Andre Löffler

Constrained Graph Layouts:
Vertices on the Outer Face and on the Integer Grid

Andre Löffler

Constrained Graph Layouts

Vertices on the Outer Face and on the Integer Grid

Würzburg
University Press

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2020
Gutachter: Dr. Steven Chaplick, Dr. Thomas C. van Dijk

Impressum

Julius-Maximilians-Universität Würzburg
Würzburg University Press
Universitätsbibliothek Würzburg
Am Hubland
D-97074 Würzburg
www.wup.uni-wuerzburg.de

© 2021 Würzburg University Press
Print on Demand

ISBN 978-3-95826-XXX-X (print)
ISBN 978-3-95826-XXX-X (online)
URN XXXXXX

Except otherwise noted, this document—excluding the cover—is licensed under the Creative Commons License Attribution-ShareAlike 4.0 International (CC BY-SA 4.0):
https://creativecommons.org/licenses/by-sa/4.0/

The cover page is licensed under the Creative Commons License Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0):
https://creativecommons.org/licenses/by-nc-nd/4.0/

Preface

Contents

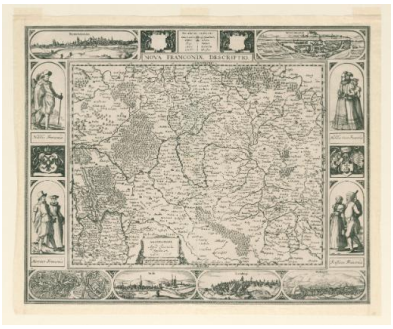
Chapter 1
Introduction

> "Within a graphic standard, a graph has infinitely many different drawings. However, in almost all data presentation applications, the usefulness of a drawing of a graph depends on its readability, i.e., the capability of conveying the meaning of the diagram quickly and clearly."
>
> – Peter Eades & Roberto Tamassia, 1989 [ET89]

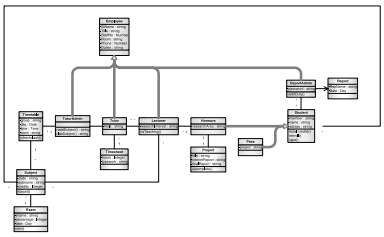
The task of producing high-quality visualizations of information combines challenges from it between being a craft as well as being an art form. Historically, visualizing data was manual work done by scholars and experts and the material used was very expensive. Thus, significant effort was put into creating *pieces of art* – a typical example can be found in Figure 1.1 (a): This geographic map of the region around Würzburg was enriched with depictions of prominent people and places of interest to illustrate the data at hand. A rather curious map of the Mediterranean sea can be found in Figure 1.2. The main body of water is shown as an oversimplified oval shape, with islands on the inside and all ports placed around it, preserving their cyclic order while ignoring real-world distances and geographic resemblances. Today, *technical schemata* such as UML diagrams are regularly used in software engineering and project management to convey complex information quickly and without ambiguity – see the UML diagram in Figure 1.1 (b): The nodes contain information about the entities they represent, whereas the links between the nodes are annotated with level and type of connection. In some cases, designers intentionally blur the lines between craft and art, creating beautiful schematic representations such as the metro map shown in Figure 1.1 (c).

Nowadays, there is a plethora of information to visualize. It ranges from the abstract data of social networks to blueprints and flowcharts to geographic data to data from many other sources. Each domain is asking for it's own meaningful and pleasant visualization to assist the viewer in capturing all relevant information. Sometimes, tremendous effort is spent drawing large networks by hand: See for example the Human Metabolism Map shown in Figure 1.3 – the layout was crafted by five people over the course of more than a year. It has proven to be an effective instrument to researchers [TSF+13]. Unfortunately, not every drawing of such size can be hand-made by experts.

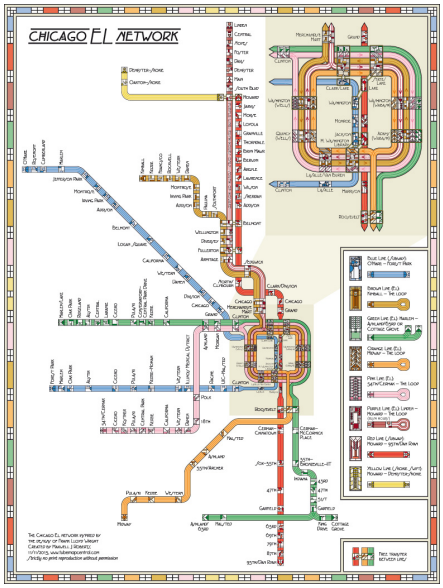
Therefore, the need for *good* automated drawing algorithms arose, giving birth to the field of *graph drawing*. Graph drawing focuses on discovering the structural properties of different classes of networks, exploiting them to a develop graphic standard tailored to each class respectively. In graph drawing, the *node-link-metaphor* is commonly used:



(a)



(b)



(c)

Figure 1.1: (a) Historic map "Nova Franconiae descriptio" of the region around Würzburg, created 1626, produced by etching[1]. (b) An UML diagram commonly used to aid in object-oriented programming, taken from *Automatic Layout of UML Class Diagrams in Orthogonal Style* by Eiglsperger et al. [EGK+04]. (c) A map of the Chicago public transport network, created by Maxwell Roberts in 2015; the design is inspired by Frank Lloyd Wright, found at `http://www.tubemapcentral.com`.

each individual datum is represented by a node and the connection between different data elements is modelled by connecting the nodes via arcs. In most of today's literature, nodes are called *vertices* and links are called *edges*, reusing the names established in discrete maths. Most commonly, heavy dots are used to represent vertices and edges are oftentimes either straight-line segments or simple curves. Modifying and augmenting these objects already allows for a lot of different drawing styles: Is the vertex labeled? How is it shaped? Are different colors used to represent different data types? Are the edges curves or straight lines? Do they have arrow-tips indicating directions? Is additional data – such as distance or connection strength – represented adjacent to them? In general, however, graphic standards are not limited to changing the visual representations of vertices and edges.

This work focuses on graphic standards that restrict where vertices are allowed to be placed. In classic graph drawing literature, the goal oftentimes is to draw a given graph

[1] Signature 36/G.f.m.9,12,139, licensed under CC BY-NC-ND 4.0 by the Würzburg University Library; found at `http://vb.uni-wuerzburg.de/ub/permalink/36gfm912139_105787462`.

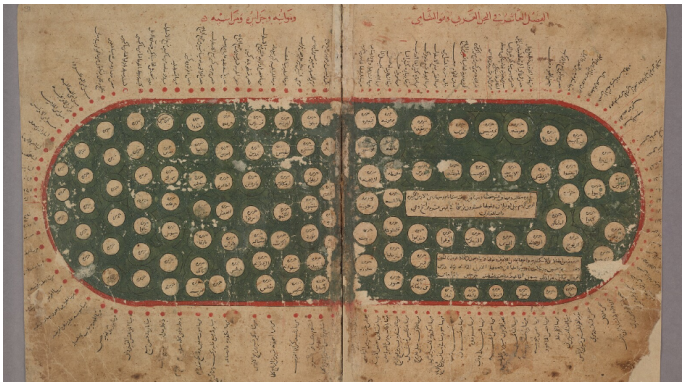


Figure 1.2: "Kitāb Gharā'ib al-funūn wa-mulaḥ al-'uyūn ('The Book of Curiosities')", 1190–1210, Bodleian Library MS. Arab. c. 90, Photo: Bodleian Libraries, University of Oxford. This arabic map of the Mediterranean sea shows the islands as well as the ports around the coast. It has all port (vertices) placed on the same layer and uses boundary labeling. It can be found online at `https://digital.bodleian.ox.ac.uk/inquire/p/d6fc79a9-a87a-48cb-aebe-edd36c2158c6`.

onto a two-dimensional plane – like a sheet of paper, blackboard or computer monitor. The placement of each vertex is described by assigning two coordinates (usually using real numbers to represent x- and y-coordinate). We now list several important results on different graphic standards.

De Fraysseix, Pach, and Pollack [dFPP90] considered drawings of bounded size and with vertices placed at integer coordinates. They managed to show that every graph that can be drawn planar – without crossing edges – has a vertex layout using integer coordinates on an area with size quadratic in the number of nodes. Networks representing hierarchical structures, such as trees, can be drawn using the layered layout by Sugiyama, Tagawa, and Toda [STT81], where vertices of the same level obtain the same y-coordinate. A special case of these layered layouts are radial layouts: Layers are represented by concentric circles and vertices on the same level have the same distance to the center. Outerplanar graphs can be imagined as an even more special case: All vertices have to be placed on the same circle, all edges are routed inside the circle and no pair of edges is allowed to intersect. Outerplanar graphs are also known as planar permutation graphs, introduced by Chartrand and Harary [CH67]. Fruchterman and Reingold [FR91] take a different approach, restricting how close together vertices are allowed to be drawn. They model the repulsive forces of springs, trying to find an equilibrium by expanding dense vertex clusters at the expense of less dense parts. Another layout restriction concerning polyline edges focuses on limiting the number of different edge slopes used in the drawing. Rectilinear and octilinear drawings – known for Manhattan-geodesic drawings [KKRW10] and classic metromap drawings [NW11] – are well-studied graphic standards with a large set of applications. For a broad overview and diverse selection of results on bounded slope numbers, consider the work of Dujmović, Suderman, and Wood [DSW07].

Unfortunately, there are graph drawing and layout problems that cannot be solved efficiently unless we have $\mathcal{P} = \mathcal{NP}$. Picking coordinates for the vertices of a planar graph such that the resulting drawing is crossing-free and all edges are straight-line and have prescribed lengths is $\mathcal{NP}$-hard, as shown by Eades and Wormald [EW90]. Given a non-planar graph, it is also $\mathcal{NP}$-hard to decide whether the vertices can be placed such that all edges are straight segments and that if two edges cross, they do so at right angles; see Argyriou, Bekos, and Symvonis [ABS12]. Minimizing the total area needed to draw a planar graph using straight-line edges at integer coordinates is $\mathcal{NP}$-hard, shown by Krug and Wagner [KW08]. Cabello [Cab06] showed that even deciding for a given set of coordinates and planar graph, if there is a mapping of the vertices to the coordinates such that the resulting drawing is also planar is $\mathcal{NP}$-hard.

As discussed above, we consider a drawing to be good, if it is visually pleasing and transports information without ambiguity. Some graphic standards are better suited for placing labels next to vertices, some allow for a better perception of graph distance and connectivity inside the network. We have already hinted applications in transit map drawing and organizational charts, but this work puts the focus on theoretical results.

1.1 Outline of this Book

In this work, we answer selected questions on two different types of graph layout strategies, that both restrict vertex placement. This naturally partitions the Chapters 3 to 7 of this book into two parts as follows: Part One examines graph drawings that have all vertices accessible from a common outer face. Part Two looks into graph drawings with vertices at integer coordinates.

1.1.1 Part One: Vertices on a Common Outer Face

Beyond Outerplanarity. In the first chapter of Part One, we look into the structural properties of graphs that admit non-planar *convex drawings*; that is, all vertices are placed in convex position – defining the boundary of the outer face – and all edges are straight lines going between the vertices.

In Chapter 3 – the first chapter of Part One –, we consider two families of graph classes with nice convex drawings. These families are defined by the crossing patterns that edges of members of these classes are allowed to make. These classes are *outer k-planar* graphs – where each edge is crossed by at most k other edges – and *outer k-quasi-planar* graphs – where no k edges can mutually cross.

For the family of outer k-planar graphs, we show $(\lfloor\sqrt{4k+1}\rfloor + 1)$-degeneracy. As an immediate consequence we get that every outer k-planar graph can be $(\lfloor\sqrt{4k+1}\rfloor + 2)$-colored, and this bound is tight. We further show that every outer k-planar graph has a balanced separator of size at most $2k + 3$. For each fixed k, these small balanced separators allow us to test outer k-planarity in quasi-polynomial time, hence recognizing

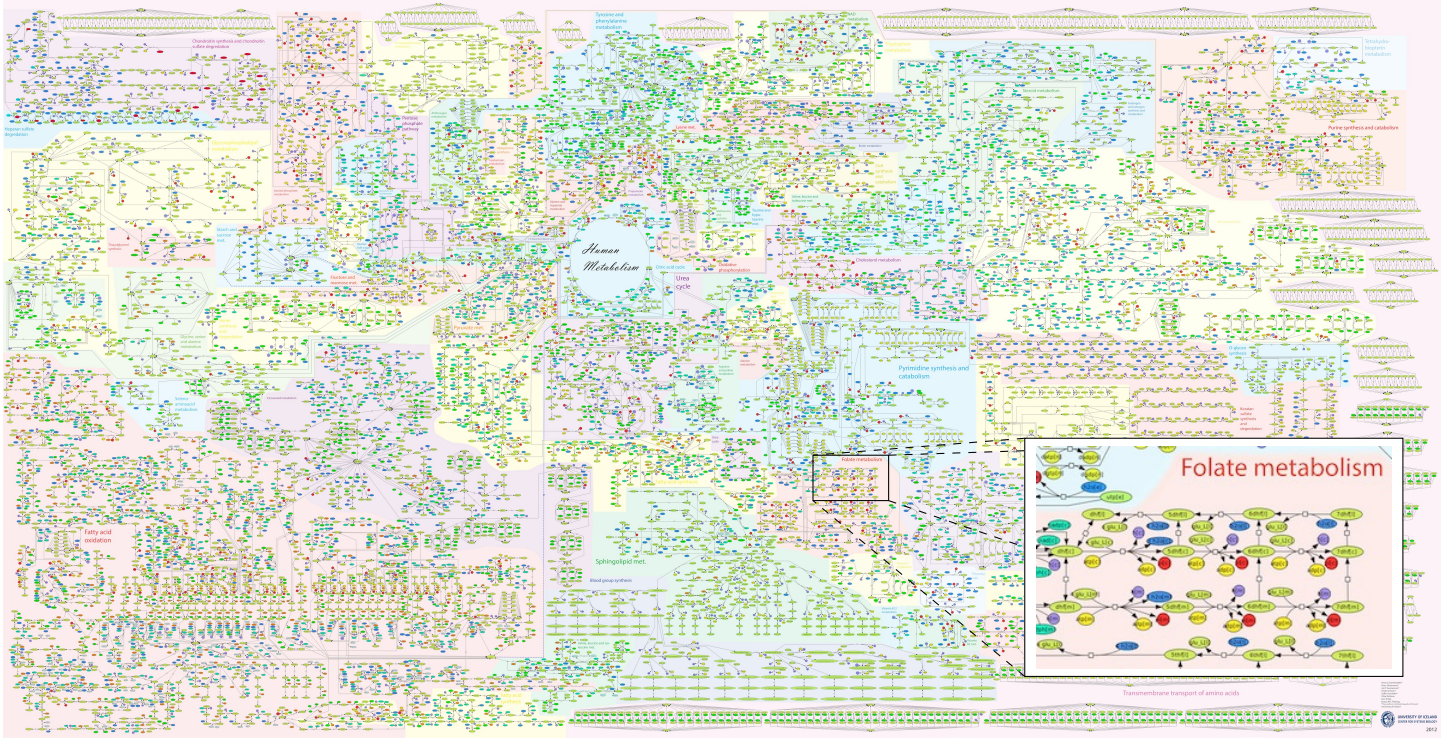


Figure 1.3: The Human Metabolism Map, published in Thiele et al. [TSF+13], is the largest hand-drawn network in biology. An interactive online explorer is available at `https://www.vmh.life/#reconmap`, see Noronha et al. [NDG+16].

membership in any class of this family is not $\mathcal{NP}$-hard unless the Exponential Time Hypothesis [IP01] fails.

The other family of graph classes is that of outer k-quasi-planar graphs. We discuss the edge-maximal graphs of this family, which have been considered previously under different names. We also construct planar 3-trees that are not outer 3-quasi-planar, showing that planar graphs and outer 3-quasi-planar graphs are incomparable. In addition, we observe simple bounds on the page number and chromatic number of all outer k-quasi-planar graph in terms of k by considering previously known results.

In the last section of this chapter, we further restrict outer k-planar and outer k-quasi-planar drawings to *closed* drawings and subsequently to *full* drawings. A drawing of a graph from either family is closed, when the sequence in which the vertex appear on the outer face's boundary is a cycle in the graph; a drawing of a graph from either family is full, when no crossing appears on the outer face's boundary. Naturally, any closed drawing is also full. For each k, we express *closed outer k-planarity* and *closed outer k-quasi-planarity* in *extended monadic second-order logic*. Thus, by Courcelle's Theorem, closed outer k-planarity is linear-time testable, since outer k-planar graphs have bounded treewidth. Using the test for closed outer k-planarity as a subroutine, we can also test full outer k-planarity in linear time.

Polygonal Boundaries. Many drawing algorithms for planar graphs work recursively. Given a planar drawing of a subgraph, each step of the recursion extends it. The extensions work in a way that maintains the structural properties of the previous drawing that made the extension possible. This is oftentimes done by joining subgraphs at distinct vertices – e.g. combining two trees by merging the root vertex of one tree with a leaf vertex of the other – or by drawing subgraphs inside the faces of the previous drawing, connecting the new vertices to those defining the face. For the latter, the shape of the face to be drawn in can be prescribed, requiring the extended drawing to respect this shape without inducing edge crossings. This raises the question if a given planar graph can be drawn inside a prescribed outer face such that the resulting drawing is also crossing free.

In Chapter 4, we consider this question in a special setting: All vertices of the subgraph are already placed on the prescribed face's boundary. Thus drawing the subgraph only requires adding the missing edges inside the face. If the shape of the face is convex, this problem is equivalent to testing whether the subgraph is outerplanar. If it is not convex, insisting on drawing edges as straight lines can easily make them cross the face's boundary. Therefore we consider the following problem: Given a drawing of the outer face as a simple polygon with m corners, an outerplanar graph with n vertices and a mapping of the vertices to the face's boundary, can the edges be drawn inside the face with at most one bend per edge? We prove that this can be decided in $O(mn)$ time if such a drawing exists. We do so by giving an algorithm that, in the positive case, also outputs such a drawing.

1.1.2 Part Two: Vertices at Integer Coordinates

Moving to the Grid Optimally. Until this point, we were concerned with where the vertices are placed, but not at what precision this placement is stored. Taken as a given that real computers work within hard limitations, and thus can only store numbers at finite precision, approximating and rounding numbers in the process is inevitable. Greene and Yao [GY86] noted that repeatedly performing geometric operations – such as checking point-in-area containment and intersecting lines – on coordinates of finite precision can result in rapidly growing rounding errors.[2]

Actually working with limited resources creates the need for an algorithm that transforms a given drawing into a drawing of lower coordinate precision without damaging the embedding or completely overthrowing geometric similarity. Preserving topology and geometric similarity are most important when working with data like real-world road networks. The transformation also removes unnecessary detail and reduces space consumption as well as the computation time of algorithms working with the coordinates. In the first chapter of Part Two – Chapter 5 – we investigate the Topologically-Safe Grid Representation problem for given straight-line drawings of planar graphs.[3] We show that the problem is $\mathcal{NP}$-hard for several different objective functions and provide an integer linear programming formulation to compute optimal solutions. We also provide an experimental evaluation on the performance and limitations of our approach.

Rounding to the Grid Heuristically. We originally started looking into the Topologically-Safe Grid Representation problem with a geographic application in mind, engaging it from a graph-drawing perspective. With an $\mathcal{NP}$-hardness result and an exact solution that is infeasible for practical applications presented in Chapter 5, we ask for an efficient algorithm that transforms a given drawing into a topologically equivalent grid drawing.

In Chapter 6, we tackle this problem from a different angle by providing a randomized heuristic algorithm. Since we will have shown that finding an optimal solution is hard, we instead ask for a reasonable result. Our algorithm consists of two two stages of simulated annealing, each a with different objective function. Stage One focuses on finding a feasible solution – a non-optimal drawing with all vertices placed on grid points – by reducing the overall "density" of the drawing by moving vertices away from each other. Stage Two takes the feasible drawing, but switches the objective to reducing the rounding error induced by the first stage. We discuss various feasibility procedures and evaluate their applicability on geographic networks. We demonstrate that a straightforward annealing approach without stage one has difficulty finding any feasible solution at

[2] Motivated by actually putting the drawings onto the *screen*, work on this topic usually considers points to be centered inside pixels. Transforming pixel-centers to (crossing) points on the integer grid only requires uniformly shifting all object by half of a unit in both dimensions.

[3] Most of the results in Chapter 5 have already been published as part of a Master's Thesis [Löf16]. We choose to include them here again for two reasons: To provide a more profound experimental evaluation; and as a foundation for the work presented in Chapter 6.

all. We also discuss parameter selection for the second simulated annealing step, which tries to reduce rounding error.

Recognizing Nets of Orthogonal Polyhedra. In computational geometry, the net of a polyhedral surface is a commonly studied object. The Rigidity Theorem given by Cauchy in 1813 can be restated within today's notation as follows: When a graph is embedded on the surface of a convex polyhedron and the angles within each face of that graph are given, the dihedral angles of the faces on the surface are uniquely determined. This naturally raises questions about other polyhedral surfaces on which graphs can be embedded uniquely. Biedl and Genç [BG09] studied orthogonal polyhedral surfaces of genus 0 with connected graphs – that is, the angles between edges as well as those between faces are multiples of 90°. Restricting faces to be orthogonal polyhedra and requiring edge lengths to be integer, we get that all corners and creases of the polyhedral surface end up being placed on the three-dimensional integer grid. Biedl and Genç give the stable linear-time BundleOrientation algorithm to translate Cauchy's theorem to orthogonal polyhedral surfaces of genus 0: This algorithm can determine the unique set of dihedral angles for a given net with rectangular faces or report that no such set of angles exists.

In Chapter 7, we consider one question that was left open by Biedl and Genç – whether or not a similar translation exists for orthogonal polyhedral surfaces of genus 1 or higher. They give an example instance of genus 1 on which their algorithm can fail. To answer their open question in the affirmative, we introduce the IteratedBundleColoring algorithm. It uses the original BundleOrientation algorithm repeatedly and exhaustively. We show that it is capable of finding a set of dihedral angles for orthogonal polyhedral surfaces of arbitrary genus. We do so by arguing how it re-discovers the dihedral angles matching those of a polyhedron realizing the input graph.

Chapter 2
Basic Definitions

This book considers questions about graph drawings under different layout constraints. To answer them, we provide algorithms as well as results from complexity theory. For a general overview on these topics, we refer to two standard textbooks – *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein [CLRS13] for algorithms and *Computers and Intractability: A Guide to the Theory of NP-Completeness* by Garey and Johnson [GJ79] for complexity theory. In this chapter, we define the essential concepts used throughout this book.

2.1 Graphs

2.1.1 Combinatorics

Graph Terminology. A *graph* $G = (V, E)$ is a tuple of sets; the elements of set V are the *vertices* – or *nodes*[1] – of G, the elements of E are the *edges*. We use $n = |V|$ and $m = |E|$ to refer to the sizes of these subsets respectively. In an *undirected* graph, each edge $e \in E$ is an unordered pair $\{u, v\} \in V \times V$ of vertices, whereas *directed* graphs use ordered tuples $(u, v) \in V \times V$, indicating that the edge goes from start vertex u to end vertex v. We use $V(G)$ and $E(G)$ to address the sets of vertices and edges in G respectively whenever the graph in question is not immediately clear from context.

The vertices of an edge are also called *endpoints*. In this thesis, we do not treat directed edges differently; hence, we use the tuple-notation for both variants, considering all edges to be undirected. Whenever the direction of an edge is relevant, we explicitly state the direction the edge is going. We say that a graph G is *connected* when for every non-empty subset of vertices $A \subset V(G)$, we find at least one edge $(u, v) \in E(G)$ with $u \in A$ and $v \in V \setminus A$.

For edge $e = (u, v)$, we say that u and v are *incident* to e (and vice versa). In addition, two vertices are *adjacent*, if they are connected by an edge – symmetrically, two edges are adjacent, if they share a common endpoint. The *degree* $\deg v$ of a vertex v is the number of edges incident to v.

A *subgraph* $G' = (V', E')$ of a graph G has the following properties: The vertex set $V' \subseteq V(G)$ is a subset of the original vertex set and the edge set $E' \subseteq E(G)$ is a subset of all edges $(u, v) \in E(G)$ with $u, v \in V'$. If a graph is not connected, the maximal

[1] We rarely use the term "node". We usually use the term "node" to refer to vertices in an auxiliary graph, making sure they are not mistaken for elements of the primary graph.

connected subgraphs of G are called it's *connected components*. Naturally, each connected component itself is a graph – possibly containing only as little as a single vertex.

The *subdivision* of an edge $e = (u, v)$ is created by deleting e from G, adding another vertex v' and adding the edges (u, v') and (v', v). A graph G' is a *subdivision* of graph G if G' can be created subdividing some of the edges of G.

Special Graphs and Graph Properties. Next, we look into some special graph classes related to relevant structural properties of abstract graphs. In the following, let $G = (V, E)$ be a graph.

Let $u, w \in V$ be vertices of G. We say that G contains a *path* P of length k connecting u and w if there is a connected k-element subset of edges $\{e_1, e_2, \ldots, e_k\} \subset E(G)$ creating the sequence $e_1 = (u, v_1), e_2 = (v_1, v_2), e_3 = (v_2, v_3), \ldots e_k = (v_{k-1}, w)$. We call u and w the end vertices of the path. Each end vertex appears only once in the edge sequence, all other vertices appear exactly twice. Naturally, we have $|V(P)| = |E(P)| + 1$. A *cycle* of length $k \geq 3$ is a closed path[2] – a sequence of edges starting and ending on the same vertex – $e_k = (v_{k-1}, u)$ and $e_1 = (u, v_1)$. A *tree* is a graph, that does not have a subgraph that is a cycle. The degree-1 vertices of a tree are called it's *leafs*. A tree containing all vertices of a given graph G is a *spanning tree* of G; a *Hamiltonian path* is a path of G that visits every vertex of G exactly once; a *Hamiltonian cycle* is a *Hamiltonian path* that is also closed. When G has a Hamiltonian cycle, we also call G a Hamiltonian graph.

A *chord* is an edge e connecting two vertices on the same cycle (of length at least 4) that e itself is not part of[3]. A cycle is an *induced cycle* if no pair of vertices on that cycle is connected by a chord.

The edge set E of the *complete graph* $K_n = (V, E)$ on n vertices contains all possible two-element subsets of V, that is, $E = \binom{V}{2}$ (and $m = n \cdot (n-1)$). A graph contains a *clique* of size k when it has K_k as a subgraph. The complete graph K_5 is shown in Figure 2.1 (a), the edges highlighted in red are a Hamiltonian cycle.

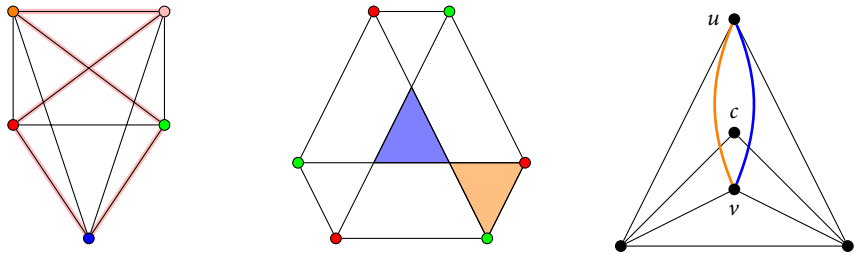
A *(vertex) coloring* of G is a function $c\colon V \to C$ mapping the vertices of G to a fixed set of colors[4] C such that for every edge $(u, v) \in E$, the $c(u) \neq c(v)$ holds. The *chromatic number* $\chi(G)$ is the size of the smallest set $C_{\min}$ for which a valid coloring of G exists; for simplicity we say that a graph G with $\chi(G) = x$ is *x-colorable*. Naturally, the chromatic number of the complete graph is $\chi(K_n) = n$, and a famous result by Appel and Haken [AH76] states that all planar graphs are 4-colorable; trees, paths, cycles of even length are 2-colorable, non-trivial cycles of odd length require a third color. A valid 5-coloring of K_5 is shown in Figure 2.1 (a).

The 2-colorable graphs are also called the *bipartite* graphs – their vertex set can be partitioned into two groups V_ℓ and V_r. The special family of *complete bipartite* graphs

[2] Cycles of length 1 are edges with both end points being the same vertex, usually called (self-) *loops*. Cycles of length 2 can only appear in directed graphs or multigraphs; they are usually called *lenses*. Both special cases will not be considered in this thesis.

[3] Think of a chord as a shortcut through the cycle. Naturally, triangles cannot have shortcuts.

[4] Despite the name, we generally use natural numbers or letters as colors, not "red, blue, green, …".



(a) (b) (c)

Figure 2.1: Illustrations of basic definitions: (a) The complete graph K_5: One of many Hamiltonian cycles in red; the vertex-colors show a valid 5-coloring. (b) The complete bipartite graph $K_{3,3}$: the central (blue) face is bounded only by partial edges and thus has no vertices incident to it; the bottom-right (orange) is not adjacent to the central face. (c) The edge (u, v) can be drawn in two different ways – going around vertex c to the left (orange drawing) or to the right (blue drawing). Each drawing crosses a different edge incident to c, creating two different subdivisions that both follow the same rotation system.

$K_{a,b}$ with $a = |V_\ell|$ and $b = |V_r|$ contain every possible edge between these groups – we have $E = V_\ell \times V_r$.

Recognition. When considering structural properties, one would generally want to test whether a given graph admits it. The *recognition problem* can generally be stated as follows: "Does graph G belong to the class of graphs with property X?" Some recognition problems are easy – such as testing planarity –, while testing other properties can be very challenging – like searching for a Hamiltonian path. Oftentimes, in graph drawing – as well as in this book – the property to test for is whether or not a given graph can be drawn according to a specific graphic standard.

2.1.2 Drawings

In the previous section, we have defined the elements of abstract graphs as well as some of their structural properties that are of general relevance to this book. To visually represent – or *draw* – an abstract graph, we use the *node-link metaphor*. We have already worked with the intuitive nature of this metaphor in the Introduction when giving an overview on different graph layout approaches. We now formally define the basic concepts and notation used for drawing graphs using this metaphor.

Vertices and Edges. A drawing Γ of graph G is a visual representation placed in some k-dimensional Euclidean space. The drawing maps each vertex v of G to a k-dimensional

coordinate vector $\Gamma(v)$ with entries from $\mathbb{R}$. Since we want to visualize drawings – to actually be able to look at them –, we restrict ourselves to either $k = 2$ or $k = 3$. For $k = 2$, we get the well-known *Cartesian plane* $\mathbb{R}^2$, most commonly associated with a blackboard or sheet of paper to draw on; for $k = 3$, we get the 3-dimensional *space* $\mathbb{R}^3$. Producing a 3-dimensional drawing oftentimes involves some 3-dimensional surface to draw onto, physically building objects representing the vertices and edges of G or using $3D$ computer graphic tools.[5] We identify the $\Gamma(v)$ with it's associated vertex and refer to the entries of the vector as x, y (and z) coordinate of v in Γ. Vertices are commonly represented by heavy dots.

Edges are drawn using *Jordan curves*. A Jordan curve is a continuous injective map d of the interval $[0,1]$ to $\mathbb{R}^2$ (or $\mathbb{R}^3$ respectively). That is, for $i, j \in [0,1]$ we have the following two conditions: When j converges towards i, we get $\lim_{j \to i} d(j) = d(i)$ – there are no abrupt changes in value – and for $i \neq j$ we have for $d(i) \neq d(j)$ – a Jordan curve never occupies the same coordinate in space twice. A drawing $\Gamma(e)$ of an edge $e = (u, v)$ is a Jordan curve with one end point translated to $\Gamma(u)$ and the other endpoint translated to $\Gamma(v)$. If the map is also linear, we call the resulting curve a *(straight) line*. For simplicity, we identify an edge e and it's drawing $\Gamma(e)$.

Planarity and beyond. In a drawing, the curves of two edges e and f can cross – that is, we get $\Gamma(e) \cap \Gamma(f) \neq \varnothing$ [6]; if they share exactly one interior point, we call this point a *crossing*. A drawing in the plane without crossing edges is a *planar* drawing and a graph that has some planar drawing is called a *planar* graph; if a graph has no planar drawing, it is nonplanar. A famous result by Fáry [Fár48] states that every planar graph also has a planar straight-line drawing. The notion of planarity can be relaxed as follows: For any constant k, a graph is *k-planar*, if it has a drawing in the plane in which no edge participates in more than k crossings.[7]

Planarity can also be characterized combinatorially: Kuratowski's Theorem [Tut63] states that a graph G is planar, if and only if none of it's subgraphs is a subdivision of the complete graph K_5 or the complete bipartite graph $K_{3,3}$. These graphs are also known as the Kuratowski graphs; they are shown in Figure 2.1 (a) and (b) respectively.

Rotation Systems and Embeddings. The *rotation system* of a graph is defined as the set of cyclic orders of incident edges around each vertex. The edges of a drawing in the plane[8] subdivide that plane into disjoint regions called *faces*. In the plane, there is exactly one unbounded *outer* face and (possibly) some *inner* faces. The boundary of each face is a closed curve composed of a cyclic sequence of edges connecting adjacent vertices and *partial* edges connecting vertices and/or crossings. We say that an edge or vertex

[5] During our research for Chapter 7, we used all of these techniques. We built polyhedral surfaces from paper cut-outs and plastic tiles, then drawing vertices and edges onto them. Several of the figures found in Chapter 7 are screenshots of digital $3D$ models created using our own implementation in JavaScript.

[6] This does not include adjacent edges "crossing" at their shared endpoint.

[7] Naturally, if a graph is k-planar, it is also $(k+1)$ planar.

[8] Or on some other two-dimensional surface.

is *incident* to a face, when it belongs to the closed curve bounding it. Similar to above, two faces are adjacent if their boundaries share an edge. The subdivision created by a drawing of graph G also induces one *embedding*, of G but for a given embedding, there is an infinite number of drawings realizing it.

We can also use a rotation system to describe an embedding. For planar graphs, these definitions are equivalent; for nonplanar graphs, there can be multiple different subdivisions – creating faces bounded by combinatorially different curves – following the same rotation system. We will sometimes prescribe the combinatorial embedding of a graph to then try finding a drawing matching the given embedding.

2.2 Algorithms

In this section, we define the algorithmic concepts relevant to this book. An *algorithm* is a finite sequence of well-defined instructions, designed to be executed by some machine (such as Turing machines, computers, or humans). An algorithm is *deterministic*, if when given an input, it will always perform the same sequence of steps to derive at the same output. A *non-deterministic* algorithm has some "freedom of choice" when performing the next instruction – some choices might lead to different outputs than others. A *randomized* algorithm employs some random number generator – such as flipping coins or creating a random order of the elements of a set. The computational model used throughout this book is the *real RAM model* – a hypothetical machine capable of infinite-precision arithmetic operations on real numbers in constant time. Details on this model can be found in the book *Computational Geometry* by Michael Shamos [Sha78].

2.2.1 Asymptotic Runtime

When looking at families of graphs or the running time of algorithms, one can often recognize some *asymptotic* behaviour. We describe such behaviours using *Landau symbols* – also known as *Big Oh Notation*. Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function that maps a value some asymptotic behaviour – e.G. input size and running time for an algorithm or number of vertices and edges for some graph. We define the following classes of functions with respect to f:

$$O(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \exists c > 0,\ \exists n_0 > 0,\ \forall n > n_0 \colon 0 < f(n) \leq c \cdot g(n)\}$$
$$\Omega(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \exists c > 0,\ \exists n_0 > 0,\ \forall n > n_0 \colon c \cdot g(n) \leq f(n)\}$$
$$\Theta(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \exists c_1, c_2 > 0,\ \exists n_0 > 0,\ \forall n > n_0 \colon c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

The growth of a function g that *lies in order of* f – or for short $g \in O(f)$ – is upper-bounded by that of f. Symmetrically, we have that for $g \in \Omega(f)$, the growth of g is lower-bounded by that of f. Intuitively, we have that $g \in \Theta(f)$ implies that f and g grow alike – that is, $g \in O(f)$ and $g \in \Omega(f)$, thus f upper- and lower-bounds g for

different constants. If we have $f \in O(n^k)$ for some constant k, we say that f is a *polynomial* function; an algorithm with runtime $O(n^k)$ is an *efficient* or *linear-time* algorithm. Opposed to efficient algorithms are the *exponential-time* algorithms. An exponential algorithm has a runtime that is lower-bounded by some exponential function (a function $f(n) = c^n$ in which the parameter appears in the exponent at least once and with some base $c > 1$).

2.2.2 Complexity and Hardness

Complexity Classes. There are two different kinds of problems that we discuss in this book. For *decision* problems, we try to find a yes/no-answer; whereas for *optimization* problems, we look for the best answer – optimizing some *objective* function.

The complexity class $\mathcal{P}$ is the class of all decision problems that are solvable by some efficient deterministic algorithm. On the other hand, the class $\mathcal{NP}$ is the class of problems solvable by non-deterministic polynomial-time algorithms. We clearly have that $\mathcal{P} \subset \mathcal{NP}$, but the question whether the two classes are the same or not – $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \subsetneq \mathcal{NP}$ – remains open. From an algorithm standpoint, most results are stated under the assumption that $\mathcal{P} \neq \mathcal{NP}$[9].

A problem T is *$\mathcal{NP}$-hard*, when there is a polynomial-time reduction from every problem $R \in \mathcal{NP}$ to T. A *reduction* from R to T is a transformation that translates an instance I_R for problem R into an instance I_P for problem T such that there is a valid solution to I_R if and only if there is one to I_T. A problem is *$\mathcal{NP}$-complete* if it is $\mathcal{NP}$-hard and in $\mathcal{NP}$.

Boolean Satisfiability. Finally, we define one of the classic problems shown to be $\mathcal{NP}$-complete by Richard Karp [Kar72] in 1972 – namely *Satisfiability with at most 3 literals per clause*, also known as 3-SAT. Cook [Coo71] considered the *Boolean Satisfiability Problem* (or SAT), showing that it is the first $\mathcal{NP}$-complete problem. An instance of SAT – a Boolean *formula* – consists of Boolean *variables* and the three operators And, Or, and Not (denoted by the symbols $\wedge$, $\vee$, and $\neg$ respectively). A formula is called *satisfiable* if it is true for some assignment of Boolean values to the variables. SAT is the problem of deciding whether a given formula is satisfiable.

The variables appear as negated or unnegated *literals* – such as x or $\neg x$. Each clause is a disjunction of literals – for instance $(x \vee \neg y \vee z)$ – and a formula is in *conjunctive normal form*, if it is a conjunction of clauses. Any SAT formula can be translated to an equivalent formula in conjunctive normal form. The issue with this is that the translated formula's size can be exponentially larger than that of the original one.

The 3SAT problem is a special version of SAT; a 3SAT formula is in conjunctive normal form and in addition, every clause contains no more than three literals. While 3SAT is $\mathcal{NP}$-complete, the variant 2SAT – restricting the clauses to contain up to two literals each – can be solved efficiently.

[9] This question has been open for about 60 years. In 2012, William Gasarch [Gas12] conducted a survey among computer scientists. Out of 152 participants, 83% expected the two classes to not be equal.

Part I

Drawing Vertices on a Common Outer Face

Chapter 3

Outer k-Planar and Outer k-Quasi-Planar Graphs

In the last few years, the focus in graph drawing has shifted from exploiting structural properties of planar graphs to addressing the question of how to produce well-structured and understandable drawings.

This becomes even more important in the presence of edge crossings, giving rise to the topic of *beyond-planar* graph classes. The primary approach here has been to define and study graph classes which allow some edge crossings, but restrict the crossings in various ways. Two commonly studied such graph classes are:

1. *k-planar graphs*, the graphs which can be drawn so that each edge (Jordan curve) is crossed by at most k other edges.

2. *k-quasi-planar graphs*, the graphs which can be drawn so that no k pairwise non-incident edges mutually cross.

Following these definitions, the 0-planar graphs and 2-quasi-planar graphs are precisely the planar graphs. Additionally, the 3-quasi-planar graphs are simply called *quasi-planar*.

In this chapter we study these two families of classes of graphs under the restriction that the points are placed in convex position and edges mapped to line segments; i.e., we apply the above two generalizations of planar graphs to outerplanar graphs and study *outer k-planarity* and *outer k-quasi-planarity*. In the following, we consider balanced separators, treewidth, degeneracy (see paragraph "Concepts" below), coloring, edge density, and recognition to study these two classes.

Concepts. We briefly define the most important graph theoretic concepts that we will study in this chapter.

A graph is *d-degenerate* when every subgraph of it has a vertex of degree at most d. This concept was introduced as a way to provide easy bounds on the chromatic number [LW70]. Namely, a d-degenerate graph can be inductively $d + 1$ colored by simply removing a vertex of degree at most d. A graph class is d-degenerate when every graph in the class is d-degenerate. Furthermore, a graph class which is *hereditary* (i.e., closed under taking subgraphs) is d-degenerate when every graph in that class has a vertex of degree at most d. Note that outerplanar graphs are 2-degenerate, and planar graphs are 5-degenerate.

Given a graph $G = (V, E)$ with n vertices, a pair A, B of subsets of V is a *separation* of G when $A \cup B = V$, and no edge of G has one end point in $A \setminus B$ and the other in $B \setminus A$. The intersection $A \cap B$ is called a *separator* and the *size* of the separation (A, B) is $|A \cap B|$. A separation (A, B) of G is *balanced* if $|A \setminus B| \leq \frac{2n}{3}$ and $|B \setminus A| \leq \frac{2n}{3}$. The *separation number* of G is the smallest number s such that every subgraph of G has a balanced separation of size at most s. There is a polynomial relation between separation number and *treewidth* introduced by Robertson and Seymour [RS84]; Namely, any graph with treewidth t has separation number at most $t + 1$ and, as Dvořák and Norin [DN14] recently showed, any graph with separation number s has treewidth at most $105s$. Graphs with bounded treewidth are well-known due to Courcelle's Theorem (see Theorem 3.12) [Cou90], i.e., having bounded treewidth means many problems can be solved efficiently.

A *quasi-polynomial time* algorithm is one with a running time of the form $2^{\text{poly}(\log n)}$. The *Exponential Time Hypothesis (or ETH for short)* [IP01] is a complexity theoretic assumption defined as follows. For $k \geq 3$, let $s_k = \inf\{\delta : \text{there is an } O(2^{\delta n})\text{-time algorithm to solve } k\text{-SAT}\}$. The Exponential Time Hypothesis states for $k \geq 3$ that $s_k > 0$; in other words, there is no quasi-polynomial time algorithm that solves 3-SAT. So, finding a problem that can be solved in quasi-polynomial time and is also $\mathcal{NP}$-hard, would contradict the ETH. In recent years, the ETH has become a standard assumption from which many conditional lower bounds have been proven [CFK+15]. Note that, in addition to violating the ETH, the existence of an $\mathcal{NP}$-hard problem which can be solved in quasi-polynomial time would also directly imply that *deterministic* and *nondeterministic exponential time – (NEXP)* and *(EXP)* – coincide. This can be proven by a padding argument similar to Proposition 2 [BH92]. Thus, having such an algorithm for a problem implies that it is extremely unlikely for that problem to be $\mathcal{NP}$-hard.

Contribution. The rest of this chapter is structured as follows.

In Section 3.2, we consider outer k-planar graphs. We show that this graph class is $(\lfloor\sqrt{4k+1}\rfloor + 1)$-degenerate, and observe that the largest outer k-planar clique has size $(\lfloor\sqrt{4k+1}\rfloor + 2)$, i.e., implying each outer k-planar graph can be $(\lfloor\sqrt{4k+1}\rfloor + 2)$-colored and this is tight. We further show that every outer k-planar graph has separation number at most $2k + 3$. For each fixed k, we use these balanced separators to obtain a quasi-polynomial time algorithm to test outer k-planarity, i.e., these recognition problems are not $\mathcal{NP}$-hard unless ETH fails.

In Section 3.3, we consider outer k-quasi-planar graphs. Specifically, we discuss the edge-maximal graphs which have been considered previously under different names [CP92, DKM02, Nak00].We provide a novel approach to show that all edge-maximal outer k-quasi-planar graphs have the maximum number of edges, namely $2(k-1)n - \binom{2k-1}{2}$. We also relate outer k-quasi-planar graphs to planar graphs.

In Section 3.4, we restrict outer k-planar and outer k-quasi-planar drawings to *full* drawings (where no crossing appears on the boundary), and to *closed* drawings (where the vertex sequence on the boundary is a cycle in the graph). The case of full outer 2-planar graphs has been considered by Hong and Nagamochi [HN16]. They showed

that full outer 2-planarity testing can be performed in linear time. We first observe that a graph is full outer k-planar if and only if its maximal biconnected components are closed outer k-planar[1], and that this equivalence also holds for full outer k-quasi-planar graphs. Then, for each k, we express both *closed outer k-planarity* and *closed outer k-quasi-planarity* in *extended monadic second-order logic*. Thus, since outer k-planar graphs have bounded treewidth, full outer k-planarity is testable in $O(f(k) \cdot n)$ time, for a computable function f. This result greatly generalizes the work of Hong and Nagamochi [HN16].

3.1 Related work

Ringel [Rin65] was the first to consider k-planar graphs by showing that 1-planar graphs are 7-colorable. This was later improved to 6-colorable by Borodin [Bor84]. This is tight since K_6 is 1-planar. Many additional results on 1-planarity can be found in a recent survey paper [KLM17]. Generally, each n-vertex k-planar graph has at most $4.108n\sqrt{k}$ edges [PT97] and treewidth $O(\sqrt{kn})$ [DEW17].

Outer k-planar graphs have been considered mostly for $k \in \{0, 1, 2\}$. Of course, the outer 0-planar graphs are the classic outerplanar graphs which are well-known to be 2-degenerate and have treewidth at most 2. It was shown that essentially every graph property is testable on outerplanar graphs [BKN16]. Outer 1-planar graphs are a simple subclass of planar graphs and can be recognized in linear time [ABB$^+$16, HEK$^+$15]. *Full outer 2-planar graphs* – a subclass of outer 2-planar graphs – can been recognized in linear time [HN16]. General outer k-planar graphs were considered by Binucci et al. [BGHL18]. Among other results, they showed that, for every k, there is a 2-tree which is not outer k-planar. Wood and Telle [WT07] considered a slight generalization of outer k-planar graphs in their work and showed that these graphs have treewidth $O(k)$.

The k-quasi-planar graphs have been heavily studied from the perspective of edge density. The goal here is to settle a conjecture of Pach et al. [PSS96] stating that every n-vertex k-quasi-planar graph has at most $c_k n$ edges, where c_k is a constant depending only on k. This conjecture has been shown for $k = 3$ [AT07] and $k = 4$ [Ack09]. The best known upper bound is $(n \log n)2^{\alpha(n)^{c_k}}$ [FPS13], where α is the inverse of the Ackermann function. Capoyleas and Pach [CP92] showed that any k-quasi-planar graph has at most $2(k-1)n - \binom{2k-1}{2}$ edges, and that there are k-quasi-planar graphs meeting this bound. More recently, it was shown that the *semi-bar k-visibility graphs* are outer $(k+2)$-quasi-planar [GKT14]. However, the outer k-quasi-planar graph classes do not seem to have received much further attention.

The relationship between k-planar graphs and k-quasi-planar graphs was considered recently. While any k-planar graph is clearly $(k+2)$-quasi-planar, Angelini et al. [ABB$^+$17] showed that any k-planar graph is $(k+1)$-quasi-planar. More specially, it has also been shown that 2-planar graphs are also quasiplanar [HT17].

[1] This was observed for full outer 2-planar graphs by Hong and Nagamochi [HN16].

The *convex* (or *1-page book*) *crossing number* of a graph [Sch14] is the minimum number of crossings which occur in any convex drawing. This concept has been introduced several times (see [Sch14] for more details). The convex crossing number is $\mathcal{NP}$-complete to compute [MKNF87]. However, recently Bannister and Eppstein [BE14] used treewidth-based techniques (via extended monadic second order logic) to show that it can be computed in linear FPT time, i.e., $O(f(c)\cdot n)$ time where c is the convex crossing number and f is a computable function. Thus, for any k, the *outer k-crossing graphs* with n vertices and m edges can be recognized in time linear in $n+m$.

3.2 Outer k-Planar Graphs

In this section we show that every outer k-planar graph is $O(\sqrt{k})$-degenerate and has separation number $O(k)$. This provides tight bounds on the chromatic number, and allows for testing outer k-planarity in quasi-polynomial time.

3.2.1 Degeneracy and Coloring

We show that every outer k-planar graph has a vertex of degree at most $\sqrt{4k+1}+1$. First we note the size of the largest outer k-planar clique and then we prove that each outer k-planar graph has a vertex matching the clique's degree. This also tightly bounds the chromatic number in terms of k, i.e., Theorem 3.3 follows from Lemma 3.1 and Lemma 3.2.

Lemma 3.1. *Every outer k-planar clique has at most $\lfloor\sqrt{4k+1}\rfloor + 2$ vertices.*

Proof. We want to derive the size n of the largest outer k-planar clique. Let a, b be two vertices of that clique; the edge ab partitions the other vertices into two sets S_ℓ and S_r – the vertices on the left and the right side of ab. Because all vertices are a clique, ab is crossed $|S_\ell|\cdot|S_r|$ times. Therefore, the edge that is crossed the most has an almost equal number of vertices on both sides; the total number of crossings then is $\left(\frac{n-2}{2}\right)^2 \leq k$ if n is even and $\frac{(n-3)(n-1)}{4} \leq k$ if n is odd. Therefore, for fixed k, the size of the clique is at most $\lfloor\sqrt{4k+1}\rfloor + 2$. □

Lemma 3.2. *An outer k-planar graph can have maximum minimum degree at most $\sqrt{4k+1}+1$ and this bound is tight.*

Proof. We want to determine the largest possible minimum degree δ of an outer k-planar graph G. By Lemma 3.1, G can contain a clique of size $\lfloor\sqrt{4k+1}\rfloor + 2$, so we know that $\delta \geq \lfloor\sqrt{4k+1}\rfloor + 1$ holds. We show that δ also cannot be larger than $\sqrt{4k+1}+1$. We show by induction that any outer k-planar graph with minimum degree $\delta \geq \sqrt{4k+1}+2$ would be too small to accommodate such a minimum degree vertex.

We say an edge ab *cuts* $h \in \mathbb{N}$ vertices, if there are h vertices to either the left or the right side of ab with respect to the embedding of G. Consider an edge ab that cuts

h vertices of the graph. Denote the number of other edges crossing ab by $\mathrm{cr}(ab)$ By assumption, for any edge ab that cuts h vertices we have $\mathrm{cr}(ab) \leq \delta h - h(h+1)$. If $\delta \geq \sqrt{4k+1}+2$, there is an $h^* \geq \frac{1}{2}(\delta - 1 - \sqrt{(\delta-1)^2 - 4(k+1)})$ such that ab cannot cut h^* vertices – it would have $\mathrm{cr}(ab) \geq \delta h^* - h^*(h^*+1) \geq k+1$. Consider the smallest such h^*. We show that there also cannot be an edge ab that cuts more than h^* vertices. For the induction , assume that no edge ab cuts between h^* and h vertices inclusive. Thus, for edge ab we get $\mathrm{cr}(ab) \leq \delta h - h(h+1) + 2(\sum_{j=1}^{h-h^*} j) > k$, where the last term accounts for the absent edges that cut more than $h - h^*$ vertices. Now, if ab cuts $h+1$ vertices and we have $\delta > 2h^*$, we get:

$$\begin{aligned}\mathrm{cr}(ab) &\geq \delta h - h(h+1) + 2(\textstyle\sum_{j=1}^{h-h^*} j) + \delta - 2(h+1) + 2(h - h^* + 1) \\ &> k + \delta - 2(h+1) + 2(h - h^* + 1) > k\end{aligned}$$

The inequality is trivially satisfied for $\delta > \sqrt{4k+1} + 2$, hence there cannot be an edge that cuts more then $l^* < \sqrt{4k+1}/2$ vertices in any outer k-planar graph with the maximum minimum degree $\delta \geq \sqrt{4k+1} + 2$. But then, such a graph can have at most $2l^* < \sqrt{4k+1}$ vertices, which is not enough to accommodate the minimum degree vertex required; a contradiction. □

Having established that outer k-planar graphs are $(\sqrt{4k+1}+1)$-degenerate, we easily obtain the following result.

Theorem 3.3. *Each outer k-planar graph is $\sqrt{4k+1}+2$ colorable and this is tight.*

3.2.2 Quasi-polynomial time recognition via Balanced Separators

We show that outer k-planar graphs have separation number at most $2k+3$ (Theorem 3.4). Via a result of Dvořák and Norin [DN14], this implies that these graphs have treewidth linear in k. However, Proposition 8.5 of [WT07] implies that every outer k-planar graph has treewidth at most $3k + 11$, i.e., a better bound on the treewidth than applying the result of Dvořák and Norin to our separators. The treewidth $3k + 11$ bound also implies a separation number of $3k + 12$, but our bound is lower. Our separators also allow outer k-planarity testing in quasi-polynomial time – see Theorem 3.5.

Theorem 3.4. *Each outer k-planar graph has separation number at most $2k + 3$.*

Proof. Consider an outer k-planar drawing. If the graph has an edge that cuts $[\frac{n}{3}, \frac{2n}{3}]$ vertices to one side, we can use this edge to obtain a balanced separator of size at most $k+2$, i.e., by choosing the endpoints of this edge and a vertex cover of the edges crossing it. So, suppose no such edge exists. Consider a pair of vertices (a, b) such that the line $\overline{ab}$ divides the drawing into a left side S_ℓ and a right side S_r having an almost equal number of vertices ($\big||S_\ell| - |S_r|\big| \leq 1$). If the edges which cross $\overline{ab}$ also mutually cross each other,

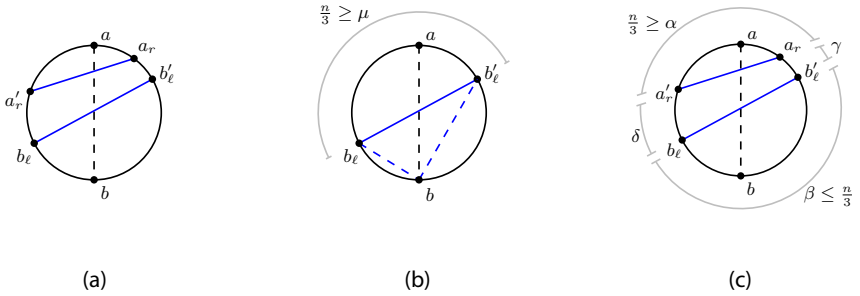


Figure 3.1: Illustrations for finding a balanced separator as in the proof of Theorem 3.4: (a) the pair of parallel edges $b_\ell b'_\ell$ and $a_r a'_r$; (b) case 1 in the proof; (c) case 2.

there can be at most k of them. Thus, we again have a balanced separator of size at most $k+2$. So, it remains to consider the case when we have a pair of edges that cross $\overline{ab}$, but do not cross each other. We call such a pair of edges *parallel*. We now pick a specific pair of parallel edges: starting from b, let b_ℓ be the first vertex along the boundary in clockwise direction such that there is an edge $b_\ell b'_\ell$ that crosses the line ab. Symmetrically, starting from a, let a_r be the first vertex along the boundary in clockwise direction such that there is an edge $a_r a'_r$ that crosses the line ab; see Figure 3.1 (a). Note that the edges $a_r a'_r$ and $b_\ell b'_\ell$ are either identical or parallel. In the former case, we see that all other edges crossing line $\overline{ab}$ must also cross the edge $a_r a'_r = b_\ell b'_\ell$, and as such there are again at most k edges crossing $\overline{ab}$. In the latter case, there are two subcases to be considered. For two vertices u and v, let $[u, v]$ be the set of vertices that starts with u and, going clockwise, ends with v. Let $(u, v) = [u, v] \setminus \{u, v\}$.

- **Case 1:** The edge $b_\ell b'_\ell$ cuts $\frac{n}{3}$ vertices to the top; see Figure 3.1 (b). In this case, either $[b'_\ell, b]$ or $[b, b_\ell]$ has $[\frac{n}{3}, \frac{n}{2}]$ vertices. We claim that neither the line $\overline{bb_\ell}$ nor the line $\overline{bb'_\ell}$ can be crossed more than k times. Namely, each edge that crosses $\overline{bb_l}$ also crosses $b_\ell b'_\ell$. Similarly, each edge that crosses $\overline{bb'_\ell}$ also crosses the edge $b_\ell b'_\ell$. Thus, we have a separator of size at most $k + 2$, regardless of whether we choose $\overline{bb_\ell}$ or $\overline{bb'_\ell}$ to separate the graph. As we observed above, one of them is balanced. Notice that the case where edge $a_r a'_r$ cuts at most $\frac{n}{3}$ vertices to the bottom is symmetric.
- **Case 2:** The edge $b_\ell b'_\ell$ cuts at most $\frac{n}{3}$ vertices to the bottom, and the edge $a_r a'_r$ cuts at most $\frac{n}{3}$ vertices to the top; see Figure 3.1 (c). We show that we can always find a *close* pair of parallel edges, i.e. one edge of the pair cuts at most $\frac{n}{3}$ vertices to the bottom and the other edge cuts at most $\frac{n}{3}$ vertices to the top, and with no edge between them parallel to either. If there is an edge e between $b_\ell b'_\ell$ and $a_r a'_r$, we

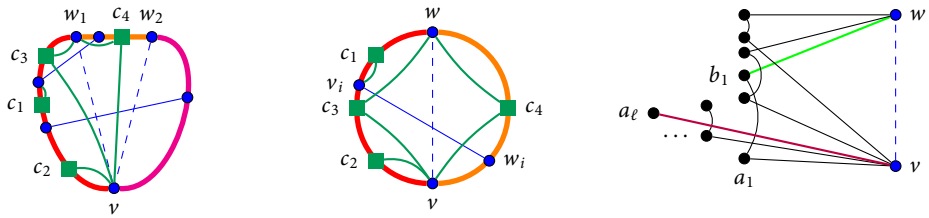


Figure 3.2: Shapes of separators, special separator S in blue, regions in different colors (red, orange, and pink), components connected to blue vertices in green: (a) closest-parallels case; (b) single-edge case; (c) special case for single-edge separators.

form a new pair by using e to replace either $a_r a'_r$ or $b_\ell b'_\ell$, depending on whether e cuts at most $\frac{n}{3}$ vertices to the bottom or to the top respectively. By repeating this procedure, we always find a close pair. Hence, we can assume that $b_\ell b'_\ell$ and $a_r a'_r$ actually form a close pair. Let $\alpha = |(a'_r, a_r)|$, $\beta = |(b'_\ell, b_\ell)|$, $\gamma = |(a_r, b'_\ell)|$, and $\delta = |(b_\ell, a'_r)|$ be the numbers vertices in the four quadrants defined by the endpoints; see Figure 3.1 (c).

Suppose that $a'_r = b_\ell$ or $a_r = b'_\ell$. We can now use both edges $b_\ell b'_\ell$ and $a_r a'_r$ (together with any edges crossing them) to obtain a separator of size at most $2k + 3$. The separator is balanced since $\alpha + \beta \leq \frac{2n}{3}$ and $\gamma + \delta \leq \frac{2n}{3}$.

So, now $a_r, a'_r, b_\ell, b'_\ell$ are all distinct. Note that $\gamma, \delta \leq \frac{n}{2}$ since each side of $\overline{ab}$ has at most $\frac{n}{2}$ vertices. We separate the graph along the line $\overline{b_\ell a_r}$. Namely, all the edges that cross this line must also cross $b_\ell b'_\ell$ or $a'_r a_r$. Therefore, we obtain a separator of size at most $2k + 2$.

To see that the separator is balanced, we consider two cases. If $\delta \geq \frac{n}{3}$ (or $\gamma \geq \frac{n}{3}$), then $\alpha + \beta + \gamma \leq \frac{2n}{3}$ (or $\alpha + \beta + \delta \leq \frac{2n}{3}$). Otherwise we have $\delta < \frac{n}{3}$ and $\gamma < \frac{n}{3}$. In this case $\delta + \alpha \leq \frac{2n}{3}$ and $\gamma + \beta \leq \frac{2n}{3}$. In both cases the separator is balanced.

□

Theorem 3.5. *Outer k-planarity of an n-vertex graph can be tested in $O(2^{\text{polylog } n})$ when k is fixed.*

Proof. Our approach is to leverage the structure of the balanced separators as described in the proof of Theorem 3.4. Namely, we enumerate the sets which could correspond to such a separator, pick an appropriate outer k-planar drawing of these vertices and their

edges, partition the components arising from this separator into *regions*, and recursively test the outer k-planarity of the regions.

To obtain quasi-polynomial runtime, we need to limit the number of components on which we branch. We do this by grouping them into regions defined by special edges of the separators.

By the proof of Theorem 3.4, if our input graph has an outer k-planar drawing, there must be a separator which has one of the two shapes depicted in Figure 3.2 (a) and (b). Here we are not only interested in the up to $2k+3$ vertices of the balanced separator, but actually the set S of up to $4k+3$ vertices one obtains by taking both endpoints of the edges used to find the separator. Note that this larger set S is also a balanced separator. We use a brute force approach to find the right set: Enumerating all sets of vertices of size up to $4k+3$, we then check whether it can be drawn similar to one of the two shapes from Figure 3.2. By fixing S we pick a subgraph G_S induced by S with $O(k)$ vertices. G_S can have at most a function of k different outer k-planar drawings. Thus, we branch on all such drawings of G_S.

We now consider the two different shapes separately. In the first case, S contains three special vertices v, w_1 and w_2; in the second case S contains two special vertices v and w. In both cases, the special vertices will be called *boundary* vertices and all other vertices in S will be called *regional* vertices. By fixing the drawing of G_S in the current branch, the regional vertices are partitioned into regions by the boundary vertices. Using structure of the separator guaranteed by the proof of Theorem 3.4, we get that no component of $G \setminus S$ can be adjacent to regional vertices which live in different regions with respect to the boundary vertices.

We first discuss the case of using G_S as depicted in Figure 3.2 (a). We start by picking the three special vertices v, w_1 and w_2 from S to behave as shown in Figure 3.2 (a). The following arguments regarding this shape of separator are symmetric with respect to the pair of opposing regions.

If there is a component connected to regional vertices of different regions, we reject this configuration. From the proof of Theorem 3.4, no component can be adjacent to all three boundary vertices: this would either contradict the closeness of the parallel edges or imply an edge connecting distinct regions. Each component is of one of four different types, depending on how it is connected to regional and/or boundary vertices; for the regions neighboring w_1 they are shown as $c_1, \ldots, c_4$ in Figure 3.2 (a).

- Components of type c_1 are connected to (possibly many) regional vertices of the same region and may be connected to boundary vertices as well. In any valid drawing, they have to be placed in the same region as their regional vertices.

- Components of type c_2 are not connected to any regional vertices and only connected to one of the three boundary vertices. Hence, they cannot interfere with other parts of the drawing – we can arbitrarily assign them to a region adjacent to their boundary vertex.

- For components that are connected to two boundary vertices – say v and w_1 – it seems to be possible for them to be placed left or right of the edge connecting v and w_1, e.g., as c_3 or c_4. The latter option c_4 is not valid: The separator was created by two close parallel edges as argued in the proof of Theorem 3.4, a contradiction

From the above discussion, we see that from a fixed configuration – a set S, a drawing of G_S, and triple of boundary vertices – if the drawing of G_S has the shape depicted in Figure 3.2 (a), we can either reject the current configuration for having bad components, or we obtain a well-defined placement into the regions defined by the boundary vertices: For components of type c_2, it suffices to recursively produce a drawing of that component together with its boundary vertex to be placed next to that boundary vertex. The other components can be partitioned into the regions and we recurse on the regions individually. This covers all cases for this separator shape.

The shape of the separator for the second case is shown in Figure 3.2 (b). We have two boundary vertices v and w and thus only two regions. This allows for the two component types c_1 and c_2 from the first case. They are also handled as described above. In addition, we have components connected to both v and w but no regional vertices. This allows for two different placement options c_3, c_4 – left or right of the line $\overline{vw}$. If there is an edge $v_i w_i$ that is part of the separator but different from vw, there cannot be more than a total of k such components; see Figure 3.2 (b). In any drawing, there will be edges connecting each component to v and w, and at least one of these edges has to cross $v_i w_i$. Since this edge can be crossed at most k times, this gives an upper bound on the number c_3 and c_4-type components. Thus, we now enumerate all different placements of these components as type c_3 or c_4 and recurse accordingly.

In the case that there is no edge $v_i w_i$, the separator is exactly the pair (v, w). This eliminates the possibility of type c_1 components and the components of type c_2 are handled as before. We argue that in this case, any a valid drawing can have most a function of k different components of type c_3 or c_4. Consider the components of type c_3, the components of type c_4 can be counted similarly. Given a valid drawing of a type c_3 component, consider the highest – clockwise last – vertex of this drawing connected to v and the lowest – clockwise fist – vertex connected to w. These two vertces define a subinterval of the left region. Considering two such intervals, they can relate in one of three ways: They overlap, they are disjoint, or one is contained in the other. We group components with either overlapping or disjoint intervals into *layers*. This is shown in Figure 3.2 (c): For simplicity, we only draw the highest and lowest connected vertices for every component and we contract every component into a single edge representing connectivity.

Let $a_1 b_1$ be the bottommost component of type c_3 – vertex a_1 is the lowest of all lower vertices. We define the first layer to be all components overlapping or disjoint of $a_1 b_1$. Now consider the (green) edge $b_1 w$ (see Figure 3.2 (c)): The total number of components disjoint from $a_1 b_1$ in the first layer is bounded by k since for every component, at least one of it's edges connecting it to v must cross $b_1 w$. Invervals that overlap $a_1 b_1$ must have an edge connecting the vertex inside $a_1 b_1$ to either v oder w. This edge then must cross

either a_1v or b_1w. This concludes that there can only be $O(k)$ components in the first layer.

New layers are defined by considering components whose intervals are fully contained in the first interval of the previous layer, starting with the second layer completely being contained inside a_1b_1. Notice that intervals fully contained in other intervals disjoint from a_1b_1 are also in the first layer and accounted for accordingly. All deeper layers are nested inside a_1b_2. To limit the total depth, let a_ℓ be the lowest vertex of the first component of the deepest layer. Consider the (purple) edge va_ℓ – it is crossed by some edge of every layer above it. As any edge can only have k crossings, there can only be $O(k)$ different levels in total. This leaves us with a total of at most $O(k^2)$ components per region and again we can enumerate their placements and recurse accordingly.

The above algorithm provides the following recurrence regarding its runtime. Denote by $T(n)$ the runtime of our algorithm which is generously upper bounded by the following expression. Let $f(s)$ be the number of different outer k-planar drawings of a graph with s vertices.

$$T(n) \leq \begin{cases} n^{O(k)} \cdot f(4k+3) \cdot n^3 \cdot n \cdot T(\frac{2n}{3}) & \text{for } n > 5k \\ f(n) & \text{otherwise} \end{cases}$$

Thus, the algorithm runs in quasi-polynomial time, i.e., $2^{\text{poly}(\log n)}$. □

3.3 Outer k-Quasi-Planar Graphs

In this section we consider outer k-quasi-planar graphs. We first describe some classes of graphs which are outer 3-quasi-planar. We then discuss edge-maximal outer k-quasi-planar drawings.

3.3.1 Comparability to Planar Graphs

Note that all sub-Hamiltonian planar graphs are outer 3-quasi-planar. One can also see which complete and bipartite complete graphs are outer 3-quasi-planar.

Proposition 3.1. *The following graphs are outer 3-quasi-planar:*

(a) $K_{4,4}$ *and* K_5*,*

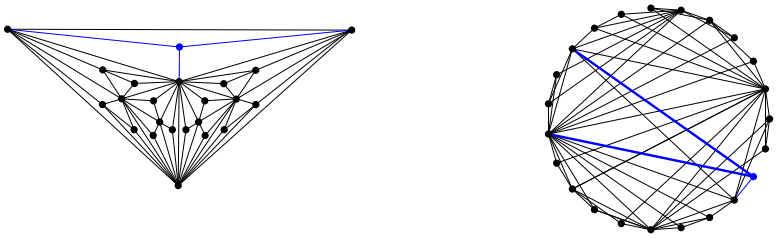
(b) planar 3*-tree with three complete levels, and*

(c) square-grids of any size.

Proof. It is easy to verify (a) by constructing a valid drawing.

(b) was experimentally verified by using MiniSat [SE05] to check a boolean expression for satisfiability. Details can be found in Section 3.6.1.

(c) follows from an old result: square-grids are sub-Hamiltonian [CLR87]. □



(a) (b)

Figure 3.3: A vertex-minimal 23-vertex planar 3-tree which is not outer quasi-planar: (a) planar drawing; (b) deleting the blue vertex makes the drawing outer quasi-planar

In addition, we state that the following complete and complete bipartite graphs which are not outer-quasi planar.

Proposition 3.2. *The following graphs are not outer 3-quasi-planar:*

(a) complete bipartite graphs $K_{p,q}$ with $p \geq 3$, $q \geq 5$,

(b) complete graphs K_n with $n \geq 6$, and

(c) planar 3-tree with four complete levels.

Furthermore, not all planar graphs have an outer quasi-planar drawing. Consider the vertex-minimal planar 3-tree on 23 vertices shown in Figure 3.3 (a); using MiniSat to check for satisfiability of the corresponding boolean expression we verified that it is not outer quasi-planar. An almost outer-quasi planar drawing of this graph can be seen in Figure 3.3 (b). It was constructed by removing the blue vertex, drawing the remaining graph in an outer quasi-planar way, and then reinserting the missing vertex.

Together, Propositions 3.1 and 3.2 immediately yield the following.

Theorem 3.6. *Planar graphs and outer 3-quasi-planar graphs are incomparable under containment.*

Remark For outer k-quasi-planar graphs ($k > 3$) containment questions become more intricate. Every planar graph is outer 5-quasi-planar because planar graphs have page number 4 [Yan89]. We also know a planar graph that is not outer 3-quasi-planar. It is open whether every planar graph is outer 4-quasi-planar.

3.3.2 Maximal Outer k-Quasi-Planar Graphs

If adding any edge to a drawing of an outer k-quasi-planar graph destroys the outer k-quasi-planarity of that drawing, it is called *maximal*. We call an outer k-quasi-planar graph maximal if it has a maximal outer k-quasi-planar drawing. Recall that Capoyleas and Pach [CP92] showed the following upper bound on the edge density of outer k-quasi-planar graphs on n vertices:

$$|E| \leq 2(k-1)n - \binom{2k-1}{2}$$

We prove that each maximal outer k-quasi-planar graph meets this bound. While two other proofs of this result can be found in the literature [DKM02, Nak00] – we thank David Wood for pointing us to them – the main result of both papers prove a slightly stronger theorem: For a drawing $G = (V, E)$, an *edge flip* produces a new drawing G^* by replacing an edge $e \in E$ with a new edge $e^* \in \binom{n}{2} \setminus E$. They [DKM02, Nak00] show that, for every two maximal outer k-quasi-planar drawings $G = (V, E)$ and $G' = (V, E')$, there is a sequence of edge flips producing drawings $G = G_1, G_2, \ldots, G_t = G'$ such that each G_i is a maximal k-quasi-planar drawing. Together with the tight example of Capoyleas and Pach [CP92], this implies the next theorem. The proof we present here directly employes an inductive argument, building on the ideas of Capoyleas and Pach [CP92]. The argument itself and the structural insights leading up to it are of independent interest.

Theorem 3.7 ([DKM02, Nak00])**.** *Each maximal outer k-quasi-planar drawing $G = (V, E)$ has:*

$$|E| = \begin{cases} \binom{|V|}{2} & \text{if } |V| \leq 2k-1, \\ 2(k-1)|V| - \binom{2k-1}{2} & \text{if } |V| \geq 2k-1. \end{cases}$$

For an outer k-quasi-planar drawing of graph G we call an edge $\overline{ab}$ a *long edge*, if a and b are separated along the outer face of G by at least $(k-1)$ vertices on both sides. In the following depictions, long edges will always be drawn vertically with a on top, dividing the graph into the two regions left and right of $\overline{ab}$. All edges that intersect the long edge are called *crossing edges*. All vertices incident to crossing edges will be called *crossing vertices* and for illustration we will label them as follows: In the left region, vertices will be labeled $v_1, \ldots, v_g$ couterclockwise from a on – from top to bottom –, and in the right region, vertices will be labeled $w_1, \ldots, w_h$, and by definition, we have $g, h \geq k-1$; see Figure 3.4 (a).

To proof maximality of the considered graph, we count the number of crossing edges in an inductive argument. We construct $(k-2)$ *hierarchical levels* – subsets of the crossing edges of G that form maximal crossing-free connected subgraphs of G. We then define a replacement-operation that uses these levels to split the original graph into two subgraphs, each with viewer vertices. We use Algorithm 3.1 to greedily build the hierarchical levels.

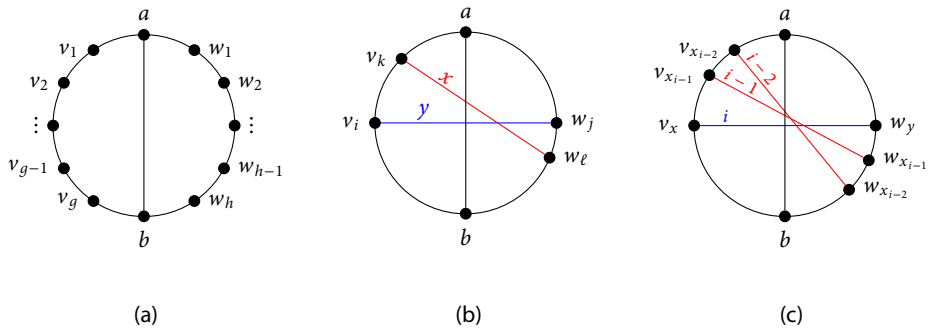


Figure 3.4: Long edges in outer k-quasi-planar graphs: (a) Labeling scheme for left and right side vertices according to the long edge $\overline{ab}$. (b) Illustration of property (P1): $v_k w_\ell$ is crossing $v_i w_j$ from above. (c) Property (P2): blue edge in level i and the first two edges of a possible certificate.

Algorithm 3.1: BUILDLEVELS(Outer k-quasi-planar Graph G, long edge $\overline{ab}$)

```
Vertices {v_1, ..., v_g} ← vertices left of ab
Set of Sets S = {S_1, ..., S_{k-2}} ← empty level sets
for i = 1 to k − 2 do
    for j = 1 to g do
        S' ← edges of v_j not crossing edges in S_i
        S_i ← S_i ∪ S'
return S
```

Lemma 3.8. *For a given outer k-quasi-planar graph G, Algorithm 3.1 generates $k-2$ hierarchical levels.*

Proof. We need to argue two things: The algorithm creates $k-2$ levels that cover all crossing edges with respect to $\overline{ab}$, and that every level is connected. We order the levels by order of construction: level y is after level x or $x \prec y$, if y is constructed after x. To do so, we first state two important properties:

(P1) If an edge (v_i, w_j) of level y is crossed by an edge (v_k, w_ℓ) of level x with $x \prec y$, then (v_k, w_ℓ) must cross (v_i, w_j) *from above*: $i > k$ and $j < \ell$; see Figure 3.4 (b).

(P2) For any edge e of level i, there is a set of edges $\mathcal{E} = \{e_1, \ldots, e_{i-1}\}$ – one from each previous level – such that $\mathcal{E} \cup e$ is a set of i pairwise crossing edges; see Figure 3.4 (c).

Property (P1) follows from the construction of level x: If there where no edge of level x crossing edge (v_i, w_j), then (v_i, w_j) would also belong to level x.

Property (P2) holds by induction: For the first level there is no previous level. Edges of level two are crossed by edges of the first level due to (P1). Any edge (v_x, w_y) of level i must be crossed by some edge $(v_{x_{i-1}}, w_{y_{i-1}})$ of level $i-1$. Inductively we know that e_{i-1} is crossed by an edge of every previous level. Together, they form a chain of pairwise crossings from above, and we get the following patterns on the indices of these edges:

$$x > x_{i-1} > x_{i-2} > \cdots > x_1 \text{ and } y < y_{i-1} < y_{i-2} < \cdots < y < y_1.$$

These patterns indicate that in fact all the considered edges are pairwise crossing. For a given edge e of level i, we call any set following the description in property (P2) a *certificate* for e to be in i and any edge of level $i-1$ crossing e can be extended to some certificate for e.

Let t be the last level and consider the last edge $e^\dagger$ taken from that level. Suppose the algorithm created too many levels, so $t > k-2$. By property (P2), the certificate of $e^\dagger$ belonging to t together with $e^\dagger$ and $\overline{ab}$ forms a set of $t+1 \geq k$ pairwise crossing edges. This contradicts that the graph from which $e^\dagger$ is taken is outer k-quasi-planar. Hence, we never create more than $k-2$ levels.

As Algorithm 3.1 greedily takes any legal edge into the current set, each level is maximal by construction.

To argue about the connectivity of the levels we carefully consider the way they will be generated in a maximal graph. We ensure that greedily picking edges never disconnects the edge set of any level. Assume we already generated levels 1 to $i-1$ and in level i we pick the edges e and f but not e', as shown in Figure 3.5 (a).

Assume that edge e' exists but is not put into level i, ending up with that level being disconnected. Consider the relative positions of the endpoints of e' in the left and right regions with respect to the other edges of level i. As e' is going upward it cannot cross other edges of level i from above due to (P1). Thus, if e' is present in our graph, it would belong to level i and i would be connected, a contradiction.

The other option to have level i be disconnected is that edge e' is missing from the graph. As the graph is supposed to be maximal, there must be a *prevention* set of $k-1$ pairwise crossing edges that prevent its existance. Due to the edge $\overline{ab}$, the prevention set can contain at most $k-2$ crossing edges. By existence of the edges e and f we know that the prevention set also cannot consist of edges only locally on one side: Any prevention set locally left would also prevent the existance of f by covering v_w; a symmetric argument can be made against a locally right set together with e and w_x. In this context, $\overline{ab}$ is considered to be both locally left and locally right. Hence, any prevention set would have to include crossing edges as well, implying two possible options: crossing edges together with locally left or right edges – see Figure 3.5 (b). For illustration we color the different types of edges of the prevention set depicted in Figure 3.5 (b) differently: black edges (Figure 3.5 (c)) start between v_u and v_w, appearing between e and f but not belonging to the same level as e; red edges (Figure 3.5 (d)) cross e' from above ending strictly below w_x; orange edges Figure 3.5 (e) cross e' from below; blue edges (Figure 3.5 (f)) are locally on one side crossing all edges of other colors. By verifying the following claim, we finish the proof.

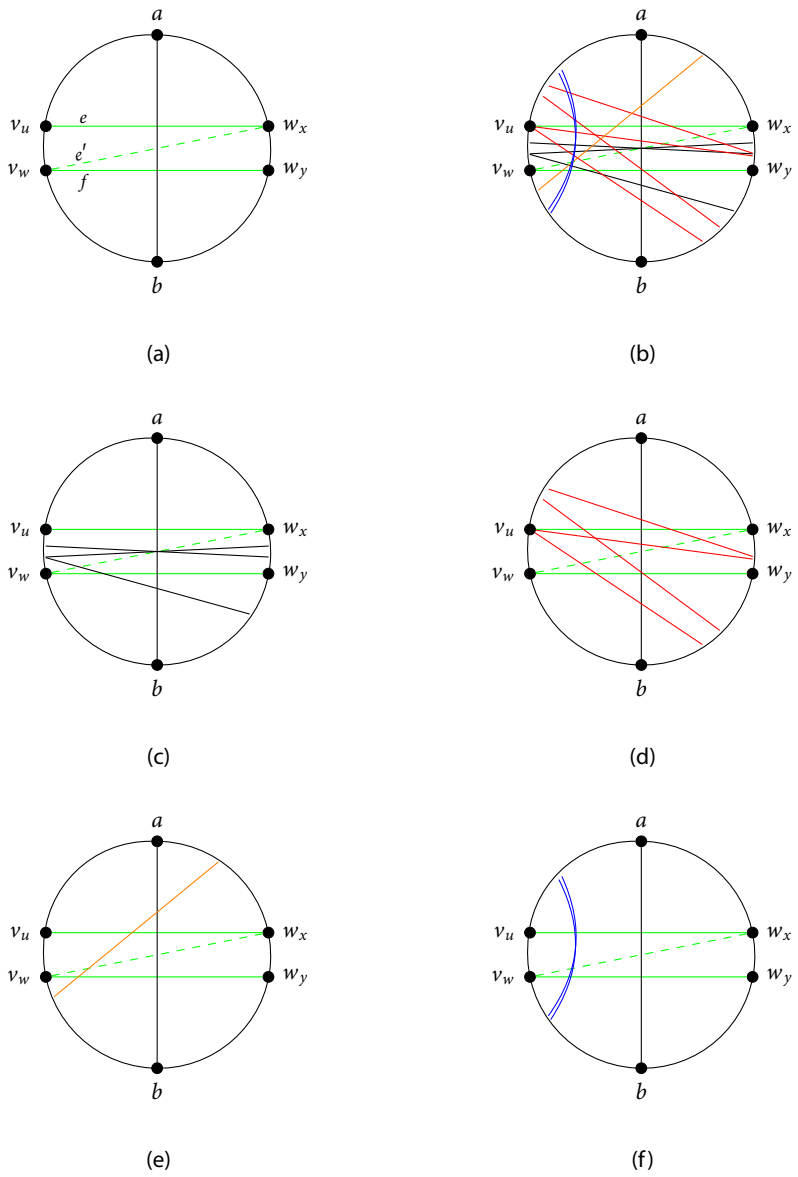


Figure 3.5: Connectivity of hierarchical level i: (a) The green level i is disconnected, as dashed edge e' is missing. (b) A full set of edges blocking e' using only locally left edges; (c) black edges start between v_u and v_w, but do not belong to the same level as e and f, (d) red eges cross e' from above and end below w_x, (e) orange edges cross e' from below, and (f) blue edges start and end on the same side.

Claim: A prevention set $\mathcal{P}$ for e' can be transformed into a prevention set for f (or symmetrically e) – contradicting the existence of that set.

To transform a prevention set $\mathcal{P}$ for e' into a prevention set $\mathcal{P}'$ for f, we proceed as follows: We keep all the orange and blue edges of $\mathcal{P}$, as they already pairwise cross and also cross f. If we replace any of the remaining edges, we need to make sure that the new edges also cross the blue and orange ones in $\mathcal{P}'$.

Any other edges in $\mathcal{P}$ belong to previous levels because of the following:

- By (P1), all red edges cross e' from above.
- For the black edges, we have two different cases: each edge either crosses f or lives completely between e and f. Again by (P1), edges crossing f do so from above. Edges that live between e and f can not be crossed by edges of level i, so they must belong to a level before level i.

We now consider the edges of $\mathcal{P}$ in the order of the level they belong to starting at $i-1$. For any such level, there is an edge crossing f whose left vertex is between the two endpoints of every blue of $\mathcal{P}$. To see that this is true, assume that there is some edge e_k of level $k < i$ crossing e' but not f. Since f is not in level k, there must be an edge $e^\times$ crossing it and its right vertex must be below the right vertex of e_k. This leaves to options for the left vertex: it can either also be the left vertex of e_k or some other vertex below it but still above v_w; in both cases, the new left vertex will be covered by all blue edges and we can replace e_k by $e^\times$. In the worst case, performing this replacement stops all remaining edges of the certificate for e' from crossing $e^\times$ because it is below them. As $e^\times$ is in level k, there must be a certificate for that. Notice that every edge of this certificate must be below its corresponding counterpart of the certificate for e' and thus still all starting vertices are covered by blue edges. Hence then we also replace all edges of layers before k by edges of the certificate for $e^\times$.

This conclude that a prevention set for e' using locally left edges can be transformed into a prevention set for f. A similar argument can be made using locally right edges by again taking a prevention set for e', transforming the certificate for e' into one for e and observing that the endpoints of of the edges of the certificate are again still covered by the locally right edges.

We have established that edge e' must exist and has to be placed into the same layer as e and f, connecting the two components. □

Considering the construction of Pach and Capoyleas [CP92], the proof of Claim 4 yields the following:

Remark. Let G be an outer k-quasi-planar graph with n vertices and let the vertices be labeled $v_1, v_2, \ldots, v_n, v_{n+1} = v_1$ according to their cyclic order along the outer face. Every vertex v_i can be adjacent to v_ℓ with $\ell \in [i-(k-1), i+(k-1)]$. Hence, these *frame* edges are present in any maximal outer k-planar graph; see Figure 3.6 (a).

Using the hierarchical levels described above, we give a *split*operation, which is used to split the graph into two smaller parts. Let G be a maximal outer k-quasi-planar graph with a long edge $\overline{ab}$ and hierarchical levels created by Algorithm 3.1. Let L_i and R_i be the vertices of G incident to the crossing edges of level i in the regions left and right of $\overline{ab}$, respectively. Splitting G into two subgraphs G_1 and G_2 is done as follows: To obtain G_1, for every level i from 1 to $(k-2)$, replace the vertices of L_i by a single *level*-vertex v_i and connect that vertex to all vertices in R_i, see Figure 3.6 (b) and (c). Finally, add to G_1 all missing frame edges to make it maximal. To obtain G_2, proceed symmetrically, exchanging the roles for left and right.

Lemma 3.9. *After applying a split operation to a maximal outer k-quasi-planar graph G, the following relations among G and it's two subgraphs G_1 and G_2 hold:*

(i) $|V(G)| = |V(G_1)| + |V(G_2)| - 2k + 2$ *and*

(ii) $|E(G)| = |E(G_1)| + |E(G_2)| - (|E'_1| + |E'_2|) + |E'| - 1$,

where E' is the set of edges of G crossing $\overline{ab}$ and E'_1, E'_2 are the sets of crossing edges added to G_1, G_2 by the split operation.

Proof. We proof the two equations (i) and (ii) individually.

(i): In G_1 and G_2 are obtained by only modifying vertices on the right or left side of G respectively, leaving the other side unmodified and $\overline{ab}$ present in both. The modification adds $k-2$ level-vertices to each graphs, so subtracting thes vertices and one copy of a and b, yields

$$|V(G)| = |V(G_1)| - (k-2) + |V(G_2)| - (k-2) - 2 = |V(G_1)| + |V(G_2)| - 2k + 2.$$

(ii): We count the edges added to both G_1 and G_2 and compare them to the number of edges removed by splitting G. To do so, we consider the structure of our hierarchical levels and the respective left and right vertices L_i and R_i. Every level is a catarpillar – a set of connected and non-crossing edges –, so there are exactly $|L_i| + |R_i| - 1$ edges in level i. Hence, the total number of crossing edges over all levels is

$$|E'| = \sum_{i=1}^{k-2} (|L_i| + |R_i| - 1).$$

The sets of edges added to G_1 and G_2 each consist of two different subsets: The first subset are the edges incident to level-vertices, the other subset contains the missing frame edges added to make G_1 and G_2 maximal. The size of the first set can be expressed by summing up the sizes of all sets L_i and R_i with $1 \leq i \leq k-2$ as follows:

$$\sum_{i=1}^{k-2} |R_i| + |L_i|.$$

The subgraphs induced by vertices and edges of the unmodified sides of G_1 and G_2 remain maximal by maximality of G. But considering the last vertex of some level, in some cases it seems possible to add an additional edge incident to that vertex to the drawing – for instance, see vertex v from Figure 3.6 (c). To count the presumably missing edges, recall how we generated the hierarchical levels. In level i we take all remaining edges of right-side vertex $w_{h-(i-1)}$ – for the first level, we take all edges of the last right-side vertex, for the second level all edges of the second-to-last vertex, and so on. The total number of edges missing this way in both subgraphs together is

$$2 \cdot \sum_{i=1}^{k-2} (i-1).$$

The distance of each level vertex to the lowest vertex on the other side in the cyclic order – counted clockwise or counter clockwise, depending on in which region the level vertex was placed – is by construction bounded by $k-1$. Hence, any edge missing this way is actually a frame edge that can safely be added to the subgraph to make it maximal.

We added $k-2$ level vertices and the maximum number of frame edges incident to them. As G is a k-quasi-planar graph, the vertices connecting these frame edges together with a and b form a clique on $k-2+2$ vertices. The number of edges in this clique, and hence the exact total amount of edges we add to G_1 and G_2 this way (without counting $\overline{ab}$) is

$$2\left(\binom{k}{2} - 1\right).$$

Putting everything together, the total number of new edges added to G_1 and G_2 is

$$|E_1'| + |E_2'| = 2\left(\binom{k}{2} - 1\right) + \sum_{i=1}^{k-2} R_i + L_i + 2(i-1).$$

Notice that we did not account for $\overline{ab}$ in any of the subgraphs yet, so we subtract one copy of it. The remaining parts of G_1 and G_2 are the unmodified sides of G and one copy of $\overline{ab}$. Simplifying the above equation completes the proof. □

To complete our inductive argument, we need to do two things: We need to prove the existence of a long edge in any maximal outer k-quasi-planar graph and consider the base cases – those maximal outer k-quasi-planar graphs with the minimum number of vertices.

Lemma 3.10. *Any maximal outer k-quasi-planar graph $G = (V, E)$ either*

(i) is a clique of size $|V| \leq 2k-1$, or

(ii) has a long edge.

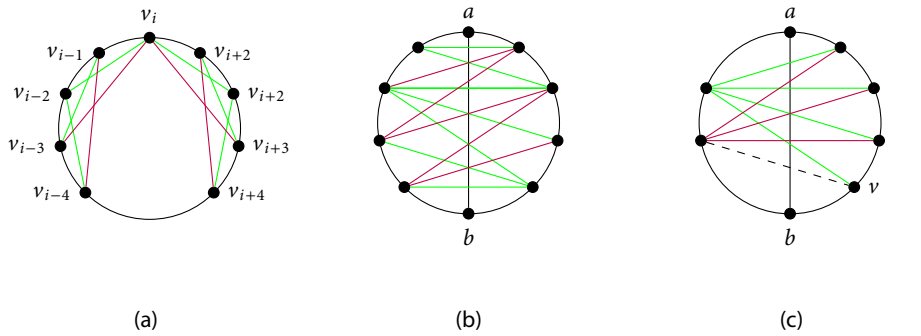


Figure 3.6: Using hierarchical levels to perform the split operation: (a) The frame of a maximal outer k-quasi-planar graph; black edges are present for any k, green edges for $k \geq 3$, purple edges for $k \geq 4$). (b) An outer 4-quasi-planar graph with 2 hierarchical levels shown in green and purple respective; (c) the resulting subgraph G_1, the presumably missing edge of v (drawn dashed) is actually a frame edge.

Proof. We consider each case individually, depending on number of vertices in the graph.

For case (i), when $|V| \leq 2k - 1$, the graph G has to have frame edges by maximality. For any vertex v these edges connect it to the closest $k - 1$ other vertices left and right of it. As any v together with it's frame edge neighbors covers all vertices of G, the graph itself is K_{2k-1}.

In the other case – (ii) – we have $|V| > 2k - 1$: Doing a simple counting argument on the number of neighbors induced by frame edge, we get that for every vertex v_i ($i \in 1, \ldots, n$), there is at least one other vertex w_i that it is not connected to via a frame edge. Consider the pair v_i, w_i for some i. If it were connected by an edge, this would be the long edge we are looking for. So suppose the edge v_i, w_i is missing. As we choose G to be maximal, there must be some set $\mathcal{S}$ of edges preventing the existence. As edges of $\mathcal{S}$ can not be part of the frame, they must span more than $k - 1$ vertices on both sides. Hence, $\mathcal{S}$ must contain at least on long edge. □

Finally, we combine the results of the lemmas presented above to complete the proof of Theorem 3.7.

Proof of Theorem 3.7. By Lemma 3.10, the graph G we consider in each step either (i) is a clique or (ii) we always find a long edge to split by. By picking a long edge in G, dividing it into two regions, building the hierarchical levels with respect to these regions and performing the split operation as described above, we get two subgraphs G_1 and G_2 of smaller size. By Lemma 3.9 we get the relationships on vertex- and edge-count between G and both subgraphs. We recursively repeat the spliting on these subgraphs until we encounter cliques. We then know that the number of edges matches the bound on total edge number.

Given a maximal outer k-quasi-planar graph G, we can recursively split it into pieces that individually retain maximality and outer k-quasi-planarity. Considering the relationship among the edge sets of Lemma 3.9 (ii) and the maximality of the subgraphs, we get the following equation:

$$\begin{aligned}|E(G)| &= |E(G_1)| + |E(G_2)| - (|E_1'| + |E_2'|) + |E'| - 1\\ &= |E(G_1)| + |E(G_2)| - 2k^2 + 5k - 3\\ &= 2(k-1)(n_1+n_2) - 2\tbinom{2k-1}{2} - 2k^2 + 5k - 3\end{aligned}$$

From the maximality of G and using Lemma 3.9 (i), we now have:

$$\begin{aligned}2(k-1)(-2k+2) &= -\tbinom{2k-1}{2} - 2k^2 + 5k - 3\\ -4k^2 + 8k - 4 &= -4k^2 + 8k - 4\end{aligned}$$

The equation balances, completing the proof. □

3.4 Testing for Full Convex Drawings via MSO_2

Hong and Nagamochi [HN16] were the first to introduce the class of *full outer k-planar graphs*. Graphs in this class have a convex drawing which is k-planar and additionally there is no crossing on the outer boundary of the drawing – every corner of the (not neccessarily simple) polygon prescribing the outer face is a vertex of the graph. Hong and Nagamochi gave a linear-time recognition algorithm for full outer 2-planar graphs. They state that a graph G is (full) outer-2-planar, if and only if its biconnected components are (full) outer-2-planar and that the outer boundary of a full outer-2-planar embedding of a biconnected graph G is a Hamiltonian cycle of G. In Theorem 3.11, we observe that this property also carries over to general outer k-planar and outer k-quasi-planar graphs. Therefore, we define the classes of *closed outer k-planar* and *closed outer k-quasi-planar* graphs, where closed means that there is an appropriate convex drawing where the circular order forms a Hamiltonian cycle.

In the following, we first give a basic introduction to Monadic Second-Order Logic (MSO_2) and Courcelles' Theorem, then use MSO_2 to express crossing patterns of closed k-planar and k-quasi-planar graphs. This will result in a linear-time algorithm for closed outer k-planar. Proving the following Theorem 3.11 will conclude this section, translating the algorithm to full outer k-planar graphs.

Theorem 3.11. *To test a given graph for full outer k-planarity or outer k-quasi-planarity it suffices to test its biconnected components for closed outer k-planarity or outer k-quasi-planarity.*

Monadic Second-Order Logic (MSO_2) – a subset of *second-order logic* – can be used to express certain graph properties. Formulas in MSO_2 can be built using following primitives:

- Variables for vertices, edges, sets of vertices, and sets of edges;
- Binary relations for equality ($=$), membership in a set ($\in$), subset of a set ($\subseteq$), and edge–vertex incidence (I);
- Standard propositional logic operators: $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$.
- Standard quantifiers ($\forall$, $\exists$) which can be applied to all types of variables.

For a graph G and an MSO_2 formula ϕ, we use $G \vDash \phi$ to indicate that ϕ can be satisfied by G in the obvious way. Properties expressed in this logic allow us to use the powerful algorithmic result of Courcelle stated next.

Theorem 3.12 ([Cou90, CE12])**.** *For any integer $t \geq 0$ and any MSO_2 formula ϕ of length ℓ, an algorithm can be constructed which takes a graph G with treewidth at most t and decides in $O(f(t,\ell)\cdot(n+m))$ time whether $G \vDash \phi$ where the function f from this time bound is a computable function of t and ℓ.*

By Proposition 8.5 of Wood and Telle [WT07] we know that outer k-planar graphs have treewidth $O(k)$. Therefore, expressing outer k-planarity by an MSO_2 formula whose size is a function of k would mean that outer k-planarity could be tested in linear time. However, this task might be out of the scope of MSO_2. The challenge in expressing outer k-planarity in MSO_2 is that MSO_2 does not allow quantification over sets of pairs of vertices v_1, v_2 when v_1 and v_2 are not connected by an edge. Namely, it is unclear how to express a set of pairs that forms the circular order of vertices on the boundary of our convex drawing. However, if this circular order forms a *Hamiltonian cycle* in our graph, i.e the given graph is closed, then we can indeed express this in MSO_2. With the edge set of a Hamiltonian cycle of our graph in hand, we can then ask that this cycle was chosen in such a way that the other edges satisfy either k-planarity or k-quasi-planarity.

Theorem 3.13. *Closed outer k-planarity and closed outer k-quasi-planarity can be expressed in MSO_2. Thus, closed and also full outer k-planarity can be tested in linear time.*

Any formula presented here assumes that a graph G is given and uses edges, vertices and incidences of G. To simplify notation in the following, we will always use e and f as variables for edges, F as a set of edges, u, v as vertices and U as a set of vertices (also including sub- and superscripted variants). In addition to the quantifiers above we also use a logical shorthand for the existence of exactly x elements ($\exists^{=x}$) satisfying a property, that all are unequal and that no $x+1$ such elements exist.

The following formula allows us to describe connectedness of subgraphs induced by an edge set F.

$$\textsc{Connected-Edges}(F) \equiv \quad (\forall F' \subset F)[\exists e, f : e \in F' \wedge f \in F \setminus F'] \wedge$$
$$\Big((\exists f, e, e^* \in F : e \in F' \wedge f \notin F')(\exists u, v)[I(u,f) \wedge I(v,e) \wedge I(v,e^*) \wedge I(u,e^*)]\Big)$$

It states that for every proper subset F' of our edge set F, we can find three edges e, f, e^* – one in F', one not in F' and one connecting set and subset.

The following formulas are used to describe Hamiltonicity of G. Formula Cycle-Set implies that the edges of F form cycles, Cycle implies maximality of the cycle and Span forces the cycle to have an edge incident to every vertex of G.

$$\begin{aligned}
\textsc{Cycle-Set}(F) &\equiv (\forall e)\Big[e \in F \Rightarrow (\exists^{=2} f)\big[f \in F \wedge e \neq f \wedge (\exists v)[I(e,v) \wedge I(f,v)]\big]\Big] \\
\textsc{Cycle}(F) &\equiv \textsc{Cycle-Set}(F) \wedge \textsc{Connected-Edges}(F) \\
\textsc{Span}(F) &\equiv (\forall v)(\exists e)[e \in F \wedge I(e,v)] \\
\textsc{Hamiltonian}(F) &\equiv [\textsc{Cycle}(F) \wedge \textsc{Span}(F)]
\end{aligned}$$

Vertex-Partition implies the existence of a partition of the vertices of G into k disjoint subsets.

$$\textsc{Vertex-Partition}(U_1, \ldots, U_k) \equiv (\forall v)\left[\left(\bigvee_{i=1}^{k} v \in U_k\right) \wedge \left(\bigwedge_{i \neq j} \neg(v \in U_i \wedge v \in U_j)\right)\right]$$

For a closed outer k-planar or closed outer k-quasi-planar graph G, we want to express that two edges e and e_i cross. To this end, we assume that there is a Hamiltonian cycle E^* of G that defines the outer face. We partition the vertices of G into three subsets C, A, and B, as follows: C is the set containing the endpoints of e, whereas A and B are the two subgraphs on the remaining vertices connected using only edges of E^*. This partition devides the vertices of G into two special sets, one left and the other one right of e. For such a partition, e_i must cross e whenever e_i has one endpoint in A and one in B.

$$\begin{aligned}
\textsc{Crossing}(E^*, e, e_i) \equiv (\forall A, B, C)\big[&\big(\textsc{Vertex-Partition}(A, B, C) \\
&\wedge (I(e,x) \leftrightarrow x \in C) \wedge \textsc{Connected}(A, E^*) \wedge \textsc{Connected}(B, E^*)\big) \\
&\rightarrow (\exists a \in A)(\exists b \in B)[I(e_i, a) \wedge I(e_i, b)]\big]
\end{aligned}$$

Now the crossing patterns for closed outer k-planarity and closed outer k-quasi-planarity can be described using the formulas presented above as follows:

$$\textsc{Closed Outer } k\textsc{-Planar}_G \equiv (\exists E^*)\Big[\textsc{Hamiltonian}(E^*) \wedge$$

$$(\forall e)\Big[(\forall e_1, \ldots, e_{k+1})\Big[\Big(\bigwedge_{i=1}^{k+1} e_i \neq e \wedge \bigwedge_{i \neq j} e_i \neq e_j\Big) \rightarrow \bigvee_{i=1}^{k+1} \neg\textsc{Crossing}(E^*, e, e_i)\Big]\Big]\Big]$$

Here we insist that G is Hamiltonian and that, for every edge e and any set of $k+1$ distinct other edges, at least one among them does not cross e.

$$\textsc{Closed Outer } k\text{-}\textsc{Quasi-Planar}_G \equiv (\exists E^*)\Big[\textsc{Hamiltonian}(E^*) \wedge$$

$$(\forall e_1, \ldots, e_k)\Big[\Big(\bigwedge_{i \neq j} e_i \neq e_j\Big) \rightarrow \bigvee_{i \neq j} \neg\textsc{Crossing}(E^*, e_i, e_j)\Big]\Big]$$

Again, we insist that G is Hamiltonian and further that, for any set of k distinct edges, there is at least one pair among them that does not cross.

This gives us linear-time recognition of closed outer k-planar graphs and we can now proceed to prove Theorem 3.11.

Proof of Theorem 3.11. Let G be a graph. It is well-known that the set of cut vertices of G can be obtained in linear time. Splitting G at the cut vertices, we obtain biconnected closed components if and only if G was full itself: When one component only has drawings showing some crossing on the outside, this carries over to a drawing of the full graph – since the biconnected components of a graph form a tree, attaching the other components can not close a cycle that covers a side of a component

Every biconnected component is considered individually and checked in linear time. Each non-cut vertex belongs to exactly one component and thus is checked only once. The total effort spent on considering cut vertices is bounded by the number of biconnected compontents. Hence, so a simple charging argument on the vertices concludes runtime analysis.

The MSO_2 formula given above guarantees that the Hamiltonian cycle – if present – is placed on the outer boundary of the drawing of each component. Putting together the individual drawings of the components can be done crossing free be reidentifying the cut vertices. □

3.5 Conclusion

In this chaper, we explored two extended outerplanar settings – namely outer k-planarity and outer k-quasi-planarity.

For the outer k-planar graphs, we have shown that they are $(\lfloor\sqrt{4k+1}\rfloor+1)$-degenerate and thus also have chromatic number $(\lfloor\sqrt{4k+1}\rfloor+2)$. We further showed that they have separators of bounded size $-2k+3$, improving the old bound obtained from the bounded treewidth. This allowed us to give an algorithm for testing outer k-planarity in quasi-polynomial time.

For the outer k-quasi-planar graphs, we looked into comparability to planar graphs, showing that outer 3-quasi-planar graphs and planar graphs are incomparable under containment by providing instances. We have also reconsidered the edge-maximal outer k-quasi-planar graphs, showing that those graphs are also edge-maximum. We obtained

this result by giving a recursive argument, enumerating crossing patterns involving carefully picked edges.

In the last section, we provided an overview to Monadic Second-Order Logic. We used MSO_2 to express closed and full outer k-planar and outer k-quasi-planar graphs – insisting on the boundaries of the biconnected components to be Hamiltonian cycles for that component. Together with the bounded treewidth and Courcelle's Theorem, we showed that full and closed outer k-planarity can be tested in polynomial time.

As some of our claims were verfied using a computer SAT solvers, we provide the formulations of our models in the next section. While we only used them to overcome exhaustive checking by hand, they are generally extendable and might be of interest for future research.

3.6 Additional Resources

3.6.1 Outer quasi-planarity checker

In this section, we describe a boolean formula for testing whether a given graph is outer quasi-planar. We present the formula in first-order logic. After transformation to boolean logic, we solve the formula using the MiniSat [SE05] solver.

A quasi outer-planar embedding corresponds to a circular order of the vertices. Cutting a circular order at some vertex v turns the circular into a linear order starting and ending at v. However, the edge crossing pattern remains the same. Therefore, we look for a linear order.

We need the following two sets of variables. For any pair of vertices $u \neq v \in V$, we introduce a boolean variable $x_{u,v}$ that expresses that vertex u is before v in the linear order. In addition, for any pair of edges $e \neq e' \in E$ we introduce a boolean variable $y_{e,e'}$ that expresses that edge e crosses edge e'. Now we list the sets of clauses present in our SAT formula.

$$x_{u,v} \wedge x_{v,w} \Rightarrow x_{u,w} \qquad \text{for each } u \neq v \neq w \in V; \tag{3.1}$$

$$x_{u,v} \Leftrightarrow \neg x_{v,u} \qquad \text{for each } u \neq v \in V; \tag{3.2}$$

$$x_{u,u'} \wedge x_{u',v} \wedge x_{v,v'} \Rightarrow y_{e,e'} \qquad \text{for each } e = (u,v) \neq e' = (u',v') \in E; \tag{3.3}$$

$$\neg(y_{e_1,e_2} \wedge y_{e_1,e_3} \wedge y_{e_2,e_3}) \qquad \text{for each } e_1, e_2, e_3 \in E \text{ with different endpoints.} \tag{3.4}$$

The first two sets of clauses describe the neccesary properties of a linear order. Clause (3.1) realizes transitivity, and clause (3.2) anti-symmetry. Clause (3.3) ensures that variable $y_{e,e'}$ is set to true whenever the linear ordering on the endpoints of e and e' implies a crossing. Finally, clause (3.4) ensures that no three edges pairwise cross.

3.6.2 Page number checker

A similar SAT solver for checking the page number of a graph has been implemented by Pupyrev [Pup18]. For a given graph G and integer $k > 0$, we provide a SAT formula that has a satisfying truth assignment if and only if G has page number k. We find a linear order of the vertices that corresponds to a k-page embedding.

To do so, we again need two sets of variables. For every pair of vertices $u \neq v \in V$, we introduce a boolean variable $x_{u,v}$ that expresses that u is before v in the linear order – as in Section 3.6.1. For every edge $e \in E$ and page $i \in \mathcal{P} = \{1, \dots k\}$, we introduce a boolean variable $p_{i,e}$ that expresses that edge e is on page i.

Now we list the clauses of our SAT formula.

$$x_{u,v} \wedge x_{v,w} \implies x_{u,w} \qquad \text{for each } u \neq v \neq w \in V; \tag{3.5}$$

$$x_{u,v} \Leftrightarrow \neg x_{v,u} \qquad \text{for each } u \neq v \in V; \tag{3.6}$$

$$\bigvee_{i \in \mathcal{P}} p_{i,e} \qquad \text{for each } e \in E; \tag{3.7}$$

$$\neg(p_{i,e} \wedge p_{j,e}) \qquad \text{for each } i \neq j \in \mathcal{P}, e \in E; \tag{3.8}$$

$$p_{i,e} \wedge_{i,e'} \implies \neg(x_{u,u'} \wedge x_{u',v} \wedge x_{v,v'}) \qquad \text{for each } i \in \mathcal{P}, e \neq e' \in E : \tag{3.9}$$

$$e = (u, v) \text{ and } e' = (u', v'). \tag{3.10}$$

The first two sets of clauses again describe the linear order – they are the same as Clauses (3.1)–(3.2). Clauses (3.7)– (3.8) guarantee that every edge is on exactly one page. Clause (3.9) ensures that two edges do not cross on the same page.

Chapter 4

One-Bend Drawings of Outerplanar Graphs with fixed Shape

One of the fundamental problems in graph drawing is to draw a planar graph crossing-free under certain geometric or topological constraints. Many classical algorithms draw planar graphs under the constraint that all edges have to be straight-line segments [Sch90, dFPP90, Tut63]. But for practical applications we do not always have the freedom of drawing the whole graph from scratch, as some important parts of the graph may already be drawn. For example, in visualizations of large networks, certain patterns may be required to be drawn in a standard way, or a social network may be updated as new people enter a social circle or as new links emerge between already existing persons. In that case, we might want to extend a given drawing to a drawing of the whole graph.

For planar graphs, this problem is known as the Partial Drawing Extensibility problem. Formally, given a planar graph $G = (V, E)$, and subgraph $H = (V', E')$ with $V' \subseteq V$ and $E' \subsetneq E$, and a planar drawing Γ_H of H, the problem asks for a planar drawing Γ_G of G such that the drawing of H in Γ_G coincides with Γ_H. This problem was first proposed by Brandenburg et al. [BEG+03] in 2003. Since then it has received a lot of attention in the previous years.

4.1 Related work

For the case of extending a given straight-line drawing using straight lines as edges, Patrignani showed the problem to be NP-hard [Pat06], but he could not prove membership in NP, as a solution may require coordinates not representable with a polynomial number of bits. Recently, Lubiw et al. [LMM18] proved that a generalization of the problem where overlaps (but not proper crossings) between edges of $E \setminus E'$ and E' are allowed is hard for the existential theory of the reals ($\exists\mathbb{R}$-hard).

These results motivate allowing bends in the drawing. Angelini et al. [ADF+15] presented an algorithm to test in linear time whether there exists any topological planar drawing of G, and Jelínek et al. [JKR13] gave a characterization via forbidden substructures. Chan et al. [CFG+15] showed that a linear number of bends ($72|V'|$) per edge suffices. This number is also worst-case optimal as shown by Pach and Wenger [PW01] for the special case of the host graph not containing edges ($E' = \emptyset$).

Special attention has been given to the case that the host H is exactly the outer face of G. Already Tutte's seminal paper [Tut63] showed how to obtain a straight-line convex

drawing of a triconnected planar graph with its outer face drawn as a prescribed convex polygon. This result has been extended by Hong and Nagamochi [HN08] to the case that the outer face is drawn as a star-shaped polygon without chords (that is, interior edges between vertices on the outer face). Mchedlidze et al. [MNR13] give a linear-time algorithm to test for the existence of a straight-line drawing of G in the case that H is an arbitrary cycle of G and Γ_H is a convex polygon. Mchedlidze and Urhausen [MU18] study the number of bends required based on the shape of the drawing of H and show that 1 bend suffices if H is drawn as a star-shaped polygon.

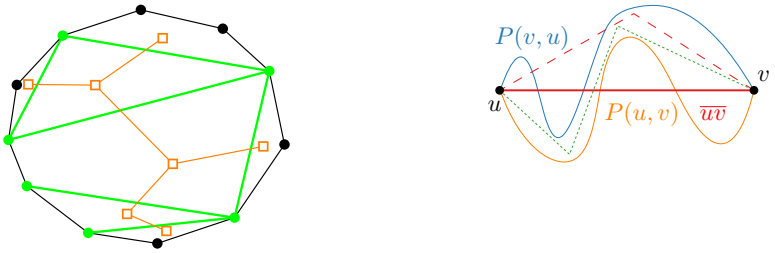
Contribution. In this chapter, we consider a special drawing extension setting: We are given a biconnected outerplanar graph $G = (V_I \cup V_H, E_I \cup E_H)$. The edge set is divided into two subsets E_H and E_I: The edges in E_H define a Hamiltonian cycle in G that prescribes the outer face; the other set of edges E_I are the *inner* edges. Having the edge set partitioned into two subsets, we can partition the vertices in a similar way: Subset V_I contains all vertices incident to edges of E_I, whereas V_H contains all other vertices. As host graph, we are given the subgraph $H = (V_I \cup V_H, E_H)$. The drawing Γ_H of H forms a simple polygon P, in which all edges are drawn as straight lines. One can imagine the vertices of V_I being mapped to the boundary of P. We want to draw the missing edges of E_I inside of the region defined by P such that Γ_G is crossing free, allowing one bend per edge of E_I.

For any constant number k of bends, there exists some instance such that G has a k-bend drawing but no $(k-1)$-bend drawing; see, e.g., Fig. 4.1 (b) for $k = 2$. Hence, it is of interest to test for a given k whether a k-bend drawing of G exists. This task is trivial for $k = 0$. In this paper, we give OneBendInPolygon – see Algorithm 4.1); it solves this problem for $k = 1$ in time $O(|V_I| \cdot p)$, where p is the number of corners of P. In the following, in Section 4.2 we establish notation and the lemmas necessary to describe the algorithm. We then state OneBendInPolygon in Section 4.3; finally in Section 4.4 we proof correctness and runtime.

4.2 Notation

In the following, we consider the setting described above, but with the notation simplified as follows: We summarize all elements of host graph H under the polygon P such that the Hamiltonian cycle coincides with the boundary ∂P. Hence, the problem at hand reduces to finding an outerplanar one-bend drawing of subgraph $G_I = (V_I, E_I)$ inside of P where the vertices V_I are already mapped to ∂P. We say that G *can be drawn in* P if there is a crossing-free drawing of G with its vertices on ∂P as defined by the mapping, its outer face drawn as ∂P, and its interior edges drawn with at most one bend per edge.

For a pair of vertices u and v mapped to ∂P, we denote by $\overline{uv}$ the straight-line segment between them. The given mapping also orders the vertices of V_I along the boundary. Starting at u and following ∂P in counterclockwise order until reaching v, we obtain the open interval $P(u, v)$ – the piece of ∂P between u and v. As a complement, we also



(a) (b)

Figure 4.1: (a) A biconnected outerplanar graph, subgraph G_I in green and the dual tree in orange. (b) For an edge $e = (u, v)$, the straight line $\overline{uv}$ intersects both $P(u, v)$ and $P(v, u)$, the dashed red 1-bend drawing of e only avoids crossing $P(u, v)$, possible 2-bend drawing in green.

have the open interval $P(v, u)$ – the piece between v and u. By concatenation of the two vertices and the two pieces we get $\partial P = u \circ P(u, v) \circ v \circ P(v, u)$.

Considering the full graph G, the faces of G induce a unique dual tree T [PS86] where each edge of T corresponds to an *interior* edge of E_I; see Fig. 4.1 (a). Any interior edge $e = (u, v)$ divides the polygon into two parts and for an edge corresponding to a leaf of the dual tree, one of these parts does not contain other vertices of V_I. The faces corresponding to these empty parts are exactly the leafs of T. In the following, we consider T to be rooted at some leaf node f_n. For some face f_i, we denote by $p(f_i)$ the parent of f_i in T, and by e_i the edge between f_i and its parent. We say that f_i and f_j are *siblings* if $p(f_i) = p(f_j)$.

OneBendInPolygon will traverse the dual tree twice – first bottom-up and then top-down. Fixing some leaf f' of T as the root, we can consider the faces of G in sequence of the bottom-up traversal $f_1, \ldots, f_n$ with $f_n = f'$. In each step, we incrementally process an interior edge of G_I, and pruning T and refining P accordingly. We will now define the pruning and refinement operations to then give a description of the algorithm.

The sequence in which the edges are processed also implies a sequence of subtrees of T. For step i ($1 \leq i \leq n$), let T_i be the subtree of T induced by the nodes $f_i, \ldots, f_n$ – hence for the first step we have $T_1 = T$ and for the final step we have $T_n = (\{f_n\}, \varnothing)$. By pruning the tree, eventally every node of T will become a leaf node.

Similar to the sequence of pruned subtrees, we also have a corresponding sequence of refined polygons $P_1, \ldots, P_n$. In the first step we have $P_1 = P$ and after the last step of the bottom-up traversal P_n incorporates all previous refinements. In step i, we process edge $e_i \in E_I$ and by choice of sequence, one of the parts induced by e_i in P_i will be a leaf in T_i. Putting a one-bend drawing of e_i into P_i using bend point b, we classify the type of corner that b will induce in P_{i+1}. We say that interior edge e_i is either

- a *reflex* edge if bend point b has to be placed in a way that enforces a reflex corner to occur at b in P_{i+1} – see Fig. 4.2 (a) – or
- a *convex* edge if a placement of b resulting in a convex corner at b in P_{i+1} or as a straight line is possible.

Let G_i be the subgraph of G_I induced by the vertices incident to the faces $f_i, \dots f_n$, hence $G = G_1$. In step i our algorithm picks a leaf f_i of T_i and processes the interior edge e_i between f_i and its parent such that the following invariant holds:

"Graph G_{i+1} can be drawn in P_{i+1} if and only if G_i can be drawn in P_i."

To maintain the invariant during the bottom-up traversal of T, we want to refine P in the least restrictive way – cutting away as little of P as possible. In the following, we will see that the

4.3 Procedure

The objective for this section is to state OneBendInPolygon – see Algorithm 4.1. Before we can do that, we describe how it will proceed in more detail, establishing the necessary lemmas on the way.

Among all leafs of T_i, the algorithm chooses the next node f_i to process as follows: If T_i has a leaf corresponding to a reflex edge, we process the corresponding interior edge next. Otherwise, all leaves in T_i correspond to convex edges, and we choose one of nodes among them that has the largest distance to root f_n in T. We do this to make sure that a convex edge is only chosen if all siblings corresponding to reflex edges have already been processed.

Let $e_i = (u, v)$ be the interior edge corresponding to leaf f_i. Let V_u and V_v be the regions inside of P_i visible from u and v, respectively, and let $V_{e_i} = V_u \cap V_v$ be their intersection – the region visible by both end points. Clearly, any valid bend point for e_i needs to be inside V_{e_i}. For any point $b \in V_{e_i}$, let $Q^b_{e_i}$ be the *obstructed region*, the subpolygon of P_i bounded by $P_i(u, v) \circ \overline{vb} \circ \overline{bu}$ – the part of P_i that is "cut off" by drawing e_i with its bend at b. We call a bend point $b \in V_{e_i}$ *minimal* for e_i if there is no other point $b' \in V_{e_i}$ with $Q^{b'}_{e_i} \subsetneq Q^b_{e_i}$.

For reflex edges, we have the following lemma regarding minimal bend points.

Lemma 4.1. *Let $e_i = (u, v)$ be a reflex edge to be drawn in polygon P_i. If there is a valid drawing of e_i in P_i, there is also a unique minimal bend point b for e_i.*

Proof. We construct b by considering the shape of $P(u, v)$ and the visibility region V_{e_i}. As e_i is a reflex edge, some parts of $P(u, v)$ must extend over $\overline{uv}$ – either by intersecting $\overline{uv}$ or by $\overline{uv}$ being completely outside of P_i.

By assumption there is some valid drawing of e_i; let b' be the bend point of that drawing; see Fig. 4.2 (a). Consider the line $\overline{ub'}$ and rotate it clockwise arround u until

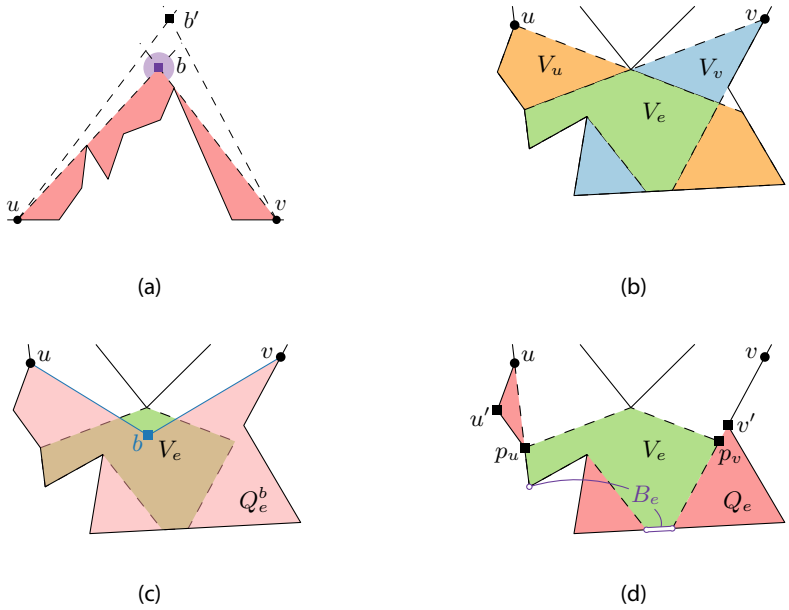


Figure 4.2: Illustrations of minimal bend points and obstructed regions for an edge e connecting u and v: (a) e is a reflex edge. We show two possible drawings using either b or b': both cut away at least the red region and b is the minimal bend point to do so. In both cases, the modified polygon has a reflex angle at the bend. (b) e is a convex edge: Visibility region V_u of vertex u in orange, visibility region V_v of vertex v in blue, and intersection $V_e = V_u \cap V_v$ in green; (c) the region Q_e^b cut off by drawing e with its bend at point b in red; (d) construction of p_u, p_v and of the obstructed region Q_e, set of bend point options B_e in purple.

intersecting $P(u,v)$. Rotating $\overline{ub'}$ towards v also moves the intersection point $b^\dagger$ of both lines closer to v. That way we obtain the obstructed region $Q_{e_i}^\dagger = P(u,v) \circ \overline{ub^\dagger} \circ \overline{b^\dagger v}$ and $Q_{e_i}^{b^\dagger} \subset Q_{e_i}^{b'}$. Doing the same with line $\overline{b^\dagger v}$, rotating counterclockwise towards u, we get the intersection point of both rotated lines b.

Moving intersection point b by a small constant distance ε to avoid intersecting $P(u,v)$ with either line segment,it becomes a valid bend point for e_i and in total we have $Q_{e_i}^{b^\dagger} \subset Q_{e_i}^{b'} \subset Q_{e_i}^{b}$. As ε decreases towards 0, the point b becomes a minimal bend point for edge e_i by construction. □

Considering convex edges, we can no longer rely on having a single minimal bend point. Hence, we need to refine our notation.

Given edge e_i, let B_{e_i} be the set of all valid minimal bend points. We define $Q_{e_i} = \bigcap_{b \in B_{e_i}} Q_{e_i}^b$ to be the region of P_i *obstructed* by all valid drawings e_i – wherever we place

the bend point of e_i, all the points of Q_{e_i} will be cut off. Conversely, for every point $p \in P_i \setminus Q_{e_i}$, there is a placement of the bend point of e_i such that p is not cut off. An example for the obstructed region and the set of possible minimal bend points for a convex edge can be seen in Fig. 4.2 (d).

Lemma 4.2. *Let e_i be a convex edge to be drawn in polygon P_i. If there is a valid drawing of e_i in P_i, then $B_e \subset \partial V_e$ and we can safely refine P_i by removing Q_e.*

Proof. For any reflex edge the unique minimal bend point is the point on the boundary of that edge's visibility region defining the smallest obstructed region (Lemma 4.1). When considering a convex edge e_i , the boundary ∂V_{e_i} and $P(u, v)$ can conincide in some points or even segments. Our goal is to preserve as much of V_{e_i} as possible for future usage while still refining the polygon. Therefore we first define the set B_{e_i} of all bend points with obstructed regions that are incomparable with respect to containment. Then we compute the region Q_{e_i} obstructed by all these bend points.

Let (u, u') and (v', v) be the segments of $P_i(u, v)$ incident to u and v respectively. Consider the ray starting at u and going towards u', coninciding with $\overline{uu'}$. Rotate this ray in counterclockwise order until it hits V_{e_i} for the first time; call this point p_u. Do the same with the ray from v to v', rotating it clockwise; let the point where it hits V_{e_i} be p_v. To identify all points belonging to B_{e_i}, consider the outline piece of the visibility region between p_u and p_v – namely $V_{e_i}(p_u, p_v)$. The piece $V_{e_i}(p_u, p_v)$ is composed of two things: Rays with origin u or v that are tangent to ∂P_i, and parts of ∂P_i itself. The set B_{e_i} is composed of all points on $\partial P_i \cap V_{e_i}(p_u, p_v)$ that are not also on some tangent ray, and of those points on tangent rays that have the largest distance to the origin of that ray.

We now describe how to construct Q_{e_i} using p_u and p_v as intersecting the individual obstructed regions for all possible bend points is infeasible. Using the points and segments constructed above we get that Q_{e_i} is the region inside P_i bounded by the following segments:

$$Q_{e_i} = \overline{vp_v} \circ V_{e_i}(p_u, p_v) \circ \overline{p_u u} \circ P_i(u, v)$$

Notice that this region is not necessarily simple, as tangent rays and parts of $V_{e_i}(p_u, p_v)$ can coincide with parts of ∂P_i.

If there is an edge e_j later in the sequence ($i < j$) of G_I that needs to have it's bend point inside Q_{e_i} then any two drawings of e_i and e_j have to intersect and G_I can not be drawn in P. Thus, cutting away Q_{e_i} is a safe refinement of P_i. □

The set B_e constructed for edge e in Fig. 4.2 (d) contains a single point (to the left), as it is the end point of tangent rays with origins u and v; there is also a segment of $P_i \subset B_e$ – including the segments' end points, as they are endpoints of tangent rays.

The complete algorithm OneBendInPolygon in pseudocode is listed in Algorithm 4.1:

Algorithm 4.1: OneBendInPolygon(Outerplanar Graph G_I, Polygon P)

```
Tree T ← dual of G_I with outer face P
Stack C ← ∅                                   /* to store convex edges */
Set L ← Set of leafs in T
Set B ← ∅                                     /* to store edge-bend point pairs */
node f_n ← node from L                        /* as root for T */
node f_i ← Nil, region Q_e ← Nil               /* iteration variables */
while (L \ f_n) ≠ ∅ do                        /* bottom-up traversal */
    if L contains leaf nodes corresponding to reflex edges then
        pick f_i ∈ L for some reflex edge e
        if V_e = ∅ then return Impossible
        compute optimal bend point b ∈ V_e and region Q_e   /* Lemma 4.1 */
        B.Add({e, b})
    else                                      /* L only contains convex edges */
        pick f_i ∈ L for some convex edge e                 /* Lemma 4.3 */
        if V_e = ∅ then return Impossible
        compute minimal bend points B_e and regions Q_e     /* Lemma 4.2 */
        C.Push({e = (u, v), P(u, v)})         /* for top-down traversal */
    P_i ← P_i \ Q_e                                         /* refine P */
    T ← T \ f_i                                             /* prune T */
    Remove f_i from L, check if parent p(f_i) is now a leaf
while C ≠ ∅ do                                /* top-down traversal */
    {e = (u, v), P_old(u, v)} ← C.Pop()
    P ← u ∘ P(v, u) ∘ v ∘ P_old(u, v)         /* re-expand P, Lemma 4.4 */
    recompute minimal bend points B_e
    if B_e ∩ P = ∅ then return Impossible     /* later refinement */
    point b ← bend point from B_e ∩ P B.Add({e, b})
    P ← P \ Q_e^b                                           /* refine P */
return B
```

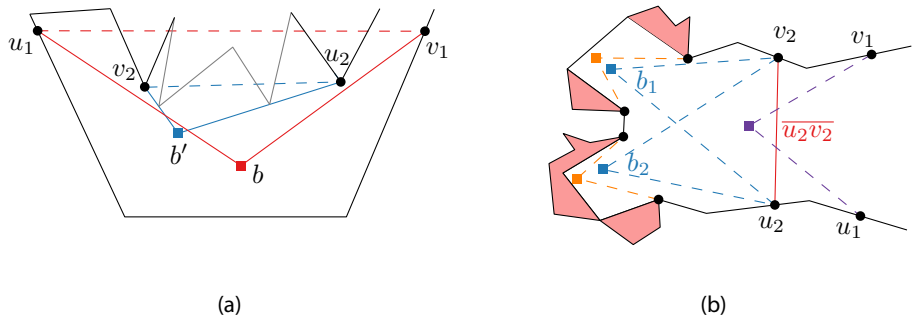


Figure 4.3: (a) The gray part of $P(u_2, v_2)$ forces b' to be placed inside $R^b_{(u_1,v_1)'}$ inducing an intersection of (u_1, v_1) and (u_2, v_2). For that to be necessary, b' must create a reflex angle. (b) Two edges $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$ with $p(e_2) = e_1$. The two possible bend points b_1, b_2 of edge e_2 are each placed in the restricted region of one of children of e_2. Fixing the one-bend drawing of e_1 to intersect $\overline{u_2v_2}$ makes e_2 become a reflex edge, eliminating any choice.

4.4 Correctness

We now proceed to show correctness of OneBendInPolygon (Algorithm 4.1). As described above, it consists of two while-loops: the first one represents the bottom-up traversal whereas the second one represents the top-down traversal. Processing the edges and safely refining the polygon accordingly is treated in Lemma 4.1 and Lemma 4.2 respectively: In step i of the first loop, our algorithm computes V_{e_i}. If we have a step with an empty visibility region, it is impossible to draw G_i in P_i, so by the invariant it is also impossible to draw G_I in P and the algorithm stops. Otherwise, the algorithm computes Q_{e_i} and creates $P_{i+1} = P_i \setminus Q_{e_i}$. If an edge e_j $(j > i)$ had to place its bend point inside Q_{e_i}, then e_i and e_j would cross, independently of the choice of the bend point of e_i; in this case, our algorithm concludes that it is impossible to draw G in P when processing e_j.

In the following, we focus on the top-down traversal: We show that G_{i+1} can be drawn in P_{i+1} if and only if G_i can be drawn in P_i.

We first analyse the possible sequences in which sibling-sets consisting of convex edges can be processed. Without being able to fix the "best" drawing of a convex edge, we need to argue that there is no bad sequence for picking the nodes of convex edges. To do so, we define $R_{e_i} = \left(\bigcup_{b \in B_{e_i}} Q^b_{e_i}\right) \setminus Q_{e_i}$ of convex edge e_i to be the region of P_i *restricted* by e_i: For each point $r \in R_{e_i}$, there are two minimal bend points b and b' for e_i such that bending e_i at b cuts off r, whereas bending e_i at b' does not.

Lemma 4.3. *Let $\mathcal{S}(f)$ be the set of all nodes in T with parent f that belong to convex edges. For any pair of edges $e_1, e_2 \in \mathcal{S}(f)$ and polygon P to draw inside, the restricted regions R_{e_1} and R_{e_2} are interior-disjoint.*

Proof. Let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ be two convex edges with parent node f and let f correspond to edge $e = (u', v')$. Consider the three pieces $P(u', v')$, $P(u_1, v_1)$ and $P(u_2, v_2)$ defined by the parent and it's two children. Since e_1 and e_2 are siblings below f, they are "next to" each other along $P(u', v')$ – that is, we get $P(u_1, v_1) \subset P(u', v')$ and $P(u_2, v_2) \subset P(u', v')$ but $P(u_1, v_1) \cap P(u_2, v_2) = \emptyset$.

For the two restricted regions R_{e_1} and R_{e_2} to possibly overlap, the piece $P(u', v')$ needs to be "folded" to have e_1 and e_2 oppose each other along P – e.g. as in Fig. 4.3 (a): Since both edges are convex, their visibility regions – and thus also their restricted regions – live between $P(u', v')$ and $\overline{u_1 v_1}$ or $\overline{u_2 v_2}$ respectively. W.l.o.g. assume that $\overline{u_2 v_2}$ is between $\overline{u_1 v_1}$ and $P(u', v')$. Hence, the structures of $P(u', v')$ that u_2 and v_2 are mapped to influence the shape of the visibility region V_{e_1} – in the worst case $\overline{u_2 v_2}$ is part of ∂V_{e_1}. The only reason for e_2 to be drawn in a way that it's bend point would be inside R_{e_1} is that some part of $P(u_2, v_2)$ intersects $\overline{u_2 v_2}$ – see the gray piece in Fig. 4.3 (a). This would imply that e_2 is a reflex edge, a contradiction. □

While bend points for reflex edges can be fixed immediately, convex bends have to remain undecided until the root of the dual tree is reached. Given a convex edge e_i encountered in step i of the bottom-up traversal and some later edge e_j (with $i < j$), two events can have impact on how e_i will be drawn:

- If e_j is a reflex edge – hence encountered later in the bottom-up traversal – it might require it's unique best bend point to be placed inside R_{e_i};
- if e_j is also a convex edge, it will be fixed before e_i in the top-down traversal. The bend point of e_j might need to be placed inside R_{e_i} – see edge (u_2, v_2) in Fig. 4.3 (b).

As all restricted regions can possibly be subject to further refinement until encountered again during the top-down traversal. To reconstruct the situation of step i while incorporating all refinements, for any convex edge $e_i = (u, v)$, we stored the piece $P_i(u, v)$ on stack $\mathcal{C}$. When f_n is eventually encountered, all safe refinements of nodes below it have been removed from P, creating the final polygon P_n; it incorporates all refinements necessary to draw any reflex edges and also the obstructed region of the last convex edge encountered. This obstructed region might in turn contain the visibility regions of other yet to be fixed convex edges. Assuming that we encountered and stored c convex edges, we create c additional *expanded* polygons $P_{n+1}, \ldots, P_{n+c}$; for each stored convex edge (e, v), we create P_j from P_{j-1} be replacing $P_{j-1}(u, v)$ with the stored piece. During the bottom-up traversal, we kept the part of P_i bound to containing root f_n; in the top-down traveral, this is also reversed – when given a choice, we keep the part of the polygon containing the next convex edge e_j – in Algorithm 4.1 this edge is stored as $\mathcal{C}$.Top. We get the following lemma:

Lemma 4.4. *Let $e_j = (u, v)$, $P_{old}(u, v)$ be the next convex edge-piece pair to process and P_{j-1} be the current refined polygon. We can obtain the simple polygon P_j to draw e_j into by replacing $P_{j-1}(u, v)$ with $P_{old}(u, v)$.*

Proof. We first prove that the current polygon's boundary ∂P_{j-1} contains both vertices u and v. Then we show that P_{j-1} can safely be expanded to P_j.

If there is any sub polygon containing only one of the two vertices, there must have been some edge separating them – with u on one side and v on the other. Then this edge and (u, v) must cross in any outer drawing, hence contradicting outerplanarity of the input graph. Hence, if they are both cut off of P_{j-1}, both must have been cut off by edge e' at the same time. This edge then separates the polygon into two pieces, both having a boundary partially composed of the drawing for e' and with one of them containing u and v; choose the boundary of this subpolygon as ∂P_{j-1}.

By definiton the vertices u and v subdivide boundary of P_{j-1} into two pieces – see Equation (4.1).

$$P_{j-1} = u \circ P_{j-1}(u, v) \circ v \circ P_{j-1}(v, u) \tag{4.1}$$

$$P_j = u \circ P_{\text{old}}(u, v) \circ v \circ P_{j-1}(v, u) \tag{4.2}$$

$$= P_{j-1} \cup u \circ P_{\text{old}}(u, v) \circ v \circ \underbrace{P_{j-1}(u, v)}_{\text{reversed}} \tag{4.3}$$

To obtain P_j from P_{j-1}, we replace piece $P_{j-1}(u, v)$ with the stored piece $P_{\text{ol}}(u, v)$ as in Equation (4.2). By processing the nodes of dual tree T in the order described above we know that all refinements that happened between the step when edge e was stored and the current step j are either part of piece $P_{j-1}(v, u)$ or cut off by that piece. Any refinement made inside piece $P_{\text{old}}(u, v)$ – not cutting it off completely – would contradict that the node corresponding to edge (u, v) being a leaf at the time. To see that P_j is free of self-intersections, notice that we expanded P_{j-1} by effectively joining it with the region described in Equation (4.3); any segment of $P_{j-1}(v, u)$ intersecting $P_{\text{old}}(u, v)$ must also have intersected $P_{j-1}(u, v)$.

With P_j being a simple polygon and edge $e = (u, v)$ being a leaf in the corresponding dual tree, we can now recompute the visibility region V_e. □

During step j of the top-down traversal OneBendInPolygon will check and (possibly) draw edge e_j without looking back or ahead. In Lemma 4.4 we established that not looking back is safe. For not looking ahead we have the following lemma.

Lemma 4.5. *Let $e_j = (u, v)$ be the convex edge to be drawn in P_j, let V_{e_j} be the visibility region of e_j, and let $\mathcal{S}(e_j)$ be the set of all not yet fixed convex children on e_j in T. If there is a valid drawing for all edges in $\mathcal{S}(e_j)$ inside their corresponding polygon before refining it by drawing e_j, then there still are valid drawings for all edges after drawing e_j.*

Proof. The edges in $\mathcal{S}(e_j)$ are siblings in T; hence, by Lemma 4.3 their respective restricted regions are area-disjoint. Pick a valid bend point b for e_j and assume that $b \in R_{e'}$ for some edge $e' \in \mathcal{S}(e_j)$. As e_j has only one bend point to place and as the restricted regions are disjoint, e' is the only edge affected by the choice of b – see edge (u_2, v_2) with two different possible bend points b_1 and b_2 in Fig. 4.3 (b). Notice that b can not be in the obstructed region $Q_{e'}$ as this region was subject to refinement, is not part of the current polygon P_j and hence can not be part of V_{e_j}. Next consider the possible drawing of e' using bend point b'; assume that placing b cuts off b' from P_j and thus also from the polygon corresponding to e'. By the definition of the restricted region $R_{e'}$ for e' we know that $b' \in R_{e'}$ and therefore, we can find an alternative bend point $b^* \in R_{e'}$ to bend edge e' at later. □

Combing the results above yields the following lemma.

Lemma 4.6. *If the visibility region for the current edge is non-empty, then G_{i+1} can be drawn in P_{i+1} if and only if G_i can be drawn in P_i.*

Proof. During the bottom-up traversal, we refine the current polygon, only making it smaller in every step. Hence whenever we have $V_{e_i} = \varnothing$ for any edge e_i inside P_i, we stop as refining further cannot increase the size of any visibility regions. Otherwise, we either compute the unique best bend point b (Lemma 4.1), or the obstructed region Q_{e_i} as described in Lemma 4.2, safely refining P_i to P_{i+1} accordingly. Since minimal bend points cannot lie in restricted regions of siblings (Lemma 4.3), only the bend point of the edge corresponding to $p(e_i)$ can possibly be placed in the restricted region R_{e_i} of e_i; therefore, any edge drawn in the bottom-up traversal is correct and safe.

During the top-down traversal, we can encounter overlapping restricted regions, but only when the nodes are in a parent-children relationship (Lemma 4.3. Then by Lemma 4.5, when there is a bend point placement for the current edge e_j, that is valid for the current refined polygon, this choice cannot have impact on decissions later in the traversal; that is because any other bend point that lies inside $R(e_j)$ must lie on the opposite side of the drawing of $p(e_j)$ (Lemma 4.4), so it cannot influence the choice of the bend point for e_j. □

We are now ready to state the main result of this chapter.

Theorem 4.7. *Given an outerplanar graph $G_I = (V_I, E_I)$, a polygon P with p corners, and a mapping of the vertices of G_I to ∂P, we can decide in $O(|V_I| \cdot p)$ time whether G_I can be drawn in P with at most one bend per edge.*

Proof. We use OneBendInPolygon as described in Algorithm 4.1. The correctness follows immediately from Lemma 4.6.

The most expensive part of the algorithm in terms of runtime is to compute the visibility region V_e for all the edges $e = (u, v)$. Since V_e is a simple polygon with at most $2p$ edges, it can be computed in $O(p)$ time [Gil14, page 15]. Doing two traversals, the visibility region of each edge needs to be computed at most twice; as outerplanar graphs can

have at most an amount of edges linear in the number of vertices, all these regions can be computed in $O(|V_I| \cdot p)$ total time. The remaining parts of the algorithm (computing the dual graph of G, choosing the order of the faces f_i in which we traverse the graph, computing Q_e, "cutting off" parts of P, and propagating the graph at the end to fix the presentation) can clearly be done within this time.

Thus, the total running time is $O(|V_I| \cdot p)$. □

4.5 Conclusion

In this chapter, we have developed an algorithm that is able to an draw outerplanar graph into a predefined simple polygon when allowed one bend per edge. When no such drawing exists, our algorithm reports the edges that enforce the crossing. This can then be used to identify the malformed segments of the boundary – forcing the bend points, enforcing the crossing – using the dual tree of the drawing. We phrased the problem as an partial representation extension problem, where the outer face is bounded Hamiltonian cycle that is pre-drawn. The task then was to add the missing chords, bending each missing edge at most once and only inside the drawing.

This technique was designed with the application in mind that a fixed drawing of some planar subgraph is already given that then needs to be completed, respecting the shapes of the prescribed faces. Considering that the mapping of the vertices to the boundary is part of it's input, our algorithm can be used to accomplish this goal – or report that no such drawing extension exists.

Acknowledgments. This work was initiated at the Workshop on Graph and Network Visualization 2019. We thank all the participants for helpful discussions and Anna Lubiw for bringing the problem to our attention.

Part II

Drawing Vertices using Integer Coordinates

Chapter 5

Moving Graph Drawings to the Grid Optimally

From a graph drawing perspective, restricting the vertex coordinates to be of integer precision can be desireable for various reasons. These reasons can stem from aesthetics, computational complexity, technical limitations or an application. While parts of this chapter were already submitted, reviewed and presented as part of a Master's Thesis [Löf16], we include it in this work for the sake of completeness: In the following two chapters, we consider the problem Topologically-Safe Grid Representation. This chapter covers the theoretical results on Topologically-Safe Grid Representation and presents an exact algorithm that finds optimal solutions to an $\mathcal{NP}$-hard problem. This motivates the results we will present in Chapter 6. With a geographic application in mind – representing road networks and administrative borders at finite precision –, we use the foundation laid out in this chapter to design and evaluate a randomized heuristic approach to solving Topologically-Safe Grid Representation.

Concepts. In the computational geometry community, a process called *snap rounding* has been proposed for line arrangements and has since become well-established. Let the euclidean plane be tiled into unit squares called pixels with center on integer coordinates. Let $\mathcal{S}$ be a finite collection of line segments $s \in \mathcal{S}$ in the plane and let $\mathcal{A}(\mathcal{S})$ be the *arrangement* of vertices, edges and faces in the plane induced by the segments and intersections of $\mathcal{S}$. Guibas and Marimont [GM98] formaly define the snap rounding paradigm as follows:

Definition 5.1. *Snap rounding* is the process of converting the arbitrary precision arrangement $\mathcal{A}(\mathcal{S})$ into a fixed-precision representation $\mathcal{A}^*(\mathcal{S}^*)$ with these properties:

(1) Fixed-precision: All vertices of $\mathcal{A}^*$ are at integer precision coordinates.

(2) Geometric similarity: For each $s \in \mathcal{S}$, the approximation s^* lies within the Minkowski sum of s and a pixel at the origin.

(3) Topological similarity: $\mathcal{A}$ and $\mathcal{A}^*$ are "topologically equivalent up to the collapsing of features" – that is, there is a continuous deformation of the segments in $\mathcal{S}$ to their snap-rounded counterparts such that no segment ever passes completely over a vertex in the arrangement.

Designed to overcome problems induced by working with infinite-precision real arithmetic machines (RAMs) [Sha78, dBHO07], this technique can also be used for limited display resolutions, such as bitmap graphics. There are several algorithms for computing such a representation that are fast and work well in practice; we present a brief survey on challenges and solutions in Section 5.1.

The concepts common to all approaches stem from the line intersection problem as stated by Greene and Yao [GY86]: Given an arrangement of line segments, each pixel that contains vertices or intersections is called *hot*. Then every segment becomes a polygonal chain whose edges (*fragments*) connect center points of hot pixels, namely those that the original segment (*ursegment*) intersects. Guibas and Marimont [GM98] showed that during snap rounding, vertices of the arrangement never cross a polygonal chain, so after snapping no two fragments cross. Moreover, the circular order of the fragments around an output vertex is the same as the order in which the corresponding ursegments intersect the boundary of its pixel. The resulting arrangement approximates the original one in the sense that any fragment lies within the Minkowski sum of the corresponding ursegments and a unit square centered at the origin. But by definition, the output of those algorithms is not topologically safe: vertices, edges or even faces can disappear from the output drawing while rounding.

Contribution. We investigate the problem of moving the given drawing of a planar graph to a given grid, priorizing the ability to recognize the original graph over geometric similarity: While we do not tolerate new point-point or point-line incidences in the rounded drawing, we accept the possibility that a vertex does not go to the nearest grid point. Presented this set of requirements, or objective is to minimize the change induced on the vertex positions. This change can by measured, for example, by the sum of the distances or the maximum distance in the Euclidean (L_2-) or Manhattan (L_1-) metric. We define our variant of this problem – Topologically-Safe Grid Representation – in Section 5.2 and show that it is $\mathcal{NP}$-hard; To do so, we giva a reduction that asks for compressing each coordinate by just a single bit. The proof is somewhat similar in concept to the proof of the $\mathcal{NP}$-hardness of Metro-Map Layout [Nöl05, Wol07], but requires additional constructions since the rounding problem does not easily allow the construction of "rigid" gadgets. In Section 5.3, we propose an integer linear program (ILP) to solve instances of Topologically-Safe Grid Representation to optimality; Our ILP formulation generalizes the ILP for Metro-Map Layout by Nöllenburg and Wolff [NW11] in the following way: Where the ILP by Nöllenburg and Wolff assumes a constant number of possible edge directions (namely 8) to obtain an octilinear drawing, our desired output asks for a number of edge directions that is quadratic in the size of the grid; on a grid of size $k \times k$, there are $\Theta(k^2)$ edge directions. The numbers of variables and constraints of our ILP are polynomial in grid size and graph size, but are quite large in practice. Thus, for an n-vertex planar graph, we must generate $O(k^2n^2)$ constraints, among others, to preserve planarity and the cyclic order of edges around the vertices. To ameliorate this, we apply delayed constraint generation, a technique used to add certain constraints only when needed. Still, runtime of our ILP is prohibitive for graphs with more than about

15 vertices – we give a set of example-instances in $2D$ in Section 5.4 to show the power and limitations of our exact approach. Our techniques can in theory also be adapted to draw (small) graphs with minimal area. This is interesting even for small graphs since minimum-area drawings can be useful for validating (counter)examples in graph drawing theory.

5.1 Related Work

5.1.1 Rounding to the Grid

Greene and Yao [GY86] considered the precision required to store the points created when intersecting line segments as it has numerous applications in computational geometry. Reducing the precision used to store these intersection points can lead to artificial *extraneous intersections*, that are not part of the originial arrangement – see Fig. 5.1. Initially, these intersections have been handled by repeatedly running an intersection detection algorithm – for example the Bentley-Ottmann sweep [BO$^+$79b] –, until no new intersections are reported. The algorithm given by Greene and Yao does not prevent extraneous intersections from occuring, but finds and rounds all intersections in one single iteration. An example output of their algorithm can be found in Figure 5.1 (c).

The approach by Hobby [Hob99] considers *tolerance squares* – unit-length pixels centered on integer grids in which segment endpoints and intersections occur. Edges passing through tolerance squares get subdivided, then every vertex in the square gets snapped to the center. While this may introduce new incidences, it avoids extraneous intersections – see Fig. 5.1 (d).

Guibas and Marimont [GM98] give a boiled down definition of snap rounding and state a dynamic algorithm based on vertical cell decompositions; Having n unrounded segments, a set $H = \{h_1, \ldots, h_{|H|}\}$ of tolerance squares, complexity of the cell decomposition C and complexity of the arrangement A, their algorithm has an output-sensitive asymptotic runtime of $O(n \log n + A + \sum_{h \in H} |h|^2 + C)$.

Goodrich et al. [GGHT97] give two simplified algorithms: The first algorithm is deterministic and based on the Bentley & Ottmann sweep [BO$^+$79b], the other algorithm is randomized using trapezoidal decomposition. Both have a matching runtime of (expected) $O(n \log n + \sum_{h \in H} |h| \log n)$, independent of arrangement complexity A.

The precision required to store vertex coordinates and to measure vertex-to-vertex distances can be bounded by grid resolution, but measuring distances between nonincident vertex-edge pairs may still require arbitrary precision. Rounding the endpoints of segments induces *drift* on the segment itself. Drift can cause a rounded segment to pass through a tolerance square it did not pass in it's unrounded state; this is demonstrated by edge e_1 in Fig. 5.2 (a) and (b). To overcome this, Halperin and Packer [HP02] augment the classic snap rounding procedure by iterating the process. This *iterated snap rounding* procedure gives an output that is equivalent to that of repeatedly applying the tolerance square-based rounding process until for all pairs of vertices and nonincident

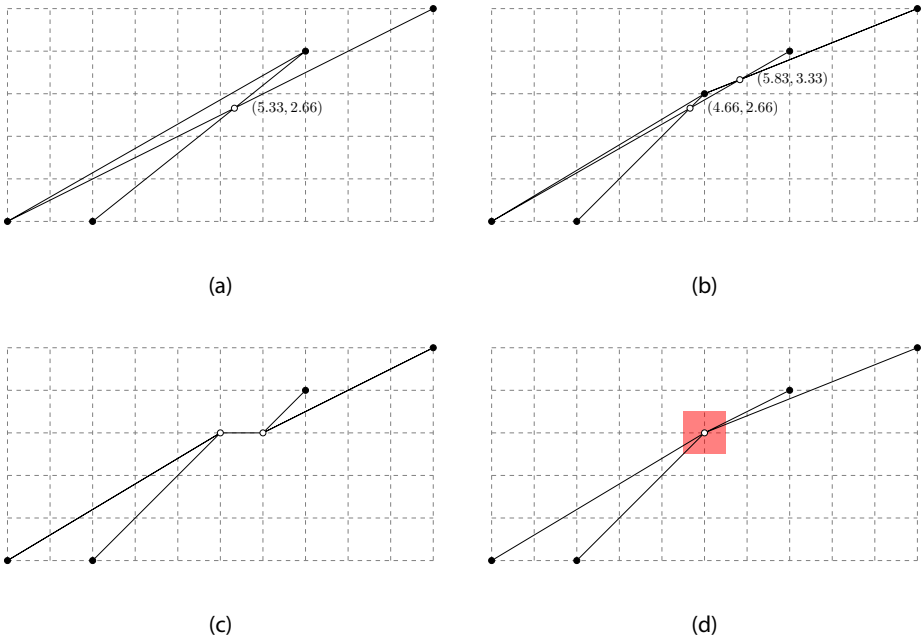


Figure 5.1: Rounding the intersection point of two line segements: (a) An example instance on four vertices and three edges, intersection point depicted as white vertex, (b) rounding the intersection to the nearest integer grid point creates two extraneous intersections. The results obtained by using the algorithms of (c) Greene and Yao and (d) Hobby (tolerance square in red).

edges, the distance is at least half the width of a pixel. Applying this to the above example – 5.2 (c) – edge e_1 gets subdivided during the second iteration and both parts subsequently become incident to the snapped vertex. Subdividing segments and rounding the pieces can imply additional drift induced by consecutively intersecting other tolerance squares, heavily deforming the output arrangement. Effort to bound the drift was made by Packer [Pac06], adding a user-specified parameter. An implementation of iterated snap rounding for $2D$ arrangements can also be found in the CGAL computational geometry framework [Pac19].

De Berg et al. [dBHO07] extend the original list of desired properties adding *non-redundancy*. A degree 2 vertex of the output is redundant if it does not stem from a segment endpoint; consider the white vertex in Fig 5.3 (b). They give an algorithm that outputs a snap-rounded arrangement in which all redundant vertices are removed by using a second vertical sweep. This algorithm has a total runtime of $O((n + I) \log n)$ with I being the number of intersections in the arrangement.

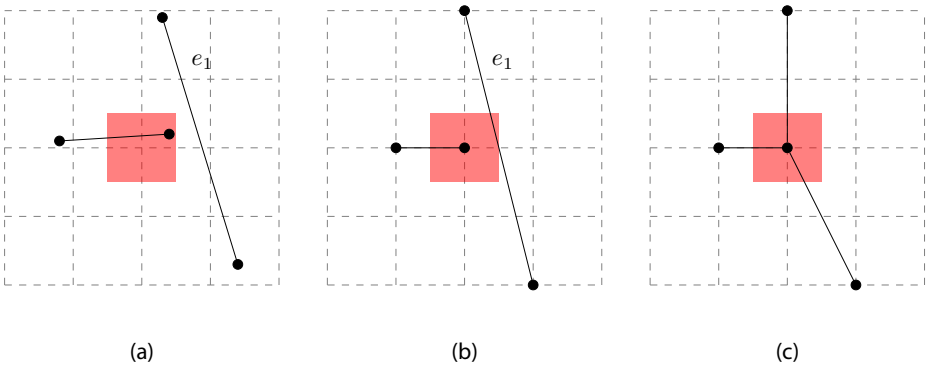


Figure 5.2: Iterated snap rounding: (a) Input segments at arbitrary precision, edge e_1 does not intersect the red tolerance square. (b) Edge e_1 intersects the tolerance square after snap rounding, requiring another iteration. (c) Resulting arrangement after iterated snap rounding, edge e_1 subdivided.

Hershberger [Her13] introduced *stable snap rounding*. Algorithms based on tolerance squares can be made sable in the following sense: the rounded arrangement does not change when re-applying the procedure. This is obtained by providing individual rules for two different types of tolerance squares, requiring $O(|H| \log n)$ additional runtime.

Most recently, rounding of arrangements in 3D has recently been studied by Devillers et al. [DLL18]. Rounding specific classes of graphs, such as Voronoi diagrams [DG02] have also been considered from a computational geometry perspective and with similar objective in mind.

5.1.2 Drawing on the Grid

From a graph-drawing perspective, restricting vertex coorinates to integer precision is a common practice. Fáry [Fár48] (among others) shows that every planar graph has a planar straight line embedding with vertices as points on the plane; this is also known as a *Fáry embedding*. Tutte [Tut63] introduces the barycenter method for drawing planar graphs. It yields drawings that need precision linear in the size of the graph.

Dolev et al. [DLT83] introduce the family of planar *nested triangles graphs* – see Fig. 5.4. Nested triangles with n vertices can be used to prove that $(2n/3-1) \times (2n/3-1)$ is a lower bound on the required area when restricting straight line drawings on the integer grid.

Motivated by these results, Schnyder [Sch90] and, independently, de Fraysseix et al. [dFPP90] have shown that any planar graph with n vertices admits a straight-line drawing on a grid of size $O(n) \times O(n)$ and that this is asymptotically optimal in the worst

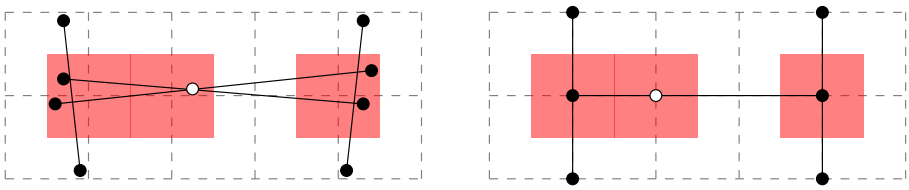


Figure 5.3: Redundant degree 2 vertices in rounded arrangements: (a) input segments (redundant intersection marked as white vertex), (b) output with (white) degree 2 vertex that is not a segment endpoint.

case. Chrobak & Nakano [CN98] have investigated drawing planar graphs on grids of smaller width, at the expense of a larger height.

Krug & Wagner [KW08] give a reduction from 3-Partition, showing that area minimization of straight line grid drawings is $\mathcal{NP}$-hard. They also give an iterative algorithm that computes a more compact drawing of a given plane graph.

Nöllenburg & Wolff [NW11] give a mixed integer program for octilinear metro-map drawings with station labels. They establish sets of hard and soft constraints to create a visually pleasant map drawing for answering navigational questions while not preserving real-world distances or travel times. While solving a very special problem, their model be adapted for other geometric tasks – we will do this in Sectio 5.3.

In relation to that, Biedl et al. [BBN+13] propose a generic ILP model for various grid-based layout problems, such as determining pathwidth, finding optimum s-t-orientations or bar k-visibility representations.

5.2 NP-Hardness

In the following we will consider a rounding task that relexes on geometric similarity of the output while enforcing topological equivalence; see properties (2) and (3) of Definition 5.1. We first give a formal definition of this new problem that we call Topologically-Safe Grid Representation and then proceed to show $\mathcal{NP}$-hardness.

Definition 5.2 (Topologically-Safe Grid Representation)**.** As *input* we take a plane graph $G = (V, E)$ with vertex positions of arbitrary precision and a rectangular bounding rectangle $B = [0, X_{\max}] \times [0, Y_{\max}]$.

The Topologically-Safe Grid Representation problem is to find an drawing Γ of G the with the following properties:

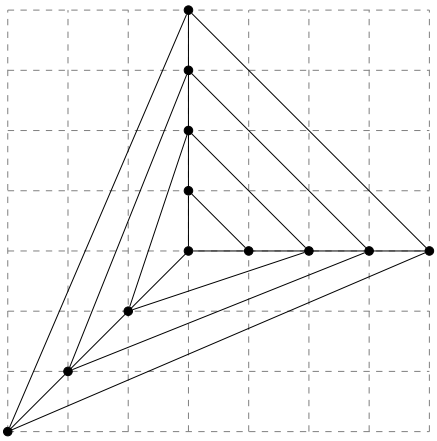


Figure 5.4: Nested triangles graph on $n = 12$ vertices

(1) All vertices of G are moved to integer coorindates that are contained inside B, and

(2) Γ is topologically equivalent to the given plane straight-line drawing of G.

(3) The total *displacement* of Γ is minimal over all such drawings: the displacement of a single vertex is the Manhattan-distance between it's original position and the rounded coordinate, and the displacement of a drawing is the the sum of over all vertex displacements.

We prove $\mathcal{NP}$-hardness of Topologically-Safe Grid Representation by considering the decision variant. Instead of searching for the smallest total displacement, we ask for a drawing with some constant displacement c. Having an algorithm to answer this question, we can use it to search for the smallest such constant $c_{\min}$. Since the searching can be done using a polynomial amount of querries, proving hardness for the decision variant also imples hardness of the original problem.

We reduce from Planar Monotone 3SAT. A formula F for this variant of 3SAT (recall from Section 2.2.2) has the following additional properties:

- The graph $H(F)$ induced by the formula – using variables and clauses as vertices and having an edge for every occurance – is *planar*.
- All clauses are *monotone* – the variables of a clause are either all negated or all unnegated.

The Planar Monotone 3SAT problem is known to be $\mathcal{NP}$-hard [dBK12]. We can assume that the graph $H(F)$ can be laid out as shown in Fig. 5.5 (a): all vertices corresponding to variables lie on the x-axis with the vertices of the all-negated clauses above them and the vertices of all-unnegated clauses below them.

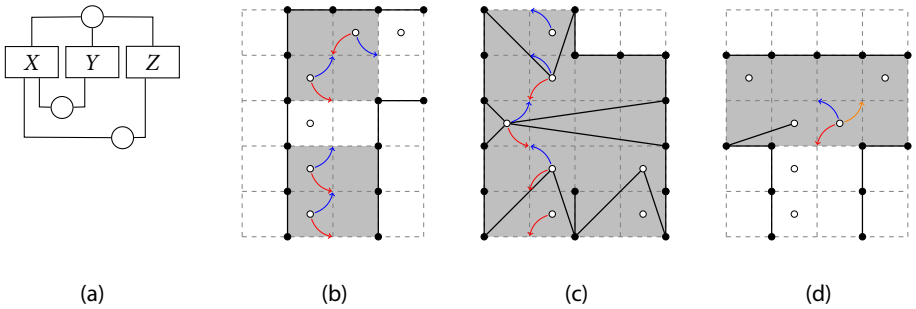


Figure 5.5: (a) Graph $H(F)$ for formula $F = (\overline{X} \vee \overline{Y} \vee \overline{Z}) \wedge (X \vee Y) \wedge (X \vee Z)$; (b–e) All gadgets used in the reduction. Inner area of each gadget highlighted gray, possible roundings indicated by arrows. (b) Vertical line gadget (bottom) connected to corner gadget (top), (c) variable gadget (with two negated and one unnegated occurrences), and (d) all-negated clause gadget with three negated variables.

Theorem 5.1. *Topologically-Safe Grid Representation is $\mathcal{NP}$-hard.*

Proof. For a given monotone, planar 3-CNF formula F, we construct a cost bound $c_{\min}$ and a plane graph G with vertices at half-integer coordinates – storing half-integers requires an additional bit compared to regular integers. The sum of all vertex movements induced by rounding G to integer coordinates is exactly $c_{\min}$ if and only if F is satisfiable. To achieve this, we introduce gadgets to resemble the elements of the formulas incidence graph $H(F)$ – variables, clauses, edges and bends – and construct G and $c_{\min}$ in polynomial time.

For exposition, we draw the vertices of G using two different styles. Black vertices start on integer grid points and do not need to be rounded. Moving a black vertex to another integer grid point is allowed, but we will show that this is not optimal if F is satisfiable. White vertices start at grid cell centers – using half-integer coordinates – and thus will always move at least $|\pm 0.5| + |\pm 0.5| = 1$ by rounding both coordinates. Let $W \subseteq V(G)$ be the set of white vertices. Now we give the construction of the various gadgets.

To get started, we introduce the line and bend gadgets. They are used to consistently transport the variable assignments to the clause gadgets. Every segment of the line gadget consists of four black vertices and two edges forming a *tunnel*, and a single white vertex inside; see Fig. 5.5 (b), lower half. Each white vertex can be rounded most cheaply – at cost 1 – to exactly two possible integer grid points inside the tunnel, depicted by the red and blue arrows. Consider the white vertices of two neighboring grid cells inside a tunnel. They share exactly one grid point that is not occupied by black vertices of the tunnel's wall. Rounding one of the white vertices to that grip point prevents the other vertex from also going there; with only one safe option of cost 1 left, the other white vertex has to mimic the movement of the first vertex. So, if the white vertex at one end

of a tunnel is rounded inward (blue arrow) the white vertex at the other end of that line must be rounded outward – we say it is *pushed*. The same machinery works for the bend gadgets, as can be seen in Fig. 5.5 (b), upper half.

Next, consider the variable gadget depicted in Fig. 5.5 (c). It can be extended horizontally to have tunnels for vertical line gadgets for every negated and unnegated occurrence at the top and bottom respectively. Inside the left-most of the center cells of this gadget, there is a white *assignment* vertex. The assignment vertex is connected to the gadget's walls by four edges forming two triangles and can be rounded up or down representing the state of the variable. Rounding the assignment vertex to either corner of its cell forces a pair of those triangle edges to coincide with a line of the grid, blocking grid points on the top or bottom tunnels, respectively. The tunnels connected on that side of the gadget are then all forced to push inwards and into the connected clause gadgets. Following the layout of Fig. 5.5 (a), we say that a variable is set to true, if the assignment vertex is rounded upwards (following the blue arrow) and false, if rounded downwards (following the red arrow).

Finally, the clause gadget is shown in Fig. 5.5 (d). In the following, we only discuss the all-negated degree-3 version of this gadget. The degree-2 version can be constructed by replacing the vertical tunnel by a black vertex at the tunnels entrance and extending the walls on both sides, connecting them to the new black vertex to form a tunnel. The all-unnegated versions of this gadget can be obtained by mirroring the all-negated versions at a horizontal line. At the center of the gadget, there is a white *satisfaction* vertex that can go to any of three possible integer grid points (or two respectively) at equal cost. These grid points belong to the three line gadgets and are only available if the corresponding line does not transmit a push. Then the satisfaction vertex can be rounded at cost 1 if and only if the clause is satisfied.

All white vertices must be rounded at cost at least 1. Thus, the rounding cost of G is bounded from below by $c_{\min} = |W|$. If F is satisfiable, there is a rounding of all vertices of W to achieves this: Round the assignment vertices according to a satisfying assignment of the variables. Then the line and corner gadgets between a clause and the (at least) one variable satisfying it does not transmit a paush. This in turn allows the corresponding satisfaction vertex to be rounded towards the entrance of that line. All rounded white vertices contribute cost of exactly 1. In the other direction, a satisfying assignment can be read off from the assignment vertices if rounding occurred at cost $c_{\min}$.

For all three candidate grid points of a satisfaction vertex to be unavailable, all line gadges connected to the clause must be forced to transmit pushes; hence, none of the variables occuring in this clause are assigned to satisfy this clause. In our construction, this shows as three variable-tunnel-clause triples. In each such triple, all integer points are occupied by white vertices, leaving no grid point for the satisfaction vertex of the clause. A topologically valid rounding of such a triple then must involve moving a black vertex, making the original position of the black vertex available by broadening the tunnel. Since all white vertices still need to be moved by at least 1, additing the cost of moving the black vertex makes G exceed the cost bound $c_{\min}$. That is, if $c_{\min}$ is exceeded, then F

is unsatisfiable: any rounding corresponding to a satisfying truth assignment is cheaper. This concludes our Karp reduction and the claim follows. □

Given that the objective function we intend to optimize is polynomially bounded by the size of the bounding rectangle and the number of vertices in the instance, the following extension of Theorem 5.1 implies that there is no fully polynomial-time aproximation scheme unless $\mathcal{P} = \mathcal{NP}$ [GJ79].

Corollary 5.2. *Topologically-Safe Grid Representation is $\mathcal{NP}$-hard in the strong sense.*

Proof. The only numerical variables in an instance of Topologically-Safe Grid Representation are vertex coordinates. In the proof of Theorem 5.1 the constructed instances have a bounding rectangle of polynomial size and thus coordinate values are limited polynomially as well. Thus the runtime of a hypothetical algorithm remains exponential in input size when unary representations are used. □

We decided to proof $\mathcal{NP}$-hardness of Topologically-Safe Grid Representation using the Manhattan distance as a cost measure, because of the integer linear program we present in Section 5.3. Linearizing Manhattan distance with standard transformations [MS97], it can easily be used as an objective function. The hardness result itself is not limited to these considerations.

Corollary 5.3. *Topologically-Safe Grid Representation is also $\mathcal{NP}$-hard when using Euclidean distance to evaluate rounding costs. In this case it is also $\mathcal{NP}$-hard to minimize the maximum movement $d_{\max}$ instead of the sum over all vertices.*

Proof. The choice of distance measure has impact on the lower bound $c_{\min}$. Using Euclidean distance instead, rounding a white vertex costs at least $\sqrt{0.5^2 + 0.5^2}$ and moving black vertices still costs (at least) 1. We obtain a minimum rounding cost of $c_{\min} = \sqrt{0.5^2 + 0.5^2} \cdot |W|$. As moving black vertices is still not accounted for, the above proof still holds.

Exploiting the fact that $\sqrt{0.5^2 + 0.5^2} < 1$, we can identify satisfiable instances by considering the maximum movement $d_{\max}$ over all vertices of G. Given an instance of Topologically-Safe Grid Representation, a maximum movement of $d_{\max} < 1$ implies that only white vertices have been rounded. Thus, the corresponding formula is satisfiable. For the case of an unsatisfiable formula, at least one black vertex needs to be moved, making $d_{\max} \geq 1$. □

The ability the make the distinction between different maximum movements – 1 versus $\sqrt{0.5^2 + 0.5^2} \approx 0.71$ – based on the satisfiability of F also gives the following.

Corollary 5.4. *Euclidean Topologically-Safe Grid Representation with the objective to minimize maximum movement $d_{\max}$ is $\mathcal{APX}$-hard.*

5.3 Exact Solution using Integer Linear Programming

In this section we provide an exact algorithm for Topologically-Safe Grid Representation by giving an integer linear programm – or ILP for short. In the following, we describe the integer variables necessary for instances of Topologically-Safe Grid Representation to then model the task of finding an optimal solution to Topologically-Safe Grid Representation by giving a linear cost function and several sets of linear constraints to ensure topological equivalence. Our model borrows ideas from a linear program for creating octilinear drawings of metro maps by Nöllenburg and Wolff [NW11], changing and extending them to fit for our requirements.

In the following presentation, we will use upper-case letters to denominate constants, whereas lower-case letters will be variables for the ILP; we further divide the variables, greek characters will be used for binary variables while variables represented by roman characters can take any natural number as value.

5.3.1 Basic Model

Recall that each instance of Topologically-Safe Grid Representation is a graph $G = (V, E)$ with real-valued vertex coordinates $p\colon V \to \mathbb{R}^2$ and a prescribed embedding. For all vertices $v \in V$, we refer to the input coordinates as tuples $p(v) = (X_v, Y_v)$ of real-valued constants. We define the integer-valued coordinates $r\colon V \to \mathbb{N}^2$ with $r(v) = (x_v, y_v)$, thus obtaining two integer variables $0 \leq x_v \leq X_{\max}$ and $0 \leq y_v \leq Y_{\max}$ to represent the rounded output coordinates in our model. The ILP will find an optimal solution by adjusting the values of these variables. This data model naturally leads to the following objective function:

$$\text{Minimize} \sum_{v \in V} |x_v - X_v| + |y_v - Y_v| \tag{5.1}$$

By using the absolute value function, Equation (5.1) is not linear, but can be made so with standard transformations [MS97]. Note that without any further constraints, this would just move vertices to (one of) the nearest integer grid points. Together with a polynomial-time checker for topological equivalence, this already implements the most basic heuristic. We will refer to this heuristic as *Instant Rounding* and build on this idea in Section 5.3.3 – and also later in Section 6.2 of the next Chapter.

5.3.2 Constraints for Topological Equivalence

Unique Vertex Coordinates. Given the represention of coorinates in our model, ensuring that vertices do not coincide can easily be done by adding the constraints of Equation (5.2). They too are not linear as stated, but can also be readily linearized.

$$(x_v \neq x_w) \vee (y_v \neq y_w) \qquad \forall v, w \in V, v \neq w \tag{5.2}$$

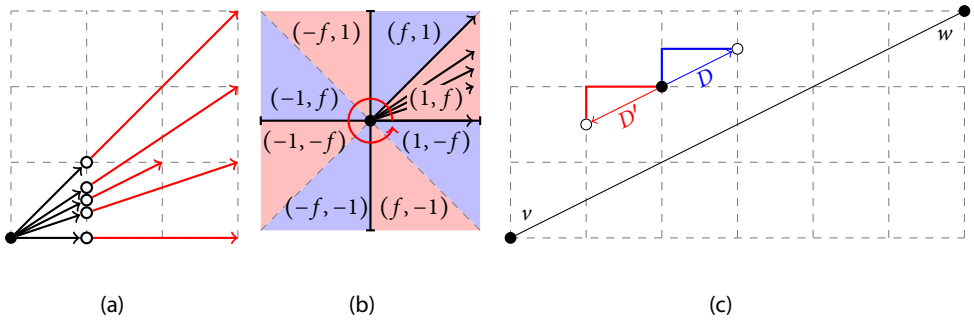


Figure 5.6: Farey sequence and direction assignment: (a) The slopes of the black vectors match the elements of depth 3 in the Farey sequence $\{0, 1/3, 1/2, 2/3, 1\}$, the red extensions point to all grid points of distance at most three within that plane octant. (b) Using the elements of the sequence as coordinates for vector endpoints (either as x or y coordinate and with different signs), we get sets of vectors for all octants; combining all sets, we get set $\mathcal{D}$. Programmatically creating $\mathcal{D}$, we can order all directions radially arround the origin. (c) Edges have two directions assigned to them – one for each endpoint. Direction D (blue) matches the slope of (v, w) as seen from v while D' (red) matches the slope seen from w. D and D' are opposites, which is also represented in the ordering of $\mathcal{D}$.

We will now introduce constraints to ensure that input and output are topologically equivalent; that is, no two edges intersect and the edges at every vertex have the same cyclic order as in the input. To do this, we first introduce the following tool:

Possible directions. The most important departure from the metro-map drawing ILP is about the number of different edge slopes. The goal of Nöllenburg and Wolff was to draw metro maps in the rather classic octilinear style. This restriction is unreasonable for planar graphs in general, hence more than eight different directions are required. A priori the only assumption on the placement of rounded vertices is that they each lie on a grid point somewhere within the given bounding rectangle. Hence, edges in a valid drawing can possibly go from any of those grid points to any other. Let $\mathcal{D}$ be the set of unique directions $D = (D_X, D_Y)$ in $[-X_{\max}, X_{\max}] \times [-Y_{\max}, Y_{\max}]$. To explicitly enumerate all elements of this set, we use the Farey sequence [GKP94]. The Farey sequence F_n recursively enumerates all fully-reduced fractions with denominator less or equal to n. More precisely, F_n contains F_{n-1} and all mediants[1] created from subsequent elements of F_{n-1}; the first three elements of the Farey sequence are shown in Equation (5.3).

$$F_1 = \left\{\frac{0}{1}, \frac{1}{1}\right\}, F_2 = \left\{\frac{0}{1}, \frac{1}{2}, \frac{1}{1}\right\}, F_3 = \left\{\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}\right\} \tag{5.3}$$

[1] Also known as the "freshman sum", the mediant of two fractions $\frac{a}{b}$ and $\frac{c}{d}$ is defined as the sum of the numerators devided by the sum of the denominators $\frac{a+c}{b+d}$.

Picking $n = \max\{X_{\max}, Y_{\max}\}$, we use the n-th element of the sequence F_n to generate the set $\mathcal{D}$ of all possible directions as follows: We use the elements in $f_1, \ldots, f_k \in F_n$ as slopes for a set of vectors; all vectors start at the origin and the j-th vector points towards coordinate $(1, f_j)$. Extending these vectors to travel a distance of n in x-direction, they togehter point to (or pass over) all grid points that are inside the bounding rectangle and the first octant. An illustration for $n = 3$ can be found in Fig. 5.6 (a). Alternating the order and signs of the endpoint's coordinates allows us to create sets of vectors for all eight octants; see Fig. 5.6 (b). From the literature [GKP94] we get that $|F_n| \approx \frac{3n^2}{\pi^2}$ and hence the number of all possible directions inside the bounding rectangle is $|\mathcal{D}| \in \Theta(n^2)$. In the following, we let the set $\mathcal{D}$ be ordered counterclockwise and enumerated using integers, starting at the positive x-axis, allowing comparison of directions by integer identifier.

No two edges cross. The following constraints ensure that nonincident edges do not cross without preventing incident edges from touching in their shared vertex. We will follow the general idea of Nöllenburg and Wolff [NW11]. While producing octilinear drawings of metro maps, they ensured planarity by forcing every pair of nonincident edges to be separated by at least some distance $D_{\min}$ in at least one of the eight octilinear directions. Enforcing this minimum distance of separation was partly an aesthetic guideline, but also guarantees planarity. We employ the same approach for the latter, but as we are not interested in the same aesthetics, some modifications are necessary. Most notably, we allow for more than eight possible edge slopes and thus need to check if a pair of nonincident edges is separated in any direction of $\mathcal{D}$. Enforcing a rather large minimum distance between edges was useful for Nöllenburg and Wolff when labeling the individual stations along the metro lines. We do not have additional labels that we need to reserve space for, therefore we pick the separation distance $D_{\min}$ such that all planar realizations on the grid are allowed; that is, $D_{\min}$ has to be small enough to separate any non-intersecting pair of edges in the output. Considering the possible slopes of an edge and the achievable minimum distance of that edge to the endpoint of some other edge, we choose $D_{\min} = 1/(\max\{X_{\max}, Y_{\max}\} + 1)$ – smaller than the smallest non-zero element in F_n.

For every pair of nonincident edges $e_1, e_2 \in E$ and all directions $D \in \mathcal{D}$, we add a binary variable $\gamma_D(e_1, e_2) \in \{0, 1\}$ to the model. A value of 1 indicates that e_1 and e_2 are apart by $D_{\min}$ in direction D. Every pair of nonincident edges must be separated in some direction (following the idea of [Nöl05]), we need to make sure that at least one of the corresponding γ is set to 1. Hence, we get the set of constraints shown in Equation (5.4).

$$\sum_{D \in \mathcal{D}} \gamma_D(e_1, e_2) = 1 \qquad \forall e_1, e_2 \in E, e_1, e_2 \text{ nonincident} \tag{5.4}$$

To enforce separation of the edges, we impose constraints on the coordinates of the edges' endpoints. We need all constraints to be contained in the model before solving, but as there are opposing directions, we also need to include sets of constraints that are apparently conflicting. Hence we only want the set of constraints to be enforced for which the corresponding γ is set to 1. To model this switch-case scenario, we extend

all necessary equations by a term composed of a multiplication of that γ by some large constant L_γ[2], adding some slack to the contraints we want to have "disabled" during solving. Let $L_\gamma = 2 \cdot \max\{X_{\max}, Y_{\max}\} + 1$. Then we require the following for any direction $D \in \mathcal{D}$, all pairs of nonincident edges e_1, e_2, and all pairs of endpoints $v \in e_1, w \in e_2$.

$$\begin{aligned} D_X \cdot (x_v - x_w) + D_Y \cdot (y_v - y_w) + (1 - \gamma_D(e_1, e_2))L_\gamma \geq D_{\min} \\ \forall D \in \mathcal{D} \quad \forall e_1, e_2 \in E, e_1, e_2 \text{ nonincident} \quad \forall v \in e_1, w \in e_2 \end{aligned} \tag{5.5}$$

Considering a single pair of nonincident edges, the constraint of Equation (5.4) yields a unique direction D with $\gamma_D = 1$. By choice of L_γ, the constraints of Equation (5.5) that involve a direction D with $\gamma_D = 0$ are trivially fulfilled.

Determine direction of incident edges. To prevent two incident edges $e_1, e_2 \in E$ from overlapping, we again generalize the metro-map drawing ILP. Without being restricted to the octilinear drawing style, we can drop the "relative position rule" and assign any direction from $\mathcal{D}$ to either edge. The direction assigned to an edge $e = (v, w)$ is relative to the endpoint of that edge – consider the opposing directions D and D' in Fig. 5.6 (c). By ensuring that the directions assigned to any pair of incident edges differ, we prevent edges those edges from overlapping.

Following the recipe for nonincident edges, we introduce a binary decision variable $\alpha_D(v, w) \in \{0, 1\}$ for every vertex $v \in V$, every vertex $w \in N(v)$ from the neighborhood of v and every direction $D \in \mathcal{D}$. Setting $\alpha_D(v, w)$ to 1 implies that the direction of edge (v, w) is D when considered from endpoint v. To ensure that every edge gets some direction assigned to it, we use the constraints presented in Equation (5.6).

$$\sum_{D \in \mathcal{D}} \alpha_D(v, w) = 1 \qquad \forall v \in V \quad \forall w \in N(v) \tag{5.6}$$

For any vertex $v \in V$, any neighbor $w \in N(v)$, and any direction $D \in \mathcal{D}$, the following ensures that edge (v, w) indeed has direction D and that the position endpoint w matches that direction. Again presented with sets of constraints apparently in conflit, we add conditional slack to the inequalities as we did in Equation (5.5). We do so by using constant $L_\alpha = 2 \cdot \max\{X_{\max}, Y_{\max}\} + 1$.

$$\begin{aligned} x_w \cdot D_Y + y_v \cdot D_X - x_v \cdot D_Y + (1 - \alpha_D(v, w))L_\alpha \geq y_w \cdot D_X \\ x_w \cdot D_Y + y_v \cdot D_X - x_v \cdot D_Y - (1 - \alpha_D(v, w))L_\alpha \leq y_w \cdot D_X \\ (1 - \alpha_D(v, w))L_\alpha + (x_w - x_v) \cdot D_X + (y_w - y_v) \cdot D_Y \geq 0 \\ \forall v \in V \quad \forall w \in N(v) \quad \forall D \in \mathcal{D} \end{aligned} \tag{5.7}$$

By setting some α to 1 – and thus satisfying the constraints of Equation (5.6) – we enable one subset of constraints from Equation (5.7), as L_α dominates all other terms. These constraints enforce the comparison between edge slope and direction, which gives us the direction of edge (v, w) with the correct sign.

[2] In Operations Research, this is known as the *Big-M method*.

With the correct value assigned to each α, we prevent overlapping incident edges and preserve cyclic orders of neighbors arround each vertex in one set of constraints.

Preserve cyclic order of outgoing edges. We test cyclic orders using the mapping of the corresponding edges onto the directions, as those are already ordered linearly by the angle between the corresponding vector and the positive x-axis. Given the input embedding, we enforce that for vertex v and the i-th neighbor w_i of direction D_i, the direction D_{i+1} of the next neighbor w_{i+1} must be later in the order of $\mathcal{D}$, hence we say $D_i < D_{i+1}$ identifier-wise. We also need to linearize the cyclic order of neighbors arround each vertex in order to be able to encode them into the model. This imposes the problem that we can not know in advance, which neighbor will be assigned the direction with the lowest identifier. Given k neighbors and knowing the "correct" first neighbor in advance, we would get the ordering of the directions $D_1 < D_2 < \cdots < D_k$ to match $w_1, w_2, \ldots, w_k, w_1, \ldots$. Unrolling the cyclic order of neighbors at the wrong point, there could then be a neighbor in the linearized order, for which the increasing-identifier condition described above can not hold, despite of the cyclic order being preserved in the output. To overcome this, we use a binary decision variable $\beta(v, w) \in \{0, 1\}$ for every vertex-neighbor pair, indicating if w is the "last" neighbor of v according to the order of $\mathcal{D}$. Thus, when the output contains an ordering $D_1 < \cdots < D_\ell \not< D_{\ell+1} < D_{\ell+2} \ldots$, we use the corresponding β to add the slack necessary to disable exactly one constraint.

$$\sum_{w \in N(v)} \beta(v, w) = 1 \qquad \forall v \in V, \deg(v) > 1 \tag{5.8}$$

$$\begin{aligned} \alpha_{D_1}(v, w_i) \leq\ & \beta(v, w_i) + \sum_{D_w \in \mathcal{D} : D_w > D_1} \alpha_{D_w}(v, w_{i+1}) \\ & \forall D_1 \in \mathcal{D} \quad \forall v \in V, N(v) = \{w_1, w_2, \ldots, w_k\}, k = \deg v > 1 \end{aligned} \tag{5.9}$$

For notational convenience, we let $w_{k+1} = w_1$, as $N(v)$ is conceptually circular. For any α set to 0, the inequalities of (5.9) are trivially satisfied. Otherwise, there has to be a neighbor whose connecting edge has direction with higher identifier (and thus the corresponding α set to 1), unless it is the last neighbor in the embedding of v. To ensure that there is only one "last neighbor"-violation of the constraints from (5.9), we introduce the constraints of (5.8). Adding β to every constraint of (5.9) also allows for the whole neighborhood of v to be rotated around it.

Theorem 5.5. *The ILP presented above solves Topologically-Safe Grid Representation.*

In addition to it's original purpose, the integer linear programm we described above can also be repurposed to produce grid drawings of planar graphs of small area. To do so, changes to the objective function are required; see Equation (5.10).

$$\text{Minimize } \max_{v \in V} y_v \tag{5.10}$$

Lemma 5.6. *Replacing the objective function (Equation* (5.1)*) from the above integer linear programm with that shown in Equation* (5.10) *allows searching for small area-drawings.*

Proof. Replacing the objective function with that of Equation (5.10), the ILP still computes a planar straight-line grid drawing with the given embedding. Given the bounding rectangle, the width of the drawing is at most $X_{\max}$. For each now constant width, we get a drawing of minimal height by scaling down the graphs orignal coordinates to make it fit into one grid cell and then "rounding it" back to the grid. Changing the bounding rectangle width, we get an algorithm to search for a drawing of smallest bounding rectangle. □

5.3.3 Delayed Constraint Generation

The integer linear programm described to solve Topologically-Safe Grid Representation works in theory; however, it is not suited for practical applications due to obstructive computation time. This problem becomes even more apparent when trying to find drawings of small area. Hence, we discuss the application of *delayed constraint generation* (also known in the context of *constraint generation*, see [Chi08]) to the original model as well as the area version. Delayed constraint generation is a technique to speed up the process of solving linear programms. We discuss it's usefulness and give an experimental evaluation on selected instances to support our claims. We will discuss this at the end of Section 5.4. The general idea behind this is that in a "reasonable" feasible solution, most constraints will trivially be satisfied. Our intuitive approach to constraint generation is the following: Consider two edges that are far apart in the input. If the endpoints of these edges don't move too much while rounding, the edges will also be far apart in the output. Hence, those edges will probably not want to cross, making the sets of constraints preventing them from doing so obsolete. If we could identify and remove all such constraints beforehand, the resulting model would be smaller in size and thus easier to solve. But as the problem at hand is $\mathcal{NP}$-hard, we cannot make good guesses on the relevance of individual constraints.

In general, the opposite is done to implement constraint generation: Creating and solving a possibly underconstrained *partial* model, an external oracle is used to verify correctness of the obtained solution. If the solution is correct, it is also optimal and feasible for the *full* model (containing all possibly relevant constraints); otherwise, the oracle reports back any parts of the solution that would have made it be infeasible for the full model. Then the voilated constraints of the full model are added to the partial model. This process is repeated until either a feasible solution is found or all constraints of the full model are added. In either case, the solution produced by the iteratively refined partial model is of the same quality as that of the full model.

We use delayed constraint generation to iteratively add almost all constraints only when needed. We extend the basic model to include the constraints relevant for the consistency of assignment of the sets of binary variables containing all the γ (Equation (5.4)), all the α (Equation (5.6)), and all the β (Equation (5.8)) to get the *empty model*. It is empty in the sense that it is missing all constraints that require a "meaningful" assignment of binary variables; hence the consistency constraints of the empty model can be satisfied almost trivially. In the following, we will use the empty model as the initial partial model

to start adding constraints to. Most basic ILP-solvers will pick one decision variable of the empty model at a time and branch on the possible values it can take. For our coordinates this process will implicitly first try moving each vertex to one of the corners of the cell containing it. In the next chapter, in Section 6.2, we reconstruct this procedure when describing the *Greedy Rounding*[3] heuristic; there we also provide an efficient implementation in C++ and discuss the overall performance of this heuristic.

Checking the output of a partial model for topological equivalence can easily be done by any algorthim to test for planariy. To serve as an oracle, we store the original cyclic orders of the input graph and extend the planarity test to also test for consistency with the stored orders of neighbors arround each vertex and make it report back any new incidences, crossing edges, and changed cyclic orders. We then use the results to add the constraints of Equations (5.2), (5.5), (5.7), and (5.9) respectively as needed. The extended partial model is then solved again. Notice that the actual runtime of such a test is small compared to the time required to set up and solve the model. Also notice that solving the almost empty basic model takes practically no time. Hence, that way we can easily and quickly find a reasonable initial set of relevant constraints. We support these runtime- and performance claims by observations made in the next section.

5.4 Experimental Performance Evaluation

In this section, we discuss the performance and limitations of the model described above. To do so, we made an implementation using Java to create the model and controll the IBM CPLEX solver. We tested this implementation on graphs of different sizes and complexity. The test instances are hand-picked to convey the impact of vertex count, size of bounding rectangle and number of "difficult" parts on the overall performance.

In the following, the column "*Full model*" is used for executions of the above ILP without any constraint generation. The column "*first*" gives the time until any feasible integer solution (not necessarily optimal) is reported by the integer solver. For both the full model and the constraint generation, the "*optimal*" column gives the time until the solver reports an optimal solution.

We performed the following experiments on a Linux machine with 16 cores (2666 MHz and 4 MB cache each), 16 GB memory and 20 GB swap space and using the Java bindings for CPLEX. To compare the full model and the constraint generation approach, we consider model size and wall-clock time spend solving. The model size is measured by considering the number of rows and columns after CPLEX completes any preprocessing and presolving steps; an entry of "†" in a table means that either no model could be created within given time and/or system memory, or that the solver did not report a result within 10 minutes.

For the output figures, white vertices represent the initial positions with the red arrows indicating actual vertex movement. For all input/output drawing pairs, the underly-

[3] Greedy Rounding will be implemented to obey the constraints of Equation (5.2), whereas the empty model does not.

ing grid represents the bounding rectangle that was allotted to each instance respectively. As a general measure for complexity of an instance, we will consider the *vertex density* γ, the ratio between the number of vertices in the input and the total number of grid points contained in the allotted bounding rectangle; $\gamma = 100\%$ implies that in the output, all grid points will be occupied.

Small Examples. We start the discussion with a group of three instances. Each instance has a rather small number of vertices as well as bounding rectangle area. The input drawings are shown in the upper row of Fig. 5.7; see Tab. 5.1 for data on instance size, model size, and time spend solving.

Consider Graph 1 (Fig. 5.7 left): The right of the two faces contains two additional vertices but the initial drawing has only one available grid point inside that face, making an "unlucky" execution of Greedy Rounding[4] fail to round either vertex. To overcome this, several of these vertices need to make locally non-optimal movements, enlarging the face. Graph 1 is of average *vertex-density* ($\gamma = 36\%$) compared to the other five instances. Because of its small size and low number of edges, the resulting model is small and thus an optimal solution is found rather quickly. However, its vertices are positioned so that many constraints are not trivially satisfied and significant effort is required even by the constraint generation approach.

Graph 2 (Fig. 5.7 center) has lower vertex-density ($\gamma \approx 30.5\%$). In addition, it is designed such that every vertex has one preferred integer grid point that is not contested by any other vertex; hence, greedily rounding each vertex already yields the optimal solution. The embedding preservation constraint of the central vertex involves all other vertices, and thus requires most of the α variables to be set properly, whereas in the basic model, these variables can be assigned freely. Any other constraint is easily satisfiable. In terms of computation time, there is not a big difference between finding the first solution and closing the integrality gap. By construction of the input graph, the embedding is trivially preserved by Greedy Rounding; this is also reflected considering model size and solution time of the constraint generation approach. Hence the optimal solution is found by the first run of the constraint generation approach almost immediately. (Note that the 0.5 second runtime includes setting up the Java environment, calling the CPLEX solver and checking topology.)

The input drawing of Graph 3 (Fig. 5.7 right) is comparable to those suggested to be input for the area-minimal drawing variant described in Lemma 5.6 – that is, all vertices are initially drawn within the same grid cell. With vertex-density $\gamma \approx 77.8\%$, this is also the most dense instance we did experiments on. First of all, small bounding box result in small and easy-to-solve full models. The size of the bounding box has extreme effect on the runtime – compare the times for the full models of Graph 3 and Graph 1, which has only two more vertices but a much larger bounding box. For such extreme instances, almost all constraints are required and repeated testing and adding of constraints results

[4] We will later describe Greedy Rounding to try each vertex in random order, moving it to the cheapest grid point that is available at that time.

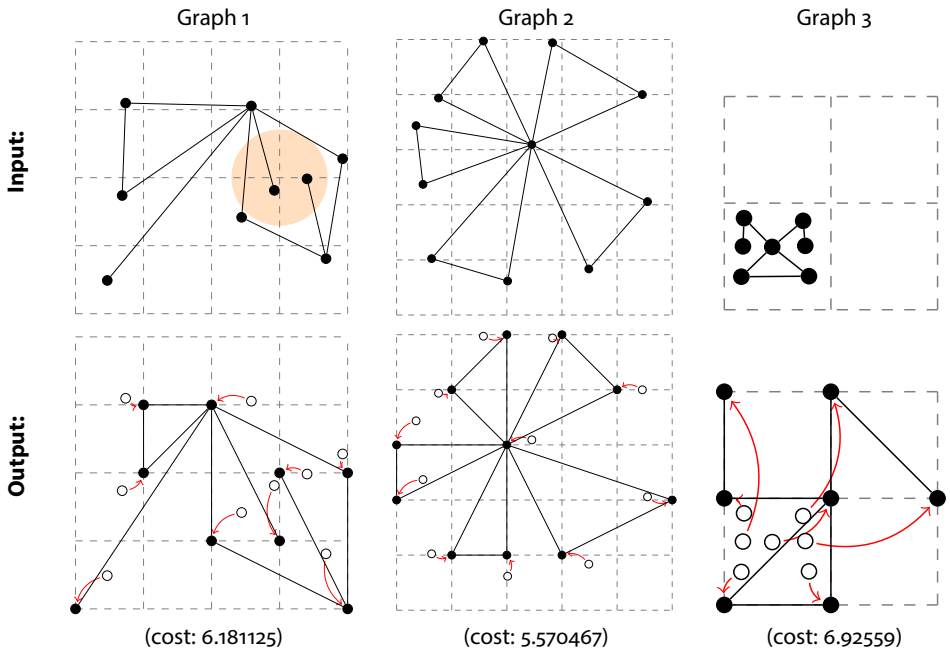


Figure 5.7: Graph 1 has one contested grid point (marked by the orange circle) that required significant effort to be resolved. Graph 2 is roundable greedily without complications. Graph 3 is similar to the instances suggested for finding area-minimal drawings, and is supposed to be challenging in general.

Table 5.1: Instance sizes and runtime measurements for Graph 1, 2, and 3, all shown in Fig. 5.7.

	Size			Full model				Constraint generation		
#	$\|V\|$	$\|E\|$	γ	rows	cols	first	optimal	rows	cols	optimal
1	9	10	36.0%	8046	2239	3.2 s	90.6 s	3791	1053	29.2 s
2	11	15	30.6%	26151	6857	5.2 s	10.6 s	2	3	0.5 s
3	7	7	77.8%	2583	896	0.5 s	4.8 s	2245	682	20.2 s

in an accumulated runtime that is higher than setting up and solving the full model in the first place.

The last example implies that constraint generation is best to be used when the instance has low-vertex density or when many vertices have uncontested preferred grid points to be rounded to. To investigate on these two implications and their relationship, we now look into larger and gradually more complex instances.

Medium-sized Examples. The performance difference between the full model and the constraint generation approach becomes more apparant when the size of the bounding rectangle increases. Consider the two instances of Fig. 5.8: They are of about the same density as the first two instances, but double in size; exact measurements on instance size and solving times can be found in Table 5.2.

Graph 4 (Fig. 5.8 left) is a path, and thus the model of this instance does not need to contain any constraints to preserve the cyclic orders of vertices. The vertices of the input drawing are spread rather evenly over the bounding rectangle, making almost all vertices have an uncontested preferred grid point to round to. Graph 5 (Fig. 5.8 right) on the other hand has one vertex more but is more connected. Graph 5 is also designed to make the Greedy Rounding fail: Trivially rounding the two upper degree-1 vertices will change their relative order arround their common neighbor; the path on the right side has several contested grid points and the vertices on it will create new incidences when rounded greedily.

These differences are also represented in the sizes of the full model for both instances. The model for Graph 5 has more than four times as big, mostly because more edges need to be considered with respect to more possible directions. Therefore the solver did not find an optimal solution for the instance of Graph 5 within 10 minutes. To capture the importance of pre-eliminating trivially satisfied constraints consider the final model sizes for the column generation approach on these instances. The lower left part of Graph 5 is a grid on nine vertices, all of which can be greedily rounded correctly. All embedding- and planarity preserving constraints for edges incident to these vertices are trivially satisfied by any good solution and thus were never added to the partial model. This results in a significant time reduction for solving the partial model to optimality – compare that time to the time it took finding any feasible solution for the full model.

Considering the partial model for Graph 4, we observe that easy instances also benefit from constraint generation, but the obtained speed-up is not nearly as high, even on a smaller bounding rectangle.

Notice that in our context, medium-sized instances have only 20 vertices and even then, we have to wait for four minutes (in the case of Graph 5) – for most scenarios, graphs of that size would be considered rather small. To emphasize the impact that constraint generation has on instances with large bounding rectangles and low vertex-density, we now consider an even more extreme example.

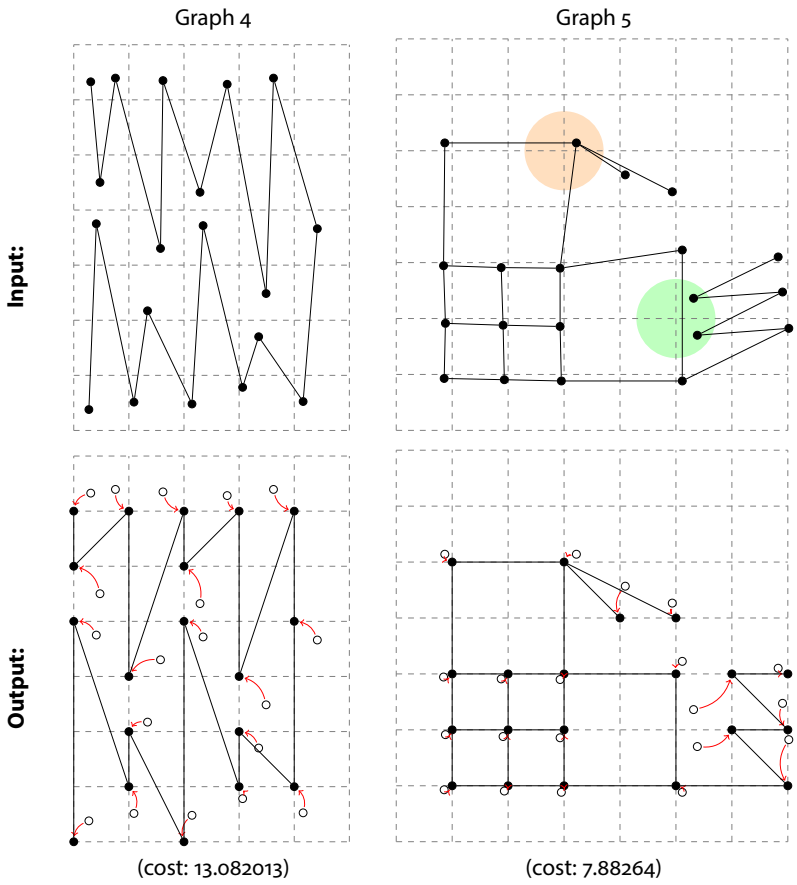


Figure 5.8: Graph 4 is a path and can be solved optimally fairly easily. Graph 5 is designed to have two complications: one changing cyclic order (orange circle) as well as new incidences and collapsing features (green circle).

Table 5.2: Instance sizes and runtime measurements for Graph 4 and 5, all shown in Fig. 5.8.

	Size			Full model				Constraint generation		
#	$\|V\|$	$\|E\|$	γ	rows	cols	first	optimal	rows	cols	optimal
4	19	18	39.5%	74957	19591	42.6 s	1105.6 s	9200	2402	21.2 s
5	20	25	31.3%	323441	82816	182.1 s	†	15127	3894	211.6 s

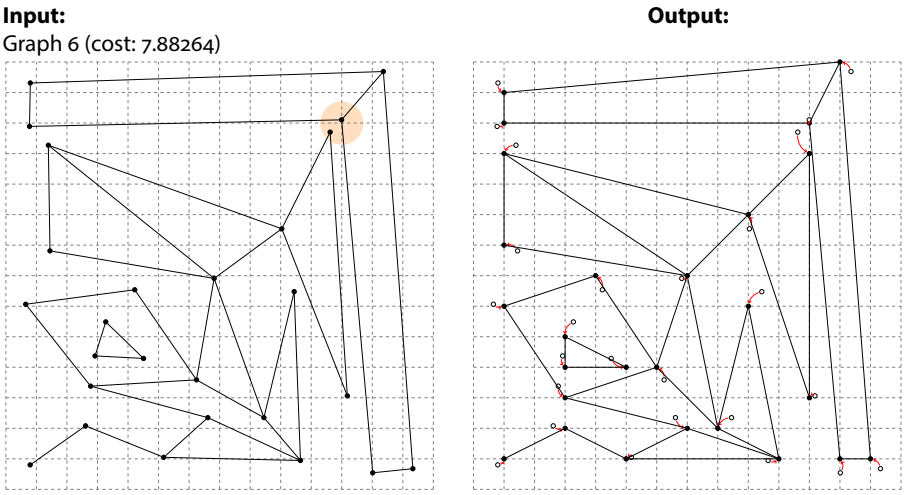


Figure 5.9: Graph 6 is the largest instance we ran experiments on. The only non-trivial part is highlighted by the orange circle.

Table 5.3: Instance sizes and runtime measurements for Graph 6, all shown in Fig. 5.9.

	Size			Full model				Constraint generation		
#	$\lvert V\rvert$	$\lvert E\rvert$	γ	rows	cols	first	optimal	rows	cols	optimal
6	26	34	11.6%	†	†	†	†	12355	3146	7.1 s

A large Example. The last instance we discuss here is shown in Fig. 5.9 (with additional data in Table 5.3). In terms of rounding complexity, this instance is almost as easy as Graph 2: there is only one pair of vertices contesting the same grid point in the top right part of the input drawing. Being an easy instance has no impact on the performance of the full model; in fact, creating all possible constraints for all possible directions in this instance already exceeded the allotted computation time and no time was left to spend actually solving the model. In contrast, the constraint generation only had to add one constraint for two vertices of the upper-right corner to the partial model. Rebuilding the model and solving with this constraint runs in reasonable time (compared to the full model). Notice that this constraint does not involve the direction set $\mathcal{D}$.

The key messages to take from this section can be summarized as follows: Small bounding rectangles result in small and quick-to-solve models, but as the bounding rectangle grows in size, so does the model – recall that the number of possible directions $|\mathcal{D}|$ is quadratic in the rectangle's larger dimension and that the number of most constraints is directly linked to $|\mathcal{D}|$. Second, when many constraints are violated during the constraint generation processes, iteratively adding the constraints results in runtime exceeding the time for solving the full model in the first place. On instances with moderate

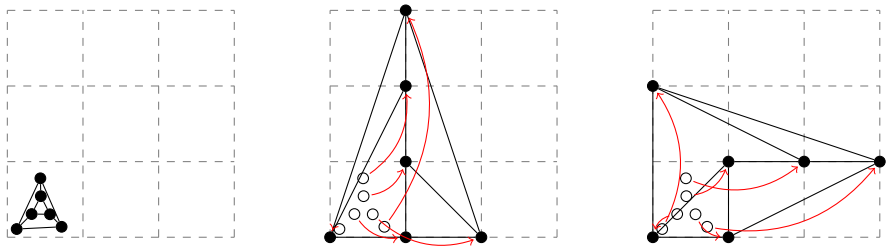


Figure 5.10: (a) A nested-triangles graph on $n = 6$ vertices, placed completely in one grid cell; (b) the ouput using the rounding objective (Equation (5.1)) – solved in 3 h 25 min; (c) the ouput using the objective function of Equation (5.10) with $X_{\max} = 3$ – solved in 10 min 31 s.

vertex-density ($\gamma < 40\%$) the constraint generation approach clearly outperforms the full model (while still being infeasible in practice).

Graph Drawing. We conclude this chapter with a brief overview on the capabilities of our integer linear programm as a graph drawing tool.

Area-minimal drawings of planar graphs are useful tools for creating (counter-) examples and thus have a long history in graph drawing [DLT83, dFPP90, FP07]. While Krug & Wagner [KW08] did provide a $\mathcal{NP}$-hardness proof, we do not know of any tool for computing optimal solutions. We demonstrate the capability of our ILP of finding such drawings and compare the model tailored to solve Topologically-Safe Grid Representation to the modified version described in Lemma 5.6. Our model does not contain constraints for fixing a particular outer face – similar to Frati & Patrignani [FP07] –, any drawing respecting the given cyclic orders is valid. In Fig. 5.10, we show a nested-triangles graph on six vertices (a), together with two area-minimal drawings created using the regular programm (b) and the modified version (c). Notice how both drawings have a non-triangular outer face different from the input and that both occupy the same area of six grid cells. Fig. 5.10 (b) was created in almost three and a half hours, whereas Fig. 5.10 (c) only took about ten mintes. This difference can be attributed to the fact that the objetive function of the area-minimizing model takes discrete values (namely the highest y-coordinate), which allows the solver to proof lower bounds faster.

Our model is capable of taking non-planar drawings of planar graphs as input – the lines separating nonincident edges in the input is not represented in the constraints – as well as (in theory) taking non-planar graphs and reporting "infeasible" for them. The latter is strongly inadvisible. We tried solving the model using a drawing of K_5 as input, but canceled computation after twelve hours.

5.5 Conclusion

In this chapter, we have discussed the problem Topologically-Safe Grid Representation – transforming a given planar drawing into a drawing with the following constraints: Each vertex has coordinates at integer precision, the topology of the original drawing is preserved, and the total displacement of all vertices is minimal among all valid transformed drawings.

We have shown Topologically-Safe Grid Representation to be $\mathcal{NP}$-hard by giving a reduction from Planar Monotone 3SAT. To find optimal solutions, we modelled the problem using integer linear programming. We created some selected test instances to evaluate the performance of our original model as well as our faster delayed constraint generation approach. To do so, we implemented the model using Java and the IBM CPLEX solver, performing experiments on a virtual machine with 16 cores. We concluded the discussion of the experiments with empirically analyzing the selected instances, pointing out why they are challenging for our implementation.

Finally, we discussed how our model can be adapted to solve another $\mathcal{NP}$-hard problem – minimizing area for planar straight-line drawings. While capable in theory, our experiments have shown the runtime to be infeasible from a practical standpoint.

The results we have presented in this chapter directly motivate those of the following Chapter 6. The integer linear programm discussed here is too slow to be used in any real world application, hence we next introduce an efficient randomized heuristic based on simulated annealing.

Chapter 6

Practical Topologically Safe Rounding of Geographic Networks in $2D$

In this chapter we consider the Topologically-Safe Grid Representation problem in an application oriented setting. We are given a road network or other map data representable by segments in the plane (e.g. administrative boundaries) and consider the problem of representing the vertices of this network at (integer) grid positions. There are several advantages to such representations, as opposed to the common practice of using floating-point numbers for coordinates. Firstly, this makes explicit what the actual precision of the representation is, in a data type without mathematical surprises [Gol91]. Also, original coordinate precision has huge impact on the precision required to safely perform geometric operations, such as intersecting or calculating overlap [Mil95]. Additionally, finding representations on particularly small grids is a natural form of data compression, since it reduces the range of the coordinate values. In geographic applications, usually large amounts of data need to be stored and processed. Specifically for mobile route planing, the devices in use often are hand-held and have limited resources: small memory, small screens, low display resolutions or slow CPUs. It is also of mathematical interest to consider the smallest grid on which the network can be represented under certain quality constraints. Furthermore, grid drawings also provide a form of schematization, by enforcing a minimum length on edges and introducing a rigid structure in dense areas. This also serves a perceptual purpose: there are no arbitrarily small features to be drawn. Such features would be hard to read and, ultimately, any visual reproduction of the network is likely to be discrete at some level anyway.

We are of course interested in *good* representations of the input network – for some measure of quality – not just an arbitrary representation. Many "rounding" and "snapping" procedures from the literature give a bound on the geometric difference between the input and output networks; usually, vertices are allowed to move only within one grid cell. They achieve this by accepting possible changes to the topology of the network, such as allowing vertices to coincide, new intersection to occur, or faces to collapse. We approach the problem from the other direction, by demanding topological equivalence between the input and output drawings and optimizing the quality of the result. This perspective is motivated by geographic networks, where connectivity, embedding (road networks), and the faces (territorial outline maps) are crucial. Our main measure of quality for the rounded output is the same as for Topologically-Safe Grid Representation: the sum over the vertices, of the Euclidean distance between their input position and output position.

This topologically-safe "rounding" problem is nontrivial, especially if the network has areas where the density of the points is high relative to the size of the grid. In fact, several variants are known to be $\mathcal{NP}$-hard [MN90, LvDW16] and no practical method for obtaining high-quality results is known. In this paper we present a practical method based on simulated annealing.

6.1 Related Work

Since related work on drawing and rounding graphs on the grid can be found in Section 5.1, we here focus on work about network compression. Recall that the approaches discussed in Section 5.1 bound the Hausdorff distance between input and output, but allow features to collapse. As argued before, these techniques may not be appropriate for geographic applications. Unfortunately, we showed in Chapter 5 that minimizing distortion in topologically-safe grid representations is $\mathcal{NP}$-hard in many settings.

Phrased as a problem of data compression, it is hard to find a minimum representation of arrangements of polygons [MN90]; also recall that the reduction we give in Section 5.2 asks for saving only a single bit on coordinate representations of embedded planar graphs. Shekhar et al. [SHDZ02] give a clustering-based approach to compress vector (road) maps; see Khot et al. [KHN+14] for a survey on road network compression techniques and challenges.

Contribution. In this paper, we consider a variant of the Topologically-Safe Grid Representation problem [LvDW16] that does not restrict the output drawing to be contained inside a bounding rectangle. Notice that the hardness proofs from Section 5.2 trivially extend to this less constrained variant. In the following, we use the Euclidean distance – denoted by $\|\cdot\|$ – to measure the cost of vertex movements.

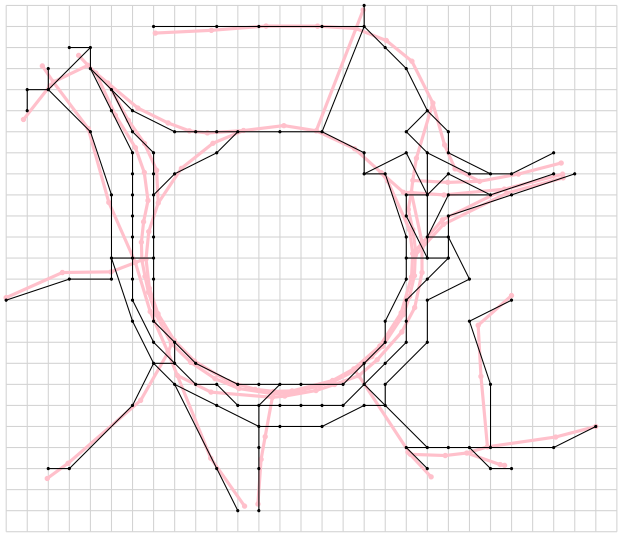
Considering the hardness of this problem, we propose a heuristic approach – in particular, a two-staged algorithm based on simulated annealing. Stage One focuses on finding a feasible solution, since finding *any* topologically equivalent grid representation that does not, in the worst case, massively distort the input is nontrivial: the goal of this stage is to find a somewhat reasonable solution that we can improve in stage two. This second stage uses simulated annealing to improve the quality of the drawing; see Figure 6.1 for two example solutions on real geographic networks.

We have implemented this approach and experimentally evaluate it on real-world networks as well as on artificially generated networks. The the former demonstrate the applicability of our approch whereas the latter allow us to statistically justify each major component of our algorithm.

6.2 Terminology and Basic Heuristics

Given an input drawing Γ_1, our two-stage approach will produce intermedate drawings Γ_k, that is, drawings where some vertices are already moved to the integer grid while

(a)



(b)

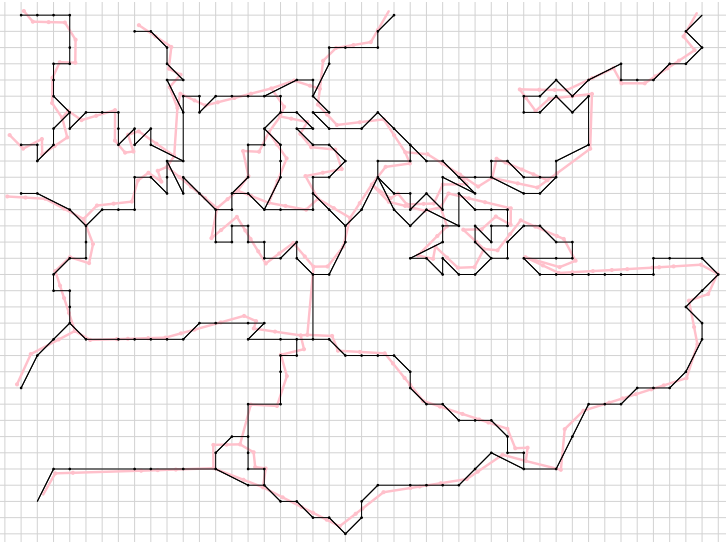


Figure 6.1: Two instances with real-valued input (light drawing) and corresponding grid representation (black drawing) computed using our algorithm. (a) Roundabout in downtown Würzburg (138 vertices, 155 edges), grid size 28×28, average vertex movement 0.600, computed in $15\,s$; (b) borders in Britain (3110 vertices, 3207 edges), grid size 240×240, average vertex movement 0.435, computed in $70\,s$, cropped to show only the south-east.

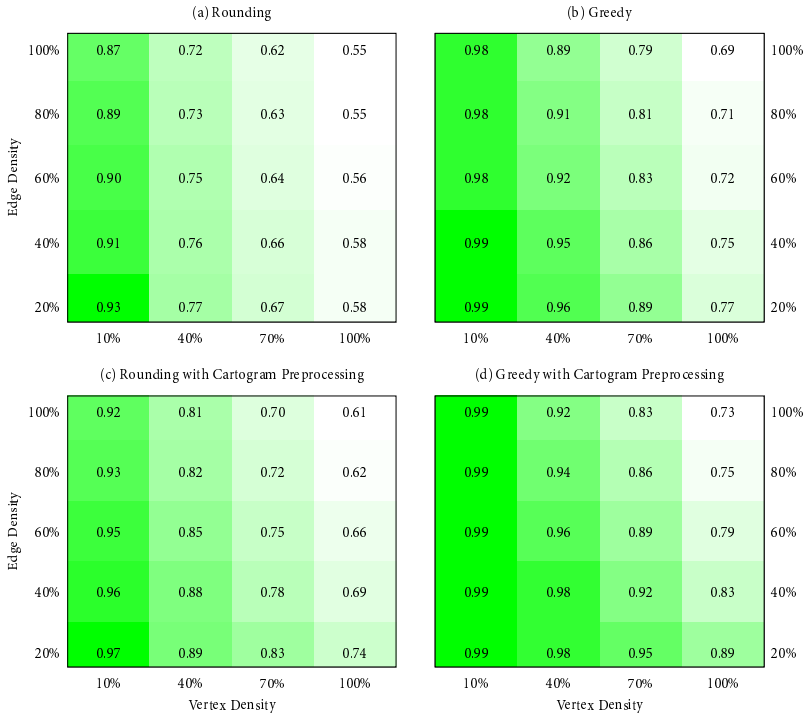


Figure 6.2: Success rates of (a) incremental rounding, and (b) incremental greedy as described in Section 6.2. Figures (c) and (d) include the cartogram preprocessing from Section 6.3.3. Each entry is for 100 random instances with area $[0, 19] \times [0, 19]$ and varying vertex and edge density (see Section 6.4.1 for a description of the random instances). Matrices are sparsified for readability; more data is available and follows the trend.

others are not. Hence, we call a vertex *nongrid* if its coordinates are not integer. As before, the *cost* of a vertex v in a drawing Γ_k (with $k > 1$) of G is defined as the Euclidean distance between its original (real) position in the input $p(v)$ and its position in the drawing $r(v)$; the cost of a drawing is the sum over the costs of its vertices. This matches the notation and objective function stated in Section 5.3.1. A drawing is called *feasible* if it is topologically equivalent to the input and all vertices are positioned on grid points. Note that the original drawing Γ_1 always has cost zero, but – except in trivial cases – is infeasible due to nongrid vertices.

In the iterative procedures we describe in this chaper, we proceed from one intermedate drawing to the next by *moving* a vertex: a move is the change of position of a single vertex of the current drawing, keeping the position of all other vertices unchanged. A move is called *valid* if it does not change the topology of the drawing, and the cost of the move is the difference in cost between the new and the old drawing.

As a baseline, we first describe several rather trivial *partial* heuristics. They relate to *rounding* vertices as known from the literature – moving each vertex to the closest available grid point –, and they are partial in the sense that they are not guaranteed to find a feasible solution.

Instant Rounding. Independently set the position $r(v)$ of each vertex to the nearest grid point. If the resulting drawing is topologically equivalent to the input, it is an optimal solution since every vertex individually moves the minimum amount possible. Otherwise the heuristic failed and does nothing.

Incremental Rounding. Round the vertices one by one, in arbitrary order, checking the validity of each move. Undo any invalid moves, leaving each such vertex v in their original position $p(v)$. In contrast to the instant rounding heuristic, this procedure may succeed in rounding a subset of the vertices even if it fails to solve the entire instance; this can be useful for quickly solving "easy" parts of the graph. The cost of this drawing is still a lower bound on the optimum, since any vertex that moves does so by the minimum amount.

Incremental Greedy. Do the following for all vertices, one by one, in arbitrary order: consider moving it to any of the four corners of the grid cell it is in.[1] Check these candidate moves for validity in nondecreasing order of cost (distance to $r(v)$); accept the first valid move and continue with the next vertex. Vertices with no valid candidates remain unrounded. Note that the rounding cost of this (partial) solution is not a lower bound on the optimum, even if it finds a valid grid position for all vertices, since greedy decisions were made.

We also present a heuristic that always finds a feasible solution. However, it can give very bad solutions, as we will discuss in Section 6.4.2.

Scale & Greedy. Repeat the following steps in order until a valid drawing is found. Uniformly scale the input network by a factor f (initially $f = 1$) and apply the Incremental Greedy algorithm. If this results in no vertices being nongrid, we are done; otherwise increase f by one and retry. This process terminates, since Incremental Greedy will succeed for large enough f. However, scaling the entire network is likely to result in high cost.

Depending on the properties of the instances under consideration, these heuristics may work quite well or experience significant difficulties: see the top row of Figure 6.2. Here we consider random planar graphs of various vertex density γ and edge density ε on a fixed area. These instances and how they are created is described in more detail in Section 6.4.1; here we include vertex densities from 10% up to 100% (meaning one vertex

[1] This is ill defined for vertices on grid points and grid lines; in fact we take the floor or ceiling on each axis, giving *at most* 4 candidate positions for the vertex. In particular, this means that vertices that already lie on a grid point are not moved.

per grid cell on average), and edge densities from approximately matching-sized edge sets (20%) up to a full Delaunay triangulation (100%).

The matrices show the success rate of the Incremental Rounding and Incremental Greedy algorithms on 100 graphs for each setting; that is, numbers and color indicate the fraction of *vertices* in the output drawings that are at integer-precision coordinates. We see that the greedy approach clearly performs better than simple rounding, but even in the sparsest graphs, rounding only moves 99% of the vertices to the grid on average. The gradient of the values indicates that vertex density seems to be more challenging for both approaches than the number of edges: although the edges also constrain the feasibility of drawings, it is particularly the (local) density of vertices that forms a problem for these heuristics. Note, for example, that a grid cell containing more than four input vertices cannot be completely rounded by these heuristics; this, among other difficult situations, is more likely to occur in instances of higher vertex density. Scale & Greedy overcomes this problem by exploding the size of the network, effectively increasing the number of available grid points at the expense of a large (rounding) cost.[2] This is efficient in terms of runtime, but often requires large scale factors on real data and therefore returns drawings that are completely unacceptable as a heuristic and impractical for Stage Two. We support this claim by experimental results, see Section 6.4.2.

6.3 The Two-Stage Algorithm

We do not allow alterations of the network and insist on moving all vertices to the grid, thus we have to handle vertex-dense clusters in input graphs. To do this, we first propose and discuss several approaches. The difficulty of Topologically-Safe Grid Representation lies not just in finding a solution with low cost – it is nontrivial to find any decent feasible solution at all. Given these difficulties (and the intractability of finding optimal solutions [LvDW16]), we now consider a local search approach. Since we do not want to require starting with a feasible solution, this local search will have to handle infeasible states: it needs to consider drawings in which not all vertices lie on a grid point (yet). However, it will never consider drawings that are not topologically equivalent to the input: Topological equivalence of current drawing and input is an invariant we maintain at all times, rather than a property we are searching for. All our search algorithms use the following neighborhood definition for intermedate drawings.

Local search neighborhood. Take any vertex and perform any valid move among the following. If the vertex nongrid, move it to a random corner of the grid cell it is currently in; otherwise, move it to one of the eight grid points surrounding it.

If we use hill climbing (always greedily picking the neighbor with lowest cost) in this neighborhood definition, nothing happens: the state representing the input drawing Γ_1

[2] From a data compression perspective, scaling by a factor two corresponds to using an additional bit on each axis of each vertex. With enough bits, the drawing can be represented without significant rounding.

already has cost zero and any neighbor of Γ_1 is more expensive. Since we generally foresee further complications due to local optima, we pick the well-known metaheuristic *simulated annealing* [KGV83]. For a description of simulated annealing, see for example Van Laarhoven and Aarts [vLA87]; below we briefly sketch the general approach. Simulated Annealing borrows its terminology from metalworking, hence we will use terms like energy, temperature and cooling, which we define next.

Simulated Annealing. Simulated annealing is an iterative local search procedure. Consider all topologically valid drawings of the input network as possible *system states*. We define the *energy* $E(\Gamma)$ – or *cost* in our terminology – of a state Γ to be the rounding cost as defined in the problem statement of Topologically-Safe Grid Representation. To transition from state Γ_k to Γ_{k+1}, we pick a random (valid) move from the *neighborhood* described above. If the new state has less energy than the current one ($E(\Gamma_{k+1}) < E(\Gamma_k)$), we *accept* it as our new current state. If the new state has the same or more energy – that is, it is worse – we can still accept it, and do so with probability $\exp(-\delta/T)$, where δ is the difference in cost and T is a variable called the *temperature*. This allows simulated annealing to escape local optima. As *cooling schedule*, we employ the standard exponential schedule $T_n = c \cdot T_{n-1}$ for some constant factor c; in this way, it becomes less likely to accept worse solutions as the search progresses. We discuss the impact of different values of c in Section 6.4.3.

We observe experimentally that a straightforward annealing approach using cost as the objective function does not perform well at all (see for example Figure 6.4 (a)): the focus on cost often prevents it from finding a feasible drawing. One possibility to overcome this would be to design an objective function that rewards feasibility. However, this presents several difficulties. Firstly, in order for the search procedure to actually *find* the feasible drawings, this added term must be "smooth" enough to provide attraction, but it is not clear how to do this. Furthermore, depending on the details of this added term, it would have to be tweaked to not interfere with the cost optimization too much. We are eager to avoid such extra tuning parameters, which would come on top of the parameter tuning required for the simulated annealing itself. Therefore, we take a different route by splitting the algorithm into two stages: one focused on finding a reasonable feasible drawing quickly, and a second straightforward annealing phase to minimize the total cost of all vertex movements. We conclude the section with optional preprocessing and postprocessing steps, and some algorithmic implementation considerations.

6.3.1 Stage One – Feasibility

We describe several feasibility procedures. Their goal is not to minimize the total movement cost, but to efficiently find a feasible drawing that can then be optimized well. We first sketch an approach that is guaranteed to find a feasible drawing quickly. Unfortunately, these drawings have impractically high cost and do not serve well for Stage Two. Then we describe a local search procedure that works well in practice.

Graph drawing. Take the input as an abstract (embedded) graph and draw it anew, ignoring the original vertex positions. This can be done, for example, with the algorithm of Harel et al. [HS98] which – efficiently and deterministically – computes a compact grid drawing preserving a given embedding. Besides several technical challenges, such as requiring the graph to be biconnected, our experiments indicate that these Harel et al. drawings do not provide a good starting point for our Stage Two: they are too dissimilar to the input in terms of shape and positions. See Section 6.4.2 for a qualitative evaluation of this approach.

Vertex-density annealing. This procedure uses the local-search neighborhood introduced above, but with a different objective function. Since locally dense regions are hard to successfully round, we want the search algorithm to make space. We therefore minimize the following objective function: the sum of inverse squared distances for all pairs of vertices.

$$f_{\text{density}}(p) = \sum_{v,w \in V, v \neq w} \frac{1}{\|p(v) - p(w)\|^2} \tag{6.1}$$

We now use simulated annealing to reduce this score by moving vertices away from each other in a topologically safe way one step at a time – either putting it to one of the four near grid points (cell boundaries) or moving to one of the neighboring eight grid-points. This function is modeled to be reminiscent of the repulsive forces in force directed graph drawing [Ead84]. Notice that actually minimizing this function is very easy: simply scale the graph arbitrarily large. Hence we do not *actually* want to minimize this objective function but rather run the search at a constant temperature of $T = 1$ and terminate the search as soon as the drawing is feasible.[3] We unconditionally accept any move that moves a nongrid vertex onto a grid point, regardless of its effect on the cost, since that is the real goal of this stage. Constantly keeping the temperature at a relatively high level ensures enough freedom of movement: the goal of this stage is to escape locally difficult situations.

Here are some observations about Vertex-density annealing: As we consider squared distances, a cluster of very close vertices will have strong impact on the energy of an otherwise sparse network. If the score of a vertex is low, it is more likely that greedy snapping will be safe as small changes can only cause problems when grid points are contested. Every network always has a density-reducing move, as some vertex on the outer face can always move away from the others. Making small moves keeps vertices rather close to their input coordinates.

Grid-density annealing. This functions identically to Vertex-density annealing, except it interprets the density more locally based on the grid, using the following procedure. Every nongrid vertex adds $\frac{1}{4}$ "density" to its four surrounding grid points and

[3] This makes "annealing" a bit of a misnomer, but for uniformity of presentation, and since we do have Kirkpatrick-style move acceptance, we call it annealing.

every other vertex adds $\frac{1}{9}$ to its eight surrounding grid points and the one it is placed on. That way, vertices effectively charge the grid points they could possibly move to within one iteration; grid points with high values are more likely to be contested by multiple vertices. Then we say the cost of a vertex is the squared density of the grid point it is on – or, for a nongrid vertex, the squared density of the nearest grid point. The annealing objective value is the sum of these vertex costs. As with Vertex-density annealing, this encourages vertices to move out of the way of other vertices, and particularly provide space for nongrid vertices (since they contribute more to the density). However, it is not immediatelly clear if its more local nature is good or bad. This is evaluated in Section 6.4.2.

Improvements based on Structural Considerations. In order to find a feasible drawing sooner, we augment the neighbor selection by adding two improvements that are both based on the special structure of the problem at hand. Both further depart from the classic simulated annealing approach described above by changing how the local neighborhood of a given drawing is generated and explored.

First, at each step, we perform the Incremental Greedy algorithm in addition to the regular local search move, immediately moving any nongrid vertices to the best available grid point on its cell. This shortcuts having to wait for the random vertex selection to pick such vertices eventually.

Secondly, rather than selecting a vertex uniformly at random, we temper how vertices are sampled in two ways. We select the vertex according to a distribution based it's individual contribution to the total density measure of the current state. This should encourage the algorithm to search in areas that are too dense, hopefully giving progress toward feasibility. Additionally, since we try moving all nongrid vertices in every iteration, we only try moving vertices already on the grid – those are the vertices that block grid points, possibly blocking other vertices from being put onto the grid. See Section 6.4.2 for a statistical evaluation discussing these modifications.

6.3.2 Stage Two – Reducing Cost

Once Stage One finishes by finding a feasible drawing, we switch to straightforward simulated annealing with the objective function we originally intended – minimizing rounding cost. Since this is a real annealing approach where we want to avoid local optima, we pick the typical exponential cooling schedule. See Section 6.4.3 for details of the parameter selection. Using any Stage One-strategy will move vertices away from their original position – some more, some less, but the network is likely to expand. Stage Two tries to undo the expansion while maintaining topology and coordinate precision.

To transition between two neighboring states Γ_k and Γ_{k+1}, we randomly pick a vertex $v \in V$ and randomly mutate it's current position by independently mutating it's (integer) coordinates – adding or substracting 1 from x_v and/or y_v. Given that v was on the integer grid before, it will be on the integer grid afterwards – the mutation will move v to one of the eight (octilinear) neighboring grid points.

The way we generate these drawings does not check topological equivalence – rotation systems might change, vertices might end up on other structures or edges might cross. To overcome this, we rely on a modified acceptance function: after generating some neighbor Γ_{k+1}, we check it for topological consistency and immediatelly reject any inconsistant drawing. If Γ_{k+1} was not rejected, we evaluate the energy level $E(\Gamma_{k+1})$ and follow the original approach proposed by Kirkpatrick [KGV83]: If the energy level (the total movement) is lower, we accept Γ_{k+1}; otherwise we can still possibly accept it, even if the energy level is higher. The propability for accepting an energy-increasing move is shown in Equation (6.2).

$$\exp\left(-\frac{E(\Gamma_{k+1}) - E(\Gamma_k)}{T}\right) \tag{6.2}$$

6.3.3 Pre- and Postprocessing

Preprocessing: Linear Cartograms Since dense areas of the input drawing are hard to resolve, some preprocessing to assist Stage One seems appropriate. Hence, we propose expanding the input using efficient linear cartograms by van Dijk et al. [vDH14, vDL18], creating a twofold set of linear equations. One on side, we ask for any edges shorter than length $\sqrt{2}$ (the diagonal of a grid cell) to be elongated and for vertices that are too close together to be moved apart; on the other side, we want the vertices to stay relatively close to their input positions. We will see in Section 6.4.2 that using linear cartograms is quite effective in practice.

This preprocessing is implemented using the following linear least-squares adjustment formulation, computing new positions p given the original positions r. Note that we create an overdetermined system of equalities on purpose; least squares adjustment will find an optimal compromise between the conflicting constraints. (See for example Kraus [Kra11] for a general introduction.)

Firstly, in the interest of the cost of the final drawing, vertices should ideally stay where they are. For every $v \in V$, we therefore have the following *vertex position* constraint from Equation (6.3) on the x and on the y axis.

$$p(v) = r(v) \tag{6.3}$$

The relative position of vertices that are connected by a long edge should also remain the same, making the length of that edge also stay the same. Edges shorter than $\sqrt{2}$ are problematic since it is likely that both endpoints contest for the same grid position. We want to introduce some additional distance between the two vertices of any short edges, stretching the edge towards length $\sqrt{2}$. Hence, we have one of the two *edge length* constraints for any $(u, v) \in E$ of the input and on both axes – either from Equation (6.4) for long edges or from Equation (6.5) for short edges.

Table 6.1: Different weights tested for the Cartogram preprocessing.

	base	low	high	final
vertex position	0.2	0.0001	1.0	0.8
edge length	4.0	0.1	8.0	4.0
constrained Delauney	2.0	0.1	4.0	2.0
vertex distance	1.0	0.1	4.0	1.5

$$p(v) = \begin{cases} p(u) + r(v) - r(u) & \text{if } \|r(u) - r(v)\| > \sqrt{2}, \text{ or} \quad (6.4) \\ p(u) + \dfrac{\sqrt{2} \cdot (r(v) - r(u))}{\|r(v) - r(u)\|} & \text{otherwise.} \quad (6.5) \end{cases}$$

We also add the constraints of Equations (6.4) and (6.5) to the non-edges missing from a constrained Delaunay triangulation. In addition, we put additional emphasis on separating any pairs of nonincident vertices that are too close, adding *vertex distance* constraints similar to Equation (6.5). See van Dijk et al. [vDvGH+13] for more on constrained Delaunay triangulations when transforming geographic networks.

Since we want to punish violations of the different sets of constraints differently, we add weight factors to the misclosures of the different equations. Intuition suggested that preventing short edges is most important, whereas trying to keep vertices at their original positions is conflicting all other constraints. Also, we deemed pushing apart any vertices that are too close to be more important than preserving the constrained Delaunay triangulation. Following these assumptions, we have considered a base setting for the weights – see Table 6.1 first column – and from these values, we pitched each weight to either extreme individually –Table 6.1 second and third column respectively –, leaving the other three unmodified.

We finally settled for the values shown in the fourth column of Table 6.1. Those will also be the values that we use throughout the rest of this chapter whenever we use cartogram preprocessing.

We only modify vertex positions accoring to the weighted minimum of misclosures, but do not check for topological consistency. Hence, the resulting drawing $\Gamma(G, p)$ using the new positions might not be topologically equivalent to the input. In that case, it would be possible to use the event constraints of Haunert et al. [vDH14]. In addtion to event constraints, they also describe a back-off procedure, that we use instead: Considering linearly interpolating the vertex positions uniformly from in the input drawing to those obtained by the least squares adjustment. The interpolation factor (ranging from 0 to 1) can be seen as a time step. We find the latest time step in this interpolation that yields a valid drawing, and output the corresponding drawing.

Postprocessing: Hill climbing. Our simulated annealing algorithm randomly chooses the next state to evaluate. Annealing theory suggests that by extending the cooling schedule the probability for the algorithm to find a global optimum converges towards 1, see Granville et al. [GKR94]. As we have no efficient means of telling if a solution is optimal or not, we have to stop Stage Two eventually and take the last accepted state as final drawing. Annealing for longer could possibly yield better results, but by choice of our cooling schedule, the annealing temperature goes to zero and the search reduces to local hill climbing. Hence we propose a straightforward hill climbing implementation as a much more efficient postprocessing: rather than sampling random vertices and attempting moves, it iteratively applies improving, valid moves to all vertices until a local optimum is reached. This suggests the possibility of annealing at a higher temperature than one normally would, and relying on the final hill climb to clean up the solution; see Section 6.5 for an evaluation on the tradeoff between hill climbing and annealing for longer time.

6.3.4 Implementation Considerations

Our algorithm often needs to test whether a particular move is valid. The expensive part in this is checking for possibly intersecting edges. Rather than the well-known (and worst-case more efficient) Bentley-Ottmann sweepline algorithm [BO79a] for line segment intersections, we implement the following bin-based approach. First, we overlay a $W \times W$ regular grid of rectangular bins over the full extent of the current drawing, then loop over all edges and put them in all bins that they intersect, and finally check all pairs of edges in each bin by brute force. If W is picked large enough, at most a small number of edges will be in any particular bin. This is efficient on realistic data for a wide range of values W. Our code uses $W = 512$ as a somewhat arbitrary trade-off between memory, number of bins, and the population of bins. For more details on this method, see also Peng and Wolff [PW14].

In fact, this grid-based approach – at least for our application and on our data – outperforms the CGAL implementation [ZWF19] of the Bentley-Ottmann algorithm by more than two orders of magnitude, even though CGAL generally has high-quality and high-performance implementations. Its problem seems to be the numerical instability of the sweepline algorithm, which requires CGAL to use high-precision arithmetic on our real-world data: something our relatively crude algorithm does not require. We additionally point out that our current implementation is single threaded, but in principle checking the bins can be easily parallelized.

We use CGAL for basic geometric computations and for constrained Delaunay triangulations [Yvi19]. We use Eigen [GJ+10] for highly-efficient sparse matrix maths in the cartogram code.

6.4 Experimental Results

In this section, we statistically evaluate various properties of our algorithm and the effectiveness of various design decisions and parameter choices. Whenever we directly compare two options, we provide a (two-sided) Wilcoxon paired signed-rank test [Wil45] and report the z-score, demonstrating statistically significant improvements; recall that a z-score of 1.96 or higher satisfies a 95% confidence level.

6.4.1 Test Instances

We provide experiments on both real-world networks and artificial instances. For the real networks, we have used GeoFabrik's OpenStreetMap shapefiles[4] and the City of Chicago Open Data Portal.[5] Since these road networks are not always planar, we have introduced vertices at any intersections in order to get plane graphs.

We also consider random artificial instances. This allows us to perform systematic and statistically significant experiments on large datasets without having to pick and sanitize real-world road networks. The instances are based on vertices sampled using binomial point processes, placing v vertices uniformly at random in a square area of X by X units – thus having $X+1$ possible coordinate values in each dimension. To generate instances with many edges, we take the Delaunay triangulation [GKS92] of this point set; for instances with fewer edges, we take a subset of those edges as described below. We therefore have three parameters describing size and complexity of an instance: the side length X of the area in which the points are sampled, the number of vertices v, and the *edge density* – number of edges as a percentage ε of those present in a complete Delaunay triangulation. As an alternative to specifying the number of vertices v, we can also consider the *vertex density* γ – the ratio between the number of vertices and the number of grid points of the area; up to rounding we have $v = \gamma \cdot (X+1)^2$. Throughout the rest of the chapter, we describe random instances by the triple of these parameters, namely as (X, γ, ε).

Note that by calculating a Delaunay triangulation of the point set of a real-world instance, we can also determine a value of γ and ε for those. As noted above, vertex density can also be interpreted as desired compression rate. While testing our algorithm on real-world instances, we noticed that aiming for $\gamma < 40\%$ gives good results in reasonable time. Therefore, we will also focus our experiments on artifical instances on those with vertex density $\gamma = 40\%$. Edge density is independent of grid resolution. We have seen in Figure 6.2 that the impact of higher edge density is less severe than that of higher vertex density when considering how complicated rounding that instance is. The road networks we consider typically have an edge density value of $\varepsilon \in [35\%, 45\%]$ – therefore we focus on artifical instances with $\varepsilon \in \{40\%, 100\%\}$. For comparison, see the instances listed in Table 6.2. When dropping edges from a Delaunay triangulation to obtain lower

[4] `https://www.geofabrik.de/data/download.html`
[5] `https://data.cityofchicago.org/`

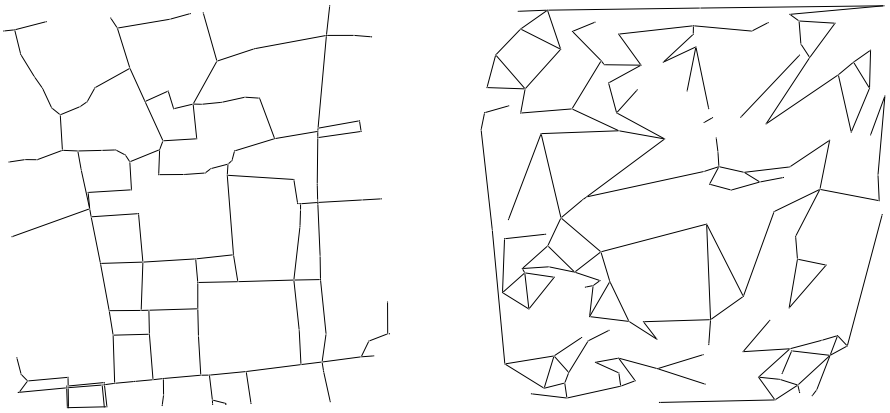


Figure 6.3: Visual comparison of real-world and artificial networks: (a) Würzburg downtown with 134 vertices and $\varepsilon = 40.5\%$; (b) an artificial instance with the same parameters.

values of ε, we need to make sure that we do not create isolated vertices. Our algorithm was designed with road networks in mind: while road networks can easily be disconnected, they usually do not have isolated – unreachable – vertices; hence, our algorithm can not handle such vertices. To avoid this problem in our artifical instances, we proceed as follows. We sample the point set and calculate a Delaunay triangulation. Then we take a perfect matching in an arbitrary DFS tree of this triangulation (which exists [Dil87]) marking those edges; we delete a uniformly random set of the other edges to achieve the desired number of edges. Since the marked edges are a perfect matching, no vertices end up being isolated in the final instance. A visual comparison is provided in Figure 6.3: left hand side is a part of Würzburg downtown, right hand side is an artifical instance with approximately the same density parameters.

6.4.2 Evaluating Stage One

As alluded to before, the basic rounding heuristics will not consistently find feasible drawings and those provided by the graph drawing algorithm are too distorted for the annealing process to find a good solution in reasonable time. In this section, we evaluate the performance of the procedures from Section 6.3.1 on artificial instances. As a baseline, we first look at the most basic form, that is, without greedy steps during the annealing and sampling all vertices uniformly. The experiments in this section are run on 1000 random instances of $(19, 40\%, 100\%)$, that is, 400 grid points in the sampled area and therefore 160 vertices and about 450 edges.

To decide between the various procedures, we first rule out Scale & Greedy and Redrawing the embedded graph; then we compare Vertex-density annealing to Grid-density annealing, with and without cartogram preprocessing and other improvements over a traditional annealing approach.

Scale & Greedy is bad. We have seen in Section 6.2 that Incremental Greedy is bound to fail if there are too many vertices inside the same grid cell. Scale & Greedy works on the idea that adding more grid lines – and thus more possible coorindate values – will eventally make Incremental Greedy work. Instead of actually inserting new lines – moving from integer to half integer, then to quarter integer and so on – we did the opposite, scaling the drawing (and all coordinates) by a constant factor. By doing so, we preserve the notion of "moving to the integer grid" and report scaling factors instead of additional bits needed for representing the refined grid. Performing the following experiment, we did a linear search for the right factor to scale the original coordinates with starting at 1; once a feasible solution is found, we report the factor and the total vertex movement induced by scaling. The instances in this experiment admit reasonable solutions with a cost of about 500 (as we will see). Scale & Greedy required an average scale factor of 3.86 (between 2 and 23), resulting in an average cost of 3329 and no solution with cost below 1047.[6] This holds true also for real-world instances, producing a feasible drawing of the roundabout in Würzburg from Figure 6.1 (a) requires a scaling factor of 4. This disqualifies Scale & Greedy as a practical Stage One procedure.

Redrawing is worse We created grid drawings of all 1000 instances using an implementation of the algorithm of Harel et al. [HS98]. In the context of classic graph drawing algorithms, the instances are very small and thus grid drawings can be computed quickly compared to our randomized annealing approaches. One could imagine that the time saved in Stage One could be spend optimizing the output for longer. However, the average cost of these drawings is an enormous 26256 (or average cost per vertex of 164). This makes it is completely impractical – consider for example that any move in Stage Two can improve the cost by at most $\sqrt{2}$.

Vertex density vs. Grid density In Section 6.3.1, we have proposed two different "density" objective functions and claimed that immediately optimizing cost has difficulty finding feasible solutions. We now experimentally evaluate this.

We ran the three options – cost, Grid-density, and Vertex-density – on each graph for 20000 iterations. Whenever a feasible solution was found within the alotted number of steps, the remaining iterations were spent in Stage Two by optimizing the output drawing for total movement cost. See Figure 6.4 for the behavior on a typical instance over time (counted by iterations): the purple line shows cost, the turquoise line shows the number of vertices that have taken a grid positions.

[6] Here we measure cost by aligning the centers of both drawns so that scaling up moves vertices away from the center.

A general trend over the 1000 instances is that Vertex-density annealing generally requires fewer steps to find a feasible solution, whereas Grid-density annealing generally finds a (first) feasible solution with lower cost; simply annealing with the objective of phase two – minimizing rounding cost – fails to find a feasible solution at all on 362 of the 1000 instances, oftentimes being stuck on the last one or two vertices. Since it is not significantly better than Incremental Greedy, this disqualifies using annealing for rounding cost to find feasible drawings. On the other hand, a Wilcoxon test shows that the differences between Vertex-density and Grid-density are significant: Vertex-density annealing finishes significantly sooner ($z \approx 28$, "winning" 688 of the 1000 instances), but with higher cost ($z \approx 10$). As a qualitative indication, when it finished first, the Grid-density solution was 15% cheaper on average (347.208 compared to 407.916). On the instances where Vertex-density was faster, the average cost of the solutions found was higher: 592.480. This suggests that Grid-density is quicker at finding relatively easy solutions, but struggles on more difficult instances.

Cartogram preprocessing The main goal of linear edge cartograms from Section 6.3.3 is to improve how Stage One finds a feasible solution.

Concerning the time taken to find a feasible solution, we observe the following: Both procedures got significantly faster – for Vertex-density, we get $z \approx 25.95$, and for Grid-density, we get $z \approx 23.35$ – and while both procedures improve, it still holds true that annealing for Vertex-density is generally the faster option for Stage One($z \approx 22.47$).

Considering the total cost of the feasible solution, we notice that those found by Vertex-density annealing got cheaper ($z \approx 3.828$), while we failed to show an improvement for the Grid-density annealing ($z \approx 0.124$). Again, the observations we made above are still true, and Grid-density annealing finds significantly cheaper solutions ($z \approx 26.53$).

This demonstrates that preprocessing instances using linear edge cartograms is highly advisable in practice; it is fast – in our implementation, it takes the time of about 50 annealing iterations on typical instances – and improves the process of finding a feasible initial solution in almost every aspect.

Incremental Greedy and Nonuniform sampling Finally we add the two additional features from Section 6.3.1: additionally running the basic Incremental Greedy heuristic at every step, and nonuniform sampling of the vertex to move.

Still on the same instances, we first investigate on the additional executions of Incremental Greedy. We notice that Vertex-density annealing still finishes earlier on 644 instances (also significant, $z \approx 21.0$). On either procedure, the speed improvement obtained by adding Incremental Greedy is highly significant – $z > 36$ for both variants. Considering the movement cost of the found feasible drawings, we look at the average total movement for both variants: The average cost of all instances where Vertex-annealing finished first was 226.3 and the average cost of all instances where Grid-density annealing finished first was 188.9 – a cost reduction by almost 50% in both cases. In fact, both

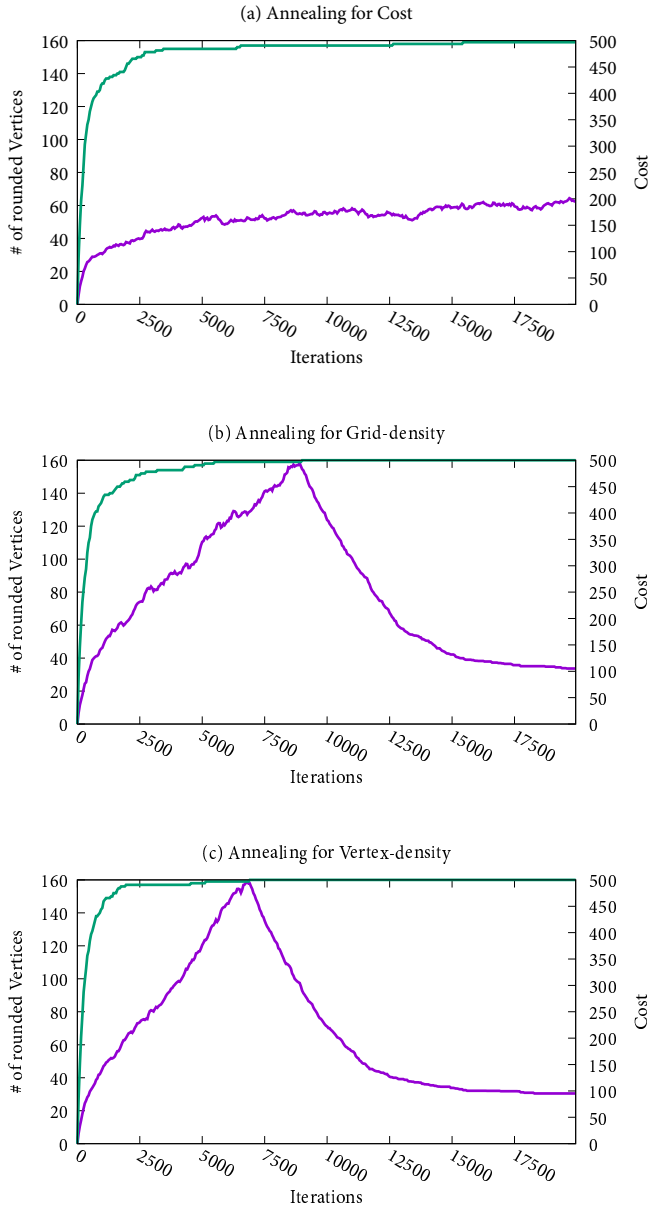


Figure 6.4: Qualitative evaluation of Stage One performance on a single artifical instance with $(19, 40\%, 100\%)$; annealing Stage One for: (a) cost, (b) Grid-density, and (c) Vertex-density. Purple line shows total rounding cost, turquoise line shows the number of rounded vertices. TODO: consider moving this figure to make it appear next to paragraph

cost improvements are also highly significant ($z > 50$ for both); the cost-difference between both procedures is also significant ($z \approx 24.92$). Considering these significance levels, we decided to include Incremental Greedy executions into both of our Stage One procedures by default.

Since we now treat vertices differently, we investigate on the option of sampling vertices nonuniformly based on their local density measure. Looking at the average costs of the winning instances for both procedures, we get 196.2 for Grid-density and 229.9 for Vertex-density. Comparing these numbers to those with uniform sampling, one might think that nonuniform sampling is actually worse. Statistical tests actually suggest that there is no significant difference between uniform and nonuniform vertex sampling; all z-scores are below 0.34, implying that neither variant shows significant differences in quality or speed. We claim that the reason for this behavior the following: Choosing the right vertex to try moving is a delicate task. On one hand, a vertex with low density is likely to have many moves that lead to valid drawings; on the other hand, none of these drawings will be significantly less dense (in either measure), as both density measures are somewhat symmetric. Vertices with high density values are likely to be close to other vertices of high density and the same is true for vertices with low density respectively. Thus, moving a low-density vertex has only limited impact on the overall density of the full drawing. Successfully moving a high-density vertex would have a much higher impact on the overall density of the resulting drawing – this observation motivated trying nonuniform sampling. However, the higher the local density of a vertex is, the harder it is to actually find a valid move for it. Hence, favoring the more dense vertices while sampling for possible moves, is likely to result in more neighboring states being rejected by the topology test.

The final modification to the vertex sampling stems directly from how Incremental Greedy changes the network at every step. Recall that Incremental Greedy will try moving every nongrid vertex onto one of the four corners of the cell containing the vertex. Mutating a nongrid vertex in the local search neighborhood – moving it to a random corner of it's cell – simply randomly tests one of the four options available to Incremental Greedy. Hence, after running Incremental Greedy on an instance, no nongrid vertex will have any valid move available to it (as it would have been performed by Incremental Greedy before). Therefore, every attempt of moving a nongrid vertex will immediatelly fail, rejecting the resulting state and effectively wasting the iteration. Instead we now only ever sample from the rounded vertices. Runing the experiments on $(19, 40\%, 100\%)$ instances – complete Delaunay triangulations on 160 vertices – and testing for changes compared to sampling from all vertices, we failed to see a significant difference – all z-scores were below 1. Since we were under the impression that sampling only from the rounded vertices worked well on real-world instances, we look further into this. To do so, we created 1000 instances of lower edge density, namely $(19, 40\%, 40\%)$, to resemble actual road networks. Performing the same experiments as before, we report the following findings: Vertex-density annealing got significantly faster ($z \approx 4.01$) while being just shy of producing significantly cheaper feasible drawings ($z \approx 1.427$). On the other hand, the solutions produced by Grid-density annealing got significantly cheaper

($z \approx 3.30$), but we could not obtain the same speedup ($z \approx 0.389$). On these instances, we also observe that the speed difference between both Stage One procedures becomes insignificant ($z \approx 1.011$) while Grid-density annealing is still cheaper ($z \approx 17.85$). This also supports the claims we made when first comparing the two options in Section 6.4.2: indeed, Grid-density seems to thrive on easier and less dense instances.

Final Conclusion on Stage One. To sum up our findings on the different options for Stage One, we ran a final experiment on the 1000 instances of $(19, 40\%, 100\%)$ with all options enabled: Cartogram preprocessing, Incremental Greedy at every Iteration, and nonuniform vertex sampling ignoring nongrid vertices. Recall that the first sets of solutions had average costs of 347.208 for the Grid-density annealing and 407.916 for the Vertex-density annealing. All modifications described and evaluated above bring these numbers down to 183.082 and 221.572 respectively, saving almost 50% on the total cost of the first feasible solution and doing so in less time.

6.4.3 Evaluating Stage Two

Now that we demonstrated how to find a reasonable initial solution, we consider Stage Two. After Stage One tried to get away from the input drawing to find a less dense but feasible drawing, Stage Two now anneals back towards the initial vertex positions. Annealing theory [KGV83] suggests that a system provided with the right cooling schedule and enough time will eventually end up in a low cost state. It is unclear how to determine that this state is reached, and a schedule that is guaranteed to achieve it with high probability is impractically slow; in practice, this requires experimental tuning.

We note that our moves from one drawing to the next are rather small, in terms of objective value: a single vertex is moved at most one grid cell, resulting is a maximum change in cost of $\sqrt{2}$ if moved diagonally. With the typical starting at temperature of $T_0 = 1.0$, this means we initially accept at least 24.3% of all cost-increasing moves. Similar to Section 6.4.2, we will evaluate various cooling schedules, step counts on large instances, and discuss when to switch from annealing to the hill climbing postprocessing as described in Section 6.3.3.

Finally, we note that unfortunately we cannot compare to optimal solutions for any interesting instances: As we have seen in Section 5.4, known exact methods are wildly infeasible and only able to handle networks so small that the comparison is not very interesting.

Cooling Schedules One of the most important parameters – besides number of iterations – for any simulated annealing process is the cooling schedule. We use the typical exponential cooling schedule $T_{i+1} = c \cdot T_i$ which naturally gets slower over time. Therefore, we requiring a choice for initial temperature T_0 and cooling factor c. We evaluate this on 200 instances with parameters $(14, 40\%, 40\%)$ and 200 instances with parameters $(14, 40\%, 100\%)$. In order to eliminate any randomness from re-running Stage One, we use precomputed feasible solutions using Vertex-density annealing. We ran 20000

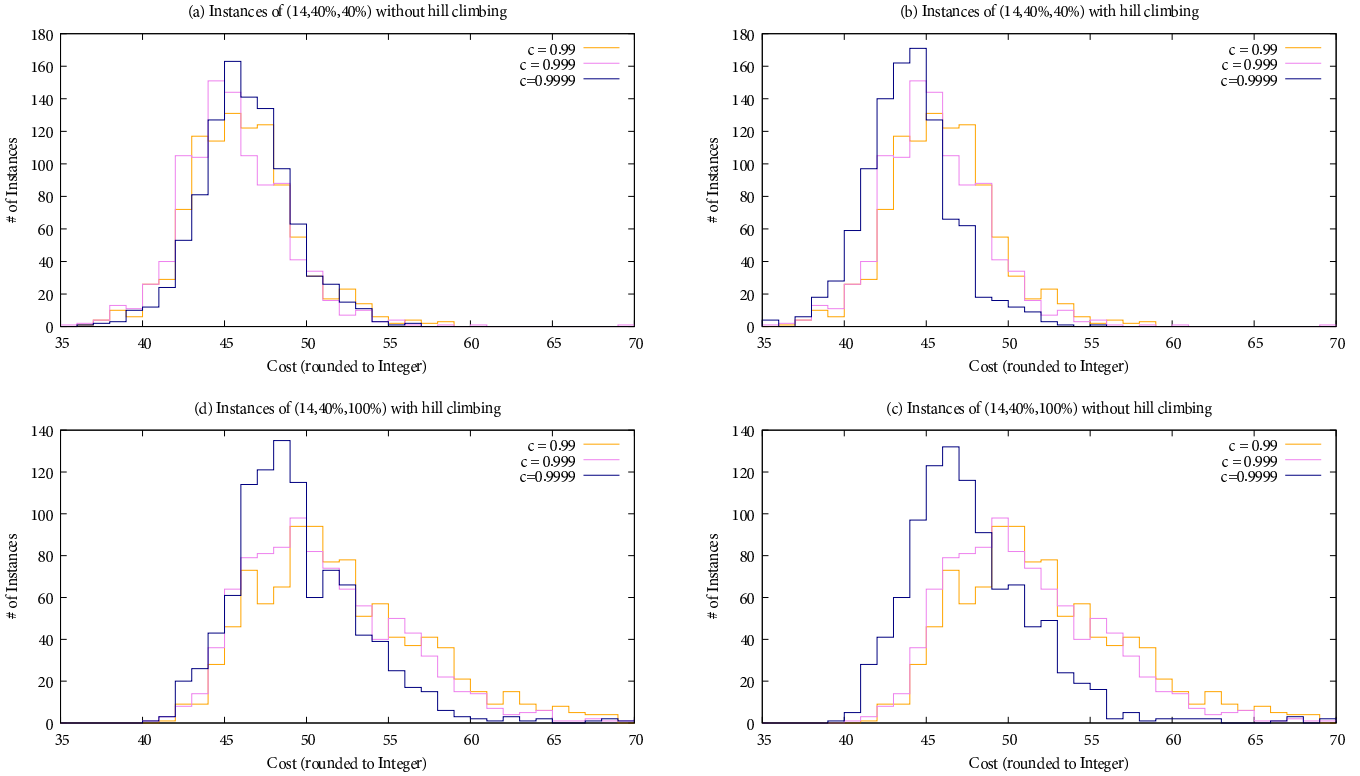


Figure 6.5: Effect of cooling factor on final total cost. Each histogram combines five runs on 200 instances, results are packed into bins of integer size. The top row shows costs of $(14, 40\%, 40\%)$ instances (a) before and (b) after hill climbing; the bottom row shows costs of $(14, 40\%, 100\%)$ instances (c) before and (d) after hill climbing. (Single outliers beyond cost of 70 ignored.)

steps in Stage Two (followed by hill climbing postprocessing) with $c_1 = 0.99$, $c_2 = 0.999$, and $c_3 = 0.9999$, doing five runs of each setting on each instance. Figure 6.5 shows a histogram of the resulting cost (rounded to the nearest integer). First consider Subfigures 6.5 (a) and (b), the instances with rather number of edges: before hill climbing (a), the choice of cooling factor does not seem to have huge impact on the quality; afterward (b), the advantage of the slower cooling provided by $c = 0.9999$ becomes visible. Looking at how the systems temperature developed over time, the reason is as follows: the schedule for $c_1 = 0.99$ reaches a temperature of effectively 0 after only 1500 iterations, compared to the 12000 iterations it takes for colling with $c_2 = 0.999$ to reach the same temperature. This means that for both parameter values, a significant amount of time was spent rejecting any move that does not strictly improve the cost: they were basically hill climbing random vertices one step at a time. This behavior is also visible in the plots: There is hardly any difference between the orange and red lines in Subfigures 6.5 (a) and (b). This is not true for the slowest schedule ($c_3 = 0.9999$), as even after 20000 iterations the temperature was still about 0.135 – while accepting is not very likely in the end, this schedule never rejected score-decreasing moves immediatelly, leaving room for improvements to be made by hill climbing. The same phenomenon can be observed looking at the second set of instances (Figure 6.5 (c) and (d)). Again, hill climbing basically did not find any improvements for c_1 or c_2, whereas c_3 managed to avoid running into local optima long enough for Hill Climbing to make a difference. The cost improvement of hill climbing on both test sets are highly significant ($z > 50$ each).

With these observations in mind, we recommend always using the hill climbing as postprocessing; it is fast and guaranteed to not worsen the final solution as hill climbing only performs a set of moves that would have been accepted by annealing even at temperature 0.

Step Count & Hill Climbing To evaluate the effect of hill climbing postprocessing, we generated feasible solutions for all of the $(19, 40\%, 100\%)$ instances from Section 6.4.2 using Vertex-density annealing with cartogram preprocessing.

On these rather large and dense instances, we run Stage Two annealing for m steps ($m \in \{0, 2500, 5000, 10000, \ldots, 40000\}$), followed by hill climbing, reporting score with and without postprocessing. To demonstrate the impact of step count – and with respect to the experiments above –, we choose the slowest of the cooling schedules above, namely $c = 0.9999$. Indicative results of these experiments are shown in Figure 6.6.

We deal with instances of maximum edge density that result in rather expensive Stage One solutions (see Section 6.4.2). This leaves quite some space for a lot of improvements – we can expect "good" final solutions to have cost below 100, or less than 1 per vertex on average. We now discuss the four subfigures of Figure 6.6 individually.

As noted before, annealing with temperature 0 yields very similar results to hill climbing. After 5000 steps, the system is still at temperature 0.607, whereas after 40000 steps, it reaches a final temperature of 0.0183. Also recall that for high temperatures, the acceptance propability of Stage Two is relatively high (between 24.5% and 16.4% for the first 2500 steps). A downside of combining a slow cooling schedule with greedy local

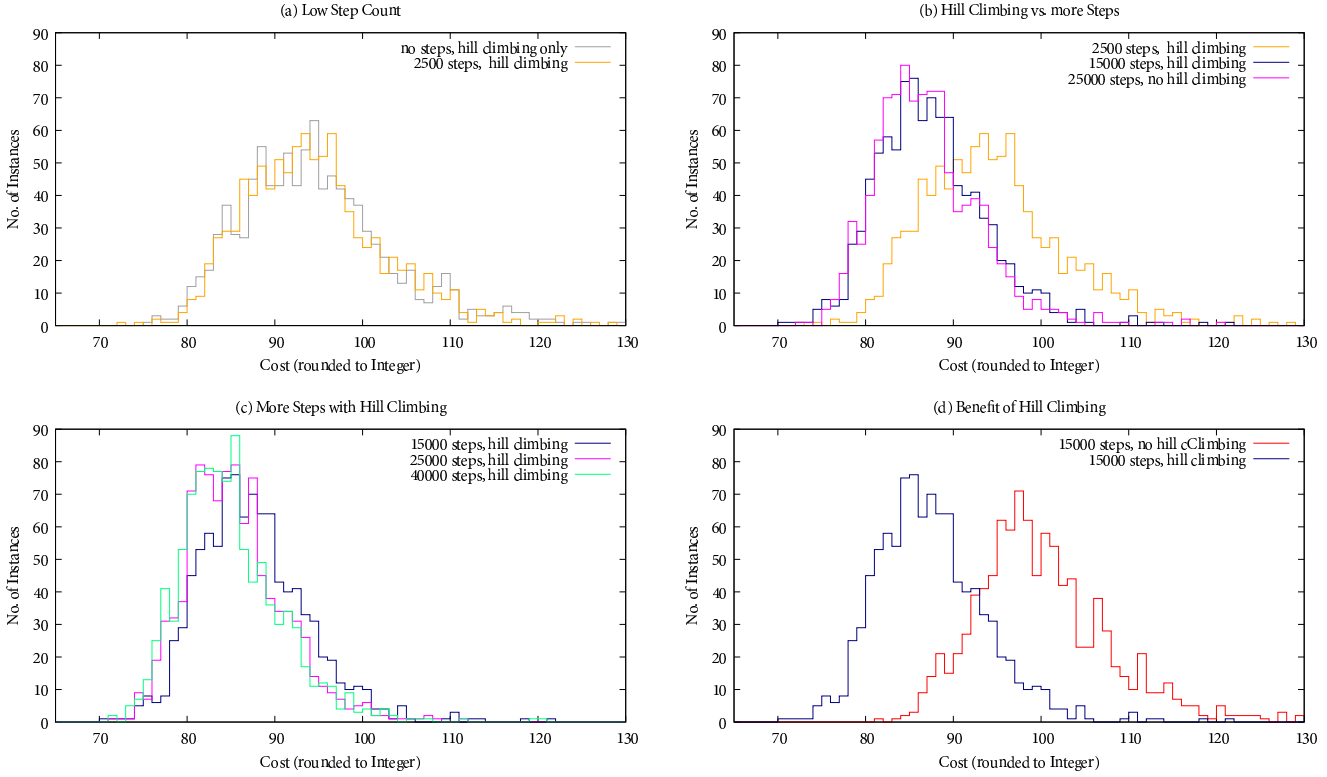


Figure 6.6: Various values of step count m for Stage Two and the effect on cost of $(19, 40\%, 100\%)$ instances with and without hill climbing: (a) $m = 0$ vs. $m = 2500$, both with hill climbing; (b) the trade-off between hill climbing and running Stage Two for more steps; (c) multiple values for $m \in \{15000, 25000, 40000\}$, with hill climbing; (d) $m = 15000$, with and without hill climbing. Some outliers have been cropped; for comparison, line colors correspond to datasets and carry over into other plots.

optimization is visible in Figure 6.6 (a). For the orange curve, we spent 2500 iterations randomly moving away from the first feasible solution, which hill climbing sometimes could not repair.

Figure 6.6 (b) compares $m = 15000$ steps with hill climbing to $m = 25000$ steps without postprocessing. For reference, we also include the data for $m = 2500$ from Subfigure (a). Notice that doing 22500 extra steps in Stage Two leads to lower costs in the final drawing, even without hill climbing. However, picking the point at which to switch from annealing to hill climbing carefully can lead to similar results in shorter time, as shown by the blue line.

Figure 6.6 (c) shows how the total movement cost changes with increasing step count. There is still a noticeable difference between high and low step counts. In addition, even after 40000 iterations of annealing, hill climbing managed to make some progress on 874 of the 1000 instances.

Figure 6.6 (d) illustrates the impact hill climbing can have on the total cost of a solution. The improvement made by hill climbing is obvious and easy to explain – Stage Two simply was not done yet. Nevertheless, the final result after postprocessing is comparable to those of higher iteration counts.

6.4.4 Real-World Data

In this section we compare real-world instances to artificial networks and evaluate our approach further. The results presented here were obtained by experiments we ran on a 2.6 GHz processor with sufficient RAM. See Table 6.2 for instance parameters and sizes and Table 6.3 for experimental results on these instances; all data was gathered over 100 runs on each instance, reporting average scores and times, together with standard deviations over these measurements. More data on more instances is publicly available online[7].

The instances were preprocessed using linear edge cartograms and we choose Vertex-density annealing with all options enabled for Stage One. Examine the instances and the computed grid drawings of Würzburg–Train Station, found in Figure 6.7 (a). The solutions of Würzburg–Train Station we computed had an average cost of 110.9 (0.860 on average per vertex, standard deviation of 8.66); notice that for these runs, we picked the vertex-grid point ratio $\gamma = 16.45\%$. The average cost over 100 runs on artifical instance 0.4_0.4_42 – resembling this real network by picking ε accoringly, but with $\gamma = 40\%$ – is 77.2 (0.483 on average per vertex, standard deviation of 1.92). While Würzburg–Train Station is less dense and thus could be expected to be easier, neither cost nor runtime reflect this; Train Station takes 63% longer on average (43.1 s vs. 26.4 s) and is 43% more expensive. To investigate this, consider that the left part of Fig 6.7 (a) (marked by the blue arrow) is significantly more dense than the rest; moving the tilted bus parking lanes to the grid results in parts of them being pushed outwards. The vertex indicated by the blue arrow (together with the middle of its incident edges) force significant distortion

[7] `http://github.com/tcvdijk/armstrong`

Table 6.2: Sizes of selected real and artificial instances. Würzburg–Train Station is shown in Figure 6.7 (a), Bus Station is the left half of Train Station; Chicago–Downtown and –Cloud Gate can be found in Figure 6.7 (b) and (c). Würzburg–Ring and parts of United Kingdom–Borders are shown in Figure 6.1 (a) and (b) respectively (Würzburg–Ring is shown on a finer grid of size 28×28).

Name		v	ε	size	γ
Würzburg	–Train Station	129	40.3%	28×28	16.5%
	–Bus Lanes	89	41.7%	15×15	39.6%
	–Ring	138	38.8%	20×20	34.5%
Chicago	–Downtown	358	35.2%	25×25	57.3%
	–Cloud Gate	578	35.1%	240×240	1.0%
United Kingdom	–Borders	3110	34.8%	250×250	5.0%
Artificial	–0.4_0.4_42	160	40%	20×20	40.0%
	–0.4_0.4_84	160	40%	20×20	40.0%
	–0.4_1.0_42	160	100%	20×20	40.0%

in this drawing. While there seem to be quite some empty grid points nearby, topology prohibits any local improvement for the vertex marked by the orange arrow and on closer reflection, this seems to be a decent drawing of this instance on a critically small grid. To support this observation, we extracted the dense part: Würzburg–Bus Lanes alone at $\gamma = 40\%$ is comparable to artificial instances of about twice its size on runtime and rounding cost.

This problem is even more obvious for Chicago–Cloud Gate, shown in Figure 6.7 (c). Note that the path to the bottom-right of the instance has way too many vertices compared to the grid size. Hence we get a rather skewed grid representation eventhough the other roads around the problematic path are generally represented quite well (and would have tolerated an even coarser grid). The average vertex cost after Stage One (3.62 per vertex) indicates how long it took to find any feasible drawing. This suggests that future work could attempt to integrate topologically-safe simplification into the grid representation workflow.

Again in Tables 6.2 and 6.3, consider the artifical instances 0.4_0.4_42 and 0.4_0.4_-84: different networks with the same parameters. The former is immediately solved by Incremental Greedy during the first iteration of Stage One, so the result is deterministic. For this instance, this was only the case when cartogram preprocessing was enabled, demonstrating its benefit. Though the average runtime on the two instances is comparable, the final drawings of the latter have higher variance. This suggests that a deterministic Stage One may be preferable, as the results obtained from Stage Two will become are more stable.

All this indicates that the performance of our algorithm is sensitive to the structure of the network, and that the particular road network representations we found in OpenStreetMap and the City of Chicago Data Open Data Portal are challenging instances. Still,

Table 6.3: Experimental results on the instances shown in Table 6.2; all numbers are the average over 100 runs with the same parameters. "Runtime" is given in seconds, single-threaded; "m" is the number of iterations in stage two; "S1 cost" and "S2 cost" are total rounding cost after Stages One and Two respectively; σ_t, σ_{S1} and σ_{S2} are the corresponding standard deviations on the measurements.

m	time	(σ_t)	S1 cost	(σ_{S1})	S2 cost	(σ_{S2})
20000	43.2 s	(14.9 s)	486.2	(119.9)	110.4	(7.20)
20000	23.9 s	(4.73 s)	242.0	(59.7)	72.6	(6.23)
20000	60.7 s	(9.54 s)	392.5	(67.2)	115.4	(4.94)
20000	83.3 s	(18.3 s)	1462.7	(106.8)	390.4	(16.0)
35000	492 s	(119.4 s)	2090.8	(313.1)	356.6	(8.69)
50000	590 s	(282.3 s)	5817.3	(938.5)	1344.2	(11.1)
20000	26.4 s	(0.88 s)	135.7	(0)	77.2	(2.15)
20000	25.6 s	(1.5 s)	205.9	(50.4)	77.9	(3.18)
20000	63.7 s	(11.5 s)	454.6	(181.9)	94.7	(10.1)

our method is able to find reasonable representations of realistic networks, which was not feasible with previous methods.

6.5 Conclusion

In this chapter, we settled one of the open questions from Chapter 5: We provided an efficient heuristic to the Topologically-Safe Grid Representation problem. It relaxes on the minimality of the total vertex displacement in the output – finding non-optimal, yet reasonable solutions – while maintaining topological equivalence of input and output drawing. This heuristic was designed to transform geographic networks into drawings requiring lower coordinate precision. To do so, we proposed a two-stage approach based on the simulated annealing metaheuristic. The first stage anneals for feasibility by reducing the overall vertex-density of the drawing; the second stage anneals the feasible drawing to reduce the total rounding cost of the output.

We demonstrated the necessity of our two-stage approach by experimental evaluation. For stage one, we proposed two different objective functions to anneal for; testing their performance on randomly generated artificial road networks, we were able to provide significance tests to show the strong and weak points of both. For stage two, we provided experiments demonstrating the impact of parameter selection to the final result. We proposed pre- and postprocessing steps and showed via significance testing that both procedures improve the result of their respective stage significantly.

We concluded the experimental evaluation of our algorithm by testing it on hand-picked real world instances of various sizes.

Acknowledgments. We thank Thomas C. van Dijk for providing high-quality implementations of the methods and procedures described and discussed in this Chapter in C++, and for making datasets and the implementation publicly available at: `https://github.com/tcvdijk/armstrong`.

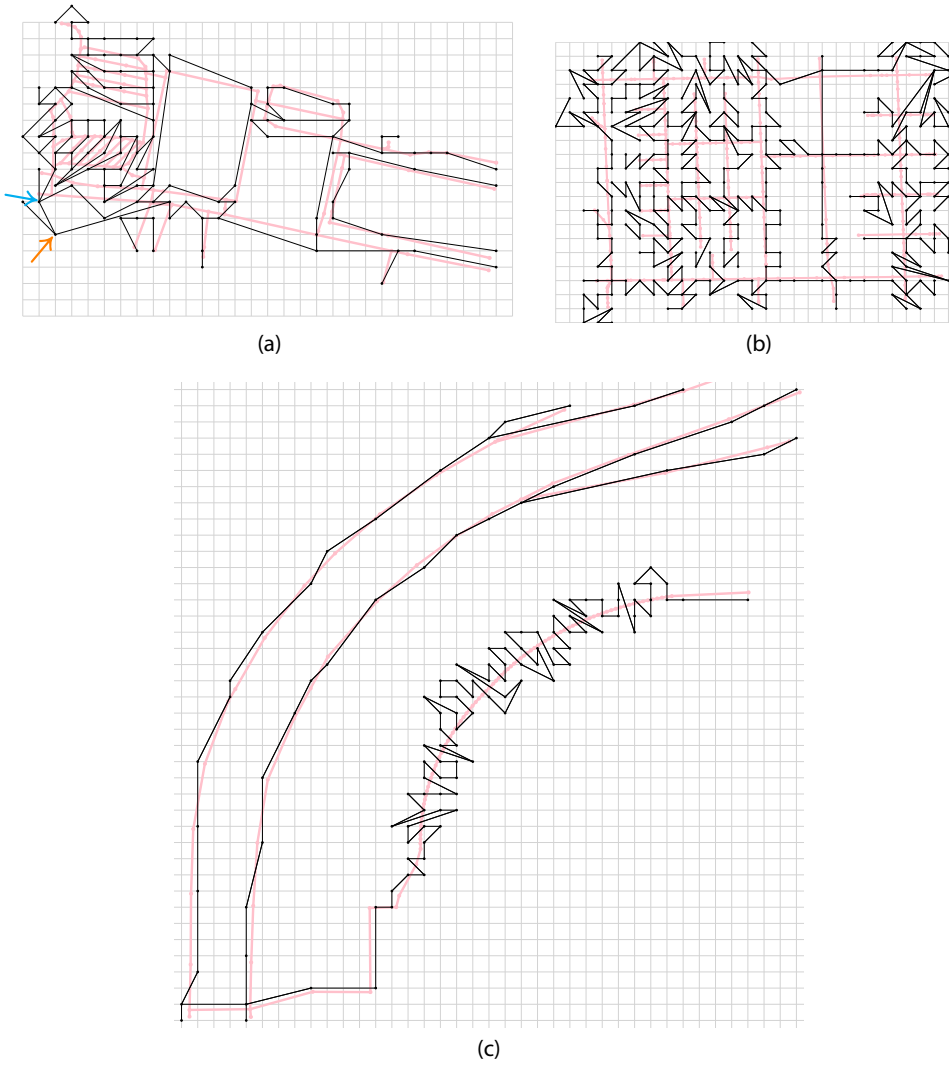


Figure 6.7: Grid representations of real-world instances. (a) Würzburg–Train Station on a grid of size 28×28; the vertex indicated by the orange arrow looks highly suboptimal. (b) Chicago–Downtown on a grid of size 25×25; (c) a crop of Chicago–Cloud Gate on a grid of size 240×240.

Chapter 7

Cauchy's Theorem for Orthogonal Polyhedra in $3D$

A classic theorem by Cauchy states that, for convex polyhedral surfaces, when the embedded graph of the surface and the angles within each face are given, the dihedral angles are determined. In this chapter, we translate Cauchy's rigidity theorem to orthogonal polyhedral surfaces of arbitrary genus. We do so by using the stable linear-time BundleOrientation algorithm by Biedl and Genç [BG09] as a subroutine. They originally created it to determine the unique set of dihedral angles of orthogonal polyhedral surfaces of genus 0 with connected graph (if this set exists).

They left open the question whether a similar translation exists for orthogonal polyhedral surfaces of higher genus, which we answer in this chapter in the affirmative. By repeated exhaustive application of the original BundleOrientation algorithm, we obtain the IteratedBundleColoring algorithm.i We show that it is capable of finding a set of dihedral angles for orthogonal polyhedral surfaces of arbitrary genus. We do so by arguing how it re-discovers the dihedral angles matching those of a polyhedron realizing the input graph.

7.1 Definitions and Motivation

We begin this chapter with formally defining the structures we consider to then precisely state the question we answer. In the following, we look at two different types of angles.

Given a single polygonal face of a graph, the *facial angles* are the angles at which two consecutive edges meet; for two adjacent flat polygonal faces in three-dimensional space, the *dihedral angle* is the angle measured along the edges they share. Moreover, all edges shared by two polygonal faces are colinear: they are on the line defined by the intersection of the two planes that the faces live in.

A *polyhedron* is a solid object in three dimensions. Looking at a polyhedron, we see straight edges and sharp point-shaped corners on its exterior. The edges form a set of induced cycles, creating two-dimensional polygons on the surface. These polygons define the *faces* of the polyhedron. Joining all polygons, we obtain the polyhedral surface bounding it – the *genus* of a polyhedron is the genus of the bounding surface.

A polyhedron induces a combinatorially (and geometrically) embedded graph via its boundary – the faces, edges, and vertices of the graph correspond to those of the polyhedron. Moreover, the relative placement and orientation of the connected components in

the graph are specified by the polyhedron. Similarly, a *polyhedral surface* in three dimensions is a closed connected orientable mesh of polygonal faces – the difference is that adjacent faces can be parallel. A polyhedral surface also induces an embedded graph, which we call a *net*. By convention, nets are connected [Zie08]. Note that, as remarked by Ziegler [Zie08], the classification of polyhedra and polyhedral surfaces up to homeomorphism is well-known: For each integer $g \geq 0$, there is exactly one topological type, *the surface of genus* g, obtainable by attaching g handles to the 2-sphere $\mathbb{S}^2$.

A classic topic in geometry is polyhedra (and polyhedral surfaces) with various geometric restrictions, the most well-studied case being the *convex* polyhedra where the vertices occur in convex position. For example, convex polyedra were a common topic of Euclid, Cauchy, and Steinitz. Recently several classic results on convex polyhedra have been considered for *orthogonal* polyhedral surfaces, for which every edge is parallel to one coordinate-axis; here, all facial and dihedral angles are multiples of 90°, and (without loss of generality) each face is perpendicular to one coordinate-axis. We outline some of these results next.

7.2 Related Work

Classical Convex Results vs. Recent Orthogonal Results. *Cauchy's rigidity theorem* states that, when constructing a convex polyhedron from a 3-connected planar graph, the facial angles determine the dihedral angles uniquely; proofs can be found in several textbooks, for example, see Aigner and Ziegler [AZ04]. This theorem breaks down for non-convex polyhedra. Consider an object composed of two six-sided polyhedra – one large and one small – with the smaller one attached to one side of the larger one (the faces they are connected at are coplanar and the smaller face is contained in the interior of the larger face). Only knowing the net of this combined polyhedron, it can be realized as a polyhedron in the two distinct ways shown in Figure 7.1 (a): adding the smaller polyhedron to the larger (orange drawing), creating a "bulge", or subtracting the smaller polyhedron (blue drawing), creating a dent. Both variants are valid nonconvex realizations, creating different dihedral angles. Moreover, there are even *flexible* polyhedra where the dihedral angles can vary continuously while maintaining the edge lengths or facial angles [Con79]. An analogue of Cauchy's rigidity theorem for orthogonal polyhedral surfaces of genus 0 is shown by Biedl and Genç [BG09] where the graph is connected. Note that, like convexity, the connectedness is necessary for the uniqueness of the dihedral angles. This is illustrated in Figure 7.1 (a): When the two joining faces touch in one edge – connecting the graph –, subtracting the smaller polyhedron would create degenerate object that has a "boundary-only" section without interior at the other side of the joined edge. In an extended version [BG08] of [BG09], Biedl and Genç show that testing whether a given embedded planar graph can be realized as an orthogonal polyhedral surface where every face is a unit square is $\mathcal{NP}$-complete. The gadgets utilize disconnectedness of the graph.

Steinitz' theorem states that the 3-connected planar graphs are precisely the graphs obtainable from the surfaces of convex polyhedra. A graph-theoretic characterization of orthogonal polyhedra of genus 0 where three mutually-perpendicular edges meet at each vertex has also been proven by Eppstein and Mumford [EM14].

Stoker's theorem states that, when constructing a convex polyhedron from a 3-connected planar graph, the dihedral angles and edge lengths determine the facial angles uniquely. However, Bield and Genç [BG11] showed that for embedded graphs with given edge lengths and dihedral angles, it is $\mathcal{NP}$-hard to decide if there is an orthogonal polyhedral surface of genus 0 realizing this input.

A related topic is whether having facial angles restricted to multiples of 90° forces all realizations to be orthogonal polyhedral surfaces. For example, Biedl et. al [BLS05] asked whether every polyhedron in which every face is a rectangle is an orthogonal polyhedral surface. Their question was answered in the negative by counterexamples of genus 7 [DO02] and genus 6 [BCD$^+$02]. On the other hand, the question has a positive answer for genus at most 2 [BCD$^+$02, DO02]. The cases of genus 3, 4, and 5 remain open.

Our main result – stated in Theorem 7.1 below – builds upon the work of Biedl and Genç [BG09] from orthogonal polyhedral surfaces of genus 0 to all orthogonal polyhedral surfaces. They already give the two-step strategy that we extend upon in this chapter.

In the first step, they show that for each face f, the axis perpendicular to f is uniquely determined by a given embedded *genus-0 (planar)* graph with specified facial angles (up to relabeling of the axes). Note that this first step already determines all of the "flat" dihedral angles, i.e., the parallel adjacent faces, and as such provides the unique graph of every realization (obtained by merging parallel adjacent faces).

The second step consists of two small substeps. They observe that connectedness of the graph and knowing the axis perpendicular to each face implies that there can be only two possible realizations of the dihedral angles, basically regarding which side of the faces is the "outside" and which side is the "inside" of the surface. With this observation in mind, they further note that having fixed edge lengths fixes (up to translation) all of the vertex positions in both sets of dihedral angles[1]. Thus, from this set of vertex coordinates they identify the correct set of dihedral angles by looking at the incident dihedral angles of a face f perpendicular to the x-axis with the maximum x-coordinate (f's dihedral angles must all be 270°). Finally, they note that quadratic time suffices to check that the constructed object avoids self-intersections using an existing algorithm [BLS05]. This resolves the genus-0 case.

Interestingly, as stated by Biedl and Genç [BG09], the second step does not require a genus-0 input, and can be applied (regardless of genus) as long as one is given the axis perpendicular to each face. Thus, to generalize their result from genus 0 to arbitrary genus, it suffices to generalize the first step of their approach. They give the original BundleOrientation algorithm – a conservative propagation algorithm, which we describe in Section 7.3 – and show that (in the case of a realizable input) it's output uniquely

[1] This can result in one vertex needing two distinct positions, in which case the instance is not realizable.

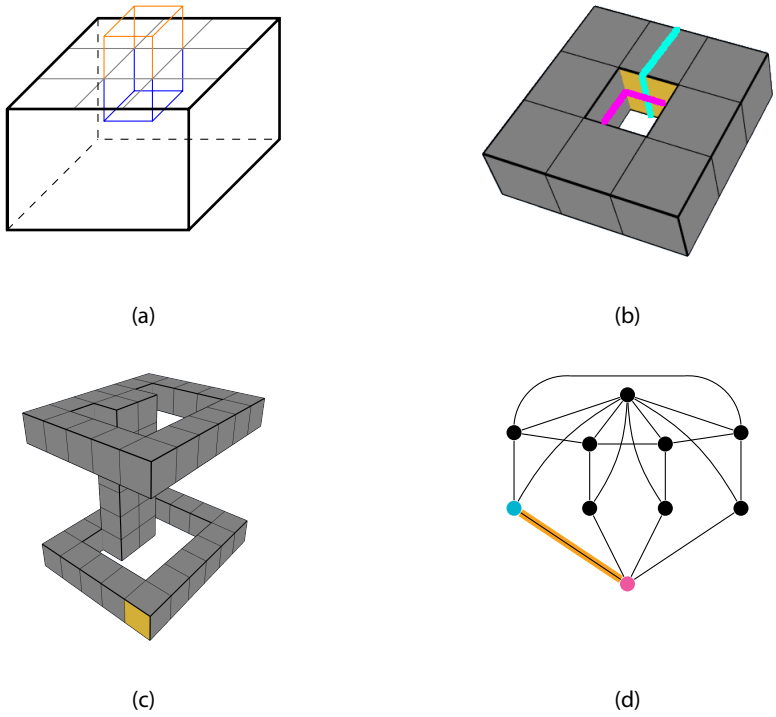


Figure 7.1: (a) Polyhedron with disconnected graph, the small box can be pushed out (orange) or dented in (blue). (b) A polyhedron of genus 1 with a bad starting face in yellow (one of four). The face's bundles (in purple and turquoise) cross only once, stopping propagation. (c) A polyhedron of genus 2 composed of two tori connected by a bridge going from the inside of one to the inside of the other. Starting at the yellow face orients the bottom torus but not the top one. (d) The underlying bundle graph $\mathfrak{B}_G$ of the polyhedron from (b), with matching colors identifying the same objects.

determines the axis perpendicular to each face with respect to some *starting face*. They demonstrate that their approach can fail to determine the axis for every face already for realizable inputs of genus 1 (see Figure 7.1 (b) and further details in Section 7.3). Additionally, there are examples of genus 2 (e.g. Figure 7.1 (c)) where no single face is sufficient to resolve the axis of every face via their algorithm. We discuss this in Section 7.3.

Contribution. The main result of this chapter is generalizing the main theorem of Biedl and Genç to also cover orthogonal polyhedra of arbitrary genus. To do so, we proof the following extended theorem:

Theorem 7.1. *Given an embedded graph G with facial angles F and edge lengths L*

1. *we can report in cubic time either that no orthogonal polyhedral surface realizes the given graph and facial angles or determine the unique coordinate-axis perpendicular to each face; and*
2. *if 1 does not reject the instance, then in additional linear time we report:*
   - *that this graph and facial angles can only belong to an orthogonal polyhedral surface for which the polyhedron bounded by it has a disconnected graph; OR*
   - *that no orthogonal polyhedral surface realizes (G, F, L); OR*
   - *the unique set of dihedral angles of any orthogonal polyhedral surface that has this graph, facial angles, and edge lengths.*

We overcome the difficulty of finding the right starting faces by describing how to use their BundleOrientation algorithm in an exhaustive fashion – obtaining the IteratedBundleColoring algorithm that progressively learns the orientations of faces, see Section 7.3. The correctness of our approach is presented in Section 7.4. There we consider an input graph and facial angles together with a hypothetical polyhedron that realizes it. We traverse the surface of this realization "layer-by-layer", maintaining the invariant that the next layer can be oriented if and only if the previous layer was oriented and matches the realization and that IteratedBundleColoring is bound to eventually find that orientation. This invariant is stated in Lemma 7.4, which we will split into sub-cases for all possible different two-layer patters, providing and individual lemma for each. By arguing that the output of IteratedBundleColoring is *stable* – that is, it is unique up to renaming the equivalence classes –, we can use this output directly as input for the original step two by Biedl and Genç, implying correctness of Theorem 7.1.

7.3 Orienting Faces by Coloring Edges

In this section we provide the IteratedBundleColoring algorithm to accomplish the following task. Input for the algorithm is a connected and embedded graph G (i.e., including the cyclic order of the edges around each vertex, and the corresponding faces) as well as the set F of facial angles, each of which is a multiple of 90°. If there is an orthogonal polyhedral surface realizing (G, F), then the algorithm will determine an *orientation* of each face, i.e., for each face f, an axis perpendicular to f or, equivalently, the *axis-plane* parallel to f will be specified. Moreover the orientations are obtained in a *stable* way: Up to renaming of the axes, every orthogonal polyhedral surface realizing (G, F) has these orientations.

The algorithm builds upon the reasoning used by Biedl and Genç [BG09] for the case of genus 0, modifying it as follows. Rather than directly working with the faces of G, it works on the *bundle graph* $\mathfrak{B}_G$. For any orthogonal face f, the edges bounding f

can be partitioned into two *(edge) bundles*. If a pair of edges is parallel (with respect to parity and the facial angles at f's corners), they are in the same bundle; otherwise, they are in different bundles. This implies that the edges of two bundles from the same face are mutually perpendicular. If edges of two bundles appear on the same face, we say that those bundles *cross* there. By definition, each edge e bounds exactly two faces and thus, in two (possibly) different bundles. As parallelism is transitive, all edges of both such bundles are parallel to e, allowing the bundles to *merge* into one. That way we have a partition of the entire edge set of G into a collection $\mathcal{B}$ of bundles. This gives us the *bundle graph* $\mathfrak{B}_G = (\mathcal{B}, \mathcal{E})$ with vertex set $\mathcal{B}$ and an edge $e \in \mathcal{E}$ between two vertices, if there is a face containing edges of both bundles.

Looking for an orthogonal polyhedron, we aim to partition the edges into three color classes depending on the coordinate-axis to which they are parallel. BundleColoring picks an edge of $\mathfrak{B}_G$ (a face of G) and colors the two bundles connected by it differently. It then repeatedly looks for a triangle in $\mathfrak{B}_G$ with two colored corners, coloring the third corner in the remaining color, until all of $\mathfrak{B}_G$ is colored or no such triangle is found anymore. As a shorthand, we use BundleColoring($\mathfrak{B}_G, e$) to say we run BundleColoring on bundle graph $\mathfrak{B}_G$ with starting edge $e \in \mathcal{E}$. Then, any 3-coloring of $\mathfrak{B}_G$ provides an orientation of the faces of G.

Remark 7.1. If the input graph G consists purely of rectangular faces, then the corresponding bundle graph has a natural intuitive structure. Each bundle corresponds to a cycle of faces obtained by tracing the outline of the surface along a path parallel to one of the axes – see Figure 7.1 (b) for the surface and (d) for the corresponding bundle graph (using the same colors).

They showed that for realizable instances (G, F) where G has genus 0, this simple conservative coloring procedure always completely colors $\mathfrak{B}_G$, and hence the orientation of the faces in any realization of (G, F) is unique up to naming of the axes. Their algorithm is easily implementable in time linear in the size of the bundle graph, which is linear in the size of G. Biedl and Genç observed that BundleColoring can fail to color all the bundles when the graph G has genus 1. For example, if, in Figure 7.1 (b), we start the bundle coloring procedure on the edge corresponding to the (yellow) "inner" face, then it will not color any bundles beyond the original two. However, if a "corner" face is used instead, then all bundles will indeed be colored. As their algorithm starts with an arbitrary edge, it could not generally cope with such examples. Therefore, Biedl and Genç left the status of nonzero-genus inputs an open problem. In fact, already for genus 2 there are realizable instances where some bundles will remain uncolored after one execution of BundleColoring, regardless of the starting edge. For example, in Figure 7.1 (c), if we start from a face of one torus (e.g., the yellow face), when the color propagation reaches the other torus, the topology steers it so that it is as though we started on an inner face of the other torus. Thus, as in the genus-1 example, some bundles remain uncolored. This implies that a more involved approach is needed for genus larger than one.

For the case of arbitrary genus inputs, we designed algorithm IteratedBundleColoring, listed in Algorithm 7.1. It first constructs the bundle graph from (G, F) and

Algorithm 7.1: IteratedBundleColoring(Graph G, facial angles F)

```
𝔅_G = (ℬ, ℰ) ← bundle graph of G with facial angles F
while |ℬ| > 3 do
    𝔅_G^old ← copy of 𝔅_G                    /* Copy to track progress */
    foreach edge e ∈ ℰ do
        merge bundles in 𝔅_G using BundleColoring(𝔅_G, e)
    /* If nothing changed and there are still uncolored
       bundles, subsequent runs will also fail.              */
    if 𝔅_G^old = 𝔅_G then return Infeasible
return ℬ          /* ℬ is now a partition of the edges of G.  */
```

then repeatedly runs rounds of BundleColoring executions on the edges of the bundle graph as a subroutine. While doing so, it intermediately adjusts the bundle graph depending on the information learned from the previous runs. Namely, if a single run of BundleColoring$(\mathfrak{B}_G, e)$ on some edge $e \in \mathcal{E}$ does not color all of the bundles, but does manage to color some bundles, we can derive the following: All bundles that have received the same color must do so in any valid 3-coloring. Thus, for each $i \in \{1, 2, 3\}$, we merge the bundles with color i into a single bundle where the neighborhood of this bundle is simply the union of the neighborhoods of its members. The IteratedBundleColoring algorithm simply repeats such rounds until no bundles are merged in a round (since this implies that no further iterations would merge bundles). For the total runtime of IteratedBundleColoring, we have the following lemma:

Lemma 7.2. *On a graph G with m edges, IteratedBundleColoring runs in $O(m^3)$ time.*

Proof. Consider the total number of merge operations using BundleColoring. They will occur at most as many times as we have bundles to begin with, i.e., less than the number of edges in G. In each round, we run at most m executions of BundleColoring, which in turn takes $O(n + m)$ time on a graph with n vertices and m edges. Thus, after at most linearly many rounds, we will indeed have a round in which no bundles merge, resulting in a total runtime cubic in the number of edges. □

In Section 7.4, we will show that for realizable instances (G, F) of arbitrary genus, this simple iterated conservative coloring procedure results in a triangle (i.e., completely colors the bundles), and as such the orientation of each the face in any realization of (G, F) is unique up to naming of the axes. This will establish Theorem 7.1.

Remark 7.2. For the correctness of our approach, it suffices to consider input instances where each face is a unit square. In particular, when an instance (G, F) can be realized, the resulting orthogonal polyhedral surface can be tessellated so that every face is a unit square, providing a corresponding tessellation G' of G. Moreover, the bundle graph of

G' only contains less information (more separate bundles) regarding which edges of G must be parallel and which must be perpendicular. So, by arguing that even the edges of any such tessellation will be completely colored by our approach, we indeed establish that the edges of G will also be completely colored.

7.4 Arbitrary Genus: The Proof of Theorem 7.1

Following the remark given by Biedl and Genç [BG09][2] on their two-step proof structure, to prove Theorem 7.1, it suffices to prove the following theorem regarding step one.

Theorem 7.3. *Given a connected embedded graph G (of arbitrary genus) with facial angles F that are all multiples of* 90°, IteratedBundleColoring *will*

- *report that no orthogonal polyhedral surface can realize this graph and facial angles, OR*
- *report all edges of the graph for which the dihedral angles must be* 180° *in any orthogonal polyhedral surface that realizes this graph and facial angles.*

Recall that, by Remark 7.2, it suffices to prove this when every face of G is a unit square. With this in mind, we set up some terminology. Let G be a graph in which all induced cycles have length four, and let F be a set of facial angles that are all 90°. Together G and F imply that the surface of any orthogonal polyhedron P realizing (G, F) has to be tessellated using unit-square faces. That the faces are supposed meet at dihedral angles that are multiples of 90° implies that P itself – if it exists – needs to be composed of solid unit cubes.

7.4.1 Proof Outline

In the following we assume that (G, F) is a realizable instance and that P is the polyhedron realizing it. We will traverse the polyhedron in a top-to-bottom fashion, considering locally confined pieces of G. Each piece will correspond to a subset of the faces on P and we will argue how IteratedBundleColoring discovers the orientations of those faces only using information available locally within the piece. This process will yield that the orientation found on P is unique (up to rotation), because BundleColoring will only ever derive the coloring of a bundle when it shares faces with bundles of the two other colors.

Traversing object and graph. We assume that P is aligned to some three-dimensional coordinate system: At least one of the faces of P is coplanar to the xy-plane π_0 at the

[2] We already stated this remark at the bottom of Section 7.2.

origin[3] – imagine P standing on that face, growing upwards. Let t be the maximum z-coordinate of all points of P. By definition, the "topmost" faces of P (having a point with z-coordinate t) are coplanar to π_t. Similarly, all "bottom-most" faces (with a point with z-coordinate 0) are coplanar to π_0. Furthermore, any plane π_i with $i \in (0, t)$ crosses through the interior of P. This allows us to define the *i-th cross section* C_i of P, the set of cubes intersecting π_i. Obviously, C_x is empty for $x \notin [0, t]$. The term "cross section" is a misnomer here, as those are usually two-dimensional outlines created by intersecting a three-dimensional object with some plane, and not also three-dimensional objects. In fact, we will later also consider the two-dimensional outlines of those objects (projecting the cubes onto the xy-plane they are intersected by).

Since G is the net of P (by assumption), cross sections of P also "intersect" G. Each possible xy-plane π_i (with $i \in [0, t]$) defines a cross section that intersects some faces of G on the surface of P. That way, we obtain subgraph G_i from cross section C_i as follows. As realization P is aligned to the coordinate system and stands on π_0, there are three options for any face f intersected by cross section C_i:

- Face f is itself coplanar to xy-plane π_i;
- One edge of f is coplanar to π_i, and either "standing on" π_i (intersected by $\pi_{i+\frac{1}{2}}$)
- OR "hanging from" π_i (also intersected by $\pi_{i-\frac{1}{2}}$).

This allows us to limit the subset of all possible cross sections to consider to those defined by xy-planes π_i with integer $i \in \{0, 1, \ldots, t\}$ and the xy-planes with coordinates half-integer above/below[4] it, namely central cross section C_i and $C_{i+\frac{1}{2}}$ and $C_{i-\frac{1}{2}}$.

We say that a face f of polyhedron P is a *roof face* when it is coplanar to integer xy-plane π_i when it appears on a (solid) cube intersected by $\pi_{i-\frac{1}{2}}$ (below it). Symmetrically, a face of P is a *ceiling face* when it appears on a cube intersected by $\pi_{i+\frac{1}{2}}$ (above it). All other faces are perpendicular to the xy-plane and we call them *wall faces*. At each wall face f there are two crossing bundles – one containing the horizontal edges of f and the other containing the vertical edges of f. Given the alignment of P to the underlying coordinate system, we say that the former extends top-to-bottom. The latter extends parallel to the half-integer xy-plane intersecting f; we refer to such bundles as *outline bundles*, as they follow the polygonal outlines of the cross sections. The half-integer indexed cross sections only intersect walls while the integer indexed cross sections collect co-planar roof/ceiling faces together with their adjacent wall faces.

Defining pieces and patterns. Considering a triple of cross sections $C_{i+\frac{1}{2}}, C_i, C_{i-\frac{1}{2}}$ of P for some integer $i \in \{0, 1, \ldots, t\}$, we can define the subgraphs $G_{i+\frac{1}{2}}, G_i$, and $G_{i-\frac{1}{2}}$ of G. Each such subgraph consists of exactly those vertices and edges of G that belong to faces on the cubes composing P intersected by each cross section respectively (ignoring

[3] An xy-plane π_k is a plane spanned by the x- and y-axes at some constant z-coordinate k.

[4] $C_{0-\frac{1}{2}}$ and $C_{t+\frac{1}{2}}$ do not intersect P. We will discuss the triples involving either of them separately.

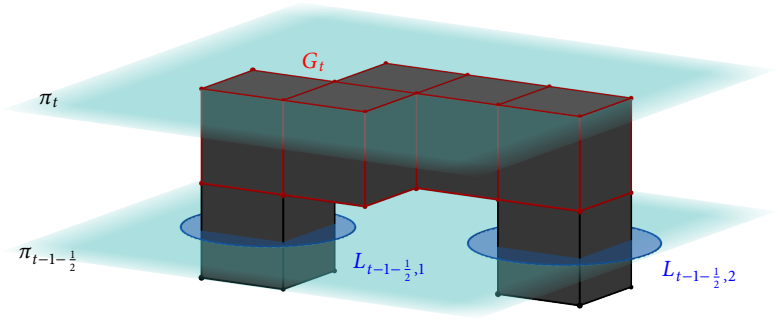


Figure 7.2: A simple polyhedron P of height 2 illustrating the notation: Plane π_t defines subgraph G_t (red), plane $\pi_{t-1-\frac{1}{2}}$ defines two disconnected layers $L_{t-1-\frac{1}{2},1}$ and $L_{t-1-\frac{1}{2},2}$ (blue). The faces of P contained in π_t are roof faces, the faces intersected by $\pi_{t-1-\frac{1}{2}}$ are wall faces.

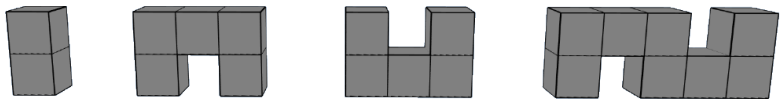


Figure 7.3: Patterns realizable by components of L_i, left to right: straight, split, merge, and mixed.

"interior" cubes not visible on the surface). The half-integer indexed subgraphs consist only of wall faces – as roof and ceiling faces can only be coplanar to integer indexed xy-planes –, and G_i contains both $G_{i+\frac{1}{2}}$ and $G_{i-\frac{1}{2}}$ as well as all roof and ceiling faces between them. Even though G and P are both connected, each such subgraph G_i can consist of multiple connected components $L_{i,1}, \ldots, L_{i,k} \in G_i$, which we refer to as *layers*. Since G_i is the graph induced by cross section C_i, we also write $L_{i,j} \in C_i$. In each layer $L_{i,j}$, we have $F_{i,j}$ as the set of faces on the intersected cubes. We use these notations for integer as well as half-integer indexed subgraphs. Notice that G_t consists of the highest roof faces P and their adjacent faces (hanging "down" from them), and G_0 consists of the lowest ceiling faces of P and their adjacent faces. An illustration on the relationship between planes, subgraphs and layers can be found in Figure 7.2.

In the following, we treat the layers of each subgraph G_i in isolation, referring to each as layer L_i – omitting the second index. We enumerate the set of five different possible patterns that we could encounter in L_i, depending on the numbers of layers in $G_{i+\frac{1}{2}}$ and $G_{i-\frac{1}{2}}$ present in L_i respectively. Patterns 2–5 (below) are illustrated in Figure 7.3.

1. In a *start* or *end* pattern, L_i contains layers of only $G_{i-\frac{1}{2}}$ or $G_{i+\frac{1}{2}}$ respectively.
2. In a *straight* pattern, L_i contains exactly one layer of both $G_{i+\frac{1}{2}}$ and $G_{i-\frac{1}{2}}$.
3. In a *split* pattern, L_i contains exactly one layer of $G_{i+\frac{1}{2}}$ but multiple of $G_{i-\frac{1}{2}}$.

4. In a *merge* pattern, L_i contains multiple layers of $G_{i+\frac{1}{2}}$ but exactly one of $G_{i-\frac{1}{2}}$.

5. In a *mixed* pattern, L_i contains multiple layers of both $G_{i+\frac{1}{2}}$ and $G_{i-\frac{1}{2}}$.

Proof invariant. To prove Theorem 7.3, we argue that IteratedBundleColoring *colors* the bundles of $\mathfrak{B}_G$. This coloring will be unique (up to renaming the colors), as we will never "guess" the color of a bundle but only derive it from having two differently colored neighbors. Actually, BundleColoring itself does not exploit the geometric nature of the problem but only works on the (unique) bundle graph of G. Having the coloring of $\mathfrak{B}_G$, we can obtain an *orientation* of the faces of G, marking each of them as perpendicular to one of the three coordinate axes.

In this analysis, we will argue how IteratedBundleColoring "puzzles" together the results obtained from multiple executions of BundleColoring – recall the loops in the code from Algorithm 7.1. We will only consider a selected subset of all executions[5], looking for bundles that have been colored by more than one of them. When we find two executions that both color the same two bundles, we say that those executions can be *synchronized*[6], merging the color classes (renaming those of one execution). During the analysis we will oftentimes argue how two (or more) layers can be synchronized.

While IteratedBundleColoring does not actually process $\mathfrak{B}_G$ in that order (since it does not know about the realization it will reconstruct), we will argue using executions of BundleColoring on faces of layers in a top-down fashion. The requirements on an upper layer allowing the algorithm to color the bundles of a lower layer are stated in Lemma 7.4 below. We will not proof Lemma 7.4 directly, but instead argue for all five patterns individually.

Lemma 7.4. *Let $\mathfrak{B}^*_G$ be the intermediate output of IteratedBundleColoring after some round and let $G_{i+\frac{1}{2}} = \bigcup_n L_{i+\frac{1}{2},n}$ and $G_{i-\frac{1}{2}} = \bigcup_m L_{i-\frac{1}{2},m}$ be the subgraphs of all upper and lower layers respectively. For each $i \in \{0, 1, \ldots, t\}$:*

1. *Given individual orientations of all upper layers in $G_{i+\frac{1}{2}}$ (the bundles of each layer have been merged to form triangles in $\mathfrak{B}^*_G$); then*

2. *the outline bundles of all layers in $G_{i-\frac{1}{2}}$ can be colored, synchronizing them to the upper layers' outlines,*

3. *which in turn enables coloring the remaining bundles of $G_{i-\frac{1}{2}}$, merging them to form a triangle in $\mathfrak{B}^*_G$.*

Our ultimate goal is to orient the faces of G. Therefore, in this analysis we classify faces by type – either 0, 1 or 2 –, depending on the number of bundles with already fixed color class, following the notation introduced by Biedl and Genç [BG09]. Again recall

[5] Unfortunately, we cannot identify this subset without the realization at hand. Therefore we choose exhaustive application of BundleColoring, eventually performing all required runs.

[6] This is something the algorithm finds eventually, we just argue that it has to happen.

that IteratedBundleColoring does not work with the faces of G directly; instead, we use the types of faces to discuss how much and what parts of a given layer have been processed. Property (1) of Lemma 7.4 implies that all faces in $F_{i+\frac{1}{2}}$ are of type 2; we will use this property as an *invariant* that we assume holds for all upper layers of any pattern.

Next, observe that this lemma indeed implies Theorem 7.3: Property (1) trivially holds for $i = t$. Moreover, for each layer within any integer-indexed cross section C_i, the bundles form a triangle in $\mathfrak{B}^*_G$, by Properties 2 and 3. Thus, as our net G is connected, it must be the case that $\mathfrak{B}^*_G$ is a triangle, and as such the orientation of every face of G has been determined.

7.4.2 Lemmata for the Patterns

In the following, we consider the patterns for any possible layer L_i of realizing polyhedron P individually, establishing a lemma similar to Lemma 7.4 for each. For now, we do not consider layers with *flat holes*. Layer $L_{i-\frac{1}{2}}$ has a flat hole, when there are two (or more) lower layers $L_{i-\frac{1}{2},a}$ and $L_{i-\frac{1}{2},b}$ with outlines in $\pi_{i-\frac{1}{2}}$, such that both lower layers are connected using roof and/or ceiling faces from L_i and such that the outline of one layer is contained in the outline of the other. The most simple flat hole is shown in Figure 7.1 (b): It has three cross sections C_1, $C_{\frac{1}{2}}$, and C_0. The middle cross section $C_{\frac{1}{2}}$ shows two outlines – one 3×3 units large, the other 1×1 (in purple) – with the larger containing the smaller and both of them connected by roof faces from C_1 (e.g. in blue). Notice that this does not imply restricting ourselves to genus-0 cases for now, as holes in planes perpendicular to xy-plane (and some other, more complicated cases) are still possible and will be handled implicitly.

To complete the analysis, we later introduce Lemma 7.11, describing how IteratedBundleColoring establishes Property 2 of Lemma 7.4 in the presence of flat holes. After coloring all bundles of a layer (Lemma 7.4 (3)), flat holes "disappear" from the bundle graph as all outside-inside outline pairs are colored the same and thus the bundles are merged in $\mathfrak{B}^*_G$.

Patches and patch spreading. In the following proofs we use the concepts of *patches* and *patch spreading*. A *patch* is a maximal connected set of faces of some subgraph G_i, defined by a "boundary"[7] of bundles with the same color: For each face of a patch, the two bundles $b^{(1)}$ and $b^{(2)}$ crossing there only extend across other faces of the same patch until eventually crossing one of these boundary bundles on either side. (As we will pick the patches to be coplanar to some plane π_i, these boundary bundles will in fact be outline bundles of upper and lower layers.) An illustration can be found in Figure 7.4 – even though we also provide a geometric interpretation there, the concept of patches is defined only on the faces of G and therefore present in the bundle graph.

[7] The boundary composed of outline bundles relates to the *bands* introduced by Biedl and Genç [BG09] In the case of hole-free start- and end patterns, the single outline bundle is indeed a band.

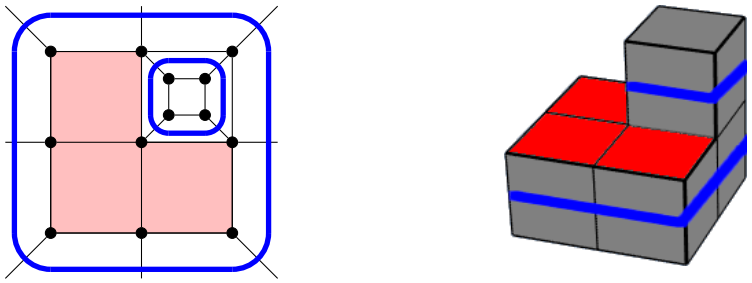


Figure 7.4: Illustration of a patch: (a) The subgraph of the net of a straight pattern; faces of the patch are drawn in red, bounding bundles in blue. (b) The same patch on the surface of the realization, forming a start pattern (with matching colors).

When the boundary bundles all have the same color – either by Property 2 of Lemma 7.4 or when in a start pattern – we can apply *patch spreading*[8]. Consider a single face of a patch – by definition, it's two bundles cross each other and also cross bundles of the third color class. Neighboring faces share one bundle and patches are connected, thus the following argument can be applied exhaustively, spreading over all faces of the patch: If a patch has some type 2 face, all of its neighbors are at least type 1 (from the shared bundle); since those neighboring faces have another bundle crossing at least one boundary bundle, those neighbors are actually type 2. We will reason about establishing Property 3 of Lemma 7.4 using patch spreading.

In start and end patterns, there is only one boundary outline bundle to be crossed by the bundles of patch faces and we can either choose the color for that outline (uncolored lower layer of a start pattern) or have it fixed and given (colored upper layer of an end pattern). In most of our cases however, we first ensure that Property 2 of Lemma 7.4 holds; thus, even when crossing different outline bundles, we still get the same coloring information from all of them.

With patch spreading established, we can now proof the first pattern-lemma.

[8] Again, this is not what the algorithm actually does, but a way to argue about the output.

Lemma 7.5 (Start- and End Pattern). *Let $L_{i+\frac{1}{2}}$ and $L_{i-\frac{1}{2}}$ be the upper and lower layer in L_i respectively.*

a) If $L_{i+\frac{1}{2}} = \emptyset$, an orientation of all faces in L_i can be obtained by running BundleColoring *starting on any roof face.*

b) If $L_{i-\frac{1}{2}} = \emptyset$ and $L_{i+\frac{1}{2}}$ is oriented, the ceiling faces in L_i can be oriented to be consistent with the coloring of $L_{i+\frac{1}{2}}$.

Proof. For case a), picking any roof face f coplanar to π_i, we get two perpendicular bundles $b^{(1)}, b^{(2)}$. Having a realization, we know that both cross the lower outline bundle $b_{i-\frac{1}{2}}$, hence we have a patch containing f and only bounded by $b_{i-\frac{1}{2}}$ (as L_i is connected and $L_{i+\frac{1}{2}}$ is empty). Running BundleColoring on f, we pick to colors for $b^{(1)}$ and $b^{(2)}$, making f type 2 and get that $b_{i-\frac{1}{2}}$ must be of the third color. This allows patch spreading, coloring all bundles in L_i, making all patch faces type 2. With all other bundles known, all faces of $b_{i-\frac{1}{2}}$ must also be type 2.

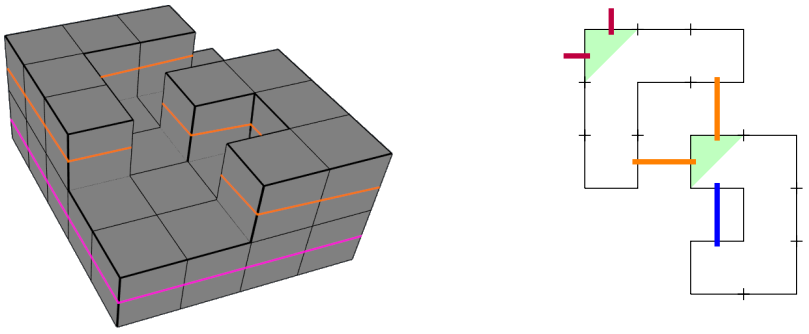
For case b), consider the outline bundle $b_{i+\frac{1}{2}}$ of $L_{i+\frac{1}{2}}$. At any convex corner of this outline, there is a ceiling face f; the two bundles of f extend onto upper wall faces. By the invariant, those bundles are colored, making the corner face type 2. With $b_{i+\frac{1}{2}}$ also colored, we argue using patch spreading as above. □

In each realizable instance, there is at least one start pattern – namely at the face defining π_t – initializing the propagation process of Lemma 7.4 and one end pattern – the face(s) coplanar to π_0.

Geometric intersection and north-western corners. The next rather simple pattern is the straight pattern. Here we first encounter an upper layer $L_{i+\frac{1}{2}}$ providing a prescribed coloring that the rest of the subgraph G_i of L_i has to be synchronized to.

Since we have the realization P at hand, we can look at the "curves" of the upper and lower cross sections of C_i on their xy-planes $\pi_{i+\frac{1}{2}}$ and $\pi_{i-\frac{1}{2}}$. Projecting the outlines of the intersected cubes down onto a common xy-plane, we obtain a two-dimensional polygonal region for each of their layers. Looking at the shared interiors of upper and lower outlines, we obtain a (nonempty) set of overlap regions. We call this set of regions the *geometric intersection* of layer L_i. As the realization is aligned to a coordinate system, so are these outlines. This allows us to characterize the regions as follows:

Each unit-length segment of an overlap region corresponds to two faces f_1, f_2 – one from the upper and one from the lower layer – coinciding with the outline of the more restrictive face. At those faces, we have three bundles – the two outline bundles (tracing the original polygonal regions in the planes) and some bundle b' perpendicular to both. In the following, we will oftentimes identify the curve with either of the two outline bundles, depending on the layer we currently consider. By construction b' extends over both faces creating f in the intersection, it is therefore present in the bundle graphs of



(a) (b)

Figure 7.5: Geometric intersection and bundle directions: (a) A layer creating a merge pattern (two upper layers, one lower layer); the intersections of lower and upper layers in with their respective xy-planes in purple and orange respectively. (b) The geometric intersection of the layer from (a) yields two curves (segments indicated by ticks). The blue bundle is fake, it is northern and southern for the same layer; The two north-western corners are indicated by the green region: The corner at the orange bundles is blocked, that at the purple bundles is unblocked.

both layers. Using the alignment of P, we can classify b' as northern, eastern, southern, or western, following the orientation of f on P. By this definition, a bundle can be northern and southern (or eastern an western) for the same curve at the same time[9]; in that case, we call the bundle *fake* northern and southern (or fake eastern and western). In the following, we will focus on non-fake northern and western bundles. The example found in Figure 7.5 shows these notions as well as the notation below.

On these curves, we now look for *north-western corners*. A *corner* is a convex 90° turn in the curve. It is *north-western*, if one of its segments is defined by a face with a northern bundle, the other segment is defined by a face with a western bundle and neither of the two bundles is fake. – the green areas in Figure 7.5 (b) depict north-western corners. We say that a northern (or western) bundle is *blocked* when there is a different curve for which it is southern (or eastern) – consider the orange bundle from Figure 7.5 (b). In that case, the blocked bundle expands onto some wall face of another curve. Similarly, a corner is *blocked* if at least one of it's bundles is blocked. It is easy to see that every curve of a geometric intersection always has at least one north-western corner.

[9] For example, consider a bundle going between the ends of a "C"-shape, like the blue bundle from Figure 7.5 (b).

Lemma 7.6 (Straight Pattern). *Let $L_i \in C_i$ be a straight pattern with upper layer $L_{i+\frac{1}{2}} \in C_{i+\frac{1}{2}}$ and lower layer $L_{i-\frac{1}{2}} \in C_{i-\frac{1}{2}}$. Given an orientation of $L_{i+\frac{1}{2}}$, the remaining parts of L_i can be synchronized to it.*

Proof. We first establish how the color of the outline bundle $b_{i-\frac{1}{2}}$ of $L_{i-\frac{1}{2}}$ can be derived. This allows us to argue using patch spreading on the remaining roof and ceiling faces.

Let $b_{i+\frac{1}{2}}$ and $b_{i-\frac{1}{2}}$ be the outline bundles of $L_{i+\frac{1}{2}}$ and $L_{i-\frac{1}{2}}$ respectively. Consider the geometric intersection of the two outlines – it is a single curve p. As there are no curves to block bundles, p has a north-western corner with two defining bundles. By Lemma 7.4 (2), those bundles are colored. They both cross $b_{i+\frac{1}{2}}$ as well as $b_{i-\frac{1}{2}}$, synchronizing them. This makes all faces of $L_{i-\frac{1}{2}}$ of type 1.

If there are patches of roof or ceiling faces in L_i, there must also be at least one face f neighboring a face of $b_{i+\frac{1}{2}}$, sharing an edge with a wall face of the upper layer and hence also sharing a bundle $b^{(1)}$. Let the other bundle of f be $b^{(2)}$ – it crosses colored bundle $b^{(1)}$ (at f) and at least one of the outline bundles making f a type 2 face for that patch, enabling patch spreading. Repeat this until all patches are oriented. □

Peeling sequence. In the following lemma for the merge pattern, we will encounter a the geometric intersection containing multiple curves.

To synchronize all outlines, we define the *peeling sequence* of the curves of a geometric intersection using the following instruction: As long as there are unprocessed curves, take a curve with an unblocked north-western corner, add it to the sequence and mark it as processed. Update the other corners by marking the northern and western bundles blocked by processed curves as not blocked (possibly marking some north-western corners as unblocked)[10].

Lemma 7.7 (Merge Pattern). *Let $L_i \in C_i$ be a merge pattern with $L_{i+\frac{1}{2},1}, \ldots, L_{i+\frac{1}{2},k} \in C_{i+\frac{1}{2}}$ and $L_{i-\frac{1}{2}} \in C_{i-\frac{1}{2}}$. Given locally consistent orientations for all layers in $C_{i+\frac{1}{2}}$, the remaining parts of L_i can be oriented consistently.*

Proof. We pick one of the upper layers as a reference and first synchronize only the other outline bundles to it. We then orienting the roof and ceiling faces between them by patch spreading. With these orientations in place, we then synchronize the color classes of the remaining layers into the reference layer.

Consider the geometric intersection of all upper layers with lower layer $L_{i-\frac{1}{2}}$; for every upper layer, we get some curve corresponding to it. Let $L_{i+\frac{1}{2},1}$ be the layer of the first element of the peeling sequence for that geometric intersection. This curve has a north-western corner with two unblocked bundles. i.e., they both cross $b_{i-\frac{1}{2}}$ and $b_{i+\frac{1}{2},1}$ only extending over roof and/or ceiling faces in between. Having bundles of two different color classes identified in both layers, we synchronize $L_{i-\frac{1}{2}}$ and $L_{i+\frac{1}{2},1}$.

[10] For the purpose of defining this sequence, use the following intuition: Imagine removing the picked curve from the geometric intersection, "peeling it away" to make other curves accessible. We will later introduce Lemma 7.8 to argue how this peeling is obtained during IteratedBundleColoring.

Following the peeling sequence, we synchronize all other outline bundles as follows: Consider the north-western corner c of the current element in the sequence. If both bundles at c cross $b_{i-\frac{1}{2}}$ are not blocked, both outline bundles get synchronized to the lower layer's outline like $b_{i+\frac{1}{2},1}$ was. If one of the bundles is blocked, it must be blocked by some curve to the north (or west respectively), crossing that upper layer's outline bundle. As that curve needs to be earlier in the sequence (by being more north and-or west), it's outline bundle is already synchronized, hence propagating the same coloring information as if crossing $b_{i-\frac{1}{2}}$. This way, IteratedBundleColoring can synchronize all outline bundles of L_i.

With all upper and lower layers synchronized, the missing patches of roof and ceiling faces can be synchronized using patch spreading like above. □

Hooking curves together. As opposed to the merge pattern above, the multiple lower layers of the split pattern are not covered by the invariant of Lemma 7.4 (1) – that is, identifying two perpendicular bundles crossing a curve does not imply synchronizing the layers they are obtained from, as the bundles of those layers are not grouped into color classes yet.

To overcome these problems, we use a similar strategy to that of Lemma 7.7: We look for the north-western corners (with bundles $b^{(h)}$ and $b^{(v)}$) in sequence, deriving the color for each corresponding outline bundle from the upper layer's outline or the curves blocking it. In the following discussion we look for a face f "near" the corner; that is, the two faces creating corner c and face f share exactly one vertex in G (they are diagonally opposite on P). To overcome the blockage, we argue having an execution of BundleColoring "nearby" and fixing the colors of two additional bundles $b^{(1)}$ and $b^{(2)}$, allowing us to explore the crossing patterns of those six bundles (including the two outlines). A schematized geometric intersection (with faces, corner, and bundles) is shown in Figure 7.6 (a). The subgraph characterizing the bundles for this configuration is shown in Figure 7.6 (b).

We demonstrate how the result of a single execution of BundleColoring on f *hooks* the outline bundles of the curves to each other. Using bundles starting in f, we can derive that the outline bundles $b_{i,1}$ and $b_{i,2}$ have to be in the same color class, even if neither outline has been synchronized to an upper outline bundle yet.

If there is no face f diagonally opposing c – because P might not have a cube in the upper layer –, we can technically also not argue using Lemma 7.8. Luckily, this missing cube certifies that there are two bundles colinear to $b^{(1)}$ and $b^{(2)}$ down from the upper layer. As those bundles are colored differently by the invariant, we can use these colors instead of those obtained by the "missing" execution. Due to the merging bundles of the same color in $\mathfrak{B}_G$, we can still assume to have the local structure in the bundle graph shown in Figure 7.6 (d).

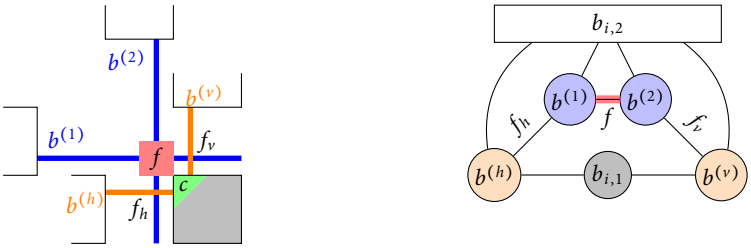


Figure 7.6: Hooking a blocked curve to the already processed parts of $\mathfrak{B}_G$, see Lemma 7.8. (a) Corner c (in green) has two (orange) bundles $b^{(h)}$ and $b^{(v)}$ that are both blocked by curves to the west and north respectively. Face f (in red) is diagonally opposite of c, the two (blue) bundles $b^{(1)}, b^{(2)}$ of f extend to the west and north respectively, eventually crossing other outline bundles (either other curves or the upper layer's outline). The pairs of bundles $b^{(h)}, b^{(2)}$ and $b^{(v)}, b^{(1)}$ cross at f_h and f_v respectively. (b) A subgraph of the bundle graph $\mathfrak{B}_G$ for the layers of (a) (with matching colors).

Lemma 7.8 (Hooking layers)**.** *Let C be the set of curves of the geometric intersection for some layer L_i. Suppose C is being processed using a peeling sequence and let c be the north-western corner of the currently processed curve. When corner c is blocked, the curve of c can be hooked to curves processed earlier in the sequence – either by using a suitable execution of* BundleColoring *or by considering the structure of P creating c.*

Proof. Let $b_{i,1}$ be the outline bundle of the currently processed curve, let c be the north-western corner of this curve and let $b^{(h)}$ and $b^{(v)}$ be the western (horizontal) and northern (vertical) bundles of c. Suppose that both $b^{(h)}$ and $b^{(v)}$ are blocked by some other curves. This layout of curves is depicted in Figure 7.6 (a); the structure in the bundle graph that enables the hooking is shown in Figure 7.6 (b), with $b_{i,2}$ being the representative of all outline bundles previously synchronized.

Let f_h be the first face that $b^{(h)}$ extends onto after the wall face of c. As $b^{(h)}$ is blocked by some curve, there must be at least one face in between them. Symmetrically define f_v as the first face of $b^{(v)}$. Consider the two other bundles of f_v and f_h, call them $b^{(1)}$ and $b^{(2)}$ respectively. By construction, these bundles cross at some face f. Face f is diagonally opposite of c on P, sharing one vertex with the wall faces composing c.

To derive that $b_{i,1}$ and $b_{i,2}$ have to be in the same color class, we need to show two things: First, $b^{(h)}$ and $b^{(v)}$ both cross $b_{i,1}$ and some outline bundle synchronized to $b_{i,2}$; and second, that $b^{(h)}$ and $b^{(v)}$ need to be in different color classes.

The first part is easy: By construction both bundles cross $b_{i,1}$. As $b^{(h)}$ extends to the west, any curve blocking it must have a north-western corner further to the west. Thus,

this curve was processed before the current curve, making it be synchronized to the outline bundle. If $b^{(h)}$ is not blocked, it trivially crosses the outline bundle that the currently processed curve is supposed to be synchronized to. This argument holds symmetrically for bundle $b^{(v)}$.

For the second part, consider the execution of BundleColoring with starting face f. The two bundles $b^{(1)}$ and $b^{(2)}$ of f cross each other, and by extending the the west/north, both also cross representatives of $b_{i,2}$. As $b^{(v)}$ crosses $b^{(1)}$ and both cross some bundle synchronized to $b_{i,2}$, both bundles also need to be in different color classes. This forces $b^{(1)}$ and $b^{(h)}$ to have the same color (symmetrically for $b^{(2)}$ and $b^{(v)}$). We now have that $b^{(h)}$ and $b^{(v)}$ are in different color classes, completing the claim.

Notice the following: If any of the three faces f, f_h, f_v does not exist, we get that (at least) one of the pairs of crossing bundles directly cross the outline we try synchronizing to, making both bundles be in different color classes. □

Since we cannot know the correct peeling sequence while running IteratedBundleColoring – the algorithm does not know the realization P –, we rely on exhaustive application. For the sake of analysis, we can assume that such a sequence exists for any valid positive instance. After hooking two curves, both outlines propagate the same coloring information when crossed, so we can effectively treat the layers creating them as as having a single outline bundle, hence merging them.

Lemma 7.9 (Split Pattern). *Let $L_i \in C_i$ be a split pattern with upper layer $L_{i+\frac{1}{2}} \in C_{1+\frac{1}{2}}$ and lower layers $L_{i-\frac{1}{2},1}, \ldots, L_{i-\frac{1}{2},m} \in C_{1-\frac{1}{2}}$. Given an orientation of $L_{i+\frac{1}{2}}$, the remaining parts of L_i can be synchronizing to it.*

Proof. Let $b_{i+\frac{1}{2}}$ be the outline bundle of the upper layer. As before, we first establish that all lower outline bundles can be synchronized to $b_{i+\frac{1}{2}}$.

Again, consider the geometric intersection of L_i and let $L_{i-\frac{1}{2},1}$ be the lower layer contributing to the first curve in a peeling sequence. As the first element cannot be blocked, there are two bundles present in $L_{i-\frac{1}{2},1}$ that are uninterrupted. Hence, these bundles allow us to synchronize $b_{i-\frac{1}{2},1}$ and $b_{i+\frac{1}{2}}$.

We now consider the remaining elements of the peeling sequence in order, synchronizing each outline bundle to $b_{i+\frac{1}{2}}$. Let c be the north-western corner of the currently processed curve. If c is not blocked, the northern and western bundle each cross both outlines, synchronizing them. If c is blocked, we argue using Lemma 7.8: We assume that there is a face f on which BundleColoring was executed, such that both outlines are hooked. The hooking certifies that the outline of the curve of c is synchronized to the outlines blocking it.

This shows that all lower outlines are synchronized to the upper outline bundle $b_{i+\frac{1}{2}}$, making all wall faces of L_i at least type 1. Next, consider the patches of roof and ceiling faces. Since L_i is connected, all patches must have a face incident to $b_{i+\frac{1}{2}}$ with a bundle coming down and making that face type 1. With all outline bundles synchronized,

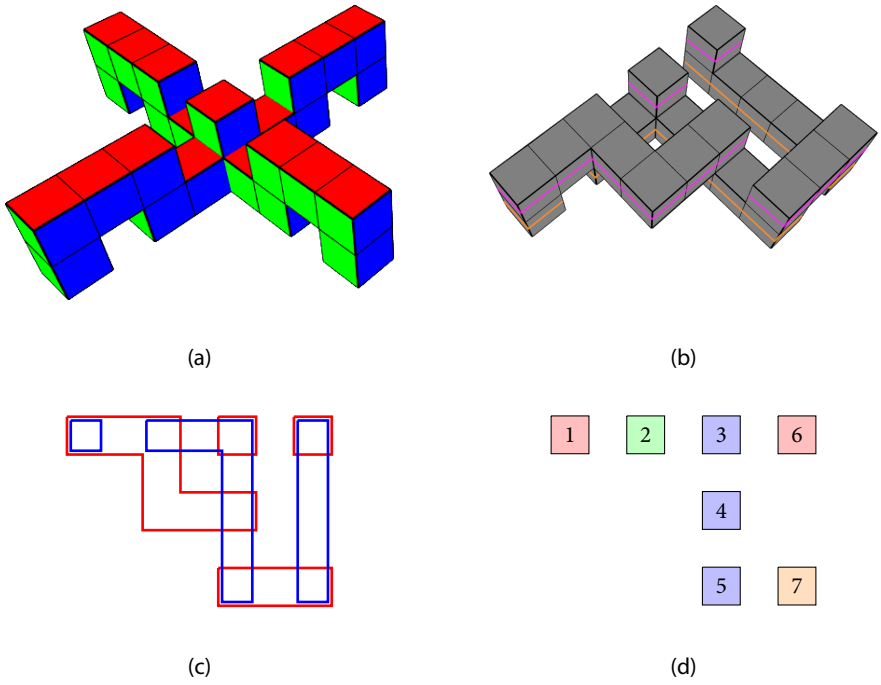


Figure 7.7: The mixed pattern: (a) The central layer $L_{i+\frac{1}{2},c}$ of this pattern has no bundle hitting the outline of the layer it stands on. (b) A layer of a polyhedron forming a mixed pattern, three upper outline bundles in purple and four lower outline bundles in orange. (c) The set of upper (in red) and lower outlines (in blue) from (b); (d) the geometric intersection of the two sets from (c) labeled from 1 to 7 by sequence of consideration, Start cases in red (1 and 6), All-Lower case in green (2), All-Upper cases in blue (3, 4 and 5), and Set-Merge case in orange (7).

that faces other bundle also gets colored uniquely, ultimately making the face type 2. Applying patch spreading to each patch colors all bundles of roof and ceiling faces in L_i, concluding the proof. □

To see the conceptual difference between the mixed pattern and regular split or merge patterns, consider the single central cube in the upper layer of in Figure 7.7 (a); call it $L_{i-\frac{1}{2},c}$. While the geometric intersection of the outlines is non-empty, neither of the two bundles going over $L_{i-\frac{1}{2},c}$ crosses the outline of the layer below it – both extend over ceiling faces and onto other upper layers on each side. To apply either Lemma 7.7 or Lemma 7.9, we would require all outline bundles to be in the same color class, but the alternating structures created by upper and lower layers prevent us from using either lemma.

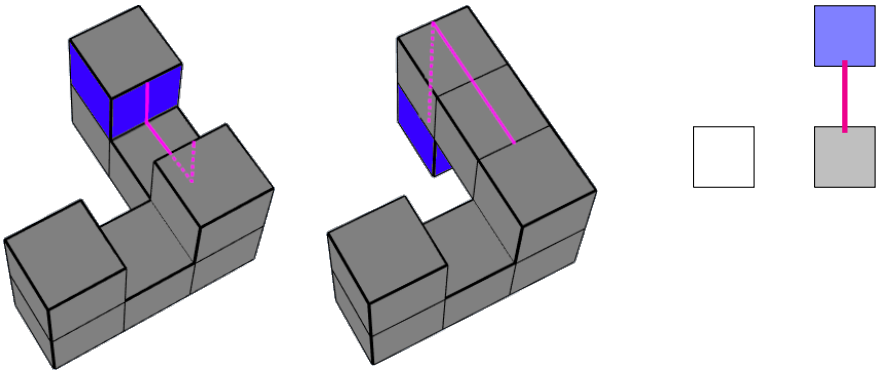


Figure 7.8: Different blocking patterns creating the same geometric intersection. The geometric intersection shown in (c) can either be obtained from the polyhedron shown in (a) or (b). The purple bundle is blocked in both configurations: in (a) we have an upper block and in (b) we have a lower block.

Lemma 7.10. *Let L_i be a mixed pattern with upper layers $L_{i+\frac{1}{2},1}, \ldots, L_{i+\frac{1}{2},k} \in C_{i+\frac{1}{2}}$ and lower layers $L_{i-\frac{1}{2},1}, \ldots, L_{i-\frac{1}{2},\ell} \in C_{i-\frac{1}{2}}$. Given individual orientations for all upper layers in $C_{i+\frac{1}{2}}$, the all of L_i can be synchronized.*

Proof. In the previous patterns we could always assume that all layers of one side (upper or lower) interact with the same layer on the other side. For the mixed pattern, this is no longer the case – When synchronizing the outline bundles of two upper-lower-layer pairs, we might end up with two *subsets* of merged bundles. Since layer L_i is connected in P, so is the corresponding subgraph of the bundle graph and eventually, all outlines will be merged (given the right executions of BundleColoring).

Recall that a curve of a geometric intersection is created by intersecting the outlines of two layers – an upper layer L_u overlapping a lower layer L_ℓ with outline bundles b_u and b_ℓ. A northern (or western) bundle is blocked when it starts on a face of b_u (or symmetrically b_ℓ) and extends upwards (to the left), but instead of crossing b_ℓ it crosses the outline bundle of some other curve. If a bundle starting on a face of b_u (or b_ℓ) is blocked, it must cross the outline bundle of some other upper (lower) layer; we distinguish between these options, calling them *upper* and *lower* blocks respectively. The different blocks are illustrated in Figure 7.8 (a) and (b).

We now classify the curves by the bundles on their north-western corner c. If both bundles of c are unblocked, the two layers start a new subset. In all other cases, at least one of the bundles at c is blocked by some curve. Since this curve is further north (or west) in the intersection, it was processed before – the layers creating the curve already

belong to some subset. With this in mind, we classify the remaining curves by the curves blocking the bundles of their corners, depending on whether all blocked bundles are blocked by curves created from layers of same subset[11] or from different subsets.

This gives the following four cases:

(1) *Start:* No blocks – creating a new subset – similar to the straight pattern (Lemma 7.6).

(2) *All-Upper:* All blocks are upper blocks, similar to the merge pattern (Lemma 7.7)

(3) *All-Lower:* All blocks are lower blocks, similar to the split pattern (Lemma 7.9)

(4) *Set-Merge:* One block is an upper block, the other block is a lower block.

An illustration of these cases can be found in Figure 7.7 (b) to (d). In the following, we consider each case, arguing how to synchronize the outline bundles. This will later allow us to use patch spreading on the roof and ceiling faces on L_i.

For Case (1), we trivially synchronize the two outline bundles – both bundles at corner c are unblocked.

The All-Upper Case (2) can only occur when synchronizing the outline of a previously unknown upper layer into a subset. The new layer and all layers of blocking curves share the same lower layer L_ℓ (otherwise, it would not have been an upper blocking); since the blocking curves are more northern and/or western, the have been synchronized to b_ℓ before. As the bundles of L_u are grouped into distinct color classes (by the invariant of Lemma 7.4 (1)), the outline bundle b_u can be synchronized.

Symmetrically, the All-Lower Case (3) can only occur when synchronizing an unknown lower layer into a subset. All blocking layers share the same upper layer L_u and the outlines of the layers creating the blocking curves are already synchronized to b_u. This allows us to hook the new layer's outline b_ℓ to the outlines of the blocking curves (using Lemma 7.8, synchronizing them.

For the Set-Merge Case (4), we are given a curve with corner c an upper and a lower block. Assume that the upper block is to the west and created by layer L_1 (with outline bundle b_1) overlapping L_ℓ and that the lower block is to the north and created by layer L_2 (with outline bundle b_2) overlapping L_u. This configuration is illustrated in Figure 7.9 (a). Let c_n and c_w be the northern and western curve respectively. Assume that L_u has the three color classes A, B, and C (from the invariant). From processing curve c_n we know that b_2 and b_u have in the same color – say this color is A; symmetrically we know from c_w that b_1 and b_ℓ have the same color – say D, as it is yet to be synchronized. To establish that b_u and b_ℓ have the same color – merging A and D, synchronizing all four outlines –, we look at the two bundles at c, namely $b^{(w)}$ and $b^{(n)}$. From the pattern we have that one of the two bundles is synchronized to the color classes of L_u (by extending over a wall face of L_u, participating in the upper block) whereas the other is not; w.l.o.g. assume that this bundle is $b^{(n)}$ and that it's color is B. Since $b^{(n)}$ crosses b_1, we have that $D \neq B$. To argue that the outline bundles get synchronized – that is, $D = A$ – we now need to

[11] This naturally includes the case when only one bundle is blocked.

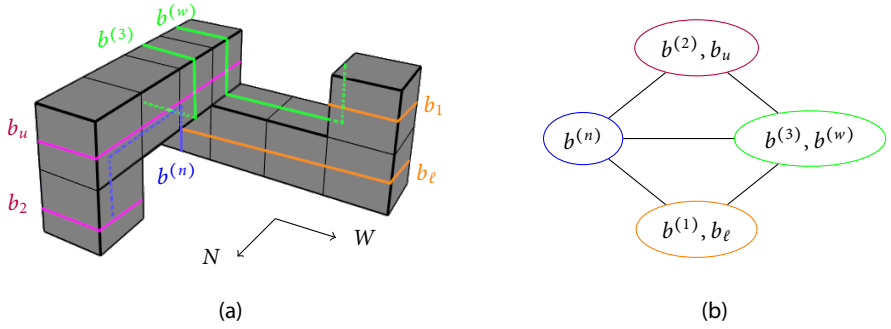


Figure 7.9: Illustration of the Set-Merge Case (4). (a) A simple polyhedron showing a Set-Merge Case (rotated such that the viewpoint is the north-western side, compass rose included). The colored bundles indicate synchronization: purple and green for the outlines of the two subsets respectively, green for the western bundles of the upper layer, blue for the (supposed) northern bundle of the lower layer. (b) The part of the bundle graph showing how the four colors interact: Green-purple and green-orange edge from $b^{(w)}$ crossing b_u and b_1 on a wall face respectively; blue-purple and blue-orange edge from $b^{(n)}$ crossing b_ℓ and b_2 on a wall face respectively; blue-green edge from $b^{(3)}$ crossing $b^{(n)}$ on the ceiling face neighboring the corner.

show that the color of $b^{(w)}$ needs to be C, as $b^{(w)}$ crosses b_ℓ. In Figure 7.9 (b), we give a matching bundle graph with A being purple, B being green, C being blue, and D being orange.

Consider the first ceiling face f_1 that $b^{(w)}$ extends over after the wall face at which it crossed b_ℓ. Let bundle $b^{(3)}$ be the other bundle of f. Symmetrically, let f_2 be the first roof face of bundle $b^{(n)}$. Figure 7.9 (a) suggests that $b^{(3)}$ must "wrap up" – that is, $b^{(3)}$ extends onto a wall face of L_u, making it part of L_u and thus colored by the invariant. To show that this wrapping up is possible in a valid realization, consider the unit square x diagonally opposing corner c in the geometric intersection of cross section π_i[12]. Considering how P is built from unit cubes, first observe that x is empty space in the geometric intersection – that is, x cannot be another corner, created by two cubes (one intersected in $\pi_{i+\frac{1}{2}}$ and one cube intersected in $\pi_{i-\frac{1}{2}}$). Wrapping up could still be prevented from a single cube in either $\pi_{i-\frac{1}{2}}$ or $\pi_{i+\frac{1}{2}}$ – possibly making $b^{(3)}$ extend over another wall or ceiling face respectively. Neither case is possible: The upper cube would share the same edge with face f_2 and a wall face of L_ℓ on P but not in G, making P degenerate; symmetrically, the lower cube would share the same edge with f_1 and a wall face of L_u. Therefore, x must indeed mark empty space in a valid realization, allowing $b^{(3)}$ to wrap around. The construction of $b^{(3)}$ implies that it has the same color as $b^{(w)}$, therefore we get that the color of $b^{(n)}$ is neither A nor B and thus C. We now have that the bundles of color class

[12] Following the idea used for the hooking face of Lemma 7.8, but with a very different result – namely, that there cannot be such a face in the Set-Merge Case of the mixed patterns.

D are crossed by bundles of colors B and C, enforcing the synchronization of classes A and D.

This concludes the enumeration of all possible patterns involving upper and lower blocks, thus arguing that IteratedBundleColoring eventually synchronizes all outline bundles in the mixed pattern. With all outlines synchronized, patch spreading synchronizes the bundles on roof and ceiling faces of π_i. Since each patch must share a bundle with some upper layer, all layers will get synchronized. □

The following – and final – lemma reconsiders all patterns above in the presence of flat holes. Recall that a flat hole is a pair of outline bundles that both appear fully within the same half-integer cross section – an inside and an outside one – and with one containing the other. Notice that flat holes (and the bundles creating them) are not immediately visible in either the supposed net G or the bundle graph $\mathfrak{B}_G$. Nevertheless, given a realization P that contains flat holes, we can argue how IteratedBundleColoring discovers the colors creating the orientations of the faces on P.

Any such outside-inside outline bundle pair corresponds to the existence of one hole, but one outer outline might contain several independent inner outlines and thus be part of multiple pairs. In addition to transmitting coloring information downwards from the (upper) layers of $C_{i+\frac{1}{2}}$ to the (lower) layers of $C_{i-\frac{1}{2}}$, we now also have to consider transmitting "sideways" from one hole outline to the other.

Since we aim to process P using the patterns presented above, we have to distinguish between holes in starting patterns and holes in lower layers. We do not need to worry about holes in the upper layers of other patterns, because by the invariant, all bundles in these upper layers were lower layers of the previous step and subsequently have been merged and virtually disappear from the contemporary bundle graph $\mathfrak{B}^*_G$.

Lemma 7.11 (Hole-Layer Lemma)**.** *Let L_i be one of the patterns above and let $L_{i-\frac{1}{2}}$ be a lower layer of that pattern that contains flat holes.*

a) If L_i is a start pattern, there is a set of executions of BundleColoring that puts all outline bundles in $L_{i-\frac{1}{2}}$ in the same class.

b) If L_i is not a start pattern, all outline bundles in $L_{i-\frac{1}{2},k}$ either get hooked to the outside outline or synchronized to some upper layers outline.

Proof. Our only concern here is to argue about how the outline bundles get synchronized using specific executions of BundleColoring. Once we can safely assume that all outline bundles are synchronized, patch spreading carries over to patches with holes. In fact, the net G (and also the bundle graph $\mathfrak{B}_G$) of a hole in a lower layer is distinguishable from an upper layer "standing" at the same spot[13].

Regarding a): Among the outlines present in $L_{i-\frac{1}{2}}$, consider a (convex) north-western corner of the outer outline. Executing BundleColoring at the roof face of this corner

[13] The gadgets used in the $\mathcal{NP}$-hardness reduction by Biedl and Genç [BG08] exploit this.

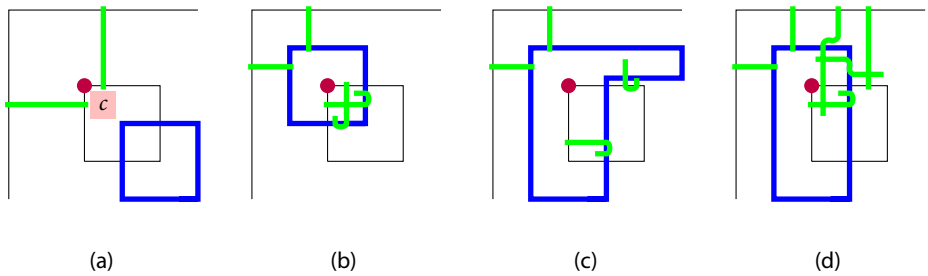


Figure 7.10: Patterns for synchronizing the inner outline of a hole to the outside outline using the upper layer (outline in blue), depending on the northern and western bundles provided by the upper layer. (a) Corner c (purple dot) is not covered by the upper layer – the bundles at c synchronize b_i to curves more northern/western. (b and c) c is covered by the upper layer, but the upper layer has two bundles of different colors wrapping down and crossing b_i, synchronizing both outlines. Notice that in (c), the northern bundle does not wrap around but follows the wall faces. (d) the corner is covered, but there is no northern bundle from the upper layer going into the hole, so synchronizing a (northern) bundle from a roof face is required.

initially fixes the three color classes that all inner outlines will to be synchronized to. We now imagine that the inner outlines are processed using a peeling sequence – despite not having a geometric intersection here, the (empty) corners of inner outlines will have northern and western bundles that are either blocked by the outlines of other holes or reach the outside outline, allowing us to naturally define a peeling sequence. This allows us to apply Lemma 7.8, hook all holes together and eventually synchronize all of them to the outside outline using executions of BundleColoring on various (diagonally opposing) roof faces.

Regarding b): For any other pattern, we can have flat holes in some lower layer $L_{i-\frac{1}{2}}$. In that case, we synchronize the inner outlines to either the lower outline $b_{i-\frac{1}{2}}$ (containing it) or the outline bundle of some (upper) layer from $L_{i+\frac{1}{2}}$. We do this by arguing that we can either add it to a peeling sequence for the upper layers or that we can synchronize the outline bundle b_i to that of the upper layer covering it. In either case, we require that the outline bundles of the upper layers all get synchronized to the lower layers outline bundle by following the reasoning provided in the lemmata for the individual patterns[14]. For a flat hole with inner bundle b_i, there are four cases depending on the "shape" of the ceiling over c[15] provided by upper layers. The four possible shapes are depicted in Figure 7.10: none of the bundles at c is part of the upper layer (a), there are two bundles of different color from the upper layer both crossing b_i, with two different options shown

[14] Here we have the additional complication that holes can actually block northern and western bundles. To do so, however, such a hole must be more to the north/west and thus have been synchronized before.
[15] Remember that c is the corner of a hole, and thus, that c is empty space.

in subfigures (b) and (c), and there are only bundles of one color from the upper layer crossing b_i (d). We now discuss the cases individually, arguing how IteratedBundleColoring synchronizes the inner outline of the hole to the upper layer's outline.

In the uncovered case (Figure 7.10 (a)), there is no upper layer covering c. In this case, we can handle synchronization of b_i as if it were an upper layer – exploiting the fact that inner lower layers and upper layers are indistinguishable in $\mathfrak{B}_G$. Treating it like an upper layer, there is some peeling sequence containing it. By the time that b_i is processed, we know that the two bundles at c – northern bundle $b^{(n)}$ and western bundle $b^{(w)}$ – each either crosses the lower layer's outline or is blocked by some other upper curve. By choice of the sequence, we know that all those bundles are already synchronized. Therefore, b_i can be synchronized using $b^{(n)}$ and $b^{(w)}$.

In the two-sided cases of Figure 7.10 (b) and (c), there is a ceiling face over c and the outline of the layer covering c has a corner over the hole defined by b_i. Let that corner be c'. In either case – c' being convex or reflex – there are two bundles of different colors from an upper layer wrapping around onto the ceiling over c. These bundles eventually cross b_i, synchronizing it to the upper layers outline.

In the one-sided case (Figure 7.10 (d)), corner c is covered by a ceiling face but we only have bundles from one direction (w.l.o.g. say west) from the upper layer but not from the other direction (say north). Since the upper layer covers c, it must have a north-western corner further north and west – hence, that layer's outline bundle is already synchronized to the outside outline bundle. To find a northern bundle crossing b_i, consider the face neighboring both outline bundles – b_i and the upper layer's outline. This face must exist, as the hole is not completely covered by the upper layer. At that face, there are two bundles: Bundle $b^{(1)}$ extends onto a neighboring wall face of the upper layer, marking it as western. The other bundle $b^{(2)}$ extends towards the north, eventually crossing some other outline (that has already been processed by choice of sequence). This marks $b^{(2)}$ as northern; as it wraps around and down onto a wall face in b_i, this synchronizes the outline bundles. □

With all of the pattern lemmata in place, we can now proof Lemma 7.4 and subsequently Theorem 7.3.

Proof of Lemma 7.4. Assuming that a realization exists, there is at least one triple of bundles forming the first set of merged bundles for $\mathfrak{B}^*_G$ – namely that of the face defining π_t. Therefore we are guaranteed that there is at least one starting pattern, that can be colored and subsequently have it's bundles merged according to Lemma 7.5 (a), establishing the condition of 1, referred to as the invariant.

Discovering and coloring more outline bundles for condition 2 – while traversing the realization downwards – we can argue that all encountered bundles can be merged using only the coloring information locally available in the triples of layer subgraphs. This is done by either Lemma 7.7, Lemma 7.9 or Lemma 7.10. In each lemma, we first argue how IteratedBundleColoring preserves the invariant by being able to merge the bundle-triangles of multiple upper layers (stemming from different starting patterns).

Then, all remaining outline bundles of the current layer will be synchronized to the upper outlines: Outer outline bundles get handled directly in the individual lemmata; inner outlines of flat holes are implicitly added to the synchronization process – this is argued in Lemma 7.11.

Finally, in each layer, all missing bundles of roof and ceiling faces get colored using patch spreading, arguing that condition 3 holds, concluding the proof. □

Proof of Theorem 7.3. In this section, we considered pairs of xy-parallel cross sections at unit distance, exhaustively enumerating all possible patterns that can be encountered in a valid realization. When our input graph is indeed the net of a polyhedron, the invariant established in Lemma 7.4 states that IteratedBundleColoring is able to orient all faces on that polyhedron by coloring the bundles in $\mathfrak{B}$.

For each of the patterns, we demonstrate that there is a finite number of single executions of BundleColoring that together merge the color classes into a triangle. Each merge is unambiguous, resulting in IteratedBundleColoring producing a stable unique coloring if one exists. Exhaustive execution as done by IteratedBundleColoring ends up performing these executions. Hence, if IteratedBundleColoring stops at some point where $\mathfrak{B}^*_G$ is not a triangle, some part of the graph could not be realized using any of the patterns, and we have an non-realizable instance. □

7.5 Conclusion

In this chapter, we introduced the IteratedBundleColoring algorithm that uses the original BundleColoring algorithm by Biedl and Genç [BG09] as a subroutine. We showed that exhaustive and repeated execution of the original algorithm can be used to report all edges of the graph for which the dihedral angles must be 180° in any orthogonal polyhedral surface that realizes this graph and facial angles. This implies that orthogonal polyhedral surfaces of arbitrary genus are rigid – that is, the net and the facial angles determine the dihedral angles.

Our algorithm has a total runtime in $O(m^3)$, but our focus in this chapter was put onto proofing correctness of IteratedBundleColoring. We believe that the runtime analysis can be improved by considering the sequence of BundleColoring-executions more carefully. The question of non-orthogonal polyhedral surfaces also remains open.

Chapter 8
Conclusion

The contents of this book are dedicated to problems regarding constrained graph layouts. We picked two general ideas – convex drawings and grid drawings –, looking into two specialized problems for each idea. This division gave birth to this book's two-part structure. We now briefly recall the main results of each chapter in order, highlighting open questions and interesting opportunities for future research.

Part One: Drawing Vertices on a Common Outer Face

Beyond Outerplanarity. The main contribution of Chapter 3 was looking into the combination of graph properties – outerplanarity and k-(quasi-)planarity – obtaining convex graph drawings with two different restrictions: limiting the number of crossings per edge or limiting the size of any sets of pairwise crossing edges.

We showed that outer k-planar graphs are $(\lfloor\sqrt{4k+1}\rfloor+1)$-degenerate, thus obtaining a coloring bound on these graphs. We have also shown that they have small balanced separators (of size $2k + 3$), allowing outer k-planarity to be tested in quasi-polygonal time.

By giving graphs of either class that are not member of the other, we have shown that the classes of outer k-quasi-planar graphs and planar graphs are incomparable. We have also proven that all maximal outer k-quasi-planar graphs are also maximum – for any outer k-quasi-planar graph, there is a way to add missing edges until the upper bound of $2(k-1)n + \binom{2k-1}{2}$ is reached.

Considering the coloring result for outer k-planar graphs, we would be curious to know the answer to the following question:

Open Problem 1. What is the chromatic number of outer k-quasi-planar graphs with respect to k?

We also gave a linear-time algorithm to recognize full and closed outer k-planar graphs. It directly follows from Courcelle's Theorem [Cou90] (by expressing closed outer k-planarity in Monadic Second-Order logic) and the fact that outer k-planar graphs have bounded treewidth. This leads to the following open research question.

Open Problem 2. Is the treewidth of outer k-quasi planar graphs bounded by some function of k?

Answering Open Problem 2 in the affirmative would improve Theorem 3.13, allowing recognition via Theorem 3.11 in linear time as well.

Polygonal Boundaries. In Chapter 4, we developed an algorithm to decide wheather or not an outerplanar graph can be drawn into a given simple (not necessarily convex) polygon, when the vertices are mapped to the boundary and when each of the edges is allowed to have one bend inside the polygon. The algorithm we proposed works with the dual tree of the outerplanar graph, refining the polygon with each drawn edge until all edges are drawn or one of the bend points can not be placed inside the polygon.

A possible application for our algorithm is recursively insert subgraphs into the inner faces of some predefined drawing. Being able to decide the question if a one-bend drawing for outerplanar graphs exists naturally gives rise to questions about related graph classes.

Open Problem 3. Can we decide if planar graphs or outer k-planar graphs can be drawn inside a simple polygon when only some of the vertices are mapped to the boundary?

Considering the bend points of our edges to be degree 2 subdivision vertices, we get a very restricted family of planar graphs drawable by our algorithm – finding a sufficent condition for larger subsets of planar graphs would be very interesting. Also, our algorithm works by iteratively refining the polygon – drawing an edge changes the boundary and planarity limits the options for further edges. What if we allowed some crossings on the inserted edges? Expanding the set of graph classes drawable by our algorithm – or similar strategies – would improve it's applicability as a subroutine.

Part Two: Drawing Vertices using Integer Coordinates

Moving to the Grid Optimally. Chapter 5 considered the task of transforming a given planar drawing into a topologically equivalent grid drawing with minimum vertex displacement – that is, a drawing with all vertices at integer coordinates, inducing the same embedding in the plane, and all vertices relatively close to their original position. This task lends itself to trying rounding-like procedures such as modified variants of snap rounding, that are well-known in computational geometry.

We showed that Topologically-Safe Grid Representation (as we call the problem above) is $\mathcal{NP}$-hard. Since we could not hope finding a modified version of an efficient snap rounding algorithm, we modelled Topologically-Safe Grid Representation as an integer linear programm. To evaluate the performance of our model, we implemented it in Java using the IBM CPLEX solver. We found our implementation to perform poorly on large instances – both area-wise and with respect to the number of vertices. While lazily generating the constraints for or model "on demand" gave some speedup, the model is still infeasible for any practical purpose. This raises the question about further improvements on the model.

Open Problem 4. As not all grid points will be used, can a column-generation-like strategy be used to iteratively "add" new grid points to the model?

Our analysis suggests that the area of the drawing has the largest impact on the wall-clock runtime of or implementation, because the sizes of most sets of constraints heavily

depend on the number of possible edge slopes. One way of limiting the number of possible slopes would be to settle Open Problem 4.

As a byproduct, our implementation can also be used to create minimum-area straight-line drawings of planar graphs – a problem also known to be $\mathcal{NP}$-hard [KW08]. Our experiments suggest that this can in practice only be used to verify or produce very small (counter-)examples. To the best of our knowledge, the problem of finding minimum-area straight-line drawings in reasonable time is still open.

Open Problem 5. Can we adapt the Topologically-Safe Grid Representation model to find area-minimal straight-line planar drawings faster?

Rounding to the Grid Heuristically. In Chapter 6, we have presented a practical heuristic for the Topologically-Safe Grid Representation problem, introduced in Chapter 5. Our algorithm follows the simulated annealing metaheuristic, but is subdivided into two distinct phases – one for feasibility, the other for optimization. The various features of the algorithm have been statistically validated as improving the runtime or the solution quality.

In Chapter 5, we presented a slow but exact algorithm, where here we gave a fast algorithm that is likely to produce reasonable results. The obvious open problems are trying to this rather vague statement: One could try finding an efficient deterministic heuristic or some approximation algorithm.

Open Problem 6. Is there a deterministic heuristic or an approximation algorithm for Topologically-Safe Grid Representation?

While we obtained a result on approximation hardness (see Corollary 5.4) for a special case, we failed to show $\mathcal{APX}$-hardness for the general Topologically-Safe Grid Representation problem.

We have considered Topologically-Safe Grid Representation as an abstract problem in isolation. Future work can consider the place of grid representations in a larger context. For example, we noted it may be useful to integrate polyline simplification for geographic data (recall the rather extreme examples in Figure 6.7). It would also be interesting to evaluate the influence that rounding to a grid representation has on subsequent steps in a pipeline, such as the length of shortest paths, the results of generalization, or, for example, map matching.

Open Problem 7. Investigate how our two-stage heuristic algorithm can be adapted to allow for better perception of rounded geographic data.

Recognizing Nets of Orthogonal Polyhedra. In the final chapter of this book – Chapter 7 – we built upon the BundleColoring algorithm by Biedl and Genç [BG09]. Analyzing a repeated sequence of exhaustive executions of BundleColoring (which we called IteratedBundleColoring), we were able to translate Cauchy's Rigidity Theorem to orthogonal polyhedra of arbitrary genus – settling an open question proposed by

Biedl and Genç. Our result encourages looking into other less regular objects – like nets with facial angles that are multiples of 45°.

Open Problem 8. Is there a translation of our rigidity theorem to other polyhedra, for instance allowing dihedral and/or facial angles of that are multiples of 45°?

While our algorithm finds it's output in a brute-force-like fashion, the analysis we used to show it's correctness relied on carefully traversing a hypothetical realization, layer by layer. This resulted in a rather simple (and easily implementable) algorithm with a rather trivial upper bound of $O(m^3)$ on the runtime. This bound was obtained from the total number of possible restarts in each round and the maximum number of rounds, but our analysis only uses a rather small but well-selected subset of these executions.

Open Problem 9. Is there a better bound on the total number of restarts performed by IteratedBundleColoring?

It is also worth noting that neither our algorithm nor runtime or analysis are impacted by the actual genus of the resulting polyhedron. This is rather surprising, as the challenges seem to be related to the inner walls of the holes of the polyhedron. It seems possible that the number of restarts (and thus also the total runtime) can be parametrized by the genus of the realization.

Bibliography

[ABB+16] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Andreas Gleißner, Kathrin Hanauer, Daniel Neuwirth, and Josef Reislhuber. Outer 1-Planar Graphs. *Algorithmica*, 74(4):1293–1320, 2016.

[ABB+17] Patrizio Angelini, Michael A. Bekos, Franz J. Brandenburg, Giordano Da Lozzo, Giuseppe Di Battista, Walter Didimo, Giuseppe Liotta, Fabrizio Montecchiani, and Ignaz Rutter. On the Relationship between k-Planar and k-Quasi Planar Graphs. In Hans Bodlaender and Gerhard Woeginger, editors, *Graph-Theoretic Concepts in Computer Science*, volume 10520, pages 59–74, 2017.

[ABS12] Evmorfia N. Argyriou, Michael A. Bekos, and Antonios Symvonis. The Straight-Line RAC Drawing Problem is NP-Hard. *Journal of Graph Algorithms and Applications*, 16(2):569–597, 2012.

[Ack09] Eyal Ackerman. On the Maximum Number of Edges in Topological Graphs with no Four Pairwise Crossing Edges. *Discrete and Computational Geometry*, 41(3):365–375, 2009.

[ADF+15] Patrizio Angelini, Giuseppe Di Battista, Fabrizio Frati, Vít Jelínek, Jan Kratochvíl, Maurizio Patrignani, and Ignaz Rutter. Testing Planarity of Partially Embedded Graphs. *ACM Transactions on Algorithms*, 11(4):1–42, 2015.

[AH76] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82(5):711–713, sep 1976.

[AT07] Eyal Ackerman and Gábor Tardos. On the maximum number of edges in quasi-planar graphs. *Journal of Combinatorial Theory, Series A*, 114(3):563–571, 2007.

[AZ04] Martin Aigner and Günter M. Ziegler. Cauchy's rigidity theorem. In *Proofs from THE BOOK*, pages 71–74. Springer Berlin Heidelberg, 2004.

[BBN+13] Therese C. Biedl, Thomas Bläsius, Benjamin Niedermann, Martin Nöllenburg, Roman Prutkin, and Ignaz Rutter. Using ILP/SAT to Determine Pathwidth, Visibility Representations, and other Grid-Based Graph Drawings. In Stephen K. Wismath and Alexander Wolff, editors, *Proceedings of the 21st International Symposium on Graph Drawing*, volume 8242 of *Lecture Notes in Computer Science*, pages 460–471. Springer, 2013.

[BCD+02] Therese C. Biedl, Timothy M. Chan, Erik D. Demaine, Martin L. Demaine, Paul Nijjar, Ryuhei Uehara, and Ming-wei Wang. Tighter bounds on the genus of nonorthogonal polyhedra built from rectangles. In *14th CCCG*, pages 105–108, 2002.

[BE14] Michael J. Bannister and David Eppstein. Crossing Minimization for 1-page and 2-page Drawings of Graphs with Bounded Treewidth. In Christian Duncan and Antonios Symvonis, editors, *Graph Drawing and Network Visualization*, volume 8871, pages 210–221. Springer, 2014.

[BEG+03] Franz-Josef Brandenburg, David Eppstein, Michael T. Goodrich, Stephen G. Kobourov, Giuseppe Liotta, and Petra Mutzel. Selected Open Problems in Graph Drawing. In Giuseppe Liotta, editor, *Graph Drawing*, volume 2912, pages 515–539, 2003.

[BG08] Therese C. Biedl and Burkay Genç. Cauchy's Theorem for orthogonal polyhedra of genus 0. Technical Report CS-2008-26, University of Waterloo, School of Computer Science, 2008.

[BG09] Therese C. Biedl and Burkay Genç. Cauchy's Theorem for Orthogonal Polyhedra of Genus 0. In Amos Fiat and Peter Sanders, editors, *17th ESA*, volume 5757 of *LNCS*, pages 71–82. Springer, 2009.

[BG11] Therese C. Biedl and Burkay Genç. Stoker's Theorem for Orthogonal Polyhedra. *Int. J. Comput. Geometry Appl.*, 21(4):383–391, 2011.

[BGHL18] Carla Binucci, Emilio Di Giacomo, Md. Iqbal Hossain, and Giuseppe Liotta. 1-page and 2-page drawings with bounded number of crossings per edge. *European Journal of Combinatorics*, 68(Supplement C):24–37, 2018. Combinatorial Algorithms, Dedicated to the Memory of Mirka Miller.

[BH92] Harry Buhrman and Steven Homer. Superpolynomial Circuits, Almost Sparse Oracles and the Exponential Hierarchy. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*, volume 652 of *Lecture Notes in Computer Science*, pages 116–127. Springer, 1992.

[BKN16] Jasine Babu, Areej Khoury, and Ilan Newman. Every Property of Outerplanar Graphs is Testable. In Klaus Jansen, Claire Mathieu, José D. P. Rolim, and Chris Umans, editors, *APPROX/RANDOM 2016*, volume 60 of *LIPIcs*, Dagstuhl, 2016. Schloss Dagstuhl, Leibniz-Zentrum für Informatik.

[BLS05] Therese C. Biedl, Anna Lubiw, and Julie Sun. When can a net fold to a polyhedron? *Comput. Geom.*, 31(3):207–218, 2005.

[BO79a] Bentley and Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, C-28(9):643–647, sep 1979.

[BO+79b] Jon L. Bentley, Thomas Ottmann, et al. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, 100(9):643–647, 1979.

[Bor84] O. V. Borodin. Solution of the Ringel problem on vertex-face coloring of planar graphs and coloring of 1-planar graphs. *Metody Diskret. Analiz.*, 41:12–26, 1984.

[Cab06] Sergio Cabello. Planar embeddability of the vertices of a graph using a fixed point set is NP-hard. *Journal of Graph Algorithms and Applications*, 10(2):353–363, 2006.

[CE12] Bruno Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press, 2012.

[CFG+15] Timothy M. Chan, Fabrizio Frati, Carsten Gutwenger, Anna Lubiw, Petra Mutzel, and Marcus Schaefer. Drawing Partially Embedded and Simultaneously Planar Graphs. *Journal of Graph Algorithms and Applications*, 19(2):681–706, 2015.

[CFK+15] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*, chapter Lower Bounds Based on the Exponential-Time Hypothesis, pages 467–521. Springer, 2015.

[CH67] Gary Chartrand and Frank Harary. Planar Permutation Graphs. *Annales de l'I.H.P. Probabilités et statistiques*, 3(4):433–438, 1967.

[Chi08] John W. Chinneck. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*, volume 118 of *International Series in Operations Research & Management Science*. Springer, 1. edition, 2008.

[CLR87] Fan R. K. Chung, Frank Thomson Leighton, and Arnold L. Rosenberg. Embedding Graphs in Books: A Layout Problem with Applications to VLSI Design. *SIAM Journal on Algebraic and Discrete Methods*, 8(1):33–58, 1987.

[CLRS13] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 3rd edition edition, 2013.

[CN98] Marek Chrobak and Shin-Ichi Nakano. Minimum-width grid drawings of plane graphs. *Computational Geometry*, 11(1):29–54, 1998.

[Con79] Robert Connelly. The Rigidity of Polyhedral Surfaces. *Mathematics Magazine*, 52(5):275–283, 1979.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, 1971.

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.

[CP92] Vasilis Capoyleas and János Pach. A Turán-type theorem on chords of a convex polygon. *Journal of Combinatorial Theory, Series B*, 56(1):9–15, 1992.

[dBHO07] Mark de Berg, Dan Halperin, and Mark Overmars. An intersection-sensitive algorithm for snap rounding. *Computational Geometry*, 36(3):159–165, apr 2007.

[dBK12] Mark de Berg and Amirali Khosravi. Optimal binary space partitions for segments in the plane. *International Journal of Computational Geometry & Applications*, 22(03):187–205, 2012.

[DEW17] Vida Dujmović, David Eppstein, and David R. Wood. Structure of graphs with locally restricted crossings. *SIAM Journal on Discrete Mathematics*, 31(2):805–824, 2017.

[dFPP90] Hubert de Fraysseix, János Pach, and Richard Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[DG02] Olivier Devillers and Pierre-Marie Gandoin. Rounding Voronoi diagram. *Theoretical Computer Science*, 283(1):203–221, jun 2002.

[Dil87] Michael B. Dillencourt. A non-Hamiltonian, nondegenerate Delaunay Triangulation. *Information Processing Letters*, 25(3):149–151, may 1987.

[DKM02] Andreas W. M. Dress, Jack H. Koolen, and Vincent Moulton. On Line Arrangements in the Hyperbolic Plane. *European Journal of Combinatorics*, 23(5):549–557, 2002.

[DLL18] Olivier Devillers, Sylvain Lazard, and William J. Lenhart. 3D Snap Rounding. 2018.

[DLT83] Danny Dolev, Frank Thomson Leighton, and Howard Trickey. Planar Embedding of Planar Graphs. Technical report, DTIC Document, 1983.

[DN14] Zdeněk Dvořák and Sergey Norin. Treewidth of graphs with balanced separations. ArXiv, 2014.

[DO02] Melody Donoso and Joseph O'Rourke. Nonorthogonal polyhedra built from rectangles. In *14th CCCG*, pages 101–104, 2002. `https://arxiv.org/abs/cs/0110059`.

[DSW07] Vida Dujmović, Matthew Suderman, and David R. Wood. Graph drawings with few slopes. *Computational Geometry*, 38(3):181–193, oct 2007.

[Ead84] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[EGK+04] Markus Eiglsperger, Carsten Gutwenger, Michael Kaufmann, Joachim Kupke, Michael Jünger, Sebastian Leipert, Karsten Klein, Petra Mutzel, and Martin Siebenhaller. Automatic Layout of UML Class Diagrams in Orthogonal Style. *Information Visualization*, 3(3):189–208, jul 2004.

[EM14] David Eppstein and Elena Mumford. Steinitz Theorems for Simple Orthogonal Polyhedra. *JoCG*, 5(1):179–244, 2014.

[ET89] Peter Eades and Roberto Tamassia. Algorithms For Drawing Graphs: An Annotated Survey. techreport CS-89-09, October 1989.

[EW90] Peter Eades and Nicholas C. Wormald. Fixed edge-length graph drawing is NP-hard. *Discrete Applied Mathematics*, 28(2):111–134, aug 1990.

[Fár48] István Fáry. On straight Lines representation of plane graphs. *ACTA Scientiarum Mathematicarum Szeged*, 11:229–233, 1948.

[FP07] Fabrizio Frati and Maurizio Patrignani. A Note on Minimum-Area Straight-Line Drawings of Planar Graphs. In *15th International Symposium on Graph Drawing*, volume 4875 of *Lecture Notes in Computer Science*, pages 339–344. Springer, 2007.

[FPS13] Jacob Fox, János Pach, and Andrew Suk. The Number of Edges in k-Quasi-planar Graphs. *SIAM Journal on Discrete Mathematics*, 27(1):550–561, 2013.

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

[Gas12] William I. Gasarch. Guest Column "the second P =?NP poll". *ACM SIGACT News*, 43(2):53–77, jun 2012.

[GGHT97] Michael T. Goodrich, Leonidas J. Guibas, John Hershberger, and Paul J. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proceedings of the 13th Annual Symposium on Computational Geometry*, pages 284–293. ACM, 1997.

[Gil14] Alexander Gilbers. *Visibility Domains and Complexity*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2014.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Fransisco, CA, 2nd edition, 1979.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison Wesley, 2nd edition, 1994.

[GKR94] V. Granville, M. Krivanek, and J.-P. Rasson. Simulated annealing: a proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):652–656, jun 1994.

[GKS92] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1-6):381–413, jun 1992.

[GKT14] Jesse Geneson, Tanya Khovanova, and Jonathan Tidor. Convex geometric $(k+2)$-quasiplanar representations of semi-bar k-visibility graphs. *Discrete Mathematics*, 331:83–88, 2014.

[GM98] Leonidas J. Guibas and David H. Marimont. Rounding Arrangements Dynamically. *International Journal of Computational Geometry & Applications*, 8(2):157–178, apr 1998.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, mar 1991.

[GY86] Daniel H Greene and F Frances Yao. Finite-resolution Computational Geometry. In *27th Annual Symposium on Foundations of Computer Science (SFCS 1986)*, pages 143–152. IEEE, IEEE, 1986.

[HEK+15] Seok-Hee Hong, Peter Eades, Naoki Katoh, Giuseppe Liotta, Pascal Schweitzer, and Yusuke Suzuki. A Linear-Time Algorithm for Testing Outer-1-Planarity. *Algorithmica*, 72(4):1033–1054, 2015.

[Her13] John Hershberger. Stable snap rounding. *Computational Geometry*, 46(4):403–416, 2013.

[HN08] Seok-Hee Hong and Hiroshi Nagamochi. Convex drawings of graphs with non-convex boundary constraints. *Discrete Applied Mathematics*, 156(12):2368–2380, 2008.

[HN16] Seok-Hee Hong and Hiroshi Nagamochi. Testing Full Outer-2-planarity in Linear Time. In Ernst W. Mayr, editor, *Graph-Theoretic Concepts in Computer Science*, volume 9224, pages 406–421. Springer, 2016.

[Hob99] John D. Hobby. Practical segment intersection with finite precision output. *Computational Geometry*, 13(4):199–214, 1999.

[HP02] Dan Halperin and Eli Packer. Iterated snap rounding. *Computational Geometry*, 23(2):209–225, 2002.

[HS98] D. Harel and M. Sardas. An Algorithm for Straight-Line Drawing of Planar Graphs. *Algorithmica*, 20(2):119–135, feb 1998.

[HT17] Michael Hoffmann and Csaba D. Tóth. Two-Planar Graphs Are Quasiplanar. In *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, volume 83, page 47. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[IP01] Russell Impagliazzo and Ramamohan Paturi. On the Complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[JKR13] Vít Jelínek, Jan Kratochvíl, and Ignaz Rutter. A Kuratowski-type theorem for planarity of partially embedded graphs. *Computational Geometry*, 46(4):466–492, 2013.

[Kar72] Richard M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, pages 85–103. Springer US, 1972.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, may 1983.

[KHN+14] Amruta Khot, Abdeltawab Hendawi, Anderson Nascimento, Raj Katti, Ankur Teredesai, and Mohamed Ali. Road network compression techniques in spatiotemporal embedded systems. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoStreaming - IWGS '14*. ACM Press, 2014.

[KKRW10] Bastian Katz, Marcus Krug, Ignaz Rutter, and Alexander Wolff. Manhattan-Geodesic Embedding of Planar Graphs. In *Graph Drawing*, pages 207–218. Springer Berlin Heidelberg, 2010.

[KLM17] Stephen G. Kobourov, Giuseppe Liotta, and Fabrizio Montecchiani. An annotated bibliography on 1-planarity. *Computer Science Review*, 25:49–67, 2017. ArXiv: http://arxiv.org/abs/1703.02261.

[Kra11] Karl Kraus. *Photogrammetry: geometry from images and laser scans*. Walter de Gruyter, 2011.

[KW08] Marcus Krug and Dorothea Wagner. Minimizing the Area for Planar Straight-Line Grid Drawings. In *Proceedings of the 15th International Symposium on Graph Drawing*, pages 207–212. Springer, 2008.

[LMM18] Anna Lubiw, Tillmann Miltzow, and Debajyoti Mondal. The Complexity of Drawing a Graph in a Polygonal Region. In Therese C. Biedl and Andreas Kerren, editors, *Proc. 26th Int. Symp. Graph Drawing Netw. Vis.*, volume 11282, pages 387–401, 2018.

[Löf16] Andre Löffler. Snapping Graph Drawings to the Grid. Master's thesis, Julius-Maximilians-Universität Würzburg, 2016. Available at `http://www1.pub.informatik.uni-wuerzburg.de/pub/theses/2017-loeffler-master.pdf`.

[LvDW16] Andre Löffler, Thomas van Dijk, and Alexander Wolff. Snapping Graph Drawings to the Grid Optimally. In *Proceedings of the 26th International Symposium on Graph Drawing*, pages 144–151. Springer, 2016.

[LW70] Don R. Lick and Arthur T. White. k-Degenerate Graphs. *Canadian Journal of Mathematics*, 22:1082–1096, 1970.

[Mil95] Victor J Milenkovic. Practical methods for set operations on polygons using exact arithmetic. In *CCCG*, pages 55–60. Citeseer, 1995.

[MKNF87] Sumio Masuda, Toshinobu Kashiwabara, Kazuo Nakajima, and Toshio Fujisawa. On the NP-completeness of a computer network layout problem. In *Proc. IEEE Int. Symp. Circuits and Systems*, pages 292–295, 1987.

[MN90] Victor J Milenkovic and Lee R Nackman. Finding compact coordinate representations for polygons and polyhedra. *IBM Journal of Research and Development*, 34(5):753–769, 1990.

[MNR13] Tamara Mchedlidze, Martin Nöllenburg, and Ignaz Rutter. Drawing Planar Graphs with a Prescribed Inner Face. In Stephen K. Wismath and Alexander Wolff, editors, *Proc. 21th Int. Symp. Graph Drawing Netw. Vis.*, volume 8242, pages 316–327, 2013.

[MS97] Bruce A McCarl and Thomas H Spreen. *Applied mathematical programming using algebraic systems*. Texas A&M University, 1997.

[MU18] Tamara Mchedlidze and Jérôme Urhausen. β-Stars or On Extending a Drawing of a Connected Subgraph. In Therese C. Biedl and Andreas Kerren, editors, *Proc. 26th Int. Symp. Graph Drawing Netw. Vis.*, volume 11282, pages 416–429, 2018.

[Nak00] Tomoki Nakamigawa. A generalization of diagonal flips in a convex polygon. *Theoretical Computer Science*, 235(2):271–282, 2000.

[NDG+16] Alberto Noronha, Anna Dröfn Daníelsdóttir, Piotr Gawron, Freyr Jóhannsson, Soffía Jónsdóttir, Sindri Jarlsson, Jón Pétur Gunnarsson, Sigurður Brynjólfsson, Reinhard Schneider, Ines Thiele, and Ronan M. T. Fleming. ReconMap: an interactive visualization of human metabolism. *Bioinformatics*, page btw667, dec 2016.

[Nöl05] Martin Nöllenburg. Automated Drawing of Metro Maps. Master's thesis, Fakultät für Informatik, Universität Karlsruhe, 2005. Available at `http://www.ubka.uni-karlsruhe.de/indexer-vvv/ira/2005/25`.

[NW11] Martin Nöllenburg and Alexander Wolff. Drawing and Labeling High-Quality Metro Maps by Mixed-Integer Programming. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):626–641, 2011.

[Pac06] Eli Packer. Iterated snap rounding with bounded drift. In *Proceedings of the 22nd Annual Symposium on Computational Geometry*, pages 367–376. ACM, 2006.

[Pac19] Eli Packer. 2D Snap Rounding. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019.

[Pat06] Maurizio Patrignani. On Extending a Partial Straight-line Drawing. *International Journal of Foundations of Computer Science*, 17(5):1061–1070, 2006.

[PS86] Andrzej Proskurowski and Maciej Syslo. Efficient Vertex- and Edge-Coloring of Outerplanar Graphs. *SIAM Journal on Algebraic and Discrete Methods*, 7:131–136, 1986.

[PSS96] J. Pach, F. Shahrokhi, and M. Szegedy. Applications of the crossing number. *Algorithmica*, 16(1):111–117, 1996.

[PT97] János Pach and Géza Tóth. Graphs drawn with few crossings per edge. *Combinatorica*, 17(3):427–439, 1997.

[Pup18] Sergey Pupyrev. Mixed Linear Layouts of Planar Graphs. In *Lecture Notes in Computer Science*, pages 197–209. Springer International Publishing, 2018.

[PW01] János Pach and Rephael Wenger. Embedding Planar Graphs at Fixed Vertex Locations. *Graphs and Combinatorics*, 17(4):717–728, 2001.

[PW14] Dongliang Peng and Alexander Wolff. Watch Your Data Structures! In *Proc. 22nd Annual Conference of the GIS Research UK*, 2014.

[Rin65] Gerhard Ringel. Ein Sechsfarbenproblem auf der Kugel. *Abhandlungen aus dem Mathematischen Seminar der Universit"at Hamburg*, 29(1):107–117, 1965.

[RS84] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[Sch90] Walter Schnyder. Embedding Planar Graphs on the Grid. In David S. Johnson, editor, *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms.*, pages 138–148, 1990.

[Sch14] Marcus Schaefer. The graph crossing number and its variants: A survey. *Electronic Journal of Combinatorics*, DS21, 2013 and 2014.

[SE05] Niklas Sorensson and Niklas Een. MiniSat v1. 13 – A Sat Solver with Conflict-Clause Minimization. *Theory and Applications of Satisfiability Testing*, (53):1–2, 2005.

[Sha78] Micheal Ian Shamos. *Computational Geometry*. Yale University, 1978.

[SHDZ02] Shashi Shekhar, Yan Huang, Judy Djugash, and Changqing Zhou. Vector map compression. In *Proceedings of the tenth ACM International Symposium on Advances in Geographic Information Systems - GIS '02*. ACM Press, 2002.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[TSF+13] Ines Thiele, Neil Swainston, Ronan M T Fleming, Andreas Hoppe, Swagatika Sahoo, Maike K Aurich, Hulda Haraldsdottir, Monica L Mo, Ottar Rolfsson, Miranda D Stobbe, Stefan G Thorleifsson, Rasmus Agren, Christian Bölling, Sergio Bordel, Arvind K Chavali, Paul Dobson, Warwick B Dunn, Lukas Endler, David Hala, Michael Hucka, Duncan Hull, Daniel Jameson, Neema Jamshidi, Jon J Jonsson, Nick Juty, Sarah Keating, Intawat Nookaew, Nicolas Le Novère, Naglis Malys, Alexander Mazein, Jason A Papin, Nathan D Price, Evgeni Selkov, Martin I Sigurdsson, Evangelos Simeonidis, Nikolaus Sonnenschein, Kieran Smallbone, Anatoly Sorokin, Johannes H G M van Beek, Dieter Weichart, Igor Goryanin, Jens Nielsen, Hans V Westerhoff, Douglas B Kell, Pedro Mendes, and Bernhard Ø Palsson. A community-driven global reconstruction of human metabolism. *Nature Biotechnology*, 31(5):419–425, 2013.

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

[vDH14] Thomas C. van Dijk and Jan-Henrik Haunert. Interactive focus maps using least-squares optimization. *International Journal of Geographical Information Science*, 28(10):2052–2075, mar 2014.

[vDL18] Thomas C. van Dijk and Dieter Lutz. Realtime linear cartograms and metro maps. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL '18*. ACM Press, 2018.

[vDvGH+13] Thomas van Dijk, Arthur van Goethem, Jan-Henrik Haunert, Wouter Meulemans, and Bettina Speckmann. Accentuating focus maps via partial schematization. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL'13*. ACM Press, 2013.

[vLA87] Peter J. M. van Laarhoven and Emile H. L. Aarts. *Simulated Annealing: Theory and Applications*. Springer Netherlands, 1987.

[Wil45] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80, dec 1945.

[Wol07] Alexander Wolff. Drawing Subway Maps: A Survey. *Informatik – Forschung & Entwicklung*, 22(1):23–44, 2007.

[WT07] David R. Wood and Jan Arne Telle. Planar decompositions and the crossing number of graphs with an excluded minor. *New York Journal of Mathematics*, 13:117–146, 2007.

[Yan89] Mihalis Yannakakis. Embedding planar graphs in four pages. *Journal of Computer and System Sciences*, 38(1):36–67, 1989.

[Yvi19] Mariette Yvinec. 2D Triangulation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019. https://doc.cgal.org/4.14/Manual/packages.html#PkgTriangulation2.

[Zie08] Günter M. Ziegler. *Polyhedral Surfaces of High Genus*, pages 191–213. Birkhäuser Basel, Basel, 2008.

[ZWF19] Baruch Zukerman, Ron Wein, and Efi Fogel. 2D Intersection of Curves. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019. https://doc.cgal.org/4.14/Manual/packages.html#PkgSurfaceSweep2.

Acknowledgements

- supervisors
- group of wue
- other coauthors
- family

should be one page

List of Publications

- A. Löffler, T. C. van Dijk, A. Wolff.
  Snapping Graph Drawings to the Grid Optimally.
  In: *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization 2016 (GD16).* pp. 144-151. Springer (2016).

- S. Chaplick, M. Kryven, G. Liotta, A. Löffler, A. Wolff.
  Beyond Outerplanarity.
  In: *Proceedings of the 25th International Symposium on Graph Drawing and Network Visualization 2017 (GD17).* pp. 546-559. Springer (2017).

- T. C. van Dijk, T. Greiner, B. den Heijer, N. Henning, F. Klesen, A. Löffler.
  Wüpstream: efficient enumeration of upstream features (GIS cup).
  In: *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '18).* pp. 626-629. ACM (2018).

- M. Beck, J. Blum, M. Kryven, A. Löffler, J. Zink.
  Planar Steiner Orientation is NP-complete.
  At: *10th International Colloquium on Graph Theory and Combinatorics (ICGT'18).* Lyon, France, 2018.

- S. Chaplick, P. Kindermann, A. Löffler, F. Thiele, A. Wolff, A. Zaft, J. Zink.
  Stick Graphs with Length Constraints.
  In: *Proceedings of the 27th International Symposium on Graph Drawing and Network Visualization (GD'19).* pp. 3-17. Springer (2019).

- T. C. van Dijk, A. Löffler.
  Practical Topologically Safe Rounding of Geographic Networks.
  In: *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19).* pp. 239-248. ACM (2019).

- P. Angelini, P. Kindermann, A. Löffler, L. Schlipf, A. Symvonis.
  One-Bend Drawings of Outerplanar Graphs Inside Simple Polygons.
  In: *Proceedings of the 36th European Workshop on Computational Geometry (EuroCG'20).* pp 70:1-70:6, Würzburg (2020).