Julius-Maximilians-Universität Würzburg
Institut für Informatik
Lehrstuhl für Informatik I
Effiziente Algorithmen und wissensbasierte Systeme

Masters' Thesis

Snapping Graph Drawings to the Grid

Andre Löffler

Handed in on 26. August 2016

Supervisors:
Prof. Dr. Alexander Wolff
Dr. Thomas C. van Dijk

Zusammenfassung

Der Übergang zwischen Modellen beliebiger Präzision und tatsächlicher Repräsentation im Speicher ist ein klassisches Problem der computerbasierten Geometrie. Innerhalb der Graphzeichen-Gemeinschaft hat sich *snap rounding* als Lösungsansatz etabliert: Graph-Darstellungen werden vereinfacht, indem End- und Schnittpunkte von Kanten auf ein Gitter gerundet werden (z.B. in [GY86, GGHT97, GM98, dBHO07, Her13]). Eine wichtige Eigenschaft dieser Darstellungen ist topologische Ähnlichkeit – zu Gunsten der Rundung dürfen Knoten, Kanten und Facetten zusammenfallen. Gerade für geographische Informationssysteme ist das Vereinfachen von Daten wichtig, da oft große Datenmengen auf limitierten (oft mobilen) Endgeräten verarbeitet werden sollen. Dabei ist ein Verlust topologischer Informationen oft schwerwiegend.

Motiviert von der Anwendung der geographischen Informationssysteme befassen wir uns in dieser Arbeit mit der Frage, wie sich planare Graphen runden lassen, ohne die Topologie des Graphen zu verändern. Wir formalisieren diese Problemstellung unter dem Namen Topologially Safe Snapping: wir suchen eine Rundung eines Graphen mit Knoten auf ganzzahligen Koordinaten, die die gegebene Topologie nicht verändert und deren Gesamtverschiebung aller Knoten möglichst gering ist. Als erstes Resultat liefern wir $\mathcal{NP}$-schwere Beweise für Topologially Safe Snapping für die euclidsche sowie die *Manhattan*-Distanz. Weiter zeigen wir unter anderem, dass keine additive Approximation konstanter Güte existieren kann und Topologially Safe Snapping nicht in *FPTAS* liegt.

Als Lösungsstrategie präsentieren wir ein Ganzzahliges Programm (*integer linear program, ILP*) basierend auf der Arbeit von Nöllenburg & Wolff [NW11] um optimale Rundungen von Graphen zu bestimmen. Obwohl theoretisch funktionsfähig, haben unsere Experimente gezeigt, dass das ILP sich auf Grund von Laufzeitproblemen nicht für große Graphen (mehr als 20 Knoten und/oder große Ausdehnung) eignet. Das von uns vorgestellte mathematische Modell lässt sich beliebig erweitern. Wir demonstrieren dies, indem wir es mit einfachen Mitteln erweitern, und es so zum Erzeugen von planaren und platzminimalen Zeichnungen benutzen zu können.

Um auch für große Graphen einen Lösungsansatz präsentieren zu können, haben wir nach effizienten Heuristiken gesucht. Unser bester Ansatz ist ein Algorithmus, der auf den Facetten des Eingabegraphen operiert. Dieser kann zwar nicht garantieren, immer alle Knoten des Graphen zu runden, jedoch wird zu keinem Zeitpunkt der Ausführung die Topologie der Eingabe verletzt. In Experimenten zeigen wir, dass auch bei dichten Graphen (hohen Knoten/Ausdehnung-Verhältnis), der überwiegende Teil aller Knoten gerundet werden kann. Damit eignet sich die vorgestellte Heuristik als Vorverarbeitungsschritt in der Kompression von geographischen Daten, bei denen topologische Korrektheit im Vordergrund steht.

Offen bleiben somit besonders die Fragen nach anderen Approximationsalgorithmen, einer allgemeingültigen Heuristik und ob sich die Laufzeit des ILP-Ansatzes verbessern lässt.

Contents

1. Introduction

In geographic applications, usually large amounts of data need to be stored and processed. This data often is given as line segments embedded in the plane, such as road networks or other maps. 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. One way to overcome this is by getting rid of unnecessary detail. This can be done by reducing coordinate precision: rounding data points to an underlying grid. We are not the first to try this approach. A formalized definition of this idea can be taken from Guibas & Marimont [GM98]:

Definition 1.1 (Snap rounding):
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 $\mathcal{S}$. Then *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 the following properties:

- *Fixed-precision representation:* All vertices of $\mathcal{A}^*$ are at integer precision coordinates.
- *Geometric similarity:* For each $s \in S$, the approximation s^* lies within the Minkowski sum of s and a pixel at the origin.
- *Topological similarity:* $\mathcal{A}$ and $\mathcal{A}^*$ are "topologically equivalent up to the collapsing of features". 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.

In the computational geometry community, a process called *snap rounding* has been proposed (as in Definition 1.1) and has since become well-established. Designed to overcome problems induced by working with infinite-precision real arithmetic machines (RAMs) [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 (see Section 1.1). But by definition, the output of those algorithms is not topologically safe: vertices, edges or even faces can disappear while rounding.

Motivated by the above geographic information system application and the definition of snap rounding, the question discussed in this thesis can bluntly be stated as the following: "What about topological equivalence?" One can not always fulfill the first two properties and topological

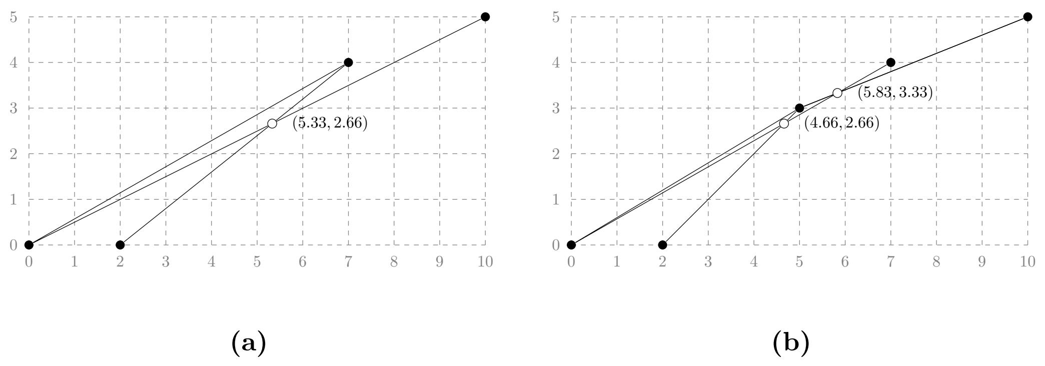


Figure 1.1.: Original motivation for snap rounding: **(a)** instance of four vertices and three edges, found intersection depicted as white vertex, **(b)** rounding the white vertex to the nearest integer grid point produces extraneous intersections.

equivalence at the same time. We will still look for fixed-precision representations of arrangements, but in order to guarantee equivalence instead of similarity, we have to relax on geometric similarity: what is the minimum total change on the vertices coordinates needed to produce a topologically-safe fixed-precision representation? We call this problem Topologially Safe Snapping – related to the *snapping* onto integer grid points.

This thesis is structured as follows: the remainder of this chapter will give an overview on recent work related to rounding and drawing planar graphs in the plane followed by a summary of our contributions. Chapter 2 gives a formal definition of and a $\mathcal{NP}$-hardness proof for Topologially Safe Snapping with some results on related variants and approximability. Chapter 3 introduces an ILP-based approach for Topologially Safe Snapping (that can also be used for space-minimal drawings) and provides experiments evaluating the performance of our model and the applicability on graph drawing tasks. Chapter 4 gives our results on finding a heuristic algorithm for Topologially Safe Snapping and discusses its performance. Finally, Chapter 5 gives some concluding remarks and points to open problems that could not be discussed in this work.

1.1. Related Work

This section subdivides into two fields: efficient snap rounding and drawing planar graphs on the integer grid. We will discuss each topic in roughly chronological order, starting with results concerning rounding.

Greene and Yao [GY86] choose the line intersection problem as paradigm because of it's numerous applications. Given lines with endpoints on integer coordinates, finding intersection points and rounding those to integer coordinates can yield bad results. One mayor problem are *extraneous intersections*: intersections induced by using rounded values for intersection point coordinates – see Figure 1.1. Prior to their algorithm, these intersections have been handled by repeatedly

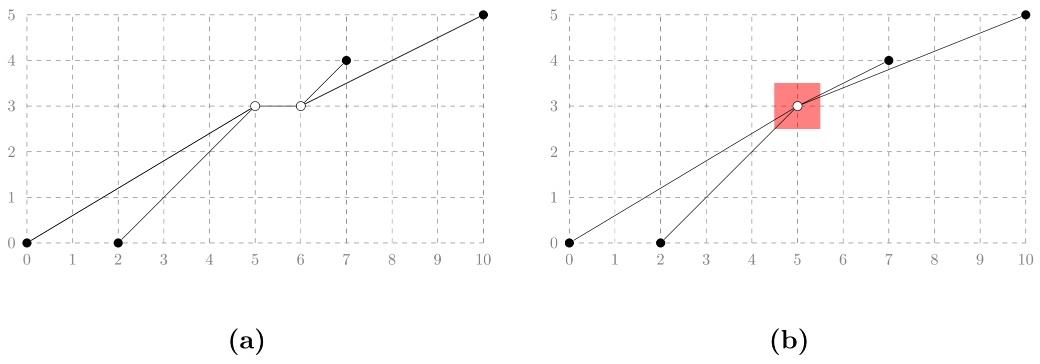


Figure 1.2.: Results of early rounding approaches: **(a)** Greene & Yao **(b)** Hobby (tolerance square in red)

running an intersection detection algorithm (for example the Bentley-Ottmann sweep [BO$^+$79]), until no new intersections are reported. They stated a list of unwanted results and give an efficient algorithm for finding and drawing intersections that avoids all of them. While not solving the problem of extraneous intersections, the algorithm proposed by Greene and Yao finds and rounds all intersections in one single iteration. An example output of their algorithm can be found in Figure 1.2 **(a)**.

Hobby [Hob99] introduces *tolerance squares* – unit square cells with center on an integer grid point. These cells are created wherever a line segment starts, ends or a crossing event occurs. Everything inside a tolerance square is snapped to its center (including other lines by subdividing them and snapping the additional vertex). While this may introduce new incidences, it avoids extraneous intersections. This can be seen in Figure 1.2 **(b)**.

Guibas & Marimont [GM98] give a boiled down definition of snap rounding, found in Definition 1.1 above. Algorithm design and analysis follow the idea of Hobby, using *hot pixels* that are very similar to tolerance squares. They employ *vertical cell decompositions*, introducing *warm pixels* related to the boundaries of the cells. Again, it rounds endpoints and intersections to representable points in a globally topologically consistent way. The algorithm is dynamic and the runtime depends on several variables: Let n be the number of unrounded segments, A the complexity of the ideal arrangement, H the set of hot pixels, $|h|$ the number of unrounded segments intersecting each individual pixel and C related to the complexity of the cell decomposition. (This notions will be used throughout this section.) The runtime of their algorithm is in $O(n \log n + A + \sum_{h \in H} |h|^2 + C)$.

Goodrich, Guibas, Hershberger & Tanenbaum [GGHT97] simplify the approach above by eliminating the need to handle and analyze warm pixels. They give two algorithms: one deterministic, also based on the plane sweep algorithm by Bentley & Ottmann [BO$^+$79]; the other one randomized using trapezoidal decomposition. Both have a matching runtime of (expected) $O(n \log n + \sum_{h \in H} |h| \log n)$ that is truly output-sensitive (the input arrangement A has no influence on total runtime).

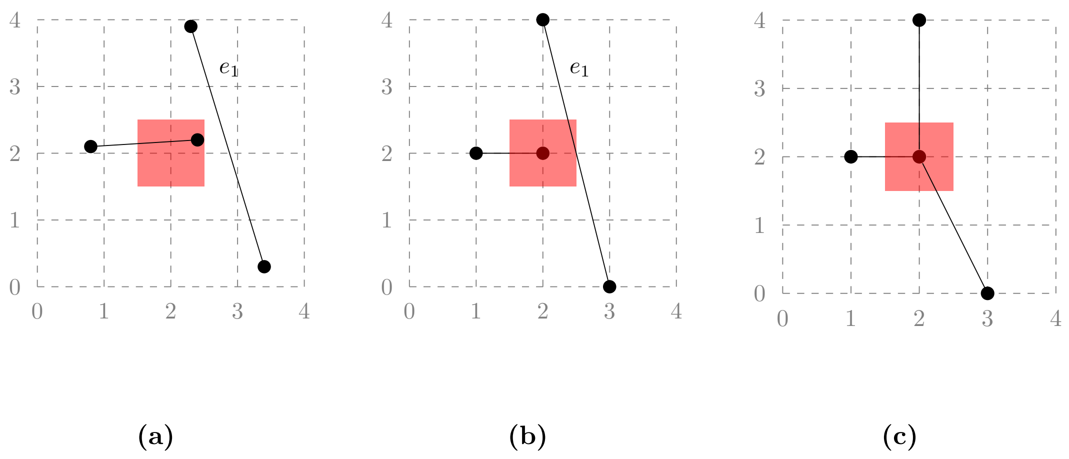


Figure 1.3.: Motivation for iterated snap rounding: **(a)** input segments (e_1 does not intersect red pixel), **(b)** after snap rounding, e_1 intersects red pixel, **(c)** result after iterated snap rounding (e_1 subdivided).

To this point, the focus was on coordinate precision. But in geometric applications, other measurements may require high- to infinite-precision representations as well. Consider Figure 1.3 **(a)** and **(b)**. While the precision of vertex coordinates and vertex-to-vertex distances are bounded by grid resolution, for nonincident vertex-edge pairs vertex-to-edge distances can still be arbitrarily small and thus require higher degrees of precision.

Halperin & Packer [HP02] augment the classic snap rounding procedure. The presented *iterated snap rounding* gives an output that is equivalent to that of repeatedly applying the known hot pixel-based rounding process until any vertex and every nonincident edge are separated by at least half the width of a pixel. An example result can be found in Figure 1.3 from left (input) to right (final result after two iterations). As the output gets degenerated even further, iterated snap rounding may only be useful for some applications. Packer [Pac06] later extends this idea by adding a user-specified parameter to bound any drift on segments, that may be induced by consecutively intersecting other hot pixels.

De Berg, Halperin & Overmars [dBHO07] extend the list of properties desired in a snap rounding output by *non-redundancy*: any degree 2 vertex of the output has to correspond with a segment endpoint (an example can be found in Figure 1.4). They give an algorithm based on two vertical sweeps that produces a snap rounding and eliminates any redundant degree 2 vertices and takes $O((n + I) \log n)$ time (I being the number of intersections).

Hershberger [Her13] divides the set of hot pixels into two groups and gives individual rules for both of them. This approach is called *stable snap rounding* – stable in the sense that re-applying the procedure to its output does not induce new changes – and can be applied to augment different existing snap rounding algorithms: as extension of Hobby's algorithm [Hob99] runtime changes to

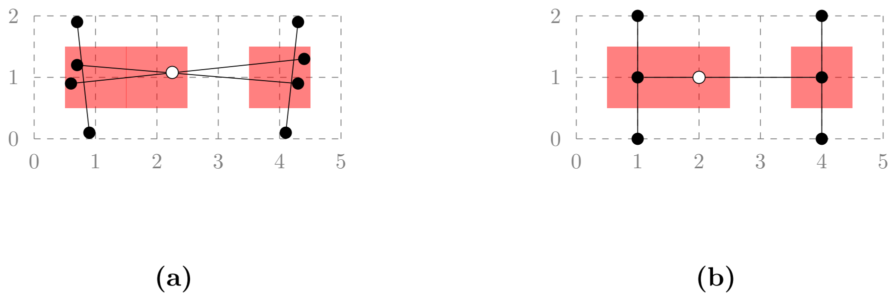


Figure 1.4.: Example for non-redundancy: **(a)** input segments (redundant intersection marked as white vertex), **(b)** output with degree 2 vertex that is not a segment endpoint (white vertex).

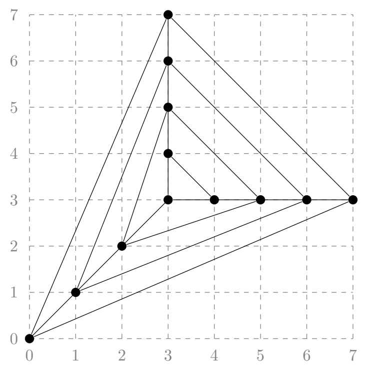


Figure 1.5.: Nested triangles graph on $n = 12$ vertices

$O(|\mathcal{A}(\mathcal{S})| \log n + \sum_{h \in H} |h|)$; algorithms computing bundled representations (as in de Berg et al. [dBHO07]) can be augmented to be stable with $O(|H| \log n)$ additional runtime.

This concludes or overview on snap rounding in two dimensions. Other results (for rounding on the sphere, in higher dimensions, of structures with special properties (like Voronoi diagrams), and others) can be found, but will not be covered here.

From a graph-drawing perspective, restricting to the grid has a (relatively) long history. Fáry [Fár48] (among others) shows that every planar graph has a planar straight line embedding with vertices as points on the plane (also known as *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, Leighton & Trickey [DLT83] give a family of planar graphs called *nested triangles graphs* – see Figure 1.5. They use these to prove an asymptotical area lower bound of $(2n/3-1) \times (2n/3-1)$ for straight line drawings on the integer grid.

Motivated by these results, Schnyder [Sch90] and, independently, de Fraysseix, Pach & Pollack [dFPP90] have shown that any planar graph with n vertices admits a straight-line drawing on a grid of size $O(n) \times O(n)$. This is asymptotically optimal in the worst case [dFPP90]. 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 to show that minimizing the area needed for straight line grid drawings is $\mathcal{NP}$-hard. They also give an iterative algorithm that, for a given plane graph, computes a more compact drawing.

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 that is useful for navigational questions. This drawing is not intended to preserve real-world distances or travel times. While solving a very special problem, the constraints used to do so can be adapted for other geometric tasks (and in fact, we will do so in Section 3.1).

Biedl, Bläsius, Niedermann, Nöllenburg, Prutkin & Rutter [BBN$^+$13] propose a generic ILP model for various grid-based layout problems, such as pathwidth, optimum *st*-orientation or bar k-visibility representations.

Again other results on more specialized problems, like orthogonal drawings or drawings for special maps, can be found, but will not be discussed here.

1.2. Our Contribution

We introduce a novel problem we call Topologially Safe Snapping and show that it is $\mathcal{NP}$-hard. Our reduction uses graphs with vertices at half-integer precision; rounding these graphs implies a compression of a single bit on both coordinates simultaneously. We take two common distance functions – namely *Euclidean*- and *Manhattan* distance – and show $\mathcal{NP}$-hardness of finding exact solutions for both. We also provide some hardness results for finding approximate solutions for variants of Topologially Safe Snapping.

We also give an integer linear program (ILP) for optimal Topologially Safe Snapping – generalizing the mixed integer program for Metro-Map Layout [NW11] – and discuss it in detail. Our program has polynomially many variables and constraints, but is practically very large, limiting its usefulness. The model can be adapted to produce straight line drawings with minimum area or other tasks and we provide some ideas to do so. This is interesting even for small graphs, since minimum area drawings can be useful for validating (counter)examples in graph drawing theory. We back the power of our program by providing experimental results that cover roundings preserving topology as well as minimum area drawings and creating planar drawings from non-planar sketches.

In addition, we introduce a rounding heuristic that can be used to round graphs to the grid efficiently. We provide pseudo-code for the algorithm and use a proof-of-concept implementation to demonstrate performance. While success-rate for rounding every vertex of a graph is well below 100%, it can still be used to reduce precision requirements (and thus safe storage space) for a reasonable amount of vertices. We demonstrate this on randomly generated planar graphs and compare induced vertex movement to optimal results provided by the ILP approach.

Parts of this thesis – namely the $\mathcal{NP}$-hardness proof of Section 2.2 and the ILP formulation of Section 3.1 – have been accepted as a short paper [LvDW16] at the 26th International Symposium on Graph Drawing.

2. NP-Hardness

As stated in Section 1.1, there are several efficient approaches for snap-rounding planar graphs that do not violate planarity while rounding, but are only "topologically similar": equivalence up to the collapsing of features, a goal stated by Guibas and Marimont [GM98].

This leads to the question if there can also be an efficient algorithm that solves the problem of rounding to integer coordinates in an optimal and topologically safe fashion. In this chapter, we will evaluate the computational complexity of the problem we call Topologially Safe Snapping.

Problem 2.1 (Topologically Safe Snapping):

Input:	Planar graph $G = (V, E)$ with Fáry embedding, bounding box $[0, X_{\max}] \times [0, Y_{\max}]$.
Output:	Rounding of G's vertices to integer coordinates with minimal total movement, that does not alter the topology.

To measure movement of a vertex v, we consider the starting position $P_v = (X_v, Y_v)$ in the Fáry embedding and the integer coordinates $p_v = (x_v, y_v)$ it is rounded to. Then the movement $m(v)$ is calculated using the *Manhattan* distance function d_M for points in the plane:

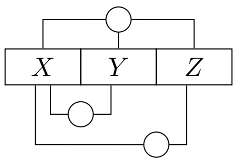
$$m(v) = d_M(p_v, P_v) = |x_v - X_v| + |y_v - Y_v|$$

The total movement of a graph now is the sum over all individual vertex movements:

$$m(V) = \sum_{v \in V} m(v)$$

We will show that Problem 2.1 is $\mathcal{NP}$-hard. In the following sections we first give the definition of another $\mathcal{NP}$-complete problem, planar monotone 3SAT, then give a Karp reduction to prove Theorem 2.1 and consider hardness of other variants related to Topologially Safe Snapping. Finally we have a look at the approximability of Topologially Safe Snapping.

$$B = (\overline{X} \vee \overline{Y} \vee \overline{Z}) \wedge (X \vee Y) \wedge (X \vee Z)$$



(a) **(b)**

Figure 2.1.: **(a)** Formula B in Planar Monotone 3SAT, **(b)** corresponding graph $H(B)$.

2.1. Planar monotone 3SAT

In this section, we describe the problem known as planar monotone 3SAT. This is the $\mathcal{NP}$-complete problem we will give a Karp reduction from.

The Boolean Satisfiability Problem (SAT) is well known as the following: Given a boolean formula B, is there a consistent valuation of its variables with either true or false such that B evaluates to true? If such an valuation exists, B is called satisfiable. Otherwise B is called unsatisfiable.

The problem 3SAT is a variant of the Satisfiability Problem, where B is in 3 Conjunctive Normal Form (3CNF). A formula B is in 3CNF, if B is a conjunction of all of its clauses, where every clause is a disjunction of at most 3 variables.

For a formula B in 3CNF, $H(B) = (V, E)$ is the induced graph with V being divided into vertices for variables and clauses. There is an edge between a variable-vertex v_v and a clause-vertex v_c if and only if variable v occurs in clause c. Planar 3SAT is the decision problem, whether a formula B in 3CNF with planar graph $H(B)$ is satisfiable or not. Planar 3SAT is known to be $\mathcal{NP}$-complete as shown by Lichtenstein in 1982 [Lic82].

A formula in Planar 3SAT is called *monotone* if and only if in each clause all occurrences of variables are either all negated or all unnegated. This gives us the following problem definition:

Problem 2.2 (Planar Monotone 3SAT):

Input:	A formula B in 3CNF that is monotone and planar.
Output:	Is B satisfiable?

Problem 2.2 has been shown to be $\mathcal{NP}$-complete. A proof can be found in the appendix of [dBK12]. For a formula B in Planar Monotone 3SAT, the graph $H(B)$ can be drawn as follows: Variable vertices are aligned on a straight horizontal line. Vertices for negated clauses are drawn atop the variables, vertices for unnegated clauses are drawn below. Edges consist of horizontal and vertical line segments with at most one bend per edge. An illustration can be found in Figure 2.1.

2.2. Hardness Proof

In this section, we will show that Problem 2.1 is $\mathcal{NP}$-hard. To do so, we consider the decision-variant of TOPOLOGIALLY SAFE SNAPPING and show that it is $\mathcal{NP}$-complete, a standard technique that can be found in [GJ79, page 114][1].

Problem 2.3 (Cost-Bounded Topologically Safe Snapping):

Input:	Instance I of TOPOLOGIALLY SAFE SNAPPING with graph $G = (V, E)$, Cost bound $c_{\min}$.
Output:	Is there a solution for I with total movement $m(V) \leq c_{\min}$?

Theorem 2.1:
Cost-Bounded TOPOLOGIALLY SAFE SNAPPING is $\mathcal{NP}$-complete.

Proof. Consider an instance of Problem 2.2 for a formula F. From F we construct a cost bound $c_{\min}$ and a plane graph G with vertices at half-integer coordinates that resembles the structure of Figure 2.1 **(b)**. This structure can also be recognized in various other proofs regarding hardness for problems on planar graphs, for example see [Wol07] or [Cab06]. The optimal total vertex movement for G induced by rounding to integer coordinates is exactly $c_{\min}$ if and only if F is satisfiable. To achieve this, we introduce gadgets for the elements of $H(F)$ – variable- and clause-vertices, edges and bends – and construct G and $c_{\min}$ in polynomial time.

For exposition, we consider two types of vertices in G:

- 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 $V_W \subset V(G)$ start at grid cell centers and thus will always move at least 1 by rounding. (Recall we use Manhattan distance to measure vertex movement.)

Now we give the construction of the various gadgets.

First, we introduce the line and bend gadgets. These are used to ensure consistency of the rounding between the the variable and the clause gadgets. Every segment of the line gadget consists of four black vertices and two edges forming a *tunnel*. Inside the tunnel, there is a single white vertex connected to right (or top) endpoints. The white vertex can be rounded most cheaply to exactly two possible integer grid points, depicted by the red and blue arrows in Figure 2.2 (top-center). By rounding a white vertex in one direction, we prohibit the neighbor in that direction to go

[1] "Whenever we show that a polynomial time algorithm for the search problem could be used to solve the corresponding decision problem in polynomial time, we are actually giving a Turing reduction between them, and hence an NP-completeness result for the decision problem can be translated into an NP-hardness result for the search problem."

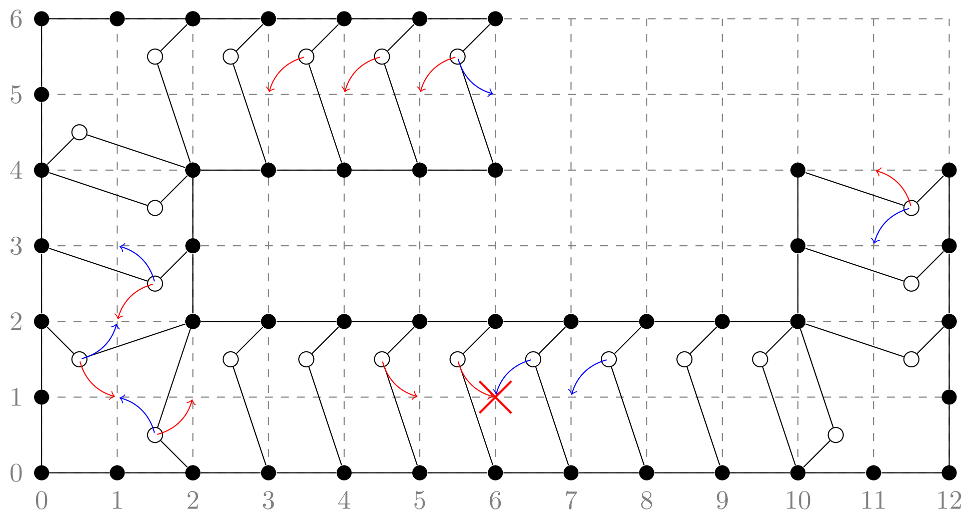


Figure 2.2.: Line and bend gadgets: (top-center) Idea behind pushes, (bottom-left) pushes around corners, (bottom-center) synchronization from end to end.

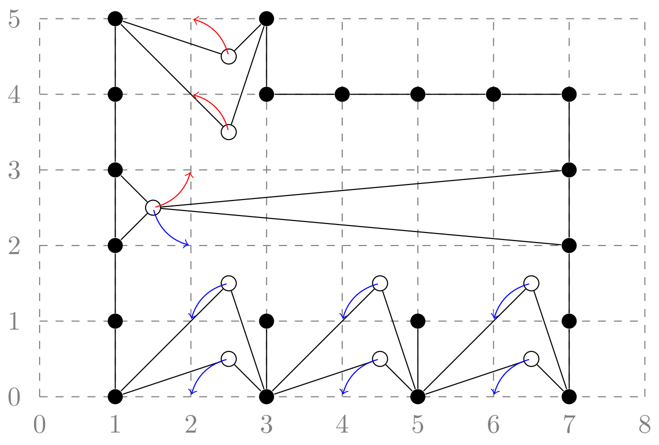


Figure 2.3.: Variable gadget with one negated and three unnegated occurrence.

the opposite way – as both vertices would end up on the same integer grid point (which violates topological safety). If a white vertex is forced to go in one direction (as the other direction may be blocked), we say it is *pushed*. As seen in Figure 2.1 **(b)**, lines may have up to one orthogonal bend. Similar to line gadgets, bend gadgets – Figure 2.2 (bottom-left) – can also be rounded most cheaply in two ways. This is used to transmit pushes through orthogonal bends. If the white vertex at one end of a combination of continuous line and bend gadgets is pushed, topological safety makes sure that the white vertex at the other end also receives a synchronous push – the push is *transmitted*.

Next, consider the variable gadget depicted in Figure 2.3. It has tunnels for vertical line gadgets for every negated and unnegated occurrence at the top and bottom respectively. At the center of this gadget, there is a white vertex that is connected to the gadget's walls by two triangles. Call this the *assignment* vertex. It can be rounded up or down, making the edges of the triangles block all integer grid points on either line. As the white vertices at the openings of the tunnels only

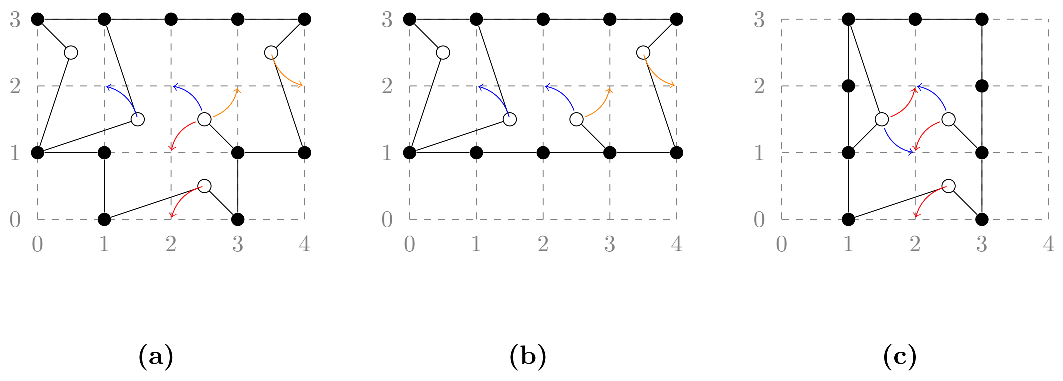


Figure 2.4.: Clause gadgets for all-negated clauses: **(a)** 3 variables, **(b)** 2 variables, **(c)** 1 variable.

have one remaining grid point to be rounded to, pushes are generated and connected line and bend gadgets are forced to transmit them. If F is satisfiable, a truth assignment can be extracted by looking at the new positions of all assignment vertices – rounded down equals TRUE and rounded up equals FALSE.

Finally, the clause gadget is shown in Figure 2.4. We give descriptions for the all-negated versions; all-unnegated versions can be constructed similarly by flipping horizontally. At the gadgets' center, there is a white *satisfaction* vertex (the degree 1 vertex). All possible roundings for this vertex are depicted by colored arrows pointing to grid points that belong to connected tunnels. These grid points are only available if the tunnel does not transmit a push. Then the satisfaction vertex can be rounded at cost 1 if and only if there is one tunnel that does not transmit a push.

The bounding box for the constructed instance of TOPOLOGIALLY SAFE SNAPPING can be picked according to the size of G.

The rounding cost of G is bounded from below by $c_{\min} = |V_W|$, since every white vertex must be rounded at cost at least 1. If F is satisfiable, there is a rounding that achieves this, because then we can round the assignment vertices such that the satisfaction vertices can be rounded at cost 1. In the other direction, a satisfying assignment can be read off from the assignment vertices if rounding occurred at cost $c_{\min}$.

If no candidate grid point is available for the satisfaction vertex, a topologically correct rounding must move a black vertex associated with that clause (of either the clause itself, the connected variables or the edges and bends connecting them); we say, a *jump-out* occurs for that clause. This adds at least 1 to the rounding cost without reducing the movement of any white vertex and thus such solutions cost strictly more than $c_{\min}$. An example for this can be found in Figure 2.5. 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. $\square$

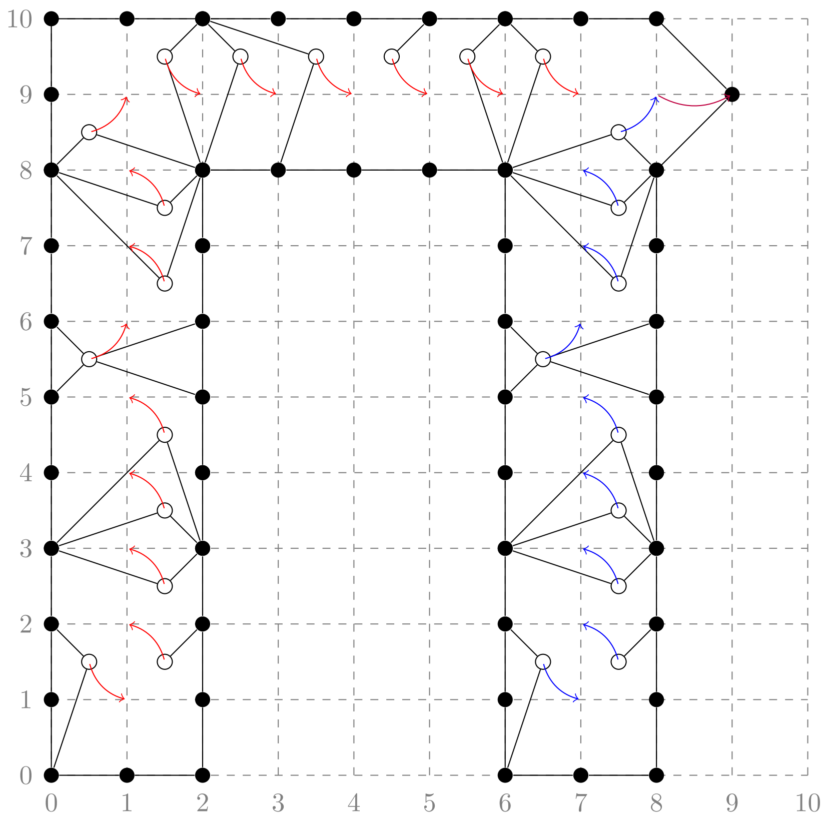


Figure 2.5.: Unsatisfiable clause $F = (\overline{X}) \wedge (\overline{Y}) \wedge (X \vee Y)$ with additional rounding cost of 1. Every white vertex is rounded at cost 1, plus the additional cost of 1 for the jump-out (moving one black vertex, depicted as purple arrow).

Theorem 2.2 (Topologically Safe Snapping):
Topologially Safe Snapping is $\mathcal{NP}$-hard.

Proof. Given an instance I of Topologially Safe Snapping with graph G. If we could produce a topologically-safe rounding (or see if there is none) for G at a given cost bound c in polynomial time, we could search for the minimum cost c' also in polynomial time.

Under the assumption that $\mathcal{P} \neq \mathcal{NP}$, Theorem 2.1 states that no such algorithm can exist. Thus it concludes, that Topologially Safe Snapping is $\mathcal{NP}$-hard. □

This result – using *Manhattan* distance d_M for measurements – raises the question about other metrics. Using the *Euclidean* distance to measure, we get the following:

$$m_E(v) = d_E(p_v, P_v) = \sqrt{(x_v - X_v)^2 + (y_v - Y_v)^2}$$

Corollary 2.3 (Euclidean Topologically Safe Snapping):
Topologially Safe Snapping is $\mathcal{NP}$-hard when using *Euclidean* distance.

Proof. Gadgets and general structure of the proof given above remain. For $c_{\min}$ consider the following: $c_{\min} = \sqrt{0.5^2 + 0.5^2} \cdot |V_W|$. Every white vertex still needs to be rounded, the rest of the proof goes through as above. □

While the proof may be very simple, Corollary 2.3 is noteworthy, as *Euclidean* distance is commonly used in geometric applications. Now instead of considering the sum of all vertex movements, another measure may be the maximum of all rounding movements:

$$m_{\max}(V) = \max\{m_E(v) \mid v \in V\}$$

Corollary 2.4 (Euclidean Min-Topologically Safe Snapping):
Consider an instance of *Euclidean* Topologially Safe Snapping with graph $G = (V, E)$. Minimizing $m_{\max}(V)$ is also $\mathcal{NP}$-hard.

Proof. Again follow the idea of the proof for Theorem 2.1. Any white vertex $v_W \in V_W$ has the rounding cost $m_E(v_W) \geq \sqrt{0.5^2 + 0.5^2} \approx 0.707$. For any black vertex $v \in V \setminus V_W$, the rounding cost is either $m_E(v) = 0$ (not moving at all) or at least 1 (moving to the next grid point or even further).

If the clause is satisfiable, then for every white vertex $m_E(v_W) = \sqrt{0.5^2 + 0.5^2}$ holds and no black vertex needs to be moved and thus $m_{\max}(V) = \sqrt{0.5^2 + 0.5^2}$. If otherwise the clause is unsatisfiable, at least one black vertex needs to be moved and so $m_{\max}(V) \geq 1$.

Then for a formula F with graph $H(F) = (V, E)$, any efficient algorithm that could tell if $m_{\max}(V) < 1$ could be used to decide if F is satisfiable. Under the assumption $\mathcal{P} \neq \mathcal{NP}$, no such algorithm can exist. □

Theorem 2.5:
Topologially Safe Snapping is $\mathcal{NP}$-hard in the strong sense.

Proof. The only numerical variables in an instance of Topologially Safe Snapping are vertex coordinates. In the proof of Theorem 2.1 the constructed instances have a bounding box of polynomial size and thus coordinate values are limited polynomially as well. Thus runtime remains exponential in input size when unary representations are used. □

2.3. Approximability

After having shown $\mathcal{NP}$-hardness of Topologially Safe Snapping and several variants, we give an overview on results concerning approximability. First we will show that Topologially Safe Snapping is in *NPO*, the class of *non-deterministic polynomial-time optimization problems.*

Definition 2.1 (*NPO*):
An optimization problem is in *NPO*, if

(1) the set of instances can be recognized in polynomial time;

(2) there is a monotone non-decreasing polynomial q such that for any instance x and any feasible solution y the following holds: $|y| \leq q(|x|)$;

(3) a solution with size bounded by q can be checked for feasibility in polynomial time;

(4) there is a monotone non-decreasing polynomial r, such that the objective function can be computed in $r(|x|, |y|)$ time for any instance-feasible solution pair.

Lemma 2.6:
Topologially Safe Snapping is in *NPO*.

Proof. We will consider the items in the same order as given in Definition 2.1:

(1) Planarity of a given graph can be checked in polynomial time.

(2) The size of any feasible solution is polynomially bounded by the bounding box for the corresponding instance.

(3) Any rounded graph can be checked for planarity and equivalence of topology in polynomial time. Verifying every vertex is at integer-precision coordinates can be done in $O(|V|)$.

(4) The objective function is calculated by comparing old and new position for every vertex. This are $O(|V|)$-many distance calculations, also possible in polynomial time.

□

While fairly obvious, this result is useful later on. Consider the following definition, taken from Van Leeuwen & Van Leeuwen [vLvL11]:

Definition 2.2 (Approximation with constant absolute error):
Let P be a problem and x be an instance of P. Let $OPT(x)$ be the value of an optimal solution for x. P can be approximated within a *constant absolute error* if there exists an algorithm A and a constant $c \geq 0$ such that for any instance x of P, $A(x)$ has runtime polynomial in the size of x and produces a feasible solution with

$$|A(x) - OPT(x)| \leq c.$$

Lemma 2.7:
TOPOLOGIALLY SAFE SNAPPING cannot be approximated with constant absolute error in polynomial time unless $\mathcal{P}$=$\mathcal{NP}$.

Proof. Let G be a graph with vertices either on a grid point or at the center of a grid cell – similar to the gadgets used in the proofs above. Let x be an instance of TOPOLOGIALLY SAFE SNAPPING for G with optimal value $OPT(x)$ and let $c \geq 0$ be a constant. Suppose for contradiction that A is a polynomial time algorithm for TOPOLOGIALLY SAFE SNAPPING with constant absolute error of c, so

$$|A(x) - OPT(x)| \leq c.$$

Using algorithm A, we construct the following algorithm:

```
**Name:** $A_{c+1}(x)$
$G' \leftarrow (c+1)$ copies of $G$ in the plane with non-overlapping corresponding bounding boxes.
$x' \leftarrow$ Instance of TSS with graph $G'$ and unified bounding box.
**return** $A(x')/(c+1)$
```

The algorithm A_{c+1} takes the drawing of the input graph, makes $(c+1)$ copies and places them in the plane with a polynomial amount of space in between. (Recall G being given at half-integer precision.) These copies can be produced in polynomial time, as well as the evaluation of $A(x')$ (by assumption). So A_{c+1} also is a polynomial time algorithm.

By definition of A_{c+1} and construction of x', we get the following equations:

$$\begin{aligned} A(x') &= (c+1) \cdot A_{c+1}(x) \\ OPT(x') &= (c+1) \cdot OPT(x) \end{aligned}$$

Applying these to the approximation guarantee for A gives:

$$\begin{aligned} |A(x') - OPT(x')| &\leq c \\ \Leftrightarrow A(x') - OPT(x') &\leq c \\ \Leftrightarrow (c+1) \cdot (A_{c+1}(x) - OPT(x)) &\leq c \\ \Leftrightarrow A_{c+1}(x) - OPT(x) &\leq \frac{c}{(c+1)} \end{aligned}$$

We know that $A(x) \geq OPT(x)$. From $c > 0$ we get $\frac{c}{(c+1)} < 1$ and by construction of G we know $A_{c+1}(x) \in \mathbb{N}$ and $OPT(x) \in \mathbb{N}$. For $A(x) - OPT(x) \leq 1$, the difference has to be 0. So by using A, we could construct a polynomial time algorithm that computes an optimal solution – a contradiction as long as $\mathcal{P} \neq \mathcal{NP}$ and no such polynomial time algorithm A can exist. $\square$

The definitions listed below are as well taken from Van Leeuwen & Van Leeuwen [vLvL11]:

Definition 2.3 (*FPTAS*, *FIPTAS*, optimum-asymptotic schemes):
Let $P \in \mathit{NPO}$ be a problem, x be an instance of P, n be the size of x and let A be an algorithm.

- A is a *fully polynomial-time approximation scheme* (*fptas*) for P, if $A(x)$ runs in $O(p(\varepsilon^{-1}, n))$ time for some polynomial function p and the output of $A(x)$ has approximation ratio $(1 \pm \varepsilon)$. If P has an *fptas*, we say $P \in \mathit{FPTAS}$.

- A is an *fully input-polynomial-time approximation scheme* (*fiptas*) for P, if $A(x)$ runs in $O(p(n))$ time for some polynomial function p and the output of $A(x)$ has approximation ratio $(1 \pm \varepsilon)$. If P has an *fiptas*, we say $P \in \mathit{FIPTAS}$.

- An approximation scheme S for P is *optimum-asymptotic*, if there is a computalbe threshold function $b : \mathbb{Q}_{\geq 1} \to \mathbb{N}$ and an associated constant ε_b with the property that $b(1/\varepsilon) \leq 1$ for each $\varepsilon > \varepsilon_b$, such that for any $\varepsilon > 0$ and any x the solution $S(x)$ is feasible and if $OPT(x) \geq b(1/\varepsilon)$, then $S(x)$ is an $(1 \pm \varepsilon)$ approximation. Optimum-asymptotic schemes and classes are marked with an $^{\infty}$ superscript.

Optimum-asymptotic approximation schemes guarantee to deliver good results if the input instance is large enough. This leads to the definition of FIPTAS^{∞} in a natural way.

Garey & Johnson [GJ79] discuss the connection between $\mathcal{NP}$-hardness and approximability. They describe a pair of functions $\mathit{length}(I)$ and $\mathit{max}(I)$:

- *length* maps the size of an instance I (the number of symbols used to describe I under some reasonable encoding) to an integer value;

- *max* maps an instance I to an integer value that corresponds to the largest number in I.

We have the following Corollary (from [GJ79, page 141]):

Corollary 2.8:
Let Π be an integer-valued optimization problem. If there exists a polynomial q in two variables and any instance I of Π satisfy the hypothesis

$$OPT(I) < q(\mathit{length}(I), \mathit{max}(I))$$

and if Π is $\mathcal{NP}$-hard in the strong sense, then Π does not have an *fptas* unless $\mathcal{P}$=$\mathcal{NP}$.

This leads to the following theorem:

Theorem 2.9:
Topologially Safe Snapping does not admit an *fptas* unless $\mathcal{P}$=$\mathcal{NP}$.

Proof. The instances considered in the proof for Theorem 2.1 have integer-valued optimal solutions; let I be such an instance. From 2.5 we know that TOPOLOGIALLY SAFE SNAPPING is $\mathcal{NP}$-hard in the strong sense. Let the value of max correspond to the size of the bounding box of I and let $length$ be a function proportional to the number of white vertices present in the graph of I. For any instance with a satisfiable formula, $OPT(I)$ is bounded by the number of white vertices. For any instance with an unsatisfiable formula, $OPT(I)$ is bounded by the number of white vertices plus the cost for all jump-outs. The number of jump-outs necessary is bounded by the number of clauses and the total size of all jump-outs is bounded by the size of the bounding box. The polynomial q can be constructed accordingly. □

Theorem 2.10:
A problem P can be approximated in polynomial time within constant absolute error if and only if it has a *fptas*$^{\infty}$ with a threshold function b that is bounded by a linear function.

Proof. See [vLvL11, Theorem 5.10]. □

Corollary 2.11:
TOPOLOGIALLY SAFE SNAPPING does not have a *fptas*$^{\infty}$ with a linear-bounded threshold function.

Proof. By Lemma 2.7 we know no approximation with constant absolute error exists. The claim follows from Theorem 2.10 immediately. □

Finally we reconsider *Euclidean min*-TOPOLOGIALLY SAFE SNAPPING (Corollary 2.4).

Corollary 2.12:
Euclidean min-TOPOLOGIALLY SAFE SNAPPING is $\mathcal{APX}$-hard.

Proof. Again we look at half-integer precision instances. As in the proof for Corollary 2.4, the rounding cost for white vertices is $\sqrt{0.5^2 + 0.5^2}$. In contrast, the cost for black vertices is at least 1. □

This concludes our discussion on approximability of TOPOLOGIALLY SAFE SNAPPING and related variants.

3. Integer Linear Program

In this chapter, we give an integer linear program (ILP) to solve Topologially Safe Snapping. At first, we describe the mathematical model in use. Then we give an IBM OPL implementation and some discussion on why to convert this into a JAVA application. The last section of this chapter is dedicated to experimental evaluations: we compare the results of snap rounding algorithms to optimal and topologically safe results of our approach; we also use our ILP to produce space-minimal drawings of graphs and compare those to classic graph drawing algorithms. The examples given will demonstrate the usefulness and limitations of our ILP.

3.1. Basic Model

We will follow the the naming conventions introduced in Chapter 2 with the following extension: variables that are constants will be in uppercase; decision variables will be in lowercase. (For some vertex v, recall X_v being the unrounded x-coordinate and x_v the corresponding rounded coordinate.)

The input for our ILP will be a graph $G = (V, E)$ with embedding given as follows:

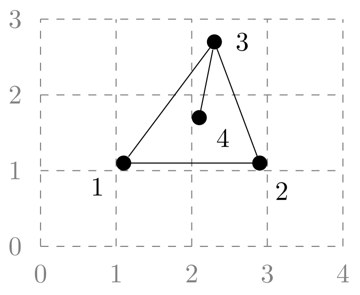
- A set of tuples representing the vertices $v \in V$, consisting of an ID given as integer, x- and y-coordinates X_v, Y_v with double precision and a list of vertex-IDs $N(v)$ representing the embedding ordered counter clockwise around that vertex.

- A array of edges, given as sets of two vertex-IDs.

We also give $X_{\max}$ and $Y_{\max}$ as coordinates for the upper right corner of the bounding box (assuming non-negative vertex coordinates). In addition there is a set of two strings "X" and "Y" and two integers telling the number of vertices and edges in the graph. Those are used for looping through both collections and provide for better code readability. A simple example can be found in figure 3.1.

Using Manhattan distance, the objective function is derived from Problem 2.1 in the natural way:

$$\text{Minimize} \sum_{v \in V} |x_v - X_v| + |y_v - Y_v| \tag{3.1}$$



```
coords = {"X","Y"};
vertexCount = 4;
edgeCount = 4;
nodes = {<1, 1.1, 1.1, {2, 3}>,
         <2, 2.9, 1.1, {3, 1}>,
         <3, 2.3, 2.7, {1, 4, 2}>,
         <4, 2.1, 1.7, {3}> };

edges = [ {1, 2}, {2, 3},
         {3, 1}, {3, 4} ];
xMax = 3;
yMax = 3;
```

(a) **(b)**

Figure 3.1.: Example for ILP input: **(a)** a simple planar graph, **(b)** its representation as ILP input.

Note that this function is not linear but can be transformed to be using standard techniques (for example, see [MS97]). The domain for the two sets of coordinate variables to optimize are defined in a natural way:

$$\begin{aligned} &\forall v \in V: \\ &\quad 0 \leq x_v \leq X_{\max} \\ &\quad 0 \leq y_v \leq Y_{\max} \end{aligned} \tag{3.2}$$

Without any further constraints, this would just move every vertex to the nearest integer grid point. While this is the minimum movement needed to have every vertex rounded to integer coordinates, several problems concerning planarity and embedding come up.

First of all, several vertices could be rounded to the same grid point, creating new incidences. This can easily be prevented by adding the constraints of Equation 3.3. It states that every pair of vertices, they have to differ in either x- or y-coordinate.

$$\begin{aligned} &\forall v, w \in V w \neq v: \\ &\qquad (x_v \neq x_w) \wedge (y_v \neq y_w) \end{aligned} \tag{3.3}$$

We are looking for a topologically-safe rounding and thus for planar input, we require the output to be planar as well. To prohibit edge crossings, we follow the idea given by Nöllenburg & Wolff [NW11]. For every edge of the input graph has some $D_{\min}$ neighborhood that only incident edges are allowed to intersect. This neighborhood is an extension of the edge by $D_{\min}$ in every considered direction $\mathcal{D}$. Applying visual guidelines, Nöllenburg & Wolff could restrict their program to only consider the set $\mathcal{D}$ of eight compass-rose direction vectors and some fixed $D_{\min}$. An illustration of this separation principle can be found in Figure 3.2.

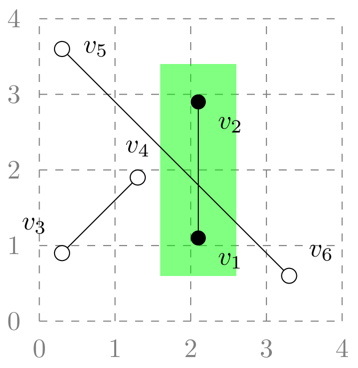
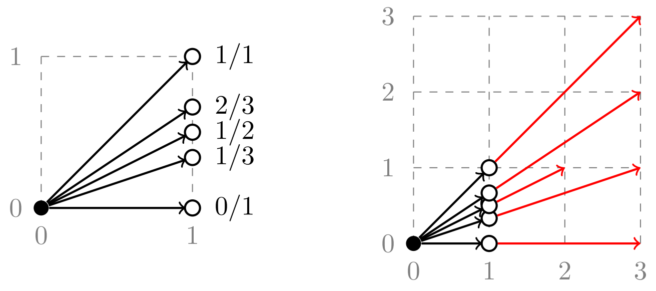


Figure 3.2.: Illustration of octilinear separation with $D_{\min} = 0.5$ and three edges. Any edge intersecting the green area risks intersecting the corresponding edge as well.

F_1:	$\frac{0}{1}, \frac{1}{1}$
F_2:	$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}$
F_3:	$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}$

(a)



(b) **(c)**

Figure 3.3.: Relationship between Farey sequence and grid points: **(a)** The first three elements of the Farey sequence. **(b)** Vectors with origin $(0, 0)$ and endpoint $(1, f), f \in F_3$. **(c)** The extension of the vectors of **(b)** to the 3×3 grid cover all integer grid points in that plane octant.

For arbitrary planar graphs those visual guidelines do not apply in general. To generalize this approach, we have to construct some $D_{\min}$ and some set of directions $\mathcal{D}$ to suite the input graph. In our set of directions we want to have an element corresponding to every unique slope an edge inside the bounding box (connecting integer grid points) can have. These elements will be some finite set of endpoints D of vectors $\overrightarrow{OD}$ (with $O = (0, 0)$), as in Equation 3.4. To construct this set, consider Definition 3.1.

$$\mathcal{D} = \{D = (D_x, D_y) \mid \vec{v} \text{ has origin } (0, 0) \text{ and endpoint } D\} \tag{3.4}$$

Definition 3.1 (Farey sequence [GKP94]):
The *Farey sequence* of order n is the sequence of completely reduced fractions between 0 and 1 which, when in lowest terms, have denominators less or equal than n, arranged in the order of increasing size.

The input bounding box covers only one quadrant of the plane, so the area we need to consider is $\mathcal{A} = [-X_{\max}, X_{\max}] \times [-Y_{\max}, Y_{\max}]$. For $k = \max\{X_{\max}, Y_{\max}\}$, the area $\mathcal{K} = [-k, k] \times [-k, k]$

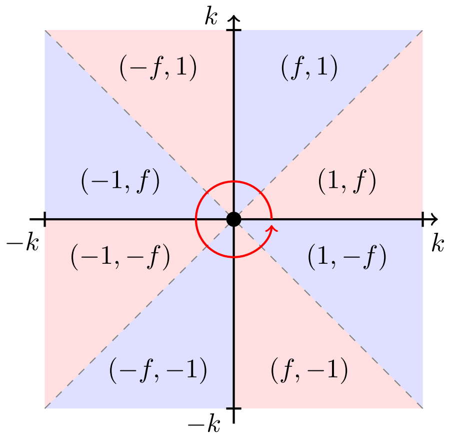


Figure 3.4.: Direction vector endpoint coordinates for every plane octant. Note that $f \in F_k$ with $k = \max\{X_{\max}, Y_{\max}\}$. These endpoints can easily be generated in counter clockwise order around the origin.

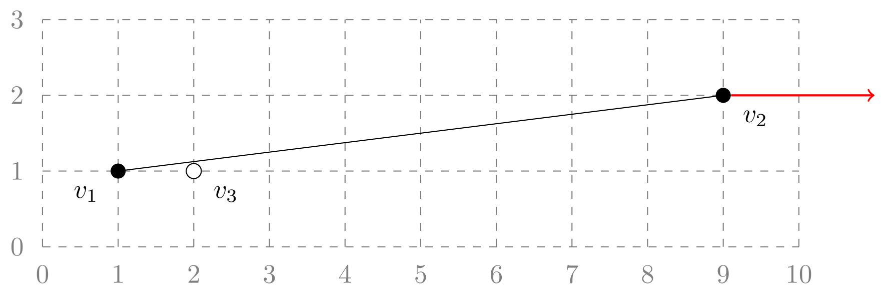
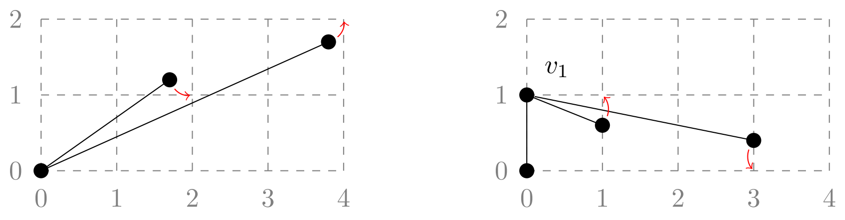


Figure 3.5.: Example for minimum edge distance on integer coordinates. As v_2 moves further to the right, the distance between (v_1, v_2) and v_3 decreases.

covers $\mathcal{A}$. The number of direction vectors in $\mathcal{K}$ is related to the fractions of the k-th element of the Farey sequence, as illustrated in Figure 3.3. We know that $|\mathcal{D}| \in \Theta(k^2)$ and in fact, the fractions can be used to directly construct all possible direction vectors by creating vector endpoints for every fraction with respect to the corresponding plane octant (see Figure 3.4). For a given bounding box, this gives us all possible directions, in which a separation can occur. Note that $\mathcal{D}$ is ordered around the origin (starting at the positive x-axis) and thus allows for comparison between directions by position in $\mathcal{D}$.

For two vertices v_1 and v_2, distance is at least $d_M(v_1, v_2) \geq 1$, but distances between vertices and edges can still be arbitrarily small (with respect to the size of the bounding box). An extreme case is illustrated in Figure 3.5. To find a suitable $D_{\min}$, we have to consider the most extreme slopes of edges that can be present inside the bounding box. It suffices to choose $D_{\min}$ according to Equation 3.5.

$$D_{\min} = \frac{1}{\max\{X_{\max}, Y_{\max}\} + 1} \tag{3.5}$$



(a) **(b)**

Figure 3.6.: Problems related to incident edges. If rounded without care: **(a)** both edges overlap; **(b)** the ordering of $N(v_1)$ changes.

Now we can introduce a binary decision variable γ for every pair of nonincident edges $e_1, e_2 \in E$ and every direction $d \in D$. These variables indicate, whether the edges are are separated in that direction. For every such pair of edges, we require one γ to be set to 1 (see Equation 3.6).

$$\begin{aligned} &\forall e_1, e_2 \in E \forall D \in \mathcal{D} : \\ &\qquad \gamma_D(e_1, e_2) \in \{0, 1\} \\ &\forall e_1, e_2 \in E, e_1, e_2 \text{ nonincident} : \\ &\qquad \sum_{D \in \mathcal{D}} \gamma_D(e_1, e_2) = 1 \end{aligned} \tag{3.6}$$

With these tools, we can formulate the planarity constraints for nonincident edges as follows:

$$\begin{aligned} &\forall D \in \mathcal{D} \forall e_1 = (v_1, v_2), e_2 = (v_3, v_4) \in E, e_1, e_2 \text{ nonincident} : \\ &\qquad D_x \cdot (x_{v_3} - x_{v_1}) + D_y \cdot (y_{v_3} - y_{v_1}) + (1 - \gamma_D(e_1, e_2)) \cdot L_\gamma \geq D_{\min} \\ &\qquad D_x \cdot (x_{v_3} - x_{v_2}) + D_y \cdot (y_{v_3} - y_{v_2}) + (1 - \gamma_D(e_1, e_2)) \cdot L_\gamma \geq D_{\min} \\ &\qquad D_x \cdot (x_{v_4} - x_{v_1}) + D_y \cdot (y_{v_4} - y_{v_1}) + (1 - \gamma_D(e_1, e_2)) \cdot L_\gamma \geq D_{\min} \\ &\qquad D_x \cdot (x_{v_4} - x_{v_2}) + D_y \cdot (y_{v_4} - y_{v_2}) + (1 - \gamma_D(e_1, e_2)) \cdot L_\gamma \geq D_{\min} \end{aligned} \tag{3.7}$$

The large constant $L_\gamma = 2 \cdot \max\{X_{\max}, Y_{\max}\} + 1$ is introduced to apply the *Big M method* for sets of constraints as follows: The value L_γ is chosen to dominate the sum of all other addends. For every pair of nonincident edges one γ is set to 1, indicating the direction in which the edges are separated. If otherwise γ is 0, the constraints of Equation 3.7 are trivially satisfied.

Nonincident edges are only problematic if they intersect after rounding. Incident edges however embrace two different problems that can not be solved by separating them (as they intentionally cross on an endpoint). Incident edges could overlap (the endpoint of one edge ending on the other edge) or the ordering of the neighbors around one vertex could change (altering topology). Both problems are displayed in Figure 3.6. To overcome both problems, we compare the slopes of edges around one vertex by looking at its neighbors. We introduce a set of binary decision variables α as described in Equation 3.8 and make sure that for every vertex-neighbor pair exactly one α is set to 1.

$$\begin{aligned} &\forall v \in V \forall n \in N(v) \forall D \in \mathcal{D} : \\ &\qquad \alpha_D(v,n) \in \{0,1\} \\ &\forall v \in V \forall n \in N(v) : \\ &\qquad \sum_{D \in \mathcal{D}} \alpha_D(v,n) = 1 \end{aligned} \tag{3.8}$$

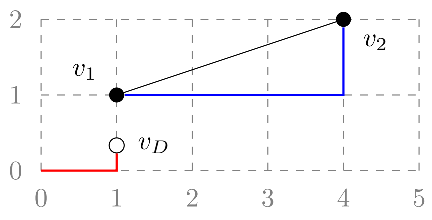
Recall the construction of $\mathcal{D}$ from vectors. The slopes of these vectors represent all slopes possible inside the bounding box. For an edge $(v,w) \in E$, we want to find the direction D with the same slope as vector $\overrightarrow{vw}$ (with same orientation). To do so, consider Equation 3.9.

$$\frac{D_y - 0}{D_x - 0} = \frac{y_w - y_v}{x_w - x_v} \Leftrightarrow x_w \cdot D_y - x_v \cdot D_y + y_v \cdot D_x = D_x \cdot y_w \tag{3.9}$$

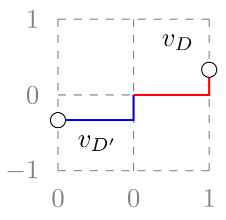
Note that Equation 3.9 is in fact linear. The idea behind this is depicted on the left side of figure 3.7. It can be used to give a set of constraints for every edge as in Equation 3.10. The first two inequalities of Equation 3.10 form an equality that could otherwise not be formulated in an ILP. However, for every edge, there are two possible directions D and D' with $D' = (-D_x, -D_y)$ (the exact opposite direction). This can be filtered out by comparing signs of coordinates, as done in line 3 of Equation 3.10. Again, we employ the Big M method by introducing the large constant $L_\alpha = 2 \cdot \max\{X_{\max}, Y_{\max}\} + 1$.

$$\begin{aligned} &\forall v \in V \forall n \in N(v) \forall D \in \mathcal{D} : \\ &\qquad x_n \cdot D_y + y_v \cdot D_x - x_v \cdot D_y + (1 - \alpha_D(v,n)) \cdot L_\alpha \geq y_n \cdot D_x \\ &\qquad x_n \cdot D_y + y_v \cdot D_x - x_v \cdot D_y - (1 - \alpha_D(v,n)) \cdot L_\alpha \leq y_n \cdot D_x \\ &\qquad (1 - \alpha_D(v,n)) \cdot L_\alpha + (x_n - x_v) \cdot D_x + (y_n - y_v) \cdot D_y \geq 0 \end{aligned} \tag{3.10}$$

Note that for an edge $(v,w) \in E$, the direction is reversed considering the other orientation: $\alpha_D(v,w) = \alpha_{D'}(w,v)$. We now give a set of constraints for both problems stated above, following this idea: for any vertex $v \in V$ two neighbors $n_1, n_2 \in N(v)$ must have $\alpha_{D_1}(v,n_1) = 1$ and $\alpha_{D_2}(v,n_2) = 1$ for some directions $D_1, D_2 \in \mathcal{D}$. As $\mathcal{D}$ is ordered, we can require for n_1 to be before n_2 in counter clockwise orientation, that D_1 is before D_2 in $\mathcal{D}$. This holds for all neighbors but the last. We introduce a binary decision variable β for every vertex-neighbor pair and requiring one β to be set to 1 for every such pair (see Equation 3.11).



(a)



(b)

Figure 3.7.: Vertex-neighbor direction determination. **(a)** The relation between the two red lines connecting $(0, 0)$ with helper node v_D are the same as for the blue lines connecting v_1 with v_2, thus the direction from v_1 to v_2 is D. **(b)** Helper node v_D and its opposite direction helper node $v_{D'}$ with same ratio.

$$\begin{aligned}
&\forall v \in V \forall w \in V : \\
&\qquad \beta(v, w) \in \{0, 1\} \\
&\forall v \in V, \deg v > 1 : \\
&\qquad \sum_{n \in N(v)} \beta(v, n) = 1
\end{aligned} \tag{3.11}$$

For a vertex v and some neighbor $n \in N(v)$, assigning $\beta(v, n) = 1$ means that n is the last neighbor for v and thus is allowed to violate our constraints. Putting things together, we get the constraints of Equation 3.12 below.

$$\begin{aligned}
\forall D_1 \in \mathcal{D} \forall v \in V, N(v) =& \{n_1, n_2, \dots, n_k\} (k = \deg v > 1) : \\
\alpha_{D_1}(v, n_1) \leq& \beta(v, n_1) + \sum_{D_n \in \mathcal{D} : D_n > D_1} \alpha_{D_n}(v, n_2) \\
\alpha_{D_1}(v, n_2) \leq& \beta(v, n_2) + \sum_{D_n \in \mathcal{D} : D_n > D_1} \alpha_{D_n}(v, n_3) \\
&\vdots \\
\alpha_{D_1}(v, n_k) \leq& \beta(v, n_k) + \sum_{D_n \in \mathcal{D} : D_n > D_1} \alpha_{D_n}(v, n_1)
\end{aligned} \tag{3.12}$$

This covers all parts of the model for the ILP and leads to the following theorem:

Theorem 3.1:
The above ILP solves Topologially Safe Snapping.

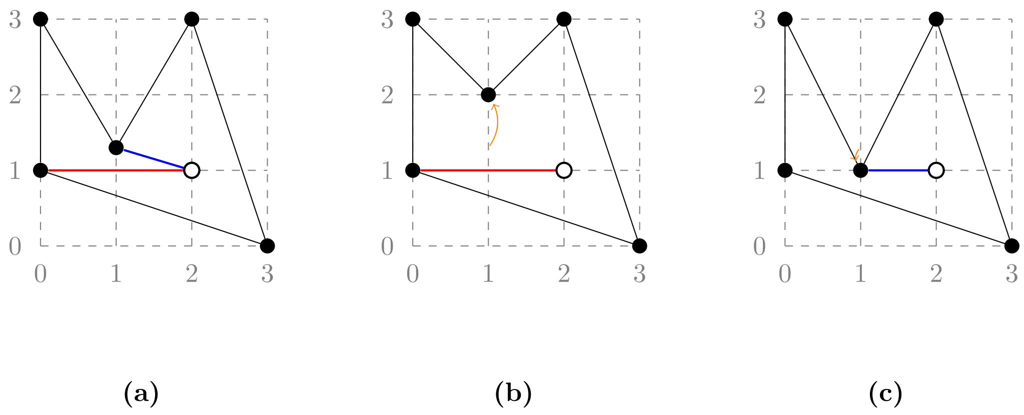


Figure 3.8.: Impact of choice for isolated vertices: **(a)** input graph with isolated vertex (white vertex) and two possible connection edges (red and blue); **(b)** result using red edge with cost 0.7; **(c)** result using blue edge with cost 0.3.

A complete implementation of the model given above can be found in Appendix A.1. Note that this model copes with disconnected graphs, but does not handle isolated vertices (without any neighbor). These vertices are moved to valid integer grid points but do not necessarily stay in the same face as in the input. For rounding those instances, connect those vertices to any vertex of the same face (in a planar way) and delete the additional edge from the output graph. This choice has to be made with caution as it may result in non-optimal roundings – see Figure 3.8.

The model presented above is rather general and thus can be adapted for various other tasks. Adjusting bounding box size gives a tool for checking if a graph has a drawing of that size – and if so, produces that drawing. For space-minimal drawings of graphs (without fixing the outer face), consider the objective function of Equation 3.13 and give a bounding box of size $|V| \times |V|$.

$$\textsc{Minimize} \max\{\{x_v \mid v \in V\} \cup \{y_v \mid v \in V\}\} \tag{3.13}$$

Adding constraints on one coordinate and modifying the objective function to minimize the other coordinate can be used to minimize space consumption in one dimension. An example can be found in Equation 3.14.

$$\begin{aligned} &\textsc{Minimize} \max\{\{y_v \mid v \in V\}\} \\ &\textsc{Subject to:} \\ &\forall v \in V : x_v \leq c \\ &\dots \end{aligned} \tag{3.14}$$

3.2. Implementation

In this section, we give some details on the implementation of the model given above. As stated above, an IBM OPL implementation can be found in the appendix. While being rather short and readable, using IBM OPL has some drawbacks. The model itself also is straight forward to understand but due to its size hard to solve in practice. To overcome both limitations at once, the implementation in use differs from that of the appendix. First we give a list of drawbacks we encountered while implementing and testing the ILP. Then we outline how these have been handled.

The main disadvantage of our ILP is its limited usability because of its prohibitive runtime. The reason for this is the rapidly growing number of constraints for larger graphs (with more vertices or on larger bounding boxes). For an ILP instance, size can be measured considering two parameters: *Columns* gives the number of variables to be considered while *Rows* gives the number of equations to be solved. IBM CPLEX uses a presolver and an aggregator for preprocessing to automatically reduce instance size. Figure 3.9 shows the growth of these numbers for nested triangle graphs (with tight bounding boxes) after the automated preprocessing steps. This observation can be backed considering the decision variables. Again, let $k = \max\{X_{\max}, Y_{\max}\}$. From the definition of the model we have the following:

$$\begin{aligned} |\mathcal{D}| &\in \Theta(k^2) \\ |\{\alpha_D(v,w) \mid D \in \mathcal{D}, v, w \in V\}| &\in O(k^2|V|^2) \\ |\{\beta(v,w) \mid v, w \in V\}| &\in O(|V|^2) \\ |\{\gamma_D(i,j) \mid D \in \mathcal{D}, i, j \in E\}| &\in O(k^2|E|^2) \end{aligned} \tag{3.15}$$

During our experiments (see Section 3.3), we observed prohibitive runtime on most small planar graphs ($|V| \leq 20$), which leads to the conclusion that the number of directions is the limiting factor. Reducing the number of directions – limiting them in any way without losing general optimality – remains an open problem.

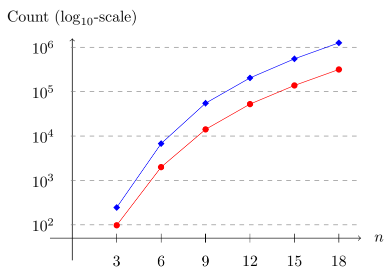
For most input graphs, a large percentage of the constraints will trivially be satisfied by most solutions: edges that are far apart in the plane will not intersect after rounding; for many vertices, rounding to the nearest integer grid point will not introduce new incidences; the ordering of neighbors around one vertex will most likely not change while rounding. A standard approach to reduce instance size is the *row generation* paradigm (see [Chi08]):

1. Select some set of constraints, build an instance without this set and solve it.

2. Examine the solution and check for errors.

3. Add those constraints to the instance that would have prevented these errors and resolve it.

4. Repeat this procedure until no errors are reported.



(a)

Nested triangles graphs on n vertices:

	◆ Rows	● Columns
$n = 3$	246	99
$n = 6$	7260	2016
$n = 9$	54585	14064
$n = 12$	205422	52062
$n = 15$	550779	138762
$n = 18$	1269600	318900

(b)

Figure 3.9.: Growth of ILP instances: **(a)** semi-logarithmic plot of ILP size parameters, **(b)** legend and numbers (after CPLEX presolve steps).

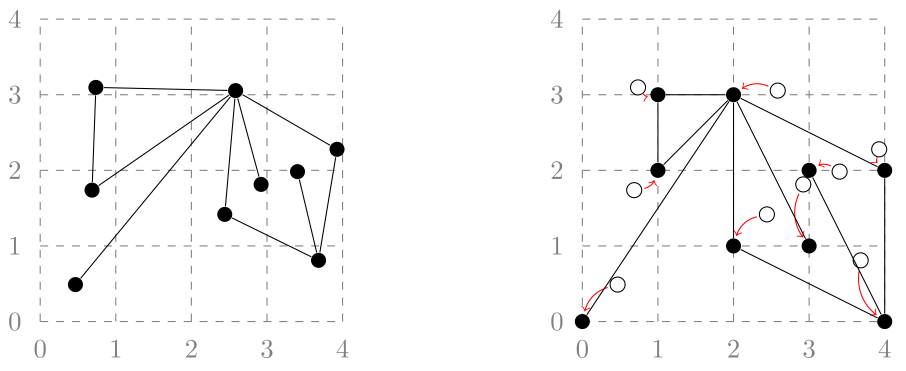
We want use this paradigm to generate most of our constraints. The OPL language however does not allow for delayed constraint generation. IBM suggests switching to some high-level programming language (such as C# or Java) and use the provided bindings to generate models. Following this suggestion, we implemented a component-based Java framework using the OPL Java bindings, JGraphT[1] and JTS Topology Suite[2] to handle planar graphs. Our framework also includes interfaces for general vertex-rounding algorithms along with components for importing and exporting graphs and various configuration options. We implemented a simple topology checker (for planarity and embedding-preservation) that is able to report any edges and vertices involved in topological inconsistencies. The topology checkers' output can be used to generate constraints as needed. We observed that several iterations of solving, checking and adding of constraints were considerably faster on most input graphs than solving the full model in the first place. This will be discussed in the following section.

3.3. Experimental Evaluation

3.3.1. Rounding

In the following, we will discuss performance of both of our implementations of the above ILP (full model and row generation) on graphs different in vertex count, size of bounding box and number of "difficult" parts. The main goal of this section is to give an intuition of what is actually difficult. We provide hand-picked examples to demonstrate the influence of the individual sets of constraints on total runtime.

[1]`http://jgrapht.org/`
[2]`https://sourceforge.net/projects/jts-topo-suite/`



Input Output

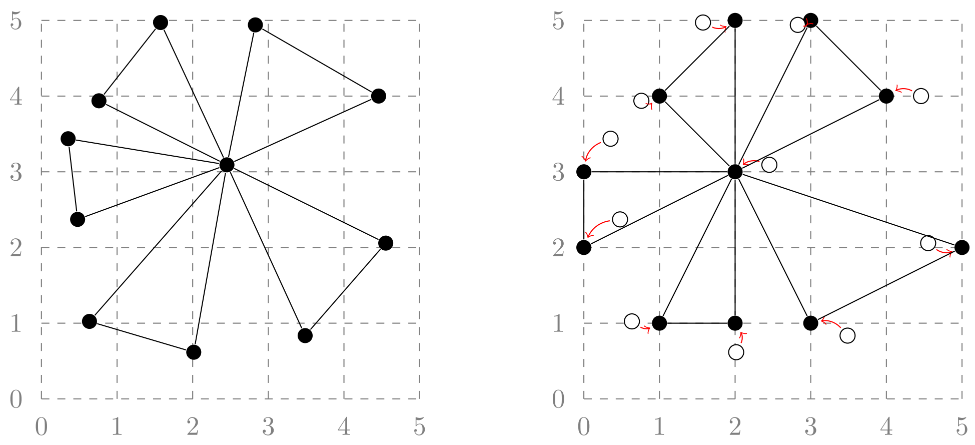
	Full model		Row generation
	first	optimal	optimal
Time	3.2 s	90.6 s	29.2 s

Figure 3.10.: Graph 1: $|V| = 9, |E| = 10$

The figures of this section are structured as follows: left hand side is the input graph on an underlying grid representing the bounding box size; right hand side is the output, white vertices representing the initial positions with the red arrows indicating actual vertex movement. Below is a table giving actual runtimes in wall clock time. In any field, "†" means that within 10 minutes of computation no result could be obtained. In the following, *full model* is used for executions of the above ILP without row generation. The column *first* gives the time until any feasible integer solution (not necessarily optimal) is reported by the integer solver. In both cases, *optimal* gives the time until the solver reports an optimal solution. We ran 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 as described above.

We start with a rather small graph (Figure 3.10). Because of its size, building the model and finding the solution is quick. However, its vertices are positioned so that many constraints are not trivially satisfied and significant effort is required even by the row generation approach.

In the graph of Figure 3.11, the by far most expensive constraint is for checking the embedding of the central node. Any other constraint is easily satisfiable. There is not a big difference between finding the first solution and closing the integrality gap time wise. Notice that every vertex has one preferred integer grid point that is not preferred by any other vertex, so just rounding to the nearest grid point already gives an optimal solution. This solution is found by the first run of the row 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 difference between the full model and the row generation approach becomes even more important when the size of the bounding box increases. Consider the graph of Figure 3.12. While still easy to round in the same

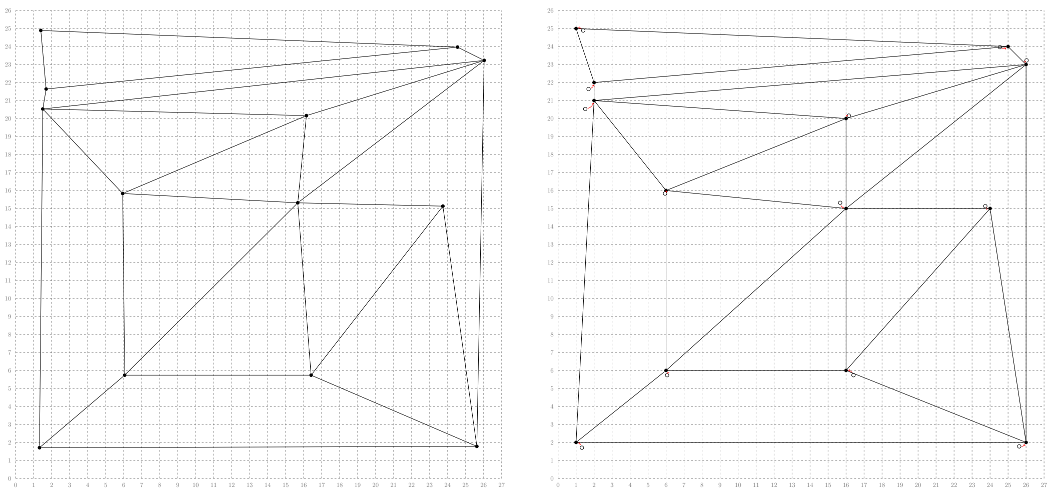


Input

Output

	Full model		Row generation
	first	optimal	optimal
Time	5.2 s	10.6 s	0.5 s

Figure 3.11.: Graph 2: $|V| = 11, |E| = 15$

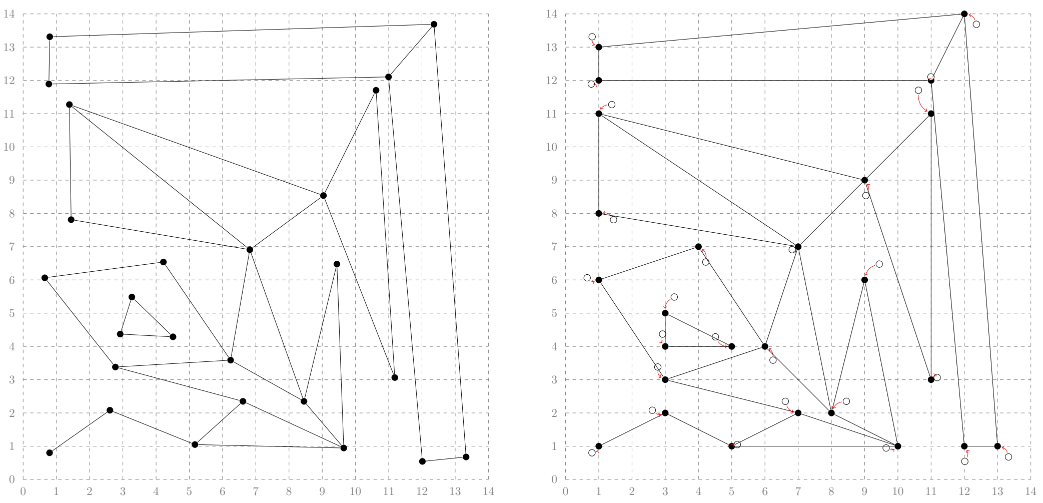


Input

Output

	Full model		Row generation
	first	optimal	optimal
Time	†	†	0.4 s

Figure 3.12.: Graph 3: $|V| = 13, |E| = 25$



Input Output

	Full model		Row generation
	first	optimal	optimal
Time	†	†	7.1 s

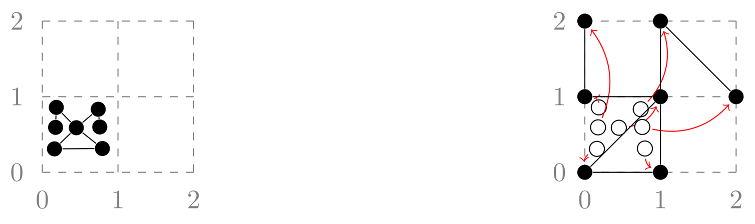
Figure 3.13.: Graph 4: $|V| = 26, |E| = 34$

sense as above, as size of the bounding box increases, so does the time for building and solving the full model. However, this has no impact on solving time for the row generation approach for "easy" graphs.

Consider the graph of Figure 3.13. While too large to round with the full model approach in the allotted time of 10 minutes, the row generation only has to add one constraint from Equation 3.3 for two vertices of the upper-right corner. 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 $\mathcal{D}$.

When rounding graphs with vertices starting in close proximity (like in Figure 3.14) several things can be noticed. First of all, small bounding boxes result in small and easy-to-solve models (as there are few possible directions). The size of the bounding box has extreme impact on the runtime (for comparison see Figure 3.10, which has only two more vertices but a much larger bounding box). Second, when many constraints are violated during the row generation processes, iteratively adding the constraints results in runtimes exceeding the time for solving the full model in the first place.

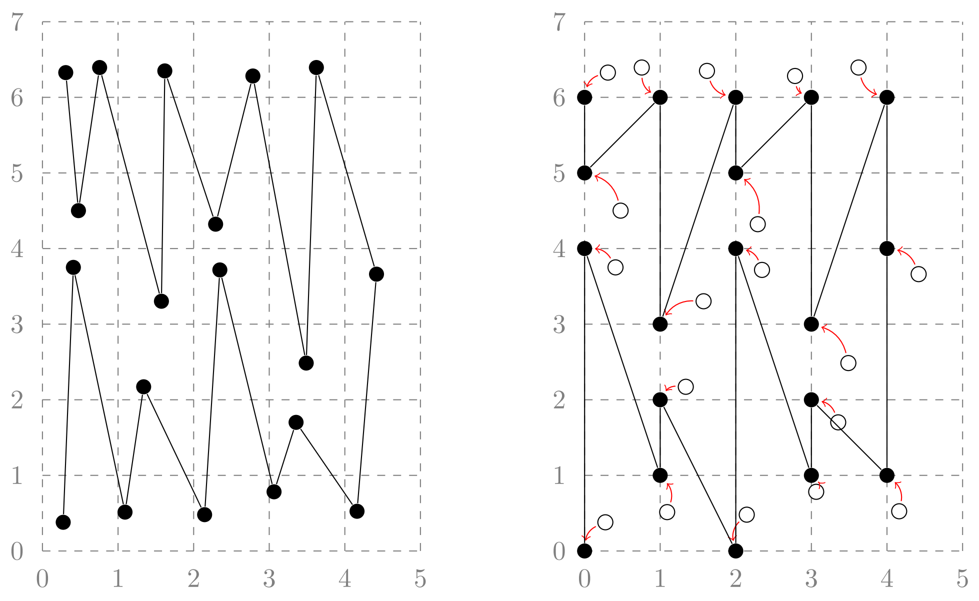
We end this section with two rather small examples (Figures 3.15 and 3.16). Both have a high number of vertices compared to the size of the bounding box and are designed to include many "difficult" parts. The row generation approach clearly outperforms the full model (while still being infeasible in practice).



Input Output

	Full model		Row generation
	first	optimal	optimal
Time	0.5 s	4.8 s	20.2 s

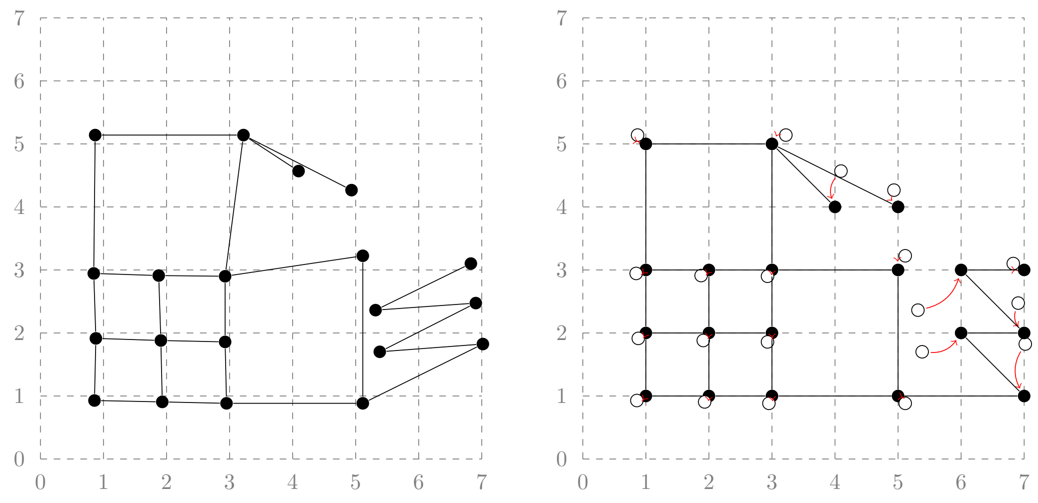
Figure 3.14.: Graph 5: $|V| = 7, |E| = 7$



Input Output

	Full model		Row generation
	first	optimal	optimal
Time	42.6 s	1105.6 s	21.2 s

Figure 3.15.: Graph 6: $|V| = 19, |E| = 18$



Input Output

	Full model first	optimal	Row generation optimal
Time	182.1 s	†	211.6 s

Figure 3.16.: Graph 7: $|V| = 20, |E| = 25$

3.3.2. Drawing

The way we define Topologially Safe Snapping, rounding is closely related to drawing planar graphs (with additional requirements on vertex placement). As described above, our model can be used – directly or with modifications – to produce fixed precision drawings of graphs that fulfill given properties, as being close to an original draft or minimizing space requirements.

In the following, we will give a survey on drawing tasks we managed to solve using the ILP given above. If required, we also give the modifications necessary to compute the drawings presented here. The figures presenting the results will follow the same style as above. Also, as we are looking at extreme or unusual cases, any execution of the ILP was done using the full model approach.

Reconsider the edge separation constraints of Equation 3.7. Given two nonincident edges, the idea is the following: those edges are separated in the input and thus must be separated in the output as well. But the actual constraints to do so don't use the initial separation but just force the construction of a separated output. Provided the existence of a planar drawing, our model can be used as follows: dismiss any constraints preserving the embedding and use the drawing as input, it can construct a crossing free drawing. For a positive example, see Figure 3.17. Running any planarity-checker beforehand is highly recommended if one would use the ILP in such way. While in theory being able to identify nonplanar graphs (by returning "infeasible"), the experiment we ran on the $K5$[3] did not terminate after several hours of computation.

[3]The complete graph on 5 vertices, one of the two *Kuratowski* graphs.

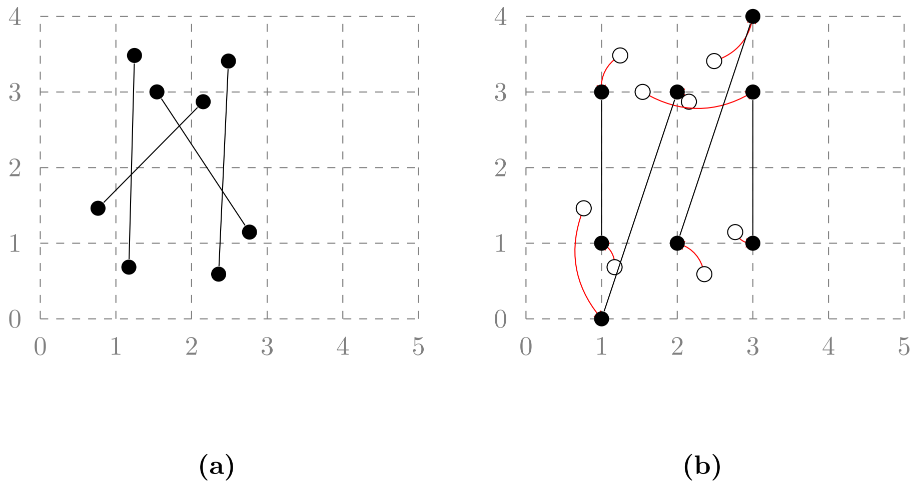


Figure 3.17.: Graph with non-planar drawing and planar ILP output: **(a)** Input graph on 6 vertices and 3 edges with drawing that exhibits 3 unwanted crossings. **(b)** Crossing-free drawing of the same graph, computed in 46.5 s.

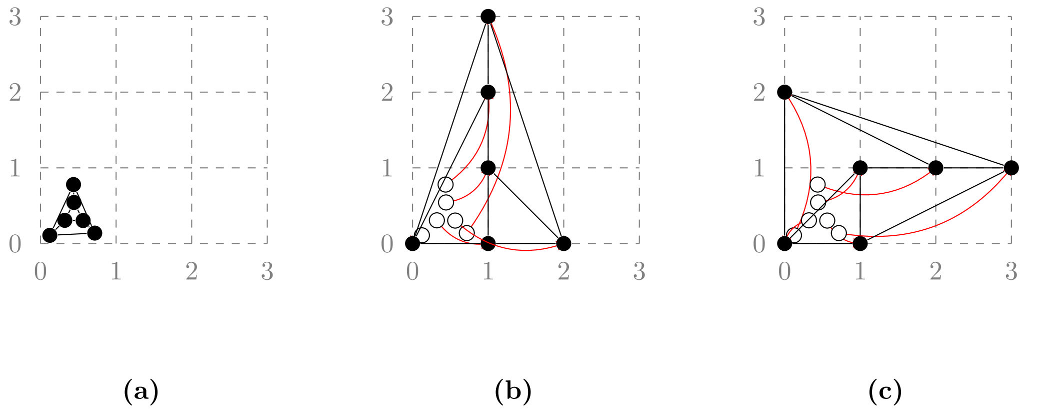


Figure 3.18.: Space-minimal drawings of the nested triangles graph on $n = 6$ vertices: **(a)** the input graph placed completely in one grid cell; **(b)** the ouput using the rounding objective (computation time 3 h 25 min); **(c)** the ouput using the objective of Equation 3.14 and limiting the x-coordinate to 3 (computation time 10 min 31 s).

Another task that has a long history in the graph drawing community is producing space-minimal planar drawings. While Krug & Wagner [KW08] did provide a proof for $\mathcal{NP}$-hardness, we do not know of any tool for computing optimal solutions. Given a planar graph with planar drawing, trying to produce space-minimal drawings subdivides into two cases: fixing the outer [DLT83, dFPP90] face or not. The latter was recently studied by Frati & Patrignani [FP07]. Our model does not include constraints directly working on the faces of a graph. Thus, without including new face-based constraints, we can only compute drawings of the latter as well. Using the complete set of constraints and a drawing of of the nested triangles graph on $n = 6$ vertices, we were able to produce the drawings of Figure 3.18.

4. Heuristic Approach

As seen in Section 3.3, the prohibitive runtime of our ILP approach raises the question about an efficient heuristic. In this chapter, we give a heuristic based on the faces of a graph. We will compare its results and applicability to those of the ILP and we also discuss its performance as a compression algorithm for vertex coordinates of graphs.

4.1. Face-based Rounding

While researching the topic of Topologially Safe Snapping, we came up with several sketches for efficient topologically safe rounding heuristics. We present the most promising among them: Face-based Rounding.

The idea is the following: For graph $G = (V, E)$ consider every vertex $v \in V$ once and in specified order. For v consider only integer grid points inside any face adjacent to v. Choose the nearest *legal* grid point among them and round v to it. We suggest the following algorithm:

Algorithm 1: Face-based Rounding

```
Data: Planar graph $G = (V, E)$ with straight line embedding.
Result: Topologically safe rounding of $G$ to integer coordinates.
$v_1 \leftarrow$ vertex closest to the origin
Sort $V$ according to order of breadth-first search discovery starting at $v_1$
for $v \in V$ do
    $C \leftarrow$ Set of legal integer grid points
    $c \leftarrow$ grid point of $C$ closest to $v$
    Round $v$ to $c$
    Update faces of $G$
return $G$
```

To determine if a grid point is legal, proceed as follows: Consider G' deduced from G by removing v and any edge incident to v. Following the *Art Gallery Problem*-terminology [dBvKOS00], place a camera on every former neighbor of v. A grid point is legal if and only if every camera can see it. This guarantees that rounding v to a legal grid point does not introduce new crossings or incidences caused by edges connected to v. This way, planarity is ensured after every rounding step and thus in the output. Considering only grid points in adjacent faces ensures topological safety in a similar way.

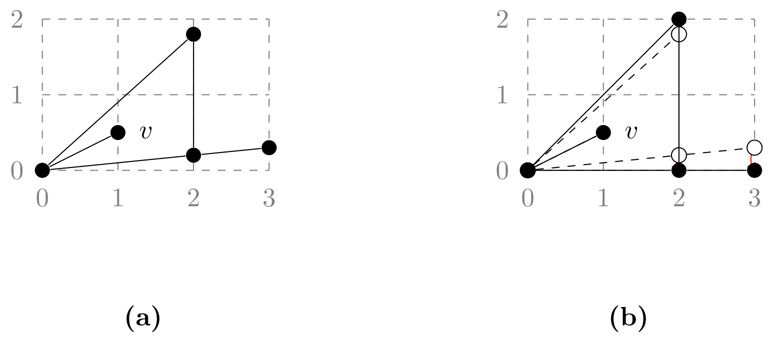


Figure 4.1.: Graph that Algorithm 1 fails to round: **(a)** input graph; **(b)** vertex v remains unrounded, as no legal grid point was available when trying to round it.

Unfortunately, the problem with this approach is rather obvious: We can not guarantee that the set C of legal grid points is non-empty for every vertex v. Also iteratively making locally optimal decisions may result in grid points becoming unavailable. However, not rounding some vertex v does not violate other desired properties of the output. Before trying to round v, the graph G has been planar and topologically safe and will only be modified in a safe way. If in one step not modifying G, we simply did not find an easy way to reduce required precision for v.

One could consider some shifting technique as used in the drawing algorithm by de Fraysseix, Pach & Pollack [dFPP90]. Given the complete drawing of a graph, we can not restrict the slopes of edges in a way similar way and thus we can not provide a reasonable shift distance that ensures planarity. (The need for arbitrarily large shifts has been present in all of our iterative rounding heuristics.)

To further explore this problem, in the following Section we evaluate performance of Algorithm 1 on the instances used in Section 3.3.1 and on sets of randomly generated planar graphs.

4.2. Comparison to ILP Approach

We start this section with instances of Section 3.3.1: first we consider those instances where both the ILP and Algorithm 1 managed to round all vertices followed by those where results differ. As the limitations of Algorithm 1 are rather obvious, we only did a proof-of-concept implementation using simple variants of some components (planarity-checking and computation of grid point visibility) that hardly are optimal considering asymptotic runtime. Therefore, we will not provide detailed comparison on wall-clock runtime of both approach unless a major difference can be noticed.

Consider the graph presented in Figure 3.11. Being the most easy instance we did test our ILP on, unsurprisingly Algorithm 1 did perform exactly the same. Induced movement on vertices is

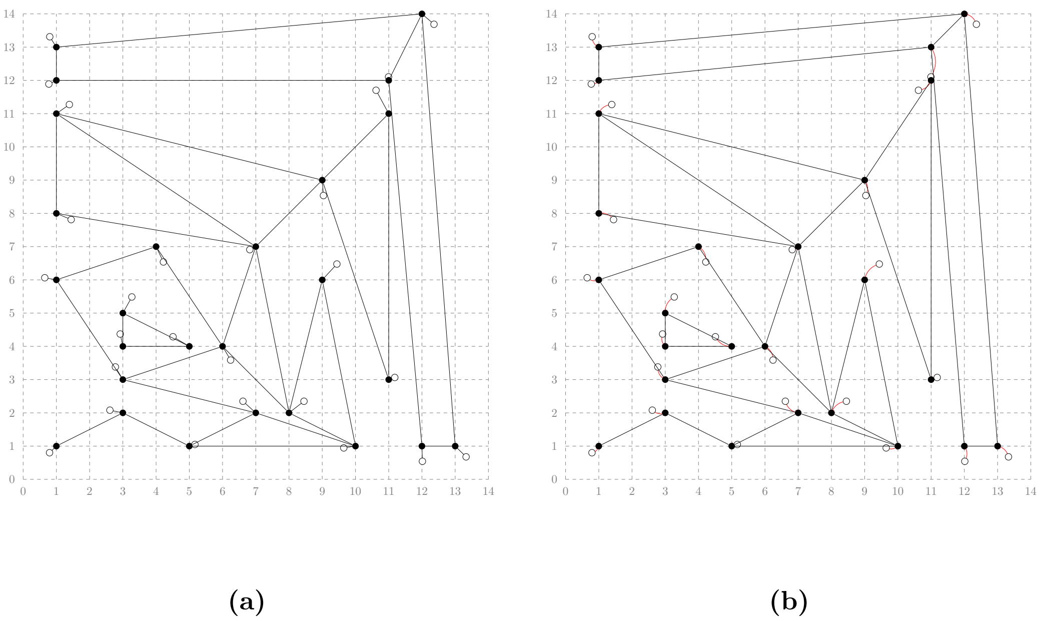


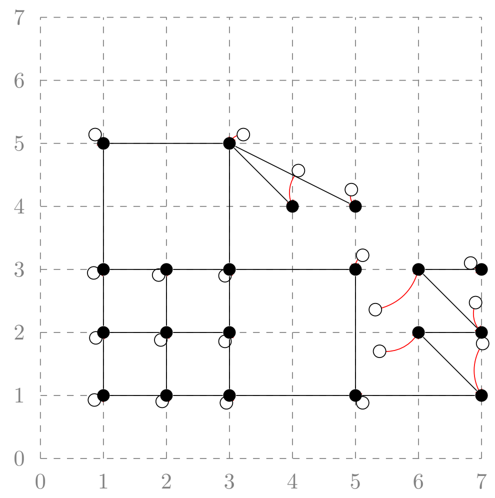
Figure 4.2.: Comparison on Graph 4 (Figure 3.13). **(a)** ILP output, **(b)** output of Algorithm 1.

equal to an optimal solution. (Notice that classical snap-rounding also yields the same result on this graph.) The same observations can be made on the graph of Figure 3.12.

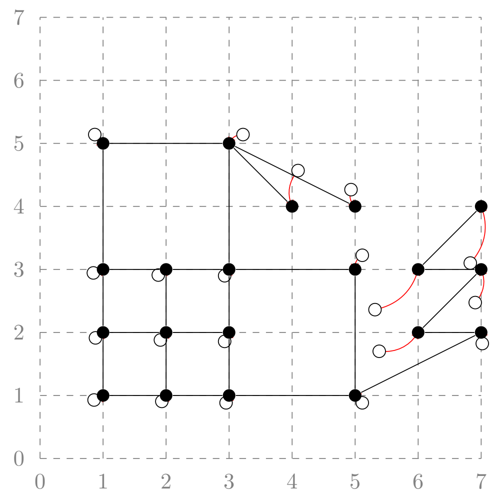
The result of classical snap-rounding algorithms on the graph found in Figure 3.15 would not be topologically safe. Concerning induced movement, both the ILP and Algorithm 1 give the same result. The ILP computation took more than 20 seconds, while our proof-of-concept implementation of Algorithm 1 finished in under one second (and implementing it more carefully will certainly reduce runtime).

Differences on output can be noticed comparing the upper-right corners of Figure 4.2 **(a)** and **(b)**. The output of Algorithm 1 is completely rounded and topologically safe. However, locally optimal decisions based on a fixed order of vertices can lead to non-optimal global results (as one would expect, considering the problem is $\mathcal{NP}$-hard). This becomes even more apparent comparing the right parts of the drawings in Figure 4.3.

We conclude this section with two examples that show cases where Algorithm 1 fails. Consider Figure 4.4 **(a)**. The unrounded vertex is adjacent to the outer face, but the graphs special structure prohibits rounding it to any grid point as non of them is visible to all neighbors. In Figure 4.4 **(b)**, two vertices are inside the same face that only contains only one integer grid point. Locally optimal decisions prevent the extension of the face that would be needed to round both inside vertices in a topologically safe way.

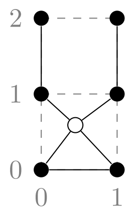


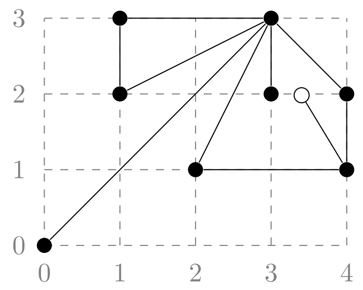
(a)



(b)

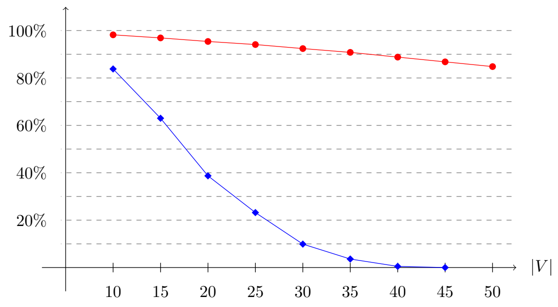
Figure 4.3.: Comparison on Graph 7 (Figure 3.16). **(a)** ILP output, **(b)** output of Algorithm 1.





(a) **(b)**

Figure 4.4.: Graphs Algorithm 1 fails on. Unrounded vertices depicted in white. **(a)** Output of Algorithm 1 with input graph from Figure 3.14. **(b)** Output of Algorithm 1 with input graph from Figure 3.10.



	$\lvert V\rvert =$	10	15	20	25	30	35	40	45	50	60	70	80	90	100
●	% avg.	98.2	96.9	95.4	94.1	92.4	90.8	88.8	86.8	84.8	80.6	76.3	72.3	68.1	64.4
◆	# of 100%	838	630	387	232	99	36	5	0	0	0	0	0	0	0

Figure 4.5.: Success rate of Algorithm 1 on the $[0, 9] \times [0, 9]$ integer grid (100 possible vertex locations). Measurements gathered over 1000 runs on random graphs each. The row with "% avg." gives the average percentage of vertices that were rounded successfully. The row with "# of 100%" gives the number of graphs that have completely been rounded.

4.3. Evaluation

As discussed above, examples where Algorithm 1 fails to round every vertex of a graph are easy to find. This however raises the questions of how common these examples are and how many vertices of a graph we can expect to be rounded by Algorithm 1. Reconsider the GIS applications that gave the motivation to work on TOPOLOGIALLY SAFE SNAPPING. Efficiently reducing the precision requirement of some vertices is beneficial concerning memory consumption, even if other vertices remain unrounded.

We implemented a simple generator for random planar graphs. Given a grid size and a number of vertices, we use *Delaunay Triangulations* of random point sets of given size on the given grid to generate connected planar graphs. To characterize these graphs, we consider a measure we call *density*. The density d of a graph $G = (V, E)$ is the ratio of vertices to be rounded and the total number of grid points inside the bounding box – density $d(G) = \frac{|V|}{(X_{\max}+1)\cdot(Y_{\max}+1)}$. Three randomly generated graphs and the corresponding output of Algorithm 1 can be found in Figure 4.6.

For taking measurements, we choose the $[0, 9] \times [0, 9]$ integer grid (100 grid points). On this grid, we did construct sets of 1000 random graphs, each set with density between 0.1 and 1. Every graph of these sets then has been rounded using Algorithm 1. For the outputs, we did consider the percentage of rounded vertices and if the graph was completely rounded. The data gathered in this process is listed and illustrated in Figure 4.5. Examining the findings, two things are to be noticed. As density goes up, the number of graphs that could completely be rounded drops to zero rapidly. However, the percentage of successfully rounded vertices decreases only linearly. Even at density 1 (one vertex per grid point), an average of 64.4% of all vertices could be rounded to integer coordinates. This demonstrates the usefulness of Algorithm 1 for compressing coordinate precision on some vertices of a given planar graph.

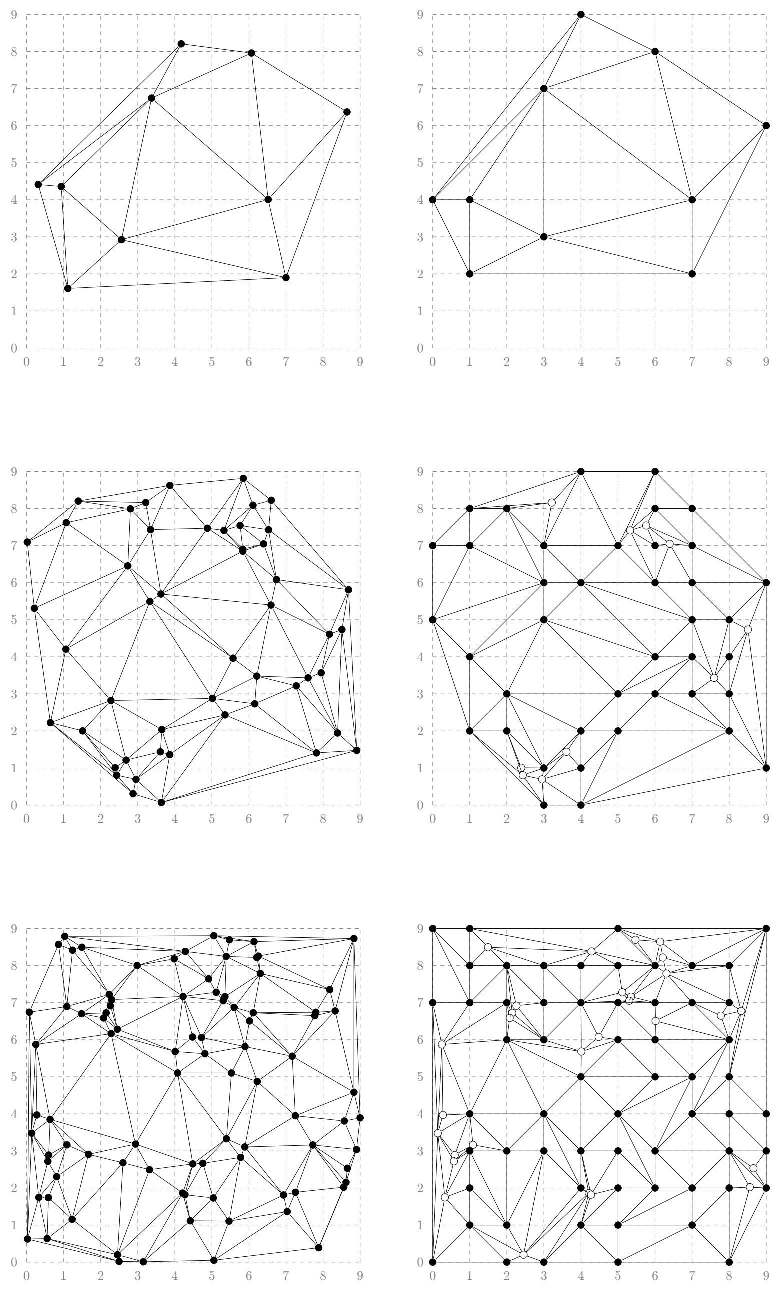


Figure 4.6.: Random graphs of varied density: **(top row)** random graph with density 0.1 and 100% success; **(middle row)** random graph with density 0.5 and 80% success; **(bottom row)** random graph with density 0.9 and 67.8% success. **(left column)** original graph; **(right column)** output of Algorithm 1 with unrounded vertices in white.

Reconsider graphs that Algorithm 1 did round completely. These roundings can be compared to roundings produced by the ILP approach of Chapter 3. To do so, we randomly constructed graphs on the $[0, 4] \times [0, 3]$ integer grid (20 grid points) with density 0.35 (7 vertices per graph) until we had a total of 150 graphs that could completely be rounded by Algorithm 1. We found these instances to be dense enough for the heuristic to be challenging and small enough for the ILP to actually compute a solution in reasonable time. On an overall average, we did observe that for a graph total rounding cost of the heuristic solution was 3.68% higher than that of the ILP solution. On 104 instances, ILP and heuristic did produce a solution of equal costs. (Which is unsurprising, as low-density graphs tend to be easy.) Considering the 46 graphs that did show a difference, on average, the solutions of Algorithm 1 where 12.02% more expensive (with a maximum of 61.5% difference on one instance).

5. Conclusion

In this thesis, we introduced and investigated the problem of Topologially Safe Snapping – rounding graphs to an underlying integer grid while ensuring topological equivalence. We have shown that it is in fact $\mathcal{NP}$-hard and also provided similar results for related variants. To solve Topologially Safe Snapping, we decided to use integer linear programming. Our model in use has been described in detail and its usefulness and limitations have been evaluated empirically. However, while our set of constraints works as intended, there may be more sophisticated formulations that could decrease overall ILP runtime. One could also consider using advanced speedup methods on our model. We have shown in Section 2.3 that we can not provide a fully polynomial-time approximation scheme and no constant additive approximation, the question about general approximability of Topologially Safe Snapping remains open.

One motivation for working on this problem was using it to produce topologically correct maps. The experiments we ran in Section 3.3 show that our model is not suitable to do so on reasonably sized data sets. Still it may be a possible tool for any cartographer. For maps where large portions can easily be represented on an underlying grid, we propose the following workflow: extract the difficult but hopefully small parts and use the ILP to solve them locally optimal. The obtained result may then carefully be integrated into the drawing. We do not have an automated procedure to perform this workflow.

Another motivation was reducing the amount of data needed to store planar graphs by reducing coordinate precision. To do so, we designed a rounding heuristic based on the graphs faces. While not able to round every vertex of any planar graph, we did use random planar graphs to demonstrate the capability to reduce required coordinate precision for a considerable amount of vertices. More sophisticated algorithms may be able to increase the expected percentage of rounded vertices.

To our knowledge, we are the first to consider Topologially Safe Snapping. This yields several topics to be considered. Future work could be dedicated to designing better heuristics. On the practical side, implementing a user-driven tool to realize the workflow proposed above may be of interest to anyone working with maps. Other practical applications for topologically safe rounding are to be found. For us, the most interesting topic would be providing (or disproving the existence of) an approximation algorithm.

Bibliography

[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 *Proceedings of the 21st International Symposium on Graph Drawing*, volume 8242 of *Lecture Notes in Computer Science*, pages 460–471. Springer, 2013.

[BO+79] Jon L. Bentley, Thomas Ottmann, et al. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, 100(9):643–647, 1979.

[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.

[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.

[CN98] Marek Chrobak and Shin-Ichi Nakano. Minimum-width grid drawings of plane graphs. *Computational Geometry*, 11(1):29–54, 1998.

[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.

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry*. Springer, 2000.

[dFPP90] Hubert de Fraysseix, János Pach, and Richard Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[DLT83] Danny Dolev, Frank Thomson Leighton, and Howard Trickey. Planar Embedding of Planar Graphs. Technical report, DTIC Document, 1983.

[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.

[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.

[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.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison Wesley, 2nd edition, 1994.

[GM98] Leonidas J. Guibas and David H. Marimont. Rounding Arrangements Dynamically. *International Journal of Computational Geometry & Applications*, 8(02):157–178, 1998.

[GY86] Daniel H. Greene and F. Frances Yao. Finite-resolution computational geometry. In *27th Annual Symposium on Foundations of Computer Science*, pages 143–152. IEEE, 1986.

[Her13] John Hershberger. Stable snap rounding. *Computational Geometry*, 46(4):403–416, 2013.

[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.

[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.

[Lic82] David Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[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 (to appear)*. Springer, 2016.

[MS97] Bruce A. McCarl and Thomas H. Spreen. *Applied mathematical programming using algebraic systems*. Texas A&M University, 1997.

[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.

[Sch90] Walter Schnyder. Embedding Planar Graphs on the Grid. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 138–148, 1990.

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

[vLvL11] Erik Jan van Leeuwen and Jan van Leeuwen. Structure of polynomial-time approximation. *Theory of Computing Systems*, 50(4):641–674, 2011.

[Wol07] Alexander Wolff. Drawing Subway Maps: A Survey. *Informatik – Forschung & Entwicklung*, 22(1):23–44, 2007.

A. Appendix

A.1. OPL Code for ILP

Here we give the complete code of an implementation of the model presented in Section 3.1 using the IBM ILOG CPLEX Optimization Studio OPL Language. A reference manual can be found here: OPL language reference[1] There also is an offline version of this file on the disk handed in with this thesis.

Full.mod

```
/*********************************************
 * OPL 12.6.2.0 Model
 * Author: andre
 * Creation Date: Feb 23, 2016 at 12:11:29 PM
 *********************************************/

{string} coords = ...; // X & Y
int vertexCount = ...; // no. of vertexCount
int edgeCount = ...; // no. of edgeCount
int xMax = ...;
int yMax = ...;
int m = maxl(xMax,yMax);

int LC_alpha = m * 2 + 1;
int LC_gamma = m * 2 + 1;
float minimumDistance = -1/(m + 1);

tuple IndexedDirection {
    key int id;
    float xPer;
    float yPer;
}

tuple Node {
    key int id;
    float X;
    float Y;
    {int} embedding;
}

sorted {float} slopes; //helper-set for creating ...
{IndexedDirection} orderedDirections;

{Node} nodes = ...;
{int} edges[1..edgeCount] = ...;

dvar int round[1..vertexCount][coords] in 0..vertexCount;
```

[1] http://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.2/ilog.odms.studio.help/pdf/opl_langref.pdf

```
dvar int inDirectionOf[n1 in 1..vertexCount][n2 in 1..vertexCount][d in
    orderedDirections] in 0..1; // aka.: alpha
dvar int beta[n in 1..vertexCount][1..vertexCount] in 0..1;
dvar int gamma[1..edgeCount][d in orderedDirections][1..edgeCount] in 0..1;

/* ********************************.
 * START PREPROCESSING BLOCK
 * ********************************/
execute {
  for (var den = 1 ; den <= m ; den++) {
    for (var num = 0 ; num <= den ; num++) {
      var slope = num/den;
      slopes.add(slope);
    }
  }

  var id = 1;
  for (var i = 0 ; i < slopes.size ; i++){ //right-top
    var s = Opl.item(slopes, i);
    orderedDirections.add(id++, 1, s);
  }
  for (var i = slopes.size-2 ; i >= 0 ; i--){ //top-right
    var s = Opl.item(slopes, i);
    orderedDirections.add(id++, s, 1);
  }
  for (var i = 1 ; i < slopes.size ; i++){ //top-left
    var s = Opl.item(slopes, i);
    orderedDirections.add(id++, -s, 1);
  }
  for (var i = slopes.size-2 ; i >= 0 ; i--){ //left-top
    var s = Opl.item(slopes, i);
    orderedDirections.add(id++, -1, s);
  }
  for (var i = 1 ; i < slopes.size ; i++){ //left-bottom
    var s = Opl.item(slopes, i);
    orderedDirections.add(id++, -1, -s);
  }
  for (var i = slopes.size-2 ; i >= 0 ; i--){ //bottom-left
    var s = Opl.item(slopes, i);
    orderedDirections.add(id++, -s, -1);
  }
  for (var i = 1 ; i < slopes.size ; i++){ //bottom-right
    var s = Opl.item(slopes, i);
    orderedDirections.add(id++, s, -1);
  }
  for (var i = slopes.size-2 ; i >= 1 ; i--){ //right-bottom
    var s = Opl.item(slopes, i);
    orderedDirections.add(id++, 1, -s);
  }
}
/* ********************************
 * END PREPROCESSING BLOCK
 * ********************************/

//minimize for all vertices the movement in Manhattan-distance
minimize
  sum (v in nodes)
    (abs(round[v.id]["X"] - v.X) + abs(round[v.id]["Y"] - v.Y));

subject to {
    // for every pair of nodes, exactly one inDirectionOf variable has to be 1
    forall (v in 1..vertexCount)
      forall(w in item(nodes,v-1).embedding)
        sum (d in orderedDirections)
          inDirectionOf[v][w][d] == 1;

    forall (n1 in 1..vertexCount)
```

```
      forall (n2 in item(nodes,n1-1).embedding)
        forall (d in orderedDirections) {
          - (1 - inDirectionOf[n1][n2][d])*LC_alpha +
          round[n2]["X"]*d.yPer + round[n1]["Y"]*d.xPer - round[n1]["X"]*d.yPer
          <= round[n2]["Y"]*d.xPer;

            (1 - inDirectionOf[n1][n2][d])*LC_alpha +
          round[n2]["X"]*d.yPer + round[n1]["Y"]*d.xPer - round[n1]["X"]*d.yPer
          >= round[n2]["Y"]*d.xPer;

            (1 - inDirectionOf[n1][n2][d])*LC_alpha
          + (round[n2]["X"]-round[n1]["X"])*d.xPer
          + (round[n2]["Y"] - round[n1]["Y"])*d.yPer
          >= 0;
        }

    forall (n in nodes)
      forall (d in orderedDirections)
        inDirectionOf[n.id][n.id][d] == 0;

    forall (d in orderedDirections) {
      forall (v in nodes) {
        forall (a in v.embedding){
              inDirectionOf[v.id][a][d]
          <= (sum (dd in orderedDirections : dd.id > d.id )
               inDirectionOf[v.id][nextc(v.embedding,a)][dd])
          + beta[v.id][a];
        }
      }
    }

    // for all nodes with deg >= 2, the sum over all betas must be exactly one
    forall (n in nodes)
      sum (a in n.embedding)
        beta[n.id][a] == 1;

    //for incident edges, there should be no seperation, so gamma == 0
    forall (e1 in 1..edgeCount)
      forall (d in orderedDirections)
        forall (e2 in 1..edgeCount)
          if ( card(edges[e1] union edges[e2]) != 4 )
            gamma[e1][d][e2] == 0;

    //no nodes end on the same grid point
    forall (v in nodes)
      forall (w in nodes)
        if (v.id != w.id) // ignore v == w
          !((round[v.id]["X"] == round[w.id]["X"]) && (round[v.id]["Y"] ==
               round[w.id]["Y"]));

    // every pair of edgeCount has to be seperated at least once
    forall (e1 in 1..edgeCount)
      forall (e2 in 1..edgeCount)
        if ( card(edges[e1] union edges[e2]) == 4 ) // ignore gamma_d_e1_e1
          sum (d in orderedDirections)
            gamma[e1][d][e2]
          >= 1;

    // determine relative positions of edgeCount
    forall (d in orderedDirections)
      forall (e1 in 1..edgeCount)
        forall (e2 in 1..edgeCount)
          forall (i in 0..1)
            forall (j in 0..1)
              if ( card(edges[e1] union edges[e2]) == 4 ){
                (d.xPer * round[item(edges[e2],i)]["X"] + d.yPer *
                    round[item(edges[e2],i)]["Y"])
                - (d.xPer * round[item(edges[e1],j)]["X"] + d.yPer *
                    round[item(edges[e1],j)]["Y"])
                - (1 - gamma[e1][d][e2])*LC_gamma
                <= minimumDistance;
              }
}

// output designed for TikZ-code generation
execute {
  for (var i = 1; i <= vertexCount ; i++) {
    writeln("\\draw [move] ("+round[i]["X"] + ","+round[i]["Y"] +
        ")node[blacknode](o"+i+"){} to ("+Opl.item(nodes, i-1).X + ","+Opl.item(nodes,
        i-1).Y + ")node[whitenode](i"+i+"){};");
  }
  for (var j = 1; j <= edgeCount ; j++) {
    writeln("\\draw [dashed]
        (i"+Opl.item(edges[j],0)+")--(i"+Opl.item(edges[j],1)+");\\draw
        (o"+Opl.item(edges[j],0)+")--(o"+Opl.item(edges[j],1)+");");
  }
}
```

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen Hilfsmittel und Quellen als die angegebenen benutzt habe. Weiterhin versichere ich, die Arbeit weder bisher noch gleichzeitig einer anderen Prüfungsbehörde vorgelegt zu haben.

Würzburg, den ____________________, ____________________________________

(Andre Löffler)