the segments in $\mathcal{P}$ and the segments in S.

Offset Polygon

At some point we need a smaller version of a polygon $\mathcal{P}$, i.e., we have to shrink the polygon somehow. These polygons are called *offset polygons*. The idea behind this concept is to move all vertices of the polygon with the same speed towards the interior of the polygon. The direction of the movement is determined by the straight skeleton of the polygon.

Aichholzer et al [AAAG96] gave a procedural definition of a *straight skeleton*. We find their description instructive and quote their thorough definition of a straight skeleton.

> While the medial axis is a Voronoi-diagram like concept, the straight skeleton is not defined using a distance function but rather by an appropriate shrinking process for [a polygon] $\mathcal{P}$. Imagine that the boundary of $\mathcal{P}$ is contracted towards $\mathcal{P}$'s interior, in a self-parallel manner and the same speed for all edges. Lengths of edges might decrease or increase in this process. Each vertex of $\mathcal{P}$ moves along the angular bisector of its incident edges. This situation continues as long as the boundary does not change topologically. There are two possible types of changes:
>
> - *Edge event*: An edge shrinks to zero, making its neighboring edges adjacent now.
> - *Split Event*: An edge is split, i.e., a reflex vertex runs into this edge, thus splitting the whole polygon. New adjacencies occur between the split edge and each of the two incident to the reflex vertex.
>
> After either type of event, we are left with a new, or two new, polygons which are shrunk recursively if they have non-zero area. [...]
>
> The straight skeleton $S(\mathcal{P})$ is the union of the pieces of angular bisectors traced by the polygon vertices during the shrinking process.

The definition of the straight skeleton is directly related to the offset polygon. The process results in a set of geometric trees, where each point in the tree has the same distance to every leaf in its subtree. Thus, we can start at each leaf and move upwards a predetermined distance and place a vertex at this position. Since the distance of the new vertex is same for all leafs in the subtree of the vertex, the position of the vertex is well defined. Thus, we can connect these vertices in the same order as predefined by the order of the leafs on the polygon. This results in the offset polygon of a polygon $\mathcal{P}$.

Computing the straight skeleton takes at most $O(n^2)$ time [FO98]. The method described in [FO98] is the reference implementation of the CGAL-Library [Cac15]. On real-world data a running time of $O(n \log n)$ can be expected [Hub11]. The time to extract the offset polygon from the straight skeleton is linear in the size of the straight skeleton and thus dominated by computation of the straight skeleton.

2.3. Approximation of Monotone Decision Functions via Binary Search

Let $[a, b] \subset \mathbb{R}$ be an interval and let $f : [a, b] \to \{0, 1\}$ be a monotone increasing binary decision function with $f(b) = 1$. We are interested in the smallest value $x^\star \in [a, b]$ with

$f(x^\star) = 1$. We can use a binary search to find an *absolute ϵ-approximation* $\hat{x}$ of the value $x^\star$, i.e., $\hat{x} - x^\star \leq \epsilon$. Let $[a_t, b_t]$ be the interval of the binary search in step t.

Theorem 2.1. *Let $[a_t, b_t]$ the interval of t-th step of a binary search over the interval $[a, b] = [a_0, b_0]$ and let $f : [a, b] \to \{0, 1\}$ be a monotone increasing binary decision function with $f(b) = 1$. Then, for any $\epsilon > 0$ the value b_t is an absolute ϵ-approximation of the smallest value $x^\star \in [a, b]$ with $f(x^\star) = 1$ after $t = \log(b - a)/\epsilon$ iterations.*

Proof. Since the function f is monotone, the following invariants hold.

1. $x^\star \in [a_t, b_t]$
2. $f(b_t) = 1$
3. for every $t > 0$ $b_t - a_t = (b_{t-1} - a_{t-1})/2$

With these observations we get the following estimation.

$$a_t \leq x^\star \leq b_t \Rightarrow b_t - x^\star \leq b_t - a_t \leq \frac{b-a}{2^t} = \frac{b-a}{2^{\log_2(b-a)/\epsilon}} = \epsilon.$$

□

3. Geometric Framework

Computing a straight-line drawing of a graph with a minimum number of crossing is a $\mathcal{NP}$-hard problem; see Section 1.1. Nevertheless, it easy to compute a planarization of a graph with a small number of crossings. In order to profit of a planarization, we require that the drawing algorithm preserves the embedding of an initial drawing. Note that preserving an embedding is a stronger restriction than preserving only the crossing number. Unfortunately, computing a straight-line representation of a planarization is $\mathcal{NP}$-hard as well; see Section 1.1.

We use an (arbitrary) initial planar straight-line drawing of the graph G_p as a starting point. We take a geometrical approach to iteratively straighten the planarization paths. We move a vertex to a new locally optimal position. The algorithm iteratively selects a vertex and minimizes the so-called *active crossing angles* of the vertex without changing the embedding of the drawing. The crossing angle cr-$\alpha(u, v, w)$ of a dissected pair (u, v, w) is the angle cr-$\alpha(u, v, w) = \pi - \angle(u, v, w)$; compare Figure 3.1. We call crossing angles *active* with respect to a vertex v, if moving v changes these angles. Two basic questions arise.

1. Given a vertex v, what are legit positions for v so that moving the vertex to this position does not change the embedding of the drawing?
2. Given a set of legit positions for v, where do we have to move the vertex in this set to minimize the active crossing angles of the vertex?

The first problem is closely related to the visibility of a vertex in a polygon. We introduce a concept called *planarity region* which describes the set of legit positions of a vertex.

In order to solve the second problem, we distinguish between the different types of a vertex. A vertex can either be a tail vertex, a dummy vertex, a dummy and tail vertex, or an independent vertex. The constrains for each of these type differ. A tail vertex has a set of

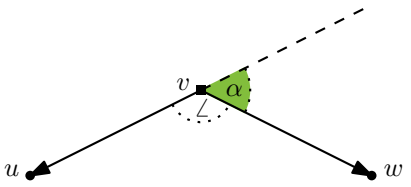


Figure 3.1.: Illustration of the crossing angle of two edges incident to a dummy vertex

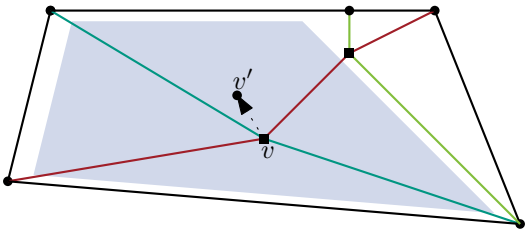


Figure 3.2.: We move a vertex v within the planarity region to its locally optimal position.

dummy neighbors, i.e., multiple planarization path touch at this vertex. If we move such a vertex, we would like to move the vertex to a position collinear with all these planarization paths. On the other hand, if we move a dummy vertex with no dummy neighbors, there are only two constraints. We would like to move the vertex to a position so that each dissected pair of v is a straight-line segment. Further, a dummy vertex can have dummy neighbors. In this case, we have to minimize the crossing angles at the dummy vertex itself and the crossing angles at the dummy neighbors; compare vertex v in Figure 3.2. A vertex can be neither a dummy nor tail vertex. Moving an independent vertex does not change any crossing angles. Thus, it is difficult to formulate an optimization problem for independent vertices. Nevertheless, a good a position for an independent vertex should clear the space for the vertices in the surrounding of the vertex.

The structure of this Chapter is guided by the previous introduced problems. First, in Section 3.1, we introduce the concept of the planarity region and problems related to this region. Secondly, we describe the different types of placement operations for a vertex in Section 3.2.

3.1. Preserving the Planarity

Preserving the embedding of a drawing is a stronger constrained than preserving the crossing number of a drawing. Figure 3.3 shows, that we can change the embedding of a drawing without adding new crossings to the drawing. We refer to the region $\mathcal{PR}(v)$ in that we can move a vertex without changing the embedding as the *planarity region* of the vertex.

The first thing we observe is that we do not have to consider the complete drawing to compute the planarity region. The planarity region is in a natural way limited by the surrounding of a vertex (depicted as the light blue region in Figure 3.3). The second observation is, that the vertex v is connected via a straight line with its neighbors and that this segments are without any crossings. Thus, the vertex does see the its neighbors, and the vertex is visible from its neighbors. Hence, the planarity region is related to the concept of visibility.

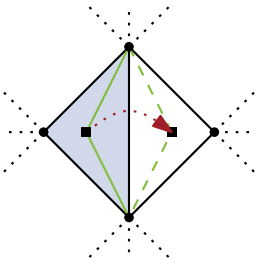


Figure 3.3.: Changing the embedding does not have to add crossings to a drawing.

First of all in Section 3.1.1 we explain the concept of visibility and describe the linear time algorithm of Simpson et al. [JS87] to compute a visibility polygon. We adapt the concept and the algorithm to our problem. We use the visibility region in Section 3.1.2 to compute the planarity region. Both algorithms assume that the surrounding is a bounded polygon. Thus, we customize the surrounding of the external face of a drawing in Section 3.1.3 to fit the problem definition. We would like to satisfy a specific distance of a vertex to the boundary of the planarity region without interfering with a previous optimal position. In Section 3.1.4 we describe a method to shrink the planarity region with the previous constraints.

3.1.1. Visibility in Weakly Simple Polygons

The placement operation of a vertex v depends on the neighbors $N(v)$ of the vertex. Let $z \in N(v)$ be a neighbor of v. In order to preserve the embedding of the drawing, we can move the vertex v to a position that does not introduce an intersection of the segment $\mathcal{S}(z, v)$ with the surrounding $\text{surr}(v)$ of the vertex v. If the surrounding of a vertex is a simple polygon, the desired region is called the *visibility polygon* of the vertex z [EA81]. Unfortunately, the vertex z can occur multiple times on the boundary. Figure 3.4(a) shows that we can move the vertex v around an edge incident to z without adding new crossing to the drawing. Nevertheless, this movement changes the embedding of the drawing.

More formally, the set of *visibility points* of a *view point* z and a (weakly) simple polygon $\mathcal{P}$ is the set of points P_z so that the $\mathcal{S}(z, p)$ lies within $\mathcal{P}$ for every point $p \in P_z$. If the surrounding $\mathcal{P}$ of the vertex v is simple, we can describe the set of visible points with a *visibility polygon* $\mathcal{P}_z$.If the view point z occurs multiple times, we have to exclude points that change the embedding of the drawing. Let u and w be two neighbors of z, so that u occurs before v in the rotational order of z and w after v. We have to restrict the placement of v so that the rotational order of z stays invariant. We can enforce this, by intersecting the set of visible points with a cone defined by the two rays $\mathcal{R}(z, u)$ and $\mathcal{R}(z, w)$. Figure 3.4 illustrates the idea. The result is a *visibility region* $\mathcal{VR}(v, z)$.

Computing the Visibility Polygon

ElGindy and Avis [EA81] introduced a linear time algorithm to compute the visibility polygon of point in a simple polygon. Lee [Lee83] simplified the algorithm. We describe the correction of Lee's algorithm by Joe and Simpson [JS87]. Their visibility algorithm handles only simple polygons. We adjust this algorithm to compute the visibility region of a weakly simple polygons in linear time in the length of the boundary of the polygon. We only need to compute the visibility region for certain points on the boundary of a polygon. Therefore, we assume the view points to be on the boundary $\mathcal{P}$ and the visibility region to be in $\mathcal{P}$.

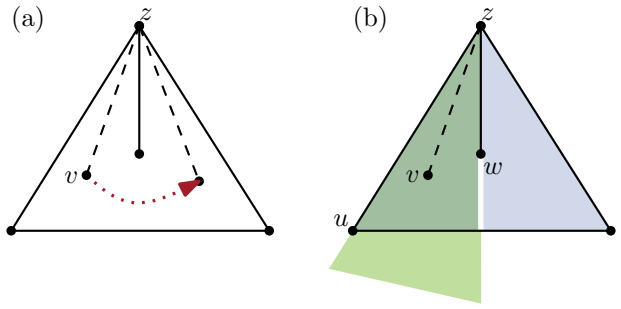


Figure 3.4.: (a) Moving within the visible points can change the embedding. (b) Restricting the visibility with a cone fixes the embedding.

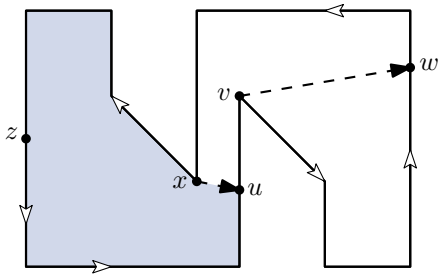


Figure 3.5.: The blue area is the visibility polygon of the vertex z.

The basic idea of the algorithm is to scan the polygon in counterclockwise order and skip all invisible segments. All other segments are (potentially) visible and pushed onto a stack. If the algorithm detects that a previous (sub-)segment cannot be visible, it removes elements from the stack until a segment or a part of a segment becomes visible again. After termination the stack represents the border of the visibility polygon. The algorithm skips only invisible segments of boundary but never a visible part. Thus, during the execution the algorithm makes only a one-sided error, i.e., the stack can contain invisible segments.

Figure 3.5 illustrates the idea of the visibility polygon algorithm. The algorithm starts counterclockwise from the vertex z. The algorithm pushes all vertices on the boundary up to vertex v on a stack. Until then every respective turn was a left turn. At vertex v the direction changes and the concave corner hides everything right of the ray $\mathcal{R}(v, v - z)$. Thus, the algorithm scans the border of the polygon until it finds the intersection w of the border with the ray. From there on, the vertices are potentially visible again and the algorithm pushes all vertices up to vertex x onto the stack. At vertex x the algorithm detects another change in direction. In this case, the algorithm notices that elements on the stack are hidden by this concave corner. Thus, the algorithm removes all invisible segments from the stack and pushes the intersection u of the ray $\mathcal{R}(x, x - z)$ on the stack. The following vertices are visible and the algorithm pushes them onto the stack. Finally, the stack consists of all visible vertices of the boundary of the polygon.

For now, assume the polygon is simple and all segments are open. Let the points $v_0, v_1, \ldots, v_n = v_0$ be the border of the polygon in counterclockwise order and let $z = v_0$ be the view point. We refer with $s_0, s_1, \ldots, s_t$ to the current points on the stack. We assume all indices to be in $i \in \mathbb{Z}_n$.

The algorithm scans the border of the polygon and depending on the last operation, and the relative position of the next vertex to the view point, the algorithm determines the next operation (`PUSH, POP, SCAN_A, SCAN_C, SCAN_D`). After scanning the vertex v_n, the stack consists of all vertices bounding the visibility region. The method `SCAN_B`, as described in [JS87], can never occur for view points on the border of the polygon. The initial operation is `SCAN_A` if zv_1v_2 is a right turn, otherwise it is `PUSH`.

Let the current operation be the `PUSH` operation at vertex v_i. Figure 3.6 illustrates which operation follows next. If zv_iv_{i+1} is a left turn, we move further in the same direction and the vertex v_{i+1} is potentially visible. Thus, a new `PUSH` operation follows. If zv_iv_{i+1} is a right turn, there is a change in direction. If $s_{t-1}v_iv_{i+1}$ is a right turn, v_i is on a concave corner, thus v_i hides v_{i+1}. Therefore, we have to scan the border for the next visible vertex (`SCAN_A`), i.e., the vertex intersecting with the ray starting at v in direction from z to v. Otherwise, the vertex v_{i+1} hides v_i and we have to remove the invisible vertices from the stack. Accordingly, we call `POP`.

The POP operation pops all hidden vertices from the stack. The operation scans the stack $s_t, s_{t-1}, \ldots, s_1$ for the first index j, so that one of the following two cases applies; compare Figure 3.7. Note, that v_i is now the vertex v_{i+1} of the previous operation.

1. zs_jv_i is a right turn and $zs_{j-1}v_i$ a left turn
2. s_j is between z and s_{j-1} and $\mathcal{S}(v_{i-1}, v_i)$ intersects with $\mathcal{S}(s_{j-1}, s_j)$.

If the first case applies, the segment $\mathcal{S}(s_{j-1}, s_j)$ intersects with the ray $\mathcal{R}(z, v_i)$ at the point w. Thus, s_{j-1} is visible but s_j is not. We remove all vertices up to s_{j-1} from the stack and push the intersection w on top of the stack.

If zv_iv_{i+1} is a right turn we move in the same direction. Thus, the next operation is POP again. Otherwise, there is a change in the direction. If $v_{i-1}v_iv_{i+1}$ is a right turn, i.e., the corner at v_i is concave, v_{i+1} is visible again. Thus, we push v_{i+1} on the stack and the PUSH operation follows. Otherwise, we can infer that we are in an invisible region and we can skip forward until we intersect with the segment $\mathcal{S}(v_i, w)$. We can infer that after the scan operation another POP operation has to follow. Hence, we call the operation SCAN_C.

If the second case of the POP operation applies, we remove all hidden points from the stack. Figure 3.7 shows that a scan operation precedes this pop operation and v_i lies within the hidden region of this previous scan and the following vertex has to be hidden again. We can deduce that the next vertex has to be visible. Thus, we call the operation SCAN_D, where w is the intersection of the segments $\mathcal{S}(v_i, v_{i-1})$ and $\mathcal{S}(s_j, s_{j-1})$.

The SCAN_A operation fast forwards to the first segment $\mathcal{S}(v_k, v_{k+1})$ that intersects at the point u with the ray $\mathcal{R}(z, s_t)$; see Figure 3.8. During the scan we might wind around the view point one time. Depending on whether or not the intersection point occurs on the ray $\mathcal{R}(z, s_t)$ before or after s_t we call a POP or SCAN D, respectively. If the scan does not wind around the polygon, we can safely call the PUSH operation.

The SCAN_C and SCAN_D operation work similar; see Figure 3.9. The only difference between both operations is the (unique) operation that succeeds them. Both operations scan the border of the polygon until it intersects with the segment $\mathcal{S}(s_t, w)$, where w is determined by the previous operation. In the context of the previous operation it is clear that after a SCAN_C operation a POP operation has to follow. On the other hand, a PUSH operation always succeeds a SCAN_D operation.

Modification for Weakly Simple Polygons

The original algorithm for the detection of the visibility polygon assumes that all points on the boundary of the polygon are in general position. If we drop the assumption, we have to take care of two new situations. First of all, three points on the boundary of the polygon could be collinear. Secondly, two points on the boundary could occur multiple times. In order to determine the next operations, we have to calculate the relative position of two vertices with respect to the view point. If the view point occurs multiple times, the vector

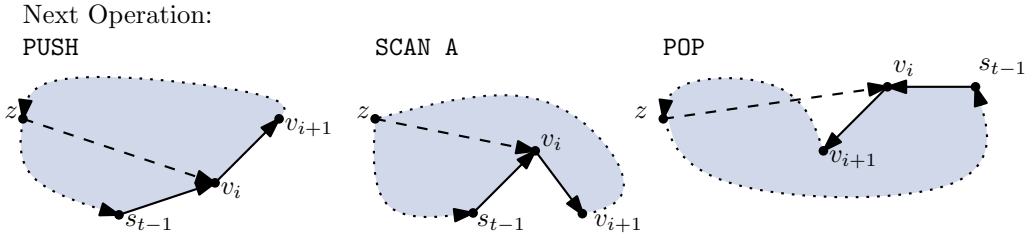


Figure 3.6.: The turns zv_iv_{i+1} and $s_{t-1}v_iv_{i+1}$ determine the operation following a PUSH operation.

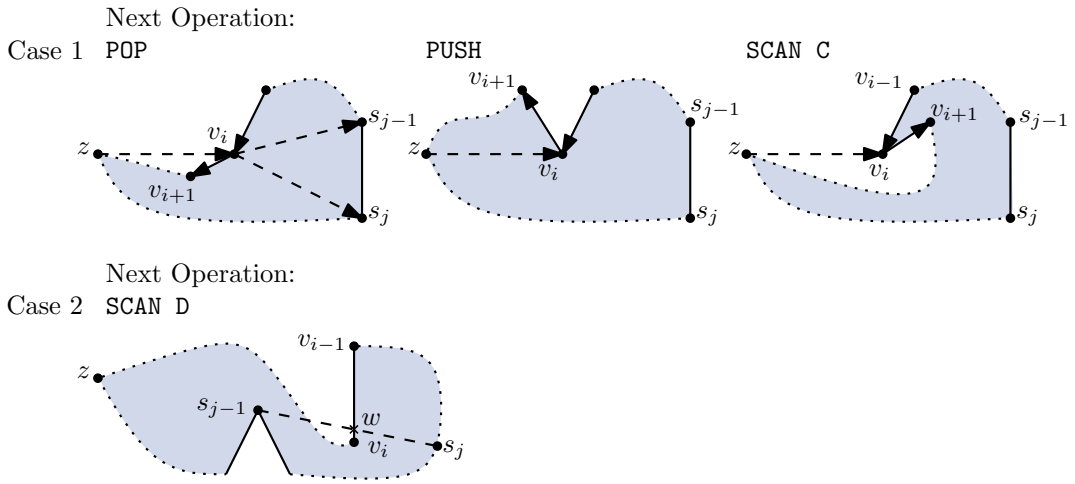


Figure 3.7.: In case 1 the turns zv_iv_{i+1} and $v_{i-1}v_iv_{i+1}$ determine the operation succeeding the POP operation. The SCAN_D operation follows always after case 2.

$v_k - z$ is a vector of length zero. Thus, the concepts of a left or right turn relative to this vector are undefined. A simple trick to avoid this ambiguity is to perturb the view point. We can, for example, temporally move the view point a small distance along the bisector of the incident edges. This way, we do not introduce a new intersection if the distance is small enough, and we can consistently decide whether or not two vectors make a left or a right turn. We make use of the same trick if the view point and two further vertices are collinear. The original algorithm does not specify the next operation in this case. Adding temporarily a little perturbation to the points helps to find the correct next operation.

We have to be careful in case of the POP operation. Figure 3.10 depicts a scenario in which moving vertices along the bisector is not helpful. With the collinear points z, v_i and v_{i+1} on the boundary, it is possible that we call two consecutive POP operations which have to carry out the same work. Both operations have to stop at the point s_t, i.e., case 1 of the POP operation has to apply. Unfortunately, in the second call zs_tv_i is not a right turn and thus the condition of case 1 does not hold. Note that the point s_t can be in the middle of the segment of the boundary. Thus, moving s_t along the bisector would move the point along the dotted line and does not resolve this problem. Fortunately, it is sufficient to extend the condition of case 1 with a collinearity check, i.e., condition of case 1 applies, or zs_jv_i is collinear, or $zs_{j-1}v_i$ is collinear. With this extended condition the second POP operation also stops at vertex s_t.

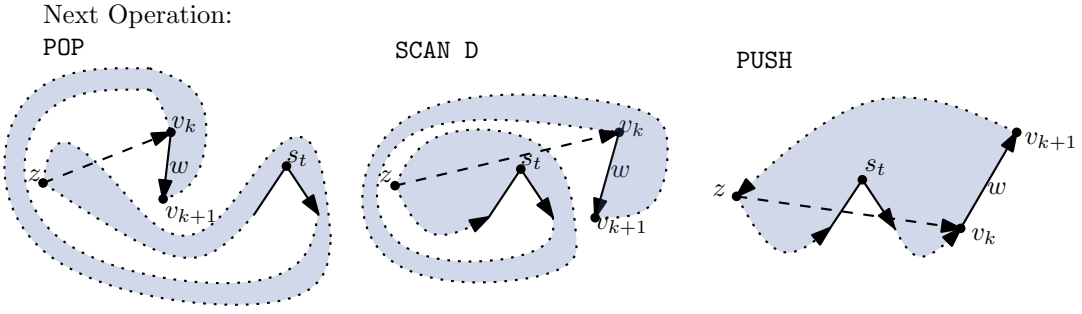


Figure 3.8.: Depending on the turn zv_kv_{k+1} and the relative position of the intersection w, we can determine the operation following SCAN_A.

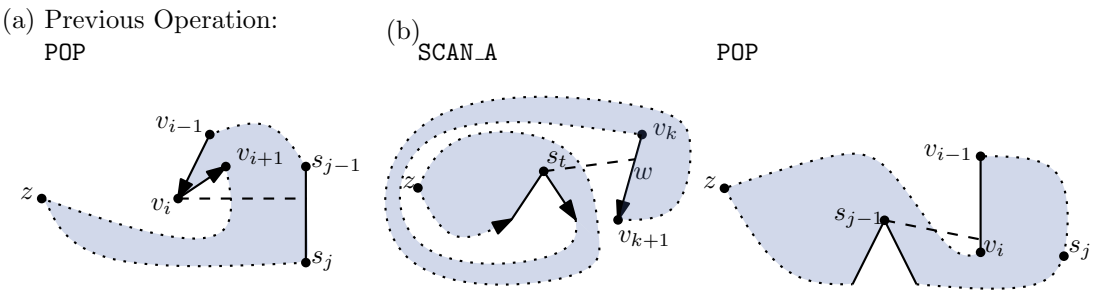


Figure 3.9.: (a) The SCAN_C Operation succeeds a POP operation. (b) A SCAN_A or a POP operation calls the SCAN_D operation. Both scans fast forward until they intersect with the dashed segment.

3.1.2. Planarity Region

Assume that we want to move a vertex v in a given planar drawing. We restrict the movement to those positions that do not introduce new crossings to the drawing. Note that only edges incident to v can get new crossing. Thus, the planarity region of the vertex v is related to the visibility of its neighbors.

Given a vertex v in the interior of a biconnected graph G, we define a region in which we can move v so that the embedding of a planar drawing of G does not change. This implies that we do not introduce new crossings to the drawing of G. We call this region the *planarity region* $\mathcal{PR}(v)$ of the vertex v.

Our approach to compute the planarity region of a vertex v is based on the observation that the planarity region is the intersection of the visibility regions of the neighbors of v in the surrounding of v. We describe the visibility region in terms of so-called *windows*. A window w is a segment in the visibility region that is not a (sub-)segment of the boundary of the original polygon $\mathcal{P}$. A window w divides the polygon $\mathcal{P}$ in one inner and one outer polygon, where the outer part is definitely not visible. We say the inner polygon $\mathcal{P}_w$ is left of w and the outer part is right of the window. We call the outer part of a window *pocket*; compare [BLM02]. Then the visibility region of a vertex v is the intersection of all inner polygons. Further, the intersection of all visibility regions is the intersection of all their inner polygons. Two windows of two different visibility regions can hide (dominate) each other.

We show that the number of windows necessary to compute the planarity region is linear in the size of the input polygon. Given the set of windows, we compute the planar subdivision of the windows with the input polygon and extract the face that is left of all windows.

Let $\mathcal{P}_N(v)$ be the intersection of all visibility regions of the neighbors of v in the surrounding of v.

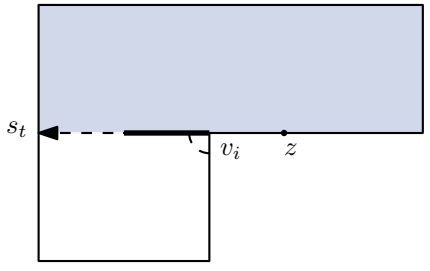


Figure 3.10.: Two consecutive POP operation where the second POP operation has to stop at vertex s_t as well.

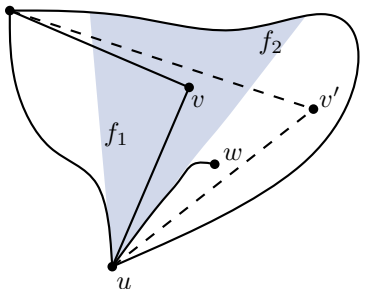


Figure 3.11.: As long as we stay in the planarity region of v, the rotational order of all vertices in f_1 and f_2 is invariant.

Theorem 3.1. *The planarity region $\mathcal{PR}(v)$ of a vertex v is the intersection of all visibility regions of the neighbors of v in the surrounding of v.*

Proof. First, we prove $\mathcal{PR}(v) \supseteq \mathcal{P}_{N(v)}$. By definition of the visibility regions, moving v within $\mathcal{P}_{N(v)}$ does not introduce new intersections to the drawing. Further, we show that the rotational order of every vertex on the surrounding is invariant. Thus, the rotational order of v is invariant. Consequently, the embedding of the graph does not change.

The rotational order of a vertex on the surrounding and for a neighbor of v that occurs only once on the surrounding cannot be changed by moving within the intersection of the visibility regions. Otherwise the vertex u is a neighbor v and occurs multiple times on the surrounding. The edges incident to v restrict the visibility; compare Figure 3.11. Thus, the rotational order of this vertex stays invariant.

Consequently, the rotational order of every vertex on the surrounding is invariant. Therefore, it is not possible to change the rotational order v itself. Hence, a vertex placement within the intersection of the visibility regions does not change the embedding of the graph.

Secondly, we show that $\mathcal{PR}(v) \subseteq \mathcal{P}_{N(v)}$. Moving v to any point $p \in \mathcal{PR}(v)$ does not change the embedding of the graph. This implies, that every neighbor v is visible from the point p. Thus, every neighbor can see the point p and the point p has to be in the intersection $\mathcal{P}_{N(v)}$ of all visibility regions. □

Computing the Planarity Region

Theorem 3.1 states that we can compute the planarity region by intersecting all visibility polygons. Each visibility polygon has at most $O(n)$ vertices [BLM02].

An upper bound on the number of intersections of ℓ visibility regions is $O(\ell^2 n)$. There are at most ℓn windows and each window can intersect at most $O(\ell)$ other windows. Thus, we can compute the intersection of the visibility regions in $O(\ell^2 n \log n)$ time. In the following, we show that we can reduce the number of intersections to $O(\ell n)$. We can thus reduce the running time to $O(\ell n \log n)$.

Not all visibility regions have to be intersected with each other. Instead, we extract all windows W of the visibility regions and compute the planar subdivision $\mathcal{P} \sqcap W$ of the surrounding polygon $\mathcal{P}$ with the windows W. Then, the face left of all window is the planarity region of the vertex v. Let $\mathcal{P}_w$ be the region left to the window $w \in W$ and let f be a face in the planar subdivision $\mathcal{P} \sqcap W$. We define the *point count* $c(p)$ for a point $p \in \mathcal{P}$ as the number of windows so that $p \in \mathcal{P}_w$. We show that the planarity region of v is never empty and a connected set of points. The vertex v is element of the planarity region, thus we can extract the face of the planar subdivision $\mathcal{P} \sqcap W$ that contains p, since every point in this face has the same face-count as p. In the following, we prove the stated claims. We assume that all segments and points are in general positions.

Lemma 3.2. *The planarity region of a vertex v is not empty.*

Proof. The vertex v is an element of the planarity region, thus the planarity region is not empty and the point count of v is $|W|$. □

Lemma 3.3. *Two points $p, q \in f$ in a face f of $\mathcal{P} \sqcap W$ have the same point count.*

Proof. Assume that the point count of both points differ, then there is a window w, so that $p \in \mathcal{P}_w$ and $q \notin \mathcal{P}_w$ or vice versa. Then w splits f into regions. This contradicts the assumption that f is a face in the planar subdivision $\mathcal{P} \sqcap W$. □

By Lemma 3.3, the *face count* $c(f)$ is the same as $c(p)$ for any point $p \in f$ and thus, the face count is well-defined. In other words, the face count $c(f)$ is equal to the number of windows $w \in W$ so that $f \subset \mathcal{P}_w$ (f is left of w). In the following we use k as the number of windows $|W|$.

Lemma 3.4. *The planarity region of a vertex is connected.*

Proof. Recall that we can compute the planarity region as the intersection of visibility regions. Every visibility region is the intersection of the regions $\mathcal{P}_w$, where w is a window of the visibility region. Thus, the planarity region is the intersection of all polygons $\mathcal{P}_w$ with $w \in W = \{w_1, w_2, \ldots, w_k\}$, where W is the set of all windows of all visibility regions. We can formulate the intersection of the polygons $\mathcal{P}_w$ as an iterative process.

Let $\mathcal{P}^i$ be the intersection of the polygons $\mathcal{P}_{w_1}, \mathcal{P}_{w_2}, \ldots, \mathcal{P}_{w_i}$ with $i = 1, 2, \ldots, k$. Each of these polygons is connected. The polygon $\mathcal{P}_{w_i}$ is connected since w_i divides the polygon $\mathcal{P}$ in exactly two parts, one (partially) visible part $\mathcal{P}_{w_i}$, and one definitely invisible face. Therefore, P^1 is connected. Assume that $\mathcal{P}^{i-1}$ is connected. We get $\mathcal{P}^i$ by intersecting P^{i-1} with $\mathcal{P}_{w_i}$. The intersection of both polygons cannot be empty due to Lemma 3.2. If one of both polygons contains the other completely, the intersection is trivially connected. Otherwise, if two windows intersect, the intersection divides each window into a visible and in an invisible segment. Thus, a window can contribute at most one edge to the planarity region and therefore to $\mathcal{P}^i$. Thus, w_i divides P^{i-1} into exactly two regions. Since P^{i-1} is connected, both regions have to be connected. Thus, P^i is connected. □

A face f is *incident to a window* w if f has an edge e that originates from the window $w \in W$, i.e., $e \subseteq w$.

Lemma 3.5. *If two faces are incident to the same window, the face count of the two faces in the planar subdivision $\mathcal{P} \sqcap W$ differs by 1.*

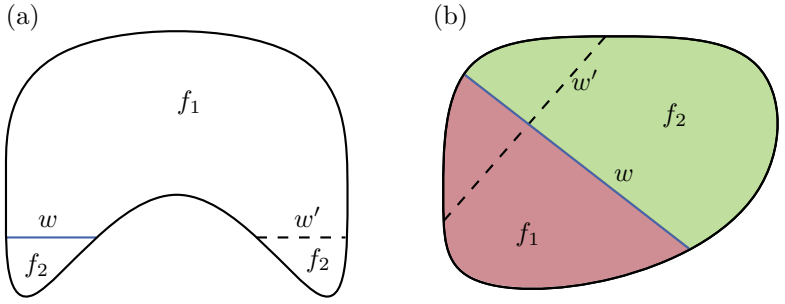


Figure 3.12.: (a) If the windows w and w' do not intersect, the face f_2 cannot be incident to both windows. (b) If w and w' do intersect and both faces are incident to both windows, the faces are not faces in the planar subdivision.

Proof. Let the two faces f_1 and f_2 be incident to the same window $w \in W$. Assume that there is a second distinct window $w' \in W$ so that the two faces are incident to w'. Recall that the endpoints of both windows are points on the boundary of the polygon. If the windows do not intersect, the planar subdivision $\mathcal{P} \sqcap \{w, w'\}$ has two distinct pockets as depicted in Figure 3.12(a). Thus, one of both faces cannot be incident to both windows.

If both windows intersect, the window w' splits both faces f_1 and f_2. Thus, the two faces are not faces of the planar subdivision $\mathcal{P} \sqcap \{w, w'\}$; compare Figure 3.12(b). Hence, there is only one window incident to both faces, and either f_1 or f_2 is a subset of $\mathcal{P}_w$. Accordingly, the face count of both faces differs by exactly 1. □

Theorem 3.6. *Let v be a vertex with the surrounding $\mathcal{P}$ and let W be the set of all windows of the visibility regions of the neighbors of v. Then the unique faces in the planar subdivision $\mathcal{P} \sqcap W$ with face count k is the planarity region of v.*

Proof. Let f be a face in the planar subdivision $\mathcal{P} \sqcap W$ with face count k. The inclusion $f \subseteq \mathcal{PR}(v)$ follows directly from the definition of the face count $c(f)$. There are exactly k distinct polygons $\mathcal{P}_w$ that contains f (f is left of all windows). Thus, f is a subset of the planarity region.

We show that $f \supseteq \mathcal{PR}(v)$. By Theorem 3.2 the planarity region $\mathcal{PR}(v)$ is not empty. Further, the planarity region is the intersection of all polygons $\mathcal{P}_w$ with $w \in W$. Thus, there is a face f in the planar subdivision $\mathcal{P} \sqcap W$ so that $\mathcal{PR}(v) \subseteq f$. Every point $p \in \mathcal{PR}(v)$ has a point count of k. Since every point in f has the same point count, as Theorem 3.3 states, the face count of f is k. □

In order to compute the planarity region, it is sufficient to extract the face f in the planar subdivision $\mathcal{P} \sqcap W$ with face count k. We know that the vertex v is within the planarity region and thus has a point count of k. Therefore, we have to find the face f with v in the interior of f. A simple approach to find this face is to shoot a ray starting at v in any direction. The window with the smallest distance to v that intersects the ray is incident to the planarity region. Thus, we can traverse the face left of the window to compute the planarity region of v.

Running Time

We conclude this section with an analysis of the running time of our algorithm. Recall that n is the number of vertices of the polygon, k the number of windows, and ℓ the number of view points. We prove two main results. First, we can limit the number of relevant (dominating) windows to a linear factor in the size of the polygon. Secondly, the number of intersections between the windows is at most $O(\ell n)$.

Each concave corner can yield a window of a visibility region [BLM02]. There can be $O(n)$ concave corners in a polygon, e.g., a star-shaped polygon. In this case, all windows of all

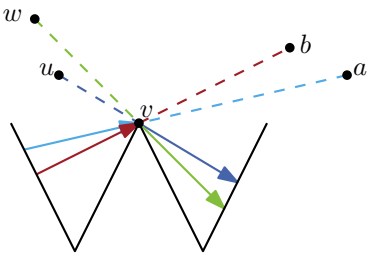


Figure 3.13.: Each concave corner has one dominating incoming and outgoing window.

visibility regions can be distinct from each other resulting in a total number of ℓn windows, where ℓ is the number of visibility regions. Windows belonging to the same view point cannot intersect with each other. Thus, a window can intersect at most ℓ other windows. This leads to $O(\ell^2 n)$ intersections. In order to compute the intersection of all the polygons directly, a sweep line algorithm has to encounter all these intersections points [CLRS09]. Figure 3.13 shows that not all windows are necessary to compute the intersection of all visibility regions.

In the following, we show that not all windows are necessary to compute the planarity region. Let $W(v)$ be the set of all windows of all visibility regions incident to the vertex v. We can partition $W(v)$ into two sets $W_{in}(v)$ and $W_{out}(v)$. Recall that the windows are directed in the same way as the input polygon, i.e., the visibility region is left to a window. The set $W_{in}(v)$ contains all windows w where v is the target of w. Consequently, $W_{out}(v)$ contains the windows that have v as the source. In each of these sets, there is one window corresponding to a view point q *dominating* the others, i.e., crossing this window would definitely break the visibility of the view point q, but not necessarily of the other view points (due to another concave corner the visibility can already be broken). Therefore, we have to store only one incoming and outgoing window per concave corner. We can check whether or not a window is dominating by looking at the rotational order of the windows around v. This requires only linear time in the number of windows incident to v. Note that after extracting a window one can directly decide whether or not the window is dominating.

Corollary 3.7. *The planarity region of a vertex v has $O(n)$ vertices, where n is the number of vertices of the surrounding* $\mathrm{surr}(v)$.

Proof. Every window w does not intersect, except at their endpoints, with the boundary $\mathcal{P}$. If it intersects with another window w', the window w' divides w into one visible and one invisible segment. Therefore, every window w attributes at most one segment to the boundary of the planarity region. Since, there are at most $O(n)$ dominating windows, the planarity region has at most $O(n)$ vertices. □

Theorem 3.8. *The number of pairwise intersections of all windows W is $O(\ell n)$, where n is the number vertices of the surrounding* $\mathrm{surr}(v)$ *and ℓ the number of visibility regions.*

Proof. Windows belonging to the same visibility region cannot intersect. Let $w \in W$ be a window and let z be the corresponding view point. The pocket of the window is a *left pocket* if it lies left of the ray starting at z and containing w. Otherwise, it is a *right pocket*. A *left window* bounds a left pocket and a *right window* bounds a right pocket.

A window w cannot intersect with two left (right) pockets of the same visibility region. Therefore, a window of a view point can intersect with at most $O(\ell)$ windows. There are at most $O(n)$ windows per view point. Thus, in the worst case, the total number of intersections is $O(\ell n)$. □

Theorem 3.9. *Let n be the number of vertices on the surrounding of v and let ℓ be the number of neighbors of v. Computing the planarity region of a vertex v can be done in $O(\ell n \log n)$ time.*

Proof. Computing all visibility regions requires $O(\ell n)$ time. Extracting the set of dominating windows W requires $O(\ell n)$ time.

The pairwise number of intersections in W is $O(\ell n)$. The number of intersections of W with $\mathcal{P}$ is $O(n)$. Therefore, the overall number of intersections i is in $O(\ell n + n)$. Computing

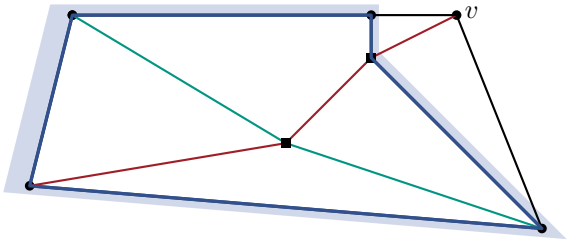


Figure 3.14.: The surrounding of a vertex on the external face is unbounded.

the planar subdivision $\mathcal{P} \sqcap W$ with a sweep-line approach takes at most $O(n \log n + i \log n)$ time [BO79].

The planar subdivision has $O(\ell n)$ vertices. We can divide a window $O(\ell)$ times. Thus, the number of edges in the planar subdivision is $O(\ell n)$. Therefore, we can extract the planarity region in $O(\ell n)$ time from the planar subdivision.

In worst case, computing the planarity region of a vertex requires $O((n+i)\log n) + \ell n \subset O(\ell n \log(n))$ time. □

3.1.3. Planarity Region of Unbounded Faces

Until now we assumed that the vertex v is not on the external face. The surrounding of a vertex is the union of all incident faces of this vertex. Thus, the surrounding of a vertex on the external face of a graph is unbounded as depicted in Figure 3.14

Our planarity region algorithm only works on bounded polygons. Thus, we restrict the external face, for example, with a bounding box. Further, our planarity region and the visibility regions algorithm requires that the polygon has no holes. Therefore, we have to connect the surrounding with a polygon that contains $\mathcal{P}$. We compute the bounding box $\mathcal{B}$ of $\mathcal{P}$ and scale it up by a factor β. We use this bounding box to limit the drawing area.

We have to find a segment s only intersecting at its endpoints with the boundaries of $\mathcal{P}$ and $\mathcal{B}$ so that we do not restrict the planarity region of the polygon $\mathcal{B} \setminus \mathcal{P}$. Let u and w be two vertices on the external face of the drawing of G adjacent to v. Since G is biconnected, u and w are distinct. The surrounding $\mathcal{P} = \text{surr}(v)$ is a compound of two paths. One path p_{ext} consists solely of vertices on the external face and the other path p_{int} of vertices in the interior of G. Note that we cannot move v around u or w and cross the line $\mathcal{L}(u, w)$ without interfering with the planarity region $\mathcal{PR}(v)$ from v; compare Figure 3.15. Therefore, we can

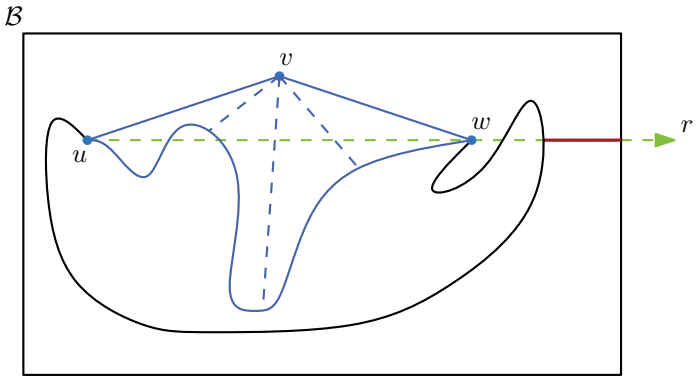


Figure 3.15.: The blue polyline depicts the internal path, the black polyline consists of nodes on the external face. Points on the black polyline right to l are not simultaneously visible from u and w.

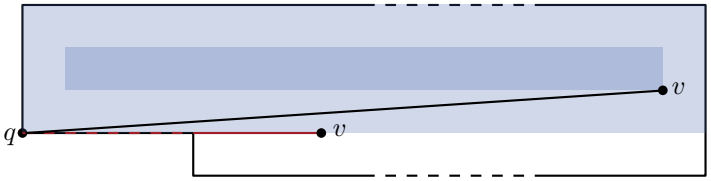


Figure 3.16.: Positions on the boundary of the planarity region may corrupt the planarity. Even after offsetting the planarity region (dark blue) the minimum vertex – edge distance is arbitrarily bad.

shoot a ray r from u through w and intersect it with p_{ext} and $\mathcal{B}$. The shortest subsegment of r connecting $\mathcal{P}$ and $\mathcal{B}$ does not restrict the planarity region of v in $\mathcal{B} \setminus \mathcal{P}$. We call this subsegment a *bridge* between $\mathcal{P}$ and $\mathcal{B}$. One endpoint of the bridge can occur multiple times on the boundary of $\mathcal{P}$. In order to find the correct intersecting segment on the boundary of the polygon, we can temporally perturb the vertices of the polygon.

3.1.4. Offsetting the Planarity Region

In each iteration we place a vertex v in its planarity region. If we place a vertex on the boundary of the planarity region, we can corrupt the planarity due to two overlapping edges; see Figure 3.16. In order to distinguish between a point on the boundary of planarity region and a point very close the planarity region, the coordinates can require a high precision. In order to avoid this scenario, we shrink the planarity region. Nevertheless, if the vertex v has already a good position (with small crossing-angles), the shrinking process should not exclude this position. Note that shrinking the planarity region controls only the distance of the vertex v to edges of the surrounding. Nevertheless, Figure 3.16 illustrates a case, where the distance of a vertex on the surrounding to an edge incident to v can be arbitrarily small.

We compute two offset distances. The first distance, is the distance of the current position of the vertex to the planarity region. The second distance, is a relative measure. We use the minimum of both values to compute the offset polygon of the planarity region. In this way, we can guarantee that the current position is in the shrinked planarity region and if the current position is near the center of the planarity region, the shrinked region is not too small.

In order to guarantee that the current position is in the shrinked region, we can compute the minimal distance $\gamma_{\min}$ to all edges on planarity region of v. This distance ensures that the vertex v is an element of the planarity region after offsetting it by the value $\gamma_{\min}$. If

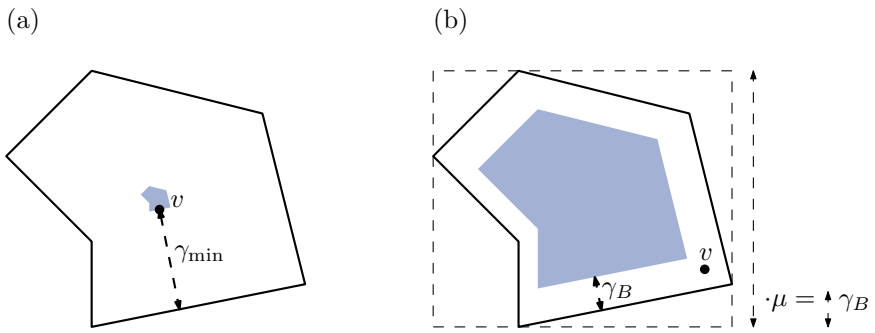


Figure 3.17.: (a) Offsetting the planarity region by the distance to the vertex v can result in a small region. (b) Offsetting by a constant factor can exclude the previous position.

the vertex is already near the center of the planarity region, offsetting the planarity region by $\gamma_{\min}$ yields a small region with only few possibilities to optimize the position of the vertex; see Figure 3.17(a). In order to avoid this, we compute a second, relative, distance. Let $\mathcal{B}$ be the bounding box of the planarity region of the vertex v. Let γ_B be the minimal edge length oft the bounding box times a value $\mu \in (0, 1)$. This offset distance guarantees a relative distance of v to the edges on the planarity region but it cannot guarantee that the original position lies in the planarity region after offsetting it by the value γ_B; compare Figure 3.17(b). We use the minimum value of $\gamma_{\min}$ and γ_B as the offset distance γ. This value guarantees that the previous position is in the planarity region after offsetting it and if the vertex lies central in the planarity region, the offsetting process leaves enough space to minimize the active crossing angles of v.

3.2. Vertex Placements

In the previous section, we described the set of legit positions for a vertex v as the planarity region of the vertex. Given this region, or a subset of this region, we can think about how to place a vertex to a locally optimal position with respect to this region. In Section 3.2.1 we handle the placement of a tail vertex. We continue with the placement operation for dummy vertices in Section 3.2.2. In Section 3.2.3, we combine both operations for dummy vertices with dummy neighbors. Finally in Section 3.2.4 , we introduce two operations to place an independent vertex.

3.2.1. Placing a Tail Vertex

A Tail vertex v has a set of dummy neighbors $D(v)$. There can multiple planarization paths that touch at the vertex v. The new position of the vertex v should optimize its position with respect to its dummy neighbors and the corresponding dissected pairs.

With the planarity region we can characterize all positions of a vertex that do not change the embedding of a drawing. Within this set of positions we have to find a position for the vertex v so that all related active crossing angles α_i are minimized. These angles are introduced by all dissected pairs incident to v. Figure 3.18 depicts all related angles of a tail vertex. Each dissected pair incident to the vertex gives us a desired direction d_q. In order to improve the angle of one dissected pair, we would like to place the vertex on a ray in the desired direction d_q. Of course, the rays of all dissected pairs do not have to intersect in one point, and even if they do, the intersection does not have to be within the planarity region. Thus, this may result in an optimization problem with possibly conflicting constraints.

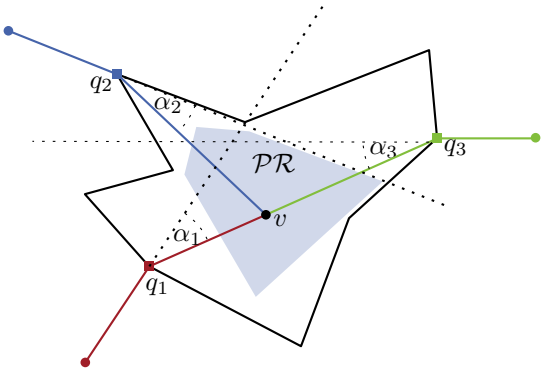


Figure 3.18.: We have to place the vertex v in such manner in the planarity region $\mathcal{PR}$ so that the angles α_i are minimized.

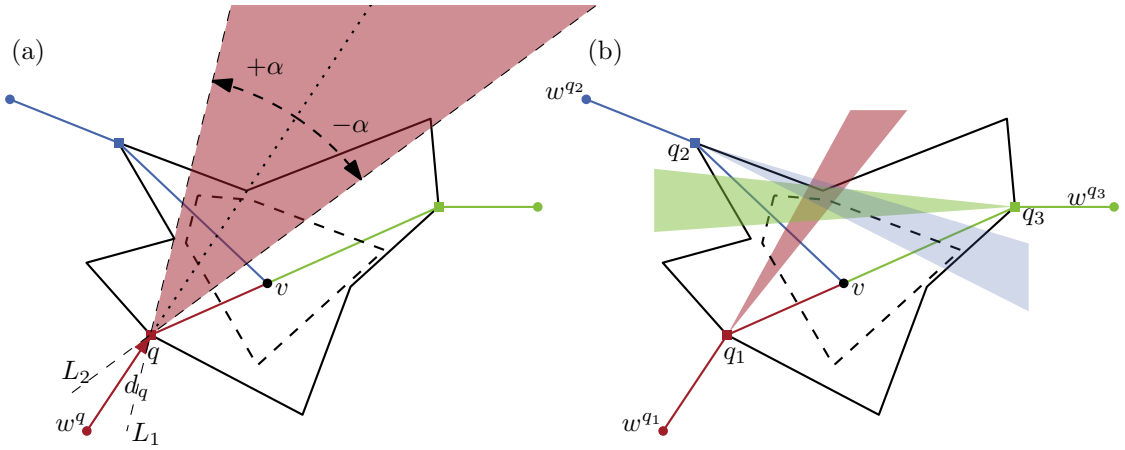


Figure 3.19.: (a) A cone with respect to one neighbor q of v. (b) The intersection of all cones with the planarity region (dashed) admits possible legit positions for the vertex v.

One way to handle the conflicting constraints is to minimize the maximum crossing angle of the vertex v. A general approach to this problem is to perform a binary search over the angles $[0, \pi)$. For each step in the binary search we can check whether or not we can place all vertices with respect to the given angle. If the check fails for one angle, there cannot be a smaller angle for which the check passes. Hence, the underlying binary decision function is monotone as required for Theorem 2.1. Therefore, the upper-bound of the binary search is an absolute ϵ-approximation for the minimal crossing angle after $O(\log(\pi/\epsilon))$ iterations. In the following, we show how to check whether or not there is a position for the vertex v for a given angle. The idea of our approach is to define a cone for each dummy neighbor with respect to a given angle. If the intersection of all cones is not empty, then there is a feasible position for the vertex v.

Let $D(v) \subseteq N(v)$ be the set of dummy neighbors of v. For each dummy neighbor $q \in D(v)$ there is a dissected pair (w^q, q, v) incident to q and v. Ideally, we can place the vertex v collinearly with w^q and q for all $q \in D(v)$. Thus, the vertices w^q and q give us a desired direction $d_q = q - w^q$. Since, there might not be a legit position which satisfies the collinearity for all q, we minimize the maximum angle $\angle w^q q v$ for every $q \in D(v)$, so that the vertex v is in the planarity region $\mathcal{PR}(v)$. We refer to this optimization problem as MinMax-Tail.

Let $\alpha \in [0, \pi)$ be the angle of the current step in the binary search. For two lines L^1 and L^2, we define a cone $\mathcal{C}(L^1, L^2)$ as the intersection (union, for an angle $\alpha \geq \pi/2$) of two half-planes $\mathcal{HP}(L^1)$, $\mathcal{HP}(L^2)$. For each $q \in D(v)$ we construct a cone $\mathcal{C}(q, \alpha) = \mathcal{C}(L^1, L^2)$ where the line L^1 (L^2) is obtained by the rotation of the lines $\mathcal{L}(q, -d_q)$ ($\mathcal{L}(q, d_q)$) about α ($-\alpha$) degrees around q.

Let $\mathcal{C}(D(v), \alpha) = \bigcap_{q \in D(v)} \mathcal{C}(q, \alpha)$ be the intersection of all such cones. If the intersection $\mathcal{C}(D(v), \alpha) \cap \mathcal{PR}(v)$ is not empty, the intersection results, by construction, in a set of legit positions for p so that each angle $\angle w^q q v$ is smaller or equal to α. Figure 3.19(a) shows this relationship.

Computation of $\mathcal{C}(D(v), \alpha) \cap \mathcal{PR}(v)$

In the previous paragraph, we introduced the basic concept of a placing a tail vertex v. In this paragraph, we show how to compute the intersection of the cones $\mathcal{C}(D(v), \alpha)$ with the planarity region $\mathcal{PR}(v)$. We divide this process into two phases. First, we compute the intersection $\mathcal{C}(D(v), \alpha)$ of all cones $\mathcal{C}_i$. Given the intersection of the cones, we can

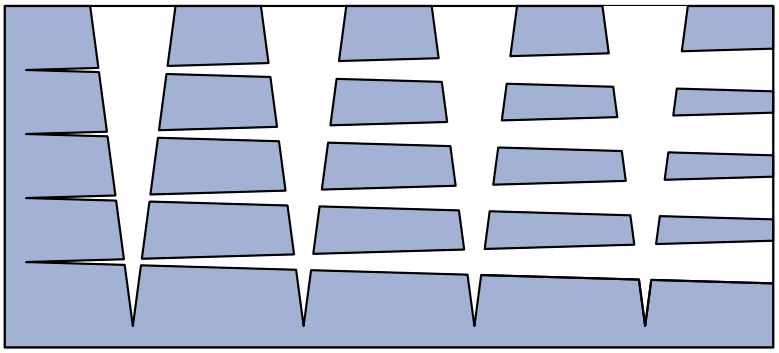


Figure 3.20.: For an angle $\alpha > \pi/2$ there is a setting of ℓ cones with ℓ^2 polygons describing the intersection of the cones.

compute the intersection with planarity region. This step is a simple application of the sweep line algorithm for line segment intersections [BO79]. Let α be the current angle. We denote the corresponding cones with $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots . \mathcal{C}_\ell\}$. Let the cone $\mathcal{C}_i$ be $\mathcal{C}(q_i, \alpha)$ with $D(v) = \{q_1, q_2, \ldots, q_\ell\}$.

The intersection of the polygon $\mathcal{PR}(v)$ with a cone $\mathcal{C}_i$ restricts the intersection of the cone with $\mathcal{PR}(v)$. Therefore, we can intersect each $\mathcal{C}$ with the bounding box $\mathcal{B}$ of $\mathcal{PR}(v)$. We can intersect these restricted cones with each other without loosing information. We use a Divide-and-Conquer approach to intersect all cones. We partition the set $\mathcal{C}$ into two equal sized sets $\mathcal{C}^a$ and $\mathcal{C}^b$. If C contains at most one element, the intersection is trivial. Otherwise, we compute the intersection of the cones in each set C^a and C^b recursively. Hence, the intersection of the cones in C is the intersection of $\mathcal{C}^a$ and $\mathcal{C}^b$.

In general, we can compute the intersection of two simple polygons with n_1 and n_2 vertices, respectively, with a sweep line algorithm in $O((n_1 + n_2 + i)\log(n_1 + n_2))$ time where i is the number of intersections [NP82]. We show, that the cost of intersecting all cones is no more then $O(\ell^2 \log \ell)$. First of all, notice that all cones share segments on the bounding box $\mathcal{B}$. The intersection of these segments contributes only a linear term to the number of intersection of all cones. The intersection of the lines ℓ_i^1 and ℓ_i^2 that define the cones dominate the number of intersections. In the worst case, the number of intersections is $O(l^2)$. Let $T(\ell)$ be the time of intersecting ℓ cones with our Divide-and-Conquer approach. Then we can express $T(\ell)$ as $T(1) = 1$ and $T(\ell) = 2T(\ell/2) + O(\ell^2 \log \ell)$. Thus, the Master Theorem [CLRS09] yields the previous stated result $T(\ell) \in O(\ell^2 \log \ell)$. Note that if the angle α is smaller than $\pi/2$, then the cones are convex and thus we can compute the intersection of the cones in linear time [OCON82].

We can use a sweep-line algorithm to determine whether or not $\mathcal{PR}(v)$ and $\mathcal{C}$ intersect.

Theorem 3.10. *Determining whether or not ℓ cones and a polygon with n vertices intersect requires $O((n+\ell^2)\log(n+\ell))$ time. One step of the binary search takes $O((n+\ell^2)\log(n+\ell))$ time.*

Proof. The intersection of all cones is the union of at most $O(\ell^2)$ polygons with a total of $O(\ell^2)$ edges; compare Figure 3.20. Thus, determining whether or not the cones intersect takes $O((n + \ell^2)\log(n + \ell^2))$ time.

We can compute the intersection of all cones in $O(\ell^2 \log \ell)$ time. Computing the planarity region can be done prior to the binary search. Thus, the total time required per step of the binary search is $O((n + \ell^2)\log(n + \ell))$. □

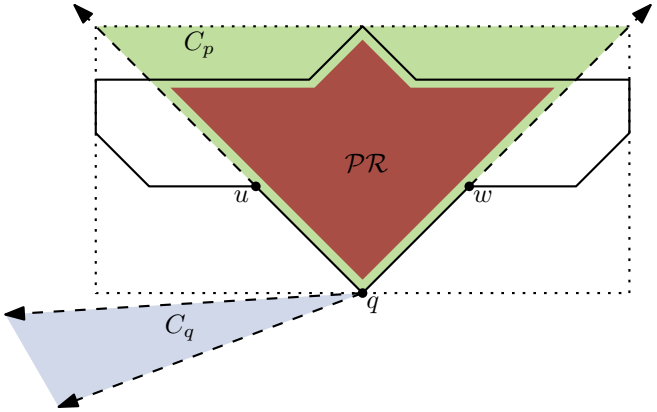


Figure 3.21.: The point q acts as the source for the cone (blue) and influences the planarity region and therefore $\mathcal{P}'$ (red). The cone $\mathcal{C}_p$ (green) starting at point q is bounded by the rays trough u and w restricts the planarity region. If the intersection of both cones is only the point $\{q\}$, the intersection with the planarity region is empty.

Corollary 3.11. *Given the planarity region, for any $\epsilon > 0$ we can compute an absolute ϵ-approximation of the* MinMax-Tail *problem in time $O(\log(1/\epsilon)(n+\ell^2)\log(n+\ell))$ where $\ell = |D(v)| \leq \deg(v)$.*

Proof. As already stated, the underlying binary decision function of our binary search is monotone. Thus, the upper-bound of the binary search is an absolute ϵ-approximation of the minimal angle by Theorem 2.1 after $O(\log(\pi/\epsilon))$ iterations. Each step of the binary search requires $O((n+\ell^2)\log(n+\ell))$ time by Theorem 3.10. This yields a running time of $O(\log(\pi/\epsilon)(n+\ell^2)\log(n+\ell))$. □

In practice, we can heuristically decrease the running time of our algorithm. A ray through q and a concave corner restrict the planarity region. Therefore, we can define a cone $\mathcal{C}_p$ starting at vertex q with two rays defined by the incident edges of q as shown in Figure 3.21. If the two cones $\mathcal{C}_p$ and $\mathcal{C}_q$ only intersect at their source point q, the intersection of $\mathcal{C}_q$ with the planarity region is empty as well. Therefore, the intersection of the cone $\mathcal{C}_q$ with $\mathcal{PR}(v)$ is empty. Accordingly, we can reject some configurations in $O(\ell)$ time.

3.2.2. Placing a Dummy Vertex

If we have to place a dummy vertex v (without dummy neighbors), there are only two active crossing angles incident to v. Let $D(v) = \{a, p, b, p\}$ be the neighbors of v so that a, b and p, q are incident to the same dissected pair. One might be tempted to place v on the intersection of the segments $\mathcal{S}(a, b)$ and $\mathcal{S}(p, q)$. However, in general the intersection may not be within the planarity region. Figure 3.22 shows that the optimal crossing angle for each dissected pair can conflict with each other. Thus, we take a similar approach to placing a tail vertex. We perform a binary search over the angles $[0, \pi)$. In each step, we check whether or not there is a position for v so that the crossing angle is at most α. This results in an absolute ϵ-approximation of the incident crossing angles. We refer to the underlying optimization problem as MinMax-Dummy. Let α' be the angle of a step in the binary search. Our binary decision function works on the angle $\alpha = \pi - \alpha'$. For a given angle α we make use of the *Theorem of Thales*, which states that moving on the boundary of the union (intersection) of two disks does not change the angle.

We have exactly two active crossing angles, namely the angle between va and vb and the angle between vp and vq. First, we consider only the angle α between va and vb bounded

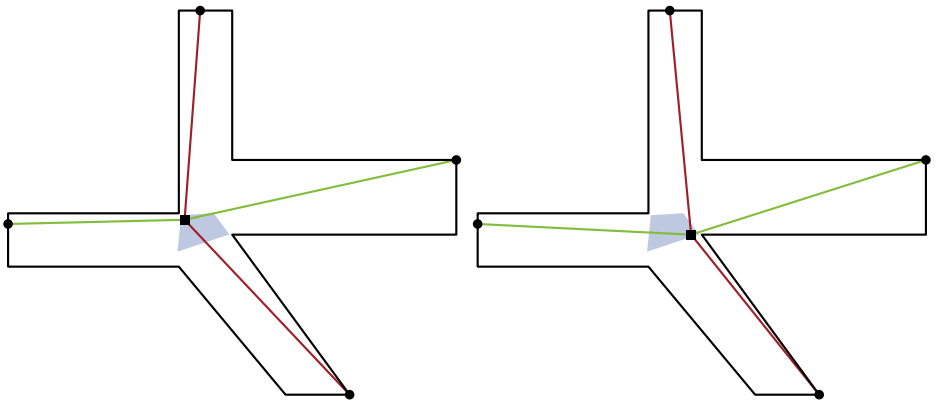


Figure 3.22.: The planarity region restricts the movement of the vertex v. A position near the top line improves the crossing angle of the green dissected pair. The optimal position for the red dissected pair is in lower right corner of the planarity region.

from below by the fixed angle $\alpha_0 \leq \alpha$. We distinguish the three cases $\alpha_0 < 90°$, $\alpha_0 = 90°$, and $\alpha_0 > 90°$.

Lemma 3.12. *The angle $\angle avb$ is at least α for $\alpha > 90°$ (for $\alpha \leq 90°$) if v lies in the intersection (union) of the two disks with a and b on their boundary and radius $|ab|/(2\sin(\alpha))$.*

Proof. We show the lemma for $\alpha > 90°$, the case $\alpha \leq 90°$ works analogously (in case $\alpha = 90°$, the two disks are the same and thus union and intersection are the same).

Consider a circle with center O and radius r containing a, b, and v such that v lies on the shorter of the two circular arcs between a and b (which ensures that $\angle avb > 90°$); see Figure 3.23(a). We first show how we have to choose the radius r so that $\angle avb = \alpha$. Let c be the center of the line segment ab. The angle α is the inscribed angle formed by the two secant lines through av and through bv. Thus, it is half the central angle β (which is the angle at o in the outside of the quadrangle $avbo$; see Figure 3.23(a)). Let γ be the angle at o in the right triangle boc. We have $\gamma = (360° - \beta)/2 = 180° - \alpha$. Moreover, the right triangle boc yields $\sin(\gamma) = |ab|/(2r)$ which is equivalent to $r = |ab|/(2\sin(\gamma))$. Plugging in $\gamma = 180° - \alpha$ yields $r = |ab|/(2\sin(\alpha))$.

For a larger radius r the angle $\angle avb$ is larger than α, and for a smaller radius it is less than α. Thus, $\angle avb$ is at most α for $\alpha > 90°$ if v lies in the intersection of the two disks with a and b on their boundary and with radius $|ab|/(2\sin(\alpha))$; see Figure 3.23(b). The same arguments work for the case where $\alpha \leq 90°$. In this case v has to lie on the larger of the

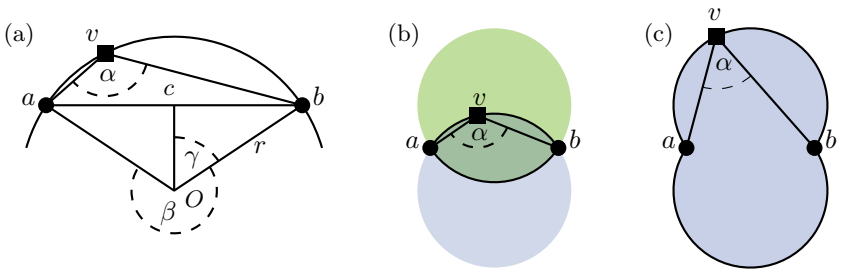


Figure 3.23.: (a) Notation for the proof of Lemma 3.12. (b)/(c) The angle $\angle avb$ is at least α for $\alpha > 90°$/for $\alpha < 90°$ if and only if v lies in the intersection (union) of the green and blue region (including its boundary, but excluding a and b) .

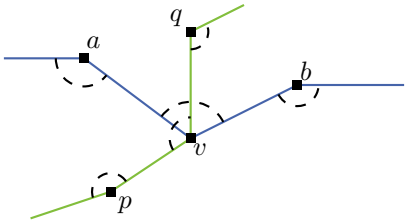


Figure 3.24.: All possible active crossing angles around a dummy vertex v

two circular arcs between a and b, thus we have to take the union of two disks instead of their intersection; see Figure 3.23(c) for the resulting region. □

The same applies for $\angle pvq$. Thus, restricting both active crossing angles $\angle avb$ and $\angle pvq$ from below restricts the possible positions v of the dummy vertex v either to the intersection of four disks, or to the intersection of the union of two disks with the union of two other disks. More formally, we have four disks $D_1, \ldots, D_4$ and we have to check whether either $D_1 \cap \cdots \cap D_4$ or $(D_1 \cup D_2) \cap (D_3 \cup D_4)$ has non-empty intersection with the planarity region $\mathcal{PR}(v)$. In the latter case, this is equivalent to testing whether $\mathcal{PR}(v)$ has a non-empty intersection with at least one of the regions $D_1 \cap D_3$, $D_1 \cap D_4$, $D_2 \cap D_3$, or $D_2 \cap D_4$. The check whether or not this intersection is empty or not requires linear time in the size of the planarity region.

If this intersection is empty for a given angle in the binary search, we can infer that there is no larger angle that admits a non-empty intersection. Thus, the corresponding decision function for the binary search is monotone as required for Theorem 2.1. Therefore, the lower bound is an absolute ϵ-approximation of the optimal angle.

Corollary 3.13. *Given the planarity region, for any $\epsilon > 0$ we can compute an absolute ϵ-approximation of the* MinMax-Dummy *problem in $O(n \log(1/\epsilon))$ time.*

Proof. One step of the binary search requires of $O(n)$ time. The underlying binary decision function is monotone as required for Theorem 2.1. Therefore, the upper bound of the binary search is an absolute ϵ-approximation after $O(\log(\pi/\epsilon))$ iterations. □

3.2.3. Placing a Dummy Vertex with Dummy Neighbors

In general, we cannot distinguish between a dummy vertex and a tail vertex, i.e., all four or some of the neighbors of a dummy vertex can be dummy vertices. Figure 3.24 depicts all possible active crossing angles.

We combine the cone construction for the binary search of Section 3.2.1 with the disk construction of Section 3.2.2 in one joint binary search. Thus, our binary decision function checks in each step of the binary search whether or not the cones of all dummy neighbors of v and the disks of the dissected pairs with v in the middle admit a non-empty intersection. As seen before, the binary decision function is still monotone and yields an ϵ-approximation of all active crossing angles at v. Since there are at most four dummy neighbors, the time to check whether or not the intersection is empty is linear in the size of the planarity region of v. We refer to the underlying optimization problem as the MinMax-Hybrid.

Corollary 3.14. *Given the planarity region, for any $\epsilon > 0$ we can compute an absolute ϵ-approximation of the* MinMax-Hybrid *problem in $O(n \log(1/\epsilon))$ time.*

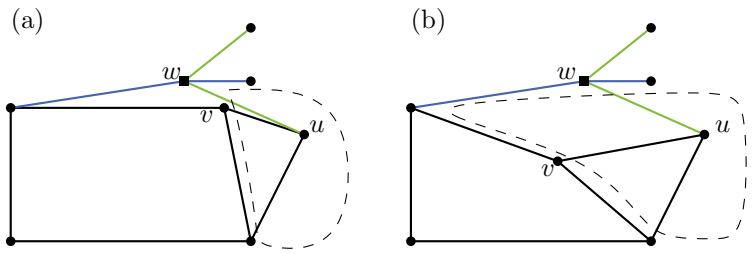


Figure 3.25.: (a) The current position of v blocks the view of the tail vertex u. Thus, hindering to optimize the active crossing angle of u at the dummy vertex w. (b) Placing the vertex v in center of its planarity region clears the space for the tail vertex u.

3.2.4. Placing an Independent Vertex

A vertex v does not have to be a dummy vertex or a tail vertex. Thus, moving this vertex does not change any active crossing angle. Nevertheless, the position of an independent vertex v can block the view of a tail or dummy vertex on the surrounding of v as depicted in Figure 3.25. This restricts the movements of the vertices on the surrounding of v. Thus, the crossing angles of these vertices are indirectly restricted by the placement of v. It is not clear, what a good optimization problem for the independent vertices is. Our idea is to clear the space for all vertices on the surrounding. In order to circumvent this problem we introduce two possibilities to place the vertex.

Geometric Center

The idea of the Geometric Center approach is to place the independent vertex v in the center of its planarity region. We define the center of the planarity region as the non-empty offset polygon with the largest offset distance o. In practice, we can compute this center of the planarity region by iteratively offsetting the polygon.

Let $\mathcal{B}$ the bounding box of $\mathcal{PR}(v)$ and d-dist($\mathcal{B}$) be the diagonal distance from the lower left to the upper right corner of the bounding box. For a given distance o we can check whether or not this distance yields an empty or non-empty offset polygon. If one distance does creates an empty offset polygon any greater distance results in an empty offset polygon. Thus, we use this as a monotone binary decision function as required of a binary search in Theorem 2.1. Note that the meaning of the lower and upper bounds is exchanged.

Theorem 3.15. *Given the planarity region, for any $\epsilon > 0$ we can place a vertex in the geometric center of the planarity region with a tolerance in $O(\log(\text{d-dist}(\mathcal{B})/\epsilon)n^2)$ time.*

Random Center

We do not have any quality guarantees for the Geometric Center heuristic. Thus, we introduce an alternative randomized approach to place an independent vertex. We place the vertex at random within the planarity region of the vertex. In order to sample the planarity region uniformly at a random, we compute the triangulation of planarity region of v. We randomly select a triangle with a probability proportional to its area and then we utilize the barycenter coordinates to pick a point within a triangle uniformly at random. In order to select a triangle uniformly at random, we assign an arbitrary order to the triangles. We compute the prefix sum of the areas of the triangles with respect to this order. We compute a number p uniformly random between 0 and the area of the polygon. We select a triangle so that the number p is between two consecutive prefix sums.

Theorem 3.16. *Given the planarity region, it takes $O(n \log n)$ time to place a vertex uniformly at random in the planarity region.*

Proof. The time of the triangulation dominates the total time. A practical algorithm to triangulate a simple polygon requires $O(n \log n)$ time [GJPT78]. Computing a probability distribution and selecting a triangle according to the distribution can be done in linear time. Computing a pseudorandom number can be done constant time [Knu97]. Computing a point uniformly at random within a triangle takes constant time. □

4. Drawing a Planarization

In this chapter, we present two algorithms to draw a planarization. First, we refine our iterative Geometric Planarization Drawing approach and discus free parameters. We use the Geometric Framework of Chapter 3 to iteratively place a vertex. In Section 4.1, we introduce different vertex orders and a heuristic to improve the final drawing of the planarization, a so-called angle relaxation for the cone construction. In Section 4.2, we modify a spring-embedder called PrEd to draw a planarization. PrEd is able to preserve the embedding of an initial drawing [Ber99, SAAB11]. We introduce new forces to the system to optimize the crossing angles of the planarization.

4.1. Geometric Planarization Drawing

Our *Geometric Planarization Drawing* approach is in a sense similar to a force-directed method. A single iteration of a force-directed method computes forces for all vertices and applies them simultaneously to all vertices. This process is repeated until a stable layout of the graph is found. We consider a drawing as stable if the maximum difference between the crossing angle of two consecutive iterations is smaller then a threshold T. Algorithm 4.1 outlines our approach. We choose some order in which we place vertices. After a placement of a vertex, we can update this order and we allow vertices to occur multiple times in this order. Given a vertex, we compute the offsetted planarity region. We apply the placement operations according to the type of the vertex. The operations discussed in the previous chapter worked with the local properties of the drawing. In this section, we discuss how to assemble this operations and take a global view on the problem. How do we choose an initial layout? In which order should we place the vertices? Should unplaced vertices have the same influence on placing a vertex as vertices that have already been placed?

With the results of the Chapter 3 we can prove the following running time of our algorithm.

Theorem 4.1. *A single iteration of the Geometric Planarization Drawing approach with out computing and updating the vertex order takes* $O(n^3 \log n \log(1/\epsilon))$ *time.*

Proof. Let n_v be the size of the surrounding of the vertex v and let d_v be the degree of the vertex v. We have to compute the planarity region and offset this region at least once per placement operation. It takes $T_{\mathcal{PR}}(n_v, d_v) \in O(n_v d_v \log n_v)$ time to compute the planarity region of a vertex v; see Theorem 3.9. We can offset the planarity region in $T_{\text{off}}(n_v) \in O(n_v^2)$ time [FO98].

Algorithm 4.1: Geometric Planarization Drawing

Input: Graph $G = (V, E)$
Output: Drawing of G_p

```
1  G_p = planarization of G
2  Find planar straight-line drawing of G_p
3  while Drawing is not stable do
4      Choose an order O
5      forall Vertices v in an order O do
6          Compute the planarity region PR(v)
7          switch Type of v do
8              case Tail:  v = solution of MinMax-Tail
9              case Dummy:  v = solution of MinMax-Dummy
10             case Dummy and Tail:  v = solution of MinMax-Hybrid
11             case Independent:  v = Geometric or Random Center
12         Update Order O
```

Depending on the type of the vertex, the time to place a vertex is either $T_{\text{tail}}(n_v, d_v, \epsilon)$, $T_{\text{dummy}}(n_v, \epsilon)$, $T_{\text{hyb}}(n_v, d_v, \epsilon)$ or $T_{\text{ind}}(n_v, \epsilon)$. We use the same approximation factor ϵ for the angle and for the distance of the geometric center operation for independent vertices. Recall that the time for the geometric center operation depends on the diagonal distance of the bounding box of the drawing. We can assume that this distance is linear (or at least polynomial) in the number of vertices of the graph.

By the Theorems 3.11, 3.13, 3.14, 3.15 and 3.16 we get the following running time estimations.

$$
\begin{aligned}
T_{\text{tail}}(n_v, d_v, \epsilon) &\in O(\log(1/\epsilon)(n_v + d_v^2)\log(n_v + d_v)) \\
T_{\text{dummy}}(n_v, \epsilon) &\in O(\log(1/\epsilon) n_v) \\
T_{\text{hyb}}(n_v, \epsilon) &\in O(\log(1/\epsilon) n_v) \\
T_{\text{ind}}(n_v, \epsilon) &\in O(\log(n/\epsilon) n_v^2) \\
T_{\mathcal{PR}}(n_v, d_v) &\in O(n_v d_v \log n_v) \\
T_{\text{off}}(n_v) &\in O(n_v^2)
\end{aligned}
$$

The number of vertices n_v on the surrounding of v is linear in the number of vertices of the graph. Further, if we estimate the degree d_v of a vertex v with n_v, the time to place a tail vertex dominates all other placement operations. Thus, the total time to place vertex takes $O(n \cdot (T_{\mathcal{PR}}(n, n) + T_{offset}(n) + T_{tail}(n, n, \epsilon)) \subset O(n^3 \log n \log(1/\epsilon))$ time. □

For practical purposes, we can assume that ϵ is a constant resulting in a running time of $O(n^3 \log n)$.

In the following sections we explain the details of the missing parameters. In Section 4.1.1, we show how to select an appropriate initial drawing. We continue with several vertex orders in Section 4.1.2. Finally in Section 4.1.3, we introduce a heuristic to relax some conflicting constraints introduced by the cone construction.

4.1.1. Initial Drawing

Our Geometric Planarization Drawing approach improves an initial drawing of the planarization G_p. In general, we can use every planar straight-line drawing algorithm to compute the initial drawing of a planarization. If the crossing angles in the initial drawing are small, it is more likely that our approach leads to a straight-line representation of the planarization. We restrict ourselves to straight-line drawing algorithms implemented in the Open Graph Drawing Framework (OGDF) [CGJ+11]. The straight-line drawing has to preserve the planarity of the planarization. OGDF provides two algorithms that fulfill this requirements.

1. ogdf::PlanarStraightLayout
2. ogdf::TutteLayout

The ogdf::PlanarStraightLayout implements Kant's graph drawing algorithm using canonical orderings [Kan96]. The algorithm draws the graph on a grid of size at most $O(n \times n)$. The Tutte Layout [Tut63] fixes the external face on a convex polygon and places all other vertices in its barycenter. This layout of a planar graph has an exponential resolution. Unfortunately, this leads to problems with the floating points arithmetics of our implementation of the Geometric Planarization Drawing approach. Thus, our evaluation in Chapter 5 solely bases on the ogdf::PlanarStraightLayout algorithm.

4.1.2. Vertex Order

The Geometric Planarization Drawing approach moves the vertices in a certain order. The vertex order should somehow reflect the complexity of placing a vertex. If moving one vertex v helps another vertex u to improve its position, v should precede u in the vertex order. We can try describe the complexity in terms of the geometry of the graph or in terms of the combinatorial structure of the graph. For example, we can assume that moving a vertex with a large planarity region might clear the space for vertices on the surrounding. Thus, we can use the area of the planarity region to order the vertices. Furthermore, it might be useful to repair a planarization path at once, i.e., at the time we encounter a planarization path we successively place all vertices on this path. In this section, we propose several vertex orders. We compare them against each other in our final evaluation in Chapter 5.

Left-Right

The left-right order of the vertices is the ordering with respect to lexicographical order of coordinates of the vertices. If we apply our algorithm with this order on the drawing sketched in Figure 4.2(a), we can place all dummy vertices at their optimal positions in one iteration. If we mirror the graph horizontally, we need four iterations to place the vertices optimally. In the first iteration we place only the first dummy vertex at its optimal position. Then the algorithm has enough space to place the second vertex and so on. In this fashion, we can define a family of drawings so that our algorithm with this vertex orders requires a linear number iterations. Computing the left-right order takes $O(n \log n)$ time.

Shelling

The idea of the outer-shelling order is to iteratively remove the external face from the graph G. Let S_0 be the vertices on the external face of the graph $G = G_0 = (V_0, E_0)$ with the embedding $\mathcal{E}$. We define $G_i = (V_i, E_i)$ with $i > 0$ as the subgraph of G_{i-1} induced by the vertices $V_i = V_{i-1} \setminus S_{i-1}$, where S_{i-1} is the external face of graph G_{i-1}; see Figure 4.1. Let the vertex u be in the shell S_i and let vertex v be in the shell S_j. The vertex u precedes the vertex v in the outer-shelling order if $i < j$. We do not define

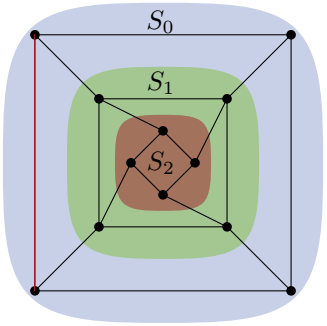


Figure 4.1.: The shell S_0 are all vertices in blue but not in the green region. The shell S_2 are all vertices in the red region.

an explicit order for vertices in the same shell. We use a breadth-first search starting simultaneously from all vertices on the external face to compute the OUTER-SHELLING order. Thus, we can compute the OUTER-SHELLING order in linear time. We refer to the inverse OUTER-SHELLING order as the INNER-SHELLING order. If we alternate in each iteration between the OUTER and the INNER-SHELLING order, we call the respective order ALTERNATING-SHELLING order. Computing the ALTERNATING-SHELLING order requires linear time in the number of vertices.

Our algorithm with an OUTER-SHELLING order requires fewer iterations than the LEFT-RIGHT order if we apply it on the previous family of drawings. We can modify the drawing in Figure 4.2(a) in such a way, that the vertices 1 and 4 are in the same shell and the vertices 2 and 3 are in the same shell. If we apply our algorithm with the OUTER-SHELLING order to this drawing, the algorithm can place the vertices 1 and 2 within the first iteration. In order to place the vertices 3 and 4 the algorithm requires two additional iterations. Mirroring the drawing at the y-axis does not significantly change the behaviour of the algorithm. On this family of drawings, the algorithm with the OUTER-SHELLING order requires about $n_d/2$ iterations to place all dummy vertices optimally, where n_d is the number of dummy vertices in the drawing.

If we merge the drawing of Figure 4.2(a) with its mirrored image as depicted in Figure 4.2(b), the algorithm with the OUTER-SHELLING order leads to a linear number of iterations for this family of drawings. We can start in the interior of the graph, i.e., with the INNER-SHELLING order, to move all vertices optimally within one iteration. Unfortunately, the OUTER-SHELLING order as well as the INNER-SHELLING order requires a linear number of iterations in case of the family of drawings sketched in Figure 4.2(c). If we apply our algorithm with the ALTERNATING-SHELLING order, we can place all vertices optimally in two iterations. The first iteration places all interior vertices optimally, then, in the second iteration, all vertices near the external face find their optimal position (or vice versa).

Area

The SHELLING orders neglects the geometry of the drawing completely. As an alternative, we define the order AREA which solely works on the geometry of the drawing. Vertices with a large planarity region are likely to find a good position with respect to the crossing angle. If this position is even near the center of the planarity region, there is a good chance that this divides the available space evenly for all vertices on the surrounding of this vertex. Thus, a vertex u precedes a vertex v in the AREA order if the area of the planarity region of u is larger than the area of the planarity region of v. Moving a vertex changes the planarity region of its neighbors. Consequently, the order of the vertices changes during one iteration. Initially, we compute the area of all planarity regions. We select the vertex with the currently largest planarity region and after placing the vertex, we update the areas for

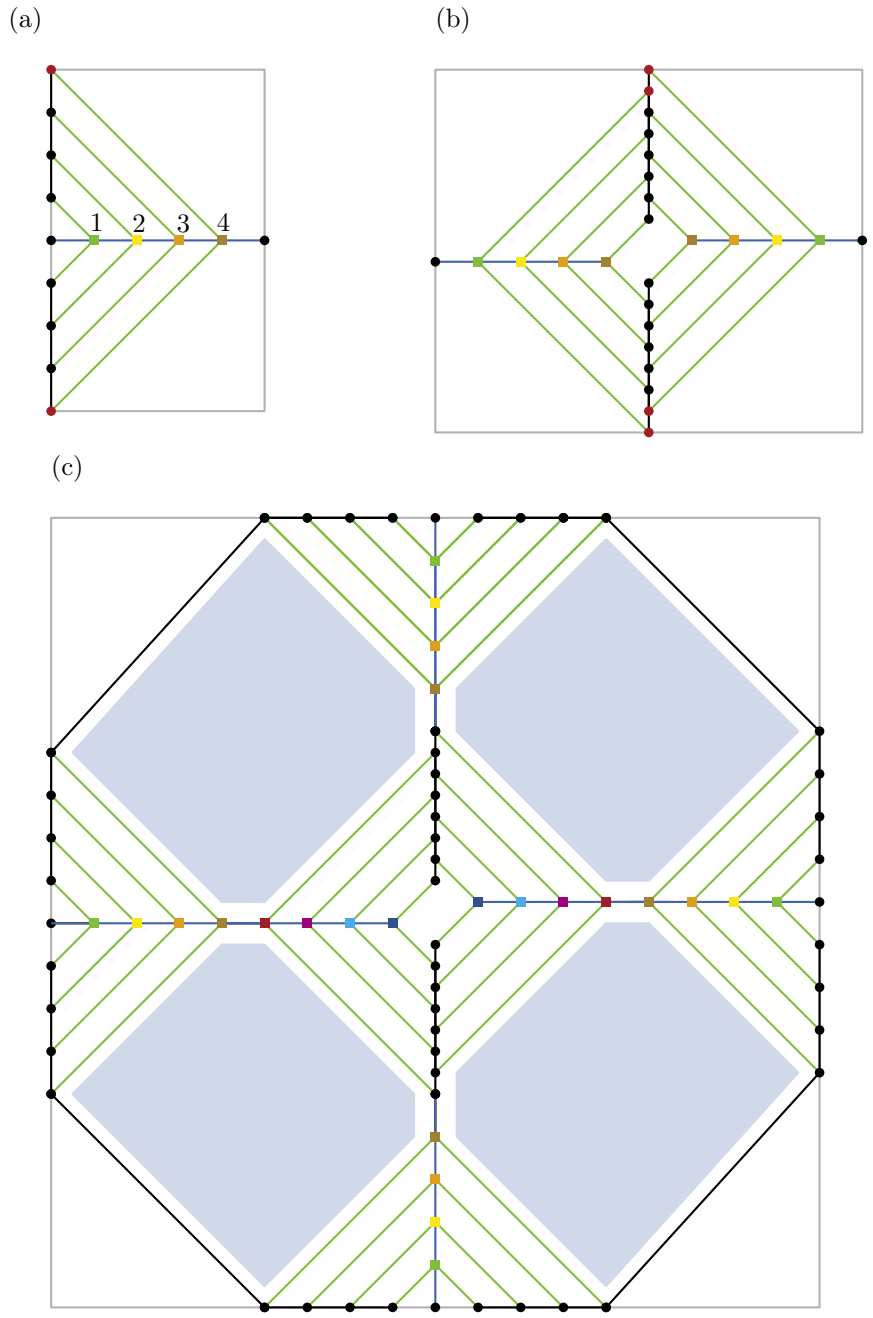


Figure 4.2.: Square vertices depict dummy vertices. The color of the dummy vertices depicts the different shells of the vertices, i.e., green vertices are supposed to be in the same shell. Assume that the red independent vertices have only little space to move, i.e., another gadget restricts the movement of the red vertices. Incident edges of the same color are one dissected pair. Assume that the light blue area is a dense graph that restricts the movement of the surrounding vertices. The vertex order affect the number of iterations until our Geometric Planarization Drawing approach finds a stable layout.

all the neighbors of the vertex. In order to keep track of the values, we can use a Priority Queue, e.g. a Binary Heap [CLRS09]. We can insert, update and remove an element from a Binary Heap in $O(\log n)$ time, where n is the number of vertices in the graph. We update the planarity region of a vertex v at most $\deg(v)$ times. Thus, in total there are $O(m) \subseteq O(n)$ additional planarity region computations in one iteration. Accordingly, computing and updating the order adds only a constant factor to the total running time of our Geometric Planarization Drawing approach. Recall that the computation of the planarity region and offsetting the planarity region dominates the total running time. One computation of the planarity region requires $O(n \deg(v) \log n)$ time. Thus, computing and updating the AREA order takes $O(n^3 \log n)$ time in total. Thus, maintaining the AREA order only add constant factor the running time of our Geometric Planarization Drawing approach; compare Theorem 4.1.

Repairing Planarization Paths

The vertex orders introduced in the previous section do not utilize any information about the planarization. They neglect whether or not an edge belongs to a planarization path. It might be useful to repair one planarization path after another, i.e., we successively handle all vertices on a planarization path. Consequently, in this order, we move a dummy vertex twice. In order to define an order on the planarization paths, we take a little detour over a vertex order O in the original graph G. This vertex order defines a lexicographical order O_E of the edges E and implicitly an order on the planarization paths. We build a sequence of vertices of the planarization, instead of a vertex order in that we place the vertices. Let $p_e = \langle v_1^e, v_2^e, \ldots, v_r^e \rangle$ be a planarization path of an edge e with $v_1 \prec v_r$ with respect to the order O. If an edge e does not have any intersections in G, we assume the planarization path to be the source u and target vertex v of e, i.e., $p_e = \langle u, v \rangle$. Our goal is to process all vertices of a planarization path p_e successively at the time we encounter the path p_e. In order to do so, we concatenate all planarization paths in the order of the corresponding edges.

Formally, a *concatenation* $p_a \cdot p_b$ of two paths $p_a = \langle u_2, u_2, \ldots u_r \rangle$ and $p_b = \langle w_1, w_2, \ldots, w_s \rangle$ with $r, s > 1$ is $p_a \cdot p_b = \langle u_1, u_2, \cdots u_r, w_1, w_2, \ldots w_s \rangle$. Let $e_1 \prec e_2 \prec \cdots \prec e_m$ be the lexicographical ordered edges with respect to the vertex order O. We define the vertex sequence S as the concatenation of all planarization paths, i.e., $S_O = p_{e_1} \cdot p_{e_2} \cdots p_{e_m}$. Each tail vertex v that is not a dummy vertex can occur only $\deg(v)$ times in this sequence. Each dummy vertex occurs only two times. Thus, the total length of this sequence is linear in the number of edges of the planarization. Let m be the number of edges in the original graph and m_p the number of edges in the planarization. Given a vertex order O, we can compute the REPAIR-PATH order in $O(m_p)$ time, since $m \leq m_p$. Thus, the running time is equal to $O(n)$ where n is the number of vertices in the planarization.

Random

Finally, we compare the introduced orders against a RANDOM order of the vertices. In each iteration we choose a new random permutation of all vertices. In this way, we expect that each vertex has the same chance to improve its crossing angle. Computing a permutation requires linear time in the length of the sequence, i.e., in the number of vertices.

4.1.3. Angle Relaxation

The cone construction introduced in Section 3.2.1 works independently of the vertex order. We can partition the set of vertices in a set of *settled* vertices that have already been placed and *unplaced* vertices. The unplaced vertices have the same influence on minimizing the crossing angle as the settled vertices. In contrast to settled vertices, the unplaced vertices

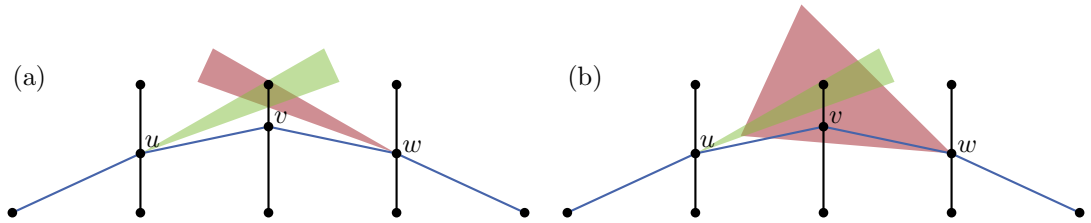


Figure 4.3.: Assume that the vertices u, v, w are processed in this order. If we place the vertex v, the vertex u already has a fixed position whereas w can adjust its position later on. (a) If every dummy neighbor of v has the same influence, the algorithms tends to keep the parabolic form of the blue planarization path. (b) Relaxing the angle of w can help to bend the planarization path in one common direction.

have another chance to improve their angle. Thus, we propose to weaken the influence of the unplaced vertices in the cone construction.

In this section, we relax the angle assigned to a cone depending on whether or not the corresponding vertex has already has been moved or not. Let v be the current vertex that we like to move and let $D(v)$ be its dummy neighbors. Each dummy neighbor $q \in D(v)$ has the same influence on the placement of v, i.e., we construct each cone $\mathcal{C}(q, \alpha)$, as defined in Section 3.2.1, with the same angle α. Figure 4.3 depicts an example where an unique angle for all vertices leads to conflicting constraints, i.e., the cone of the dummy neighbors forces the vertex v to move upwards. The minimal crossing angle for the vertex itself would force a movement downwards. As a result, the vertex stays somewhere in between. Thus, the planarization path keeps its parabolic form. If we increase the size of the red cone, as depicted in Figure 4.3(b), the vertex v has more space to optimize its own crossing angle and we force the planarization path in a direction preset by the edge $\{u, v\}$.

We accomplish this by assigning an independent angle α_q to each dummy neighbor q of v. Depending on whether or not a vertex already been moved, we relax the angle of the corresponding cone. We partition the dummy neighbors $D(v)$ in two sets $D_{\prec}(v) = \{q \in D(v) \mid q \prec v\}$ and $D_{\succ}(v) = \{q \in D(v) \mid q \succ v\}$. We like to relax the angle of the vertices in the set $D_{\succ}(v)$. Let α be the angle of the current step of the binary search. We assign α to each vertex $q \in D_{\prec}(v)$. We refer to $\delta \in [0, 1)$ as the *angle relaxation weight*. For every vertex q in $D_{\succ}(v)$ we interpolate the angle α_q with this weight between α and π. Thus, $\alpha_q = (1 - \delta) \cdot \alpha + \delta \cdot \pi$ for each dummy neighbor $q \in D_{\succ}(v)$.

4.2. A Force Directed Approach

Force-directed graph drawing is a fast and a simple method to draw graphs. Consequently, it is an obvious choice as a benchmark for our Geometric Planarization Drawing algorithm. A typical force-directed approach is the spring-embedder [Ead84, FR91]. The idea of the spring-embedder is to assign an initial layout to the graph and replace all edges with springs and vertices with steel-rings with repulsive forces. The final layout is a system with a minimal total energy.

In practice, we compute an attractive force $F^a(u, v)$ between adjacent vertices, a repulsive force $F^r(u, v)$ between every pair of vertices and a repulsive force $F^e(v, (a, b))$ between every vertex v and every edge $\{a, b\}$ that is not incident to v. We accumulate the forces to a vector $F^i(v)$ as follows.

$$F^i(v) = \sum_{(u,v)\in E} F^a(u, v) + \sum_{u\in V} F^r(u, v) + \sum_{(a,b)\in E} F^e(v, (a, b)) - \sum_{u\in V,(v,w)\ inE} F^e(u, (v, w))$$

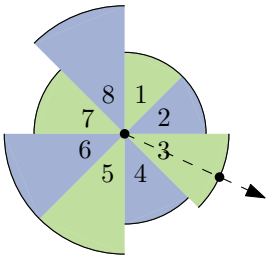


Figure 4.4.: The radius $R_i(v)$ of the radial arcs Z_i restrict the movement of a vertex v.

We apply the forces $F^i(v)$ to all vertices simultaneously. We repeat this process until we find a stable state of the system. One drawback of this version of the spring-embedder is, that it does not necessarily preserve the embedding of the initial layout. For us, this is a crucial requirement to draw a planarization. Bertault [Ber99] introduced a modification to this spring-embedder called *PrEd*. This force-directed approach is able to preserve the embedding of the initial drawing of a planar graph. PrEd accomplishes this by defining eight radial zones $Z_1, Z_2, \ldots, Z_8$. The zones Z_i are equal sized partitions of the unit disk. We can match each force to exactly one radial zone Z_i. For each vertex v, we compute a radius $R_i(v)$ for each radial arc Z_i. The radius $R_i(v)$ describes maximal movement of the vertex to preserve the planarity for all direction captured by the arc Z_i; compare Figure 4.4. The values $R_i(v)$ are derived from the minimal vertex edge distance. For further details see [SAAB11]. Bertault uses the same forces as Fruchterman and Reingold for his spring-embedder PrEd.

$$
\begin{aligned}
F^a(u,v) &= \frac{\operatorname{dist}(u,v)}{\delta}(v-u) \\
F^r(u,v) &= \frac{-\delta^2}{\operatorname{dist}(u,v)^2}(v-u) \\
F^e(v,(a,b)) &= \begin{cases} -\frac{(4\cdot\delta-\operatorname{dist}(v,i_v))^2}{\operatorname{dist}(v,i,v)}(i_v-v) & \text{if } i_v \in (a,b), \operatorname{dist}(i_v,v) < 4\cdot\delta, v \neq a, v \neq b \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

We compute the forces $F^i(v)$ for each vertex v. We match each force to a radial zone Z_j and restrict the magnitude of the force with the radius $R_j(v)$. The factor δ describes the optimal vertex–edge distance. Like Fruchterman and Reingold we use the value $\sqrt{A/n}$ for δ, where A is the area of the bounding box of the graph.

4.2.1. PrEd for Planarization Drawings

PrEd itself only preserves the planarity of the initial drawing but it does not optimize the crossing angles. In order to optimize the planarization, we introduce two new forces. Depending on the type of the vertex we apply either a dummy force F^d, a force F^t for the tail vertex, or, as already introduced, a force F^i for the independent vertex.

Let v be a dummy vertex and let (u, v, w) be a dissected pair incident to v. Our goal is to place v collinearly with u and w. We accomplish this by moving v in the direction of the bisector $\operatorname{bisect}(u,v,w)$ of the vectors $u-v, w-v$, i.e., $\operatorname{bisect}(u,v,w) = ((u-v)/\operatorname{dist}(u,v) + (w-v)/\operatorname{dist}(w,v))/2$. Let the point $\operatorname{colin}(u,v,w)$ be the intersection of the line $\mathcal{L}(v, \operatorname{bisect}(u,v,w))$ with the segment $\mathcal{S}(u,w)$. In order to move the vertex towards the intersection $\operatorname{colin}(u,v,w)$, we use the following dummy force F^d with an attenuation factor $\lambda > 0$.

$$F^d(v,(u,w)) = \frac{\text{dist}(v, \text{colin}(u,v,w))}{\lambda} \text{bisect}(u,v,w)$$

Suppose that v is a tail vertex and let (u,w,v) be a dissected pair incident to v. Ideally, we can place the vertex v on the extension of the segment $\mathcal{S}(u,w)$; compare Figure 4.5(b). A radial movement of the vertex v around w accomplishes this. Over several iteration of the spring-embedder we can imitate this movement by translating the vertex v tangential to this radial arc. Thus, let $\text{orth}(u,w,v)$ be the vector orthogonal to the vector $w-v$ so that $\text{orth}(u,w,v)$ moves towards the extension of the segment $\mathcal{S}(u,w)$, i.e., $\text{orth}(u,w,v)$ is oriented to the opposite site of the bisector $\text{bisect}(u,w,v)$, relative to the vector $v-w$. We require $\text{orth}(u,w,v)$ to be normalized. Let v_1, v_2 and w be three different collinear points. The length $\text{dist}(v_1,w)$ and $\text{dist}(v_2,w)$ should not affect the angle by which we move v_1 or v_2, i.e., the force should improve the crossing angle independently from the distance of w to v_1 and to v_2. Assume that the $\text{dist}(v_1,w) = 1$. If we translate v_1 by the vector $\text{orth}(u,w,v)/\kappa$, we have to translate v_2 by the vector $\text{dist}(v_2,w) \cdot \text{orth}(u,w,v)/\kappa$, with $\kappa > 0$. On the opposite tail vertex u of v of the same dissected pair, the computed force points in the opposite direction. Thus, we scale the force about a factor $1/2$ to avoid an unstable behaviour. Overall this results in the following force F^t.

$$F^t(v,(u,w)) = \frac{\text{dist}(w,v)}{2 \cdot \kappa} \cdot \text{orth}(u,v,w)$$

Let $DP(v)$ be the set of all dissected pairs incident to v. The force on the tail vertex v is the sum of forces $F^t(v,(u,w))$, i.e., $F^t(v) = \sum_{((u,w,v))\in DP(v)} F^t(v,(u,w))$. If v is a dummy vertex, we accumulate F^d for all dissected pairs. Since v might be a tail vertex, we add the tail force $F^t(v)$ to the respective force, i.e., $F^d(v) = F^t(v) + \sum_{((u,v,w)\in DP(v)} F^d(v,(u,w))$.

For a fair comparison with the Geometric Planarization Drawing approach, we restrict the spring-embedder to a scaled bounding box of the drawing. For vertices on the external face we apply an additional force to repel the vertices from the bounding box. In order to ensure that no vertex crosses the bounding box, we restrict the zones of an external vertex to the minimal distance to the bounding box.

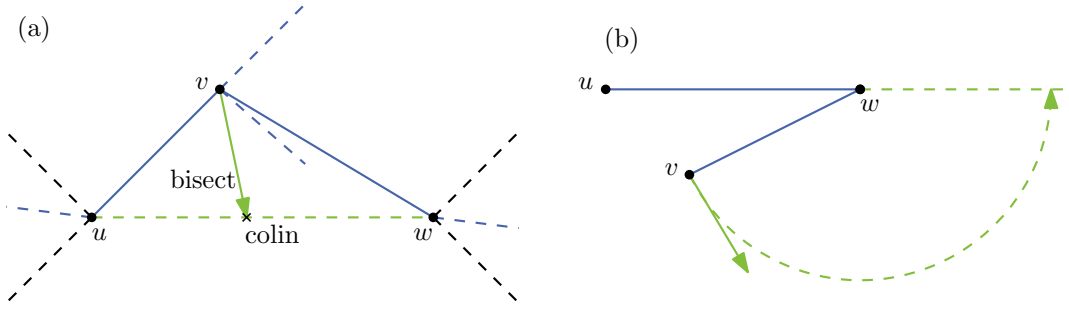


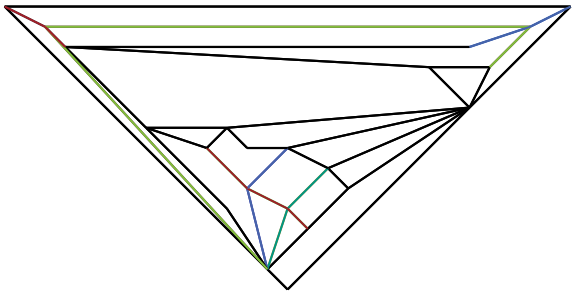
Figure 4.5.: Sketch of our local force. (a) If v is a dummy vertex, move it a long the bisector of the adjacent segments. (b) The vertex v is a tail vertex of a planarization path. We move v gradually along an arc by translating v in direction of the perpendicular vector.

5. Evaluation

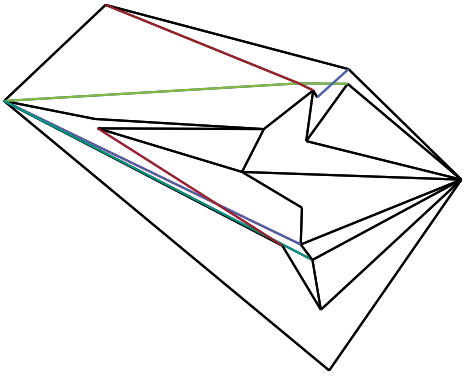
Drawing a planarization is a difficult task, it is as hard as the Existential Theory of the Reals; see Section 1.1. Thus, there is only a small chance of finding an algorithm that can solve this problem efficiently. In case of our Geometric Planarization drawing approach, we iteratively place a vertex to locally optimal position. We cannot guarantee any specific bounds on the global quality of our drawings. Thus, we take an experimental approach in order to evaluate our Geometric Planarization Drawing algorithm. We use two metrics to measure the quality of a drawing. First, we use the crossing angle of each dissected pair as a local property. Secondly, we measure the *stretch* of a planarization path to reflect the global qualities of a drawing. The stretch is ratio of the length of the planarization path to the length of an ideal path.

There are several parameters for our Geometric Planarization Drawing approach. For example, we introduced different orders in which we place the vertices or a relaxation for the angles of the cone construction. For some of the combinations of the parameters it is difficult to predict how well they perform, or which of them achieves the best quality. Figure 5.1 depicts the final results of different version of our Geometric Graph Drawing approach applied to one Rome graph; compare the configurations with Table 5.2. Figure 5.1a shows the initial drawing of the graph. All of the drawings are almost optimal, at least it is hard to perceive a deviation from the ideal paths. The GPD-PRR-8 in Figure 5.1f, i.e., our Geometric Graph Drawing approach with the Path-Repair vertex order and a relaxation for the cone construction, yields a drawing very different from the other ones. Note that the right edges of the lower corner visually collapse into one edge. Beside the quality of the drawings, there is a difference in the number of iterations until the algorithm finds the final drawing. The GPD-Path-Repair configuration requires only 3 iterations to find the optimal drawing of the graph, whereas the GPD-Relax-1 configuration requires 26 iterations.

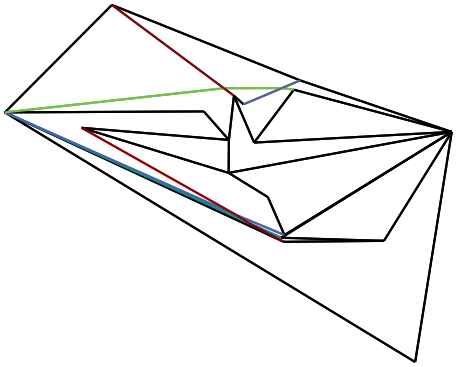
In this chapter, we evaluate the quality and the running time of different configurations of our Geometric Planarization Drawing approach. First of all, we give a brief description of our experimental setup and a characterization of the Rome graphs in Section 5.1. In Section 5.2, we show that our algorithms improve the initial drawing. Further, we compare the quality of the different configurations of our Geometric Planarization Drawing approach and our PrEd implementation. Finally, in Section 5.3, we evaluate the running time of our algorithm. We divide this into two parts, the time per iteration and the number of iterations required by a configuration of our approach.



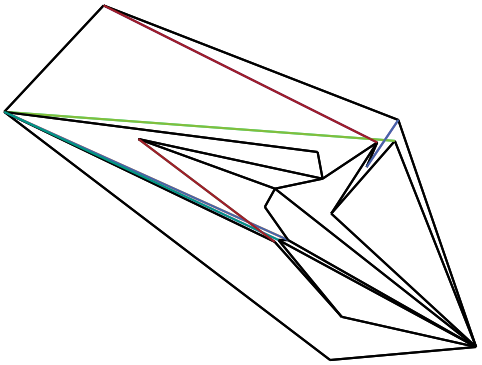
(a) Initial Drawing of graph *grafo2920* of the rome graph benchmark.



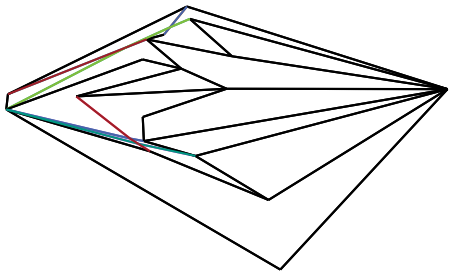
(b) GPD-Base: Final drawing after 14 iterations.



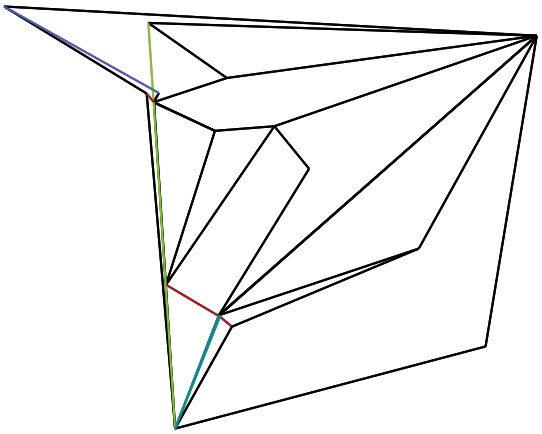
(c) GPD-Area: Final drawing after 5 iterations.



(d) GPD-Path-Repair: Final drawing after 3 iterations.



(e) GPD-Relax-1: Final drawing after 26 iterations.



(f) GPD-PRR-8: Final drawing after 17 iterations

Figure 5.1.: Different configurations of our Geometric Graph Drawing approach applied to graph grafo2920 of the rome graph benchmark.

5.1. Experimental Setup

In this section, we describe our experimental setup in terms of the used hardware and the used software libraries. All experiments were conducted on a server with two Intel(R) Xeon(R) E5430 CPUs with 32GB of RAM. Each CPU has four cores clocked at 2.66GHz. The program was compiled with g++ version 4.8.3 on OpenSuse 13.2 with the flags `-frounding-math -std=c++11 -O3 -DNDEBUG -march=native -fopenmp`. We use OGDF [CGJ+11] version 2012.07 to planarize our graphs and to compute the initial drawing. For all geometric operation, we use CGAL [CGA15] version 4.5.2. Stefan Huber supplied us with his STALGO library for computing the straight skeleton of a polygon and offsetting polygons [HH11, HH12]. We created our plots with R [R C15] in combination with PLYR [Wic11] and GGPLOT2 [Wic09].

5.1.1. Configurations

In this Section, we describe different configurations of our Geometric Planarization Drawing approach. Some of the parameters are the same for every configuration like the initial drawing. We have chosen the fixed parameters as follows.

1. *Initial Drawing*: The only practical algorithm for planar straight-line drawings implemented in OGDF is the PLANARSTRAIGHTLAYOUT (PSL). Thus, we are not able to analyze the influence of the initial drawing to our Geometric Planarization Drawing approach.
2. *Bounding Box*: We limit the area with the bounding box of the initial drawing increased by factor of two, i.e., $\beta = 2$ in Section 3.1.3.
3. *Offsetting*: We offset each planarity region by at most $\gamma = 0.1$; see Section 3.1.4.
4. *Approximation*: We use $\epsilon = 10^{-3}$ as an approximation guarantee in Section 3.2.1, 3.2.2, 3.2.3 and Section 3.2.4.

Further, there are several free parameters, for example the order in which we place the vertices. Table 5.2 lists all evaluated configurations with the following parameters.

1. *Vertex order*: The vertex orders described in Section 4.1.2 control the order in which we place the vertices in one iteration. The choices are the ALTERNATING-SHELLING order, the PATH-REPAIR order with a breath-first-search starting at the leftmost vertex as an underlying vertex order, or a RANDOM order.
2. *Angle relaxation weight*: We select an angle relaxation weight δ from the set $\{0, 0.1, 0.2, 0.4, 0.6, 0.8\}$; see Section 4.1.3.
3. *Independent center*: We have two possibilities to place an independent vertex either in the GEOMETRIC center by shrinking the planarity region or by selecting a RANDOM point in the planarity region; Section 3.2.4.

Floating Point Arithmetic

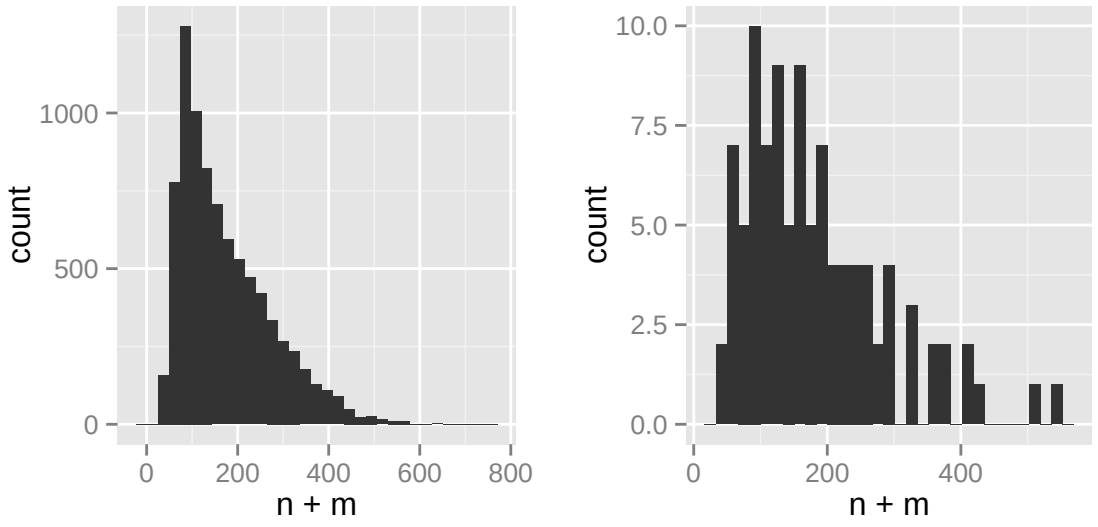
A not so obvious parameter of our implementation is the representation of the floating point numbers. The representation and the handling of the numbers affect the total running time of our implementation. In general, double precision floating points are not enough to handle complex geometric problems in practice. Recall that a drawing of a 1-planar graph requires an exponential area [HELP12]. Therefore, we need a number of bits that is linear in the number of vertices to represent the coordinates in a drawing.

We represent our coordinates as quotient of arbitrary large integers. We use the Gmpq data type [Hem15] implemented in the GNU Multiple Precision Arithmetic Library[1]. The Gmpq number would increase with each iteration of our Geometric Graph Drawing approach.

[1] Available at `https://gmplib.org/`

Table 5.2.: Configurations for our Geometric Graph Drawing approach.

Name	**Initial Drawing**	**Vertex Order**	**Angle Relaxation Weight**	**Independent Center**
GPD-Random-Center	PSL	Alternating-Shell	0	Random
GPD-Base	PSL	Alternating-Shell	0	Geometric
GPD-Random-Order	PSL	Random	0	Geometric
GPD-Path-Repair	PSL	Path-Repair	0	Geometric
GPD-Relax-1	PSL	Alternating-Shell	0.1	Geometric
GPD-Relax-2	PSL	Alternating-Shell	0.2	Geometric
GPD-Relax-4	PSL	Alternating-Shell	0.4	Geometric
GPD-Relax-6	PSL	Alternating-Shell	0.6	Geometric
GPD-Relax-8	PSL	Alternating-Shell	0.8	Geometric
GPD-PRR-1	PSL	Path-Repair	0.1	Geometric
GPD-PRR-2	PSL	Path-Repair	0.2	Geometric
GPD-PRR-4	PSL	Path-Repair	0.4	Geometric
GPD-PRR-6	PSL	Path-Repair	0.6	Geometric
GPD-PRR-8	PSL	Path-Repair	0.8	Geometric



(a) All non-planar Rome graphs

(b) Rome-100

Figure 5.3.: Distribution of number of vertices

Due to the size of the numbers, the running time of our algorithm becomes impractical. As a compromise we use the Gmpq representation for all operations within a single iteration. After each iteration we round the numbers to double precision floating points. As a consequence, two vertices can collapse into a single point. We stop the algorithm at this point and use the previous drawing for our evaluation. For a few drawings the algorithm stops early due to problems with the floating point arithmetics in the offsetting procedure.

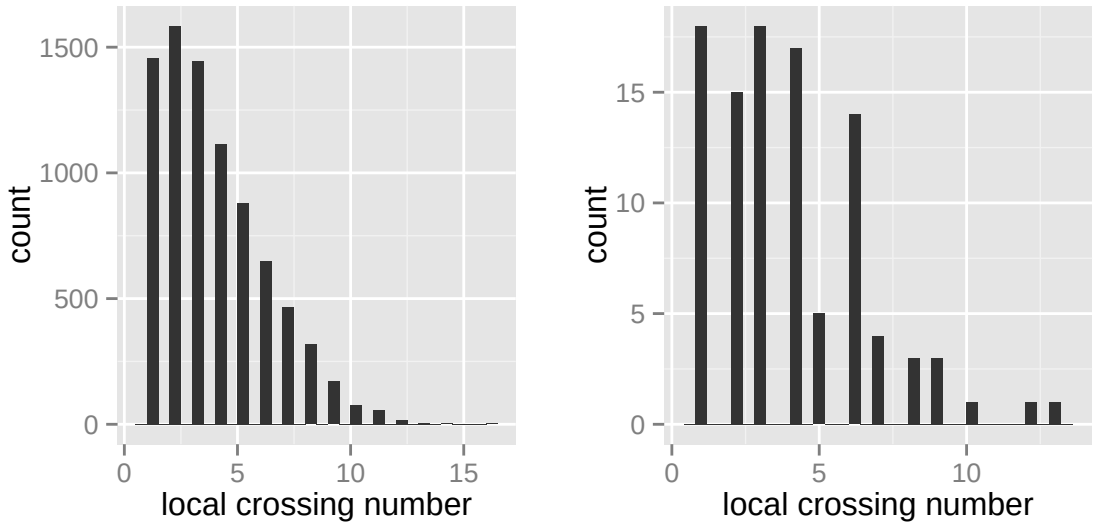
5.1.2. Rome Graphs

In order to evaluate our Geometric Graph Drawing approach we use the Rome graphs as a benchmark[2]. The dataset consists of 11'534 graphs with 8'253 non-planar graphs. We use the non-planar Rome graphs for our analyzes. Our experiments are evaluated on 100 of these graphs, chosen uniformly at random; we refer to these graphs as *Rome-100*. A list of these can be found in the Appendix A. The graphs are not necessarily biconnected. Thus, we compute all maximal biconnected components. We apply the algorithms to each non-planar component independently. Every graph in our testset has exactly one non-planar biconnected component.

We use the optimal edge insertion algorithm introduced by Gutwenger et al. [GMW05]. The algorithm extracts a maximal planar subgraph from a (non-planar) graph G. It iteratively reinserts the *non-planar* edges not contained in the planar subgraph. The insertions minimizes the total number of crossings on this edges. In the following we analyze several characteristics of the planarizations of the Rome-100 graphs. We show that on real-world graphs the local crossing number and several other measures are correlated. Thus, we can use the local crossing number as an explanatory variable.

Figure 5.3 shows the distribution of the size of the planarized graphs, where the size is the sum of the number of vertices and number of edges in the planarization. The graph at the median has a size of 147. The average size of the graphs is 174. The first quantile is at 93 and the third quantile at 233.

[2]Rome graphs are available at `http://www.graphdrawing.org/data/`



(a) All non-planar Rome graphs

(b) Rome-100

Figure 5.4.: Distribution of number of k-planar Rome graphs.

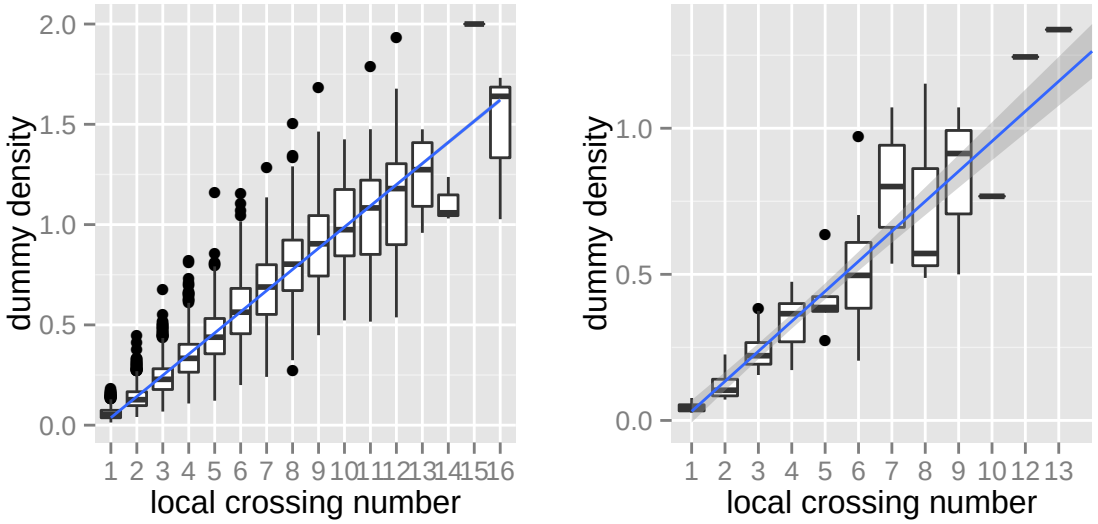
Figure 5.4 shows a similar distribution for the local crossing number graphs. The median local crossing number is 3. On average the local crossing number is 3.74. The first quantile is at 2 and the third quantile at 5. All Rome graphs, not just the Rome-100 graphs, are k-planar, with $k \leq 16$.

We define the *dummy density* as the ratio of the number of dummy vertices to the number of vertices in the original graph. A planarization with a dummy density of 0.5 has one dummy vertex for two vertices in the original graph. Figure 5.4 suggests that our planarizations of the Rome graphs have a linear association between the dummy density and the local crossing number.

If the optimal edge insertion algorithm has to reinsert a large number of non-planar edges, the likelihood of a pairwise intersection between non-planar edges increases. Thus, the number of dummy vertices increases as well. A linear regression confirms this expectations; compare Figure 5.5. The two variables are positively correlated with an r^2 value of 0.82. A value near 1.0 suggests a strong correlation between the two values.

We can observe a similar behavior for the size of the maximal *connected dummy component*. A connected dummy component of a planarization is a maximal connected subgraph in the planarization such that every vertex is a dummy vertex. Figure 5.6 shows the relationship between the local crossing number and the size of the largest connected dummy component of a graph. The plot suggests a correlation between the two variables. If we assume a quadratic association between the two variables, we get a coefficient of determination $r^2 = 0.79$.

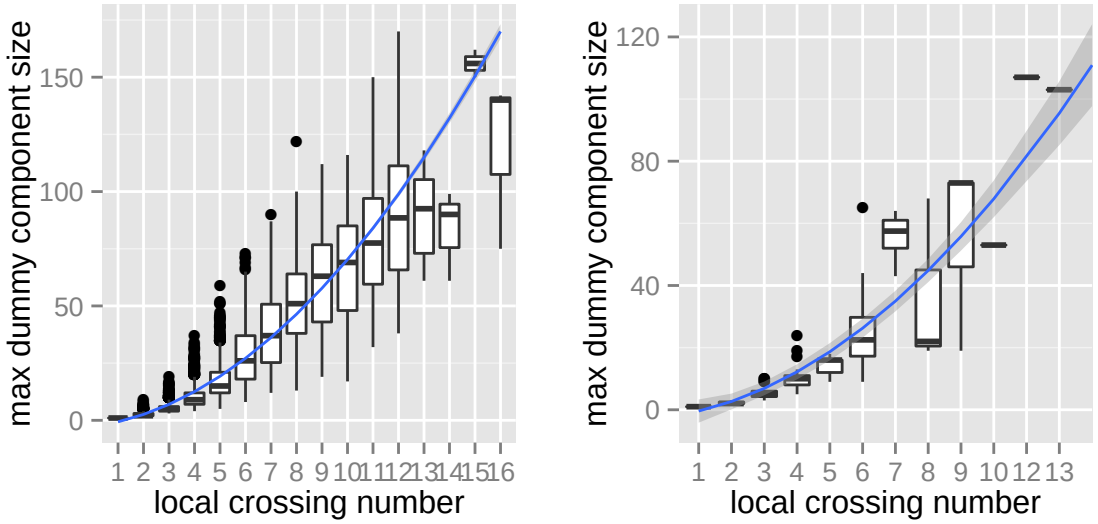
From here on, we assume that there is a correlation on real-world graphs between the local crossing number and the dummy density, and a correlation between the local crossing number and the size of the largest connected dummy component. Therefore, we use solely the local crossing number for the further analyzes of the quality and the running time of our Geometric Planarization Drawing approach. Note that in our experiments we categorize a drawing as k-planar if the longest planarization path in the drawing has exactly k dummy vertices.



(a) All non-planar Rome graphs

(b) Rome-100

Figure 5.5.: Density of the dummy vertices with respect to the local crossing number of the graph.



(a) All non-planar Rome graphs

(b) Rome-100

Figure 5.6.: Maximum number of dummy vertices in one dummy component with respect to the local crossing number.

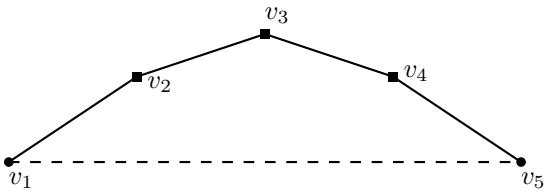


Figure 5.7.: Every dissected pair has a small crossing angle. Nevertheless, the path deviates clearly from the ideal path.

5.2. Quality

In this section, we analyze the quality of the Geometric Planarization Drawing approach. Using statistical tests, we confirm the following three hypothesis: 1) Our algorithm significantly improves the initial drawing. 2) The drawings have better crossing angles then our PrEd implementation. 3) Depending on the local crossing number, it is useful to select different configurations of our algorithm.

We use two metrics to quantify the quality of a planarization: the crossing angle and the stretch. A small maximum crossing angle of a planarization path can result in a drawing far from a straight-line segment; compare Figure 5.7. Thus, we use the stretch of a path as a second measure for the quality of a drawing. We define the stretch of a planarization path as the ratio of the length of the planarization path to the length of the ideal path (a straight-line segment from the source to the target vertex of the planarization path). Formally it is

$$\text{stretch}(p_e = \langle v_1, v_2, \ldots, v_r \rangle) = \frac{\sum_{i=2}^{r} \text{dist}(v_{i-1}, v_i)}{\text{dist}(v_1, v_n)}.$$

5.2.1. Experimental Design and Statistical Tests

A common way to compare two drawings or the running time of an algorithm is to compare different characterizations of the respective score, e.g. on average of the scores, or the different quantiles of the scores. In general, outliers can strongly influence the average. Therefore, the average is not a robust way to quantify the quality of a drawing. The median, the first and third quantile are only useful if the underlying distribution is known. Otherwise, depending on the population, the algorithm can focus on optimizing these three values. Consider a distribution with four clusters, one cluster just right above the minimum, the first and third quantile, and the median. This would suggest an improvement of the complete drawing but this improvement is not representative for the complete drawing. Instead, we take a statistical approach to evaluate the crossing angle and the stretch.

Our evaluation focuses on the comparison of two drawings of the same graph, e.g. the initial versus the final drawing of a graph or the final results of two different algorithms. Let $\mathcal{G} = \{G_1, G_2, \ldots, G_s\}$ be the test sample of graphs, i.e., in our case the 100 randomly selected Rome graphs and let $\mathcal{G}^1, \mathcal{G}^2$ denote the two sets of drawings of the test sample. It is difficult to assign a representative quality measure to each graph. Aggregating all angles would yield a loss of information. Therefore, we do not handle each graph as an individual but every dissected pair or every planarization path. For example, we compare whether or not the crossing angle cr-$\alpha_1(u, v, w)$ of the dissected pair (u, v, w) in the first drawing is smaller than the crossing angle cr-$\alpha_2(u, v, w)$ of the same dissected pair in the second drawing. Thus, we can treat the sets of both measurements as dependent. According to Sheskin [She03], this would allow the following three statistical tests.

1. The t-test for two depending samples

2. The Wilcoxon matched-pairs signed ranks test
3. The binomial sign test for two dependent samples

The t-test for two depending samples requires the underlying populations of the two samples to be normally distributed. Our aim is to find a drawings where the most dissected pairs have a crossing angle of 0 (or stretch of 1). Consequently, the expected distribution is not symmetric. Thus, our hypothesis cannot support an underlying normally distributed population. The Wilcoxon matched-pairs signed-ranks test is not applicable with the same argument, since the test requires the distribution to be symmetric around the median. The binomial sign test for two dependent sample bases on the following two assumption.

1. The sample has been randomly selected from its population.
2. The measurements can be rank-ordered.

The graphs are randomly selected, thus we consider the first assumption to be fulfilled. We can assign ranks to each measurement by sorting the measurements. Both functions, the crossing angle and the stretch, satisfy both assumption. Therefore, we use this test to quantify the improvement of one heuristic over the other.

Binomial Sign Test for two Dependent Samples

The description of binomial sign test for a single sample and two samples is based on the book "Handbook of Parametric and Nonparametric Statistical Procedures" by David J. Sheskin [She03].

$$\frac{r}{n} \leq \frac{1}{2} \tag{5.1}$$

$$P(\geq r) = \frac{1}{2^n} \sum_{k=r}^{n} \binom{n}{k} \leq \alpha \tag{5.2}$$

The binomial test for two dependent samples applies the binomial test for a single sample. We give a brief description of the binomial test for single samples. With a binomial test for a single sample we can compute how likely it is that in sequences of 1 and 0, the 1 occurs with a different probability then the 0. The directional Alternative Hypothesis states that the probability of the quantity of 1 in the sequence is greater than 0.5. The Null Hypothesis states the contrary. In order to reject the Null Hypothesis with a significance level of α, the Equations 5.1 and 5.2 have to hold, where r is the number of 0, n the total length of the sequence. The function $P(\geq r)$ is the probability that r or more 1 occur in the sequence. The Equation 5.2 is the binomial distribution with probability of 1/2.

We can compute the likelihood that the scores of two samples represent different populations with the binomial test for two dependent samples. Each score in the first sample has a matched score in the other sample. The test compares the scores in the first samples with the corresponding scores in the other sample, i.e., $s_{1,i} < s_{2,i}$. The comparison results in a sequence of 0 and 1 . We can apply the binomial test for one sample to evaluate the likelihood that the occurrence of a one is more likely than the occurrence of a 0. Thus, we have a likelihood that the two scores represent two different samples and the scores of the first sample is smaller than the scores of the second sample.

We can use this test to compare two drawings with each other. We can assign a score to each dissected pair or to each planarization path. The score of a dissected pair is the crossing angle and the stretch is the score for the planarization paths. Thus, we get a likelihood that the crossing angle or the stretch of one drawing is less then the corresponding value in the other drawing.

$$\text{cr-}\alpha_1(u,v,w) + \Delta < \text{cr-}\alpha_2(u,v,w) \tag{5.3}$$

$$\text{stretch}(p_e) \cdot \Lambda < \text{stretch}(p_e) \tag{5.4}$$

The test does not say anything about the distance between the two populations. We augment the binomial sign test for two samples with an *additive buffer* Δ and a *multiplicative buffer* $\Lambda \geq 1$. We check whether or not the two scores represent two different populations if we add (or multiply) the buffer to the first population. Equation 5.3 depicts the comparison for a dissected pair (u, v, w) and Equation 5.4 shows the comparison for a planarization path p_e. We can use a binary search to approximate the maximum buffer $\Delta^\star$ and $\Lambda^\star$ so that the two samples are likely to represent two different population. We say that one sample has as an *advantage* of $\Delta^\star$ ($\Lambda^\star$) over the other sample.

All our tests are conducted with a significance level of $\alpha = 0.05$.

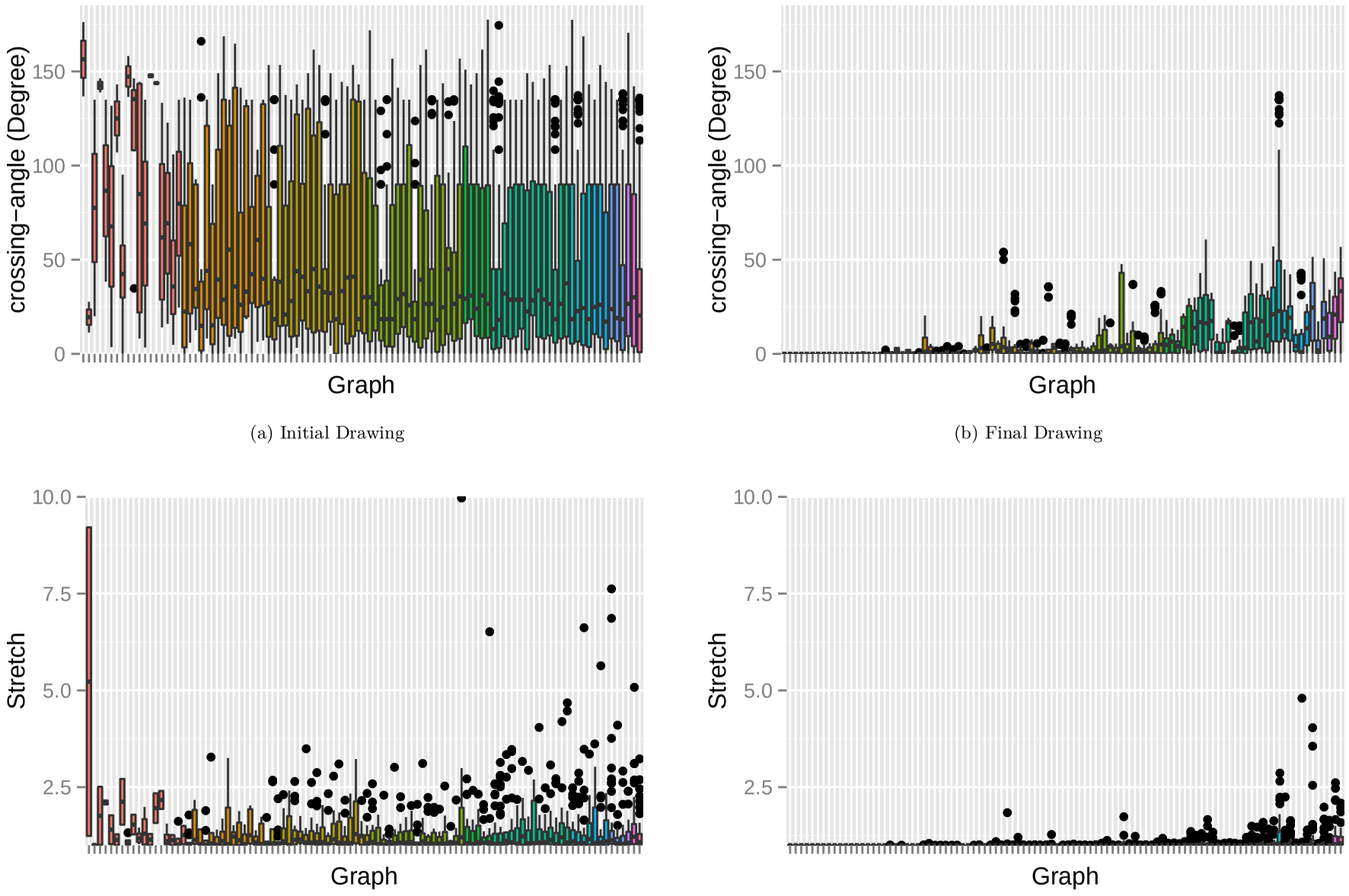
5.2.2. Initial Drawing versus Final Drawing

First of all, we examine whether or not the Geometric Planarization Drawing approach, improves the quality of the initial drawing of a graph. For this evaluation, we use the GPD-Base configuration of our algorithm, i.e., there is are no additional heuristics and we move the vertices in the order of the Alternating-Shelling order. The Figures 5.8a, 5.8b give a first impression that our algorithm yields a significant improvement of the crossing angles compared to the initial drawing. Both plots use a box plot with the set of all graphs on the x-axis ordered with respect of the local crossing number. We omit the names of graphs in this plot. The y-axis shows the crossing angle of each dissected pair. For visual aid, the colors of the boxes represent the local crossing number. The boxes of the box plot contain all values between the first and the third quantile. The bold line within the box represent the median of the values. The whiskers depict the largest (smallest) value smaller (greater) then 1.5 times the distance between the third and first quantile. Every other point depicts an outlier.

The average angle over all initial drawing is about 44°. The first quantile is at 6°, the median at 26° and the third quantile at 89°. In comparison, the average angle over all final drawings is 12°, the first quantile is 0.2°, the median at 6° and the third quantile at 20° degrees. All values show an improvement of the angle. Unfortunately, the first quantile, the median and the third quantile are not a good choice to show an improvement of the angle. The final drawing can only improve exactly these three angles. In this case, the final drawing would not be significantly better then the initial drawing. Our evaluation shows, that the third quantile of the final drawing is below the median of the first drawing. Thus, at least 25% of the angles are improved.

The binomial test for two dependent samples compares all values and translates the comparison into a probability measure. We think, that this value supplements the traditional characteristics. The binomial tests shows that the final drawing has an advantage of about $\Delta^\star = 16°$ to the initial drawing, i.e., we can add a buffer of 16° to the crossing angles of the final drawing and the populations are different at significance level of 95%.

The Figures 5.8c and 5.8d show the distribution of the stretch. It suggests an improvement of the stretch. The average stretch of the initial drawing is 1.3, the first quantile at 1.01 the median is at 1.08 and the third quantile is at 1.3. For the final drawing we have an average stretch of 1.04, the first quantile and the median is at 1 and the third quantile are 1.01. In this case, the median is (at least almost) optimal, thus the stretch of at 25% paths has been improved. The binomial sample test for stretch shows a relative advantage of $\Lambda^\star = 1.03$. Thus, we can add about 3% to the stretch of the final drawing and the populations are different at a significance level of 95%.



(a) Initial Drawing

(b) Final Drawing

(c) Initial Drawing

(d) Final Drawing

Figure 5.8.: Distribution of the crossing angle and the stretch per drawing.

5.2.3. Pairwise Comparison of GPD Configurations and PrEd

In this section, we compare the different configurations of our Geometric Planarization Drawing approach against each other and against our implementation of PrEd. In case of our Geometric Planarization Drawing implementations, we compare the final drawings. In case of PrEd, we performed 250 iterations per graph. We use the drawing with the minimal maximal crossing angle to compare the PrEd approach against the Geometric Planarization Drawings. We partition the set of graphs into classes according to its local crossing number.

Figure 5.9 shows matrices for the local crossing numbers 1 and 3 and Figure 5.10 the matrices for local crossing number 7 and 13. The x-axis and the y-axis show the name of the configuration; compare Table 5.2. The value and the color in each cell represent the buffer $\Delta^\star$ (rounded down to an integer) so that the configuration on the x-axis has an advantage over the configuration on the y-axis. If we cannot show an advantage, the respective cell is gray.

For example, the plot for 1-planar graphs shows, that every Geometric Planarization Drawing configuration has an advantage over the PrEd implementation, but the advantage is smaller than 1. Of course, PrEd cannot have an advantage over itself. Thus, the value of the pair (PrEd, PrEd) is gray. The advantage of the GPD configurations over PrEd increases with the local crossing number.

For 1-planar graphs the plot shows that the configuration GPD-Area and GPD-Relax-1 have an advantage of 17° over the configuration GPD-PRR-4. We would expect that both configurations have a similar advantage over GPD-PRR-6 and GPD-PRR-8. As the plot shows, this is not the case. If we place a tail vertex, we can place the vertex anywhere in the intersection of the cones with the planarity region. In our implementation, we place the vertex at the leftmost intersection point between the boundary of the cones and the planarity region. Figure 5.11 shows one example graph for both configurations. Both configurations place the vertex u before the vertex v. Unfortunately, the new position of u in the GPD-PRR-4 configuration is in the desired direction of v. Thus, after one iteration v blocks the visibility of u for this configuration. The GPD-PRR-4 configuration is not able to recover from this decision. In case of the GPD-PRR-6, the placement of u is not in the desired direction of v and thus, the new position of v does not block the visibility of v. After one more iteration, the algorithm is able to place all vertices in a good position. There are a few more graphs with a similar behavior, resulting in an advantage of GPD-Area and GPD-Random-Order over GPD-PRR-4 and not over GPD-PRR-6.

Figure 5.11c shows that the intersection of the cone with the planarity region can be large. For tail vertices with one dummy neighbor the relaxation is not very useful. We expect the heuristic to perform better on graphs with a high local crossing number (and thus, having longer planarization paths). With an increasing local crossing number, the chance of conflicting constraints increases. Our plots in Figure 5.9 and Figure 5.10 support this hypothesis. For a small local crossing number the angle relaxation does not have significant advantage over any other configuration. However, with an increasing local crossing number the importance of the angle relaxation increases.

Among the configurations without the angle relaxation there is no vertex order that has an advantage over another vertex order. Especially, there is no vertex order that has an advantage over the configuration GPD-Random-Order. Nevertheless, the values for the buffer of the GPD-Area configuration and the GPD-Relax-1 configuration dominate every other column, i.e., the value in the column is greater than or equal to every other value in the same row. For drawings with a local crossing number of 3, the GPD-Path-Repair order dominates every other configuration. There is no configuration that dominates

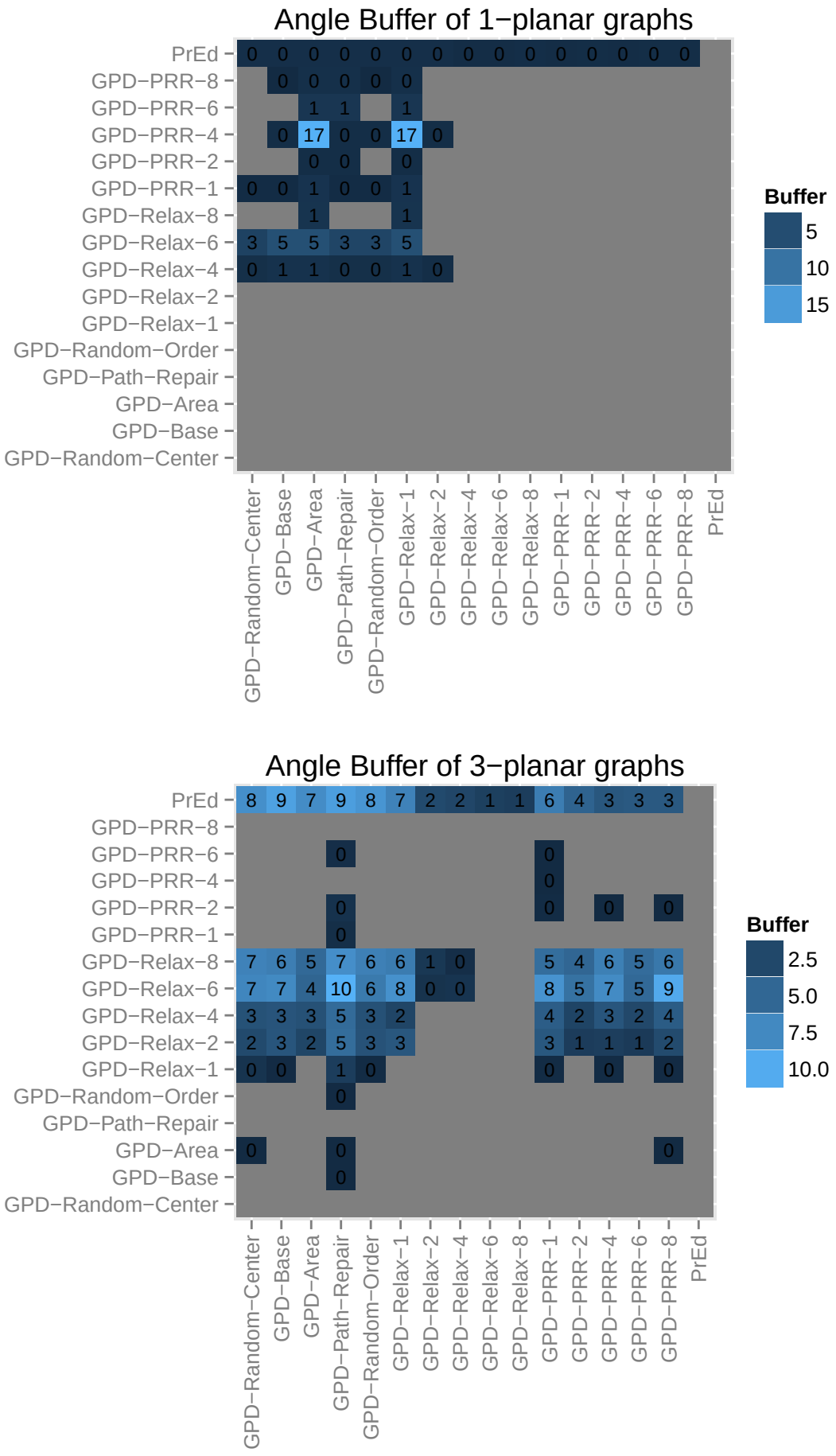


Figure 5.9.: Pairwise comparison of Geometric Planarization Drawing configurations and PrEd. Each cell represents the advantage in degree.

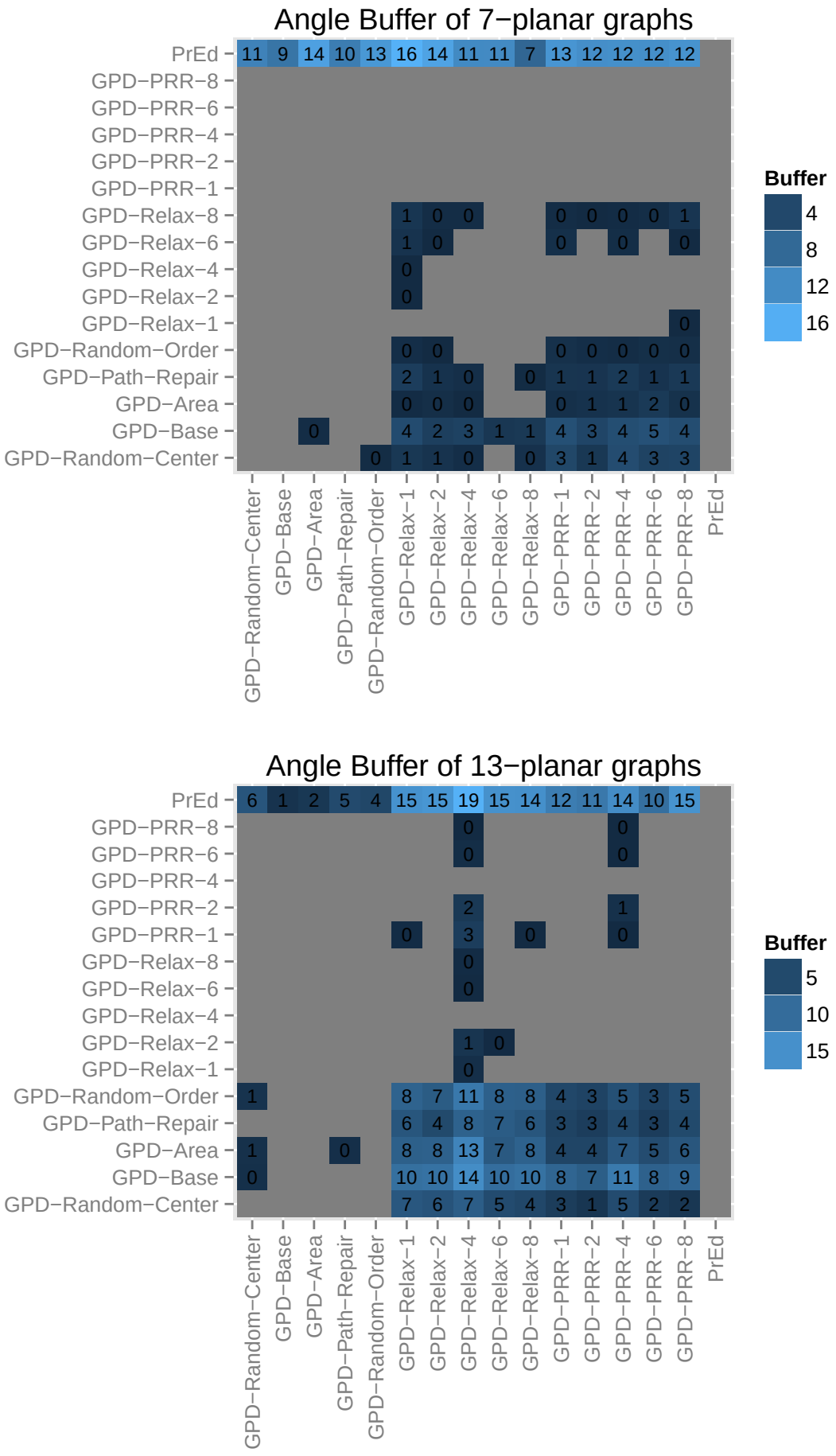
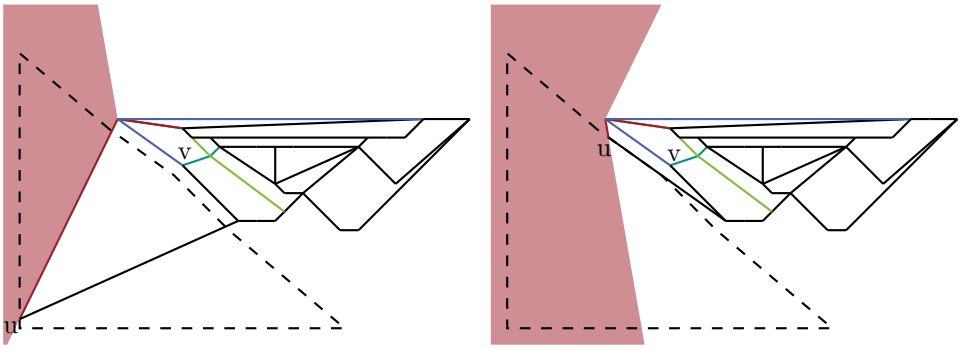
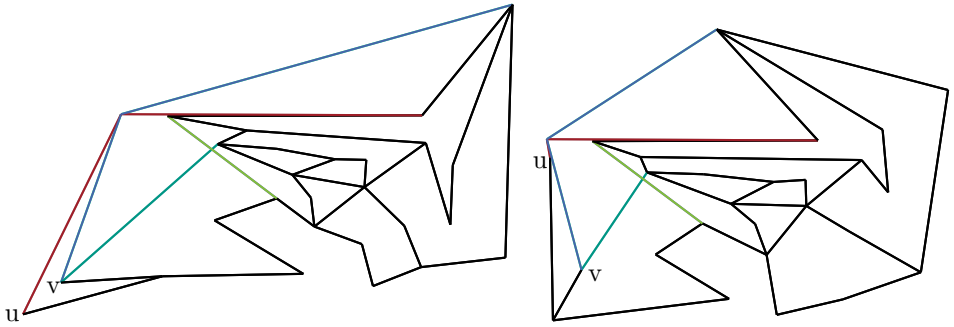


Figure 5.10.: Pairwise comparison of Geometric Planarization Drawing configurations and PrEd. Each cell represents the advantage in degree.



(a) Placement of the vertex with configuration GPD-PRR-4

(b) Placement of the vertex with configuration GPD-PRR-6



(c) Drawing after the first iteration with configuration GPD-PRR-4

(d) Drawing after the first iteration with configuration GPD-PRR-6

Figure 5.11.: Drawing of the graph “grafo5226.36” after the placement of the vertex and after the first iteration

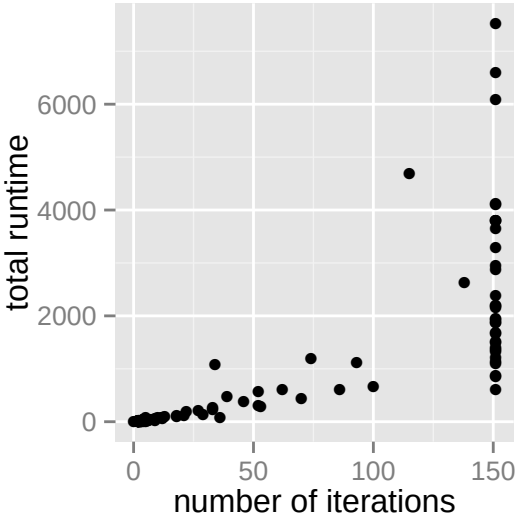


Figure 5.12.: GPD-Base: Total number of iterations used for a single drawing versus the total running time.

another in case of the 7-planar drawings. If we neglect the row corresponding to PrED, the GPD-PRR-6 would dominate the other configurations. The GPD-Relax-6 configuration dominates every other configuration in case of the 13-planar drawing.

We generated the same plots for the stretch measurements. There are only small differences in the stretch between the configurations. The corresponding plots are in Appendix B.

5.3. Running Time

In this section, we analyse the running time of our Geometric Planarization Drawing implementation. We can divide the total running time into two parts: the number of iterations until the implementation converges and the running time per iteration. Note that the running time of one iteration depends on the hardware and on the implementation of the geometric operations. Our experiments were conducted on a server with older hardware as described in Section 5.1. Figure 5.12 shows the correlation between the number of iterations and the total running time. Note that the algorithm stops when no further improvements are possible, or at the latest after 150 iterations. This leads to a wide range for the total running time at iteration 150.

5.3.1. Time per Iteration

We give a short overview over the running time of a single iteration. The worst case running time of our Geometric Planarization Drawing implementation is $O(n^3 \log n)$. Figure 5.13 shows the average running time per iteration of a drawing on the y-axis versus the number of edges on the x-axis. Taking the logarithm of the average running time suggests a degree of 2.75 for the underlying polynomial running time. A non-linear correlation against a polynomial of degree two and three does not yield a conclusive statement. We get r^2 between 0.5 and 0.6.

Figure 5.14a shows on the x-axis the maximum number of bits required to represent a single coordinates of a vertex in a single iteration. Note that the different coordinates of the vertices of a drawing may require a different number of bits. The y-axis shows the time required to compute the drawing. The colors of the data point indicate the local crossing

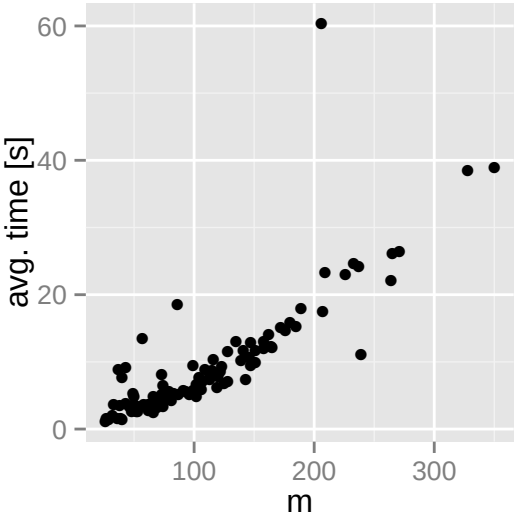
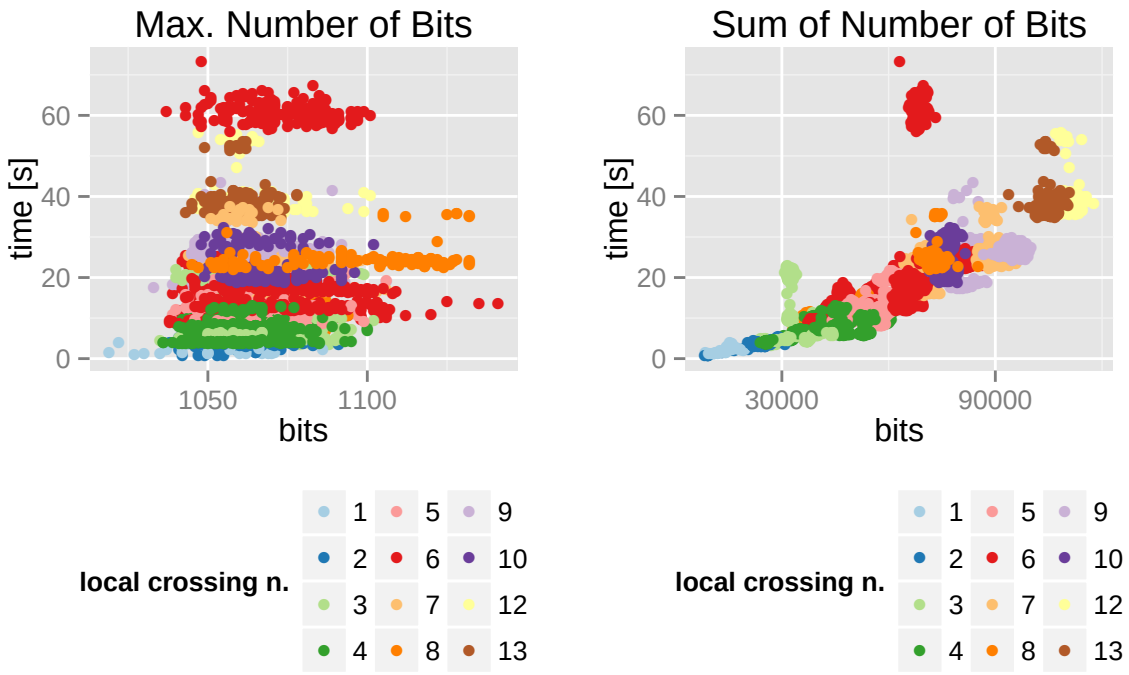


Figure 5.13.: GPD-Base: Average running time per iteration of a single drawing with respect to the number of edges.

number of the corresponding drawing. The coordinates require at most 1000 to 1200 bits in order to represent the drawing. We did not record the number of bits for a different GPD configuration, but as the plot shows, we think there is no (clear) correlation between the maximum number of bits and the running time for a single iteration.

In contrast, Figure 5.14b depicts on the x-axis the total number of bits required to represent a drawing. As the number of bits per coordinate use only a small spectrum, the plot is similar to Figure 5.13. Nevertheless, we see a red and green cluster which mark two different outliers in the running time.

These outliers can be traced back to the following behaviour. We would expect the running time of the computation of a drawing to be stable over all iterations. Figure 5.15 shows that this is not the case for our running time measurements. The Figure shows the current iteration on the x-axis and the average running time of the operation per vertex on the y-axis. For example, in Figure 5.16a iteration 100 requires on average about $0.1s$ to place one vertex. Offsetting the planarity regions requires on average less then $0.05s$. The computation of the planarity region is even faster. Note that the offsetting data series contains the iterative offsetting for independent vertices as well. The two plots show that the running time varies about a factor of 1.5. For some graphs, we observed a factor of about 4. We were not able to trace this behavior back to one specific operation of our algorithm. We observe the same decline in running time with similar proportion on two different subroutines: the offsetting and the planarity region. We measured the running time to compute a third drawing. We repeated this experiment 16 times on the same drawing. In order to rule out a hardware defect on our system described in Section 5.1, we conducted the experiment on a second compute server. Figure 5.16 shows two of the measurements. Thus, the running time measurement behaves in a unpredictable manner. Again, we observe the same behaviour of the offset computation and the planarity region. Recall that we use an external implementation for the offset computation. Since the number of bits required per iteration is almost constant, we neglect the possibility of memory effects. We cannot exclude the possibility of a lower clock rate for some time, maybe due to a too high temperature, with a 100% certainty.



(a) Runtime versus the maximum number of bits of required to represent one coordinate of the drawing.

(b) Runtime versus the total number of bits required to represent the drawing

Figure 5.14.: GPD-Base: The plots show the relationship between the bits per coordinate and the running time of one iteration

Figure 5.15.: Average running time per iteration to draw two different graphs with the GPD-Base configuration.

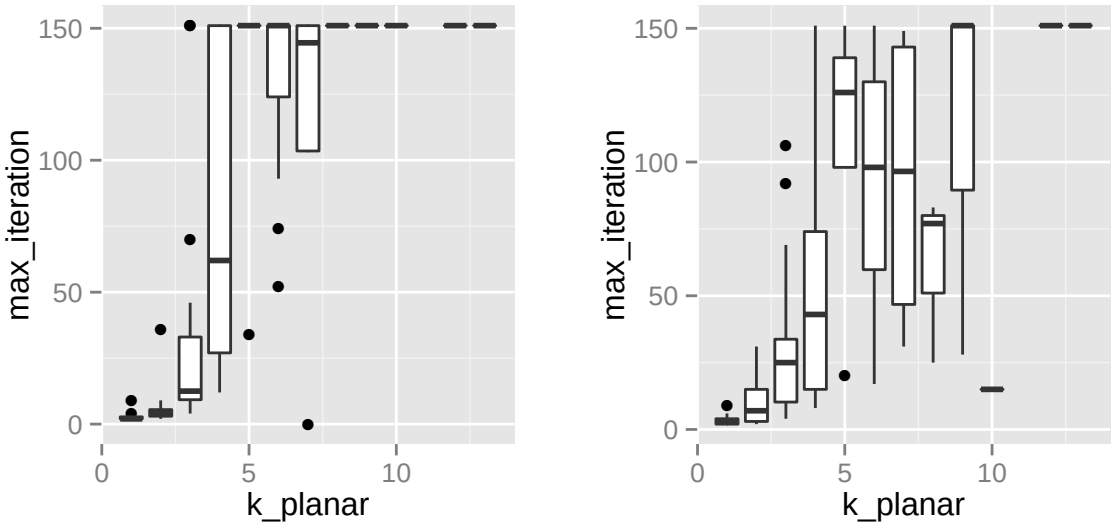
Figure 5.16.: Average running time per iteration of two computation of the same drawing.

5.3.2. Number of Iterations

Depending on the vertex order and the strength of the angle relaxation, the number of iterations vary; compare Figure 5.17. In order to evaluate the effect of the configurations on the number of iterations, we take a similar approach to evaluating the crossing angles. We compute a buffer for the number of iterations between each pair of configurations. Figure 5.18 shows the advantage of one configuration over the other. The first figure shows that most configurations have an advantage over the GPD-Relax configurations. The GPD-Relax configurations requires 150 iterations for most graphs, which is our upper bound on the number of iterations. The GPD-Relax configurations use the alternating-shelling order and relax the angles for the cones. The vertex order of one iteration is the inverse of the previous iteration. For a placement of one vertex, we relax those angles, which we did not relax in the previous iteration. This yields conflicting decisions between two consecutive iterations. The GPD-PRR configurations use a sequence of vertices, which is more consistent between two iterations. Therefore, the relaxation effects the same or at least not too many different vertices between two iterations. This yields a better overall performance. For every other pair of configurations the test does not show a great advantage of one configuration over another.

Number of Iterations versus Quality

We conclude this section with a short comparison of four of our configurations, two configurations without the angle relaxation activated (GPD-Base and GPD-Path-Repair), two configurations with an angle relaxation weight of 0.4 (GPD-Relax-4 and GPD-PRR-4). Figure 5.19 shows the plots for drawings with local crossing of 1, 3, 7 and 13. Each of the plots shows the relationship between the number of iterations and the maximum crossing angle in a single drawing. We can observe that the GPD-Base configuration performs best in case of 1-planar drawings (all orange points are hidden by the purple cluster). In the



(a) Path-Repair (b) GPD-PRR-4

Figure 5.17.: Number of iterations

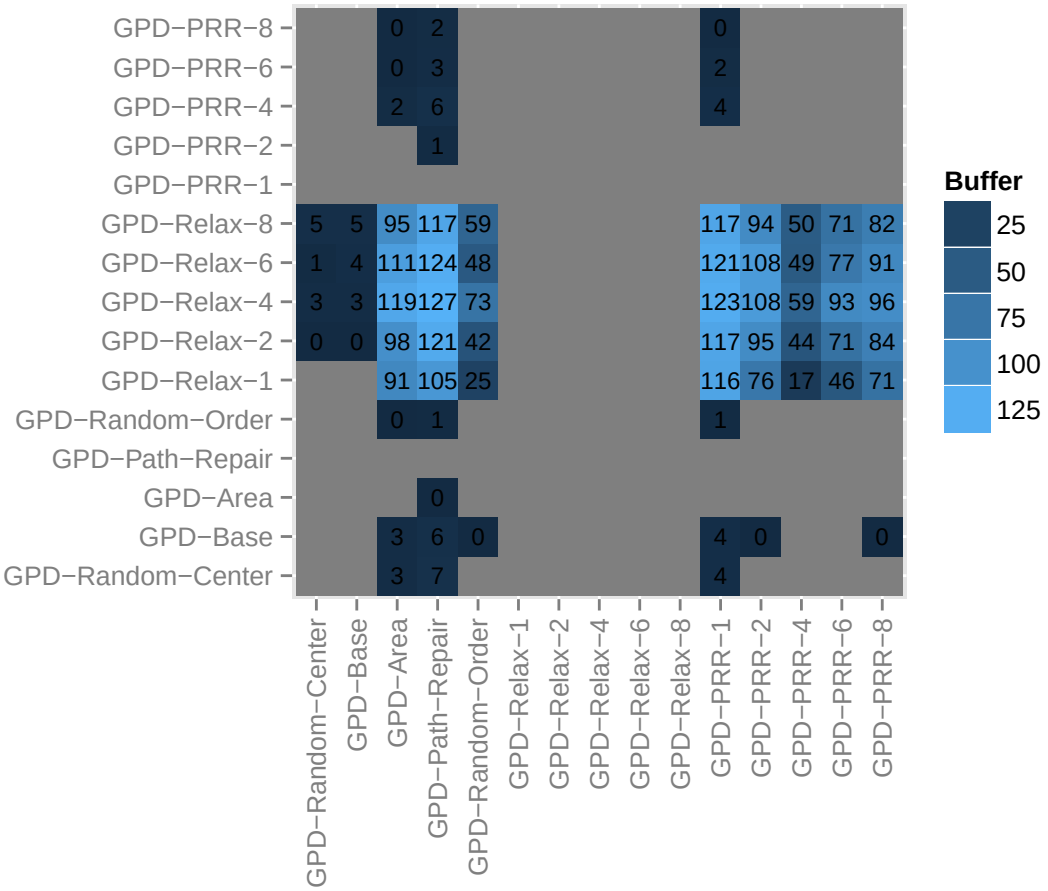
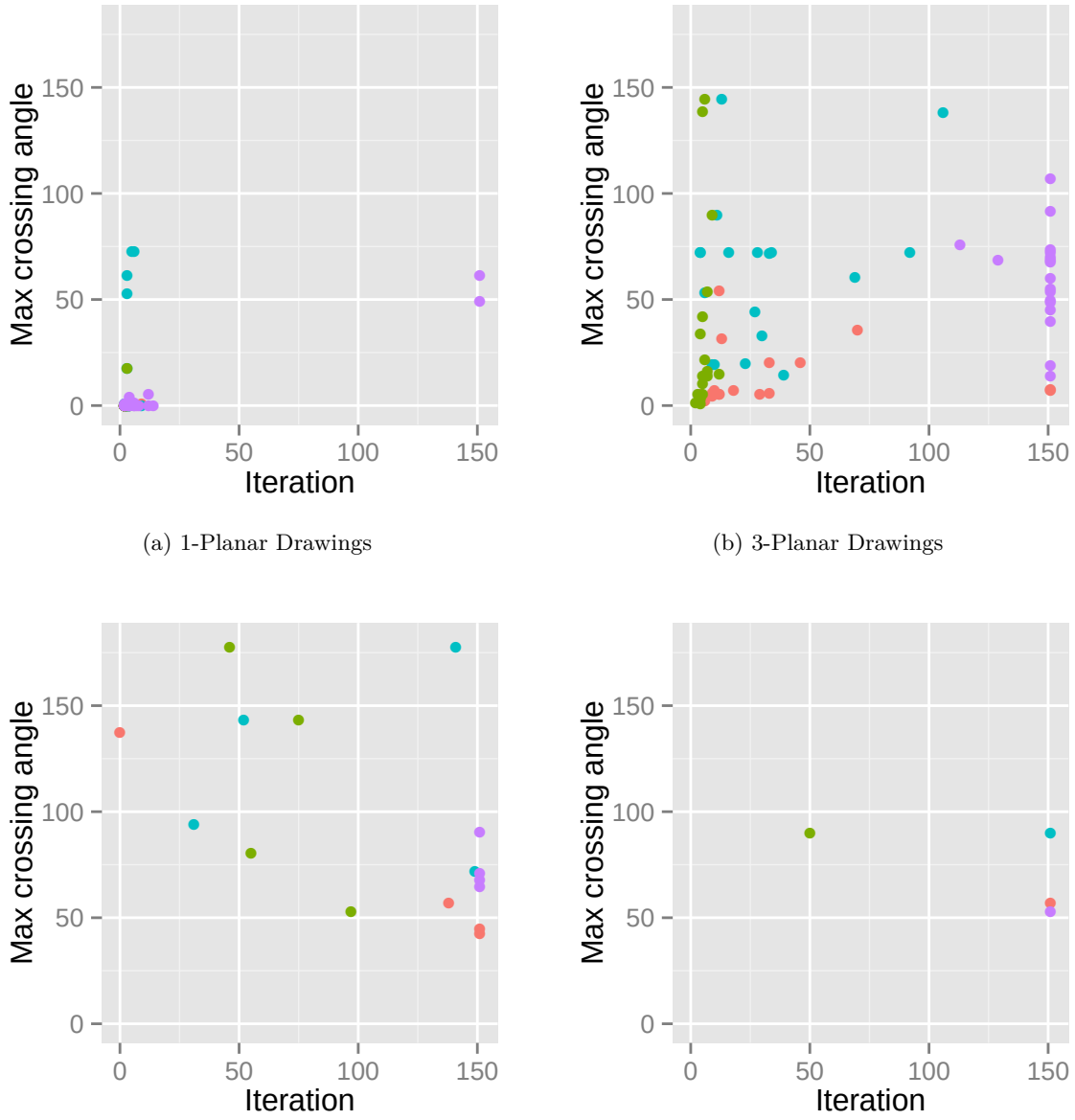


Figure 5.18.: Pairwise comparison of Geometric Planarization Drawing configurations. Each cell represents the advantage in the number of iterations.

other plots we can observe that in order to achieve a good quality of the final drawing, the configuration tend to require more iterations, especially for graphs with high local crossing number. Further, we can observe that with an increasing local crossing number the angle relaxation gains relevance.



(a) 1-Planar Drawings

(b) 3-Planar Drawings

(c) 7-Planar Drawings

(d) 13-Planar Drawings

Figure 5.19.: Maximum crossing angle versus number of iteration. Each point represents a single drawing. The color corresponds to a configuration: orange = GPD-Base; green=GPD-Path-Repair; blue = GPD-PRR-4; purple = GPD-Relax-4.

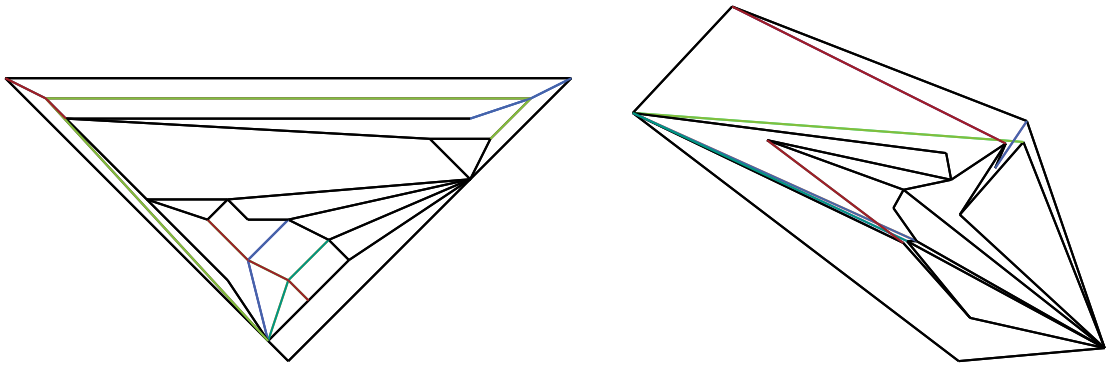
6. Conclusion

The study of Purchase [Pur97] suggests that for non-planar graph minimizing the total number of crossings is the most important optimization criteria to create good and readable drawings of non-planar graphs. Minimizing the number of crossings is known to be $\mathcal{NP}$-hard [GJ83, Bie91]. Furthermore, we can require that the drawing is a straight-line representation and ask for a straight-line drawing with a minimal number of crossings. This problem is $\mathcal{NP}$-hard as well [Bie91]. Thus, there is only little hope to find an efficient algorithm that computes a straight-line representation of a graph with a minimal number of crossings.

Nevertheless, computing a planarization of non-planar graphs and the drawing of planar graphs are well-studied problems. In practice, the single edge insertions algorithm and similar heuristics compute a good planarization with a small number of crossings. We used this topological result as a starting point to iteratively straighten the planarization paths with the restriction to preserve the combinatorial structure of an initial drawing. Mnëv [Mnë88] and Shor [Sho91] were the first to prove that the problem *stretchability* is $\mathcal{NP}$-hard and thus, the problem of drawing a planarization is $\mathcal{NP}$-hard. We are not aware of any heuristics that try to find a straight-line representation of a planarization. Thus, the Geometric Planarization Drawing approach is the first of its kind. We used a geometric approach to tackle the problem. The core of our algorithm is the planarity region. This region allows us to characterize the set of points where we can move a vertex without changing the embedding of the planarization. Our vertex placement operations optimize the crossing angles of a vertex within this region.

We took an experimental approach to evaluate our drawings. With a binomial sign test for two dependent samples extended by a buffer, we were able to show that all our GPD configurations outperforms our (adjusted) implementation of the spring-embedder PrEd. Our approach draws all 1-planar graphs of our benchmark nearly optimally, i.e., with no perceivable deviation from a straight-line representation. Most 2 and 3-planar graphs are close to optimal with only small deviations. However, there is no clear preferences between the different vertex orders.

Further, we introduced an *angle relaxation* heuristic to improve the drawing with a high local crossing number. We relaxed some constraints depending on whether or not the corresponding vertex has already been moved within one iteration. The evaluation confirmed our expectations: the relaxation is a tool with that we can further improve the quality of complex drawings, i.e., drawings with a high local crossing number. It is important



(a) Initial Drawing of graph *grafo2920* of the rome graph benchmark.

(b) GPD-Path-Repair: Final drawing after 3 iterations.

Figure 6.1.: It might be hard to distinguish between two planarization paths

that between two consecutive iterations that the vertex order is fairly stable. Otherwise, the relaxation can yield an alternating placement of the vertices between two consecutive iterations. In case of the ALTERNATING-SHELLING order in combination with an angle relaxation, the number of iterations significantly increases in comparison to the other configurations.

6.1. Future Work and Open Problems

In comparison to a force-directed approach, our Geometric Planarization Drawing approach is not able to control the minimal vertex–edge distance. This results in aesthetically unpleasing drawings. The planarization paths are almost optimal but it might be hard to visually distinguish between two planarization paths; compare Figure 6.1. We observe that our drawings with a lower crossing number admit a straight-line representation of the planarization even then if we increase the distance between the planarization paths. Thus, it would interesting to see how additional constraints, which control the vertex–edge distance would affect the overall all quality of the drawings.

We minimize the maximum active crossing angle of each vertex separately. By minimizing the maximum, we are able to avoid solving a quadratic program with potentially non-convex constraints. Nevertheless, it would be interesting to optimize the sum of the squared crossing angles such that the vertex is within the planarity region. Triangulating the planarity region yields a set of independent convex problems. Instead of minimizing one quadratic program with non-convex constraints, this yields a linear number of independent quadratic programs with a constant number of linear constraints. This smaller problem might be easier to tackle.

The Geometric Planarization Drawing approach might be applicable for the *incremental straight-line drawing* problem. Let G be graph with a (planar) straight-line drawing and let e be a new edge. In the problem we are interested in whether or not there is a straight-line drawing of $G + e$ where the drawing of G stays (fairly) stable. Note the resemblance to the *planarity of a partially embedded graph* problem; compare with Section 1.1. If we cannot insert e into G without intersections, we can use the single edge insertion algorithm to add the edge to the embedding. We can think of the result of the algorithm as an ordered sequence of edges, which the edge e crosses. With our approach we can determine locally optimal positions of the dummy vertices on these edges. We can locally adjust the positions of vertices on the surrounding of the dummy vertices to iteratively improve the drawing. In this way, the layout hopefully stays (fairly) stable.

Bibliography

[AAAG96] Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. A novel type of skeleton for polygons. pages 752–761, 1996.

[Ber99] François Bertault. A force-directed algorithm that preserves edge crossing properties. In *Proceedings of the 7th Internation Symposion on Graph Drawing (GD'99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 351–358. Springer, 1999.

[Bie91] Daniel Bienstock. Some provably hard crossing number problems. *Discrete & Computational Geometry*, 6(1):443–459, 1991.

[BLM02] Prosenjit Bose, Anna Lubiw, and J.Ian Munro. Efficient visibility queries in simple polygons. *Computational Geometry*, 23(3):313–335, 2002.

[BO79] John L. Bentley and Thomas A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979.

[BTT84] Carlo Batini, Maurizio Talamo, and Roberto Tamassia. Computer aided layout of entity relationship diagrams. *Journal of Systems and Software*, 4(2):163–173, 1984.

[Cac15] Fernando Cacciola. 2D straight skeleton and polygon offsetting. `http://doc.cgal.org/4.5.2/Manual/packages.html#PkgStraightSkeleton2Summary`, 2015.

[Can88] John Canny. Some algebraic and geometric computations in PSPACE. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC'88)*, pages 460–467. ACM, 1988.

[CEX14] Hsien-Chih Chang, Jeff Erickson, and Chao Xu. Detecting weakly simple polygons. *CoRR*, abs/1407.3340, 2014.

[CFG+14] Timothy M. Chan, Fabrizio Frati, Carsten Gutwenger, Anna Lubiw, Petra Mutzel, and Marcus Schaefer. Drawing partially embedded and simultaneously planar graphs. In *Proceedings of the 22nd International Symposium on Graph Drawing (GD'14)*, volume 8871 of *Lecture Notes in Computer Science*, pages 25–39. Springer, 2014.

[CG12] Markus Chimani and Carsten Gutwenger. Advances in the planarization method: Effective multiple edge insertions. In *In the Proceedings of the 19th International Symposium on Graph Drawing*, volume 7034 of *Lecture Notes in Computer Science*, pages 87–98. Springer Berlin Heidelberg, 2012.

[CGA15] CGAL Project, The. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.5.2 edition, 2015. `http://doc.cgal.org/4.5.2/Manual/packages.html`.

[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.

[CGMW09] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Christian Wolf. Inserting a vertex into a planar graph. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'09)*, pages 375–383. Society for Industrial and Applied Mathematics, 2009.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, Cambridge, MA, USA, 2009.

[DBT96] Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996.

[DLR11] Walter Didimo, Giuseppe Liotta, and Salvatore A. Romeo. Topology-driven force-directed algorithms. In *Proceedings of the 18th International Symposium on Graph Drawing (GD'10)*, volume 6502 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2011.

[EA81] Hossam ElGindy and David Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2(2):186 – 197, 1981.

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

[Fár48] István Fáry. On straight line representation of planar graphs. *Acta Univ. Szeged. Sect. Sci. Math.*, 11:229–233, 1948.

[FO98] Petr Felkel and Stepan Obdrzalek. Straight skeleton implementation. In *Proceedings of Spring Conference on Computer Graphics*, pages 210–218, 1998.

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

[GHKR14] Luca Grilli, Seok-Hee Hong, Jan Kratochvíl, and Ignaz Rutter. Drawing simultaneously embedded graphs with few bends. In *Proceedings of the 22nd International Symposium on Graph Drawing (GD'14)*, volume 8871 of *Lecture Notes Computer Science*, pages 40–51. Springer, 2014.

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

[GJPT78] Michael R. Garey, David S. Johnson, Franco P. Preparata, and Robert E. Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 7(4):175–179, 1978.

[GM01] Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2001.

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

[Guy72] Richard K. Guy. Crossing numbers of graphs. In *Graph Theory and Applications*, volume 303 of *Lecture Notes in Mathematics*, pages 111–124. Springer, 1972.

[HELP12] Seok-Hee Hong, Peter Eades, Giuseppe Liotta, and Sheung-Hung Poon. Fáry's theorem for 1-planar graphs. In *Proceedings of the 18th Annual International Conference on Computing and Combinatorics (COCOON'12)*, volume 7434 of *Lecture Notes in Computer Science*, pages 335–346. Springer, 2012.

[Hem15] Hemmer, Michael and Hert, Susan and Pion, Sylvain and Schirra, Stefan. `http://doc.cgal.org/4.6/Manual/packages.html#PkgNumberTypesSummary`, 2015.

[HH11] Stefan Huber and Martin Held. Motorcycle graphs: Stochastic properties motivate an efficient yet simple implementation. *Journal of Experimental Algorithmics*, 16:1–3, 2011.

[HH12] Stefan Huber and Martin Held. A fast straight-skeleton algorithm based on generalized motorcycle graphs. *International Journal of Computational Geometry & Applications*, 22(05):471–498, 2012.

[Hub11] Stefan Huber. *Computing straight skeletons and motorcycle graphs: theory and practice*. PhD thesis, University of Salzburg, Austria, 2011.

[JS87] Barry Joe and Richard B. Simpson. Corrections to Lee's visibility polygon algorithm. *BIT Numerical Mathematics*, 27(4):458–473, 1987.

[Kan96] Goos Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.

[Knu97] Donald Ervin Knuth. *The art of computer programming: Seminumerical Algorithms*, volume 2. Addison–Wesley, Bostpn, USA, 1997.

[Lee83] Der-Tsai Lee. Visibility of a simple polygon. *Computer Vision, Graphics, and Image Processing*, 22(2):207 – 221, 1983.

[Mnë88] Nikolai E. Mnëv. The universality theorems on the classification problem of configuration varieties and convex polytopes varieties. In *Topology and Geometry — Rohlin Seminar*, volume 1346 of *Lecture Notes in Mathematics*, pages 527–543. Springer, 1988.

[NP82] Jürg Nievergelt and Franco P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25(10):739–747, 1982.

[OCON82] Joseph O'Rourke, Chi-Bin Chien, Thomas Olson, and David Naddor. A new linear algorithm for intersecting convex polygons. *Computer Graphics and Image Processing*, 19(4):384 – 391, 1982.

[Pur97] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 1997.

[PW01] János Pach and Rephael Wenger. Embedding planar graphs at fixed vertex locations. *Graphs and Combinatorics*, 17(4):717–728, 2001.

[R C15] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015. `http://www.R-project.org/`.

[SAAB11] Paolo Simonetto, Daniel Archambault, David Auber, and Romain Bourqui. Impred: An improved force-directed algorithm that prevents nodes from crossing edges. *Computer Graphics Forum (EuroVis'11)*, 30(3):1071–1080, 2011.

[Sch10] Marcus Schaefer. Complexity of some geometric and topological problems. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *Lecture Notes in Computer Science*, pages 334–344. Springer, 2010.

[She03] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. CRC Press, Boca Raton, FL, USA, 2003.

[Sho91] Peter Shor. Stretchability of pseudolines is NP-hard. In *Applied Geometry and Discrete Mathematics–The Victor Klee Festschrift*, volume 4 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 531–554. Amer. Math. Soc., 1991.

[Tam87] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987.

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

[Wic09] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer, New York, USA, 2009.

[Wic11] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 2011. `http://www.jstatsoft.org/v40/i01/`.

[Zie00] Thomas Ziegler. *Crossing minimization in automatic graph drawing*. PhD thesis, Universität des Saarlandes Saarbrücken, Saarland, 2000.

Appendix

A. List of Rome-100 Graphs

List of 100 Rome graphs used in Chapter 5 to evaluate our Geometric Planarization Drawing approach.

1. grafo10079.41.graphml
2. grafo10153.100.graphml
3. grafo10240.97.graphml
4. grafo10275.97.graphml
5. grafo10325.96.graphml
6. grafo10592.98.graphml
7. grafo10861.100.graphml
8. grafo10914.96.graphml
9. grafo10964.95.graphml
10. grafo11018.99.graphml
11. grafo11092.97.graphml
12. grafo11118.99.graphml
13. grafo11145.99.graphml
14. grafo11147.100.graphml
15. grafo11280.99.graphml
16. grafo11585.36.graphml
17. grafo11637.38.graphml
18. grafo11655.91.graphml
19. grafo1210.51.graphml
20. grafo1406.46.graphml
21. grafo1509.32.graphml
22. grafo1556.63.graphml
23. grafo3424.46.graphml
24. grafo3468.44.graphml
25. grafo3549.40.graphml
26. grafo3665.54.graphml
27. grafo3668.59.graphml
28. grafo3737.54.graphml
29. grafo4001.41.graphml
30. grafo4008.62.graphml
31. grafo4218.55.graphml
32. grafo4472.68.graphml
33. grafo4502.55.graphml
34. grafo4536.83.graphml
35. grafo4550.65.graphml
36. grafo4592.56.graphml
37. grafo4630.64.graphml
38. grafo4636.82.graphml
39. grafo4839.63.graphml
40. grafo4959.59.graphml
41. grafo4973.66.graphml
42. grafo499.37.graphml
43. grafo5042.56.graphml
44. grafo5088.54.graphml
45. grafo5097.71.graphml
46. grafo5226.36.graphml
47. grafo5255.47.graphml
48. grafo5483.60.graphml
49. grafo5583.33.graphml
50. grafo5633.45.graphml
51. grafo5861.82.graphml
52. grafo5862.60.graphml
53. grafo610.29.graphml
54. grafo6127.55.graphml
55. grafo6181.40.graphml
56. grafo6319.80.graphml
57. grafo6330.39.graphml
58. grafo6392.40.graphml
59. grafo6527.69.graphml
60. grafo6530.41.graphml
61. grafo6543.37.graphml
62. grafo6545.71.graphml
63. grafo6646.77.graphml
64. grafo6827.53.graphml
65. grafo7149.74.graphml
66. grafo7278.42.graphml
67. grafo752.76.graphml
68. grafo7551.39.graphml
69. grafo7641.39.graphml
70. grafo7891.83.graphml
71. grafo7995.65.graphml
72. grafo8039.65.graphml
73. grafo8059.61.graphml
74. grafo8085.76.graphml
75. grafo8094.86.graphml
76. grafo8103.71.graphml
77. grafo8151.76.graphml
78. grafo8291.77.graphml
79. grafo8388.72.graphml
80. grafo8416.64.graphml
81. grafo8433.75.graphml
82. grafo8499.99.graphml
83. grafo8720.100.graphml
84. grafo8883.71.graphml
85. grafo8910.91.graphml
86. grafo8944.97.graphml
87. grafo9006.61.graphml
88. grafo9038.73.graphml
89. grafo9124.64.graphml
90. grafo9135.85.graphml
91. grafo9280.62.graphml
92. grafo9360.63.graphml
93. grafo947.15.graphml
94. grafo9602.82.graphml
95. grafo9671.63.graphml
96. grafo9746.68.graphml
97. grafo9754.67.graphml
98. grafo9762.78.graphml
99. grafo9775.61.graphml
100. grafo9844.82.graphml

B. Pairwise Comparison of the Stretch

For completeness, the plots for the pairwise comparison of the stretch in Section 5.2.3.

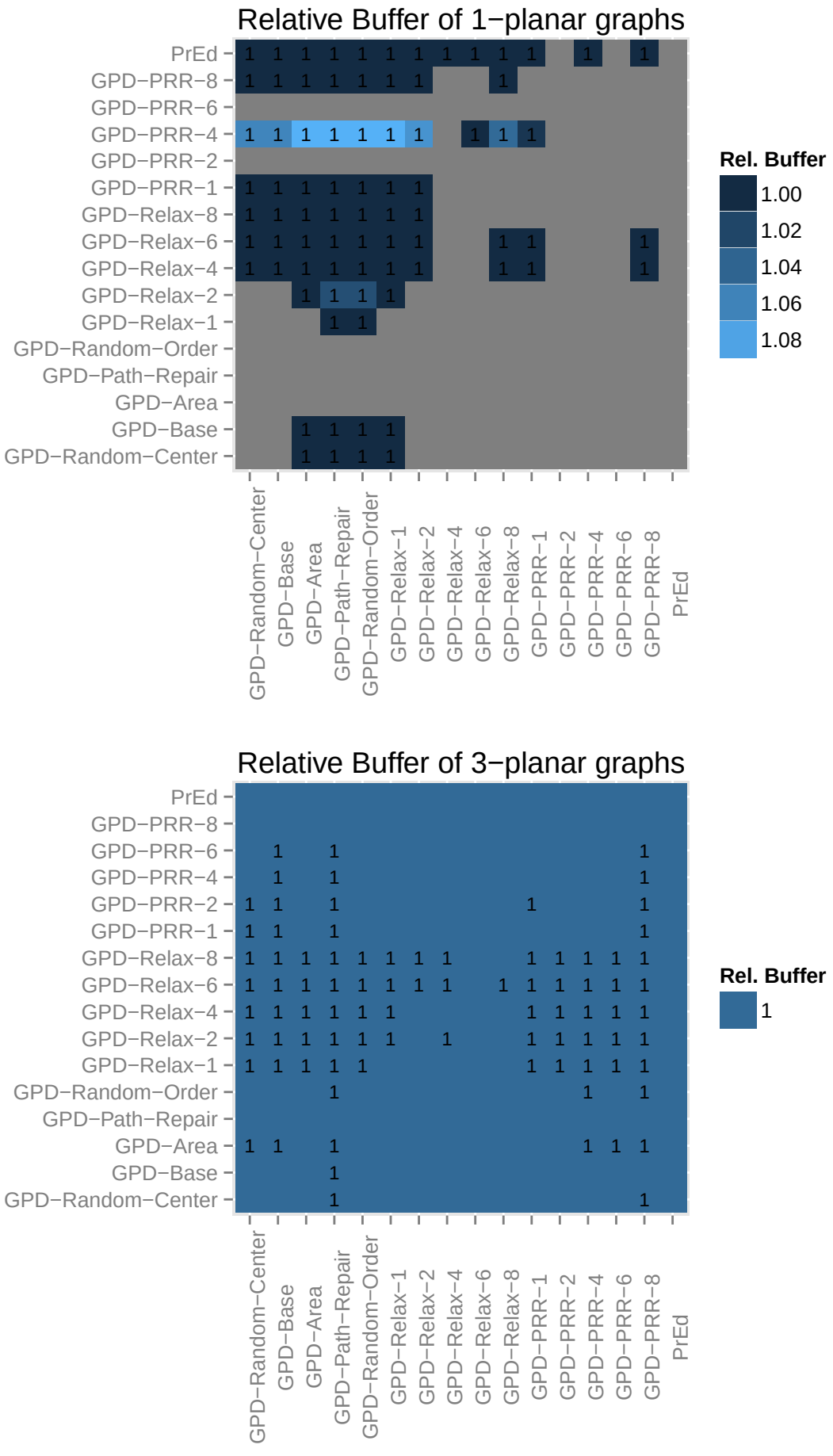


Figure B.1.: Pairwise comparison of Geometric Planarization Drawing configurations and PrEd. Each cell represents the relative advantage of the stretch. (cont.)

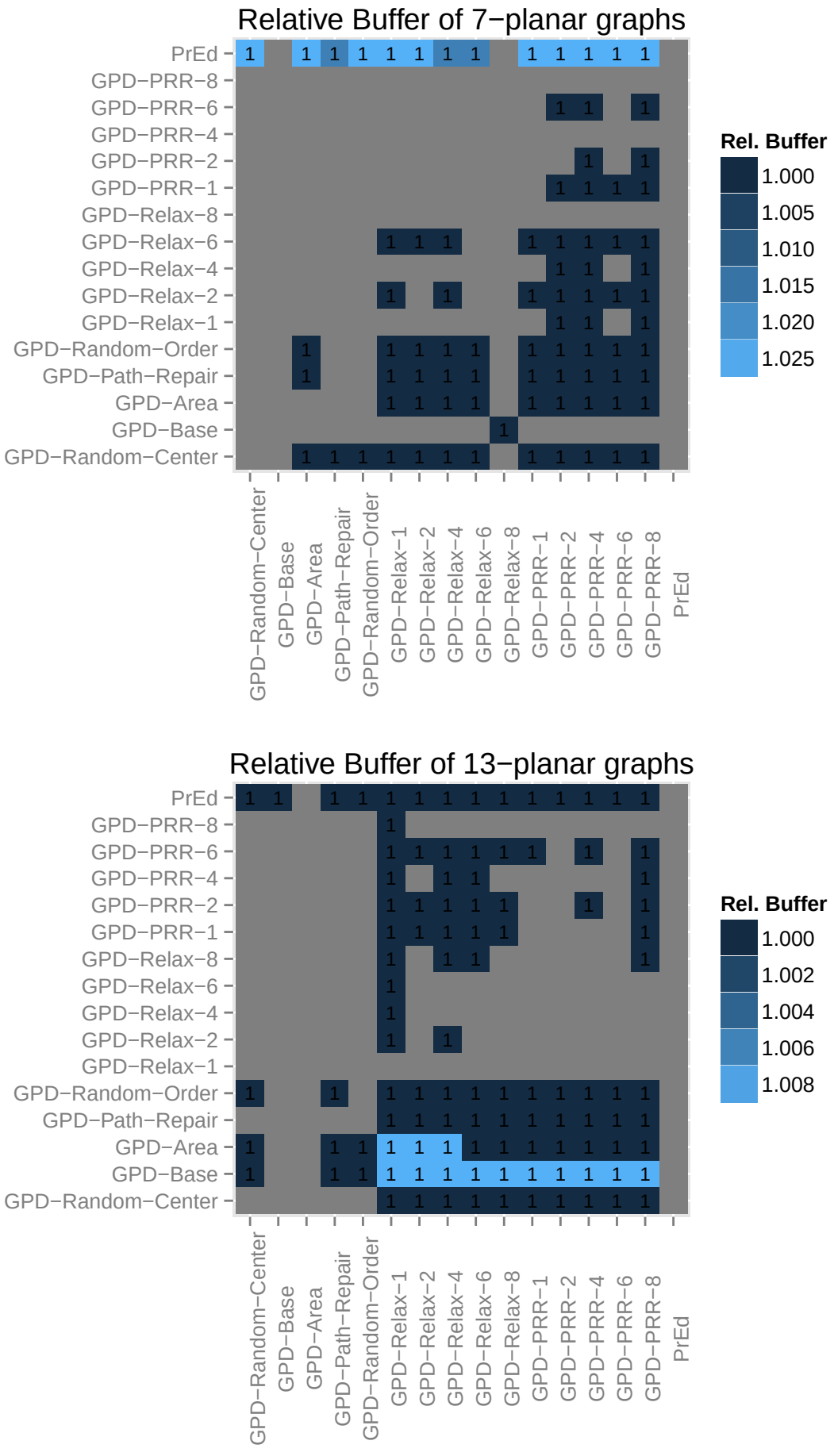


Figure B.1.: Pairwise comparison of Geometric Planarization Drawing configurations and PrEd. Each cell represents the relative advantage of the stretch.