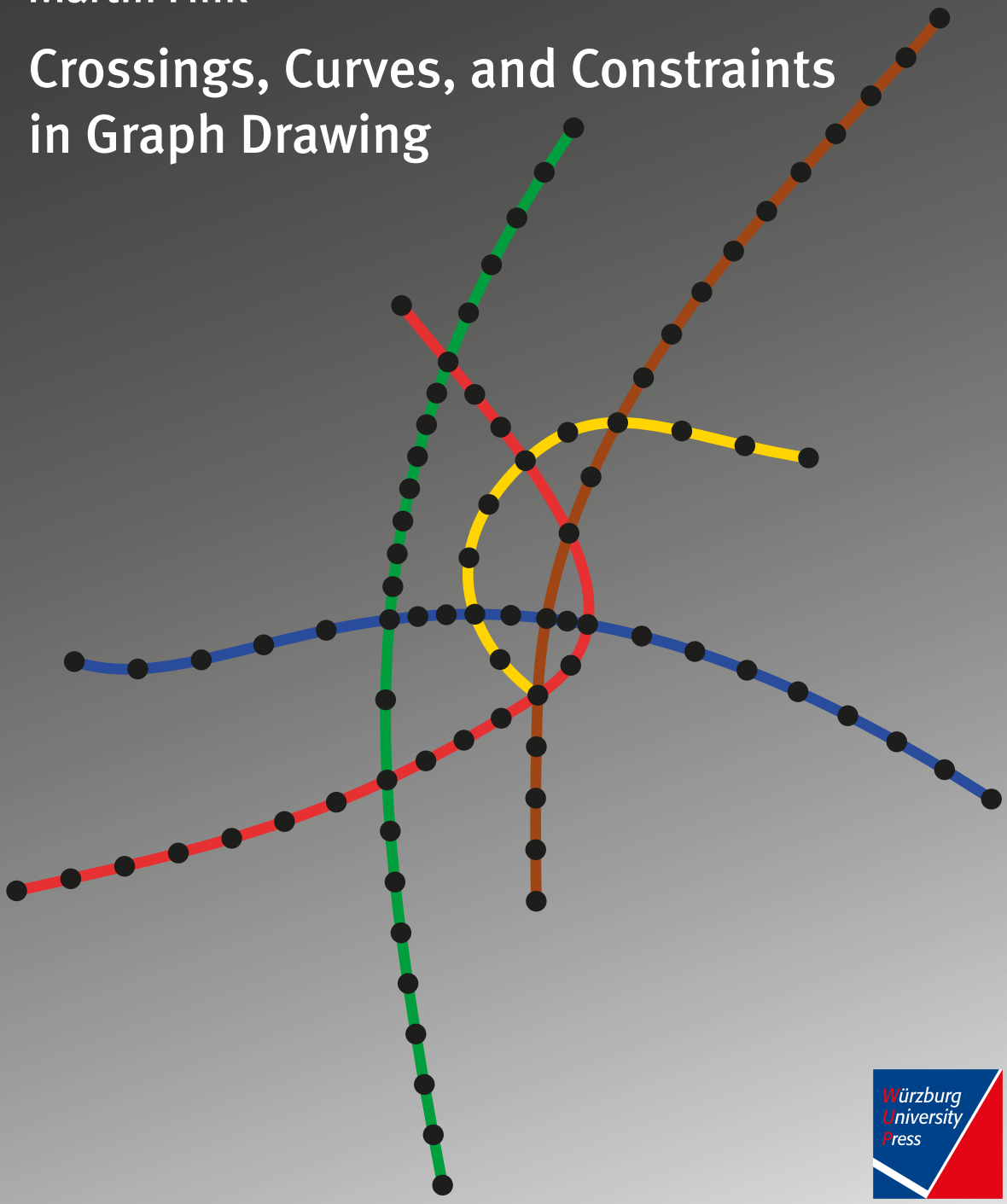
Martin Fink

Crossings, Curves, and Constraints in Graph Drawing



Würzburg University Press

Martin Fink

Crossings, Curves, and Constraints in Graph Drawing

Martin Fink

Crossings, Curves, and Constraints in Graph Drawing

Würzburg University Press

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2014
Gutachter: Prof. Dr. Alexander Wolff, Prof. Dr. Michael Kaufmann

Impressum

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

Print on Demand 2014

ISBN 978-3-95826-002-3 (print)
ISBN 978-3-95826-003-0 (online)
URN urn:nbn:de:bvb:20-opus-98235

Preface

In this thesis, Martin Fink deals with the visualization of graphs. A graph consists of a set of objects called *vertices* and a set of connections between pairs of objects called *edges*. Graphs are used to model networks, both abstract (such as social networks) and physical (such as river or subway networks). In order to help users to understand the structure of networks and their underlying graphs, the field of Graph Drawing studies problems related to the layout of graphs. The central question is how to best draw the vertices and edges of a given graph in the plane (or some other space) such that certain features of the graph are revealed. Typically, vertices are drawn as points, little disks, or squares, and edges are drawn as curves (so-called Jordan arcs) that connect the corresponding vertices. Such drawings are called *node-link* representations.

In Martin Fink's thesis, the three words that make the title reappear many times: *crossings*, *curves*, and *constraints*. Clearly, in a node-link representation, crossings cannot be avoided—except if the given graph is planar, which holds only for a small subset of graphs. Hence, it is fundamental for Graph Drawing algorithms to deal with crossings. Martin Fink's thesis contains some important contributions to the treatment of crossings. Traditionally, edges have mostly been drawn as straight-line edges, polygonal, or rectilinear paths; these are called *(edge) drawing styles*. Only very recently, other drawing styles such as (Bézier) curves have appeared in the literature. For two specific application areas, Martin Fink is the first to use such curves. Last but not least, constraints have long been used in Graph Drawing to model application-driven requirements beyond the drawing style of edges. In this thesis, constraints usually concern the placement of vertices; a vertex must either be placed at a specific position, at one within a given set of positions or simply not "too far" from a given position.

The thesis consists of three parts, which can be read independently of each other. It pays, however, to check the preliminaries in Chapter 2, where the author gives a gentle introduction into graphs and Graph Drawing, and establishes some basic complexity-theoretic concepts.

In part I, Martin Fink considers the problem of drawing *metro maps*, an application of Graph Drawing that has received considerable attention over the last few years. The author presents the first algorithm that draws the metro lines of a metro map using Bézier curves. The algorithm is based on a *spring embedder*, that is, an iterative procedure that simulates a physical system with attracting and repelling forces. Martin Fink also considers a very different, more theoretical type of problem related to metro maps: how to avoid crossings between different metro lines. In particular, the author introduces a new way to define and count crossings, which he calls *block crossings*, and presents provably good algorithms. In my opinion, this new approach to an old problem in Graph Drawing makes for a very solid contribution.

Part II deals with *point-set embeddability* of graphs. In this setting, which has been studied for more than twenty years, one is not only given a graph to be drawn, but also a set of points, and the task is to place each vertex on one of the given point such that the graph can be drawn in a specific drawing style. Martin Fink combines this old problem with new constraints: he

allows crossings between edges of the graph, but insists on large crossing angles (that is right angles or near-right angles).

In the final part of the thesis, part III, the author investigates two versions of the *boundary labeling* problem. In this problem, one is given a map with some sites and the objective is to place the labels outside the map and connect them with the sites they label. Usually the connections must not intersect each other. Such a label placement is advantageous if the map background is important and should not be superimposed by labels. In the first version of the problem, Martin Fink uses, among others, Bézier curves in order to label sites in focus-and-context maps. In the second version of the problem, labels must be connected to several sites (which are of the same type, for example, cafés).

In his thesis, Martin Fink presents some profound research, both practical and theoretical. He does an excellent job in motivating his problems and in explaining and illustrating his solutions, some of which are technically quite involved. I enjoyed reading this thesis, and I wish it will have many readers.

Alexander Wolff
Chair I – Efficient Algorithms and Knowledge-Based Systems
Institute of Computer Science
University of Würzburg

Contents

Chapter 1

Introduction

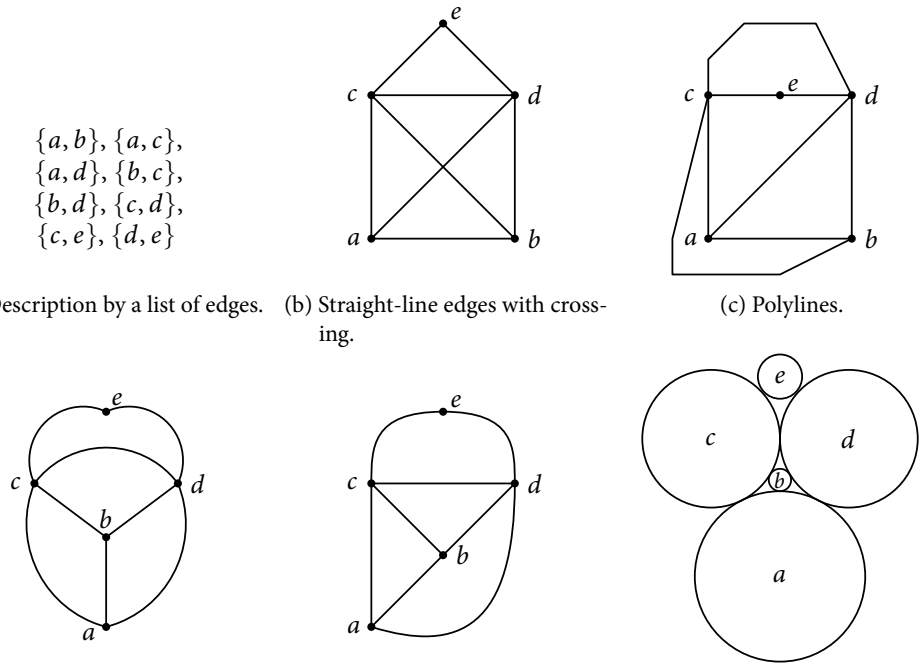
Graphs are one of the most frequently used tools for modeling data. In computer science, graphs are used both for solving problems—with the help of graph-based algorithms—and for making information accessible to users of applications. This is possible because many problems and many types of information can be modeled using graphs. Any graph consists of two sets: a set of well-distinguishable objects, called *vertices*, and a set of connections, called *edges*. Any edge of a graph models a relation connecting one *vertex* of the graph to another vertex. Hence, the simplest example of data that can be modeled by a graph is a physical network.

For example, the street network of a city can be modeled by making any junction a vertex. An edge connects two vertices if there is a single street segment that connects the two junctions corresponding to the vertices; this street segment must not be split by a junction in its middle. Similarly, a wired computer network can be modeled as a graph by making the devices (computers, switches, etc.) vertices of the graph while wires connecting two devices become edges. Another example are transportation networks such as the metro network of a city, for example, the *London Underground* or the *Métro de Paris*. In this setting, the stations of the network become vertices; two stations are connected by an edge if there is a direct connection between these stations that does not have an intermediate stop.

Graph Drawing. If a graph is just used internally by some software for solving a problem with the help of a graph algorithm, it is not important for the user how the graph is represented. As the user will only see the solution of the problem, the graph representation best suitable for the algorithm may be used. However, if we use a graph for showing information to the user, the right *visualization* is crucial. Presenting lists of vertices and edges as text to the user only works for very small graphs. Even for such a small graph, a *drawing* is usually much easier to read and interpret; see Figure 1.1. Therefore, and due to the importance of graphs for modeling data, there is a need for algorithms that compute good visualizations of graphs.

With a good drawing, the human ability of quick visual perception of structures enables a user to intuitively understand the structure of a graph. Also certain tasks, like finding a shortest path that connects two vertices of the graph, can be answered much faster and more easily with a good visualization. The research area of *graph drawing* is primarily concerned with the development of algorithms for drawing graphs.

In a drawing of a graph, the vertices are usually represented by points in the plane (or by small disks) and the edges are drawn as curves connecting vertices; see Figure 1.1b and Figure 1.1d. Such drawings are sometimes also called *node–link diagrams*. Alternative methods for visually representing graphs exist. An example is shown in Figure 1.1f where the vertices are drawn as



(a) Description by a list of edges.

(b) Straight-line edges with crossing.

(c) Polylines.

(d) Planar drawing with circular arcs.

(e) Bézier curves.

(f) Contact of disks.

Figure 1.1: An example of a graph shown as a list of edges compared to drawings of the same graph in different styles.

disks in the plane and an edge is represented by a contact of the disks of the corresponding vertices. However, in this thesis, we will always use the classic—and most commonly used—drawings of graphs as node–link diagrams.

There are several styles for drawing the edges of a graph. An edge may be drawn as a straight-line segment (see Figure 1.1b), a polyline (see Figure 1.1c), a circular arc (see Figure 1.1d), a parametric curve (see Figure 1.1e), or in other styles. Furthermore, we can distinguish drawings in which edges cross (as in Figure 1.1b) and drawings which are crossing-free; drawings without edge crossings are called *planar*. Unfortunately, only a relatively small, but well-studied, subset of all graphs can be drawn in a planar way—the *planar graphs*.

Constraints. In most applications in which a graph is used for displaying information and, therefore, needs to be drawn, one does not simply want to present some drawing of the graph, but a drawing that follows certain *constraints*. As a first example, one may insist on a planar drawing. This is a *hard constraint*, which means that we are only interested in drawings that satisfy this constraint. Such hard constraints determine the drawing *style* or drawing *convention*. We can also think of the corresponding *soft constraint* or *optimization* constraint. An optimization constraint aims at optimizing the *aesthetics* of drawings. In our example, we could allow crossings

between edges but try to find a drawing with as few crossings as possible, that is, we want to minimize the number of crossings. This is a first example of an NP-hard problem in graph drawing, as Garey and Johnson showed [GJ83]; there are many more graph drawing problems that are NP-hard as we will see. Apart from planarity, also the restriction to a certain style for drawing the edges is a very common hard constraint in graph drawing. In the following, we will see more constraints that are used in this thesis.

Vertex Positions. If one wants to draw a graph in order to show additional information on top of a previous visualization, the position of vertices may already be fixed. As an example, one may want to show the flight connections between different airports or trading connections between different cities on top of map. Under this constraint, a drawing with straight-line edges is already fixed. If, however, we have more freedom when drawing the edges, the drawing can be optimized, for instance, by trying to avoid crossings, or by maximizing the angles occurring between crossing edges.

Also the optimization variant of this constraint can occur; that is, we have a *desired position* for each vertex. We do not require that the vertex is drawn exactly at the desired position, but we prefer drawings in which it is drawn close to this position. As a possible objective, we want to find a drawing that minimizes the sum of the distances between the vertices and their desired positions. We will see this optimization criterion for creating drawings of metro networks in Chapter 3: in such a drawing, the user must be able to find the location of a station of the network on the map and, therefore, the distance to the geographic location should be small. However, deviations from the desired positions are allowed if they help to improve the readability of the drawing.

In a variant of the previous hard constraint, it is also possible that we want to draw a graph such that only a prescribed set of positions is used for placing vertices. However, the exact position for each individual vertex is not given, which means that we have to map any vertex to a distinct point of the given set, and then find a feasible drawing. The problem of finding a drawing under these constraints is known as *point-set embeddability* which is covered in Part II. Very special constraints for the vertex positions occur in *boundary labeling* (handled in Part III), where the exact position of only a part of the vertices is prescribed.

Edges and Crossings. As already mentioned, there are several possible styles for drawing the edges of a graph. A natural constraint is to restrict the drawings to use only a specific style for edges. There are, however, several degrees of such restrictions. For instance, if the edges have to be drawn as polylines, one may further restrict that any edge consists only of at most three straight-line segments, or that all straight-line segments have to be axis-parallel, that is, horizontal or vertical; the drawing style resulting from the latter constraint is called the *orthogonal drawing style*; see Figure 1.2 for an example. We will consider orthogonal drawings in Chapter 7.

A very common constraint is to insist on a planar drawing. Since such a drawing does not always exist, other constraints for the crossings can also make sense. As already mentioned, the soft constraint of minimizing the number of crossings has been investigated and is NP-hard. Also hard constraints for the numbers of crossings have been considered. For example, the class

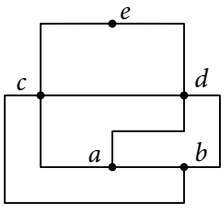
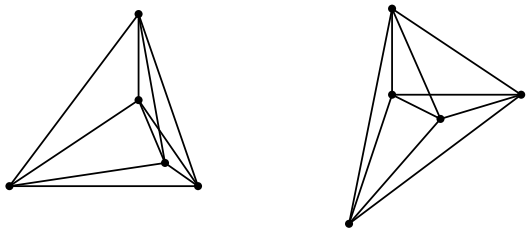


Figure 1.2: An orthogonal drawing.



(a) Small crossing angle. (b) Larger crossing angle.

Figure 1.3: Comparison between crossing angles.

of 1-planar graphs is a generalization of planar graphs; a graph is 1-planar, if it can be drawn such that any edge is involved in at most one crossing.

The angle of crossings has also been considered as a constraint. If two edges cross with a very large angle, it is much easier to distinguish between the edges than if the crossing angle is small. There have been studies indicating that drawings of graphs in which all crossing angles are large are almost as readable as crossing-free drawings [HHE08]; see Figure 1.3. Formulated as a hard constraint, we can insist that any crossing angle is at least α for a constant angle α close to 90°. We will use constraints of this type in Chapter 6.

We will also consider another crossing problem in which not crossings between edges of the graph, but crossings between paths drawn on top of the graph are considered. This problem occurs when visualizing transportation networks like metro networks: Often several transportation lines of such a network run in parallel, that is, they share edges of the underlying graph. At the end of a parallel subpath, however, the lines split, which can make crossings between them necessary. Hence, we try to visualize the lines in such a way that the number of crossings is minimized. Crossings between metro lines will be considered in Chapters 4 and 5.

Curves. A special drawing style for edges are smooth curves such as circular arcs. Drawing edges using smooth curves allows us to have more flexibility for routing the edges, compared to straight-line edges, while we can still avoid having sharp bends, which happens if we use polylines. In Chapters 3 and 8 we will use a special class of parametric curves called *Bézier curves*, or, more formally, curves in Bézier representation. They allow us to choose the direction in which the edge leaves an incident vertex. For drawing metro maps, we will use this in order to ensure that a metro line does not have sharp bends although it is composed of several edges of the network.

Outline of the Thesis

This thesis consists of three parts, each dealing with a different area of graph drawing. Part I is devoted to drawing *metro maps*, that is, visualizations of metro networks. Part II covers *point-set embeddability* problems; in this setting, the positions of vertices are restricted to a prescribed set of points. Finally, Part III deals with *boundary labeling*; in boundary labeling, interesting

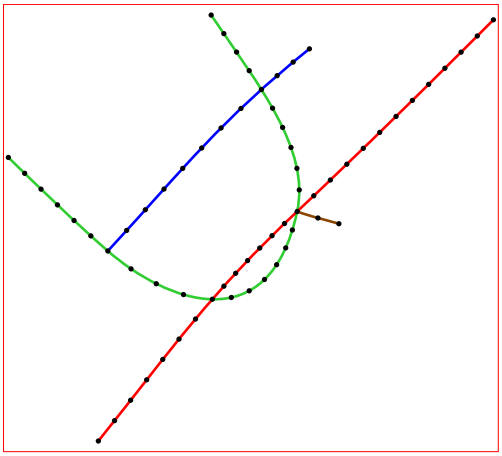


Figure 1.4: A metro network drawn using Bézier curves.

sites on a map are labeled by connecting the label text to the site via an edge. Before starting with the three main parts, a brief introduction into terminology, techniques, and related topics is given in Chapter 2.

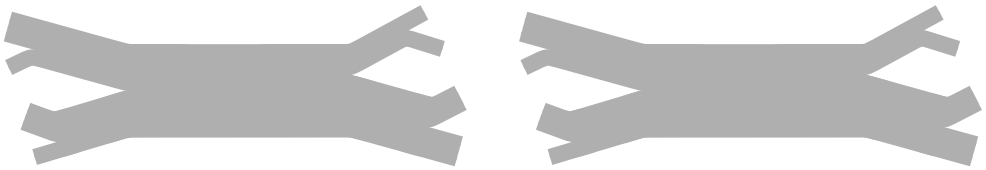
Part I: Metro Maps

In the first part of the thesis, we consider the problem of drawing metro maps. More specifically, we consider two subproblems independently. We first want to create a drawing of the graph modeling the network of metro stations. In a second step, the different metro lines running in this network should be visualized on top of the drawing.

Chapter 3: Drawing Metro Maps using Bézier Curves

Chapter 3 is devoted to drawing the graph that represents a metro network. Traditionally, most metro maps are drawn in the *octilinear* drawing style. In octilinear drawings of graphs, edges are drawn as polylines consisting only of horizontal, vertical, and diagonal segments (at an angle of 45°). However, a user study of Roberts et al. [RNL+13] shows that the planning speed for trips in the network can be increased by using curvy metro maps in which edges are drawn as smooth curves. We present an algorithm for drawing metro maps using curves; more specifically, our algorithm uses *Bézier curves* for drawing the edges; see Figure 1.4 for an example. We try to optimize the drawing by having a small visual complexity. This is done by minimizing the number of single curves used in the drawing.

This chapter is based on joint work with Herman Haverkort, Martin Nöllenburg, Maxwell Roberts, Julian Schuhmann, and Alexander Wolff [FHN+13].



(a) 12 individual crossings. (b) 3 block crossings.

Figure 1.5: Metro lines drawn on top of a part of a metro network with different distribution of the crossings.

Chapter 4: Metro-Line Crossing Minimization

In Chapter 4, we turn to the problem of visualizing the *metro lines*. The method presented in the previous chapter, that is, in Chapter 3, is—similar to other algorithms for drawing metro maps—focused on drawing the underlying graph of the metro network. An essential feature of metro maps is, though, the visualization of the metro lines on top of the network (see Figure 1.5a for an example): only the visualization of the metro lines makes it possible to plan trips in the metro network. Hence, we want to make it easy for users to follow a metro line on the map, so that they can see which stations are served by the line. The most important optimization criterion for making the lines easy to follow is to have as few crossings between lines as possible; the problem of minimizing the number of crossings is known as *metro-line crossing minimization*.

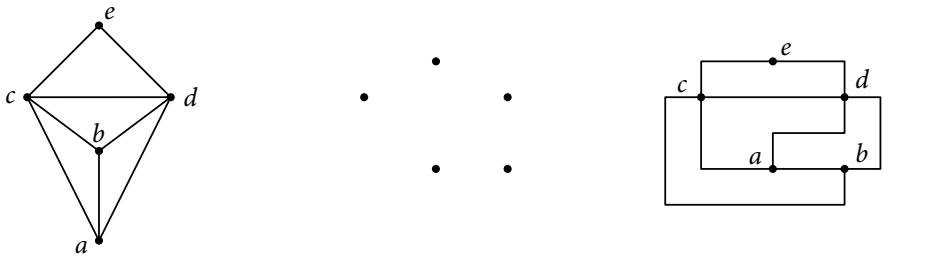
In this chapter, we show that the most general version of metro-line crossing minimization is NP-hard. For a well-known version of the problem, we present the first approximation algorithm, and we develop an efficient algorithm for a restricted set of networks. Finally, we develop a fixed-parameter algorithm for metro networks whose graph is a tree.

This chapter is based on joint work with Sergey Pupyrev [FP13a] and—for Section 4.5—on joint work with Sergey Pupyrev and Alexander Wolff (not yet published).

Chapter 5: Ordering Metro Lines by Block Crossings

In Chapter 5, we introduce and study a variant of metro-line crossing minimization. We improve visualizations of metro networks by not only minimizing the number of crossings between metro lines, but also taking the distribution of the crossings into account. Our idea is the following (see Figure 1.5b for an illustration). Suppose, on some edge of the network there are two contiguous *blocks* of lines running in parallel. If each line of the first block crosses each line of the second block, we can arrange all these crossings in one *block crossing*, in which the whole blocks change their order, while the lines within a block stay parallel. This improves the readability compared to a random distribution of the individual crossings; see Figure 1.5.

We thus study the problem of minimizing the number of block crossings between metro lines. We show that also block crossing minimization is NP-hard—even for networks of small degree and edges that have not more than eleven metro lines passing through them. Furthermore, we present a heuristic that finds a solution on general graphs with a bounded number of block crossings. For some restricted classes of networks, we present approximation algorithms. Finally,



(a) The graph $G = (V, E)$. (b) The point set S. (c) A planar orthogonal embedding of G on S.

Figure 1.6: A point-set embedding of a graph on a point set with no prescribed vertex-point mapping

we adjust the fixed-parameter algorithm for ordinary metro-line crossing minimization to minimize the number of block crossings.

This chapter is based on joint work with Sergey Pupyrev [FP13b] and—for Section 5.6—on joint work with Sergey Pupyrev and Alexander Wolff (not yet published).

Part II: Point-Set Embeddability

In the second part of the thesis, we study *point-set embeddability* problems. In point-set embeddability, the task is to draw a graph using a set of prescribed positions for placing the vertices; see Figure 1.6 for an example. There are two main subsettings depending on whether the *mapping* of the vertices to the points in the given set is prescribed or can be chosen when finding a drawing. We study point-set embeddability problems in which edges must be drawn as polylines with a small number of bends. We are especially interested in nonplanar drawings. However, we allow only crossings with large crossing angles.

Chapter 6: Point-Set Embeddability and Large Crossing Angles

In Chapter 6, we introduce and study point-set embeddability with large crossing angles. We first show that the problem is NP-hard if the edges must be drawn as straight-line segments. We, hence, focus on the versions with polyline edges and prescribed vertex-point mapping. We show how to embed any graph on any point set with three bends per edge and right-angle crossings. Furthermore, we present embeddings of any graph on any point set with one or two bends per edge such that all crossings have an angle of at least $90° - \varepsilon$, where we can choose $\varepsilon > 0$ arbitrarily small. For all embeddings that we construct, we also analyze the area requirement.

This chapter is based on joint work with Jan-Henrik Haunert, Tamara Mchedlidze, Joachim Spoerhase, and Alexander Wolff [FHM+12].

Chapter 7: Orthogonal Point-Set Embeddability on the Grid

In Chapter 7, we study point-set embeddability in the orthogonal drawing style; that is, edges must be drawn as polylines consisting only of horizontal and vertical segments. Furthermore,

we insist that vertices and bends of edges must be placed on grid points, that is, points with integer coordinates. This constraint is natural for orthogonal drawings; any orthogonal drawing can be converted by moving vertices and bends until all of them lie on integer positions.

We show that, under these constraints, point-set embeddability without prescribed vertex-point mapping is NP-hard; this is independent of the number of bends that we allow per edge. Surprisingly, the problem remains NP-hard even if we allow crossings. For the problem variants in which we are given the precise position of each vertex, we show that point-set embeddability can be solved efficiently for up to one bend per edge. For two or three bends, the nonplanar version is NP-hard. For the planar version we show hardness if the number of bends is not restricted.

This chapter is based on joint work with Jan-Henrik Haunert, Tamara Mchedlidze, Joachim Spoerhase, and Alexander Wolff [FHM+12] and—for the hardness results—on joint work with Alexander Wolff (not yet published).

Part III: Boundary Labeling

The third part of the thesis is devoted to a style of labeling maps, called *boundary labeling*. In boundary labeling problems, one wants to label interesting sites on the map; that is, the sites have to be annotated by a label text, which often is the name of the site. In normal map labeling, the labels are placed on the map next to the corresponding sites. However, this approach has two disadvantages. First, in regions with too many sites, the labels can overlap. Second, parts of the map are hidden behind the labels. To overcome these problems, boundary labeling can be used: The labels are moved away from their sites to the boundary of the map or of a *focus region*. For showing the correspondence between a label and a site, the two objects are then connected via an edge, called leader; see Figure 1.7. Hence, boundary labeling is a special graph drawing problem (related to point-set embeddability), in which the position of a part of the vertices (the sites) is completely fixed, while the possible positions for the remaining vertices (the labels) are restricted to the boundary of the map or of the focus region.

Chapter 8: Algorithms for Labeling Focus Regions

In Chapter 8, we study boundary labeling with circular focus regions. We use two main styles depending on the placement of the labels. In the first style, we use conventional horizontal text labels which are connected to the sites via straight-line leaders. In the second style, we use radial labels whose text runs radially away from the center of the focus region. For both styles we also present methods that improve the solutions by replacing the straight-line leaders by Bézier curves that enter the label in the direction of the text.

We present algorithms that label as many sites as possible and ensure that the leaders are crossing free and the labels do not overlap. We extend these results to the version with preferences for labeling the sites. We also present approaches for selecting a subset of the sites that can be labeled simultaneously; as an additional requirement, we want the labeled sites to represent the spatial distribution of sites in the focus region

This chapter is based on joint work with Jan-Henrik Haunert, André Schulz, Joachim Spoerhase, and Alexander Wolff [FHS+12].

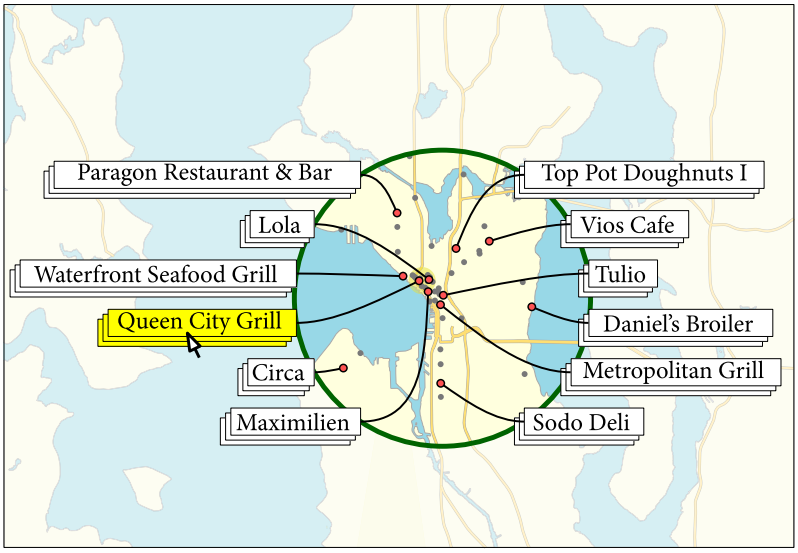


Figure 1.7: Boundary Labeling used for showing positions of restaurants in Seattle.

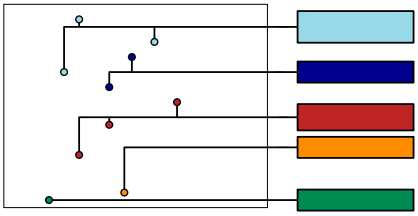


Figure 1.8: Many-to-one boundary labeling with backbones.

Chapter 9: Many-to-One Boundary Labeling with Backbones

In the final chapter, we study many-to-one boundary labeling with backbones. In this scenario of boundary labeling, the sites are not annotated by their respective name but by their class. For example, when labeling restaurants in a city map, one may want to indicate the type of cuisine offered by each restaurant. In this setting, we allow that sites of the same class are connected to the same instance of their label. We draw each label as a polyline consisting of a horizontal segment—incident to the label—and a vertical segment; we allow that the horizontal segments overlap for sites connected to the same label because this does not cause ambiguity; see Figure 1.8. We call the overlapping horizontal segment the *backbone* of the label instance. We present algorithms for labeling all sites with the minimum number of label instances or with the minimum total length of the leaders. We also consider the version in which only one label may be placed for each class. In this case we have to allow crossings because otherwise there might be no feasible drawing. We show that minimizing the number of crossings is NP-hard.

This chapter is based on joint work with Michalis Bekos, Sabine Cornelsen, Seok-Hee Hong, Michael Kaufmann, Martin Nöllenburg, Ignaz Rutter, and Antonios Symvonis [BCF+13].

Acknowledgements

I would like to thank Alexander Wolff for giving me the opportunity to do a PhD in his group, for his frequent and very helpful advice and for sending me to several conferences, workshops, and research visits, which led to some of the chapters of this thesis. I would also like to thank all my colleagues in Würzburg for interesting discussions and especially Jan Haunert and Joachim Spoerhase for the nice and successful collaboration, which resulted in some joint papers, two of which contributed to this thesis. I am also grateful to my coauthors from other universities: Michalis Bekos, Sabine Cornelsen, Herman Haverkort, Seok-Hee Hong, Michael Kaufmann, Tamara Mchedlidze, Martin Nöllenburg, Sergey Pupyrev, Maxwell Roberts, Ignaz Rutter, Julian Schuhmann, André Schulz, and Antonios Symvonis. Of these, I would like to especially thank Sergey Pupyrev for hosting me three weeks in Ekaterinburg and for our interesting and successful collaboration on metro-line crossing minimization (see Chapters 4 and 5). I am grateful to Julian Schuhmann for doing the implementation and tests for the curvy metro maps. Finally, I would also like to thank my sister, Marion Fink, for proofreading the introduction of this thesis.

Chapter 2

Preliminaries

This chapter contains a brief introduction to graphs, graph drawing, complexity, and related topics. This is not meant as a full introduction to these topics, but as an opportunity for developing a common terminology and for introducing the basic concepts that we will use later. For real introductory works we refer to the literature. An introduction to graphs and graph theory is given in the book of Diestel [Die10]. The books *Graph Drawing* by Di Battista et al. [DETT99] and *Drawing Graphs* edited by Kaufmann and Wagner [KW01] both give an introduction to basic concepts and algorithms for drawing graphs. The recently published *Handbook on Graph Drawing and Visualization* edited by Tamassia [Tam13] comprises many topics on graph drawing and related areas. An introduction to many elementary algorithms and to the runtime analysis of algorithms can be found in the book *Introduction to Algorithms* by Cormen et al. [CLRS09]. For additional information on complexity classes and especially on NP-hard problems, we refer to the book of Garey and Johnson [GJ79].

2.1 Graphs

A *directed* graph is defined by a pair $G = (V, E)$ where V is a set of distinguishable objects, the *vertices* of the graph, and $E \subseteq V \times V$ is the set of *edges*. The vertices are often also called *nodes*. Usually, $n = |V|$ denotes the number of vertices and $m = |E|$ denotes the number of edges. Naturally, $m \in O(n^2)$. Any directed edge $e \in E$ is a pair $e = (u, v)$ with two vertices $u, v \in V$. We say that e is the edge from u to v and call u and v the endvertices of the edge e. Sometimes, the notation $uv = (u, v)$ is used as an abbreviation.

An *undirected* graph is defined by a pair $G = (V, E)$, where any edge $e \in E$ is a set of two different vertices, that is, $E \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$. Frequently, the notation $e = (u, v)$ from directed graphs is also used for undirected graphs; in this case (u, v) and (v, u) are identified. In the graph drawing problems in this thesis, we usually work with undirected graphs.

If there is an edge $e = \{u, v\}$ (or $e = (u, v)$), we say that the vertices u and v are *adjacent* or *neighbors*; similarly, we say that the edge e is *incident* to the vertex u. We denote the set of neighbors of a vertex $v \in V$ by $N(v) := \{u \in V \mid \{u, v\} \in E\}$. The *degree* of a vertex v is the number of adjacent vertices (or of incident edges), denoted by $\deg v = |N(v)|$. Vertices with degree 0 are called *isolated*. The *maximum degree* Δ in a graph is the maximum over the degrees of all vertices, that is, $\Delta = \max_{v \in V} \deg v$.

A *subgraph* of $G = (V, E)$ is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. For a set $V' \subseteq V$, we say that $G' = (V', E')$ is the subgraph *induced* by V' if E' contains any edge of G

whose endvertices are both contained in V', that is, $E' = \{\{u, v\} \in E \mid u, v \in V'\}$. The subgraph induced by V' can also be denoted by $G[V']$.

A *path* in a graph $G = (V, E)$ from a vertex u to a vertex v is a sequence

$$P = (u = v_0, v_1, v_2, \ldots, v_{k-1}, v_k = v)$$

of vertices with $\{v_i, v_{i+1}\} \in E$ (or $(v_i, v_{i+1}) \in E$) for $i = 0, \ldots, k-1$. We say that P is a path of length k because P consists of k edges. The vertices u and v are also called the endvertices of the path P. For any i with $0 < i < k$, we say that P traverses the vertex v_i or that it passes through v_i. A path is *simple* if it contains any vertex at most once. A *cycle* is a path $C = (v_0, \ldots, v_k)$ with $v_0 = v_k$ and $k > 0$. A cycle is simple if it contains any vertex except $v_0 = v_k$ at most once.

A graph $G = (V, E)$ is *connected*, if, for any pair of vertices $u, v \in V$, there is a path from u to v. A connected component of a graph is a maximal subset $V' \subseteq V$ such that the induced subgraph $G[V']$ is connected; often also $G[V']$ is called the connected component. Any graph can be uniquely partitioned into connected components; a connected graph consists only of one connected component. In graph drawing, connected components can often be drawn independently. Hence, many algorithms assume that the input is a connected graph.

A graph that does not contain any simple cycle is called a *forest*; a connected forest $T = (V, E)$ is a *tree*. It is well-known and easy to check that any tree consists of exactly $n - 1$ edges and that any pair of vertices in a tree is connected by a unique simple path. Any tree can be *rooted* at some arbitrary vertex $r \in V$. For a rooted tree, we call the unique neighbor of a vertex $v \in V$ that lies on the unique simple path connecting v to the root r the *parent* of v, or $p(v)$. The other neighbors of v are called its children. It is easy to see that all children of v have v as their parent vertex because the unique path connecting a children to r passes through v. If we remove the edge $\{v, p(v)\}$, the graph is split into two connected components; the connected component containing v is called the *subtree* of v, denoted by $T[v]$.

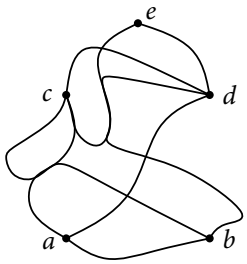
A vertex $v \in V$ of degree 1 in a tree is called a *leaf*; sometimes the term leaf is even used for a vertex of degree 1 in any graph. The interior vertices of a tree are the vertices of a degree higher than 1.

We call a graph $P = (V, E)$ for which there exists a simple path containing all edges also a *path*. Similarly, a graph with a simple cycle containing all edges is also called a cycle. Note that a path is a special case of a tree. Any path has exactly two leaves; all interior vertices have degree 2. Another special type of a tree is the caterpillar. A tree $T = (V, E)$ is a caterpillar if the subgraph induced by the set $V' \subseteq V$ of interior vertices is a path.

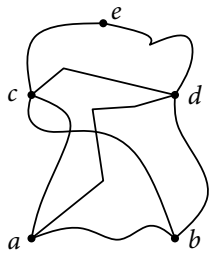
A (perfect) *matching* is a graph $G = (V, M)$ in which each vertex has degree 1. Any vertex in G is adjacent to exactly one other vertex, its matching partner. For an arbitrary graph $G = (V, E)$, a subset $M \subseteq E$ of the edges is called a matching if each vertex has degree 0 or 1 in the subgraph (V, M).

2.2 Graph Drawing

Let $G = (V, E)$ be an undirected graph. A *drawing* of G is a function Γ that maps the vertices and edges of G into the plane $\mathbb{R}^2$. The image of a vertex v is a point $\Gamma(v)$; the points of vertices have to be distinct, that is, $\Gamma(u) \neq \Gamma(v)$ for $u \neq v$. The image of an edge $e = \{u, v\}$ is a simple



(a) No feasible drawing due to overlaps.



(b) A feasible drawing.

Figure 2.1: Infeasible and feasible drawings of the same graph.

open Jordan curve $\Gamma(e)$ whose two endpoints are the points $\Gamma(u)$ and $\Gamma(v)$. We insist that the interior of the curve $\Gamma(e)$ does not contain $\Gamma(w)$ for any vertex $w \in V$.

There is a huge difference between a graph and its drawing. For any graph, there are many different drawings. However, when speaking about a specific drawing, the vertices and edges are often identified with their representation in the drawing, that is, we speak about the position of a vertex v when we mean $\Gamma(v)$ and about an edge e when we mean the drawn $\Gamma(e)$ of this edge.

Crossings and Planarity. While the drawing of an edge must not contain a vertex in its interior, it is allowed that the drawings of two edges e_1 and e_2 share a point of their interior. In this case, we say that e_1 and e_2 *cross* in the drawing. However, we do not allow that the edges overlap, that is, that there is a part of the drawing of e_1—more than a finite number of points—that is also contained in the drawing of e_2; see Figure 2.1. In general, drawings with few crossings are preferred. If a drawing does not have any crossing, we say that the drawing is *planar*; we also call a graph planar, if there exists a planar drawing of this graph. Most graphs are nonplanar: Following from Euler's formula, any planar graph has at most $3n - 6$ edges. However, a lot of research in graph drawing is devoted to planar graphs.

For graphs that are not planar, there are two main ideas for creating drawings. First, a drawing should have as few crossings as possible. Unfortunately, finding a drawing with the minimum number of crossings is NP-hard as Garey and Johnson showed [GJ83]. The second idea is that *crossing angles*—the angles defined by the curves representing the two edges in a crossing—should be large, that is, as close to 90° as possible. A drawing in which any crossing angle is 90° is called a *right-angle crossing* (or RAC) drawing. A generalization are large-angle crossing drawings, in which any crossing angle must be at least α for a constant angle $\alpha \in (0; 90°]$. In Chapter 5 we will use these requirements for crossing angles.

Faces and Embeddings. Suppose, we are given a planar drawing Γ of a graph $G = (V, E)$. We can observe that the representation of the vertices and edges in the drawing divides the plane into several regions, called *faces*. Any face is bounded vertices and edges lying on its boundary. Furthermore, there is always exactly one face of infinite area, called the *outer face*; see Figure 2.2 for an example.

Certainly, any edge can be incident to at most two different faces in a drawing, one on each of its sides. Similarly, any vertex is adjacent to at most $\deg v$ different faces. Furthermore,

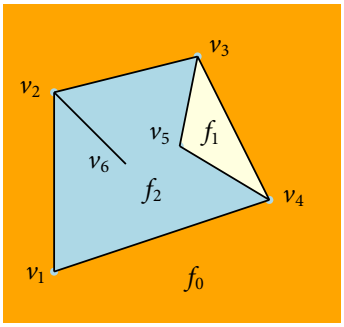
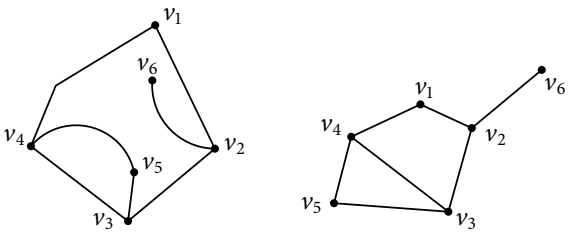


Figure 2.2: Planar drawing of a graph with three faces f_0, f_1, and f_2, where f_0 is the outer face.



(a) Drawing with the same embedding as in Figure 2.2.

(b) Drawing with different (outerplanar) embedding.

Figure 2.3: Two drawings of the graph shown in Figure 2.2.

we can describe each face of a drawing by the clockwise order of edges and vertices on its boundary; here, clockwise means that when following the sequence of edges, the described face is always to the right of the current edge. We call the description of all faces of a planar drawing a *(combinatorial) embedding* of the graph. Note that—following from the description of the faces—the combinatorial embedding implicitly also describes the clockwise order of the edges incident to each vertex.

Clearly, the combinatorial embedding corresponding to a given planar drawing Γ of a graph $G = (V, E)$ is unique. In contrast, there are many drawings with the same combinatorial embedding. Furthermore, G can have several different combinatorial embeddings; see Figure 2.3. A special type of planar graphs are *outerplanar* graphs. A graph is outerplanar if it has a combinatorial embedding in which all vertices are incident to the outer face; see Figure 2.3b.

Suppose, the drawing Γ is nonplanar. By replacing each crossing in the drawing by a dummy vertex, we get a modified planar graph G', also called a planarization of G, together with a planar drawing Γ'. The combinatorial embedding corresponding to Γ' then also describes the nonplanar drawing Γ of G. By this modification, we can also speak about nonplanar embeddings.

Basic Algorithms. An essential problem in graph drawing is deciding whether a given graph $G = (V, E)$ is planar. This problem can be solved efficiently, and there are several algorithms for planarity testing with a linear runtime, for example, the algorithm of Hopcroft and Tarjan [HT74]. Usually, such algorithms do not only decide whether G is planar, but also output a combinatorial embedding if the graph is planar; see, for example, the work of Mehlhorn and Mutzel [MM96] based on the algorithm of Hopcroft and Tarjan. Therefore, many algorithms for drawing planar graphs assume that a combinatorial embedding is part of the input; then, usually, a drawing realizing the given embedding is computed.

Not all planar embeddings of a graph can be realized in any drawing style for the edges. A remarkable exception to this are straight-line edges: Any combinatorial embedding of any planar graph can be realized as a straight-line drawing, for instance by using the algorithm of de Fraysseix, Pach, and Pollack [dFPP90] or the one of Schnyder [Sch90]. Another well-known result exists for orthogonal drawings, that is, drawings in which the edges are polylines

Input: a graph $G = (V, E)$, $\varepsilon > 0$, integer $K > 0$
Obtain initial drawing.
while number of iterations $< K$ **and** maximum displacement $> \varepsilon$ **do**
    Compute forces on vertices.
    Sum up forces and obtain movement vector for each vertex.
    Apply the forces to the current drawing.
return output drawing

Algorithm 2.1: Basic structure of force-directed algorithms.

consisting of vertical and horizontal segments only. Clearly, only planar graphs of maximum degree 4 can be drawn in this style; otherwise, a pair of edges incident to the same vertex would overlap. However, any embedding of a planar graph of maximum degree 4 can be realized in the orthogonal drawing style. Even the total number of bends of the edges can be minimized efficiently as Tamassia showed [Tam87].

Force-Directed Algorithms. A well-known class of graph drawing algorithms are *force-directed algorithms*. These heuristics are popular because they are easy to understand and extend and, especially, because they can be used for drawing any graph, not just planar graphs. The force-directed approach was introduced in 1984 by Eades in his *spring embedder* algorithm [Ead84]. Later, many other force-directed algorithms were developed; one of the most popular is the one of Fruchterman and Reingold [FR91]. Here, we give only a brief introduction and focus on the version of Fruchterman and Reingold. In Chapter 3, we will develop a new force-directed algorithm for drawing metro maps. In Chapter 8, we will develop very simple force-directed algorithms for boundary labeling with Bézier curves.

Normally, force-directed algorithms create a straight-line drawing of a graph. As all edges are drawn as straight-line segments, the only relevant output is the position of each vertex.

The idea is simple: We start with some arbitrary—in some cases even random—drawing of the graph. Then, iteratively the drawing is improved, until we finally get some satisfactory result. For force-directed algorithms, the main ingredient for improving the current drawing are *forces*. Any force is a function assigning to any vertex of the drawing a desired movement vector. Usually, forces are defined locally, taking into account only the position of one or two vertices, for example, the endvertices of an edge. In an iteration, all forces exerted by edges or by other vertices on each vertex are computed. For each vertex, the forces sum up to a movement vector. At the end of each iteration, the computed movements are applied and the drawing is modified. Then, the next iteration starts. The basic structure of force-directed algorithms is shown in Algorithm 2.1.

There are two very common forces: An attractive force between adjacent vertices and a repelling force between all pairs of vertices. The basic idea idea is that all edges should have approximately equal length l and that non-adjacent vertices should have larger distance than adjacent vertices. Here, we use the forces defined by Fruchterman and Reingold [FR91].

First, suppose that the vertices v and u are adjacent, that is, there is an edge $\{u, v\} \in E$. Then, the vertex u exerts the attracting force

$$F^{\text{att}}(u, v) = \frac{d(u, v)}{l} \cdot \overrightarrow{vu}$$

on v. This movement vector points from v to u, that is, the force tries to move v to be closer to u. Furthermore, the force is stronger if the distance $d(u, v)$ between v and u is larger in the current drawing.

Additionally, *any* vertex u exerts on any vertex v the repulsive force

$$F^{\text{rep}}(u, v) = \left(\frac{l}{d(u, v)}\right)^2 \cdot \overrightarrow{uv}.$$

This force tries to move v away from u. The closer the two vertices are, the stronger is the force.

Note that an adjacent vertex u exerts both an attractive and a repelling force on v. These two forces can then be seen as a combined force $F^{\text{att}}(u, v) + F^{\text{rep}}(u, v)$, which moves v towards u if $d(u, v) < l$ and moves v away from u if $d(u, v) > l$. If $d(u, v) = l$, we have $F^{\text{att}}(u, v) + F^{\text{rep}}(u, v) = 0$.

Many different optimized and specialized algorithms based on the force-directed approach have been developed; see, for example, Kobourov's recent survey [Kob13] in the *Handbook on Graph Drawing*.

Edge Styles. We have already mentioned several styles for drawing the edges of a graph. The simplest edge drawings are shortest connections between the endvertices, that is, straight-line segments. Another drawing style, which we will especially use in Part II, are polylines, that is, sequences of straight-line segments that are connected by bends. If an edge consists of k segments, it has $k - 1$ *bends*. Hence, we can measure the complexity of an edge by the number of its bends. For limiting the complexity of drawings, we often restrict ourselves by setting an upper bound for the number of bends of an edge. We did already mention the orthogonal drawing style. This style is a restricted version of polyline edges, in which only horizontal and vertical segments are allowed. As mentioned before, only graphs of maximum degree 4 can be drawn in the orthogonal style, while any planar graph of maximum degree 4 has a planar orthogonal drawing.

Another style for drawing edges are smooth curves. They overcome the problem of polylines, that is, they do not have sharp bends, but still yield more flexibility for routing the edges than straight-line edges do. A simple example of smooth curves are circular arcs. Often, however, parametric curves with even more flexibility are used. We will next introduce an example of such parametric curves that we will use in this thesis.

Bézier Curves. We use so-called *cubic Bézier curves*. For more information about Bézier curves in general, we refer to the book of Prautzsch et al. [PBP02]. A cubic Bézier curve C is given by the cubic polynomial

$$P_C\colon [0, 1] \to \mathbb{R}^2, t \mapsto (1 - t)^3 p + 3(1 - t)^2 t p' + 3(1 - t) t^2 q' + t^3 q,$$

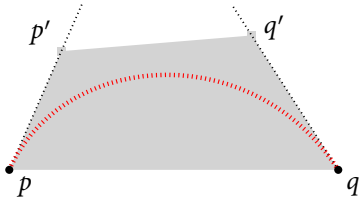


Figure 2.4: A cubic Bézier curve.

where p, p', q', and q are the *control points* of C; see Figure 2.4. We call p and q also the *endpoints* of C, because $P_C(0) = p$ and $P_C(1) = q$.

By considering the derivative of P_C, it is not hard to see that the curve C leaves the endpoint p in the direction of p', that is, the line pp' is the tangent of C at p, since $P'_C(0) = 3(p' - p)$. Similarly, one can check that the curve C enters the point q coming from the direction of q', that is, the line $q'q$ is the tangent of C at q, since $P'_C(1) = 3(q - q')$.

Another well-known property of a Bézier curve is that the curve is always contained in the convex hull of its control points (see the gray shaded region in Figure 2.4).

When creating drawings with Bézier curves, we often need to check whether two Bézier curves intersect or come too close to each other. Computing intersections of cubic curves is not easy. Since we just want to ensure that curves are not *too close*, it suffices to test *polygonizations* of the curves at hand. Given a cubic Bézier curve with polynomial P and an accuracy $\lambda \in \mathbb{Z}_{>0}$, we define the polygonization of C to be the polygonal chain connecting the points $P(0), P(1/\lambda), \ldots, P((\lambda-1)/\lambda), P(1)$. The larger we make λ, the more precise but also the slower our closeness check gets. As a speed-up, we can first test whether the convex hulls intersect.

2.3 Complexity and NP-Hardness

In this thesis, we often have to speak about the complexity of algorithms and of problems. As everywhere, also in graph drawing, there are decision problems and optimization problems. An example is the crossing number problem. In its optimization version, one has to find a drawing of a given graph such that the number of crossings is minimum. We can turn this into a decision problem by asking whether a drawing with a most k crossings exists. Of course, if we can optimally solve the optimization version, we can also solve the decision variant. When we say that a problem is (NP-)hard, we normally mean that its decision version is hard. By a correspondence as in the example, we can however, also speak about hardness of optimization problems.

We assume that the reader is familiar with the basic knowledge about the runtime analysis of algorithms. When denoting the asymptotic runtime of algorithms, we usually use the big O notation. Recall that the runtime of an algorithm is a function $f\colon \mathbb{N} \to \mathbb{N}$ that maps the input size of the algorithm to the time necessary for solving an input of this size. The class

$$O(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \exists_{c>0, n_0 \in \mathbb{N}} \forall_{n \geq n_0} g(n) \leq c \cdot f(n)\}$$

contains all runtime functions that are asymptotically at least as fast as f.

Polynomial-Time Algorithms. We are especially interested in the classes $O(n^k)$ for a constant k. If a constant k exists such that the runtime of an algorithm is in $O(n^k)$, then we say that it is a *polynomial-time* algorithm, or an *efficient* algorithm. Note that here, efficient just means that the runtime is bounded by some polynomial; it does not necessarily mean that the algorithm is fast enough for practical use.

The class P consists of all problems for which a deterministic algorithm exists that solves the problem in polynomial time. There is also the class NP which contains all problems that can be solved by a *nondeterministic* polynomial-time algorithm. It is well-known that $P \subseteq NP$. However, the question whether $P = NP$ is still open; it is one of the most important questions in computer science.

NP-Hardness. For many problems it is known that they are in NP; however, no efficient algorithm is known. A well-known class of problems is the class of NP-hard problems. Informally, a problem is *NP-hard* if it is as hard as any problem in NP; that is, if this problem can be solved efficiently, we can solve any problem in NP efficiently. Therefore, as most people assume that $P \neq NP$, there is little hope for solving NP-hard problems efficiently.

More formally, a decision problem A is NP-hard, if, for any problem B in NP, there exists a *polynomial-time* reduction from B to A. A polynomial-time reduction from B to A transforms any instance I_B of B of size n in polynomial time into an instance I_A of size $p(n)$—for a polynomial function p—such that there is a feasible solution to I_B if and only if there is a feasible solution to I_A.

A special class of NP-hard problems are problems that are NP-complete, which means that these problems are both NP-hard and contained in NP.

There are many problems that are known to be NP-hard or even NP-complete; the book of Garey and Johnson [GJ79] contains many of them. Usually, NP-hardness of a problem A is proven by giving a polynomial-time reduction from some known NP-hard problem B to A. Since this reduction can be concatenated with a polynomial-time reduction from any problem C in NP to B, this yields that there is a polynomial-time reduction from any problem in NP to A; hence, A is NP-hard.

In this thesis, we will show that some of the considered problems are NP-hard. Hence, we now list some important NP-hard problems that we will use in hardness proofs.

SAT The input of the *Satisfiability* problem (or SAT, for short) consists of a boolean formula F in conjunctive normal form over a set $X = \{x_1, \ldots, x_n\}$ of variables; that is, the formula F is a conjunction of clauses. Any clause c has the form $c = (l_1 \vee \ldots \vee l_k)$, where, for $i = 1, \ldots, k$, l_i is a literal. Any literal is a negated or an unnegated variable, that is, $l_i = \neg x_j$ or $l_i = x_j$ for some variable $x_j \in X$. Note that the formula F is fully described by its set C of clauses. Hence, we can describe the SAT instance by (X, C).

Given such a boolean formula F, one has to decide whether there exists a truth assignment $X \to \{\texttt{false}, \texttt{true}\}$ such that F is satisfied, that is, it evaluates to `true`. This is the case, if for each clause $c \in C$ at least one literal $l_i \in c$ evaluates to true, meaning that x_j is `true` if $l_i = x_j$ or that x_j is `false` if $l_i = \neg x_j$.

Note that SAT has been the first problem known to be NP-complete. In 1971, Cook showed this in a fundamental proof [Coo71].

3SAT The problem 3SAT is the version of SAT in which any clause consists of at most three literals. This restriction is still NP-complete. In contrast, the version 2SAT—with at most two literals in a clause—can be solved efficiently. Note that, if there is a clause containing only one literal, then its variable can easily be eliminated by fixing it to the adequate truth value. Hence, we can assume that each clause contains two or three literals.

Planar 3SAT A version of 3SAT that is still NP-hard, and that is especially relevant for showing hardness of graph drawing problems is Planar 3SAT. An instance (X, C) of Planar 3SAT is an instance of 3SAT for which the graph $G_{XC} = (X \cup C, E_{XC})$ is planar. G_{XC} is defined by the edge set

$$E_{XC} = \{\{x, c\} \mid x \in X, c \in C \text{ and } x \in c \text{ or } \neg x \in c\},$$

that is, there is an edge connecting a variable x and a clause c if x occurs in c.

Planar 3SAT is known to be NP-hard even if any variable occurs in exactly three different clauses [DJP+94], that is, $|\{c \in C \mid x \in c \text{ or } \neg x \in c\}| = 3$ for each $x \in X$.

Not-All-Equal 3SAT A variant of 3SAT with modified definition of satisfiability is Not-All-Equal 3SAT. An instance (X, C) is also an instance of 3SAT. However, in Not-All-Equal 3SAT, the instance (X, C) is satisfiable if and only if there exists a truth assignment to the variable set X so that each clause $c \in C$ contains a literal that evaluates to `true` and a literal that evaluates to `false`. Not-ALL-Equal 3SAT is also NP-complete.

3Partition Partitioning problems are another example of a problem class containing many hard problems. In the problem 3Partition, we are given a set $A = \{a_1, \ldots, a_{3n}\}$ of $3n$ positive integers. We have to decide whether there exists a partition of A into n sets $A_1, \ldots, A_n$ of three numbers each, such that all A_i have the same sum $s = 1/n \cdot \sum_{i=1}^{3n} a_i$.

3Partition is known to be *strongly* NP-hard. This means that we can assume that each number $a_i \in A$ is only polynomial in n. As a practical consequence for proving hardness of problems, this allows us to model a_i by a set of a_i objects, like vertices or edges, as a unary encoding.

Approaches for Hard Problems. For NP-hard problems, there is little hope to find an efficient algorithm. For optimization problems, however, we still can try to find feasible solutions of reasonable quality.

The simplest approach for hard problems is developing a *heuristic*, that is, an algorithm that finds a feasible solution using reasonable operations. We do not know anything about the quality of the solution. Alternatively, we can give an algorithm that promises at least a bound on the quality of the solution. For instance, for a minimization problem, we can try to find an algorithm that outputs a solution whose cost we can bound with respect to the input size, for example, the size of the input graph. We do, however, still not know how the quality relates to an optimum solution.

A more advanced approach is to find an *approximation algorithm*. Suppose that an algorithm for a minimization problem finds a solution of value ALG for an instance where the value of an optimum solution is OPT. If we can find a constant c such that for any input we have $\text{ALG} \leq c \cdot \text{OPT}$, we say that we have a c-approximation algorithm. There are also approximation

algorithms where c is not a constant but a value whose size depends just on the size of the input—but not on the actual input itself. As an example, we can have a $(\log n)$-approximation algorithm where n is the input size.

Fixed-Parameter Tractability. A relatively new approach is fixed-parameter tractability. Additionally to the regular input of a problem, a *fixed-parameter* algorithm has an additional parameter k. The algorithm then needs to solve the problem exactly in $O(f(k) \cdot n^c)$ time, where $f: k \mapsto \mathbb{R}^+$ is a computable function, n is the input size, and c is constant. The runtime can, hence, be separated into two factors: a polynomial factor, depending only on the input size, and a factor of arbitrary (computable) time, depending only on the parameter k. If such an algorithm exists, we say that the problem is fixed-parameter tractable with respect to the parameter k.

There are two main cases for the parameter k. On the one hand, k can be part of the output of the optimization variant of the problem. For example, for crossing minimization, we can ask whether there exists a solution with at most k crossings, where k is the parameter. One the other hand, k can also describe a property of the input instance such as the maximum degree of an input graph. In this thesis, we will use both types of parameters. For further information on fixed-parameter tractability, we refer to the books of Downey and Fellows [DF99, DF13] or of Niedermeier [Nie06].

Part I

Metro Maps

Chapter 3

Drawing Metro Maps using Bézier Curves

The automatic layout of metro maps is a well-known problem in graph drawing that has been investigated quite intensely over the last few years. Previous work has focused on the *octilinear* drawing style where edges are drawn horizontally, vertically, or diagonally at 45°; this is the style used in many official metro maps in cities like London, Paris, or Tokyo. Due to the limitation to segments of four different slopes, octilinear metro maps naturally have a schematized look. On the downside, however, they also contain sharp bends in metro lines.

We suggest the use of the *curvilinear* drawing style, inspired by manually created curvy metro maps; instead of straight-line segments, we use Bézier curves for drawing edges of the metro network. In this drawing style, we are able to forbid metro lines to bend (even in stations); this allows the user of such a map to trace the metro lines more easily. In order to create such drawings, we use the *force-directed* framework; the drawing is gradually optimized, based on *forces* that can be defined using physical analogies. Our method is the first that directly represents edges as curves and operates on these curves.

3.1 Introduction

The problem of drawing metro maps automatically has been investigated by a number of publications over the last decade. Using the terminology of graph drawing, it is stated as follows. The input is a plane *graph* $G = (V, E)$, a map $\Pi\colon V \to \mathbb{R}^2$ that associates with each vertex its *geographic location*, and a *line cover* $\mathcal{L}$ of G; the elements of $\mathcal{L}$ are paths in G with the property that every edge is contained in at least one path. The desired output is a drawing of G that fulfills or optimizes a set of aesthetic constraints. The paths in $\mathcal{L}$ are the *metro lines*; hence, we also call the vertices of G *stations*. In this chapter, as well as in other methods for drawing metro maps, the focus is on drawing the graph. The metro lines are taken into account in so far as they should have few bends when drawn on top of the graph. However, the actual insertion of the metro lines into a drawing of the graph is an additional problem that is usually solved as a post-processing. Since one wants to have as few crossings between metro lines, the problem is known as *metro-line crossing minimization*; we will care about metro-line crossing minimization in Chapters 4 and 5.

Note that, in theory—especially for metro-line crossing minimization—, it is often assumed that the metro lines are simple paths. In practice, this is the case for most lines. However, some

lines can also be cycles or traverse a vertex more than once. The algorithm presented in this chapter creates feasible drawings even if there are lines that are not simple paths.

Previous algorithmic approaches [HMdN06, SRMW11, NW11] for drawing metro maps have all used a similar set of constraints comprising topology preservation, bend minimization, minimization of geographic distortion, edge length uniformity, non-overlapping station label placement, and *octilinearity*, that is, the requirement that edges must be drawn horizontally, vertically, or diagonally at 45°; see Figure 3.1a.

Octilinearity vs. Curvilinearity. Geographic designers seem to use a set of constraints, similar to the ones mentioned for previous algorithmic approaches. Especially octilinearity is used for most metro maps; see, for example, the book of Ovenden [Ove03]. Such *schematic* maps potentially offer usability benefits by simplifying line trajectories, and hence reducing the amount of information that is irrelevant for deciding how to travel from one station to another. However, there is often a mistaken belief that it is merely the use of straight lines and a restricted angle set that benefits the user, and as a consequence many human designers fail to optimize octilinear maps, converting chaotic real-life line trajectories into complex sequences of short straight-line segments and bends [Rob12].

In other instances, the network structure itself makes the benefits of octilinearity difficult to realize. A number of systems worldwide suffer from this, including the Paris Métro. In some cases, using a different level of linearity may help. For example, one could use multiples of 30°, that is, horizontal and vertical segments, as well as segments with slopes of 30° and 60°. If these slopes better match the line trajectories of the network, they can permit more effective optimization. However, in the case of a dense interconnected network, where line trajectories are complex, a linear schematic may simply fail to offer sufficient simplification because of the network structure—irrespective of whether a human or a computer attempts the design.

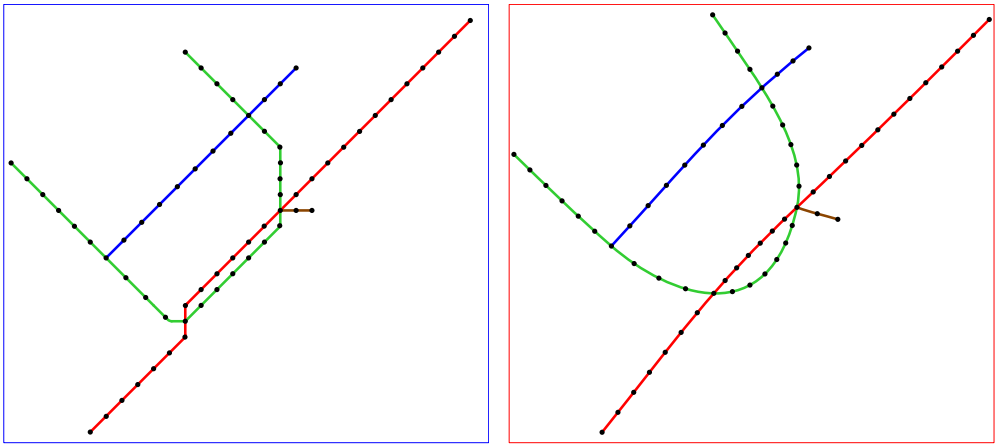
Under such circumstances, where the density of bends cannot be reduced, a curvilinear schematic map may be attempted instead; see Figure 3.1b and Figure 3.2 for curvilinear drawings of the metro networks of Montréal and of Sydney, respectively. Such a map seeks to simplify line trajectories, using curves rather than straight lines. The underlying logic is that if a linear schematic yields sequences of many visually disruptive bends, then gentle curves with imperceptible radius change are preferable. This translates into using (fixed-degree) Bézier curves subject to the following criteria:

(B1) Any pair of Bézier curves that are consecutive on a metro line must meet in a station and must have the same tangent there.

(B2) The aim for each individual metro line is to consist of the smallest number of Bézier curves necessary in order to maintain interchanges.

(B3) Points of inflection should be avoided.

In the specific case of the Paris Métro, such a design is able to smooth and to emphasize the orbital lines (lines 2 and 6), simplifying the appearance of the network and making salient its underlying structure. In a user study, a hand-drawn curvilinear design based on the above criteria out-performed the conventional octilinear Paris metro map, with up to 50% improvement in planning speed [RNL+13]. Figure 3.3 shows a manually created example of a curvilinear metro map that tries to take the criteria into account.



(a) Octilinear drawing.

(b) Curvilinear drawing created by our algorithm.

Figure 3.1: Octilinear and curvilinear drawings of the metro network of Montréal.

Previous Work. Previous algorithmic work on drawing metro maps used (mostly) octilinear polylines rather than smooth curves for representing edges. Hong et al. [HMdN06] presented a force-directed algorithm for drawing metro maps; their algorithm finds a drawing with slopes approximating octilinearity. Afterwards they use an interactive external labeling system to place station labels with few overlaps.

The algorithm of Merrick and Gudmundson [MG07] first covers the metro network by paths and then uses a path simplification approach for creating a schematized drawing. Another fast heuristic based on the schematization of paths was presented by Dwyer et al. [DHM08].

Stott and Rodgers [SRMW11] suggested another approach; who used multicriteria optimization based on hill climbing for drawing metro maps. Their approach performs local vertex moves as long as they improve the quality measure. This way, they are able to create drawings in which almost all edges are octilinear. They also integrated a label placement heuristic, so that one iteration of vertex movements alternates with one label placement iteration until no more local improvements are possible.

Nöllenburg and Wolff [NW11] used mixed-integer linear programming (MIP) for producing metro maps. Their approach always satisfies hard constraints like octilinearity and overlapping-free labeling, and optimizes soft constraints, for example, the number of bends or geographic distortion. The runtime is high and determined by the time needed to solve the MIP with an external solver; an instance of their model may have no feasible solution at all. Yet, the layout quality in their case study is high and judged as the most similar to manually designed maps in an expert survey conducted with 41 participants who compared their layouts with those of Hong et al. [HMdN06] and Stott and Rodgers [SRMW11].

Ribeiro et al. [RRL12] presented a fast force-directed algorithm for drawing metro maps. Their algorithm uses straight-line edges and allows configuration with different parameters. It does, however, not use a specific style such as octilinearity.

Figure 3.2: Metro Network of Sydney drawn by our algorithm.

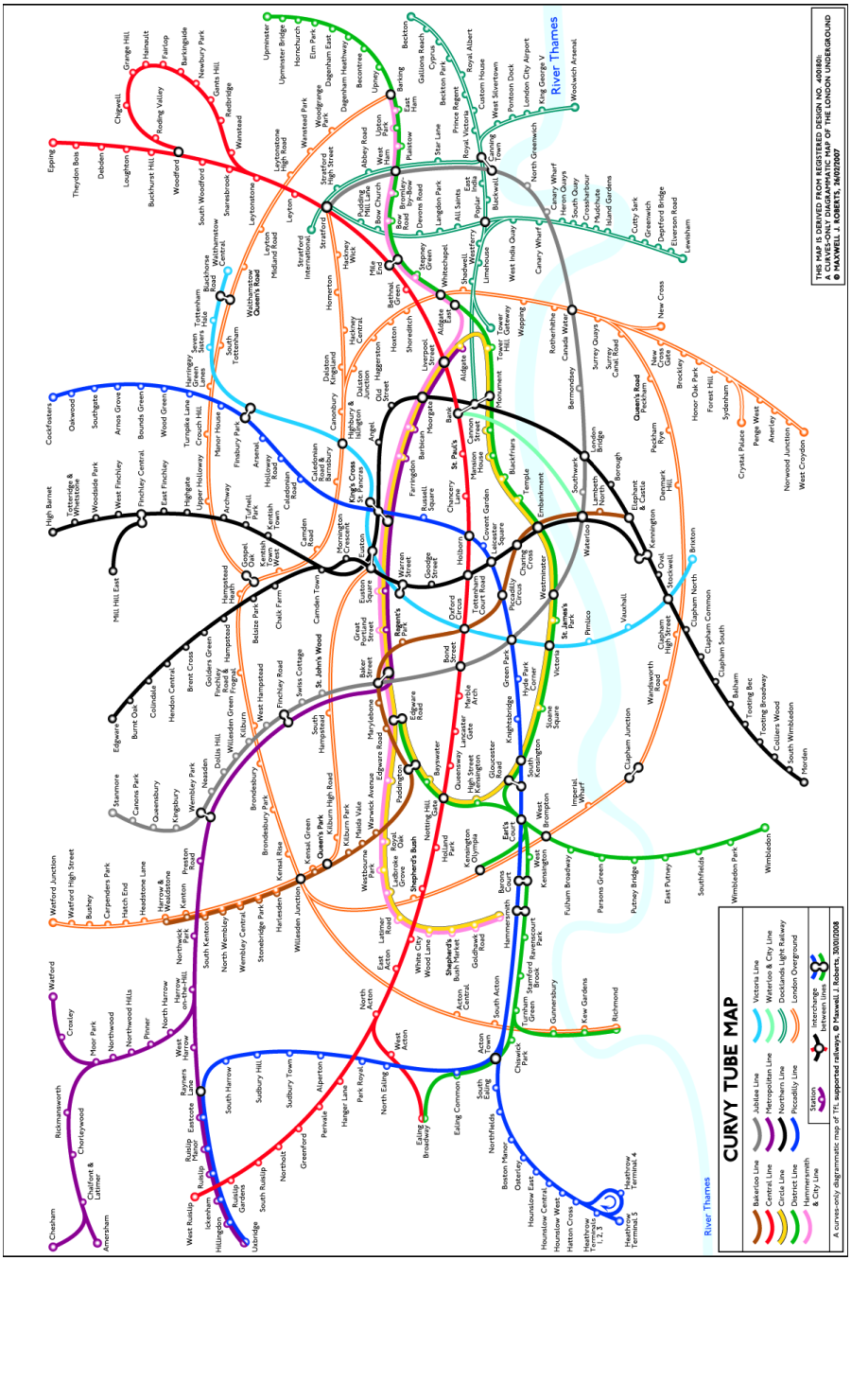


Figure 3.3: Curvilinear map of the London Underground [Rob12]. ©Maxwell J. Roberts, 2008. Reproduced with permission. All rights retained by the designer.

Ti and Li [TL14] presented a method that first enlarges dense areas of the network by distorting the geographic map and then schematizes the resulting distorted network using an external stroke-based algorithm.

Wang and Chi [WC11] developed a system for octilinear on-demand focus-and-context metro maps; in their maps, they highlight routes returned by a route planning system while showing the rest of the network as less important context information. Their approach can also be used for drawing non-focused metro maps. They deform the given geographic map by minimizing a set of energy terms modeling the aesthetic constraints. Labeling is performed independently. Their method is both fast and creates good layouts, for example, for mobile devices.

In graph drawing—without the metro map setting—, curves have been used before. For example, several works considered *Lombardi* drawings, introduced by Duncan et al. [DEG+12]. In this drawing style, edges are drawn as circular arcs. As an additional requirement, for each vertex all angles occurring between the incident edges have to be equal. Planar Lombardi drawings exist for trees [DEG+13], outerpaths [LN13] (a special type of outerplanar graphs), and all planar graphs of maximum degree 3 [Epp13].

Chernobelskiy et al. [CCG+12] presented heuristics—based on the force-directed approach—that create *near-Lombardi* drawings; in this style, edges are still drawn as circular arcs, but one does no longer insist on equal angles around vertices.

Similarly, Finkel and Tamassia [FT05] developed a force-directed algorithm for general-purpose graph drawing using Bézier curves for drawing the edges. Brandes and Wagner [BW00] did the same in the context of transportation networks for visualizing train connection data with fixed positions of stations. They developed a force-directed algorithm that draws single edges between fixed locations as Bézier curves. In both cases, the authors turned all control points into vertices of the graph and used algorithms for straight-line drawings.

Our Contribution. Our drawing algorithm is based on the force-directed approach. In contrast to the algorithms of Finkel and Tamassia [FT05] and of Brandes and Wagner [BW00], our algorithm does not turn the control points of the curves into vertices. We introduce new forces that operate on the curves by moving vertices and control points in different ways. Our new forces aim at producing drawings that take the above requirements (B1) to (B3) into account.

We first describe our basic algorithm (see Section 3.3). By construction, it ensures requirement (B1), that is, there are no bends within metro lines. We improve the visual complexity of the output of the basic algorithm by merging pairs of Bézier curves that are consecutive along a metro line, wherever this is possible; see Section 3.4 (and Figure 3.13). This optimizes requirements (B2) and (B3). Force-directed algorithms depend a lot on their initial configuration; we run our algorithm on both octilinear drawings and geographic layouts (see Section 3.5). We have implemented our algorithm (in Java) and tested it on the metro maps of London, Montréal, Sydney, and Vienna; see Section 3.6 for the results.

3.2 Preliminaries

In what follows, we review the two main ingredients that we use, with a focus on the properties that we will need for our algorithm (see also Section 2.2 for a more detailed introduction): First,

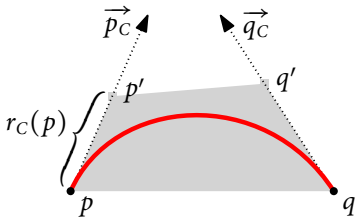


Figure 3.4: Cubic Bézier curve.

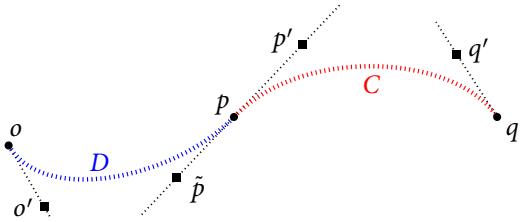


Figure 3.5: A smooth concatenation of Bézier curves C and D in point p.

we detail how our algorithm treats the Bézier curves which we use for drawing edges. Second, we quickly recall the force-directed approach.

Bézier Curves. Bézier curves are a special type of parametric curves; see Section 2.2 for a brief introduction. We use *cubic* Bézier curves. Recall that a cubic Bézier curve C is given by the cubic polynomial

$$P_C\colon [0,1] \to \mathbb{R}^2,\ t \mapsto (1-t)^3 p + 3(1-t)^2 t p' + 3(1-t) t^2 q' + t^3 q,$$

where p, p', q', and q are the *control points* of C; see Figure 3.4. We call p and q also the *endpoints* of C. We say that p' is the control point of C *at* p and that q' is the control point of C *at* q.

We use the fact that the curve leaves p in the direction of p', that is, the line pp' is the tangent of C at p. Now, if there is another curve D with a control point $\tilde{p}$ at p such that its tangent is the same but $\tilde{p}$ is on the opposite side of p with respect to p', then the concatenation of C and D is smooth in p; see Figure 3.5. Our algorithm will ensure this behavior for consecutive edges of a metro line by construction. This makes it easier for the user of our metro maps to trace metro lines visually because we avoid bends in metro lines. Technically, we encode the position of p' by a unit-length vector $\overrightarrow{p_C}$ that gives the direction of the tangent and by the distance $r_C(p)$ between p and p'. Since we want to share a single tangent, as an object, between multiple curves, we allow $r_C(p)$ to be negative. This is used when we have to model that an edge of the metro network leaves a station in exactly the opposite direction as another edge; this is desired, for instance, if a metro line passes through a station.

Our algorithm repeatedly needs to check whether two Bézier curves intersect or come too close to each other. As explained in Section 2.2, this can be done by using *polygonizations* of the curves. Furthermore, we can speed up our check by first testing the convex hulls for intersections; if they do not intersect, the curves cannot cross.

Force-Directed Algorithms. Following the force-directed framework (compare the short introduction in Section 2.2), our algorithm starts with some initial plane drawing, and then, iteratively, computes *forces* on the vertices (and control points). A force is a desired movement vector. At the end of each iteration, the computed forces are applied and the drawing is modified. Then the next iteration starts. Common forces are repulsive forces between vertices, and attractive forces between adjacent vertices. In general, forces are defined so that they tend to improve the drawing gradually. As all the forces together add up to the desired movement

Input: plane graph $G = (V, E)$, $\varepsilon > 0$, integer $K > 0$
Obtain initial crossing-free drawing with Bézier curves.
while number of iterations $< K$ **and** maximum displacement $> \varepsilon$ **do**
    Compute forces on vertices.
    Compute forces on curves.
    Apply the forces to the current drawing.
return improved output drawing

Algorithm 3.1: Basic structure of the force-directed algorithm using curves.

vectors, the forces have to be weighted so that they have the right relative strength. This is done by multiplying the force vectors with some weight factor; finding well-working factors is a matter of testing; see Section 3.6.

While we reuse standard forces known from the literature, we also define new forces that are specific to our drawing style. Whenever we have such a force that works on the shape of a curve, we will use the representation for control points introduced above: If a force tries to move a control point, then this is represented as a force that tries to rotate the tangent used by this control point, and another force that tries to modify the (signed) distance between vertex and control point.

3.3 Basic Algorithm

Our algorithm follows the general idea of other force-directed algorithms; its basic structure is shown in Algorithm 3.1.

Additionally, we have to deal with the metro lines in the given set $\mathcal{L}$. From the point of view of a station, we (usually) want each pair of incident edges that belong to the same metro line to leave the station in opposite directions. Thus, we need to maintain extra data in addition to the graph structure and layout. First, we need the set $\mathcal{L}$ of metro lines with access from lines to the edges they use and, vice versa, from the edges to the lines using them. Second, for each vertex, we have a set of tangents given by unit-length vectors pointing away from the vertex. Third, for each edge e incident to a vertex v, we have a pointer to a tangent $\overrightarrow{t}$ of this vertex as well as the signed distance $r_e(v)$. Tangent and distance describe the position of the control point of e at v.

Input Data and Initial Drawing. Our force-directed algorithm needs an initial drawing which must be crossing-free, with edges drawn as Bézier curves. If several edges incident to the same vertex v are to use the same tangent—but possibly in opposite directions—then this must be indicated in the input since such constraints are properties of the metro network. In each iteration of our algorithm—right from the start—we assume that we have such a feasible drawing. In Section 3.5 we describe how to compute an initial Bézier drawing given either a straight-line or an octilinear drawing of the metro network.

3.3.1 Forces on Vertices

We use the standard forces defined by Fruchterman and Reingold [FR91] (see also Section 2.2); they strive to move non-adjacent vertices far apart from each other and to make adjacent vertices have a common distance l. The second goal is especially useful for metro maps as the number of intermediate stops is normally a better indicator for the travel time than the geographic distance. We let any vertex u exert on any vertex v the repulsive force

$$F^{\text{rep-vtx}}(u,v) = \left(\frac{l}{d(u,v)}\right)^2 \cdot \overrightarrow{uv}.$$

If v and u are adjacent, vertex u additionally exerts the attracting force

$$F^{\text{att}}(u,v) = \frac{d(u,v)}{l} \cdot \overrightarrow{vu}$$

on v.

As a metro map represents a geographic metro network, stations should not be too far away from their actual location on the map. Therefore, we also have, for any vertex v, a force

$$F^{\text{orig}}(v) = \overrightarrow{v\Pi(v)}$$

that attracts v to its geographic position $\Pi(v)$.

3.3.2 Forces on Tangents and Control Points

Whereas previous force-directed graph-drawing algorithms did not directly operate on curves, we now present new forces for that very purpose—in order to take advantage of the power of Bézier curves.

Improving the Shape of a Curve. Consider an edge $e = uv$ that is represented as a curve with control point u' at u. If the distance $d(u,u')$ is small compared to the length of e, the curve could be very sharp, and almost have a bend. If, on the other hand, u' is far from u, the curve gets too long. As a compromise, we aim at having $|r_e(u)| = d(u,u') = d(u,v)/3$, which worked well in our tests. To achieve this, we combine an attracting and a repulsive force on u' like the Fruchterman-Reingold forces. We do not want to change the tangent, just the (signed) distance $r_e(u)$ between u and u' in the direction of the tangent vector. Hence, the desired change is

$$F^{\text{shp}}(u,u') = \left(\frac{(d(u,v)/3)^2}{|r_e(u)|} - \frac{|r_e(u)|^2}{d(u,v)/3}\right) \cdot \operatorname{sgn}(r_e(u)).$$

Note that this force is a scalar and, hence, the same type of object as the distance $r_e(u)$.

Additionally, we aim at straightening curves, as a straight-line segment is the simplest type of Bézier curve and avoids sharp bends within the edge. To this end, we move vertices as well as tangents.

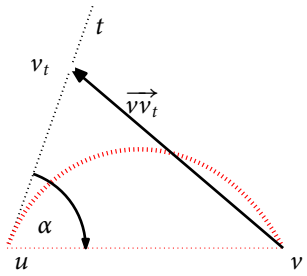


Figure 3.6: Straightening a Bézier curve.

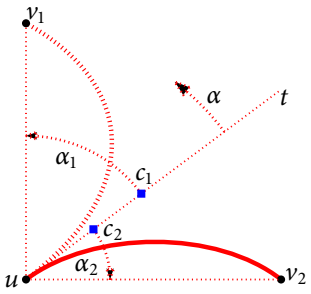


Figure 3.7: Rotational forces applied to a tangent used by two edges uv_1 and uv_2. The resulting force is a rotation by angle α.

First, we try to move vertex v so that it lies on the tangent t of uv at u. Let v_t be the point on t at distance l (the desired edge length) from u; see Figure 3.6. Now the force

$$F^{\text{str-vtx}}(u, v) = \overrightarrow{vv_t}$$

moves v towards v_t.

Second, we aim at rotating tangent t at vertex u so that v lies on t as also indicated in Figure 3.6. Let α be the signed angle between t and $\overrightarrow{uv}$. The basic idea is to rotate the tangent t by angle α so that the tangent has the direction of $\overrightarrow{uv}$. There may, however, be multiple edges incident to u using t as their tangent that all try to rotate t by different angles. A bad curvature of long edges is worse than a bad curvature of short edges; with the same rotation of the tangent, the control point—and, thus, the curve—changes much more if the distance between vertex and control point is high. Therefore, we do not simply sum up the individual forces on t, but use a sum that is weighted by the control point distances (as in the law of the lever); see Figure 3.7. Let $v_1, \ldots, v_k$ be the vertices whose edges $uv_1, \ldots, uv_k$ use tangent t with control points $c_1, \ldots, c_k$ and imply a desired change of the tangent by angles $\alpha_1, \ldots, \alpha_k$. Then the rotational force is

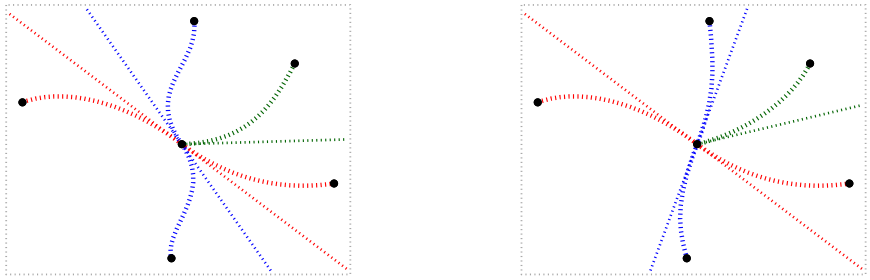
$$F^{\text{str-tng}}(t, u, v_1, \ldots, v_k) = \frac{\sum_{i=1}^{k} \alpha_i \cdot d(u, c_i)}{\sum_{i=1}^{k} d(u, c_i)}.$$

Again, the force is a scalar, as a rotation is a one-dimensional movement.

Improving the Angular Resolution. We also aim at a good angular resolution at vertices, that is, we want to have large angles between edges leaving a vertex in different directions, that is, having different tangents. For any pair of different tangents t_1, t_2 at a vertex v we, therefore, add a repulsive force

$$F^{\text{rep-tng}}(t_1, t_2) = \frac{1}{\alpha(t_1, t_2)}$$

on t_1, where $\alpha(t_1, t_2) \in [-\pi, \pi]$ is the (signed) angle formed by t_1 and t_2. Note that, when measuring the angle, we have to take into account that some vectors are used in both directions while others are just used at one side of the vertex; see Figure 3.8.



(a) before (b) after

Figure 3.8: Improving the angular resolution.

3.3.3 Avoiding Crossings by Limiting Forces

In our output drawing we do not want crossings between edges. As we start with a planar drawing, we can achieve this by ensuring that we never introduce crossings. For straight-line drawings, Bertault [Ber99] does this in his force-directed algorithm by introducing limitations of movements. To this end, he uses zones, that is, octants of directions; for each octant of a vertex, he computes a maximum displacement that is allowed if the drawing should stay planar. However, intersections of Bézier curves are more difficult to compute and predict than crossings of straight-line segments. Instead of using a zone-based approach, we therefore check, for each pair of edges, whether the intended changes to the drawing would result in a crossing. If this is the case, we change the movement vector of both endpoints so that the absolute value is half of the original value. We do this until the application of the new forces does not result in a crossing on any edge.

3.4 Decreasing the Visual Complexity

The main visual complexity of a drawing of a metro map with curves is created by a large number of inflection points (compare requirement (B3)), especially if adjacent curves of a metro line do not fit well together. Ideally, a metro line consists of just *one* Bézier curve, thus making it easy to trace the line visually (compare requirement (B2)). Often, this is not possible as intersections of a line with other lines restrict its shape. We can, however, reduce the number of curves significantly by merging consecutive curves on the same line. In our initial drawing, any edge of the graph representing the metro network is a single Bézier curve. In a step of our algorithm, we replace two consecutive curves by a single curve if this does not change the topology of the network. We now sketch how we handle different cases for merging edges that are incident to a vertex v. We distinguish the cases depending on the lines passing through v.

Merging Curves on Intermediate Nodes. Suppose a degree-2 vertex v has two incident edges $e_1 = uv$ and $e_2 = vw$ lying on a common metro line ℓ. Then the two edges share a tangent at v and leave v in opposite directions. We merge the edges into a new edge $e = uw$. We use the control point of e_1 at u and the control point of e_2 at w and check whether the drawing of e

intersects that of any other edge; see Figure 3.9. If this is not the case, we remove e_1 and e_2 and insert e into the graph and its drawing, otherwise we keep e_1 and e_2 and discard e. Theoretically, the chance of avoiding intersections could be increased by testing different values for the control point distances of the merged edge. Our tests, however, suggest that this is not necessary, since, in the final drawings, almost no vertices of degree 2 remained.

To keep track of vertices that are lost by merging edges, we have to maintain a sorted list of such vertices for each edge. As the lists of both e_1 and e_2 may already contain virtual vertices, we concatenate the two lists with v in between to get the list for e. We do not only use this list for producing the final drawing, in which we place the virtual vertices evenly distributed along the drawing of e, but we also use the number of intermediate vertices for adjusting the desired length of the edge, especially when computing the attraction between u and w. If l is the desired length for simple edges and e contains k virtual vertices, then the desired length of e is $(k+1)l$ because e represents $k+1$ edges of the real metro network.

If $\deg(v) = 1$, that is, if v is the *terminal* of some line, and the edge incident to v contains some virtual vertices, then v typically represents a terminal located in a sparsely occupied suburb. We can give more freedom to the drawing of such end edges by decreasing the influence of the force $F^{\text{orig}}(v)$ that attracts v to its original position $\Pi(v)$, for example, by scaling the force by some value $c < 1$ (in our tests, we used $c = 1/50$). This allows v to be placed closer to the center, which makes the drawing more compact.

Merging Curves on Simple Interchange Nodes. Merging pairs of curves that meet at vertices of higher degree is difficult since it is not clear how to ensure that three or more merged curves actually meet in (or close to) a single point. We first restrict ourselves to degree-4 vertices in which two lines intersect.

Suppose that a vertex v is adjacent to vertices u, u', w, w' via edges e_1, e_1', e_2, and e_2'. Line ℓ contains the edges $e_1 = uv$ and $e_2 = vw$, and line ℓ' contains $e_1' = u'v$ and $e_2' = vw'$. We want to replace the concatenation of e_1 and e_2 by $e = uw$ and that of e_1' and e_2' by $e' = u'w'$. If we manage to do so, we represent v as a virtual vertex, that is, as the intersection of e and e'; see Figure 3.10. At the same time, we have to make sure that the only new crossing that is introduced is the crossing of e and e' that represents v. We try to find appropriate curves for e and e' by adjusting the distances of the control points to the respective endpoints while keeping the tangents (as we did for vertices of degree 2). For the distance $|r_e(u)|$, we test values in the interval $[|r_{e_1}(u)|, d(u,w)]$ at equal distances. It makes sense to require $|r_e(u)| \geq |r_{e_1}(u)|$ since the combined curve is longer than e_1 and the new control point should not be too far from u. By testing all different combinations of discretized distances for the four involved control points, we found feasible solutions in most cases.

Note that there is an additional constraint: the crossing that now represents v should divide both new edges e and e' roughly in proportion to the numbers of virtual vertices on e and e', respectively, on the two sides of v. If e contains k virtual vertices left of v and k' virtual vertices right of v, then the intersection with e' should have a distance to u that is $(k+1)/(k+k'+2)$ times the total length of the curve of e. We allow a deviation from this optimal position by a factor of δ (we used $\delta = 20\%$ in our tests) times the length of the part of the curve to the left of v and to the right of v, respectively. We call the allowed range on e the *δ-zone* of e.

In all further steps of the algorithm, we have to adhere to these zones for crossing edges. A further merging of lines including e is only allowed if v stays in the allowed δ-zone. Furthermore,

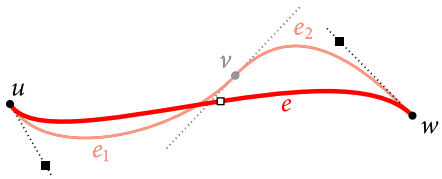


Figure 3.9: Merging edges e_1 and e_2 into a new edge e; vertex v joining the edges e_1 and e_2 will be drawn at the appropriate position on edge e.

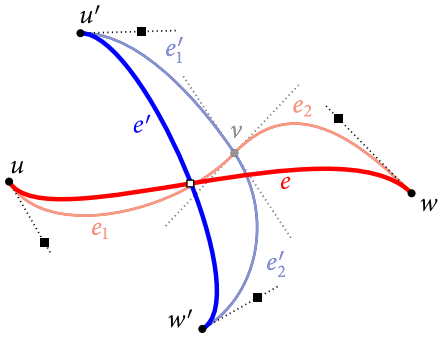


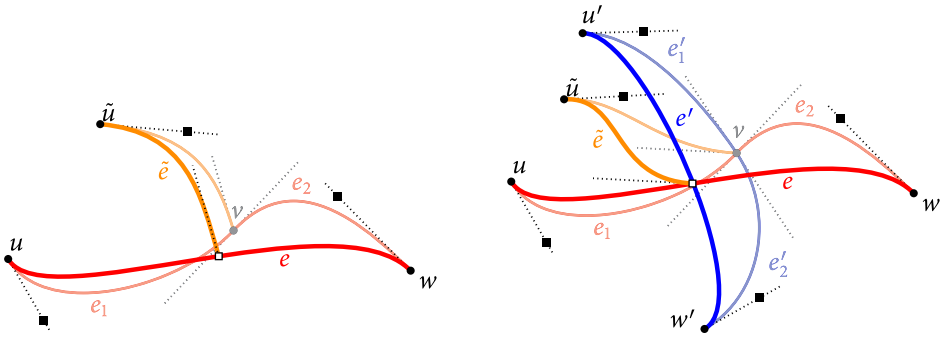
Figure 3.10: Merging edges e_1 and e_2 into a new edge e and edges e'_1 and e'_2 into a new edge e'; vertex v will be drawn at the intersection of the edges e and e'.

we do not allow movements that violate these constraints. So we also consider these zones in the force limitation phase at the end of each iteration.

Additional Possibilities for Merging Curves. To further improve our drawings, we can also try to merge curves that intersect in vertices of degree other than 2 or 4. Consider a vertex v of degree 3 that is traversed by a line ℓ_1 and that is the terminal of another line ℓ_2 with a different tangent. We can then merge the two edges of ℓ_1 and represent v by its position on ℓ_1. The edge of ℓ_2 still has to maintain its own tangent and control point at v. This is also possible if there are several lines whose tangent is not linked to ℓ_1. Similarly, if two lines ℓ_1 and ℓ_2 traverse a vertex v, we can merge their edges incident to v so that their crossing represents v—if none of the remaining edges shares its tangent with ℓ_1 or ℓ_2; see Figure 3.11 for examples where edges can be merged in vertices of degrees 3 and 5. Note that it is still possible that some of the remaining edges have a common tangent.

3.5 Creating a Feasible Input Drawing

As input, our algorithm expects the embedded graph representing the metro network, the coordinates of stations, and information regarding the metro lines; see Section 3.3. Some of this information can be guessed automatically. Suppose, for example, that exactly two of the edges incident to v are used by a line ℓ. Then, we assume that ℓ traverses the station and, hence, the two edges must have the same tangent, leaving v in opposite directions. Otherwise, we assume that the input contains an *annotation* saying, for example, that the two edges leave v in the same direction. If, on the other hand, there are three or more edges incident to v with the same line on them, then annotation is needed in any case. We will generally assume that annotations exist at any vertex. In our implementation, however, we do not need an annotation in cases where there is no doubt on the tangents at a vertex.



(a) Merging edges e_1 and e_2 into a new edge e with an extra edge $\tilde{e}$ incident to the linking node v.

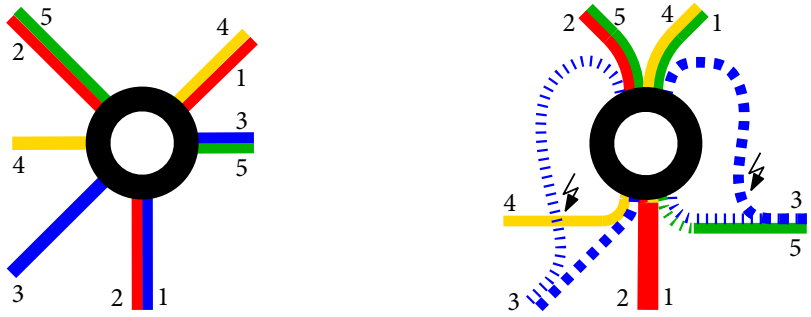
(b) Merging edges e_1 and e_2 into a new edge e and edges e'_1 and e'_2 into a new edge e' with an extra edge $\tilde{e}$ incident to the linking node v.

Figure 3.11: Examples for merging edges in vertices of degrees other than 2 or 4.

We need an initial feasible *Bézier* drawing before the first iteration of the force-directed algorithm starts; see Section 3.3. This drawing must guarantee that (i) tangents obey their annotations and (ii) the topology (that is, the embedding) is the same as in the plane input graph. We discuss two ways to obtain such an initial drawing depending on the input graph: either from an octilinear drawing or from the straight-line drawing induced by the coordinates of the stations.

Initial Drawing from Octilinear Layout. Suppose that we are given an octilinear layout which may either be computed, for example, using the mixed integer program of Nöllenburg and Wolff [NW11], or be a manually generated plan such as an official metro map. If there are bends in edges, we first transform these bends into dummy vertices that do not correspond to stations. Later, the algorithm may delete such dummy vertices by merging the two incident curves. Given the dummy vertices, we now have a straight-line drawing. To get a drawing using only Bézier curves, we place each control point at its incident endpoint (or, conceptually, at a very small distance) and let each tangent point towards the other endpoint of the edge. Now, we still have the same straight-line drawing, but the edges are technically Bézier curves. In this process, we must respect the annotations of the tangents, so that the right curves have common tangents at a vertex. Unfortunately, there can be situations in which this is not possible; see Figure 3.12. In practice, however, such situations are quite unlikely; they never occurred in our tests.

At any vertex where the tangents are not yet correct, we now choose new tangents. We do this one after the other, starting at tangents that are shared by edges. We choose the first tangent so that it is closest to the tangents it replaces. We insert any new tangent so that the clockwise order of the adjacent vertices is correct (if this is possible). Finally, by moving the control points very close to the vertex, we get a drawing that is arbitrarily close to the straight-line drawing and that does, hence, have no crossings.



(a) The (octilinear) input drawing.

(b) Lines 1 and 2 share their tangent, which has, hence, also to be used by lines 4 and 5. Line 3 either creates a crossing or violates the shared tangents in the input.

Figure 3.12: Example where it is impossible to keep the embedding and ensure that both edges of each line share the same tangent.

Initial Drawing from Geographic Layout. If we do not have an octilinear drawing, the initial drawing can also be constructed given just the coordinates of the stations. Similarly to Nöllenburg and Wolff [NW11], we start with the straight-line drawing induced by the station positions. This drawing may have crossings; we replace them by dummy vertices and get a crossing-free straight-line drawing. This drawing can then be transformed into a crossing-free Bézier drawing as presented in the previous paragraph.

Since the introduced crossings, as dummy vertices, are preserved over all iterations, they will also be present in the output drawing. Fortunately, their number is small in practice. (From a network point-of-view it indeed makes sense to have stations at crossings.) For example, the large London network (which was built by competing companies operating single lines) has only four crossings—the same as in the official tube map.

Note that, in the initial drawing, there are only two different tangents at a dummy vertex, each for one of the two crossing lines; this is also the case in the final drawing. Additionally, in the more advanced version of the algorithm, we can even transform the dummy vertex to a (dummy) virtual vertex before the algorithm starts; see Section 3.4.

3.6 Implementation and Tests

We implemented our algorithm in Java. For testing we used the metro networks of four cities: London (297 vertices, 217 of which have degree 2, 13 metro lines), Vienna (90/71/5), Montréal (69/59/4), and Sydney (173/144/10); see Figure 3.14, Figure 3.13d, Figure 3.1b, and Figure 3.2 for illustrations of the networks. The input data contained the graph structure as well as information on the lines and geographic positions of stations. We also used octilinear layouts of these cities as initial drawings, which we generated using the MIP approach of Nöllenburg and Wolff [NW11]. In both cases, tangents were annotated where necessary; see Section 3.5.

Effects of Virtual Vertices. We were especially interested in how far making vertices virtual influenced the visual complexity of metro maps. Figure 3.13 shows the power of virtual vertices for the metro map of Vienna. Starting with an octilinear layout (Figure 3.13a), the first drawing (Figure 3.13b) was computed without virtual vertices and, hence, no curves were merged. Clearly, the drawing does not have any sharp bends. The attraction to the geographic position of vertices, however, caused some unnecessary inflection points. Next, we added the possibility to create virtual vertices of degree 2 (Figure 3.13c). For Vienna, this worked on all intermediate vertices, reducing the number of Bézier curves significantly. Finally, we also enabled virtual vertices of higher degree (Figure 3.13d). For Vienna, this worked for 8 of 9 possible vertices. Two metro lines were represented by just one curve, while the three other lines need two curves each.

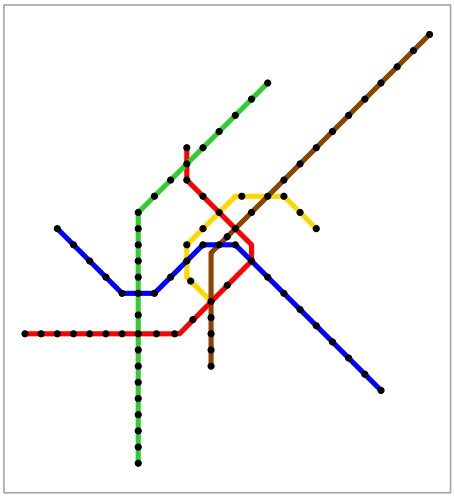
It turned out that including virtual vertices of degree 2 always worked well, and that they were fast to handle. There were almost no remaining vertices of this type even after the very first iteration; hence, testing the remaining degree-2 vertices was fast in all following iterations. In contrast, trying to merge edges at vertices of higher degree was much slower because potentially many combinations of control point positions had to be tested. Additionally, we observed that once many virtual vertices of higher degree had been added, the drawing did not change much any more. Apparently, the additional constraints on the crossings make the drawing more rigid, and many movements get forbidden. Therefore, we decided to first have many, that is, hundreds, of iterations without caring about virtual vertices of degree more than 2, and then treat them in a single (more time-consuming) final iteration.

Running Time. On the largest instance, the Underground of London, the running time for creating a drawing starting with an octilinear layout (see Figure 3.14) was 935 seconds on a 3 GHz dual-core computer with 4 GB RAM. This includes the 872 seconds spent on the last iteration, in which curves were merged by inserting virtual vertices of degree higher than 2. In contrast, the first 500 iterations just took 63 seconds.

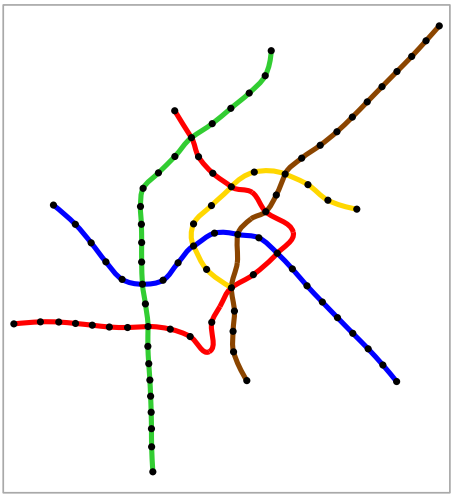
Weights of Forces. As noted in Section 3.2, weight factors are needed that let different forces work well together. We group the forces depending on the object on which they operate. In our tests, the following factors turned out to work well:

$$F^{\text{res}}_{\text{vert}} = (F^{\text{rep-vtx}} + F^{\text{att}} + 10F^{\text{orig}} + 3F^{\text{str-vtx}})/100 \quad \text{for vertices,}$$
$$F^{\text{res}}_{\text{tan}} = 150F^{\text{rep-tng}} + 0.03F^{\text{str-tng}} \quad \text{for tangents, and}$$
$$F^{\text{res}}_{\text{cpdist}} = F^{\text{shp}}/20 \quad \text{for control point distances.}$$

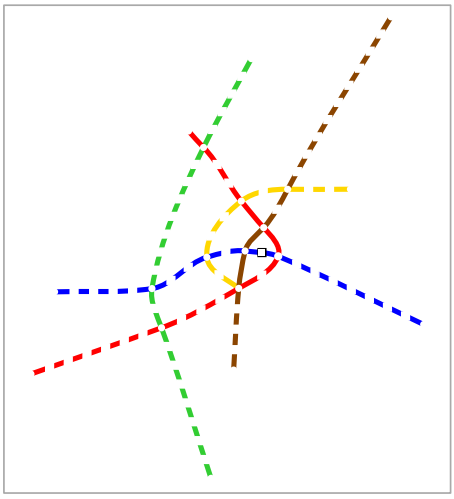
Initial Drawing and F^{orig}. We tested the algorithm both with a straight-line drawing and an octilinear layout as input. When we defined F^{orig} using the geographic station locations, the version using the octilinear layout performed slightly better. The best results, however, were achieved when using the octilinear layout as input and defining F^{orig} with respect to the vertex positions in the octilinear input drawing. In this case, the center had more space, and more curves could be merged, which reduced the visual complexity. Figures 3.2, 3.1b, 3.13 and 3.14 were computed this way.



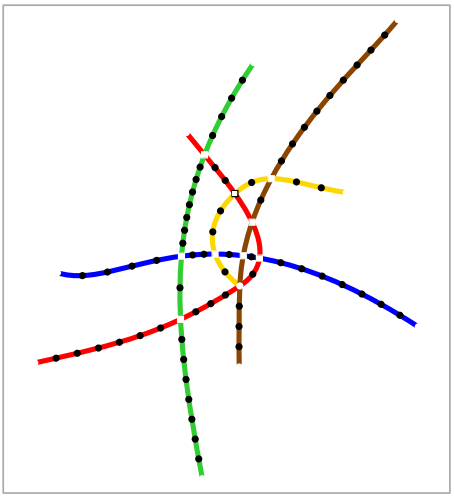
(a) Initial octilinear drawing.



(b) Drawing without virtual vertices.



(c) Drawing with virtual vertices of degree 2 (highlighted by squares).



(d) Drawing with virtual vertices of degree 2 and additionally with virtual vertices of higher degree (highlighted by squares).

Figure 3.13: Metro network of Vienna: Initial octilinear drawing and drawings produced by our algorithm, with increasing use of virtual vertices.

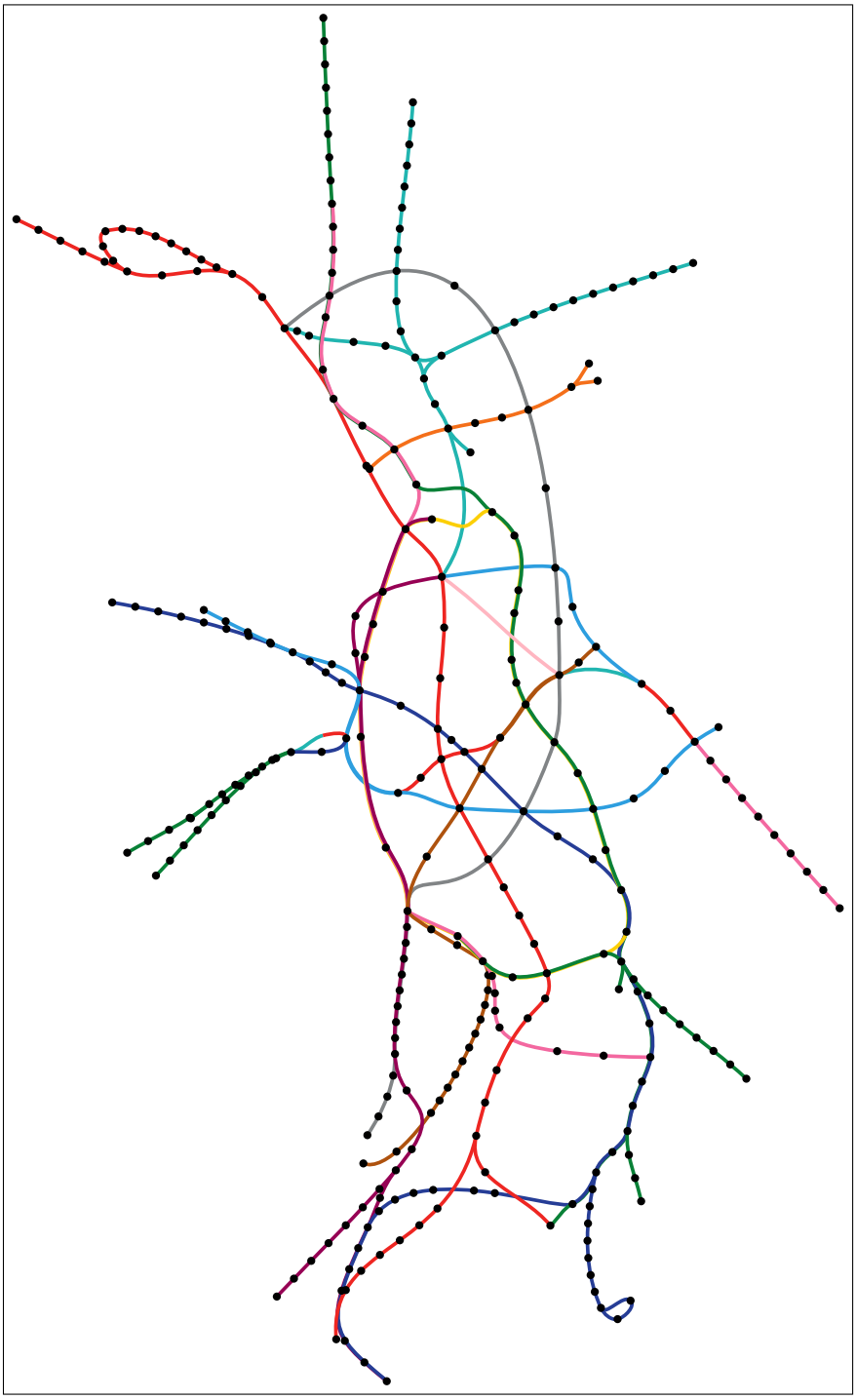


Figure 3.14: Metro Network of London drawn by our algorithm.

3.7 Concluding Remarks

The implementation of our algorithm performed well on small and medium-size networks (Vienna, Sydney, and Montreal in our tests) as well as sparse regions of large networks. In such cases, many curves could be merged so that, in the end, lines consisted only of few curves. We conclude that spending the extra time for merging as many curves as possible should always be invested. In denser regions, such as the center of London, many curves were actually merged, but there are also a number of vertices that did not allow this, making the drawing rather complex.

Open Problems. A first open problem is incorporating the placement of station labels into our algorithm. One can, of course, try to use an external labeling algorithm on the output drawing. However, it is possible that there is no space for all station labels in the output drawing. Taking the labels into account while creating the drawing would overcome this problem—but also make the algorithm more complicated.

Global Optimization. As future work, however, we suggest studying an alternative approach to generating curvy metro maps automatically by approximating each metro line globally as one C^2-continuous cubic spline right from the start rather than piece-by-piece for every edge. Curve-fitting techniques from computer graphics could be applied for finding splines that interpolate or approximate the input points with low error; the challenge would be to additionally define and implement appropriate constraints that allow for a sufficient and maybe context-dependent amount of distortion to smooth unimportant bends and yet ensure, for example, the desired angular properties in vertices of degree at least 3.

Drawings with Circular Arcs. As an alternative to Bézier curves, circular arcs may also be used for drawing metro maps. Note that representing any edge by a single circular arc will normally not suffice since the tangents on the two endpoints cannot be chosen independently. Drawing edges as a combination of two circular arcs connected at a point of common tangent can, however, be tried.

For smaller networks and networks that have many lines that either go from the center to the suburbs or go round a central region, we suggest *concentric drawings*. One first has to choose a *center* in the central region of the network. Then, edges can be represented by sequences of circular arcs—with the fixed center—and segments that are radial with respect to the same center; see Figure 3.15 for an example. If the structure of the networks allows such a concentric drawing, it should be easier to find one—at least with a fixed center—since the drawing style is more restricted. Furthermore the concentric drawing style by itself reduces the complexity of the drawing due to its focus on the center; see also our poster [FLW14] (with Magnus Lechner and Alexander Wolff) on the ongoing work on concentric metro maps.

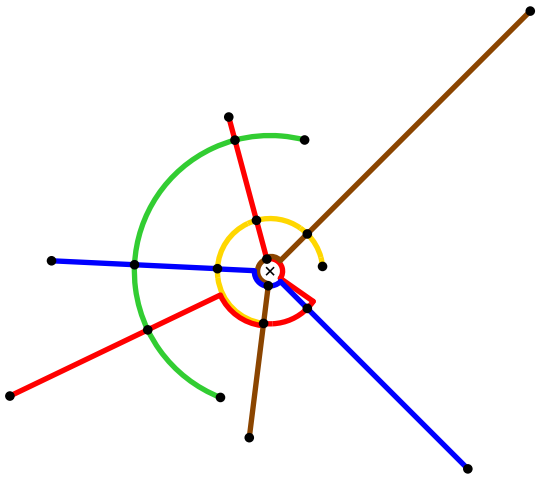


Figure 3.15: Sketch of a concentric drawing of the metro network of Vienna; intermediate stations are not shown. The drawing was automatically generated using a work-in-progress implementation [FLW14].

Chapter 4

Metro-Line Crossing Minimization

In the previous chapter, we presented an algorithm for creating curvy drawings of metro networks. In our algorithm, as well as in other algorithms for drawing metro maps, what is actually drawn is the underlying graph. As a postprocessing, the *metro lines* are then inserted into the drawing. When doing so, one has to care about several lines sharing an edge of the network. Crossings between metro lines will often be necessary. In order to make the metro lines in the drawing easy to follow, we want to insert them in such a way that the total number of crossings is minimized; this is known as the *metro-line crossing minimization* (MLCM) problem: Given an embedded graph G and a set $\mathcal{L}$ of simple paths in G, called *lines*, order the lines on each edge so that the total number of crossings is minimized.

Although there has been some research on metro-line crossing minimization, the complexity of MLCM has been an open problem, so far. In contrast, the problem variant MLCM-P, in which line ends must be placed in outermost position on their edges, is known to be NP-hard.

In this chapter, our main results answer two open questions: First, we show that the general MLCM problem is NP-hard. Second, we give an $O(\sqrt{\log|\mathcal{L}|})$-approximation algorithm for MLCM-P.

Our further results are as follows. For both problem variants, we can efficiently check whether a solution without crossings exists. For MLCM-P, we develop a fixed-parameter tractable algorithm with respect to the number of crossings; we also solve MLCM-P optimally on some non-trivial instances. Finally, we present a fixed-parameter tractable algorithm for both MLCM and MLCM-P on trees with respect to the sum of the maximum degree of the underlying graph and the maximum number of lines per edge.

4.1 Introduction

An important part of transportation networks like metro networks are transportation *lines* that connect different points using streets or railway tracks of the underlying network. As we have seen, such networks can be modeled as graphs. The edges represent railway track or road segments connecting the vertices. The lines become *paths* in the graph.

Usually, lines that share an edge are drawn individually along the edge in distinct colors; see Figure 4.1. Often, some lines must cross, and one normally wants to have as few crossings of metro lines as possible. The *metro-line crossing minimization* (MLCM) problem has been introduced by Benkert et al. [BNUW07]. The goal is to order the lines along each edge such that the number of crossings is minimized. Although the problem has been studied, many questions remain open.

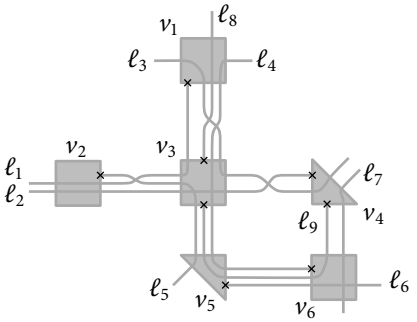
Figure 4.1: A part of the official metro map of Paris.[1]

We present our results in terms of the problem of metro-map visualization; however, crossing minimization between paths on an embedded graph is used in various fields. In very-large-scale integrated (VLSI) chip layout, a wire diagram should have few wire crossings [Gro89]. Another application is the visualization of biochemical pathways [Sch02]. In graph drawing the number of edge crossings is considered one of the most important aesthetic criteria.

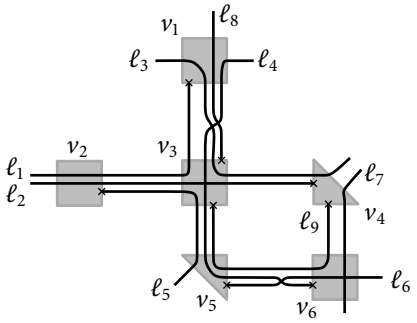
Recently, a lot of research, both in graph drawing and information visualization, has been devoted to *edge bundling*. In this setting, some edges are drawn close together—like metro lines—which emphasizes the structure of the graph [Hol06, CZQ+08, PNBH12]; minimizing crossings between parallel edges arises as a subproblem [PNBH12]. More precisely, bundled graph drawings can be interpreted as drawings of a modified graph, in which the edges of the original graph are paths—like metro lines—that often share edges. Metro-line crossing minimization then helps to have few crossing between these paths.

Problem Definitions. The input is an *embedded* graph $G = (V, E)$ together with a set $\mathcal{L} = \{\ell_1, \ldots, \ell_{|\mathcal{L}|}\}$ of simple paths in G. We call G the *underlying network* or the *underlying graph*, the vertices *stations*, and the paths *lines*. The endpoints v_0, v_k of a line $\ell = (v_0, \ldots, v_k) \in \mathcal{L}$ are *terminals*, and the vertices $v_1, \ldots, v_{k-1}$ are *intermediate stations*. For each edge $e = (u, v) \in E$ let L_e be the set of lines passing through e.

[1] Cropped from the official metro map, which is available online at `http://www.ratp.fr/fr/ratp/c_23590/plans-metro/` (Accessed on December 14, 2013). © Régie autonome des transports parisiens (RATP), Paris



(a) $\pi_{v_3v_4} = (\ell_3, \ell_2)$ and $\pi_{v_3v_1} = (\ell_1, \ell_8, \ell_4, \ell_3)$. The lines ℓ_3 and ℓ_4 have an unavoidable edge crossing on $\{v_1, v_3\}$. In contrast, the crossing of ℓ_2 and ℓ_3 on $\{v_3, v_4\}$ is avoidable. In v_3 there is an unavoidable vertex crossing of the lines ℓ_2 and ℓ_8. As the vertex crossing of ℓ_2 and ℓ_5 in v_3 is avoidable, the solution is not feasible.



(b) A feasible solution satisfying the periphery condition. Note that the two vertex crossings occurring in vertices v_3 and v_6 are both unavoidable and, hence, allowed.

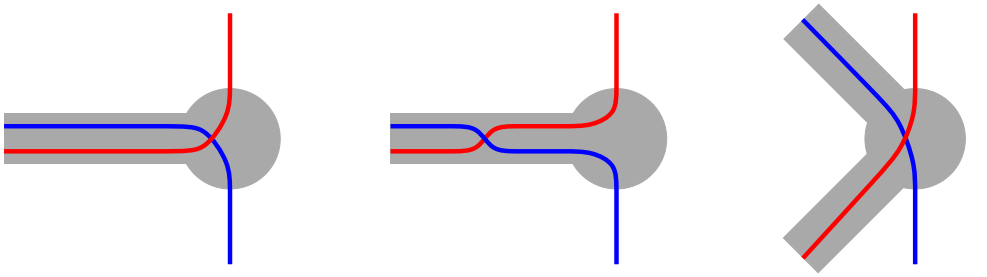
Figure 4.2: 9 lines on an underlying network of 6 vertices and 9 edges.

Following previous work [ABKS10, Nöl10], we use the *k-side* model; each station v is represented by a polygon with k sides, where k is the degree of v in G; see Figure 4.2. Note that for $k = 2$ we can still represent the station by a rectangle and use two opposite sides for connecting the edges to it; for $k = 1$ we use just one side. Each side of the polygon is called a *port* of v and corresponds to an incident edge $(v, u) \in E$. Recall that the input is an embedded graph; hence, the clockwise order of edges incident to a vertex is fixed. A line $(v_0, \dots, v_r)$ is represented by a polyline starting at a port of v_0 (on the boundary of the polygon), passing through two ports of v_i for $1 \le i < r$, and ending at a port of v_r.

For each port of $u \in V$ corresponding to $(u, v) \in E$, we define the *line order* $\pi_{uv} = (\ell_1 \dots \ell_{|L_{uv}|})$ as an ordered sequence of the lines in L_{uv}; π_{uv} specifies the clockwise order at which the lines L_{uv} are connected to the port of u with respect to the center of the polygon. Note that there are two different line orders π_{uv} and π_{vu} on any edge (u, v) of the network, describing the orders at the two ends of the edge (u, v). A *solution*, or a *line layout*, specifies line orders π_{uv} and π_{vu} for each edge $(u, v) \in E$.

A *line crossing* is a crossing between a pair of polylines corresponding to a pair of lines on the graph. We distinguish two types of crossings; see Figure 4.2a. An *edge crossing* between lines ℓ_1 and ℓ_2 occurs whenever $\pi_{uv} = (\dots \ell_1 \dots \ell_2 \dots)$ and $\pi_{vu} = (\dots \ell_1 \dots \ell_2 \dots)$ for some edge $(u, v) \in E$ since line ℓ_1 is above ℓ_2 at vertex u and below ℓ_2 at vertex v, assuming that the edge is drawn horizontally with u to the left. Note that the line order at a port is always described relative to the vertex. Hence, even if the order of lines does not change on edge $e = (u, v)$, that is, there is no crossing on e, the permutation π_{vu} is reversed compared to π_{uv}. We still say that the oder does not change, with the obvious meaning that there is no crossing.

We now consider the concatenated cyclic sequence π_u of the orders $\pi_{uv_1}, \dots, \pi_{uv_k}$, where $(u, v_1), \dots, (u, v_k)$ are the edges incident to u in clockwise order. Note that lines that pass



(a) Avoidable vertex crossing. (b) Avoidable vertex crossing of (a) moved to an edge. (c) Unavoidable vertex crossing.

Figure 4.3: Avoidable and unavoidable vertex crossings.

through vertex u are found twice in the sequence π_u. A *vertex crossing* between ℓ_1 and ℓ_2 occurs in u if $\pi_u = (\dots \ell_1 \dots \ell_2 \dots \ell_1 \dots \ell_2 \dots)$ or $\pi_u = (\dots \ell_2 \dots \ell_1 \dots \ell_2 \dots \ell_1 \dots)$. Intuitively, the lines change their relative order inside u.

A crossing of lines ℓ_1 and ℓ_2 is called *unavoidable* if ℓ_1 and ℓ_2 cross in any line layout; otherwise the crossing is *avoidable*. A crossing is unavoidable if neither ℓ_1 nor ℓ_2 have a terminal on their common subpath and the lines split on both ends of this subpath in such a way that their relative order has to change; see Figure 4.2 for examples of the crossing types. Following previous work, we insist that *avoidable vertex crossings are not allowed* in a solution, that is, these crossings are not hidden below a station symbol; this is called the *edge crossings* model. Furthermore, *we do not count unavoidable vertex crossings* since they occur in any solution; see Figure 4.3.

A pair of lines may share several common subpaths, and the lines may cross multiple times on the subpaths. For the simplicity of presentation, we always assume that there is at most one common subpath of two lines; we call this the *path intersection property*. Our results do, however, also hold for the general case as every common subpath can be considered individually.

Problem Variants. Several variants of metro-line crossing minimization have been considered so far. The original metro-line crossing minimization (MLCM) problem is formulated as follows.

Problem 4.1 (*MLCM*). *For an instance* $(G, \mathcal{L})$ *consisting of an embedded graph* $G = (V, E)$ *and a set* $\mathcal{L}$ *of lines on* G*, find a line layout with the minimum number of crossings.*

Note that the embedding of the graph does not necessarily have to be planar; the crucial part for metro-line minimization is that the clockwise order of edges incident to each vertex is prescribed by the embedding. Crossings between edges are allowed. As the lines on two crossing edges cross in any line layout, there is no need to count such crossings.

In practice, it is desirable to avoid gaps between adjacent lines; to this end, every line is drawn so that it starts and terminates in the topmost or bottommost part of a port; see Figure 4.2b. In fact, many manually created maps follow this *periphery condition* introduced by Bekos et al. [BKPS08]. Formally, we say that a line order π_{uv} at the port of u satisfies the periphery condition if $\pi_{uv} = (\ell_1 \dots \ell_p \dots \ell_q \dots \ell_{|L_{uv}|})$ with $p \leq q$, where u is a terminal for the lines

$\ell_1, \ldots, \ell_p, \ell_q, \ldots, \ell_{|L_{uv}|}$ and u is an intermediate station for the lines $\ell_{p+1}, \ldots, \ell_{q-1}$. The problem is known as metro-line crossing minimization *with periphery condition* (MLCM-P).

Problem 4.2 (*MLCM-P*)**.** *For an instance* $(G, \mathcal{L})$ *consisting of an embedded graph* $G = (V, E)$ *and a set* $\mathcal{L}$ *of lines on* G*, find a line layout with the minimum number of crossings that satisfies the periphery condition on each port.*

In the special case of MLCM-P with *side assignment* (MLCM-PA), the input additionally specifies for each line end on which side of its port it terminates; Nöllenburg [Nöl10] showed that MLCM-PA is computationally equivalent to the version of MLCM in which all lines terminate at vertices of degree one.

As MLCM and MLCM-P are NP-hard even for very simple networks, we introduce the additional constraint that no line is a subpath of another line. This is often the case for bus and metro transportation networks; if, however, there is a line that is a subpath of a longer line then one can also visualize it as a part of the longer line. We call the problems with this new restriction Proper-MLCM and Proper-MLCM-P.

Previous Work. Line crossing problems in transportation networks were first studied by Benkert et al. [BNUW07], who introduced the *metro-line crossing minimization* (MLCM) problem. They described a quadratic-time algorithm for MLCM on instances whose underlying network consists of a single edge with attached leaves. As far as we are aware, this is the only known result on MLCM so far; no efficient algorithms are known for the case of two or more edges. The complexity status of MLCM has been open.

However, the variants MLCM-P and MLCM-PA have been considered. First, Bekos et al. [BKPS08] studied MLCM-P and proved that the variant is NP-hard on paths. Motivated by the hardness, they introduced the variant MLCM-PA and presented efficient algorithms for paths and "left-to-right" trees (in which all lines have a common direction). Later, polynomial-time algorithms for MLCM-PA on general graphs were found with gradually improving running time by Asquith et al. [AGM08] ($O(|\mathcal{L}|^3 \cdot |E|^{5/2})$ time), Argyriou et al. [ABKS10] ($O((|\mathcal{L}|^2 + |E|)|E|)$ time), and Nöllenburg [Nöl10] ($O(|\mathcal{L}|^2|V|)$ time), until Pupyrev et al. [PNBH12] presented a linear-time algorithm ($O(|V| + |E| + k)$ time, where k is the total length of the metro lines). Asquith et al. [AGM08] formulated MLCM-P as an integer linear program that finds an optimal solution for the problem on general graphs; note that this approach requires exponential time in the worst case. Okamoto et al. [OTU13a] worked on MLCM-P on paths (see also the full version of their paper [OTU13b]). They showed how to efficiently decide whether there is a solution without crossings. In Section 4.3, we will show how this can be done for general graphs. Furthermore, they presented an exact algorithm for paths and a fixed-parameter tractable algorithm with respect to the number of lines per edge of the path.

A lot of recent research, both in graph drawing and information visualization, is devoted to *edge bundling* where some edges are drawn close together—like metro lines The linear-time algorithm for MLCM-PA of Pupyrev et al. [PNBH12] has been developed in this context.

In VLSI design, the problem of minimizing intersections between nets (physical wires) arises [Gro89, MS95]. Net patterns with fewer crossings are likely to have better electrical characteristics and require less wiring area as crossings consume space on the circuit board; hence, it is an important optimization criterion in circuit board design. This problem is equivalent to MLCM-PA. Groeneveld [Gro89] suggested an algorithm for MLCM-PA on general graphs in

problem	graph class	result	reference
MLCM	caterpillar	NP-hard	Thm. 4.1
	single edge	$O(\|\mathcal{L}\|^2)$ time	[BNUW07]
	tree	FPT for $\Delta + c$	Thm. 4.9
	general graph	crossing-free test	Thm. 4.2
MLCM-P	path	NP-hard	[ABKS10]
	tree	FPT for $\Delta + c$	Thm. 4.11
	general graph	ILP	[AGM08]
	general graph	$O(\sqrt{\log \|\mathcal{L}\|})$-approx.	Thm. 4.5
	general graph	crossing-free test	Thm. 4.3
	general graph	FPT for #crossings	Thm. 4.4
Proper-MLCM-P	general graph with consistent lines	$O(\|\mathcal{L}\|^2(\|\mathcal{L} + \|E\|))$ time	Thm. 4.7
MLCM-PA	general graph	$O(\|E\| + \|V\|\|\mathcal{L}\|)$ time	[PNBH12]

Table 4.1: Overview of results for the metro-line crossing minimization problem.

this context. Another method for graphs of maximum degree four was developed by Chen and Lee [CL98].

Our Results. Table 4.1 summarizes our contributions and previous results. We first prove that MLCM is NP-hard even on caterpillars, that is, paths with attached leaves (Section 4.2.1), thus, answering an open question of Benkert et al. [BNUW07] and Nöllenburg [Nöl09]. As crossing *minimization* is hard, it is natural to ask whether there exists a *crossing-free* solution. We show that there is a crossing-free solution if and only if there is no pair of lines forming an unavoidable crossing; this criterion can easily be checked (Section 4.2.2).

We then study MLCM-P (Section 4.3). Argyriou et al. [ABKS10] and Nöllenburg [Nöl09] asked for an approximation algorithm. To this end, we develop a 2SAT model for the problem. Using the 2SAT formulation we obtain an $O(\sqrt{\log |\mathcal{L}|})$-approximation algorithm for MLCM-P. This is the first approximation algorithm in the context of metro-line crossing minimization. We also show how to find a crossing-free solution in polynomial time, if such a solution exists. Moreover, we prove that MLCM-P is *fixed-parameter tractable* with respect to the maximum number k of allowed crossings, via the fixed-parameter tractability of 2SAT.

Next, we study the new variant Proper-MLCM-P (Section 4.4). We present efficient algorithms for solving Proper-MLCM-P optimally on caterpillars, *left-to-right trees*, and many other instances described in Section 4.4. The class of left-to-right trees was also considered by Bekos et al. [BKPS08] and by Aryriou et al. [ABKS10] in the context of metro-line crossing minimization. Actually, our algorithm can be applied to any graph if the lines on the graph satisfy a simple property that we call *consistent line directions*. On such instances, with the help of some transformations, we can reduce the problem of finding a crossing-minimum solution to the problem of finding a minimum edge cut in a flow network. This is the first polynomial-time exact algorithm for the variant in which avoidable crossings may be present in an optimal solution.

For both MLCM and MLCM-P, we consider the restricted variant of the problems in which the maximum degree Δ as well as the maximum *edge multiplicity* c, that is, the maximum number of lines per edge, are bounded (Section 4.5). For the case where the underlying network is a tree, we show that both MLCM and MLCM-P are fixed-parameter tractable with respect to the combined parameter $\Delta + c$.

Finally, we consider practical aspects of metro-line crossing minimization. We show how to find a (not necessarily optimal) line layout in the cases where some of the required constraints are not fulfilled.

4.2 General Metro-Line Crossing Minimization

We begin with the most flexible problem variant MLCM, and show that it is hard to decide whether there is a solution with at most $k > 0$ crossings, even if the underlying network is a caterpillar. In contrast, we give polynomial-time algorithms for deciding whether a crossing-free solution exists.

4.2.1 NP-Completeness

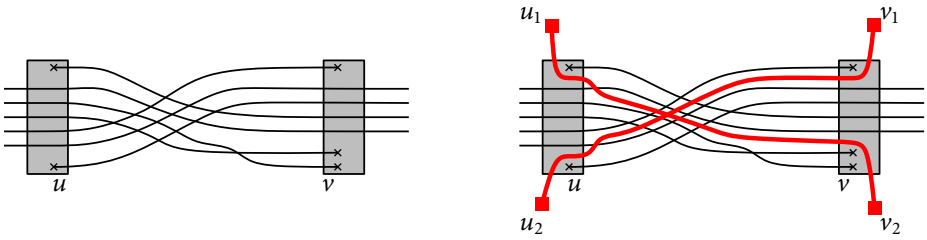
Argyriou et al. [ABKS10] showed that the restricted version MLCM-P is NP-hard even if the underlying graph is a path. It is easy to see that this does not hold for the general version MLCM. In fact, any such instance has a crossing-free solution. Later, we can see this with the help of Theorem 4.2 since, on a path, there is no pair of lines with an unavoidable crossing. There is, however, also a very simple algorithm for creating such a solution: Traverse the path from left to right and maintain an ordered sequence of the lines. Whenever a line starts, add it at the end of the sequence; when a line ends, it is simply removed. This does not create any crossings. Since in MLCM lines can end anywhere on their respective port, the solution is feasible.

If, however, the graphs become just slightly more complex than paths, MLCM is NP-hard: We show that the problem is NP-complete even for caterpillar graphs, that is, paths with attached vertices of degree 1.

Theorem 4.1. *MLCM is NP-complete even on caterpillar graphs.*

Proof. We prove hardness by reduction from MLCM-P; as mention above, MLCM-P is known to be NP-hard on paths [ABKS10]. Suppose that we have an instance of MLCM-P consisting of a path $G = (V, E)$ and a set $\mathcal{L}$ of lines on the path. We want to decide whether it is possible to order the lines with periphery condition and at most k crossings.

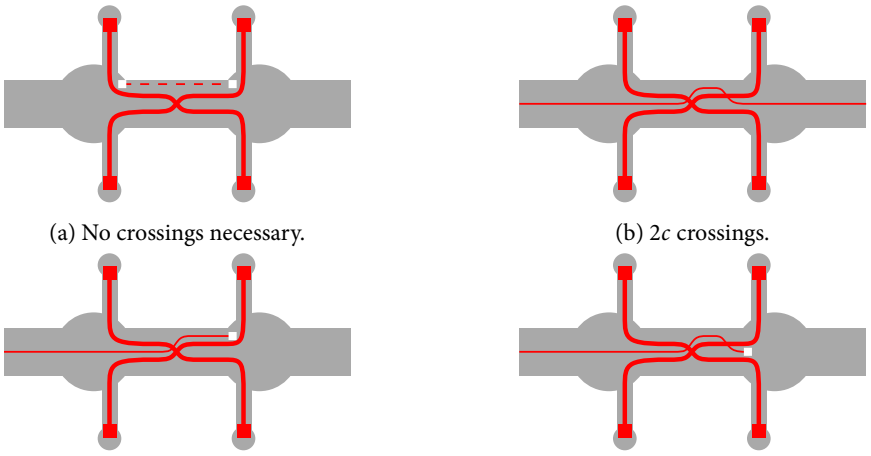
We create a new underlying network $G' = (V', E')$ by adding some vertices and edges to G. We assume that the path G is embedded along a horizontal line and specify new positions relative to this line. For each edge $e = (u, v) \in E$, we add vertices u_1, u_2, v_1, and v_2 and edges (u, u_1), (u, u_2), (v, v_1), and (v, v_2) such that v_1 and u_1 are above the path and v_2 and u_2 are below the path. Next, we add $c = |\mathcal{L}|^2$ lines connecting u_1 and v_2, and c lines connecting u_2 and v_1 to $\mathcal{L}' \supseteq \mathcal{L}$; see Figure 4.4. We call the added structure the *red cross* of e, the added lines *red lines*, and the lines of $\mathcal{L}$ *old lines*. We claim that there is a number K such that a solution of MLCM-P for $(G, \mathcal{L})$ with at most k crossings exists if and only if a solution of MLCM for $(G', \mathcal{L}')$ with at most $k + K$ crossings exists.



(a) MLCM-P-solution on edge (u, v).

(b) Insertion of a red cross into the solution with minimum number of additional crossings.

Figure 4.4: Insertion of the red lines of our red cross gadget into an MLCM-P solution on an edge (u, v).



(a) No crossings necessary.

(b) $2c$ crossings.

(c) c crossings.

(d) $2c$ crossings.

Figure 4.5: Crossings of lines with the red cross on an edge.

Let $e = (u, v) \in E$ be an edge of the path, and let $\ell \in L_e$ be a line on e. If ℓ has its terminals at u and v, that is, ℓ lies completely on e, then ℓ never has to cross any other line in G or G' (see Figure 4.5a); hence, we can assume that such a line ℓ does not exist.

Now, assume that the line ℓ has none of its terminals at u or v. It is easy to see that ℓ has to cross all $2c$ lines of the red cross of e (see Figure 4.5b). Finally, suppose that ℓ has just one terminal at a vertex of e, say at u. If the terminal is placed above the edge (u, u_1) then ℓ has to cross all red lines connecting u_2 and v_1 but can avoid the red lines connecting u_1 and v_2; that is, c crossings with red lines are necessary on edge e (see Figure 4.5c). Symmetrically, if the terminal is below (u, u_2) then only the c crossings with the red lines from u_1 to v_2 are necessary. If the terminal is between the edges (u, u_1) and (u, u_2), however, then all $2c$ red lines must be crossed (see Figure 4.5d). There are, of course, always c^2 unavoidable internal crossings of the red cross of e.

Let $c_e = c_e^t + c_e^m$ be the number of old lines on e, where c_e^t is the number of old lines on e that have a terminal at u or v, and c_e^m is the number of old lines on e that have no terminal at u or v. In any solution there are at least $c_e^t \cdot c + 2 \cdot c_e^m \cdot c + c^2$ crossings on e in which at least one red line is involved. It is easy to see that placing a terminal between red lines leaving towards a leaf never brings an advantage. On the other hand, if just a single line has an avoidable crossing with a block of red lines, the number of crossings increases by $c = |\mathcal{L}|^2$, which is more than the number of crossings in any solution for $(G, \mathcal{L})$ without double crossings. Hence, no optimal solution of the lines in G' has avoidable crossings with red blocks and, therefore, any optimal solution satisfies the periphery condition; thus, after deleting the added edges and red lines, we obtain a feasible solution for MLCM-P on G.

Let $K := |E| \cdot c^2 + \sum_{e \in E} (c_e^t + 2c_e^m) \cdot c$ be the minimum number of crossings involving red lines in the graph G'. Suppose that we have an MLCM-solution on G' with at most $K + k$ crossings. Then, after deleting the red lines, we obtain a feasible solution for MLCM-P on G with at most k crossings. On the other hand, if we have an MLCM-P-solution on G with k crossings, then we can insert the red lines with just K new crossings as follows. Suppose that we want to insert the block of red lines from u_1 to v_2 on an edge $e = (u, v) \in E$. We start by putting them immediately below the lines with a terminal on the top of u. Then we cross all lines below until we see the first line that ends on the bottom of v and, hence, must not be crossed by this red block. We go to the right and just keep always directly above the block of lines that end at the bottom side of v; see Figure 4.4. When we reach v, we have not created any avoidable crossing. Once we have inserted all blocks of red lines, we obtain a solution for the lines on G' with exactly $K + k$ crossings. This completes the proof of the NP-hardness.

It remains to show that MLCM is contained in NP. As Argyriou et al. [ABKS10] observed for MLCM-P, one can simply guess orders for all ports and then check, for any combination of orders, in polynomial time whether the orders form a feasible solution with at most k crossings. This also works for MLCM; the only difference is that more line layouts are feasible for MLCM. Hence, MLCM is NP-complete. □

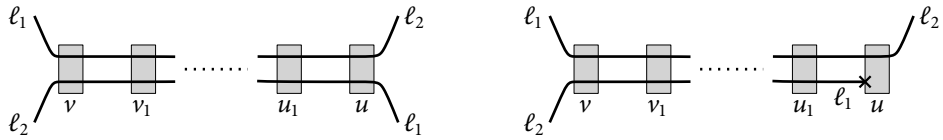
4.2.2 Recognition of Crossing-Free Instances

As we have seen, MLCM in general is NP-hard. In contrast, we will now see that it is easy to check whether an instance of the problem allows a crossing-free solution.

Suppose that we are given an instance $(G, \mathcal{L})$ of MLCM and we want to check whether there exists a solution without any crossings. If a crossing-free exists solution, then, obviously, there cannot be a pair of lines with an unavoidable crossing. We show that this necessary condition is already sufficient.

Consider a pair of lines ℓ_1, ℓ_2 with a common subpath $P = (v, v_1, \dots, u_1, u)$; see Figure 4.6. Suppose that the lines *split* at v, that is, neither ℓ_1 nor ℓ_2 terminates at v. Since vertex crossings are not allowed in our model, there is a unique order between ℓ_1 and ℓ_2 at the port vv_1 in any feasible solution of MLCM. Furthermore, in any crossing-free solution, the relative order of ℓ_1 and ℓ_2 is the same on all ports.

We arbitrarily fix a direction for each edge of the underlying network. For an edge $e = (u, v) \in E$ directed from u to v and for a pair of lines $\ell_1, \ell_2 \in L_{uv}$, we say that ℓ_1 is *above* ℓ_2 if $\pi_{uv} = (\dots \ell_1 \dots \ell_2 \dots)$ in any crossing-free solution taking just these two lines into account—and disregarding all other lines. Otherwise, if $\pi_{uv} = (\dots \ell_2 \dots \ell_1 \dots)$ in any crossing-free



(a) Line ℓ_1 is above line ℓ_2 at the port vv_1 but below ℓ_2 at the port uu_1; a crossing of ℓ_1 and ℓ_2 is unavoidable.

(b) A terminal on a common subpath of ℓ_1 and ℓ_2 at u; the crossing is avoidable.

Figure 4.6: Avoidable and unavoidable crossings.

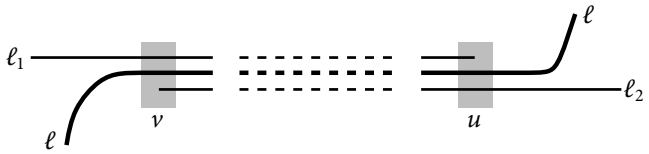


Figure 4.7: A separator ℓ of lines ℓ_1 and ℓ_2.

solution, we say that ℓ_1 is *below* ℓ_2. Note that on some other edge e', ℓ_1 may be below ℓ_2, depending on the direction of e'. We say that a line ℓ *lies* between ℓ_1 and ℓ_2 if ℓ_1 is above ℓ and ℓ is above ℓ_2 on e. First, a useful observation.

Observation 4.1. *The lines ℓ_1, ℓ_2 have an unavoidable crossing if and only if they split in such a way that, on some edge e, ℓ_1 has to be above ℓ_2 and at the same time ℓ_2 has to be above ℓ_1.*

We assume that no line is a subpath of another line because a subpath can be reinserted parallel to the longer line in a crossing-free solution. Consider a pair of lines ℓ_1 and ℓ_2 whose common subpath P starts in u and ends in v. If u (or, similarly, v) is a terminal neither for ℓ_1 nor ℓ_2 then either there is a unique relative order of the lines along P in any crossing-free solution or a crossing is unavoidable; see Figure 4.6. Hence, we assume that u is a terminal for ℓ_1, v is a terminal for ℓ_2, and we call such a pair of lines *overlapping*. Suppose there is a *separator* for ℓ_1 and ℓ_2, that is, a line ℓ on the common subpath of ℓ_1 and ℓ_2 that has to be below ℓ_1 and above ℓ_2 (or the other way round) as shown in Figure 4.7. Then, ℓ_1 has to be above ℓ_2 in any crossing-free solution.

The only remaining case is a pair of lines without a separator. With the help of the following lemma we will be able to simplify the instance by merging such pairs.

Lemma 4.1. *Let ℓ_1, ℓ_2 be a pair of overlapping lines without a separator, for which the number of edges of the common subpath is minimum. If there exists a crossing-free solution, then then there also exists a crossing-free solution in which ℓ_1 and ℓ_2 are immediate neighbors in the orders on their common subpath, that is, they are never separated by a line lying between them.*

Proof. First, let us show that no line has its terminal in an intermediate station of the common subpath of ℓ_1 and ℓ_2. Suppose there is such a line ℓ. Then ℓ forms an overlapping pair with either ℓ_1 or ℓ_2—say ℓ_1 without loss of generality—, whose common subpath is shorter than the one of ℓ_1 and ℓ_2. Hence, there is a separator ℓ' of ℓ_1 and ℓ_1; ℓ' also separates ℓ_1 and ℓ_2 in contradiction to the choice of ℓ_1 and ℓ_2.

(a) Lines S between ℓ_1 and ℓ.

(b) Lines between ℓ_1 and ℓ reordered.

Figure 4.8: Rerouting lines between ℓ_1 and ℓ_2. All shown lines share a subpath, which is shown in the drawings. At any position with a terminal, there is a node (not drawn).

Now, suppose that there is no crossing-free solution in which ℓ_1 and ℓ_2 are immediate neighbors on their complete subpath. We fix a crossing-free solution in which the number of lines lying between ℓ_1 and ℓ_2 is minimal and suppose that, without loss of generality, ℓ_1 is above ℓ_2.

Let ℓ be a line lying between ℓ_1 and ℓ_2. If we follow line ℓ to the left and to the right, it either ends at a subpath with ℓ_1 or ℓ_2, or it leaves these two lines. If ℓ leaves both ℓ_1 to the left and ℓ_2 to the right, then ℓ must be a separator, a contradiction. On the other hand, if ℓ leaves only one of the lines, say ℓ_1, it overlaps with the other one, that is, with ℓ_2, and forms an overlapping pair with it.

Let ℓ be the topmost line that lies between ℓ_1 and ℓ_2 in the solution and overlaps with ℓ_1 (or, symmetrically, the bottommost line that overlaps with ℓ_2). By modifying the ordering, considering lines above ℓ and below ℓ_1, and some lines not overlapping with ℓ_1, it is possible to reroute ℓ so that it does not lie between ℓ_1 and ℓ_2, hence decreasing the number of lines between ℓ_1 and ℓ_2, in contradiction to the choice of the solution. To this end, let $S \supseteq \{\ell_1, \ell\}$ be the smallest superset of ℓ and ℓ_1 such that for any pair of lines $\ell', \ell'' \in S$ any line that lies between ℓ' and ℓ'' in the solution is also contained in S; see Figure 4.8a. Note that no pair of lines in S has a separator because this separator would also be a separator for ℓ_1 and ℓ_2.

If $S = \{\ell_1, \ell\}$ we can easily reroute ℓ to be above ℓ_1. Otherwise, we apply the following procedure. For any pair of overlapping lines in S that are immediate neighbors—that is, there is no line lying in between—, we reroute the right line to be immediately above the left line, which is possible as there is no separating line. Eventually, ℓ will be above ℓ_1; otherwise, there would still be steps to be performed; see Figure 4.8b. Hence, we can create a solution in which there is at least one line less between ℓ_1 and ℓ_2, a contradiction. □

In the situation of the previous lemma, we can safely merge ℓ_1 and ℓ_2 into a new line ℓ that starts and ends at the terminals of ℓ_1 and ℓ_2 that are not on the common subpath of the two lines. We will now use this merging for simplifying the instance so that we can conclude that any instance without unavoidable crossings allows a crossing-free line layout.

Theorem 4.2. *Let $(G = (V, E), \mathcal{L})$ be an instance of MLCM. A crossing-free solution exists if and only if there is no pair of lines with an unavoidable crossing.*

Proof. If there is a pair of lines with an unavoidable crossing, then naturally there is no crossing-free solution. Now assume that there is no unavoidable crossing. We will show how to find a crossing-free solution.

Using Lemma 4.1, we can merge a pair of overlapping lines without a separator into a new line. The merging cannot introduce an unavoidable crossing as we will see. Suppose there would

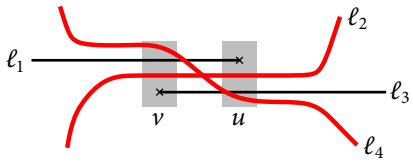


Figure 4.9: Unavoidable crossing of two separators of ℓ_1 and ℓ_3.

be a line ℓ forming an unavoidable crossing with the merged line ℓ' of ℓ_1 and ℓ_2. The lines ℓ and ℓ' have to split on both ends with with different side constraints. The splits have to be on different sides of the common subpath of ℓ_1 and ℓ_2, otherwise there already was an unavoidable crossing of ℓ with ℓ_1 or ℓ_2. From the splits we get relative orders of ℓ with ℓ_1 and ℓ_2 such that either ℓ_1 is above ℓ and ℓ is above ℓ_2, or ℓ_2 is above ℓ and ℓ is above ℓ_1. In both cases ℓ already was a separator for ℓ_1 and ℓ_2 and we would not have merged the lines.

We iteratively perform merging steps until any overlapping pair has a separator. Note that there might be multiple separators for a pair, but all of them separate the pair in the same relative order; otherwise, we would have a pair of separators with an unavoidable crossing; see Figure 4.9.

After the merging steps, for any pair of lines sharing an edge, we either obtain a unique relative order for crossing-free solutions, or the pair has a separator.

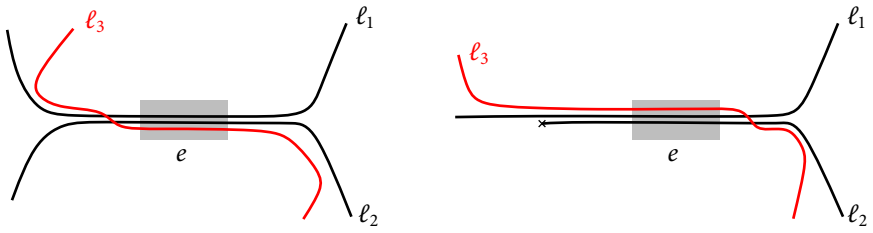
We now create a directed *relation graph* G_e for each edge $e \in E$. Vertices of the graph are the lines L_e passing through e. Edges of G_e model the relative order of the lines in a crossing-free solution; we have an edge (ℓ_1, ℓ_2) (similarly, (ℓ_2, ℓ_1)) in G_e if ℓ_1 and ℓ_2 split in such a way that ℓ_1 is above (below) ℓ_2 in any crossing-free solution.

Let us prove that all relation graphs are acyclic. Suppose there is a cycle in a relation graph G_e. We choose the shortest cycle C. A cycle of length 2 is equivalent to a pair of lines with an unavoidable crossing; hence, such a cycle cannot exist.

Now, suppose there is a cycle $C = (\ell_1, \ell_2, \ell_3)$ of length 3. Lines ℓ_1 and ℓ_2 share a common subpath and split on one side in the order (ℓ_1, ℓ_2). The splitting for realizing the edge (ℓ_2, ℓ_3) can not be realized on this subpath; otherwise, we would also get the edge (ℓ_1, ℓ_3). Similarly, the splitting for (ℓ_3, ℓ_1) also can not be realized on the subpath. Hence, we have to distribute the two splittings to the two sides of the subpath, which is not possible without introducing an unavoidable crossing with ℓ_1 or ℓ_2; see Figure 4.10.

Finally, if the shortest cycle C has length at least four, then there exists a path $(\ell_1, \ell_2, \ell_3, \ell_4)$ of length four without chords. As there is no edge between ℓ_1 and ℓ_3, ℓ_1 and ℓ_3 have to be an overlapping pair and ℓ_2 is a separator for them. On the other hand, ℓ_4 is also a separator for ℓ_1 and ℓ_3, but separates them in another relative order. It is easy to see that there is an unavoidable crossing of ℓ_2 and ℓ_4, a contradiction; see Figure 4.9. Hence, the relation graphs are acyclic.

Now, in a relation graph G_e for any pair of lines $\ell_1, \ell_2 \in L_e$, there is either a directed edge connecting the lines in G_e, or the lines are overlapping; in the latter case, there has to exist a separator for ℓ_1 and ℓ_2 and, hence, a directed path of length 2 connecting ℓ_1 and ℓ_2 in G_e. Since G_e is acyclic, there exists a topological ordering of the lines L_e. Due to the existence of separators—and, hence, connecting paths for overlapping lines—the topological ordering is



(a) ℓ_1 and ℓ_2 split on both sides. (b) ℓ_1 and ℓ_2 split only on one side.

Figure 4.10: Situations occurring in the proof of Theorem 4.2 for lines ℓ_1, ℓ_2, and ℓ_3 that have a common edge e.

unique. We obtain a crossing-free solution by using the ordering for every edge. As the relative order of any pair of lines is the same for all edges, there cannot be a crossing. □

The proof yields an algorithm for finding a crossing-free solution if there is no unavoidable crossing. It needs $O(|\mathcal{L}|^2|E|)$ time for deleting subpaths as well as iteratively merging the shortest unseparated overlapping pair. Finally, we can get the relative order of each pair of lines on all edges in $O(|\mathcal{L}|^2|E|)$ time and we can order the lines on all edges. Hence, after reinserting the deleted or merged lines, we obtain a crossing-free solution in $O(|\mathcal{L}|^2|E|)$ time.

4.3 Metro-Line Crossing Minimization with Periphery Condition

We now turn to the problem variant MLCM-P, that is, the version of MLCM in which line ends must be outermost at their ports. As mentioned in the previous section, MLCM-P has been shown to be NP-hard by Argyriou et al. [ABKS10]. Similar to what we did for MLCM, we will show that checking whether there is a crossing-free solution can be achieved in polynomial time also for MLCM-P. We will also show that the problem variant is fixed-parameter tractable and develop an approximation algorithm. Our results are based on a 2SAT model that we will develop first.

4.3.1 A 2SAT model for MLCM-P

Let $(G = (V, E), \mathcal{L})$ be an instance of MLCM-P. Our goal is to decide, for each line end, on which side of its terminal port the line end should be placed. For convenience, we arbitrarily choose one side of each port and call it "top", the opposite side is called "bottom". For each line ℓ starting at vertex u and ending at vertex v, we create binary variables ℓ_u and ℓ_v, which are `true` if and only if ℓ terminates at the top side of the respective port. We formulate the problem of finding a truth assignment that leads to a crossing-free solution as a 2SAT instance for the given instance of MLCM-P. Note that Asquith et al. [AGM08] already used 2SAT clauses as a tool for developing their ILP for MLCM; the variables in the clauses represent above/below relations between line ends. In contrast, in our model a variable directly represents the position of a

(a) The lines cross. (b) Crossing eliminated by rerouting.

Figure 4.11: Avoiding a crossing of lines with a common terminal by rerouting.

line on the top or bottom side of a port. This allows us to derive further results using the 2SAT formulation.

As a preparation, we show that lines sharing a terminal never have to cross. As a consequence, we later will not have to care about pairs of such lines.

Lemma 4.2. *Let ℓ, ℓ' be a pair of lines sharing a terminal v. We can transform any solution in which ℓ and ℓ' cross into a solution with fewer crossings in which ℓ and ℓ' do not cross.*

Proof. Assume that ℓ and ℓ' cross in a solution. We switch the positions of the line ends at the common terminal v between ℓ and ℓ' and reroute the two lines between the crossing's position and v. By reusing the route of ℓ for ℓ' and vice versa, the number of crossings does not increase; see Figure 4.11. On the other hand, the crossing between ℓ and ℓ' is eliminated. □

Let ℓ, ℓ' be two lines whose common subpath P starts at vertex u and ends at vertex v. Observe that terminals of ℓ and ℓ' that lie on P can only be at u or v. If neither ℓ nor ℓ' has a terminal on P then a crossing of the lines does not depend on the positions of the terminals; it only depends on how the lines split at u and v. Hence, we assume that there is at least one terminal at u or v. We model a possible crossing between ℓ and ℓ' by a 2SAT formula, the *crossing formula* of ℓ and ℓ', consisting of at most two clauses. The crossing formula evaluates to true if and only if ℓ and ℓ' do not cross. For simplicity, we assume that the top sides of the terminal ports of u and v are located on the same side of P. If this is not the case, the variable ℓ_u must be substituted by its inverse $\neg\ell_u$ in the formula. We consider four cases; see Figure 4.12 for illustrations.

(f1) Suppose that u and v are terminals for ℓ and intermediate stations for ℓ', that is, ℓ is a subpath of ℓ'; see Figure 4.12. Then, ℓ does not cross ℓ' if and only if both terminals of ℓ lie on the same side of P. This is expressed by the crossing formula

$$(\ell_u \wedge \ell_v) \vee (\neg\ell_u \wedge \neg\ell_v) \equiv (\neg\ell_u \vee \ell_v) \wedge (\ell_u \vee \neg\ell_v).$$

Note that only variables for line ℓ occur in this formula. The same formula may occur multiple times, caused by a different line ℓ'.

(f2) Suppose that u is a terminal for ℓ and an intermediate station for ℓ', and v is a terminal for ℓ' and an intermediate station for ℓ; see Figure 4.12. Then there is no crossing if and only if both terminals lie on opposite sides of P. This is expressed by the crossing formula

$$(\ell_u \wedge \neg\ell'_v) \vee (\neg\ell_u \wedge \ell'_v) \equiv (\ell_u \vee \ell'_v) \wedge (\neg\ell_u \vee \neg\ell'_v).$$

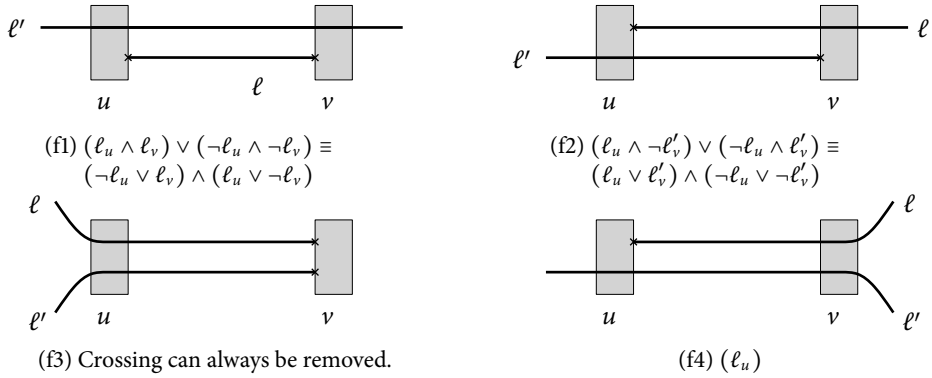


Figure 4.12: The four cases for crossing formulas.

(f3) Suppose that both ℓ and ℓ' terminate at the same vertex, say at v; see Figure 4.12. By Lemma 4.2, a solution of MLCM-P with a crossing of ℓ and ℓ' can be transformed into a solution with fewer crossings in which ℓ and ℓ' do not cross. Hence, we do not need to introduce a crossing formula in this case. Instead, we can find a line layout in which the lines possibly cross and, afterwards, improve the line layout by removing the crossing of ℓ and ℓ'.

(f4) In the remaining case, there is only one terminal of ℓ and ℓ' on P. Without loss of generality, let ℓ terminate at u; see Figure 4.12. The lines have to cross if and only if the line end of ℓ at u is placed on the wrong side of the line ℓ'. Hence, a crossing is triggered by a single variable. Depending on the fixed terminals or leaving edges at v and u, the crossing formula consists of the single clause

$$(\ell_u) \quad \text{or} \quad (\neg\ell_u).$$

Note that, like in case (f1), the same clause can occur multiple times, caused by different lines ℓ'.

4.3.2 Crossing-Free Solutions

The first thing we can do with the help of the 2SAT formulation is checking whether there exists a crossing-free solution of an MLCM-P instance. We can do this using the following algorithm. First, we check for unavoidable crossings by analyzing every pair of lines individually. Second, the 2SAT model is satisfiable if and only if there is a solution of the MLCM-P instance without avoidable crossing. Note that assigning sides to the line ends at the ports using a satisfying truth assignment of the 2SAT model could still result in crossings since we did not introduce crossing formulas in case (f3); such crossings can, however, easily be remove with the help of Lemma 4.2. Since 2SAT can be solved in linear time [EIS76], there are at most $|\mathcal{L}|^2$ crossing formulas, and we can compute the 2SAT formulation in $O(|E||\mathcal{L}|^2)$ time by checking all pairs of lines on any edge, we conclude as follows.

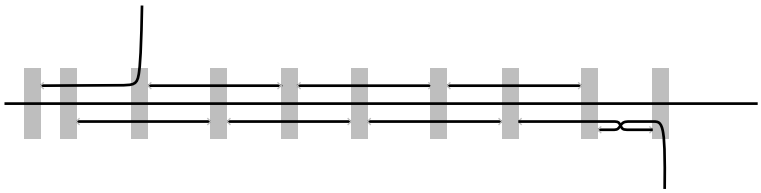


Figure 4.13: An instance of MLCM-P that does not have a crossing-free solution. There is, however, no small substructure of lines that gives a contradiction to the assumed existence of a crossing-free solution. Any proper subset of the lines allows a crossing-free solution. Note that the example can easily be extended to an arbitrary number of lines.

Theorem 4.3. *Let $(G = (V, E), \mathcal{L})$ be an instance of MLCM-P. Deciding whether there exists a crossing-free solution for $(G, \mathcal{L})$ can be accomplished in $O(|E||\mathcal{L}|^2)$ time.*

Recall that for MLCM the existence of a crossing-free solution is equivalent to the absence of unavoidable crossings; see Theorem 4.2. In contrast, instances of MLCM-P without unavoidable crossings do not always allow crossing-free solutions. Moreover, for any k, there is an instance with k lines such that any subset of $k-1$ lines admits a crossing-free solution, while the instance of all k lines requires at least one crossing; see Figure 4.13 for an example. Hence, there is no simple criterion for the existence of crossing-free solutions based on small forbidden substructures. Nevertheless, we can efficiently check whether a solution without avoidable crossings exists by using the 2SAT model.

For the special case that G is a path, Okamoto et al. [OTU13a] presented an algorithm that decides in linear time whether a crossing-free solution of MLCM-P exists.

4.3.3 Fixed-Parameter Tractability

We have seen that we can check whether an instance of MLCM-P allows a solution without crossings in polynomial time. Now, we will see that we can also check whether a solution with at most k crossings exists—if k is a constant—in polynomial time. More precisely, we will even show that MLCM-P is *fixed-parameter tractable* with respect to the number k of allowed crossings. We will use the 2SAT model for obtaining a fixed-parameter tractable algorithm. Recall that we must show that we can check in $O(f(k) \cdot \text{poly}(I))$ time whether there is a solution with at most k avoidable crossings, where f must be a computable function and I is the input size.

We want to relate crossings in a solution to unsatisfied clauses in the 2SAT formulation. In case (f4) this is easily possible because the crossing formula consists of just one clause. In cases (f1) and (f2), however, the crossing formulas consist of two clauses. By analyzing the truth assignments it is easy to see that for any truth assignment at least one of the two clauses is satisfied. Hence, we get the following observation.

Observation 4.2. *Let ℓ and ℓ' be two lines whose possible crossing is described by a crossing formula of type (f1), (f2), or (f4). For any truth assignment, the crossing formula contains exactly one unsatisfied clause if ℓ and ℓ' cross in the corresponding line layout; if the lines do not cross, then all clauses of the crossing formula are satisfied.*

Hence, minimizing the number of crossings is the equivalent to maximizing the number of satisfied clauses in the corresponding 2SAT instance. Maximizing the number of satisfied clauses, or solving the MAX-2SAT problem, is NP-hard [GJ79].

However, the problem of deciding whether it is possible to remove a given number k of m 2SAT clauses so that the formula becomes satisfiable is fixed-parameter tractable with respect to the parameter k [RO09]. This yields the following result.

Theorem 4.4. *MLCM-P is fixed-parameter tractable with respect to the number k describing the maximum number of avoidable crossings that a feasible solution may contain, with a runtime of* $O(15^k \cdot k \cdot |\mathcal{L}|^6 + |\mathcal{L}|^2|E|)$

Proof. We show that the 2SAT formula for the instance $(G = (V, E), \mathcal{L})$ of MLCM-P can be made satisfiable by removing at most k clauses if and only if there is a line layout with at most k avoidable crossings.

First, suppose that it is possible to remove at most k clauses from the 2SAT model so that there is a truth assignment satisfying all remaining clauses. Fix such a truth assignment and consider the corresponding assignment of sides to the terminals. By Observation 4.2, any crossing leads to an unsatisfied clause in the 2SAT formula, and no two crossings share an unsatisfied clause. Furthermore, only the removed clauses can be unsatisfied. Hence, we have a side assignment that causes at most k avoidable crossings.

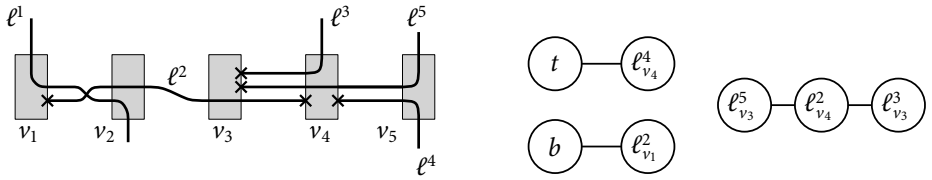
Now, we assume that there is an assignment of sides for all terminals that causes at most k crossings. By Observation 4.2, there are at most k unsatisfied clauses since any crossing just leads to a single unsatisfied clause. The removal of these clauses creates a new, satisfiable, formula.

Hence, the MLCM-P instance has a solution with at most k avoidable crossings if and only if the 2SAT formula can be made satisfiable by removing at most k clauses. By using the $O(15^k km^3)$-time algorithm for 2SAT of Razgon and O'Sullivan [RO09]—where m is the number of clauses—, we obtain a fixed-parameter algorithm for MLCM-P whose runtime is $O(15^k k|\mathcal{L}|^6 + |\mathcal{L}|^2|E|)$. □

Note that the result of Theorem 4.4 does also hold if k is the number of crossings, also counting the unavoidable crossings. We just have to determine the number k' of unavoidable crossings by comparing all pairs of lines; then, we can apply Theorem 4.4 with the number $k - k'$ of allowed avoidable crossings.

4.3.4 Approximating MLCM-P

Using insights into the 2SAT formulation that we developed in the previous sections, we can now derive an approximation algorithm for MLCM-P. The proof of Theorem 4.4 yields that the number of avoidable crossings in a crossing-minimal solution of MLCM-P equals the minimum number of clauses that we need to remove from the 2SAT formula in order to make it satisfiable. Furthermore, a set of k clauses, whose removal makes the 2SAT formula satisfiable, corresponds to an MLCM-P solution with at most k avoidable crossings. Note that we do not need to consider unavoidable crossings since they occur both in optimum and approximative solutions. Hence, an approximation algorithm for the problem of making a 2SAT formula satisfiable by removing the minimum number of clauses (also called Min 2CNF Deletion) yields an approximation for MLCM-P of the same quality. As there is an $O(\sqrt{\log m})$-approximation algorithm for Min 2CNF deletion [ACMM05], we obtain the following result.



(a) An instance $(G, \mathcal{L})$ of Proper-MLCM-P. (b) Graph G_{tb} for the instance $(G, \mathcal{L})$.

Figure 4.14: A small instance of Proper-MLCM-P. The generated 2SAT formulas are: $(\ell^2_{v_1})$ for the crossing of ℓ^1 and ℓ^2; $(\neg\ell^4_{v_4})$ for the crossing of ℓ^5 and ℓ^4; $(\ell^2_{v_4} \vee \ell^3_{v_3}) \wedge (\neg\ell^2_{v_4} \vee \neg\ell^3_{v_3})$ for the crossing of ℓ^2 and ℓ^3; $(\ell^2_{v_4} \vee \ell^5_{v_3}) \wedge (\neg\ell^2_{v_4} \vee \neg\ell^5_{v_3})$ for the crossing of ℓ^2 and ℓ^5.

Theorem 4.5. *There is an $O(\sqrt{\log|\mathcal{L}|})$-approximation algorithm for MLCM-P.*

4.4 The Problem Proper-MLCM-P

In this section, we consider the problem Proper-MLCM-P; in this variant of MLCM-P no line in $\mathcal{L}$ is a subpath of another line. First, we focus on graphs whose underlying network is a caterpillar. There, the top and bottom sides of ports are given naturally; see Figure 4.14a.

Based on the 2SAT model described in the previous section, we construct a graph $G_{tb} = (V_{tb}, E_{tb})$, which has a vertex ℓ_u for each variable of the model (and, hence, for each line end) and two additional vertices t and b, representing the top and bottom side of a port (or `true` and `false`), respectively. Since no line is a subpath of another line, our 2SAT model has only the two types of crossing formulas (f2) and (f4); compare Section 4.3.1. For case (f2), that is, the crossing formula $(\ell_u \vee \ell'_v) \wedge (\neg\ell_u \vee \neg\ell'_v)$, we create an edge (ℓ_u, ℓ'_v) in G_{tb}. The edge models a possible crossing between lines ℓ and ℓ'; that is, the lines cross if and only if ℓ terminates on top (bottom) of u and ℓ' terminates on top (bottom) of v. For a crossing formula of type (ℓ_u) (case (f4)), we add an edge (b, ℓ_u) to G_{tb}; similarly, we add an edge (t, ℓ_u) for a formula $(\neg\ell_u)$. The edges (b, ℓ_u) and (t, ℓ_u) model that there is a crossing if the line end of ℓ at u is on the bottom or on the top, respectively; see Figure 4.14b for an example.

Any truth assignment to the variables is equivalent to a b-t cut in G_{tb}, that is, a cut separating b and t. More precisely, the b-side of the partition corresponds to the `false` variables and the t-side of the partition corresponds to the `true` variables. Any edge in the graph models the fact that two variables should not be assigned to the same side as the corresponding line ends would cause a crossing otherwise. Hence, any line crossing corresponds to an *uncut* edge. Therefore, for finding a line layout with the minimum number of crossings, we need to solve the well-known MIN-UNCUT problem on G_{tb}, which is defined as follows.

Problem 4.3 (MIN-UNCUT)**.** *Given a graph $G = (V, E)$, partition the set V of vertices into two sets S_t, S_b so that the number of uncut edges (v, u) (with either $v, u \in S_t$ or $v, u \in S_b$) is minimized.*

As an additional constraint, we want that $t \in S_t$ and $b \in S_b$. In general, the problem MIN-UNCUT is NP-hard [GJ79] because optimum solutions of MIN-UNCUT are also optimum solutions of the NP-hard maximum cut problem. However, it turns out that the graph G_{tb}

has a special structure, which we call *almost bipartite*; this structure will allow us to solve MIN-UNCUT efficiently on G_{tb}.

Definition 4.1 (Almost bipartite graphs)**.** A graph $G = (V, E)$ is called *almost bipartite* if it is the union of a bipartite graph $H = (V_H, E_H)$ and two additional vertices b, t whose edges may be incident to vertices of both sides of the partition of H; that is, $V = V_H \cup \{b\} \cup \{t\}$ and $E = E_H \cup E'$, where

$$E' \subseteq \{(b, v) \mid v \in V\} \cup \{(t, v) \mid v \in V\}.$$

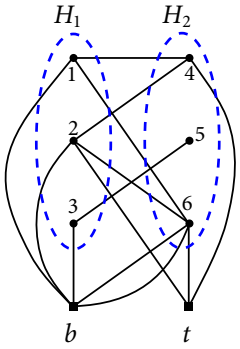
For the graph G_{tb}, the bipartition is given by the fact that "left" (similarly, "right") terminals of two lines can never be connected by an edge in G_{tb} since crossing formulas of type (f2) always involve a left and a right terminal. More formally, let L_l be the set of variables for terminals at the leftmost end of a line and let L_r be the set of of variables for terminals on the rightmost end of a line. Then $V_{tb} = L_l \dot\cup L_r \dot\cup \{b, t\}$ and the subgraph induced by $L_l \dot\cup L_r$ is bipartite with respect to the sets L_l and L_r. We now show that MIN-UNCUT can be solved optimally for almost bipartite graphs in polynomial time.

Almost bipartite graphs are a subclass of *weakly bipartite graphs*, see the work of Barahona [Bar83]. Weakly bipartite graphs have no simple combinatorial characterization. It is known that MAX-CUT and MIN-UNCUT can be solved in polynomial time on weakly bipartite graphs using the ellipsoid method [GP81]. However, the algorithm might be not fast in practice. As mentioned by Grötschel and Pulleyblank [GP81], "it remains a challenging problem to find a practically efficient method for the max-cut problem in weakly bipartite graphs which is of a combinatorial nature and does not suffer from the drawbacks of the ellipsoid method". In the following we present such an algorithm for the special case of almost bipartite graphs. Our algorithm is based on a maximum flow computation in a modified graph.

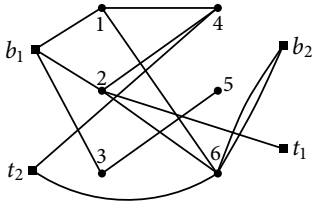
Theorem 4.6. *MIN-UNCUT can be solved in $O(n^3)$ time on almost bipartite graphs of n vertices.*

Proof. Let $G = (V, E)$ be an almost bipartite graph consisting of a bipartite graph $H = (V_H, E_H)$, two additional vertices b and t, and a set E' of edges connecting some vertices of H to b and t as in Definition 4.1. The special vertices b and t have to belong to different vertex sets S_b and S_t of G. We create a new graph $\tilde G = (\tilde V, \tilde E)$ from G as follows. First, we split vertex b into new vertices b_1 and b_2 and we split vertex t into new vertices t_1 and t_2 such that b_1 and t_2 are connected to the vertices of the first side H_1 of the partition of H, and b_2 and t_1 are connected to the second side H_2 of the partition of H. Formally, for each edge $(b, v) \in E$ with $v \in H_1$, we create an edge $(b_1, v) \in \tilde E$; for each edge $(b, v) \in E$ with $v \in H_2$, we create an edge $(v, b_2) \in \tilde E$. Similarly, edges $(v, t_1) \in E'$ are created for all $(t, v) \in E$ with $v \in H_1$, and edges $(t_2, v) \in E'$ are created for all $(t, v) \in E$ with $v \in H_2$. The construction is illustrated in Figure 4.15b.

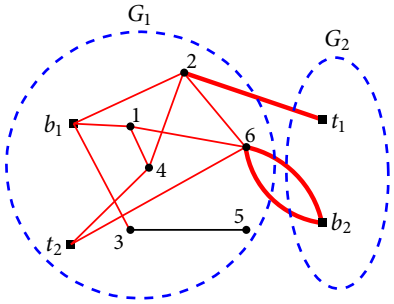
Now, for each edge $(u, v) \in \tilde E$, we assign capacity 1, and compute a maximum flow between the pair of sources b_1, t_2 and the pair of sinks b_2, t_1. After introducing a supersource (connected to b_1 and t_2) and a supersink (connected to b_2 and t_1), this can be done in $O(n^3)$ time by using the maximum flow algorithm of Edmonds and Karp. Note that we find an integral maximum flow in $\tilde G$ because all capacities are integers.



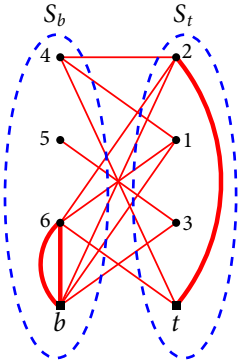
(a) The almost bipartite input graph.



(b) The modified graph $\tilde{G}$.



(c) A minimum cut in $\tilde{G}$ computed using a maximum flow. The edges crossing the cut are bold red.



(d) A cut with the minimum number of uncut edges (bold red) based on the minimum cut in $\tilde{G}$.

Figure 4.15: Solving MIN-UNCUT on an almost bipartite graph. The maximum flow (minimum cut) with value 3 results in vertex partitions $V_b^1 = \{b_1, 4, 5, 6\}$, $V_t^1 = \{t_2, 1, 2, 3\}$, $V_b^2 = \{b_2\}$, and $V_t^2 = \{t_1\}$. The optimal partition $S_b = \{b, 4, 5, 6\}, S_t = \{t, 1, 2, 3\}$ induces three uncut edges $(b, 6)$, $(b, 6)$, and $(t, 2)$.

A maximum flow corresponds to a maximum set of edge-disjoint paths starting at b_1 or t_2 and ending at b_2 or t_1. Such a path corresponds to one of the following structures in the original graph G since b_1 and b_2 correspond to b and t_1 and t_2 correspond to t:

(i) an odd cycle containing vertex b (that is, a cycle with an odd number of edges),

(ii) an odd cycle containing vertex t, or

(iii) an even path between vertices b and t (that is, a path with an even number of edges).

Note that, if a graph has an odd cycle, then at least one of the edges of the cycle is uncut in any solution of MIN-UNCUT. The same holds for an even path connecting b and t in G because b and t have to belong to different sides of the partition. Since the maximum flow corresponds to the edge-disjoint odd cycles and even paths in G, the value of the flow is a lower bound for a solution of MIN-UNCUT.

We now want to prove that the value of the maximum flow in $\tilde{G}$ is also an upper bound by showing how to construct a partition of V into S_b and S_t with $b \in S_b$ and $t \in S_t$ such that the number of uncut edges is equal to the value of the maximum flow. By Menger's theorem, the value of the maximum flow in $\tilde{G}$ is the cardinality of the minimum edge cut separating sources and sinks. Let E^* be the minimum edge cut and let G_1 and G_2 be the corresponding disconnected subgraphs of $\tilde{G}$, where $b_1 \in G_1$ and $b_2 \in G_2$; see Figure 4.15c. The graph G_1 is bipartite since $H \cap G_1$ is bipartite; vertex b_1 is only connected to vertices of H_1 and vertex t_2 is only connected to vertices of H_2. Therefore, there is a 2-partition of vertices of G_1 such that b_1 and t_2 belong to different sides of the partition; let us denote the sides by V_b^1 and V_t^1. Similarly, there is a 2-partition of G_2 into V_b^2 and V_t^2 with $b_2 \in V_b^2$ and $t_1 \in V_t^2$. We combine these partitions so that $S_b = \{b\} \cup \left(V_b^1 \cup V_b^2\right) \setminus \{b_1, b_2, t_1, t_2\}$ and $S_t = \{t\} \cup \left(V_t^1 \cup V_t^2\right) \setminus \{b_1, b_2, t_1, t_2\}$. The sets S_b and S_t form the required partition of V for MIN-UNCUT; see Figure 4.15d. The set of uncut edges is the transformation of E^* back to the original graph G, which completes the proof. □

As a direct corollary, we get a polynomial-time algorithm for Proper-MLCM-P on caterpillars: We first build the 2SAT model and then, by analyzing the crossing formulas, we build the graph G_{tb}. By applying Theorem 4.6 to G_{tb}, we find a partition into S_b and S_t for the line ends. By assigning line ends in S_b to the bottom side and line ends in S_t to the top side of the respective port, we get a solution for Proper-MLCM-P, which—after removing potential crossings of type (f3) with the help of Lemma 4.2—is crossing-minimal.

We can, actually, also use the same method for a larger set of instances. The only special property of caterpillars that we used was that there is a meaning of left and right line ends and of the top and bottom side of ports that is consistent over all pairs of lines. We now relate this property to the directions of lines.

Definition 4.2. Let $G = (V, E)$ be a graph and let $\mathcal{L}$ be a set of lines on G. We say that the lines allow *consistent line directions* on G if each line can be directed so that for each edge $e \in E$ all lines $\ell \in L_e$ on this edge have the same direction.

If the underlying graph is a path then we can consistently direct the lines from left to right. Similarly, consistent line directions exist for *left-to-right trees*, which have been considered by Bekose et al. [BKPS08] and by Argyriou et al. [ABKS10] for metro-line crossing minimization;

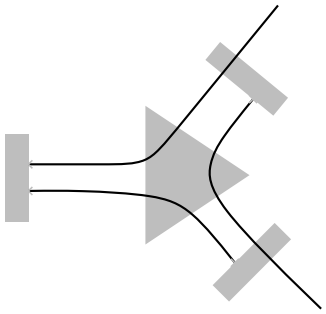


Figure 4.16: An example that does not allow consistent line directions.

they are also called *upward trees*; see also Section 5.4.2. Upward trees are trees for which there is an embedding with all lines being monotone in some direction. Note that not all trees allow consistent line directions; see Figure 4.16. Furthermore, there are also instances whose underlying graph is not a tree that still allow consistent line directions. This is the case, for example, if the graph is a simple cycle.

Given a graph $G = (V, E)$ with a set $\mathcal{L}$ of lines, it is easy to test whether there are consistent line directions—and finding such directions if they exist. This can be done by simply giving an arbitrary direction to some first line, and then applying the same direction on all lines sharing edges with the first line until all lines have directions or an inconsistency is found.

Given an instance with consistent line directions, we naturally get a left and a right terminal for any line; any line starts at its left terminal and ends at its right terminal. Similarly, the top and bottom side of a port refer to the side if we direct the respective edge so that the lines go from left to right. Using this idea, we can now show that we can solve Proper-MLCM-P optimally on any instance that allows consistent line directions.

Theorem 4.7. *Let $G = (V, E)$ be an embedded graph with a set $\mathcal{L}$ of lines. If the instance $(G, \mathcal{L})$ admits consistent line directions, then Proper-MLCM-P can be solved in $O(|\mathcal{L}|^2(|\mathcal{L}|+|E|))$ time.*

Proof. Given consistent line directions, we assign top and bottom sides of each port as follows. Consider a port of $u \in V$ corresponding to an edge $(u, v) \in E$. Let π_{uv} be the order of the lines at the port, with $\pi_{uv} = (\ell_1 \dots \ell_p \dots \ell_q \dots \ell_{|L_{uv}|})$ $(p \leq q)$, where u is a terminal for the lines $\ell_1, \dots, \ell_p$ and for the lines $\ell_q, \dots, \ell_{|L_{uv}|}$ and u is an intermediate station for the lines $\ell_{p+1}, \dots, \ell_{q-1}$. We say that the lines $\ell_1, \dots, \ell_p$ terminate at the top side of the port if the lines L_{uv} are directed from u to v; otherwise, $\ell_1, \dots, \ell_p$ terminate at the bottom side of the port. Analogously, $\ell_q, \dots, \ell_{|L_{uv}|}$ terminate at the bottom side of the port if the lines are directed from u to v and at the top side otherwise.

Now, we consider a pair of lines ℓ and ℓ' that have a common subpath P starting at vertex u and ending at vertex v. It is easy to see that the top sides of the terminal ports of u and v are located on the same side of P. Hence, in our 2SAT model, we have only crossing formulas of the type $(\ell_u \vee \ell'_v) \wedge (\neg\ell_u \vee \neg\ell'_v)$ with variables ℓ_u and ℓ'_v and no other combinations of literals—apart from clauses consisting of a single literal. Therefore, the graph G_{tb} contains an edge (ℓ_u, ℓ'_v) for the pair of lines.

Now, we can show that G_{tb} is almost bipartite. To this end, we prove that there is no odd cycle containing only vertices ℓ_u with $u \in V$, $\ell \in \mathcal{L}$ and neither b nor t. Suppose there is such a cycle C. Let ℓ_u^1 and ℓ_v^2 be two neighboring vertices of C; we know that the common subpath P of ℓ^1 and ℓ^2 starts at vertex u and ends at vertex v in G. We may assume, without loss of generality, that the lines are directed from u to v. Consider the port at u corresponding to the first edge (u, u_1) of P. The direction of the lines is *aligned* with the port; that is, due to the consistent line directions, the lines are directed from u to u_1. We then also say that the line end of ℓ^1 at u is its *left* end. Now, consider the port at v corresponding to the last edge (v, v_1) of P. Here, the direction of the lines is *opposite* to the port; that is, the lines are directed from v_1 to v. The line end of ℓ^2 at v is its *right end*. It is easy to see that, for the next line end described by the vertex ℓ_w^3 in the cycle C, the direction of the lines is again aligned with the corresponding port of w, and the line end is a left end. Moreover, for every line ℓ^{2k+1} the corresponding port is aligned with the direction of lines, and for every ℓ^{2k} the direction of lines is opposite to the port. Hence, there cannot exist an odd cycle C.

Now we have seen that G_{tb} is almost bipartite. Therefore, we can apply Theorem 4.6 for solving MIN-UNCUT on G_{tb}. The cut then gives rise to an optimum solution of Proper-MLCM-P for the instance $(G, \mathcal{L})$.

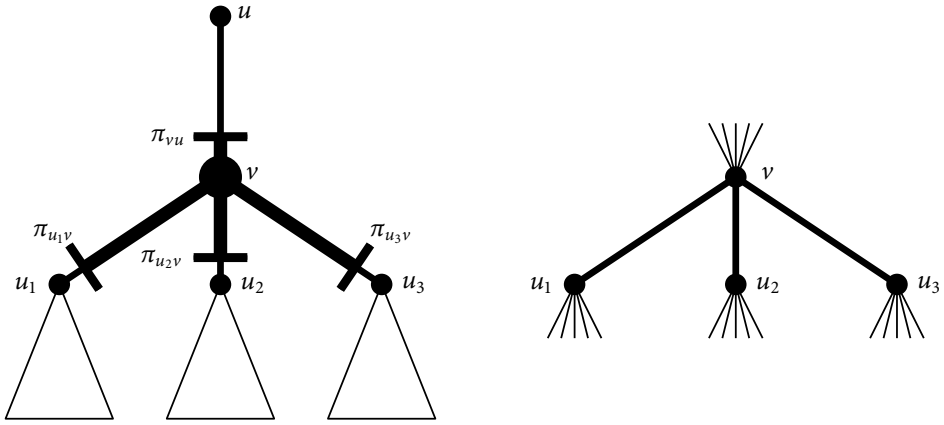
Recall that building the 2SAT clauses needed $O(|\mathcal{L}|^2|E|)$ time. Within the same time, we can build the graph G_{tb}. Since G_{tb} has $O(|\mathcal{L}|)$ vertices, we then find optimum side assignments in $O(|\mathcal{L}|^3)$ time by Theorem 4.6. Hence, the total runtime is $O(|\mathcal{L}|^2(|\mathcal{L}| + |E|))$. □

4.5 MLCM with Bounded Maximum Degree and Edge Multiplicity

We now introduce two additional restrictions for metro-line crossing minimization. First, we consider instances in which the *maximum degree* Δ of a station is bounded by some constant. Second, we assume that on any edge e, there is at most a constant number c of lines, that is, $|L_e| \leq c$; we say that c is the maximum *edge multiplicity*. For metro maps both restrictions are realistic: In the popular octilinear drawing style, the maximum possible degree is 8. Furthermore, even in huge metro networks, edges that are served by more than 10 lines are unlikely to occur, as Nöllenburg [Nöl09] pointed out.

We now show that the restricted problem variant of both MLCM and MLCM-P can be solved in polynomial time if the underlying network is a tree. We first focus on MLCM; we will then see that the results can easily be extended to MLCM-P.

We develop a dynamic program that solves MLCM on instances whose underlying network is a tree. First, we root the tree $T = (V, E)$ at some arbitrary leaf r. Let $v \in V \setminus \{r\}$, and let u be the parent node of v. We say that a line *contributes* to the subtree $T[v]$ if at least one of its terminals is a vertex of $T[v]$; the line *leaves* the subtree if one of its terminals is in $T[v]$ and the other one is outside. Any line that leaves the subtree $T[v]$ passes through the edge $e = (u, v)$. If we fix the order π_{vu} of L_e at the port of v corresponding to the edge e, an optimum solution for $T[v]$ is independent of an optimum solution for the remaining graph; in other words, we can combine any optimum solution for $T[v]$ resulting in the order π_{vu} with any optimum solution for the remaining graph resulting in the same order π_{vu}. Let $\mathrm{cr}[v, \pi_{vu}]$ be the number of crossings in an optimum solution for $T[v]$ that results in the order π_{vu} at node v on the edge (u, v). If there



(a) Subtree $T[v]$; the remaining instance, which is bounded by the orders π_{vu}, π_{u_1v}, π_{u_2v}, and π_{u_3v} is drawn bold.

(b) Remaining instance of constant size for the subtree; permutations are replaced by edges leaving in the right order.

Figure 4.17: Computation of $\mathrm{cr}[v, \pi_{vu}]$ for a subtree $T[v]$ in the dynamic program.

is no feasible solution, that is, if any solution for π_{vu} has avoidable vertex crossings, then we let $\mathrm{cr}[v, \pi_{vu}] = \infty$.

If v is a leaf, then, for any order π_{vu}, $\mathrm{cr}[v, \pi_{vu}] = 0$. Now, suppose that v has children $u_1, \ldots, u_k$, with $k < \Delta$. For computing the value $\mathrm{cr}[v, \pi_{vu}]$, we test all combinations of permutations π_{u_iv} for u_i with $i = 1, \ldots, k$; see Figure 4.17a. Given such permutations, we can combine optimum solutions for the subtrees $T[u_1], \ldots, T[u_k]$ resulting in orders $\pi_{u_1v}, \ldots, \pi_{u_kv}$ with an optimum solution for the remaining instance, which consists of the edges $(u_1, v), \ldots, (u_k, v)$ and is described by the orders $\pi_{u_1v}, \ldots, \pi_{u_kv}$ and π_{vu}; see the bold region in Figure 4.17a and the transformed instance shown in Figure 4.17b. Let $f(v, \pi_{vu}, \pi_{u_1v}, \ldots, \pi_{u_kv})$ be the number of crossings in an optimum solution of this remaining instance; note that this value can be computed in constant time because the remaining instance has only constant size. Then,

$$\mathrm{cr}[v, \pi_{vu}] = \min_{\pi_{u_1v}, \ldots, \pi_{u_kv}} \left(f(v, \pi_{vu}, \pi_{u_1v}, \ldots, \pi_{u_kv}) + \sum_{i=1}^{k} \mathrm{cr}[u_i, \pi_{u_iv}] \right).$$

Note that $f(v, \pi_{vu}, \pi_{u_1v}, \ldots, \pi_{u_kv}) = \infty$ if the permutations lead to an infeasible solution with avoidable vertex crossings. The table $\mathrm{cr}[\cdot, \cdot]$ has at most $n \cdot c! = O(n)$ entries, each of which can be computed in constant time. Hence, we get the following theorem.

Theorem 4.8. *MLCM can be solved optimally in $O(n)$ time on tree instances of maximum degree Δ and maximum edge multiplicity c if both Δ and c are constants.*

We now want to analyze the running time for computing an entry $\mathrm{cr}[v, \pi_{vu}]$ more precisely. First, there are at most $(c!)^{\Delta-1}$ combinations for the orders $\pi_{u_1v}, \ldots, \pi_{u_kv}$. Second, for computing $f(v, \pi_{vu}, \pi_{u_1v}, \ldots, \pi_{u_kv})$, we can try all combinations for the orders on the edges around v. If such a combination leads to a feasible solution, we can solve each edge—as a permutation of

constant size—individually. The number of crossing on an edge (u_i, v) is exactly the number of pairs of lines whose order changes between the two ports of the edge. This number can easily be computed in $O(c^2)$ time.

Overall, evaluating $f(v, \pi_{vu}, \pi_{u_1 v}, \ldots, \pi_{u_k v})$ is then possible in $O((c!)^{\Delta-1}\Delta c^2)$ time. The total time for finding an optimum solution is, hence, $O(nc! \cdot (c!)^{\Delta-1} \cdot (c!)^{\Delta-1}\Delta c^2) = O(n(c!)^{2\Delta-1}\Delta c^2)$. As the parameters c and Δ are well-separated from n, we can conclude as follows.

Theorem 4.9. *MLCM is fixed-parameter tractable on tree instances with respect to the parameter $c + \Delta$, where Δ is the maximum degree and c is the maximum edge multiplicity. The problem can be solved in $O(n(c!)^{2\Delta-1}\Delta c^2)$ time.*

It is now easy to see that the algorithm can be adapted to MLCM-P. The only necessary change is to enforce the periphery condition for all orders on ports. To this end we just restrict all permutations used in the table and in the computation of values to this property. Hence, we get the following theorem.

Theorem 4.10. *MLCM-P can be solved optimally in $O(n)$ time on tree instances of maximum degree Δ and maximum edge multiplicity c if both Δ and c are constants.*

Checking whether the used permutations are allowed is easy and does not change the runtime. Hence, we also get fixed-parameter tractability for MLCM-P with the same runtime as for MLCM.

Theorem 4.11. *MLCM-P is fixed-parameter tractable on tree instances with respect to the parameter $c + \Delta$, where Δ is the maximum degree and c is the maximum edge multiplicity. The problem can be solved in $O(n(c!)^{2\Delta-1}\Delta c^2)$ time.*

Improved Runtime. With a little more effort, we can improve the runtime for the fixed-parameter algorithms for MLCM and MLCM-P. So far, the table cr contained, for the edge (u, v) connecting vertex v to its parent u, an entry of the form $\mathrm{cr}[v, \pi_{vu}]$ that represent the minimum number of crossings in a subtree for a fixed order on the port connecting the subtree $T[v]$ to the rest of the graph. Now, we additionally store an entry $\mathrm{cr}[u, \pi_{uv}]$ where π_{uv} is the order of lines at the port of u on the edge (u, v). The value $\mathrm{cr}[u, \pi_{uv}]$ describes the minimum number of crossings in a feasible solution for MLCM (or MLCM-P, respectively) in the subtree $T[v]$ and on the edge (u, v) connecting $T[v]$ to u, given the order π_{uv}.

For computing an entry $\mathrm{cr}[u, \pi_{uv}]$, we try all possible (feasible) orders at the port of v on the edge (u, v); the entry is determined by the feasible order π_{vu} that minimizes the sum of the number of crossings on the edge (u, v) and the crossings $\mathrm{cr}[v, \pi_{vu}]$ in the subtree $T[v]$.

We also need to modify the computation of entries of the original type $\mathrm{cr}[v, \pi_{vu}]$. Instead of using other entries of the same type as we did before, we now use the new entries corresponding to the other ports of v that point downwards into the subtrees. If these orders are fixed, the remaining instance now consists only of the vertex v. We just have to distinguish whether the orders lead to avoidable vertex crossings in v. More precisely, for $k < \Delta$, let $u_1, \ldots, u_k$ be the children of v and let $\pi_{vu_1}, \ldots, \pi_{vu_k}$ be orders of the lines on edges $(u_1, v), \ldots, (u_k, v)$ at the respective ports of v; see Figure 4.18. Then,

$$\mathrm{cr}[v, \pi_{vu}] = \min_{\pi_{vu_1}, \ldots, \pi_{vu_k}} \left(f'(v, \pi_{vu}, \pi_{vu_1}, \ldots, p_{vu_k}) + \sum_{i=1}^{k} \mathrm{cr}[v, \pi_{vu_i}] \right)$$

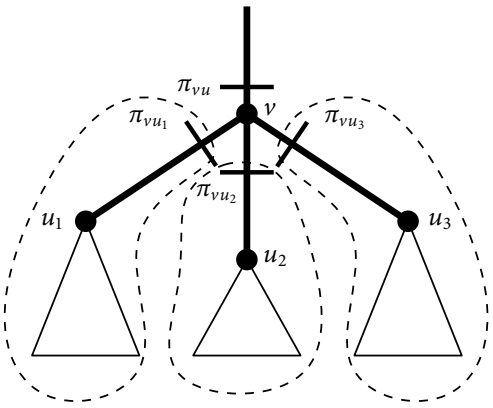


Figure 4.18: Computation of $\mathrm{cr}[v, \pi_{uv}]$. The recursion is done using values $\mathrm{cr}[v, \pi_{vu_i}]$ with $i = 1, \ldots, k$, which describe optimum solutions for subtrees including the edges (u_1, v) connecting the subtrees to v.

The numbers of crossings on the edges incident to v are already counted by the entries $\mathrm{cr}[v, \pi_{vu_1}]$, …, $\mathrm{cr}[v, \pi_{vu_k}]$. Hence, with f' we just have to check whether the orders of lines around v are feasible; that is, $f'(v, \pi_{vu}, \pi_{vu_1}, \ldots, p_{vu_k}) = 0$ if the orders π_{vu} and $\pi_{vu_1}, \ldots, p_{vu_k}$ do not lead to avoidable vertex crossings. For MLCM-P, the individual orders must additionally satisfy the periphery condition. Otherwise, let $f'(v, \pi_{vu}, \pi_{vu_1}, \ldots, p_{vu_k}) = \infty$.

In the improved version, the only thing we have to do when evaluating f' is checking all pairs of lines with a common edge for possible vertex crossings, which we can do in $O(\Delta c^2)$ time. As there are at most $O((c!)^{\Delta-1})$ combinations for the orders $\pi_{vu_1}, \ldots, \pi_{vu_k}$, we can compute any entry of the table cr in $O((c!)^{\Delta-1}\Delta c^2)$ time. As there are still only $O(nc!)$ such entries, the total runtime of the improved dynamic program is $O(n(c!)^{\Delta}\Delta c^2)$.

Theorem 4.12. *MLCM and MLCM-P are fixed-parameter tractable on tree instances with respect to the parameter $c + \Delta$, where Δ is the maximum degree and c is the maximum edge multiplicity. The problem can be solved in $O(n(c!)^{\Delta}\Delta c^2)$ time.*

Note that for the special case of paths, the maximum degree is $\Delta = 2$ and, hence, constant. We get a fixed-parameter algorithm with respect to the only parameter c. The algorithm runs in $O(n(c!)^2 c^2)$ time. Since only MLCM-P is NP-hard on paths, while MLCM is trivial, this makes only sense for MLCM-P. However, for the special case of MLCM-P on paths, Okamoto et al. [OTU13a] presented a specialized fixed-parameter algorithm with a runtime of $O(n2^c c^3)$.

4.6 Practical Considerations on Metro-Line Crossing Minimization

Not all real-world transportation network meet the requirements implied by the models for MLCM and MLCM-P that we used following previous work. For example, lines are not necessarily simple paths as many metro maps have circular or tree-like lines. Thus, the existing

algorithms cannot be applied. Furthermore, both MLCM and MLCM-P are NP-hard even for very simple underlying graphs. Therefore, we propose two approaches to overcome these obstacles for practical purposes; the approaches also point to possible directions for future work.

Line Simplification and Insertion. In many metro networks, there are just few lines that are no simple paths. We suggest first creating a simplified instance with the desired properties by deleting few (parts of) lines. Then, after obtaining a solution for the simplified instance, the deleted parts can be reinserted with as few crossings as possible. We will show that a single line can be inserted into an existing line layout optimally with respect to the number of newly introduced crossings. A number of extensions is possible in this direction. For example, how can we find a good set of edges and lines whose removal results in a simplified instance? Furthermore, an algorithm that inserts several lines optimally into a given solutions would also be helpful.

Optimal Insertion of a Line into an Existing Solution. We now explore a simple heuristic for computing line orders. The heuristic works iteratively by inserting lines into an existing order. Let $\ell_1, \ldots, \ell_{|\mathcal{L}|}$ be the input lines. The heuristic consists of $|\mathcal{L}|$ iterations; in iteration i line ℓ_i is inserted into the current line orders for lines $\ell_1, \ldots, \ell_{i-1}$. This can be done optimally, that is, the number of crossings that we introduce in a single step is minimum with respect to the previous line layout. Note that this does not mean that the heuristic is globally optimal since accepting additional crossings in one iteration may save more crossings later.

Lemma 4.3. *Let $G = (V, E)$ be an embedded graph, let $\mathcal{L}$ be a set of lines on G, and let π_{uv} be a fixed order of the lines for each $(u, v) \in E$. There is an $O(k|\mathcal{L}|^2)$-time algorithm for inserting a line $\ell \notin \mathcal{L}$ with k vertices into the existing line layout so that the number of newly introduced crossings is minimized.*

Proof. Let $\ell = (v_1, v_2, \ldots, v_k)$ with $v_i \in V$ for $i = 1, \ldots, k$. We need to find positions in the permutations $\pi_{v_1v_2}, \pi_{v_2v_1}, \pi_{v_2v_3}, \ldots, \pi_{v_kv_{k-1}}$ for line ℓ such that the resulting line layout is both feasible and crossings-minimal. To this end, we create a directed graph $H = (U, E')$. The vertices of H contain all possible positions for ℓ in the permutations of ports; we will add an edge between two positions if the respective ports lie on the same edge or at the same vertex. The idea is to rely the problem of finding positions for ℓ to finding a shortest path in H; see Figure 4.19 for a sketch.

More precisely, we first create two special vertices $s, t \in U$. Additionally, we create vertices as follows. Consider an edge (v_i, v_{i+1}) that is traversed by line ℓ. The current order of lines on the ports of v_i and v_{i+1} corresponding to (v_i, v_{i+1}) contains the lines $L_{v_iv_{i+1}}$. Let $h = |L_{v_iv_{i+1}}|$. Then both ports contain $h + 1$ possible positions for inserting line ℓ. We number the positions from 1 to $h + 1$ and create a vertex $\pi^j_{v_iv_{i+1}} \in U$ for each position in the order $\pi_{v_iv_{i+1}}$ at the port of vertex v_i and a vertex $\pi^j_{v_{i+1}v_i} \in U$ for each position in the order $\pi_{v_{i+1}v_i}$ at the port of vertex v_{i+1}, with $1 \le j \le h + 1$. Let V_i^+ be the set of vertices modeling positions in the order $\pi_{v_iv_{i+1}}$ and let V_{i+1}^- be the set of vertices modeling positions in the order $\pi_{v_{i+1}v_i}$.

We now create edges as follows. First, for each vertex $u \in V_1^+$ we create an edge $(s, u) \in E'$ and for each vertex $u' \in V_k^-$ we create an edge $(u', t) \in E'$. The length of all these edges is 0. Second, let $u_j \in V_i^-$ and $u_{j'} \in V_i^+$. A possible edge between u_j and $u_{j'}$ models the transition of ℓ from position j at the port of (v_{i-1}, v_i) at vertex v_i to position j' at the port of (v_i, v_{i+1}) at

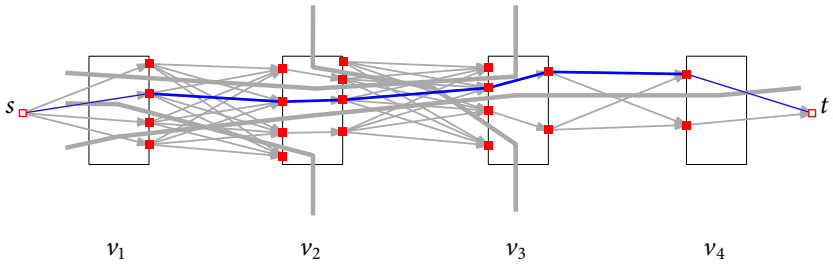


Figure 4.19: Insertion of a line $\ell = (v_1, v_2, v_3, v_4)$ (blue) into an existing line layout. Vertices of the graph H are shown as red boxes; the edges of H are indicated in gray.

vertex v_i. Such a transition must not create unavoidable vertex crossings. Hence, we create the edge $(u_j, u_{j'})$ of length 0 if the transition does not result in an avoidable crossing. Finally, we consider a pair of vertices $w_j \in V_i^+$ and $w_{j'} \in V_{i+1}^-$. We create an edge $(w_j, w_{j'})$ that models the transition of ℓ from position j in the port of (v_i, v_{i+1}) at vertex v_i to position j' at the port of (v_i, v_{i+1}) at vertex v_{i+1}. Any such transition is feasible. The transition may, however, result in (new) crossings. We set the length of the edge $(w_j, w_{j'})$ to be the number of new crossings for this transitions; it is easy to determine this number.

Now, any feasible line layout corresponds to an s-t path in H via the positions chosen for ℓ in the different orders; vice versa, any simple s-t path corresponds to a feasible line layout. Furthermore, the number of crossings in a feasible solution is equal to the length of the corresponding s-t path. Hence, minimizing the number of additional crossings is equivalent to finding a shortest s-t path. Since H is acyclic, this can be done in linear time with respect to the graph H. There are $O(k|\mathcal{L}|)$ vertices and $O(k|\mathcal{L}|^2)$ edges in H. Hence, we can find a shortest path in $O(k|\mathcal{L}|^2)$ time. □

The solution found by the previous algorithm is, of course, feasible for MLCM. For inserting a line with the minimum number of additional crossings so that the solution is also feasible for MLCM-P, a simple modification suffices: Instead of creating a vertex for any position in the order of a port, we just create a vertex for any such position at which line ℓ may be inserted without violating the periphery condition.

Fixed-Parameter Algorithms. Both MLCM and MLCM are NP-hard; we were, however, able to construct fixed-parameter algorithms. In Section 4.3.1, we presented a fixed-parameter algorithm for MLCM-P with respect to the maximum number of crossings. In the previous Section, we presented fixed-parameter algorithms for MLCM and MLCM-P on tree instances with respect to the maximum degree and edge multiplicity. Designing such an algorithm for general graphs is an interesting open problem. Furthermore, it is unclear whether the dependency on the degree is actually necessary. In contrast, even for very small constant maximum degrees both variants certainly remain NP-hard if the edge multiplicity is unbounded: the problems MLCM-P and MLCM are NP-hard even for paths and caterpillars of maximum degree 6, respectively. Possibly faster fixed-parameter algorithms with respect to just the edge

multiplicity are worth to be constructed. Another open question is whether also MLCM is fixed-parameter tractable with respect to the number of crossings.

Crossing Distribution. So far, the focus has been on minimizing the number of crossings and not on the visualization of crossings, although two line orders with the same crossing number may look quite differently; see also the following chapter. Therefore, an important practical problem is the visual representation of computed line crossings. In our opinion, crossings of lines should preferably be close to the end of their common subpath as this makes it easier to recognize that the lines do cross. It is not always possible to find an optimal solution in which every pair of lines crosses at the end of their common subpath as Pupyrev et al. observed [PNBH12]. It would be interesting to find a solution with a small number of crossings and a reasonable distribution of crossings.

For making a metro line easy to follow the important criterion is the number of its bends. Hence, an interesting question is how to sort metro lines using the minimum total number of bends. Bereg et al. [BHNP13] studied this problem for the case of a single edge.

4.7 Concluding Remarks

In this chapter, we studied several variants of metro-line crossing minimization. As a main result, we proved that the general problem version MLCM is NP-hard. For MLCM-P, we presented an $O(\sqrt{\log|\mathcal{L}|})$-approximation algorithm, as well as an exact $O(|\mathcal{L}|^2(|\mathcal{L}|+|E|))$-time algorithm for Proper-MLCM-P on instances with consistent line directions. We also developed simple polynomial-time algorithms for checking for the existence of crossing-free solutions for MLCM and MLCM-P. For instances whose underlying graph is a tree, we developed fixed-parameter algorithms for both MLCM and MLCM-P. The parameter of these algorithms is the combination of the maximum degree and the maximum edge multiplicity.

Open Problems. From a theoretical point of view, there are still many interesting open problems; the most important ones are the following.

1. Is there an approximation algorithm for MLCM?
2. Is there a constant-factor approximation algorithm for MLCM-P?
3. What is the complexity status of Proper-MLCM/Proper-MLCM-P in general, that is, for instances without consistent line directions? Note that both in our hardness proof for MLCM and in the hardness proof of Argyriou et al. [ABKS10] for MLCM-P, many lines that are subpaths of other lines are used.

On the practical side, we have already discussed several problems; see Section 4.6. The most important are the development of practically usable algorithms and algorithms that take also the distribution of crossings into account. We will work on the latter problem in the next chapter.

Chapter 5

Ordering Metro Lines by Block Crossings

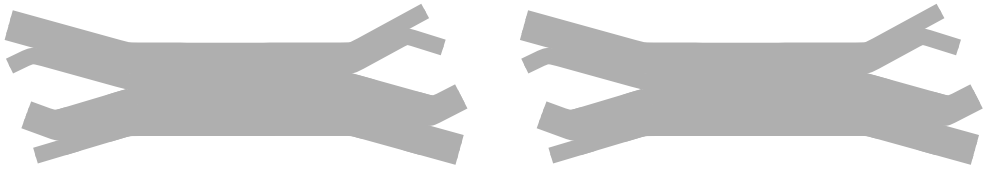
In the previous chapter, we considered metro-line crossing minimization, that is, the problem of ordering the metro lines a drawing of a metro network so that the total number of crossings between metro lines is minimized. However, not all solutions with the same number of crossings are visually equivalent. For improving the readability of metro maps, we suggest merging single crossings into *block crossings*, that is, crossings of two neighboring groups of consecutive lines; see Figure 5.1

Unfortunately, minimizing the total number of block crossings is NP-hard even for very simple graphs, which follows from a result on sorting permutations by a certain type of operation. We give approximation algorithms for special classes of graphs and an asymptotically worst-case optimal algorithm for block crossings on general graphs. Furthermore, we show that the problem remains NP-hard on planar graphs even if both the maximum degree and the number of lines per edge are bounded by constants; on trees, this restricted version becomes tractable.

5.1 Introduction

As mentioned above, in metro-line crossing minimization the focus has, so far, been on the number of crossings of lines and not on their visualization; two line orders with the same crossing number may, however, look quite differently; see Figure 5.1.

Our aim is to improve the readability of metro maps by computing line orders that are aesthetically more pleasing. To this end, we merge *pairwise* crossings into crossings of blocks of lines minimizing the number of *block crossings* in the map. Informally, a block crossing is an intersection of two neighboring groups of consecutive lines sharing the same edge; see Figure 5.1b. We consider two variants of the problem. In the first variant, we want to find a line ordering



(a) 12 pairwise crossings. (b) 3 block crossings.

Figure 5.1: Optimal orderings of a metro network.

with the minimum number of block crossings. In the second variant, we want to minimize both pairwise and block crossings.

As mentioned before, metro-line crossing minimization also arises as a subproblem in edge bundling. There, many edges of the graph can be drawn close together like metro lines. Hence, in the corresponding metro-line crossing minimization instance, there can be edges with many lines—significantly more than in real-world metro maps, where usually not more than ten lines appear on a common edge. Hence, block crossings can also greatly improve the readability of bundled graph drawings.

Problem Definition. The input for our problem is the same as for general metro-line crossing minimization—compare Section 4.1—that is, we are given an embedded underlying graph $G = (V, E)$ and a set $\mathcal{L} = \{\ell_1, \ldots, \ell_{|\mathcal{L}|}\}$ of lines in G.

For $i \leq j < k$, a *block move* (i, j, k) on the sequence $\pi = [\pi_1, \ldots, \pi_n]$ of lines on e is the exchange of two consecutive blocks $\pi_i, \ldots, \pi_j$ and $\pi_{j+1}, \ldots, \pi_k$. Interpreting $e = (u, v)$ directed from u to v, we are interested in *line orders* $\pi^0(e), \ldots, \pi^{t(e)}(e)$ on e, so that $\pi^0(e)$ is the order of lines L_e at the beginning of e (that is, at the port of vertex u corresponding to the edge e), $\pi^{t(e)}(e)$ is the order at the end of e (that is, at the port of vertex v corresponding to e), and each $\pi^i(e)$ is an ordering of L_e so that $\pi^{i+1}(e)$ is derived from $\pi^i(e)$ by a block move. If $t + 1$ line orders with these properties exist, we say that there are t *block crossings* on edge e.

Recall that, the order of the lines at a port of a vertex is always relative to this vertex. Hence, seen as a permutation, the order $\pi^{t(e)}$ is actually the reversed order of the port of v corresponding to e. Furthermore, we stress that the output, that is, the *line orders*, replace the old output, that is, the order at the ports; the line orders implicitly contain the orders at the ports.

As in the previous chapter, we use the *edge crossings* model, that is, we do not hide crossings under station symbols if possible. Recall that two lines sharing at least one common edge either do not cross or cross each other on an edge but never in a vertex.

As for MLCM and MLCM-P, unavoidable vertex crossings are allowed and not counted as they exist in any solution.

The *block crossing minimization* (BCM) problem is defined as follows.

Problem 5.1 (BCM). *Let $G = (V, E)$ be an embedded graph and let $\mathcal{L}$ be a set of lines on G. For each edge $e \in E$, find line orders $\pi^0(e), \ldots, \pi^{t(e)}(e)$ that yield a feasible solution of MLCM such that the total number of block crossings, $\sum_{e \in E} t(e)$, is minimum.*

In this chapter, we restrict our attention to instances with two properties. First, as in the previous chapter, we assume the *path intersection property*, that is, two lines share at most one common subpath. Second, any line terminates at nodes of degree one and no two lines terminate at the same node (*path terminal property*). Recall that normal metro-line crossing minimization can be solved in linear time on such instances—as Pupyrev et al. [PNBH12] showed—which are also instances of MLCM-PA.

If both properties hold, a pair of lines either has to cross, that is, a crossing is *unavoidable*, or it can be kept crossing-free, that is, a crossing is *avoidable*; see Figure 5.2. The orderings that are optimal with respect to pairwise crossings are exactly the orderings that contain just unavoidable crossings (Lemma 2 in the paper of Nöllenburg [Nöll10]); that is, any pair of lines crosses at most once, in an equivalent formulation. Intuitively, double crossings of lines can easily be

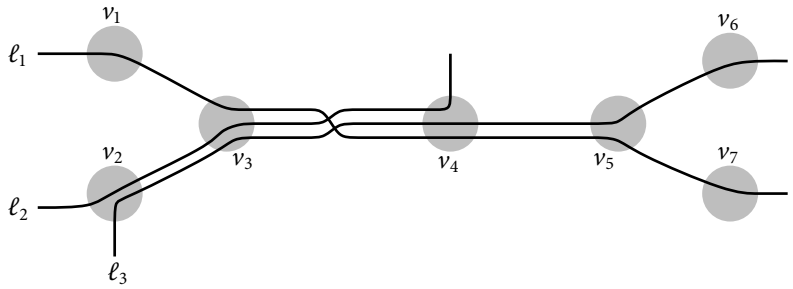


Figure 5.2: Lines ℓ_1 and ℓ_2 have an unavoidable edge crossing on the edge (v_3, v_4). The unavoidable crossing of ℓ_1 and ℓ_3 could also be placed on (v_4, v_5). An avoidable crossing of ℓ_2 and ℓ_3 is forbidden in solutions with monotone block crossings.

eliminated by rerouting the two lines, thus decreasing the number of crossings. As this property is also desirable for block crossings, we use it to define the *monotone block crossing minimization* (MBCM) problem. Note that feasible solutions of MBCM must have the minimum number of pairwise crossings; that is, they must be optimal solutions of MLCM.

Problem 5.2 (MBCM). *Given an instance $(G = (V, E), \mathcal{L})$ of BCM, find a feasible solution that minimizes the number of block crossings subject to the constraint that any two lines cross at most once.*

We will see that there are instances for which BCM allows fewer crossings than MBCM does; see Figure 5.3 in Section 5.2.

Our Contribution. We introduce the new problems BCM and MBCM. To the best of our knowledge, ordering lines by block crossings is a new direction in graph drawing. So far BCM has been investigated only for the case that the *skeleton*, that is, the graph without terminals, is a single edge [BP98], while MBCM is a completely new problem.

We first analyze MBCM on a single edge (Section 5.2), exploiting, to some extent, the similarities to *sorting by transpositions* [BP98]. Then, we use the notion of *good pairs* of lines, that is, lines that should be neighbors, for developing an approximation algorithm for BCM on graphs whose skeleton is a path (Section 5.3); we properly define good pairs so that changes between adjacent edges are taken into account. Yet, good pairs can not always be kept close; we introduce a good strategy for breaking pairs when needed.

Unfortunately, the approximation algorithm does not generalize to trees. We do, however, develop a worst-case optimal algorithm for trees (Section 5.4). It needs $2|\mathcal{L}| - 3$ block crossings and there are instances in which this number of block crossings is necessary in any solution. We then use our algorithm for obtaining approximate solutions for MBCM on the special class of *upward trees*.

As our main result, we present an algorithm for obtaining a solution for BCM on general graphs (Section 5.5). We show that the solutions constructed by our algorithm contain only monotone block moves and are, therefore, also feasible solutions for MBCM. We analyze the upper bound on the number of block crossings that the algorithm yields. While the algorithm

graph class	BCM		MBCM	
single edge	11/8-approx.	[EH06]	3-approx.	Thm. 5.2
path	3-approx.	Thm. 5.3	3-approx.	Thm. 5.4
tree	$\leq 2\|\mathcal{L}\| - 3$ cross.	Thm. 5.5	$\leq 2\|\mathcal{L}\| - 3$ cross.	Thm. 5.5
upward tree	6-approx.	Thm. 5.1	6-approx.	Thm. 5.6
general graph	$O(\|\mathcal{L}\|\sqrt{\|E\|})$ cross.	Thm. 5.7	$O(\|\mathcal{L}\|\sqrt{\|E\|})$ cross.	Thm. 5.7
	bounded degree & edge multiplicity			
tree	FPT	Thm. 5.9	FPT	Thm. 5.9
planar graph	NP-hard	Thm. 5.11	NP-hard	Thm. 5.10

Table 5.1: Overview of our results for BCM and MBCM.

itself is simple and easy to implement, proving the upper bound is non-trivial. Next, we show that the bound is tight; we use a result from projective geometry for constructing worst-case examples in which any feasible solution contains many block crossings. Hence, our algorithm is asymptotically worst-case optimal.

Finally, we consider the restricted variant of the problems in which the maximum degree Δ as well as the maximum *edge multiplicity* c (the maximum number of lines per edge) are bounded (Section 5.6). For the case where the underlying network is a tree, we show that both BCM and MBCM are fixed-parameter tractable with respect to the combined parameter $\Delta + c$. On the other hand, we prove that both variants are NP-hard on general graphs even if both Δ and c are constant. Table 5.1 summarizes our results.

Related Work. Apart from the relevant work for metro-line crossings minimization in general, there are some works in the direction of block crossings.

In the context of VLSI layout, Marek-Sadowska and Sarrafzadeh [MS95] considered not only minimizing the number of crossings, but also suggested distributing the crossings among circuit regions in order to simplify net routing.

As we will later see, BCM on a *single* edge is equivalent to the problem of sorting a permutation by block moves, which is well studied in computational biology for DNA sequences; it is known as *sorting by transpositions* [BP98, CI01]. The task is to find the shortest sequence of block moves transforming a given permutation into the identity permutation. BCM is, hence, a generalization of sorting by transpositions from a single edge to graphs. The complexity of sorting by transpositions was open for a long time; only recently it has been shown to be NP-hard [BFR12]. The currently best known algorithm has an approximation ratio of 11/8 [EH06]. The proof of correctness of that algorithm is based on a computer analysis, which verifies more than 80,000 configurations.

To the best of our knowledge, no tight upper bound for the necessary number of steps in sorting by transpositions is known. There are several variants of sorting by transpositions; see the survey of Fertin et al. [FLR+09]. For instance, Vergara et al. [HV98] used *correcting short block moves* to sort a permutation. In our terminology, these are monotone moves such that the combined length of exchanged blocks does not exceed three. Hence, their problem is a restricted

variant of MBCM on a single edge; its complexity is unknown. The general problem of sorting by (unrestricted) monotone block moves has not been considered, not even on a single edge.

5.2 Block Crossings on a Single Edge

For getting a feeling for the problem, we restrict our attention to the simplest networks consisting of a single edge with multiple lines passing through it, starting and ending in leaves; see Figure 5.3a. Subsequently, we will be able to reuse some of the ideas for a single edge for longer paths and even for trees.

On a single edge, BCM can be reformulated as follows. We choose a direction for the edge $e = (u, v)$, for example, from bottom (u) to top (v). Then, any line passing through e starts on the bottom side in a leaf attached to u and ends at the top side in a leaf attached to v. Suppose we have n lines $\ell_1, \ldots, \ell_n$. The indices of the lines and the order of edges incident to u and v yield a necessary order τ (as a permutation of $\{1, \ldots, n\}$) of the lines on the bottom side of e, that is, at u, and a necessary order π of the lines at the top side of e; compare Figure 5.3a.

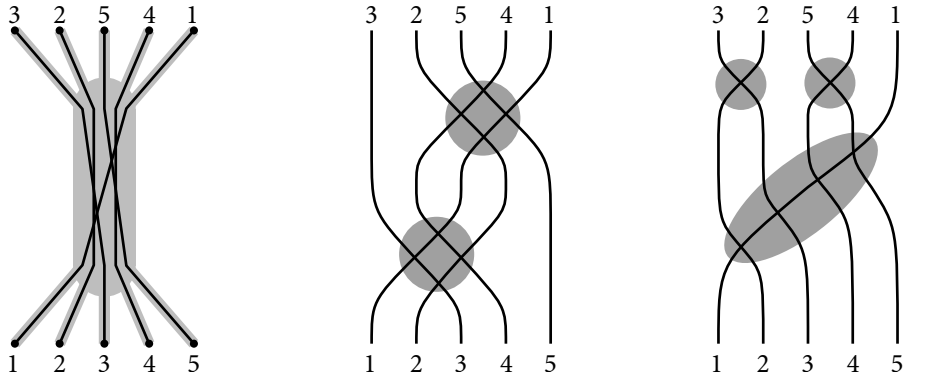
Given these two permutations π and τ, the problem now is to find a shortest sequence of block moves transforming π into τ. By relabeling the lines we can assume that τ is the identity permutation, and the goal is to sort π. This problem is *sorting by transpositions* [BP98], which is, hence a special case of BCM. Sorting by transpositions is known to be NP-hard as Bulteau et al. [BFR12] showed. Hence, BCM, as a generalization, is also NP-hard.

Theorem 5.1. *BCM is NP-hard even if the underlying network is a single edge with attached terminals.*

As sorting by transpositions is quite well investigated, we concentrate on the new problem of sorting with monotone block moves; that means that the relative order of any pair of elements changes at most once. The problems are not equivalent; see Figure 5.3 for an example where dropping monotonicity reduces the number of block crossings in optimum solutions. Hence, we do not know the complexity of MBCM on a single edge. The problem is probably NP-hard even on a single edge, but even for BCM (that is, sorting by transpositions) the NP-hardness proof is quite complicated. As we are mainly interested in more complex networks, we just give an approximation algorithm for MBCM on a single edge. Later, we will see that on general planar graphs, MBCM is indeed NP-hard even if there are few lines per edge (see Section 5.6.2).

For sorting by transpositions and, hence, for BCM on a single edge, there is an 11/8-approximation algorithm by Elias and Hartmann [EH06]. We will now present a simple 3-approximation algorithm for MBCM on a single edge.

Terminology. We first introduce some terminology following previous work where possible. Let $\pi = [\pi_1, \ldots, \pi_n]$ be a permutation of n elements. For convenience, we assume that there are extra elements $\pi_0 = 0$ and $\pi_{n+1} = n + 1$ at the beginning of the permutation and at the end, respectively.



(a) Input instance with pairwise crossings.

(b) Two block moves on the central edge.

(c) Three monotone block moves on the central edge.

Figure 5.3: Sorting Permutation [3, 2, 5, 4, 1] by block moves and by monotone block moves; block moves are highlighted.

A *block* in π is a sequence of consecutive elements $\pi_i, \ldots, \pi_j$ with $1 \leq i \leq j \leq n$. A *block move* (i, j, k) with $1 \leq i \leq j < k \leq n$ on π maps

$$\pi = [\ldots, \pi_{i-1}, \pi_i, \ldots, \pi_j, \pi_{j+1}, \ldots \pi_k, \pi_{k+1}, \ldots] \quad \text{to}$$
$$[\ldots, \pi_{i-1}, \pi_{j+1}, \ldots, \pi_k, \pi_i \ldots, \pi_j, \pi_{k+1}, \ldots],$$

that is, exchanges the blocks $\pi_i, \ldots, \pi_j$ and $\pi_{j+1}, \ldots, \pi_k$.

A block move (i, j, k) is *monotone* if $\pi_q > \pi_r$ for all $i \leq q \leq j < r \leq k$, that is, if any element in the first block $\pi_i, \ldots, \pi_j$ is greater than any element of the second block $\pi_{j+1}, \ldots, \pi_k$. We denote the minimum number of block moves needed to sort π by $\mathrm{bc}(\pi)$ and the minimum number of monotone block moves needed for sorting π by $\mathrm{mbc}(\pi)$.

An ordered pair (π_i, π_{i+1}) (with $0 \leq i \leq n$) is a *good pair* if $\pi_{i+1} = \pi_i + 1$, and a *breakpoint* otherwise. Intuitively, sorting π is a process of creating good pairs (or destroying breakpoints) by block moves. The identity permutation $[1, \ldots, n]$ is the only permutation with only good pairs and no breakpoints.

A permutation is *simple* if it contains no good pairs. Any permutation can be uniquely simplified without affecting its distance to the identity permutation [CI01]. This is done by "glueing" good pairs together, that is, treating the two lines as one line and relabeling. Let $\mathrm{gp}(\pi)$ and $\mathrm{bp}(\pi)$ denote the number of good pairs and of breakpoints in π. As there are $n+1$ pairs (p_i, π_{i+1}) with $0 \leq i \leq n$ and any such pair is either a good pair or a breakpoint, we have $\mathrm{gp}(\pi) + \mathrm{bp}(\pi) = n+1$ for any permutation π. The number $\mathrm{bp}(\pi) = n+1-\mathrm{gp}(\pi)$ of breakpoints can, hence, be interpreted as the number of missing good pairs because the identity permutation $\mathrm{id} = [1, \ldots, n]$ we have $\mathrm{bp}(\mathrm{id}) = 0$ and $\mathrm{gp}(\mathrm{id}) = n + 1$. The identity permutation is the only permutation with this property. Recall that a simple permutation τ does not have good pairs. Hence, $\mathrm{gp}(\tau) = 0$ and $\mathrm{bp}(\tau) = n + 1$.

A Simple Approximation. It is easy to see that a block move (i, j, k) affects three pairs of adjacent elements in π (the pairs (π_i, π_{i+1}), (π_j, π_{j+1}), and (π_k, π_{k+1})). Therefore the number of breakpoints can be reduced by at most three in any block move. This implies $\mathrm{mbc}(\pi) \geq \mathrm{bc}(\pi) \geq \lceil \mathrm{bp}(\pi)/3 \rceil$ for any permutation π as Bafna and Pevzner [BP98] pointed out. Clearly, $\mathrm{bp}(\pi) - 1$ moves suffice for sorting any permutation, which yields a simple 3-approximation for BCM.

We suggest the following algorithm for sorting a *simple* permutation π using only monotone block moves: In each step find the smallest i such that $\pi_i \neq i$ and move element i to position i, that is, exchange blocks $\pi_i, \ldots, \pi_{k-1}$ and π_k, where $\pi_k = i$. Clearly, the step destroys at least one breakpoint, namely $(\pi_{i-1} = i - 1, \pi_i)$. Furthermore, the move is monotone as element i is moved only over larger elements. Therefore, $\mathrm{mbc}(\pi) \leq \mathrm{bp}(\pi)$ and the algorithm yields a 3-approximation.

By first simplifying a general permutation, applying the algorithm, and then undoing the simplification, we can also find a 3-approximation for permutations that are not simple.

Theorem 5.2. *We can find a 3-approximation for MBCM on a single edge in $O(n^2)$ time.*

Clearly, the 3-approximation can be found in $O(n^2)$ time. If we need to output all permutations of block moves, $\Omega(n^2)$ time is also necessary, because there can be a linear number of block moves (for example, for simple permutations). If we do not want to know the sequence of permutations but just the sequence of block moves (i, j, k), this can be improved to $O(n \log n)$ time by proceeding as follows.

Recall that we use simple permutations. In increasing order, we move the elements $i = 1, \ldots, n - 1$. In any step, the monotone block move is described by (i, k, k), where k is the current index of the element i. Hence, the crucial part is determining this index without actually performing the block move. For doing so, we create a binary tree of height $O(\log n)$ whose leaves are the elements $1, \ldots, n$ ordered by their indices in the input; we also store the initial index in any leave. In any step i, when moving the element i, we will mark its leave as deleted and update some additional values; we will, however, never change the structure of the tree. In order to find the element i in the tree, we store, for each inner vertex, the minimum of the elements $i, \ldots, n$ in the subtree. When we mark the element i as deleted, we can easily update the minima, while follow the path to the root, in $O(\log n)$ time.

In any vertex of the tree, we store an additional offset value, which is initially 0. Now, suppose we are in step i, that is, we want to move the element i—which is currently at some position k—to position i with the move (i, k, k). In this move, several—at positions $i, \ldots, k - 1$—are moved by back by one position. The moved elements are exactly the elements $j > i$ that have been placed left of i in the input. In the binary tree, they are represented by the leaves left of the leaf of i that have not yet been deleted. Hence, we do the following. When deleting i from the tree, we follow the path to the root and always increment the offset value for subtrees left of the path; that is, whenever we reach a vertex of the tree coming from a right subtree, we increment the offset for the root of the neighboring left subtree. If we always do so, we can calculate the new position of any leaf by adding the sum of the offset values on the path to the root to the original position. Since updating the values in one step takes only logarithmic time, we need $O(n \log n)$ time in total.

Exhaustive Search. We will later need optimum solutions for both BCM and MBCM on a single edge with a constant number of lines (up to 11) for constructing gadgets for a hardness proof in Section 5.6.2. Therefore, we now show how to find optimum solutions in terms of permutations based on exhaustive search.

The idea is simple. For a permutation π of length n, that is, a permutation of id $= [1, 2, \ldots, n]$, we consider the graph $G_n = (S_n, E_n)$ whose vertex set S_n is the set of all permutations of length n. Two permutations π_1, π_2 are connected by a directed edge $(\pi_1, \pi_2) \in E_n$ if there is a block move that transforms permutation π_1 into π_2. If we are interested in monotone block crossings, then we add the edge (π_1, π_2) only if this block move is monotone.

Now, it is easy to see that $\mathrm{bc}(\pi)/\mathrm{mbc}(\pi)$ is equal to the length of a shortest path between id and π in G_n. Such a path can be found by breadth first search. This takes $O(n!n^3)$ time because there are $n!$ vertices and $O(n!\binom{n}{3}) = O(n!n^3)$ edges in G_n.

We make the source code of the implementation used by us available online[1]. It includes the breadth first search that computes the (monotone) block crossing distance to the identity permutation for all permutations of size n, as well as the code that we used for finding suitable sets of permutations for the different gadgets of our NP-hardness proof (see Section 5.6.2). Note that—due to the large number of permutations—for computing distances of permutations of size 11 a lot of RAM is necessary; in our tests we needed 5 GB; a complete breadth first search needed about an hour, and checking all permutations for finding a suitable set of permutations for the variable gadget took even longer. The computations for the other gadgets with shorter permutations need, of course, significantly less RAM and are much faster. If enough memory is available, just checking all permutations we used can be done within about 15 minutes.

5.3 Block Crossings on a Path

Now we consider an embedded graph $G = (V, E)$ consisting of a path $P = (V_P, E_P)$ with attached terminals. In every node $v \in V_P$ the clockwise order of terminals adjacent to v is given, and we assume that the path is oriented from left to right. We say that a line ℓ *starts* at its leftmost vertex on P and *ends* at its rightmost vertex on P. As we consider only crossings of lines sharing an edge, we assume that the terminals connected to any path node v are in such an order that first lines end at v and then lines start at v; see Figure 5.4. We will first concentrate on developing an approximation algorithm for BCM. Then, we will show how to modify the algorithm for monotone block crossings.

5.3.1 BCM on a Path

We suggest a 3-approximation algorithm for BCM. Similar to the single edge case, the basic idea of the algorithm is to consider good pairs of lines. A *good pair* is, intuitively, an ordered pair of lines that will be adjacent—in this order—in any feasible solution when one of the lines ends. We argue that our algorithm creates at least one additional good pair per block crossing, while even the optimum creates at most three new good pairs per crossing. To describe our algorithm we first define good pairs.

[1] `http://lamut.informatik.uni-wuerzburg.de/blockcrossings/BlockCrossings.java`

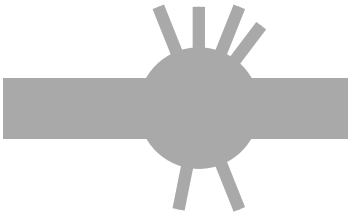


Figure 5.4: Lines starting and ending around a vertex of the path.

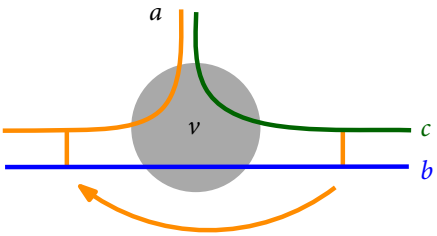


Figure 5.5: Inheritance of a good pair; (a, b) inherits form (c, b).

Definition 5.1 (Good pair)**.** Let a and b be two lines. The ordered pair (a, b) is a *good pair* if one of the following two conditions holds.

(i) Lines a and b end in the same node $v \in P$ and a and b are consecutive in clockwise order around v.

(ii) There are a line c and an interior vertex v of the path P such that c is the first line that enters P in v from above, a is the last line ending in v above P as shown in Figure 5.5, and (c, b) is a good pair.

Note that case (i) of the definition follows the definition of good pairs on a single edge; compare Section 5.2. In case (ii) we say that the good pair (a, b) is *inherited* from (c, b) and identify (a, b) with (c, b), which is possible as a and c do not share an edge. Analogously, there is inheritance for lines starting/ending below P.

As a preprocessing step, we add two virtual lines, t_e and b_e, to each edge $e \in E_P$. The line t_e is the last line entering P before e from the top and the first line leaving P after e to the top. Symmetrically, b_e is the last line entering P before e from the bottom and the first line leaving P after e to the bottom. Although virtual lines are never moved, t_e participates in good pairs, which models the fact that the first line ending after an edge must be brought to the top. Symmetrically, b_e participates in good pairs modeling the fact that the first line ending after an edge must be brought to the bottom.

We now investigate some important properties of good pairs. We first can observe that good pairs are well-defined, that is, a line participates in at most two good pairs (above and below) on each edge.

Lemma 5.1. *Let $e \in E_p$ be an edge and let $\ell \in L_e$. Then ℓ is involved in at most one good pair (ℓ', ℓ) for some $\ell' \in L_e$ and in at most one good pair (ℓ, ℓ'') for some $\ell'' \in L_e$.*

Proof. Let $e = (u, v)$ be the rightmost edge with a line $\ell \in L_e$ that violates the desired property. Assume that the first part of the property is violated, that is, there are two different good pairs (ℓ'_1, ℓ) and (ℓ'_2, ℓ). If ℓ ends at vertex v, there clearly can be at most one of these good pairs because all good pairs have to be of case (i).

Now, suppose that ℓ also exists on the edge $e' = (v, w)$ to the right of e on P. If both ℓ'_1 and ℓ'_2 existed on e', we would already have a counterexample on e'. Hence, at least one of the lines ends at v, that is, at least one of the good pairs results from inheritance at v. On the other hand, this can only be the case for one of the two pairs, suppose for (ℓ'_1, ℓ). Hence, there has to be

another good pair (ℓ_3', ℓ) on e', a contradiction to the choice of e. Symmetrically, we see that there cannot be two different good pairs (ℓ, ℓ_1'') and (ℓ, ℓ_2''). □

A line does not have to be part of a good pair everywhere. We can, however, show that any line is part of a good pair on its last edge of the path.

Lemma 5.2. *If $e = (u, v) \in E_P$ is the last edge before line ℓ ends to the top, then there exists a line ℓ' on e that forms a good pair (ℓ', ℓ) with ℓ. Symmetrically, if e is the last edge before ℓ ends to the top, then there exists a line ℓ'' on e that forms a good pair (ℓ, ℓ'') with ℓ.*

Proof. We suppose that ℓ ends to the top; the other case is analogous. We consider the clockwise order of lines ending around v. If there is a predecessor ℓ' of ℓ, then, by case (i) of the definition, (ℓ', ℓ) is a good pair. Otherwise, ℓ is the first line ending at v above the path. Then, the virtual line t_e that we added is its predecessor, and (t_e, ℓ) is a good pair. □

In what follows, we say that a solution or an algorithm *creates* a good pair (a, b) in a block crossing if the two lines a and b of the good pair are brought together in the right order by that block crossing; analogously, we speak of *breaking* a good pairs if the two lines are neighbors in the right order before the block crossings and are no longer after the crossing.

Lemma 5.3. *There are only two possibilities for creating a good pair (a, b):*

(i) The lines a and b start at the same node consecutively in the right order.

(ii) A block crossing brings a and b together.

Proof. In the interior of the common subpath of a and b, the good pair (a, b) can only be created by block crossings because either a and b cross each other or lines between a and b cross a or b. Hence, (a, b) can only be created without a block crossing at the moment when the last of the two lines, say a, starts at a node v. In this case a has to be the first line starting at v on the top of P. This implies that, due to inheritance, there is a good pair (c, b), where c is the last line ending at v to the top. It follows that the good pair (c, b), which is identical to (a, b), existed before v. Analogously, we get a contradiction if b is the first line starting at v on the bottom of P. □

In case (i) of the lemma, we also say that (a, b) is an *initial good pair*. Analogously to the lemma, a good pair can only be destroyed by a crossing or the end of both lines.

It is easy to see that any solution, especially an optimal one, has to create all good pairs. As we identify good pairs resulting from inheritance with the original good pair—resulting from case (i) of Definition 5.1—, it suffices to consider good pairs resulting from two lines ending at the same vertex consecutive in clockwise order. As the lines must not cross in this vertex, they must be together before this vertex is reached.

Recall that our main idea is to use that, in analogy to the case of a single edge, a block crossing can create at most three new good pairs. There will be few cases in which a block crossing has to break a good pair. We show that such a crossing cannot increase the number of good pairs at all.

Lemma 5.4. *In any block crossing the number of good pairs increases by at most* 3*. In a block crossing that breaks a good pair, the number of good pairs does not increase.*

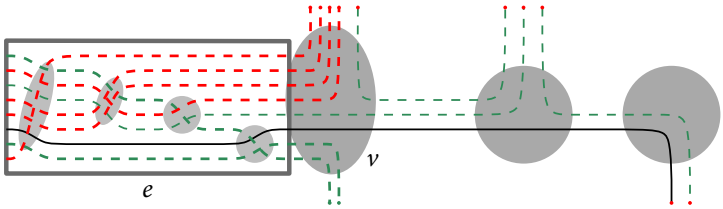


Figure 5.6: Ordering the lines on edge e in a step of the algorithm.

Proof. We consider a block crossing on some edge that transforms the sequence

$$\pi = [\ldots, a, b, \ldots, c, d, \ldots, e, f, \ldots] \quad \text{into} \quad \pi' = [\ldots, a, d, \ldots, e, b, \ldots, c, f, \ldots],$$

that is, the blocks $b, \ldots, c$ and $d, \ldots, e$ are exchanged. The only new pairs of consecutive lines that π' contains compared to π are (a, d), (e, b), and (c, f). Even if these are all good pairs, the total number of good pairs increases only by three.

Now, suppose that the block crossing breaks a good pairs. The only candidates are (a, b), (c, d), and (e, f). If (a, b) was a good pair, then the new pairs (a, d) and (e, b) cannot be good pairs because, on one edge, there can only be one good pair $(a, \cdot)$ and one good pair $(\cdot, b)$; see Lemma 5.1. Hence, only (c, f) can possibly be a new good pair. Since one good pair is destroyed and at most one good pair is created, the number of good pairs does not increase. The cases that the destroyed good pair is (c, d) or (e, f) are analogous. □

Using good pairs, we formulate our algorithm as follows; see Figure 5.6 for an example. We traverse P from left to right. On an edge $e = (u, v) \in E_P$ of the path, there are *red lines* that end at v to the top, *green lines* that end at v to the bottom, and *black lines* that continue on the next edge. We bring the red lines in the right order to the top by moving them upwards. Doing so, we keep existing good pairs together. If a line is to be moved, we consider the lines below it consecutively. As long as the current line forms a good pair with the next line, we extend the block that will be moved. We stop at the first line that does not form a good pair with its successor. Then, we move the whole block of lines linked by good pairs in one block move to the top. Next, we bring the green lines in the right order to the bottom, again keeping existing good pairs together. There is an exception: sometimes one good pair on e cannot be kept together. If the moved block is a sequence of lines containing both red and green lines, and possibly some—but not all—black lines, then the block has to be broken; see the block (d, a, b, e) in Figure 5.7. Note that this can only happen in one move on an edge; there can only be one sequence containing both red and green lines because all red lines are part of a single sequence and all green lines are part of a single sequence due to case (i) of Definition 5.1. There are two cases when the sequence of good pairs has to be broken:

(i) A good pair in the sequence contains a black line and has been created by the algorithm previously. Then, we break the sequence at this good pair.

(ii) All pairs containing a black line are initial good pairs, that is, they have not been created by a crossing. Then, we break at the pair that ends last of these. When comparing the end

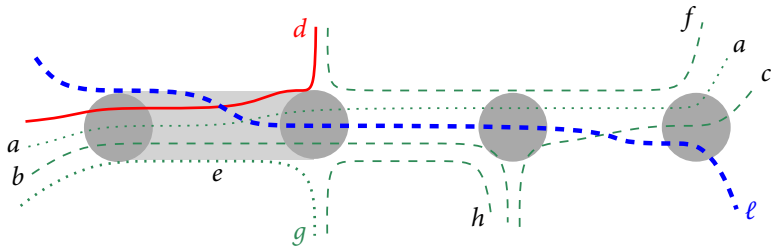


Figure 5.7: The (necessary) insertion of line ℓ forces breaking a good pair, that is, (a,b) $(\equiv (a,c))$, (d,a) $(\equiv (f,a))$, or (b,g) $(\equiv (b,h))$, on edge e.

of pairs we take inheritance into account, that is, a good pair ends only when the last of the pairs that are linked by inheritance ends.

After an edge has been processed, the lines ending to the top and to the bottom are on their respective sides in the right relative order. Hence, our algorithm produces a feasible solution. We show that the algorithm produces a 3-approximation for the number of block crossings. A key property is that our strategy for case (ii) is optimal.

Lemma 5.5. *Let* ALG *and* OPT *be the number of block crossings created by the algorithm and an optimal solution, respectively. Then,* $\text{ALG} \leq 3\,\text{OPT}$.

Proof. Block crossings that do not break a good pair always increase the number of good pairs. If we have a block crossing that breaks a good pair in a sequence as in case (i) then there has been a block crossing that created the good pair previously as a side effect, that is, there was an additional (red or green) good pair whose creation caused that block crossing. Hence, we can say that the destroyed good pair did not exist previously and still have at least one new good pair per block crossing.

If we are in case (ii), that is, all good pairs in the sequence are initial good pairs (see Figure 5.7), then these good pairs also initially existed in the optimal solution. It is not possible to keep all these good pairs because the remaining black lines have to be somewhere between the block of red lines and the block of green lines. Hence, even the optimal solution has to break one of these good pairs, on this edge or previously.

Let $\overline{\text{bc}}_{\text{alg}}$ and $\overline{\text{bc}}_{\text{opt}}$ be the numbers of broken good pairs due to case (ii) in the algorithm and the optimal solution, respectively. In a crossing in which the algorithm breaks such a good pair the number of good pairs stays the same as one good pair is destroyed and another created. On the other hand, in a crossing that breaks a good pair the number of good pairs can increase by at most two even in the optimal solution (actually, this number cannot increase at all; see Lemma 5.4). Let gp be the total number of good pairs and let gp_{init} be the number of initial good pairs. Note that, according to Definition 5.1, good pairs resulting from inheritance are not counted separately for gp as they are identified with another good pair. We get $\text{gp} \geq \text{ALG} - \overline{\text{bc}}_{\text{alg}} + \text{gp}_{\text{init}}$ and $\text{gp} \leq 3 \cdot \text{OPT} - \overline{\text{bc}}_{\text{opt}} + \text{gp}_{\text{init}}$. Hence, $\text{ALG} \leq 3\,\text{OPT} + (\overline{\text{bc}}_{\text{alg}} - \overline{\text{bc}}_{\text{opt}})$ combining both estimates.

To prove an approximation factor of 3, it remains to show that $\overline{\text{bc}}_{\text{alg}} \leq \overline{\text{bc}}_{\text{opt}}$. First, note that the edges where good pairs of case (ii) are destroyed, are exactly the edges where such a sequence of initial good pairs exists; that is, the edges are independent of any algorithm or

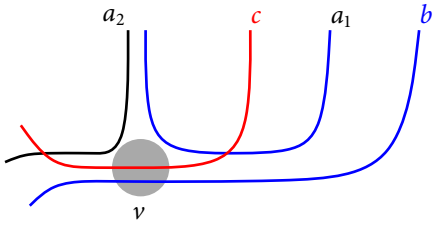


Figure 5.8: Line c prevents that (a_2, b) inherits from (a_1, b).

solution. We show that, among these edges, our strategy ensures that the smallest number of pairs is destroyed, and pairs that are destroyed once are reused as often as possible for breaking a sequence of initial good pairs.

To this end, let $e'_1, \ldots, e'_{\overline{\mathrm{bc}_{\mathrm{alg}}}}$ be the sequence of edges where the algorithm destroys a new good pair of type (ii), that is, an initial good pair that has never been destroyed before. We follow the sequence and argue that the optimal solution destroys a new pair for each of these edges. Otherwise, there is a pair e'_i, e'_j (with $i < j$) of edges in the sequence where the optimal solution uses the same good pair p on both edges. Let p' and p'' be the pairs used by the algorithm on e'_i and e'_j, respectively, for breaking a sequence of initial good pairs. As p' was preferred by the algorithm over p, we know that p' still exists on e'_j. As p' is in a sequence with p, the algorithm still uses p' on e'', a contradiction. This completes the proof. □

We can now conclude with the following theorem; the running time is obvious.

Theorem 5.3. *There is an $O(|\mathcal{L}|(|\mathcal{L}|+n))$-time algorithm for finding a 3-approximation for BCM on instances where the underlying network is a path of length n with attached terminals.*

5.3.2 MBCM on a Path

The algorithm for paths presented in the previous section does not guarantee monotonicity of the solution. It can, however, be turned into a 3-approximation algorithm for MBCM. To achieve this, we will adjust the definition of inheritance of good pairs, as well as the step of destroying good pairs, and we will sharpen the analysis.

We first modify our definition of inheritance of good pairs. We prevent inheritance in the situations in which keeping a pair of lines together at the end of an edge is not possible without either having a forbidden crossing in the following vertex or violating monotonicity. We concentrate on inheritance with lines ending to the top; the other case is symmetric.

Suppose we have a situation as shown in Figure 5.8 with a good pair (a_1, b). Line c must not cross b. On the other hand it has to be below a_2 near node v and separate a_2 and b there. Hence, bringing or keeping a_2 and b together is of no value, as they have to be separated in any solution. Therefore, we modify the definition of good pairs, so that the pair (a_2, b) does not inherit from (a_1, b) in this situation; we say that line c is *inheritance-preventing* for (a_1, b).

Apart from the modified definition of good pairs, one part of our algorithm needs to be changed in order to ensure monotonicity of the solution. A block move including black lines could result in a forbidden crossing. We focus on the case, where black lines are moved together

with red lines to the top. This can only occur once per edge. The case that black lines are moved together with green lines to the bottom is symmetric. Let $b_0, b_1, \ldots, b_k$ be the sequence of good pairs from the bottommost red line $r = b_0$ on. If there is some line ℓ above the block that must not be crossed by a line b_i of the block, then we have to break the sequence. We consider such a case and assume that i is minimal. Hence, we have to break one of the good pairs in $(r, b_1), (b_1, b_2), \ldots, (b_{i-1}, b_i)$. Similar to case (i) in the algorithm for BCM, we break a pair of this sequence that is not initial. If all the pairs are initial (case (ii)), we choose the pair (b_{j-1}, b_j) with $j \leq i$ minimal such that the end node of b_j is below the path, and break the sequence there. Note that line ℓ must end below the path, otherwise it would prevent inheritance of at least one of the good pairs in the sequence. Hence, also b_i ends below the path, and b_j is well-defined.

It is easy to see that our modified algorithm still produces a feasible ordering. We now show that the solution is also monotone.

Lemma 5.6. *The modified algorithm produces an ordering with monotone block crossings.*

Proof. We show that any pair of lines that cross in a block crossing is in the wrong order before the crossing. Monotonicity of the whole solution then follows. We consider moves where blocks of lines are brought to the top; the other case is symmetric.

Suppose that a red line r is brought to the top. As all red lines that have to leave above r have been brought to the top before, r crosses only lines that leave below it, that is, lines that have to be crossed by r. If a black line ℓ is brought to the top, then it is moved together in a block that contains a sequence of good pairs from the bottommost red line r' to ℓ. Suppose that ℓ crosses a line c that must not be crossed by ℓ. Line c cannot be red because all red lines that are not in the block that is moved at the moment have been brought to the top before. It follows that r' has to cross c. Hence, we can find a good pair (a, b) in the sequence from r' to ℓ such that a has to cross c but b must not cross c. In this case, the algorithm will break at least one good pair between r' and b. It follows that c does not cross ℓ, a contradiction. □

We have now seen that the modified algorithm creates feasible solutions for MBCM. It remains to proof the approximation factor as we modified the algorithm. This can be done similar to BCM.

Lemma 5.7. *Let* $\mathrm{ALG}_{\mathrm{mon}}$ *be the number of block crossings created by the algorithm for MBCM and let* $\mathrm{OPT}_{\mathrm{mon}}$ *be the number of block crossings of an optimal solution for MBCM. It holds that* $\mathrm{ALG}_{\mathrm{mon}} \leq 3\,\mathrm{OPT}_{\mathrm{mon}}$.

Proof. As for the nonmonotone case, all block crossings that our algorithm introduces increase the number of good pairs, except when the algorithm breaks a sequence of initial good pairs in case (ii). Again, also the optimal solution has to have crossings where such sequences are broken. As for BCM, let gp be the total number of good pairs, let $\mathrm{gp}_{\mathrm{init}}$ be the number of initial good pairs, let $\overline{\mathrm{bc}}_{\mathrm{alg}}$ be the number of broken good pairs of case (ii) for the algorithm, and let $\overline{\mathrm{bc}}_{\mathrm{opt}}$ be the number of such broken pairs for the optimum solution.

In a crossing of case (ii), the two lines of the destroyed pair lose their partner. Hence, there is only one good pair after the crossing, and the number of good pairs does not change at all; compare Lemma 5.4.

Hence, $\mathrm{gp} \geq \mathrm{ALG}_{\mathrm{mon}} - \overline{\mathrm{bc}}_{\mathrm{alg}} + \mathrm{gp}_{\mathrm{init}}$ and $\mathrm{gp} \leq 3\left(\mathrm{OPT}_{\mathrm{mon}} - \overline{\mathrm{bc}}_{\mathrm{opt}}\right) + \mathrm{gp}_{\mathrm{init}}$. Combining both estimates, we get

$$\mathrm{ALG}_{\mathrm{mon}} \leq 3\,\mathrm{OPT}_{\mathrm{mon}} + (\overline{\mathrm{bc}}_{\mathrm{alg}} - 3\overline{\mathrm{bc}}_{\mathrm{opt}}).$$

Let $\overline{\mathrm{bc}}_{\mathrm{alg,top}}$ be the number of splits for case (ii) where the block move brings lines to the top, and let $\overline{\mathrm{bc}}_{\mathrm{alg,bot}}$ be the number of such splits where the move brings lines to the bottom. Clearly, $\overline{\mathrm{bc}}_{\mathrm{alg}} = \overline{\mathrm{bc}}_{\mathrm{alg,top}} + \overline{\mathrm{bc}}_{\mathrm{alg,bot}}$. We get

$$\begin{aligned} \mathrm{ALG}_{\mathrm{mon}} &\leq 3 \cdot \mathrm{OPT}_{\mathrm{mon}} + (\overline{\mathrm{bc}}_{\mathrm{alg}} - 3 \cdot \overline{\mathrm{bc}}_{\mathrm{opt}}) \\ &\leq 3 \cdot \mathrm{OPT}_{\mathrm{mon}} + (\overline{\mathrm{bc}}_{\mathrm{alg,top}} - \overline{\mathrm{bc}}_{\mathrm{opt}}) + (\overline{\mathrm{bc}}_{\mathrm{alg,bot}} - \overline{\mathrm{bc}}_{\mathrm{opt}}). \end{aligned}$$

To complete the proof, we show $\overline{\mathrm{bc}}_{\mathrm{alg,top}} \leq \overline{\mathrm{bc}}_{\mathrm{opt}}$. Symmetry will yield $\overline{\mathrm{bc}}_{\mathrm{alg,bot}} \leq \overline{\mathrm{bc}}_{\mathrm{opt}}$ and, hence, $\mathrm{ALG}_{\mathrm{mon}} \leq 3\,\mathrm{OPT}_{\mathrm{mon}}$.

Let $e'_1, \ldots, e'_{\overline{\mathrm{bc}}_{\mathrm{alg,top}}}$ be the sequence of edges where the algorithm uses a new good pair as a breakpoint for a sequence of type (ii) when lines leave to the top, that is, a good pair that has not been destroyed before. Again, we argue that even the optimal solution has to use a different breakpoint pair for each of these edges. Otherwise, there would be a pair e', e'' of edges in this sequence where the optimal solution uses the same good pair p on both edges. Let p' and p'' be the two good pairs used by the algorithm on e' and e'', respectively. Let $p' = (\ell', \ell'')$. We know that ℓ' leaves the path to the top and ℓ'' leaves to the bottom as described in case (ii). Because all lines in the orders on e' and e'' stay parallel—otherwise they could not form a sequence of good pairs—, we know that lines above ℓ' leave to the top, and lines below ℓ'' leave to the bottom. In particular, p' still exists on e'', as p stays parallel and also still exists.

As in the description of the algorithm, let a and b be lines such that (a, b) is the topmost good pair in the sequence for which a line c exists on e'' that crosses a but not b. If (a, b) is below p', then the algorithm would reuse p' instead of the new pair p'', since (a, b) is in a sequence below p; hence, also p' is in the sequence and above (a, b).

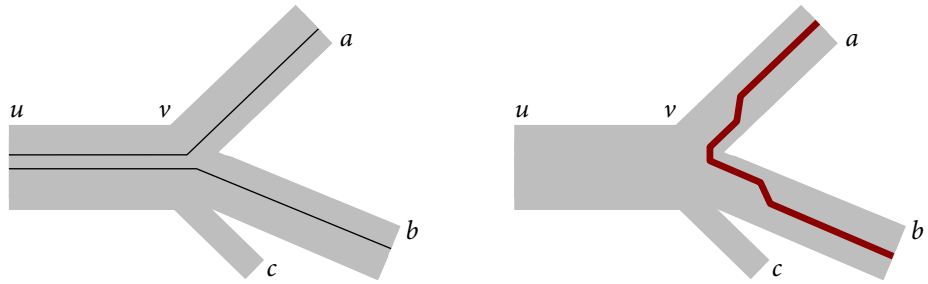
Now suppose that (a, b) is above p'. The pair (a, b) is created by inheritance because c ends between a and b. As both a and b end to the top, separated from the bottom side of the path by p', this inheritance takes place at a node, where a is the last line to end on the top side. But in this case c prevents the inheritance of the good pair (a, b) because it crosses only a, a contradiction. □

The modified algorithm still needs $O(|\mathcal{L}|(|\mathcal{L}| + n))$ time. We can now conclude with the following theorem.

Theorem 5.4. *There is an $O(|\mathcal{L}|(|\mathcal{L}| + |n|))$-time algorithm for finding a 3-approximation for MBCM on instances where the underlying network is a path of length n with attached terminals.*

5.4 Block Crossings on Trees

In the following we focus on instances of BCM and MBCM where the underlying network is a tree. As we have seen in Section 4.4, there are examples of trees that do not allow consistent line directions. This is in contrast to paths, where we could direct all lines from left to right and use this to define good pairs of lines analogously to the case of a single edge. For general trees, we



(a) Situation before inserting the line.

(b) The line is inserted on edges (v, a) and (v, b).

Figure 5.9: Insertion of a new line (red, bold) into the current order on edges (v, a) and (v, b).

do not have an approximation algorithm. We will, however, present an algorithm that yields a worst-case optimal bound on the number of block crossings. Then, we consider the special class of *upward trees* which have an additional constraint on the lines; for upward trees we develop a 6-approximation for BCM and MBCM.

5.4.1 General Trees

We can show that a linear number of monotone block crossings suffices for any tree. More precisely, $2|\mathcal{L}| - 3$ block crossings suffice (and are sometimes necessary).

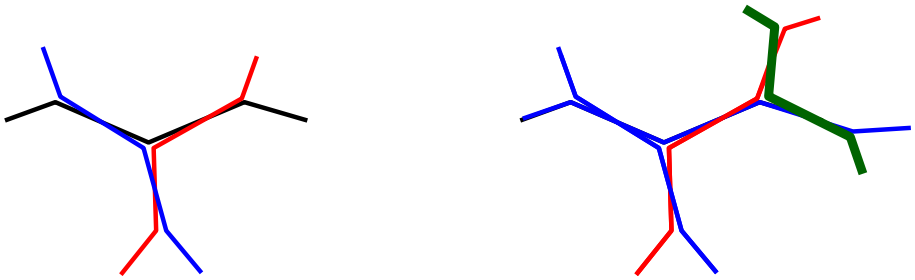
Theorem 5.5. *Given an embedded tree $T = (V, E)$ of n vertices and a set $\mathcal{L}$ of lines on T, we can order the lines with at most $2|\mathcal{L}| - 3$ monotone block crossings in $O(|\mathcal{L}|(|\mathcal{L}| + n))$ time.*

Proof. We give an algorithm in which paths are inserted one by one into the current orders; for each newly inserted path we create at most two additional monotone block crossings. The first line that we insert into the empty orders cannot create a crossing, and the second line crosses the first one at most once. Hence, we need $2|\mathcal{L}| - 3$ monotone block crossings in total.

We start with an edge $e = (r, w)$ incident to a terminal r, that is, a leaf of the tree. As r is a terminal, there is only one line ℓ on the edge e which will be the first line that we insert into the orders of the solution that we are building. We now assume that the tree is rooted at r and that all edges are directed pointing away from the root.

When the algorithm processes an arbitrary edge $e = (u, v)$, the lines in L_e will already be ordered; that is, they do not need to cross on yet unprocessed edges of T because the necessary crossings for pairs of lines in L_e have been placed on edges treated before. We consider all unprocessed edges $(v, a), (v, b), \ldots$ incident to v in clockwise order and build the right order for them. The relative order of lines that also pass through (u, v) is kept unchanged on the new edges. For all lines passing through v that have not been treated before, we apply an insertion procedure; see Figure 5.9.

Consider, for example,the insertion of a line ℓ passing through (v, a) and (v, b). Close to v, that is, at the ports of v corresponding to (v, a) and (v, b), we add ℓ on both edges at the innermost position such that we do not get vertex crossings with lines that pass through (v, a) or (v, b). We find the correct position of ℓ in the current order of L_{va} at the end of edge (v, a)



(a) Instance for $|\mathcal{L}| = 3$.

(b) The instance for $|\mathcal{L}| = 4$ is created by adding a line (green, bold) to the instance for $|\mathcal{L}| = 3$.

Figure 5.10: Examples for trees with $2|\mathcal{L}| - 3$ necessary crossings. By adding more lines using the same construction by which the instance for $|\mathcal{L}| = 4$ was created from the one for $|\mathcal{L}| = 3$, instances with an arbitrary number of lines can be created.

at a relative to the lines already inserted so far, and insert ℓ using a single block crossing. This crossing will be the last one on (v, a) going from v to a. Similarly, ℓ is inserted into L_{vb}.

We have to make sure that lines that do not have to cross are inserted in the right order. As we know the right relative order for a pair of such lines, we can make sure that the one that has to be innermost at node v is inserted first. Similarly, by considering the clockwise order of edges around v, we know the right order of line insertions such that there are no avoidable vertex crossings. When all new paths are inserted, the orders on $(v, a), (v, b), \ldots$ are correct; we proceed by recursively processing these edges.

Suppose that monotonicity is violated, that is, there is a pair of lines that crosses twice. Then, the crossings must have been introduced when inserting the second of those lines on two edges incident to a node v. This can, however, not happen, as at node v the two edges are inserted in the right order. Hence, the block crossings of the solution are monotone. □

In comparison to the cases of a single edge and of a path, where we had at most $|\mathcal{L}|$ block crossings, the bound for trees has doubled. We can, however, show that the new bound $2|\mathcal{L}| - 3$ is tight, that is, that there are tree instances where this number of block crossings is necessary even for an optimum solution.

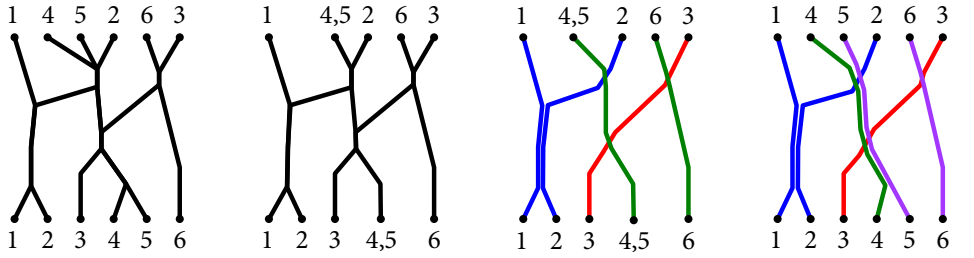
Worst-Case Examples. Consider the graph shown in Figure 5.10. The new bold green line in Figure 5.10b is inserted so that it crosses two existing paths. The example can easily be extended to instances of arbitrary size where $2|\mathcal{L}| - 3$ block crossings are necessary in any solution.

Unfortunately, there are also examples in which our algorithm creates $|\mathcal{L}| - 1$ crossings while a single block crossing suffices; see Figure 5.11 for $|\mathcal{L}| = 5$. The extension of the example to any number of lines is straightforward. This shows that the algorithm does not yield a constant-factor approximation.

(a) Started at the leftmost edge, the algorithm, produces 4 crossings.

(b) In an optimum solution one block crossing suffices.

Figure 5.11: Worst case example for our algorithm for trees shown for five edges. It can easily be extended to an arbitrary number of edges (and crossings).



(a) Input instance with pairwise crossings.

(b) Simplified instance; 4 and 5 are merged.

(c) Line ordering on simplified instance.

(d) Simplification undone for solution.

Figure 5.12: The algorithm for upward trees in three steps applied to a simple instance. The instance is drawn in the style of a permutation with lines numbered from 1 to 6.

5.4.2 Upward Trees

Next, we introduce an additional constraint on the lines, which helps us to approximate the minimum number of block crossings. We consider an *upward* tree T with a set of lines $\mathcal{L}$. The instance $(T, \mathcal{L})$ is an upward tree if T has a planar upward drawing—respecting the given embedding—in which all paths are monotone in vertical direction, and all path sources are on the same height as well as all path sinks; see Figure 5.12a. Note that upward trees require consistent line directions, but are even more restricted. Bekos et al. [BKPS08] already considered such trees (under the name "left-to-right trees") for the metro-line crossing minimization problem. Note that a graph whose skeleton is a path is not necessarily an upward tree.

Our algorithm consists of three steps. First, we perform a simplification step that removes some lines. Second, we use the algorithm for trees presented in the previous section on the simplified instance. Finally, we re-insert the removed lines into the constructed order without introducing new block crossings. We first consider MBCM. We start by analyzing the upward embedding; see Figure 5.12 for an illustration of the steps of the algorithm.

Given an upward drawing of T, we read a permutation π produced by the terminals on the top similar to the case of a single edge; we assume that the terminals produce the identity permutation on the bottom. Similar to the single-edge case, the goal is to sort π by a shortest sequence of block moves. Edges of T restrict some block moves on π; for example, the blocks $[1, 4]$ and $[5]$ in Figure 5.12a cannot be exchanged because there is no suitable edge with all these lines. However, we can use the lower bound for block crossings on a single edge, see Section 5.2:

For sorting a *simple* permutation π, at least $\lceil \mathrm{bp}(\pi)/3 \rceil = \lceil (|\mathcal{L}|+1)/3 \rceil$ block moves are necessary. We stress that the simplicity of π is crucial here because the algorithm for trees may create up to $2|\mathcal{L}| - 3$ crossings. To get an approximation, we show how to simplify a tree.

Consider two non-intersecting paths a and b that are adjacent in both permutations and share a common edge. We prove that one of these paths can be removed without changing the optimal number of monotone block crossings. First, if any other line c crosses a then it also crosses b in any solution (i). This is implied by the monotonicity of the block crossings, by planarity, and by the y-monotonicity of the drawing. Second, if c crosses both a and b then all three paths share a common edge (ii); otherwise, there would be a cycle in the graph due to planarity. Hence, given any solution for the paths $\mathcal{L} \setminus \{b\}$, we can construct a solution for $\mathcal{L}$ by inserting b parallel to a without any new block crossing. To insert b, we must first move all block crossings involving a to the common subpath with b. This is possible due to observation (ii). Finally, we can place b parallel to a.

To get a 6-approximation for an upward tree T, we first remove lines until the tree is simple. Then we apply the insertion algorithm presented in Section 5.4.1, and finally re-insert the lines removed in the first step. The number of block crossings is at most $2|\mathcal{L}'|$, where $\mathcal{L}'$ is the set of lines of the simplified instance. As an optimal solution has at least $|\mathcal{L}'|/3$ block crossings for this simple instance, and re-inserting lines does not create new block crossings, we get the following result.

Theorem 5.6. *Given an embedded upward tree $T = (V, E)$ of n vertices and a set $\mathcal{L}$ of lines on T, we can find a 6-approximation for MBCM in $O(|\mathcal{L}|(|\mathcal{L}| + n))$ time.*

If we consider BCM instead of MBCM, we face the problem that we do not know whether every solution for the simplified instance can be transformed into a solution for the input instance without additional crossings. However, we can observe that the solutions that our algorithm finds for the simplified instance are always monotone and, hence, can be transformed back. Furthermore, dropping lines can never increase the necessary number of block crossings. Hence, also for BCM we have the lower bound of $\lceil (|\mathcal{L}'|+1)/3 \rceil$ block crossings. Summing up, we also get a 6-approximation for BCM by using the same algorithm.

Corollary 5.1. *Given an embedded upward tree $T = (V, E)$ of n vertices and a set $\mathcal{L}$ of lines on T, we can find a 6-approximation for BCM in $O(|\mathcal{L}|(|\mathcal{L}| + n))$ time.*

5.5 Block Crossings on General Graphs

In this section, we consider general graphs. We suggest an algorithm that achieves an upper bound on the number of block crossings and show that it is asymptotically worst-case optimal. Our algorithm uses only monotone block moves, that is, each pair of lines crosses at most once. The algorithm works on any embedded graph; it does not even need to be planar, we just need to know the circular order of incident edges around each vertex.

The idea of the algorithm is as follows. We process the edges in some arbitrary order. When we treat an edge, we sort the lines that traverse it. A crossing between a pair of lines can be created on the edge only if this edge is the first one treated by the algorithm that is used by both lines of the pair; see Algorithm 5.1 for the structure of the algorithm.

foreach edge e with $|L_e| > 1$ **do**
  Build order of lines on both sides of e.
  Merge lines that are in the same group on both sides.
  Find the largest group of consecutive lines that stay parallel on e.
  Insert all other lines into this group and undo merging.

Algorithm 5.1: Ordering the lines on a graph.

The crucial part is sorting the lines on an edge. Suppose we currently deal with edge e and want to sort L_e. Due to the path intersection property, the edge set used by the lines in L_e forms a tree on each side of e; see Figure 5.13. We cut these trees at the edges that have already been processed. Then, each line on e starts at a leaf on one side and ends at a leaf on the other side. Note that multiple lines can start or end at the same leaf representing an edge that has previously been treated by the algorithm.

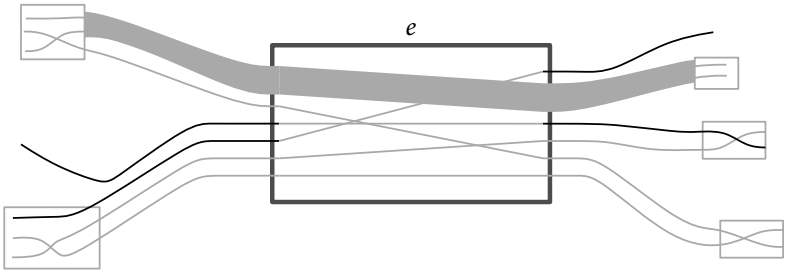
From the tree structure and the orders on the ports of the edges processed previously, we get two orders of the lines, one on each side of e. We consider *groups of lines* that start or end at a common leaf of the tree (such as the group of red lines in Figure 5.13). All lines of a group have been seen on a common edge, and, hence, have been sorted. Therefore lines of the same group form a consecutive subsequence on one side of e, and have the same relative order on the other side of e.

Let g and g' be a group of lines on the left and on the right side of e, respectively. Suppose that the set $\mathcal{L}'$ of lines starting in g on the left and ending in g' on the right consists of multiple lines. As the lines of g as well as the lines of g' stay parallel on e, $\mathcal{L}'$ must form a consecutive subsequence (in the same order) on both sides. Now, we *merge* $\mathcal{L}'$ into one representative, that is, we remove all lines of $\mathcal{L}'$ and replace them by a single line that is in the position of the lines of $\mathcal{L}'$ in the sequences on both sides of e. Once we find a solution, we replace the representative by the sequence. This does not introduce new block crossings as we will see. Consider a crossing that involves the representative of $\mathcal{L}'$, that is, the representative is part of one of the moved blocks. After replacing the representative, the sequence $\mathcal{L}'$ of parallel lines is completely contained in the same block. Hence, we do not need additional block crossings.

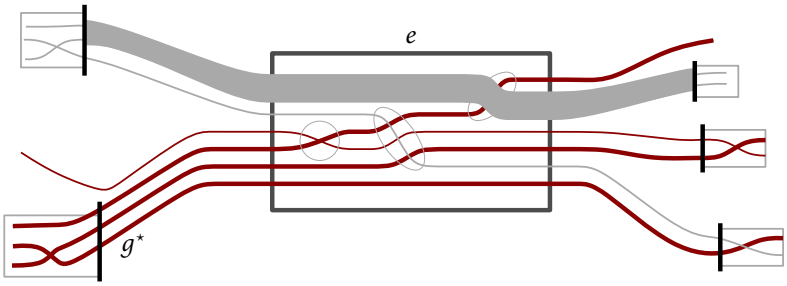
We apply this merging step to all pairs of groups on the left and right end of E. Then, we identify a group $g^\star$ with the largest number of lines after merging, and insert all remaining lines into $g^\star$ one by one. Clearly, each insertion requires at most one block crossing; in Figure 5.13 we need three block crossings to insert the lines into the largest (red) group $g^\star$. After computing the crossings, we undo the merging step and obtain a solution for edge e.

Theorem 5.7. *Given an instance $(G = (V, E), \mathcal{L})$ of MBCM, Algorithm 5.1 computes a feasible solution in $O(|E|^2|\mathcal{L}|)$ time. The resulting number of block crossings is bounded by $|\mathcal{L}|\sqrt{|E'|}$, where $E' \subseteq E$ is the set of edges with at least two lines.*

Proof. First, it is easy to see that no avoidable crossings are created, due to the path intersection property. Additionally, we treat all edges with at least two lines, which ensures that all unavoidable crossings will be placed. Hence, we get a feasible solution using only monotone crossings.



(a) Cutting edges (marked) define groups. The lines marked in gray are merged as they are in the same group on both sides.



(b) Sorting by insertion into the largest group $g^\star$ (red, fat). The merged lines always stay together, in particular, when their block crosses other lines.

Figure 5.13: Sorting the lines on an edge e in a step of our algorithm.

Our algorithm sorts the lines on an edge in $O(|\mathcal{L}||E|)$ time. We can build the tree structure and find the orders and groups by following all lines until we find a terminal or an edge that has been processed before in $O(|\mathcal{L}||E|)$ time. Merging lines and finding the largest group needs $O(|\mathcal{L}|)$ time; sorting by insertion into this group and undoing the merging can be done in $O(|\mathcal{L}|^2)$ time. Note that $|\mathcal{L}| \leq |E|$ due to the path terminal property.

For analyzing the total number of block crossings, we maintain an *information* table T with $|\mathcal{L}|^2$ entries. Initially, all the entries are empty. After processing an edge e in our algorithm, we fill the entry $T[\ell, \ell'] = e$ for each pair (ℓ, ℓ') of lines that we see together for the first time. The main idea is that with b_e block crossings on edge e, we fill at least b_e^2 new entries of T. This ultimately yields the desired upper bound of $|\mathcal{L}|\sqrt{|E|}$ for the total number of block crossings.

More precisely, let the *information gain* $I(e)$ be the number of pairs of (not necessarily distinct) lines ℓ, ℓ' that we see together on a common edge e for the first time. Clearly, $\sum_{e \in E} I(e) \leq |\mathcal{L}|^2$. Suppose that $b_e^2 \leq I(e)$ for each edge e. Then,

$$\sum_{e \in E} b_e^2 \leq \sum_{e \in E} I(e) \leq |\mathcal{L}|^2.$$

Using the Cauchy-Schwarz inequality $|\langle x, y \rangle| \leq \sqrt{\langle x, x \rangle \cdot \langle y, y \rangle}$ with $x = (b_e)_{e \in E'}$ as the vector of block crossing numbers and $y = (1)_{e \in E'}$, we see that the total number of block crossings is

$$\sum_{e \in E'} b_e = |\langle x, y \rangle| \leq \sqrt{\langle x, x \rangle \cdot \langle y, y \rangle} = \sqrt{\left(\sum_{e \in E'} b_e^2 \right) \cdot |E'|} \leq \sqrt{|\mathcal{L}|^2 |E'|} = |\mathcal{L}| \sqrt{|E'|}.$$

It remains to show that $b_e^2 \leq I(e)$ for an edge e. We analyze the lines after the merging step. Consider the groups on both sides of e; we number the groups on the left side $\mathfrak{L}_1, \ldots, \mathfrak{L}_n$ and the groups on the right side $\mathfrak{R}_1, \ldots, \mathfrak{R}_m$. For $1 \leq i \leq n$ let $l_i = |\mathfrak{L}_i|$ and for $1 \leq j \leq m$ let $r_j = |\mathfrak{R}_j|$. Without loss of generality, we can assume that $\mathfrak{L}_1$ is the largest of the $n + m$ groups and we will insert all remaining lines into $\mathfrak{L}_1$.

Then, $b_e \leq |L_e| - l_1$. Let s_{ij} be the number of lines that are in group $\mathfrak{L}_i$ on the left side and in group $\mathfrak{R}_j$ on the right side of e. Note that $s_{ij} \in \{0, 1\}$, otherwise we could still merge lines. Then $l_i = \sum_{j=1}^{m} s_{ij}$, $r_j = \sum_{i=1}^{n} s_{ij}$, $s := |L_e| = \sum_{i=1}^{n} \sum_{j=1}^{m} s_{ij}$, and $b_e = s - l_1$. In terms of this notation, the information gain is

$$I(e) = s^2 - \sum_{i=1}^{n} l_i^2 - \sum_{j=1}^{m} r_j^2 + \sum_{i=1}^{n} \sum_{j=1}^{m} s_{ij}^2,$$

which can be seen as follows. From the total number s^2 of pairs of lines on the edge, we have to subtract all pairs of lines that are in the same group on the left or on the right side of the edge; we must be careful not to subtract pairs that are in the same group on the left and on the right side twice. By applying the following Lemma 5.8 to the values s_{ij} (for $1 \leq i \leq n$ and $1 \leq j \leq m$), we get $b_e^2 \leq I(e)$.

To complete the proof, note that the unmerging step neither decreases $I(e)$ nor does it change b_e. □

Lemma 5.8. *For $1 \leq i \leq n$ and $1 \leq j \leq m$, let $s_{ij} \in \{0, 1\}$. Let $l_i = \sum_{j=1}^{m} s_{ij}$ for $1 \leq i \leq n$ and let $r_j = \sum_{i=1}^{n} s_{ij}$ for $1 \leq j \leq m$ such that $l_1 \geq l_i$ for $1 \leq i \leq n$ and $l_1 \geq r_j$ for $1 \leq j \leq m$. Let $s = \sum_{i=1}^{n} \sum_{j=1}^{m} s_{ij}$, $b = s - l_1$, and $I = s^2 - \sum_{i=1}^{n} l_i^2 - \sum_{j=1}^{m} r_j^2 + \sum_{i=1}^{n} \sum_{j=1}^{m} s_{ij}^2$. Then, $b^2 \leq I$.*

Proof. It is easy to see that, for any $1 \le i \le n, 1 \le j \le m$, it holds that $s_{ij}(s_{ij} - s_{1j}) \ge 0$ as $s_{ij} \in \{0,1\}$. Using this property in the last line of the following sequence of (in-)equalities, we see that

$$\begin{aligned}
I - b^2 &= \left(s^2 - \sum_{i=1}^{n} l_i^2 - \sum_{j=1}^{m} r_j^2 + \sum_{i=1}^{n}\sum_{j=1}^{m} s_{ij}^2\right) - \left(s^2 - 2sl_1 + l_1^2\right) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} s_{ij}^2 + 2l_1(s - l_1) - \sum_{i=2}^{n} l_i^2 - \sum_{j=1}^{m} r_j \sum_{i=1}^{n} s_{ij} \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} s_{ij}^2 + 2l_1 \sum_{i=2}^{n}\sum_{j=1}^{m} s_{ij} - \sum_{i=2}^{n} l_i \sum_{j=1}^{m} s_{ij} - \sum_{i=1}^{n}\sum_{j=1}^{m} s_{ij} r_j \\
&= \sum_{i=2}^{n}\sum_{j=1}^{m} s_{ij}\left(s_{ij} + 2l_1 - l_i - r_j\right) - \sum_{j=1}^{m} s_{1j}\left(r_j - s_{1j}\right) \\
&= \sum_{i=2}^{n}\sum_{j=1}^{m} s_{ij}\left(s_{ij} + \underbrace{2l_1 - l_i - r_j}_{\ge 0}\right) - \sum_{j=1}^{m} s_{1j} \sum_{i=2}^{n} s_{ij} \\
&\ge \sum_{i=2}^{n}\sum_{j=1}^{m} \underbrace{s_{ij}\left(s_{ij} - s_{1j}\right)}_{\ge 0} \ge 0.
\end{aligned}$$

□

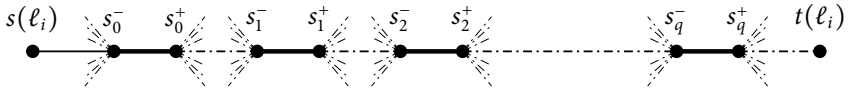
Next we show that the upper bound on the number of block crossings that our algorithm achieves is asymptotically tight. To this end, we use the existence of Steiner systems for building (nonplanar) worst-case examples of arbitrary size in which many block crossings are necessary.

Theorem 5.8. *For any prime power q, there exists a graph $G_q = (V_q, E_q)$ of $\Theta(q^2)$ vertices with a set of lines $\mathcal{L}_q$ so that $\Omega\left(|\mathcal{L}_q|\sqrt{|E'_q|}\right)$ block crossings are necessary in any solution, where $E'_q \subseteq E_q$ is the set of edges with at least two lines.*

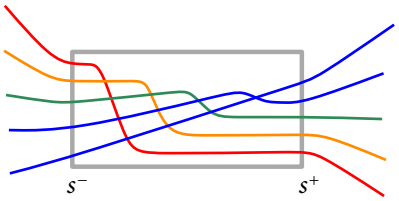
Proof. Let q be a prime power. From the area of projective planes it is known that an $S(q^2 + q + 1, q+1, 2)$-Steiner system exists [VB06], that is, there is a set $\mathcal{S}$ of $q^2 + q + 1$ elements with subsets $S_1, S_2, \ldots, S_{q^2+q+1}$ of size $q+1$ each such that any pair of elements $s, t \in \mathcal{S}$ appears together in exactly one set S_i.

We build the graph $G_q = (V_q, E_q)$ by first adding vertices s^-, s^+ and an edge (s^-, s^+) for any $s \in \mathcal{S}$. These edges will be the only ones with multiple lines on them, that is, they form E'_q. Additionally, we add an edge (s^+, t^-) for each pair $s, t \in \mathcal{S}$. Next, we build a line ℓ_i for each set S_i as follows. We choose an arbitrary order $s_0, s_1, s_2, \ldots, s_q$ of the elements of S_i; then, we introduce extra terminals $s(\ell_i)$ and $t(\ell_i)$ in which the new line $\ell_i = \left(s(\ell_i), s_0^-, s_0^+, s_1^-, s_1^+, \ldots, s_q^-, s_q^+, t(\ell_i)\right)$ starts and ends, respectively; see Figure 5.14a.

As any pair of lines shares exactly one edge, the path intersection property holds. For each $s \in \mathcal{S}$, we order the edges around vertices s^- and s^+ in the embedding so that all $q+1$ lines on the edge representing s have to cross by making sure that the order of the lines is exactly reversed between s^- and s^+; see Figure 5.14b. Then, at least $q/3$ block crossings are necessary on each



(a) Line ℓ_i is routed through the edges representing $s_0, s_1, s_2, \ldots, s_q$.



(b) The order of the lines is reverted between s^- and s^+.

Figure 5.14: Construction of the worst-case example.

edge (compare the case of a single edge in Section 5.2), and, hence, $(q^2+q+1)q/3 = \Theta(q^3)$ block crossings in total. On the other hand, $|\mathcal{L}|\sqrt{|E'|} = (q^2+q+1)\sqrt{q^2+q+1} = \Theta(q^3)$. □

Note that the graphs for the worst-case instances in the previous proof are not planar. It is an interesting question to decide whether the upper bound is also asymptotically tight for planar instances.

5.6 Instances with Bounded Maximum Degree and Edge Multiplicity

Similar to general metro-line crossing minimization, also block crossing minimization is interesting with bounded maximum degree and bounded edge multiplicity; compare Section 4.5. Recall that, in this setting, all stations have constant maximum degree Δ, and the maximum edge multiplicity is a constant c, that is, $|L_e| \leq c$ for each edge e.

We first show that the restricted problem variants of both BCM and MBCM can be solved in polynomial time if the underlying network is a tree. On the other hand, we prove that the restricted variants are NP-hard on planar graphs.

5.6.1 Restricted (M)BCM on Trees

We want to modify the dynamic program presented in Section 4.5 for MLCM. For BCM this is quite easy: We just need to count block crossings instead of single crossings when solving the problem on a single edge with at most c lines. The rest does not need to be changed. For MBCM, we additionally need to guarantee that two lines cross at most once. Similar to the modification for MLCM-P, we can do this by disregarding combinations of permutations that lead to forbidden crossings when combining solutions for subtrees. Hence, we get the following result.

Theorem 5.9. *BCM and MBCM can be solved optimally in $O(n)$ time on tree instances of maximum degree Δ and maximum edge multiplicity c if both Δ and c are constants.*

Now, we want to analyze the runtime. The only relevant modification with respect to the runtime is the modified computation of the number of block crossings—instead of single crossings—needed for sorting an edge with at most c lines if the order on both ports is already fixed. We can do this by using breadth-first search in the graph of permutations of at most c elements; compare Section 5.2. We have $O(c!)$ vertices—the permutations—and $O(c!\binom{c}{3}) = O(c!c^3)$ edges—resulting from the up to $\binom{c}{3}$ block moves that are possible in one step. Note that for MBCM we additionally have to disallow nonmonotone block moves. Summing up, we have to replace $O(c^2)$ by $O(c!c^3)$ in the runtime analysis, resulting in a total runtime of $O(n(c!)^{2\Delta}\Delta c^3)$. Again, this runtime yields that we have a fixed-parameter tractable algorithm.

Corollary 5.2. *BCM and MBCM are fixed-parameter tractable on tree instances with respect to the parameter $c + \Delta$, where Δ is the maximum degree and c is the maximum edge multiplicity, with a runtime of $O(n \cdot (c!)^{2\Delta}\Delta c^3)$.*

5.6.2 NP-Hardness of Restricted BCM and MBCM

The hardness of sorting by transpositions implies that BCM is NP-hard even on a single edge; compare Theorem 5.1. For the restricted version, however, this does not hold because the edge can contain only a constant number of lines. In fact, due to the FPT algorithm presented in the previous section, neither BCM nor MBCM can be NP-hard on trees for any constant maximum degree and edge multiplicity. However, we will see that for general planar graphs even the restricted problems are NP-hard. We start with MBCM.

We will now show that restricted MBCM is NP-hard on general planar graphs. More specifically, MBCM is NP-hard even if the maximum degree is 3 and there is no edge with more than 11 lines.

Theorem 5.10. *MBCM is NP-hard on planar graphs even if the maximum degree is 3 and the maximum edge multiplicity is 11.*

Proof. We show hardness by reduction from Planar 3SAT, which is known to be NP-hard even if any variable occurs in exactly three different clauses [DJP+94]; compare Section 2.3. Let (X, C) be an instance of Planar 3SAT where X is the set of variables and C is the set of clauses. Recall that any clause contains only two or three literals. The graph $G_{XC} = (X \cup C, E_{XC})$ with the edge set $E_{XC} = \{\{x, \gamma\} \mid \text{variable } x \text{ occurs in clause } \gamma\}$ describing the occurrence of variables in clauses is planar.

We now construct an instance $(G = (V, E), \mathcal{L})$ of MBCM modeling the 3SAT instance. To this end, we take a fixed planar embedding of G_{XC}. We replace each variable $x \in X$ in G_{XC} by a *variable gadget* V_x and each clause $\gamma \in C$ by a *clause gadget* C_γ. If $x \in \gamma$, then the edge $\{x, \gamma\}$ becomes an edge $\{v_x, v_\gamma\}$ where v_x and v_γ are vertices of the variable gadget and the clause gadget, respectively. If $\neg x \in \gamma$, we replace the edge $\{x, \gamma\}$ by a path (v_x, u, u', v_γ) where u and u' are vertices of a *negator gadget* N_x. In both cases, we call the edges of the connection between the gadgets the *variable path*. By placing the gadgets in the positions of the respective vertices of G_{XC} and routing the variable paths along the edges, we get a planar embedding of G.

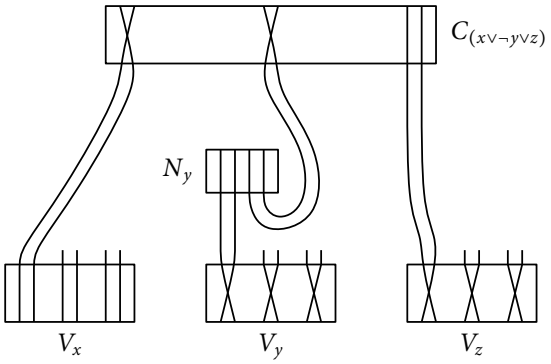


Figure 5.15: Connections of variable paths for a clause $\gamma = (x \vee \neg y \vee z)$ where x is `false` and y and z are `true`.

On any edge outside of a gadget, exactly two lines represent a literal. Looking in the direction of the clause gadget, we say that the literal state is `true` if the lines crossed in the previous gadget, and that it is `false` otherwise. We will build the gadgets in such a way that crossings occur only within gadgets in crossing minimal solutions. Furthermore, we will connect the lines so that any pair of lines representing a literal has to cross, that is, it can be either in `true` or in `false` state. Figure 5.15 shows the connections for the variables of a clause.

We first define the properties that we need for our gadgets. In the descriptions, we use global constants k_{var}, k_{neg}, k_{cls}, and k'_{cls} for numbers of crossings.

Variable Gadget: The variable gadget has three port edges e_1, e_2, and e_3 that are part of variable paths, and each of these edges has exactly two lines on it. These edges and lines are the only ones that leave the gadget. In a crossing-minimal solution in which the three pairs of lines either do or do not cross inside the gadget, there are exactly k_{var} crossings in the gadget. Any solution in which some, but not all, of these pairs cross inside the gadget has at least $k_{\text{var}} + 1$ crossings.

Negator Gadget: The negator gadget is basically a version of the variable gadget with only two ports. There are two port edges e_1 and e_2, each with a pair of lines. In crossing-minimal solutions in which both pairs either do or do not cross inside the gadget, there are exactly k_{neg} crossings. In the configurations in which exactly one of the pairs crosses inside the gadget, there are at least $k_{\text{neg}} + 1$ crossings.

Clause Gadget: The clause gadget has three (or two) port edges, each with a pair of lines. If at least one of the pairs does not cross inside the gadget, there are exactly k_{cls} crossings; if all pairs cross inside the gadget, at least $k_{\text{cls}} + 1$ crossings are necessary.

We also need a version of the clause gadget with only two port edges, both with a pair of lines. In this version, there are exactly k'_{cls} crossings if at least one of the pairs of lines does not cross inside the gadget; otherwise, there are at least $k'_{\text{cls}} + 1$ crossings.

Given such gadgets, we build the network that models the 3SAT instance. We are interested only in *canonical solutions*, that is, solutions in which (i) all crossings are inside gadgets and (ii) any variable gadget has exactly k_{var} crossings, any negator gadget has exactly k_{neg} crossings, and any clause gadget has exactly k_{cls} crossings (or k'_{cls} crossings if the clause has just two literals), resulting in a total number K of allowed crossings. It is easy to see that canonical solutions are exactly the solutions with at most K crossings. We claim that, if there is a canonical solution, the instance of 3SAT is satisfiable.

To see this, we analyze the variable gadget. As there are only k_{var} crossings in a canonical solution, the pairs of lines modeling the variable values either all cross, or all stay crossing-free. Hence, after leaving the gadget, the three pairs all have the same state, `true` if they crossed, and `false` otherwise. As there are no crossings outside of gadgets, this state can only change on the variable path if it contains a negator.

Suppose a variable path contains a negator gadget. In this case two lines ℓ_1 and ℓ_2, coming from a variable gadget, are connected by port edge e_1, and two lines ℓ_3 and ℓ_4, leaving towards a clause gadget, are connected by port edge e_2. As we consider a canonical solution, there are only two possibilities. If both pairs do not cross inside the negator, the pair $\{\ell_1, \ell_2\}$ has to cross in the variable gadget and, therefore, is in `true` state. Then, the pair $\{\ell_3, \ell_4\}$ is in `false` state, as the lines do not cross in the negator gadget. On the other hand, if both pairs cross inside the negator, the pair $\{\ell_1, \ell_2\}$ represents `false`, and $\{\ell_3, \ell_4\}$ represents `true`. Hence, the negator gadget works as desired.

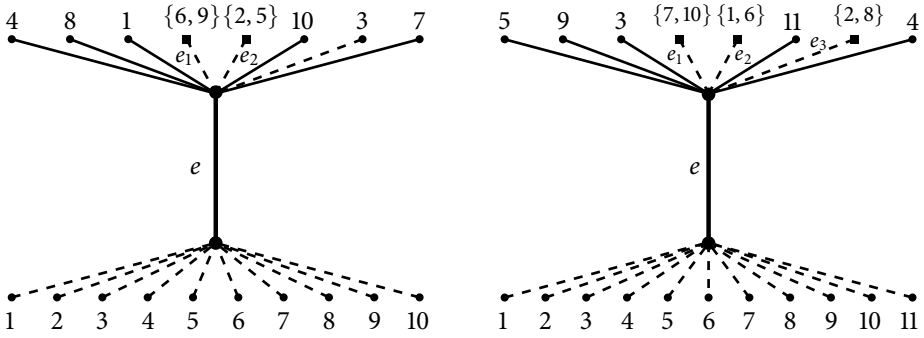
Finally, we consider the clause gadgets. As there are only k_{cls} crossings (or k'_{cls} crossings in the version with only two literals), at least one of the variable pairs does not cross inside the gadget, which means that it is in `true` state. Hence, the clause is satisfied.

Now, suppose we are given a truth assignment that satisfies all clauses. We want to build a canonical solution for the block crossing problem. To this end, we fix, for each variable gadget, the order of the pairs of lines (crossing or non-crossing) corresponding to the truth value of the variable, which is the same for all port edges. Then, we take the appropriate solution with k_{var} block crossings for this gadget. Next, for each negator gadget, there is exactly one possible realization with k_{neg} block crossings given the state of the pair of lines on the ingoing port edge. Finally, for each clause gadget, there is at least one variable pair that did already cross, as the given truth assignment satisfies all variables. Hence, we can realize the clause gadget with only k_{cls} block crossings (or k'_{cls} block crossings in the version with two literals). Therefore, we can find a canonical solution.

We have now seen that, assuming that there are appropriate gadgets, the satisfiability of a given instance of Planar 3SAT is equivalent to deciding whether the corresponding instance of MBCM has a canonical solution. For completing the proof, it remains to show how to build the gadgets with the desired properties.

Negator gadget. The negator gadget is illustrated in Figure 5.16a. It consists of an edge e with 10 lines, two port edges e_1 and e_2 with two lines each, and 16 edges, connected to leaves, with one line per edge. Assuming that the lines on e form the identity permutation on the lower end of the edge, we can read different permutations on the upper end, depending on the solution. However, the upper permutation always follows the *permutation template*

$$\pi_{\text{neg}} = [4, 8, 1, a_1, a_2, b_1, b_2, 10, 3, 7],$$



(a) Negator gadget. (b) Main part of the variable gadget.

Figure 5.16: Gadgets for the NP-hardness proof. Lines starting/ending in leaves of the graph and passing through port edges (dashed)) are indicated by numbers (or sets of two numbers for port edges).

where $\{a_1, a_2\} = \{6, 9\}$ and $\{b_1, b_2\} = \{2, 5\}$. The pairs $\{a_1, a_2\}$ and $\{b_1, b_2\}$ are on the port edges e_1 and e_2, respectively, and can be connected to a variable or negator gadget.

The important property of the permutations of type π_{neg} is that there are only two ways to arrange the lines in any solution of MBCM with the minimum number of block crossings. It is not hard to check that

- $\text{mbc}(\pi) = 5$ if $\pi = [4, 8, 1, \underline{6, 9}, \underline{2, 5}, 10, 3, 7]$ or $\pi = [4, 8, 1, \underline{9, 6}, \underline{5, 2}, 10, 3, 7]$ and
- $\text{mbc}(\pi) = 6$ in the remaining cases, that is, if $\pi = [4, 8, 1, \underline{6, 9}, \underline{5, 2}, 10, 3, 7]$ or $\pi = [4, 8, 1, \underline{9, 6}, \underline{2, 5}, 10, 3, 7]$.[2]

Given a canonical solution, we can assume that the pairs of lines a_1, a_2 and b_1, b_2 do not cross on the edges e_1 and e_2 since crossings on these edges can be moved to e without increasing the total number of block crossings in the solution. Hence, in a canonical solution, both pairs of lines $\{a_1, a_2\}$ and $\{b_1, b_2\}$ either cross on e or do not cross there.

Variable gadget. The basic part of the variable gadget is illustrated in Figure 5.16b. Its structure is similar to the negator gadget: The gadget consists of an edge e with 11 lines, three port edges e_1, e_2, and e_3 with two lines each, and 16 edges with one line per edge. Again, we can assume that all the crossings are located on e in a canonical solution. The lines on e form a permutation of the template

$$\pi_{\text{var}} = [5, 9, 3, a_1, a_2, b_1, b_2, 11, c_1, c_2, 4],$$

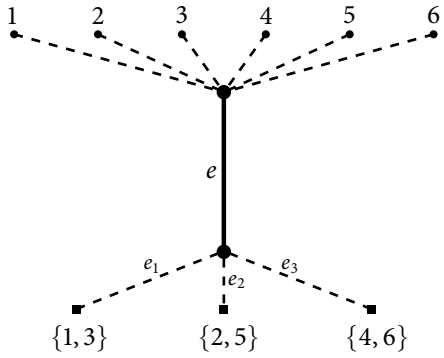
where $\{a_1, a_2\} = \{7, 10\}$, $\{b_1, b_2\} = \{1, 6\}$, and $\{c_1, c_2\} = \{2, 8\}$.

One can check that

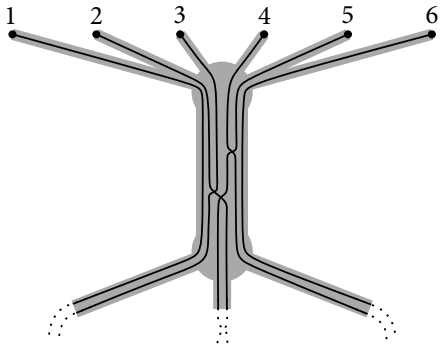
- $\text{mbc}(\pi) = 6$ if $\pi = [5, 9, 3, \underline{7, 10}, \underline{1, 6}, 11, \underline{8, 2}, 4]$ or $\pi = [5, 9, 3, \underline{10, 7}, \underline{6, 1}, 11, \underline{2, 8}, 4]$ and
- $\text{mbc}(\pi) = 7$ in the remaining six cases that follow the template π_{var}.

In other words, in a canonical solution the pairs of lines $\{a_1, a_2\}$, $\{b_1, b_2\}$, and $\{c_1, c_2\}$ form either the state $(\texttt{true}, \texttt{true}, \texttt{false})$ or $(\texttt{false}, \texttt{false}, \texttt{true})$ in the gadget. Therefore, we use

[2] One can use the exhaustive search method presented in Section 5.2 for solving MBCM exactly for these instances.



(a) Clause gadget.



(b) Sketch of a solution for the clause gadget for $c = (l_1 \vee l_2 \vee l_3)$, where l_1 and l_3 are `true` and l_2 is `false`.

Figure 5.17: The clause gadget for the NP-hardness proof. Lines starting/ending in leaves of the graph and passing through port edges (dashed)) are indicated by numbers (or sets of two numbers for port edges).

an additional negator connected to the pair $\{c_1, c_2\}$ by the port edge e_3, so that, in a canonical solution, the variable gadget encodes either `true` or `false` for all variable pairs at the same time.

Clause gadget. The clause gadgets is illustrated in Figure 5.17a. It consists of an edge e with 6 lines, three port edges e_1, e_2, and e_3 with two lines each, and 6 edges with one line per edge. The lines form a permutation of the template

$$\pi_{\text{cls}} = [a_1, a_2, b_1, b_2, c_1, c_2],$$

where $\{a_1, a_2\} = \{1, 3\}$, $\{b_1, b_2\} = \{2, 5\}$, and $\{c_1, c_2\} = \{4, 6\}$.

One can check that

- $\text{mbc}(\pi) = 3$ if $\pi = [\underline{3, 1}, \underline{5, 2}, \underline{6, 4}]$ and
- $\text{mbc}(\pi) = 2$ in the remaining five cases of permutations following template π_{cls}.

Hence, in a crossing optimal solution, at least one of the pairs of lines, $\{a_1, a_2\}$, $\{b_1, b_2\}$, and $\{c_1, c_2\}$, must *not* cross inside the gadget, that is, the corresponding literal must be `true`; see Figure 5.17b for an example of such a configuration.

By dropping the edge e_3 and the corresponding two lines 4 and 6 and renaming line 5 to 4, we get a variant for clause gadgets with two literals. Then, we have a permutation of the template $\pi'_{\text{cls}} = [a_1, a_2, b_1, b_2]$ where $\{a_1, a_2\} = \{1, 3\}$ and $\{b_1, b_2\} = \{2, 4\}$. One can check that

- $\text{mbc}(\pi) = 2$ if $\pi = [\underline{3, 1}, \underline{4, 2}]$ and
- $\text{mbc}(\pi) = 1$ in the remaining three cases following the template π'_{cls}.

Again, in a canonical solution, at least one of the literals corresponding to the pairs $\{a_1, a_2\}$ and $\{b_1, b_2\}$ must be `true`.

We have now seen that the desired gadgets exist. Additionally, we have seen that no edge contains more than 11 lines. So far, the maximum degree of the underlying graph is 12. We can,

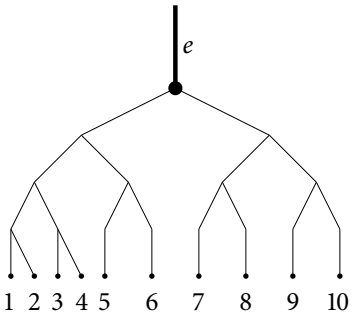


Figure 5.18: Lower part of a negator gadget modified for maximum degree 3.

however, easily modify the gadgets so that the maximum degree is 3. We do this as follows. On both sides of the central edge e of each gadget, we replace the node where the lines split by a tree-like structure in which the lines split into only two groups per step; see Figure 5.18. Note that this modification does neither allow to save block crossings, nor does it make additional crossings necessary. This completes the proof. □

The general variant of (nonmonotone) BCM is NP-hard even for a single edge; see Theorem 5.1. For constant maximum degree and edge multiplicity, however, the problem is tractable on trees; see Theorem 5.9. Next we show that on general planar graphs BCM is NP-hard even for constant maximum degree and edge multiplicity. To this end, we modify the negator and variable gadgets; the clause gadget does not need to be changed because the properties of the permutations we used there still hold if we allow nonmonotone block moves.

Negator gadget. The structure for the variable gadget stays the same. We just replace the used permutation template by

$$\pi_{\text{neg}} = [3, a_1, a_2, 4, 7, b_1, b_2],$$

where the lines $\{a_1, a_2\} = \{1, 6\}$ leave the gadget on port edge e_1 and the lines $\{b_1, b_2\} = \{2, 5\}$ leave the gadget on e_2.

One can check that

- $\text{bc}(\pi) = 3$ if $\pi = [3, \underline{1, 6}, 4, 7, \underline{2, 5}]$ or $\pi = [3, \underline{6, 1}, 4, 7, \underline{5, 2}]$ and
- $\text{bc}(\pi) = 4$ in the remaining two cases for template π_{neg} (note that we now use nonmonotone block crossings).

Hence, both pairs of lines $\{a_1, a_2\}$ and $\{b_1, b_2\}$ either cross in the gadget or do not cross there in a canonical solution.

Variable gadget. Also the structure of the variable gadget stays the same. We just replace the used permutation template by

$$\pi_{\text{var}} = [6, a_1, a_2, b_1, b_2, c_1, c_2],$$

where the lines a_1 and a_2 leave the gadget on port edge e_1, b_1 and b_2 leave the gadget on e_2, and c_1 and c_2 leave it on e_3; furthermore, $\{a_1, a_2\} = \{1, 4\}$, $\{b_1, b_2\} = \{3, 7\}$, and $\{c_1, c_2\} = \{2, 5\}$.

One can check that
- $\mathrm{bc}(\pi) = 3$ if $\pi = [6, \underline{1}, \underline{4}, \underline{3}, \underline{7}, \underline{5}, \underline{2}]$ or $\pi = [6, \underline{4}, \underline{1}, \underline{7}, \underline{3}, \underline{2}, \underline{5}]$ and
- $\mathrm{bc}(\pi) = 4$ in the remaining six cases for template π_{var}.

Hence, in a canonical solution the pairs of lines a_1, a_2, b_1, b_2, and c_1, c_2 form either the state $(\texttt{true}, \texttt{true}, \texttt{false})$ or $(\texttt{false}, \texttt{false}, \texttt{true})$ in the gadget. Again, we use an additional negator connected to the pair c_1, c_2 by the port edge e_3 for ensuring that the variable gadget encodes either `true` or `false` for all variable pairs at the same time.

Using the new gadgets, we immediately get the reduction for BCM. We note that we can ensure maximum degree 3 by the same construction that we used for MBCM. Note that both negator and variable gadget for BCM use fewer lines compared to MBCM; the maximum number of lines on an edge is 7.

Theorem 5.11. *BCM is NP-hard on planar graphs even if the maximum degree is* 3 *and the maximum edge multiplicity is* 7*.*

We point out that the hardness results for bounded degree and edge multiplicity imply that, in contrast to the case of trees, BCM and MBCM are not fixed-parameter tractable with respect to these parameters on general graphs. The problems could, however, be fixed-parameter tractable with respect to different parameters such as the number of crossings.

5.7 Concluding Remarks

We have introduced the new variants BCM and MBCM of the metro-line crossing minimization problem in which one wants to order the lines taking more advanced crossings into account. We have presented approximation algorithms for single edges, paths, and upward trees. Then we have developed an algorithm that bounds the number of block crossings on general graphs and have showed that our bound is asymptotically tight. Finally, we have investigated the problems under bounded maximum degree and edge multiplicity, both of which are valid assumptions for practical purposes. Under these restrictions, we have solved BCM and MBCM optimally on trees by giving a fixed-parameter tractable algorithm. Additionally, we have proven that BCM and MBCM are NP-hard on general graphs even if maximum degree and edge multiplicity are small.

Open Problems. As our results are the first for block crossing minimization, there are still many interesting open problems. First, the complexity status of MBCM on a single edge would be interesting to know, mainly from a theoretical point of view. The hardness proof for BCM is quite complicated and does not easily extend to MBCM. Second, a challenging task is to develop an approximation algorithm for BCM on general graphs. The third important question is whether there exists a fixed-parameter tractable algorithm for BCM and MBCM on paths, trees, and general graphs with respect to the allowed number of block crossings. For the problem MLCM-P, we presented such an algorithm in Section 4.3; however, for block crossings this seems to be much more difficult since BCM is already NP-hard on a single edge.

Recently, Bereg et al. [BHNP13] investigated the problem of drawing permutations with few bends; they represented each element of the permutation as a line, similar to a metro line. Also

for the visual complexity of a metro line an important criterion is the number of its bends or inflection points, that is, the points where the direction of the line changes. Hence, an interesting question is how to visualize metro lines using the minimum total number of inflection points.

Part II

Point-Set Embeddability

Chapter 6

Point-Set Embeddability and Large Crossing Angles

In many applications one wants to visualize a graph in such a way that the vertices are placed at desired positions. This means that either the exact position for each vertex is prescribed, or there is a set of input points and we have to draw the graph such that each vertex is placed on one of the input points. This setting is known as *point-set embeddability* (PSE).

So far, point-set embeddability has almost exclusively been considered for planar drawing styles. However, drawings with crossings can be almost as well-readable as planar drawings if the crossing angles are large enough. In this chapter, we consider point-set embeddability with large crossing angles. We either require that we just have right-angle crossings, or that all crossing angles are close to 90°.

We first show that point-set embeddability is NP-hard for any choice of α if the edges must be drawn as straight-line segments. Next, we show how to create embeddings with minimum crossing angle $\alpha > 0$ for any graph and any point set such that each edge may have one or two bends. In both cases, we use only bounded area for our embeddings. Finally, we show that three bends per edge suffice for being able to find a RAC embedding of any graph on any set of points.

6.1 Introduction

In point-set embeddability (PSE) problems we are given not only a graph that is to be drawn, but also a set of points in the plane that specify where the vertices of the graph can be placed. The problem class was introduced by Gritzmann et al. [GMPP91] more than twenty years ago. They showed that any n-vertex outerplanar graph can be embedded on any set of n points in the plane (in general position) such that edges are represented by straight-line segments connecting the respective points and no two edge representations cross. Later on, the PSE question was also raised for other drawing styles, for example, by Pach and Wenger [PW01] and by Kaufmann and Wiese [KW02] for drawings with polygonal edges, so-called *polyline drawings*. In these and most other works, however, planarity of the output drawing was an essential requirement.

Experiments on the readability of drawings [HHE08] showed that polyline drawings with angles at edge crossings close to 90° and a small number of bends per edge are almost as readable as planar drawings. Motivated by these findings, Didimo et al. [DEL11] defined right-angle-crossing (RAC) drawings where pairs of crossing edges must form a right angle and, more generally, αAC drawings (for $\alpha \in (0, 90°]$) where the crossing angle must be at least α.

In this chapter, we investigate the intersection of the two areas, point-set embeddability and the RAC or αAC drawing style. The resulting problem αAC point-set embeddability is defined as follows.

Definition 6.1 (αAC PSE)**.** Given an n-vertex graph $G = (V, E)$ and a set S of n points in the plane, determine whether there exists a bijection $\mu\colon V \to S$, and a polyline drawing of G so that each vertex v is mapped to $\mu(v)$ and the drawing is αAC, that is, all crossing angles are at least α. If such a drawing exists and the largest number of bends per edge in the drawing is b, we say that G *admits an* αAC_b *embedding on* S.

The point-set embeddability problem with right-angle crossings—or RAC PSE—is the special version of αAC PSE with $\alpha = 90°$. Analogously to αAC PSE, we say that the graph G admits a RAC_b embedding on S if there is a feasible RAC embedding with at most b bends per edge.

If we insist on straight-line edges, the drawing is completely determined once we have fixed a bijection between vertex set and point set. If we allow bends, however, PSE is also interesting *with mapping*, that is, if we are given a bijection μ between vertex and point set. We call an embedding using μ as the mapping μ*-respecting*. The maximum number of bends per edge in a polyline drawing is the *curve complexity* of the drawing.

Motivation. There are three previous results that motivated us to study RAC and αAC point-set embeddings—even for planar graphs.

- Rendl and Woeginger [RW93] have already considered a special case of the question that we investigate in this chapter, that is, the interplay between planarity and RAC in PSE. They showed that, given a set S of n points in the plane, one can test in $O(n \log n)$ time whether a perfect matching admits a RAC_0 embedding on S. They required that edges are drawn as axis-aligned line segments, that is, they allowed only horizontal and vertical segments. They also showed that if one additionally insists on planarity, the problem becomes NP-hard.

- Pach and Wenger [PW01] showed for the polyline drawing scenario with mapping that, if one insists on planarity, $\Omega(n)$ bends per edge are sometimes necessary even for the class of paths and for points in convex position.

- Cabello [Cab06] proved that deciding whether a graph admits a planar straight-line embedding on a given point set is NP-hard even for 2-outerplanar graphs.

Our Contribution. In order to measure the size of our drawings, we assume that the given point set S lies on a grid Γ of size $n \times n$ where $n = |S|$. We further assume that the points in S are in *general position*, that is, no two points lie on the same horizontal or vertical line. We call S a *1-spaced* $n \times n$ *grid point set*, following previous work of di Giacomo et al. [GFF+13]; a point set is 1-spaced if the horizontal and the vertical distance of each pair of points is at least 1. We require that, in our output drawings, bends lie on grid points of a (potentially larger or finer) grid containing Γ.

We show the following results on RAC and on αAC PSE, which all hold even if the mapping is prescribed—except for the hardness result.

- We modify Cabello's result [Cab06] to show that RAC_0 (and $\alpha\mathrm{AC}_0$) PSE is NP-hard (Theorem 6.1). Hence, we focus on the case with bends in the remaining part of the chapter.

- For any $\varepsilon > 0$, we can find a $(\pi/2 - \varepsilon)\mathrm{AC}_1$ embedding on any 1-spaced $n \times n$ grid point set within area $O(n^2)$ on a grid refined by a factor of $O(1/\varepsilon^2)$ (Theorem 6.2); this area requirement is optimal [GDLM11]. In the planar case, it is NP-hard to decide whether a 1-bend point-set embedding exists—both with [GKO+09] and without [KW02] prescribed mapping.

  Without refining the grid, we get a $(\pi/2 - \varepsilon)\mathrm{AC}_2$ drawing within area $O(nm)$ (Theorem 6.3).

- Every graph with n vertices and m edges admits a RAC_3 embedding on any 1-spaced $n \times n$ grid point set within area $O\big((n+m)^2\big)$ (Theorem 6.4). For being able to find a RAC drawing of arbitrary graphs, curve complexity 3 is needed—even if the point set is not prescribed: Arikushi et al. [AFK+12] showed that in the RAC_1 and RAC_2 style no graphs with more than $6.5n$ and $74.2n$ edges can be drawn, respectively. In the planar case (with mapping), the curve complexity for PSE is $\Omega(n)$ [PW01].

Related Work. Besides the work of Rendl and Woeginger [RW93] that we mentioned above, the study of point-set embeddability has primarily focused on the planar case, in connection with straight-line and polyline edges. As we already mentioned, Pach and Wenger [PW01] showed that there are examples where embedding a path on a set of points in convex positions with a prescribed mapping makes a linear number of bends per edge necessary if the drawing has to be planar. For the setting without prescribed mapping, Kaufmann and Wiese [KW02] showed that it is possible to find a planar embedding of any planar graph on any point set with just two bends per edge. Furthermore, they showed that deciding whether such an embedding with just one bend per edge exists is NP-hard for general planar graphs, while there is always an embedding with one bend per edge for four-connected planar graphs. If the number of bends is not bounded, then any planar graph can be embedded on any point set with any prescribed mapping as Halton showed [Hal91].

In the straight-line drawing style, Bose [Bos02] presented algorithms that embed outerplanar graphs on point sets with improved runtime and space requirement compared to the work of Gritzmann et al. [GMPP91]. Bose et al. [BMS97] developed algorithms for embedding problems of rooted trees.

Efrat et al. [EEK07] showed that point-set embeddability with circular-arc edges and prescribed mapping is NP-hard.

There are also some works on point-set embeddability in the orthogonal drawing style which we will cover in the following chapter, that is, in Chapter 7.

A topic closely related to point-set embeddability are *universal point sets*. A universal point set for planar graphs of n vertices is a point set on which any planar graph of n vertices can be embedded with straight-line edges. For larger n, more than n points are necessary; Kurowski [Kur04] proved a lower bound of $1.235n$ for the size of a universal point set. In a recent work, Bannister et al. [BCDE13] showed how to construct universal point sets of size $n^2/4 - \Theta(n)$ for any n.

Although RAC and αAC drawings have been introduced very recently, there is already a large body of literature on the problem. Regarding the area of RAC drawings, Didimo et al. [DEL11] proved that a RAC_3 drawing of an n-vertex graph uses area $\Omega(n^2) \cap O(m^2)$. Di Giacomo et al. [GDLM11] showed that, for RAC_4 drawings, area $O(n^3)$ suffices and that, for any $\varepsilon > 0$, every n-vertex graph admits a $(\pi/2 - \varepsilon)\mathrm{AC}_1$ drawing within area $\Theta(n^2)$. Our results for RAC_3 and AC_1 drawings (in Theorems 6.4 and 6.2) match the ones cited here, in spite of the fact that vertex positions are prescribed in our case. Van Kreveld [vK10] recently defined and investigated a number of quality ratios between RAC and planar drawings of planar graphs.

6.2 Straight-Line RAC and αAC Point-Set Embeddability

We first focus on the straight-line versions of the embeddability problems, that is, on RAC_0 PSE and on $\alpha\mathrm{AC}_0$ PSE. In the straight-line version, an embedding is completely fixed by the vertex–point mapping. Hence, the versions with prescribed mapping are trivial: We just have to check whether the embedding given by the prescribed mapping is a feasible RAC or αAC drawing by checking all pairs of edges.

In contrast, we can show that RAC_0 PSE and $\alpha\mathrm{AC}_0$ PSE without prescribed mapping are NP-hard. For planar straight-line point-set embeddability NP-hardness was proven by Cabello [Cab06]. We can modify the construction used in his hardness proof so that we see hardness also for PSE with large crossing angles.

Theorem 6.1. *αAC_0 PSE is NP-hard for any constant $0 < \alpha \leq \pi/2$ even for 2-outerplanar graphs and 1-spaced sets of grid points.*

Proof. Cabello [Cab06] proved NP-hardness of planar straight-line PSE of 2-outerplanar graphs by reducing from 3-Partition, which is strongly NP-hard (compare Section 2.3). We modify the point set S used in his reduction for an instance of 3-Partition such that any pair of possible straight-line segments—both defined by a pair of points—crosses at an angle less than α. This will ensure that any $\alpha\mathrm{AC}_0$ embedding is actually planar. The special properties of the point set needed for the reduction must be preserved by our modifications. Additionally, our modified point set must not contain two points on the same grid row or column.

The properties of Cabello's point set S (see Figure 6.1) that are used in his hardness proof are the following.

(1) There are three subsets L, M, and T of points. For each of these sets, all the points lie on one straight line. The three straight lines for L, M, and T are parallel.

(2) There are four special points p_0, p_1, p_2, p_3 with $p_1, p_3 \in M$ and $p_0, p_2 \notin L \cup M \cup T$. Note that $S = L \cup M \cup T \cup \{p_0, p_2\}$.

(3) The relative position (that is, the horizontal order) of the point sets L, $\{p_0\}$, M, T, and $\{p_2\}$ is as shown in Figure 6.1.

(4) The boundary of the convex hull of S contains exactly p_1, p_2, p_3, and all the points in L, that is,
$$\partial\,\mathrm{CH}(S) \cap S = \{p_1, p_2, p_3\} \cup L.$$

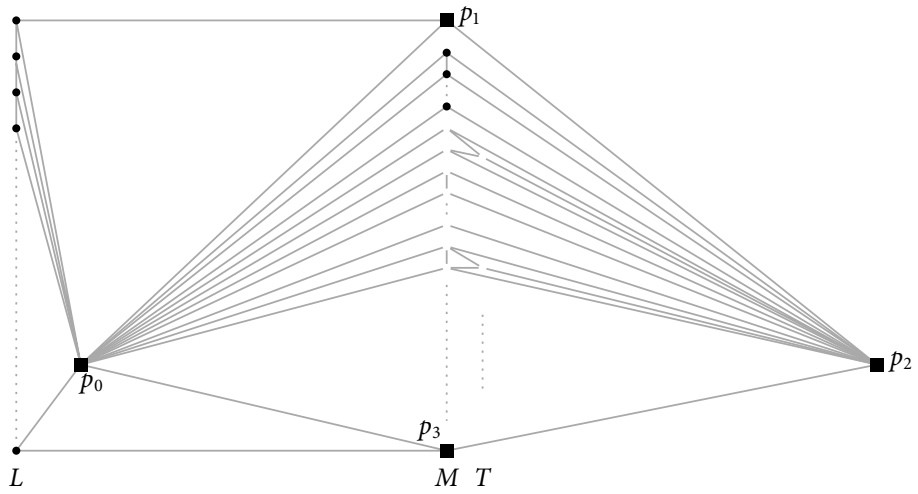


Figure 6.1: Point set and graph of Cabello's hardness proof.

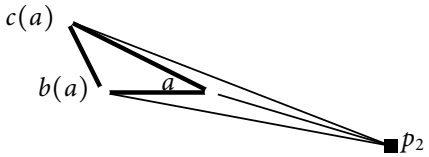


Figure 6.2: K_4 drawn using points a, $b(a)$, $c(a)$, and p_2.

(5) The graph K_4 can be drawn using p_2 and three points $\{a, b, c\} \subseteq M \cup T$ as vertex positions such that no other point is overlapped if and only if $a \in T$, $b \in M$ is left of a on the same height, and $c \in M$ is the upper vertical neighbor of b.

We now transform the point set S such that any two straight lines defined by pairs of points of S cross at an angle less than α.

Let h be the height of the bounding box of S. We treat the three lines containing the points in L, M, and T individually. First, we rotate each line counterclockwise such that the points lie on a diagonal of the grid. Then, we horizontally stretch the grid by an integer factor $w > h \cdot \cot(\alpha/2)$. Finally, we horizontally arrange L, p_0, M, T, and p_2 such that between two sets there is a horizontal gap of width at least w. Now, for any pair of points, the horizontal distance is at least w. Hence, the angle formed by the straight line defined by this pair and a horizontal line is at most $\operatorname{arccot}(w/h) < \alpha/2$, which implies that the crossing angle of two different straight lines is less than α. Hence, any feasible $\alpha\mathrm{AC}_0$ embedding must be planar.

Now, we have to find a position for p_2 that guarantees property (5). For a point $a \in T$, let $b(a) \in M$ be the point directly left of T, and let $c(a) \in M$ be the point directly above $b(a)$ in M; see Figure 6.2. The graph K_4 can be drawn using these points and p_2 if and only if the straight line between p_2 and $b(a)$ and the straight line between p_2 and $c(a)$ do not intersect the triangle $a, b(a), c(a)$. Let a_{bot} and a_{top} be the bottommost and the topmost point of T, respectively.

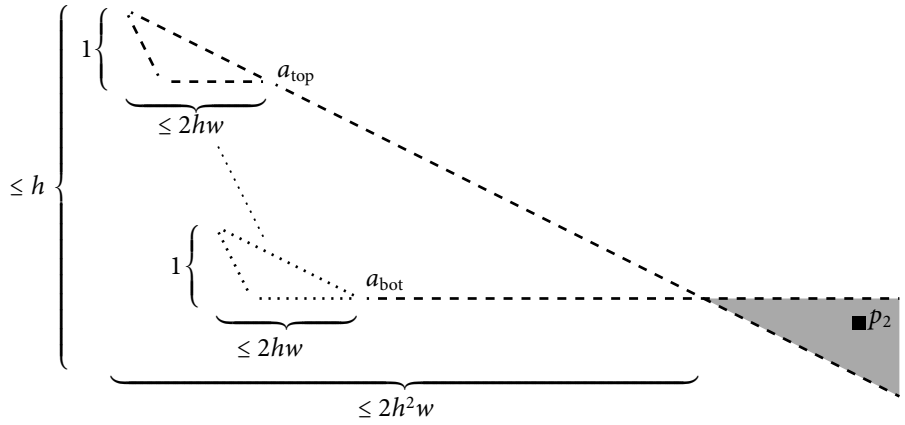


Figure 6.3: Region of feasible positions for p_2.

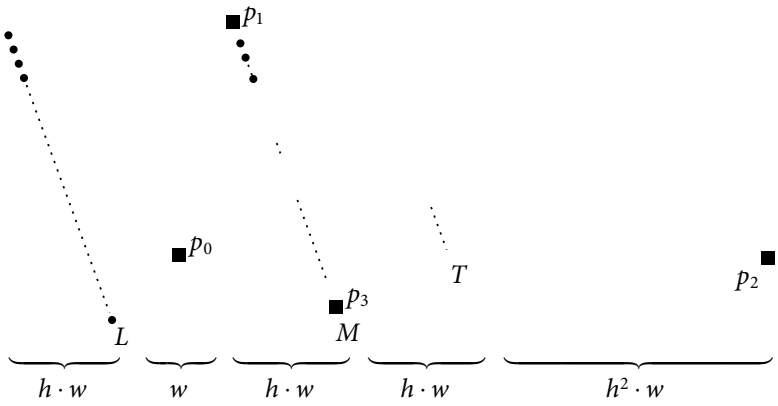


Figure 6.4: Point set with rotated lines and huge horizontal gaps.

Considering the triangles of a_{bot} and a_{top} one can check that p_2 can be placed one row below and more than $2h^2w$ columns right of a_{bot}; see Figure 6.3. Furthermore, if p_2 is in that position, K_4 cannot be drawn on p_2, $a \in T$, and $b, c \in M$ unless $\{b, c\} = \{b(a), c(a)\}$ which means that property (5) holds. Figure 6.4 shows the horizontal arrangement of the new point set.

It is clear that properties (1)–(4) also hold. The only problem is that we have grid rows with multiple points, which means that the point set is not 1-spaced. To avoid this, we first refine the grid by factor 5, that is, we add 4 rows or columns between each consecutive pair of original grid rows or columns, respectively. Then, we move all points in the sets $\{p_0\}$, M, T and $\{p_2\}$ upwards by 1, 2, 3, and 4 units of the new grid, respectively.

Finally, we get a point set S' with properties (1)–(5), that is, we can substitute S by S' in Cabello's hardness proof. All points of S' lie on a grid of size $O(h) \times O(h^2 \cdot w) = O(h) \times O(h^3 \cot(\alpha/2))$, where the height h of Cabello's construction is polynomial in the size of the 3-Partition instance. Furthermore, there is at most one point in any row or column of the grid, that is, S' is 1-spaced.

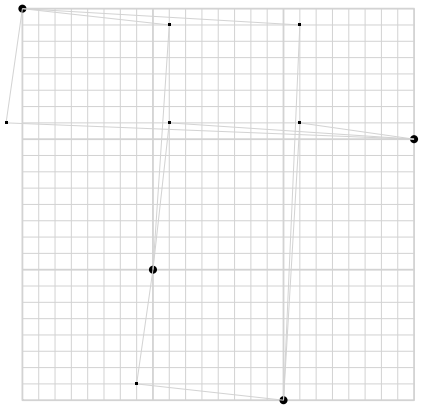


Figure 6.5: Drawing of K_4 on a grid refined by factor $\lambda = 8$.

As any crossing angle for a graph drawn on points of S' is less than α, there is an αAC_0 embedding of the graph on S' if and only if there is a planar straight-line embedding of the graph on S'. Hence, together with properties (1)–(5), an αAC_0 embedding of the graph in Cabello's construction on S' exists if and only if there is a feasible solution for the input instance of 3-Partition. □

6.3 αAC_1 Point-Set Embeddings

As αAC_0 PSE is hard, we now turn to versions in which bends are allowed. We first consider αAC_1 PSE. Recall that we require the bends of edges to lie on grid points.

For drawing arbitrary graphs with right-angle crossings, the RAC_3 style is necessary [AFK+12], that is, we can not draw all graphs in the RAC style with just one or two bends per edge; hence, we first focus on αAC_1 PSE with $\alpha < \pi/2$. The following result shows that with just one bend per edge we can embed any graph on any 1-spaced grid point set if we allow the bends to lie on points of a *refined* grid.

Theorem 6.2. *Let $G = (V, E)$ be a graph with n vertices and m edges, let S be a 1-spaced $n \times n$ grid point set, and let $0 < \varepsilon < \frac{\pi}{2}$. Then G admits a $(\frac{\pi}{2} - \varepsilon)AC_1$ embedding on S (with or without prescribed vertex–point mapping) on a grid that is finer than the original grid by a factor of $\lambda \in O(\cot \varepsilon) = O(1/\varepsilon^2)$.*

Proof. If the mapping $\mu\colon V \to S$ is not given, let μ be an arbitrary mapping. The idea of our construction is as follows. For each edge $e \in E$, we first choose one of the two possible drawings of e with one bend so that both segments lie on grid lines (of the original grid). This yields a drawing of the graph with many overlaps of edges. Then, we slightly twist each edge so that its horizontal segment becomes *almost horizontal*, meaning that it has a negative slope close to 0. At the same time, we make the vertical segment *almost vertical*, meaning it has a very large positive slope; see Figure 6.5.

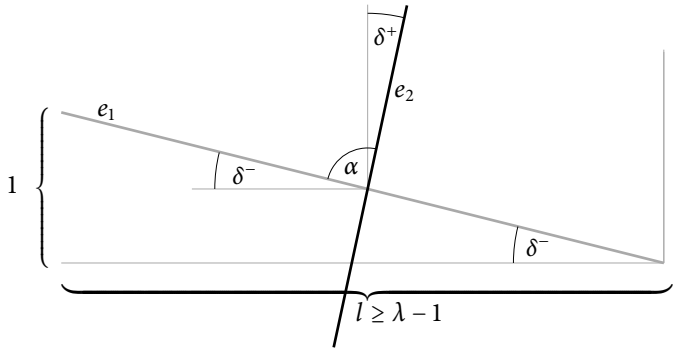


Figure 6.6: Angles in the 2-bend-drawing.

As we want all bends to be on grid points, we first refine the grid by an integral factor of $\lambda = 1 + \lceil \cot \varepsilon \rceil$. We do this by inserting, at equal distances, $\lambda - 1$ new rows or columns between two consecutive grid rows or columns, respectively. Now, a point $s = (a, b) \in S$ lies at position $(\lambda a, \lambda b)$ with respect to the new $\lambda n \times \lambda n$ grid.

Let e be an edge and let (e_x, e_y) be the original position of the bend of e with respect to the new grid. We choose the new position of the bend to be the unique grid point diagonally next to (e_x, e_y) such that the horizontal and vertical segments of e become almost horizontal and almost vertical, respectively. If we apply this construction to all edges, we get a drawing in which none of the almost horizontal and almost vertical segments incident to some vertex v can overlap. Moreover, two almost horizontal or two almost vertical segments incident to different vertices neither overlap nor intersect because S is 1-spaced. Thus, any crossing involves an almost horizontal and an almost vertical segment.

Let e_1 and e_2 be two crossing edges such that the almost horizontal segment involved in the crossing belongs to e_1. We can assume that the smaller angle of the crossing occurs to the top left of the crossing; the other case is symmetric by a rotation of the plane. Let δ^- be the angle formed by the almost horizontal segment of e_1 and a horizontal line, and let δ^+ be the angle formed by the almost vertical segment of e_1 and a vertical line; see Figure 6.6. Then the crossing angle of e_1 and e_2 is $\alpha = \pi/2 - \delta^- + \delta^+ \geq \pi/2 - \delta^-$. For δ^- to be maximal, the horizontal length l of the almost horizontal segment has to be minimal. As this length cannot be less than $\lambda - 1$, we get $\delta^+ \leq \operatorname{arccot}(\lambda - 1) \leq \operatorname{arccot}(\cot \varepsilon) = \varepsilon$. Hence, the crossing angle α is at least $\pi/2 - \varepsilon$. Note that $\cot \varepsilon \in O(1/\varepsilon^2)$. □

Note that we use at most one row or column of the refined grid adjacent to the bounding box of the input points in each of the four directions. Hence, the area requirement is $O\left((n \cdot \cot \varepsilon)^2\right)$ in terms of the finer grid. Di Giacomo et al. [GDLM11] showed that there are graphs for which any $(\pi/2 - \varepsilon)\mathrm{AC}_1$ drawing needs $\Omega(n^2)$ drawing area even without the restriction to an input point set.

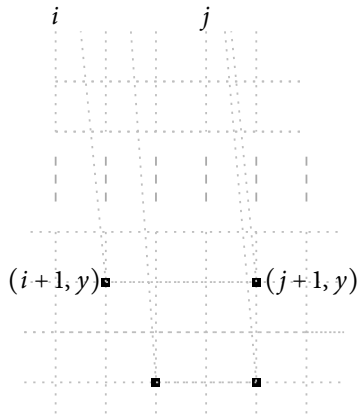


Figure 6.7: Constructing a 2-bend drawing with large crossing angles.

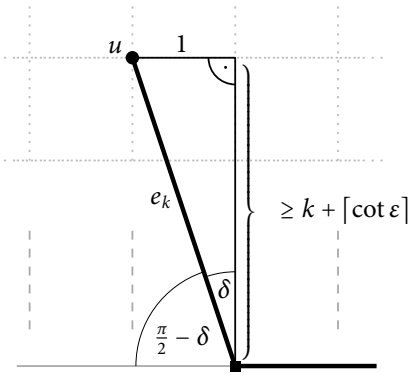


Figure 6.8: Angles in the 2-bend drawing.

6.4 αAC_2 Point-Set Embeddings

We now allow two bends per edge, that is, we move to αAC_2 PSE. Here, we do not need to refine the grid. Similar to αAC_1 PSE, our result holds for both scenarios, with and without given vertex–point mapping. Again, it is not possible to embed any graph with right-angle crossings and just two bends per edge. Hence, we consider the case that $\alpha < \frac{\pi}{2}$.

Theorem 6.3. *Let $G = (V, E)$ be a graph with n vertices and m edges, let S be a 1-spaced $n \times n$ grid point set, and let $0 < \varepsilon < \frac{\pi}{2}$. Then G admits a $(\frac{\pi}{2} - \varepsilon)AC_2$ embedding on S (with or without prescribed vertex–point mapping) within area $O(n(m + \cot \varepsilon)) = O(n(m + 1/\varepsilon^2))$.*

Proof. If the vertex–point mapping $\mu\colon V \to S$ is not prescribed, let μ be an arbitrary mapping. Let $v_1, \ldots, v_n$ be an ordering of V so that $p_i := \mu(v_i)$ has x-coordinate i. Each edge $e = uv$ has exactly two bends, a u-bend and a v-bend, where the u-bend is the bend closer to u when following e from u to v. For $i = 1, \ldots, n$, we place all v_i-bends in column $i + 1$. All middle segments of edges will be horizontal. Thus, the bends for an edge $e = v_i v_j$ are at positions $(i + 1, y)$ and $(j + 1, y)$ in some row $y < 0$ below the original grid; see Figure 6.7. By using a dedicated row for each edge, we achieve that no two middle segments intersect. By construction, no two first or last edge segments intersect. Hence, crossings occur only between the horizontal middle segments and first or last segments of edges. By making the y-coordinates of the middle segments small enough, we will achieve that all crossing angles are at least $\pi/2 - \varepsilon$.

Let $E = \{e_1, \ldots, e_m\}$ be an ordering of the set of edges of G, and let $uv = e_k$ be one of these edges. We set the y-coordinates of the middle segment of e_k to $-k - \lceil \cot \varepsilon \rceil$. Let $e_{k'}$ be an edge whose horizontal segment intersects the first segment of e_k. The crossing angle is $\pi/2 - \delta$, where δ is the angle between the vertical line through the u-bend and the first segment of uv; see Figure 6.8. We have $\delta \le \operatorname{arccot}(k + \lceil \cot \varepsilon \rceil) \le \varepsilon$. Thus, the crossing angle is at least $\pi/2 - \varepsilon$. □

Observe that our area bounds for $\alpha\mathrm{AC}_1$ and for $\alpha\mathrm{AC}_2$ are quite reasonable: for a minimum crossing angle of 70°, the drawings provided by Theorems 6.2 and 6.3 use grids of sizes at most $(3n) \times (3n)$ and $n \times (m+3)$, respectively.

6.5 RAC_3 Point-Set Embeddings

We now allow three bends per edge. Didimo et al. [DEL11] have shown that any graph with n vertices and m edges admits a RAC_3 drawing within area $O(m^2)$—without prescribed point set for the vertices. Their proof uses an algorithm of Papakostas and Tollis [PT00] for drawing graphs such that each vertex is represented by an axis-aligned rectangle and each edge by an *L-shape*, that is, an axis-aligned 1-bend polyline. Didimo et al. turn such a drawing into a RAC_3-drawing by replacing each rectangle by a point. In order to make the edges terminate at these points, they add at most two bends per edge. We now show how to compute a RAC_3-drawing of the same size (assuming $n \in O(m)$)—although we are restricted to the given point set.

Recall that curve complexity 3 is actually necessary for finding a RAC drawing for arbitrary graphs—even without a prescribed point set [AFK+12].

Theorem 6.4. *Let $G = (V, E)$ be a graph with n vertices and m edges and let S be a 1-spaced $n \times n$ grid point. Then G admits a RAC_3 embedding on S (with or without prescribed vertex–point mapping) within area $O\big((n+m)^2\big)$.*

Proof. If the vertex–point mapping $\mu\colon V \to S$ is not given, let μ be an arbitrary mapping. Let $v_1, \ldots, v_n$ be an ordering of V so that $p_i := \mu(v_i)$ has x-coordinate i. We construct a RAC_3 drawing as follows. Each edge has exactly three bends and four straight-line segments. We ensure that intersections involve only the two middle segments of edges, and that these middle segments have only slope +1 or −1.

For an edge uv, we call the bend directly connected to u by a segment the *u-bend*, the bend directly connected to v by a segment the *v-bend*, and the remaining bend the *middle bend*. We start constructing the drawing by placing the v-bends for each vertex v, starting with v_n. We set the y-coordinate y_n of the first v_n-bend to 0. Then, for $i = n, n-1, \ldots, 1$, observe that there are exactly $\deg v_i$ many v_i-bends, which we place in column $i+1$ starting at y-coordinate y_i below the $n \times n$ grid using positions $\{(i+1, y_i), (i+1, y_i-2), (i+1, y_i-4), \ldots, (i+1, y_i-2\cdot(\deg v_i - 1))\}$; see Figure 6.9. We connect each vertex with its associated bends without introducing any intersection since we stay inside the area between columns i and $i+1$. We set $y_{i-1} = y_i - 2 \cdot (\deg v_i - 1) - 3$. If v_i has degree 0, we do not place bends but set $y_{j-1} = y_j - 3$ to avoid overlaps and crossings. Then we continue with v_{i-1}.

Since we place the bends from right to left and from top to bottom by moving our "pointer" by L_1- (or Manhattan) distances 2 or 4, each pair of these bends has even Manhattan distance. To draw an edge uv, we first select a "free" u-bend position and a free v-bend position. The two middle segments go to the right at slopes +1 and −1. Since u- and v-bend have even Manhattan distance, the middle bend has integer coordinates, that is, it lies on a grid point.

Let u and v be two vertices with u-bend b_u of some edge uu' and v-bend b_v of some edge vv', respectively. The segments $\overline{ub_u}$ and $\overline{vb_v}$ cannot intersect; we want to see that the middle segment starting at b_u also cannot intersect $\overline{vb_v}$. Such an intersection can only occur if u lies to the left of v and the middle segment lies above b_v. In this case, b_v lies above b_u with a y-distance that is

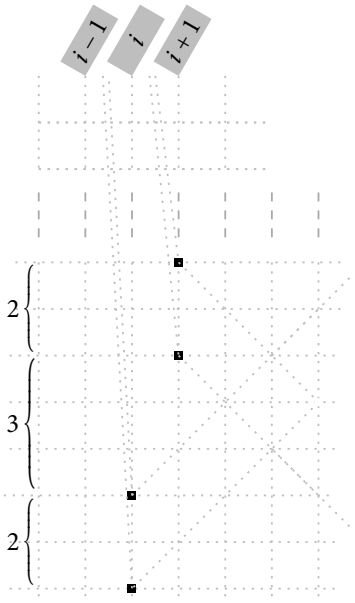


Figure 6.9: Construction of a RAC_3 embedding.

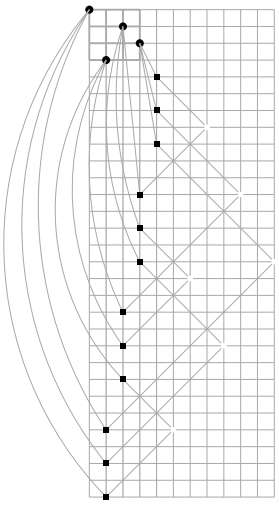


Figure 6.10: RAC_3 embedding of K_4 as in the proof of Theorem 6.4; some straight-line segments have been replaced by circular arcs for the sake of clarity.

greater than their x-distance. As all middle segments have a slope of at most +1, b_v lies above the relevant middle segment, which, hence, cannot intersect $\overline{vb_v}$.

Therefore, an intersection can only occur between two middle segments, one with slope +1 and one with slope −1. Such segments always intersect at an angle of 90°.

It remains to prove the space limitation. Clearly, the drawing of any edge requires not more horizontal than vertical space. On the other hand, for any vertex v, we need at most $2 \cdot \deg v + 3$ rows below the grid, resulting in a total vertical space requirement of $O(n + m)$. This completes the proof. □

Figure 6.10 shows an embedding of K_4 on a given point set created using our construction.

6.6 Concluding Remarks

In this chapter, we have opened an interesting new area: the intersection of point-set embeddability and drawings with crossings at large angles. We have done first steps in investigating the problems RAC PSE and αAC PSE.

First, we have shown that the straight-line version $\alpha\mathrm{AC}_0$ point-set embeddability is NP-hard for any minimum crossing angle α. For the versions with bends, we have seen that any graph allows an $\alpha\mathrm{AC}_2$ embedding on any point set—even if the vertex–point mapping is prescribed.

This is also true for αAC_1 PSE; in this case, however, we had to use a refined grid in order to be able to place all bends on grid points.

In the RAC drawing style, we could show that any graph can be embedded on any point set even with prescribed embedding using three bends per edge. Furthermore, the drawing area requirement of the drawings of our algorithm is asymptotically worst-case optimal—even for drawings without the restricted input point set.

Open Problems. While we did first steps in investigating RAC and αAC PSE, there are also several open problems.

First, for being able to find αAC_1 embeddings for all graphs and place the bends on grid points, we refined the input grid. Is it also possible to find such embeddings without refining the grid? This seems relatively unlikely, at least if we do not want to leave the bounding box of the input points.

In the RAC setting, Di Giacomo et al. [GDLM11] have shown that any n-vertex graph admits a RAC_4-drawing that uses area $O(n^3)$. Can we achieve the same bound in the PSE setting?

It is known that less than three bends per edge do not suffice for being able to draw all graphs with right-angle crossings. Hence, it would be interesting to know the complexity of RAC_2 PSE and RAC_1 PSE. In other words: Can we efficiently test whether a given graph has a RAC2 embedding (or RAC_1 embedding) on a given 1-spaced $n \times n$ grid point set? If this is the case, can we minimize the drawing area?

Chapter 7

Orthogonal Point-Set Embeddability on the Grid

In this chapter, we investigate planar and nonplanar point-set embeddability in the orthogonal drawing style on an input grid. In the orthogonal drawing style, edges must be drawn as polylines consisting of horizontal and vertical straight-line segments. Hence, if we allow crossings, any crossing angle will naturally be a right angle. Orthogonal drawings are often realized on a grid where not only the vertices have to be placed on grid points, but also the bends of edges. Therefore, all edge segments lie on grid lines. We also demand this for feasible orthogonal point-set embeddings.

As our main-result, we show that orthogonal point set embeddability without prescribed vertex–point mapping is NP-hard, no matter whether or not we enforce planarity and no matter how many crossings we allow per edge. We also consider problem variants with prescribed mapping. While some cases are tractable, others are still NP-hard; in particular, planar orthogonal point-set embeddability without any restriction on the number of bends is NP-hard. We also consider the problems on 1-spaced point sets, where some variants become tractable.

7.1 Introduction

Orthogonal drawings are very popular. In the orthogonal drawing style, edges are drawn as polylines, with the additional requirement that any segment of an edge is horizontal or vertical. Hence, at any bend, there is a 90°-turn. Due to the schematized appearance, the orthogonal drawing style is often used for technical visualizations such as UML diagrams or drawings of electric circuits.

Also for orthogonal drawings, one may often want to place the vertices at specific positions. Hence, point-set embeddability—as introduced in the previous chapter—is worth to be investigated for the orthogonal drawing style. We will do this in this chapter. We will consider both planar and nonplanar orthogonal point-set embeddability. Note that, if we do not insist on planarity, we automatically get only right-angle crossings. Hence, nonplanar orthogonal PSE is a restricted version of RAC PSE, which was presented in the previous chapter.

As before, we insist that all bends are placed at integer coordinates, that is, on grid points. This implies that any edge segment lies on a grid line. Hence, we speak of *orthogonal PSE on the grid*.

Problem Definition. The input of all problem variants are a graph $G = (V, E)$ of maximum degree four and a set S of $n = |V|$ points on the integer grid. Note that we normally do not require the point set to be 1-spaced. If we want to specify the precise position of each vertex, the input also contains a *mapping* μ; recall that the mapping is a bijection $\mu\colon V \to P$. We first define nonplanar orthogonal point-set embeddability (NPO PSE).

Definition 7.1 (NPO_b PSE)**.** Given an n-vertex graph $G = (V, E)$ and a set S of n grid points, determine whether there exists a bijection $\mu\colon V \to S$, and an orthogonal drawing of G so that each vertex v is mapped to $\mu(v)$ and each bend lies on integer coordinates. If such a drawing exists and the largest number of bends per edge in the drawing is b, we say that G *admits an* NPO_b embedding on S.

The variant *planar orthogonal point-set embeddability* (PO PSE) is defined analogously; the only difference is that crossings are forbidden.

Definition 7.2 (PO_b PSE)**.** Given an n-vertex graph $G = (V, E)$ and a set S of n grid points, determine whether there exists a bijection $\mu\colon V \to S$, and a *planar* orthogonal drawing of G so that each vertex v is mapped to $\mu(v)$ and each bend of an edge lies on integer coordinates. If such a drawing exists and the largest number of bends per edge in the drawing is b, we say that G *admits a* PO_b embedding on S.

We also use the variants PO_∞ and NPO_∞ PSE in which there is no restriction for the number of bends. As for RAC PSE and αAC PSE, we also speak of μ-respecting embeddings if the vertex–point mapping μ is prescribed.

Some problem variants become easier for 1-spaced point sets. In this chapter, we normally do not assume that the point set is 1-spaced.

Previous Work. In Section 6.1 we did already discuss previous work on point-set embeddability. The work of Rendl and Woeginger [RW93] is worth to mention again. Recall that they considered straight-line orthogonal point-set embeddability, without prescribed mapping, for matchings. They showed that, given a set S of n points in the plane, one can decide in $O(n \log n)$ time whether a perfect matching admits a NPO_0 embedding on S. For the planar version, they showed that PO_0 PSE is NP-hard for matchings.

O'Rourke [O'R88] showed that a set of orthogonal polygons can uniquely be reconstructed from its vertex set in $O(n \log n)$ time. If the point set may also contain interior points of edges, the problem does, however, become NP-hard as Rappaport showed [Rap86]; this also implies hardness of PO_0 PSE without prescribed mapping.

Rahavan et al. [RCS86] considered a problem that is equivalent to PO_1 PSE with prescribed mapping for perfect matchings; they were able to solve the problem in quadratic time. We will later generalize their result to all graphs and also nonplanar embeddings.

A special case of orthogonal drawings are *Manhattan-geodesic* drawings which require that the edges are shortest orthogonal connections, that is, monotone chains of axis-parallel line segments. This convention was recently introduced by Katz et al. [KKRW10]. As one of their main results, they proved that planar Manhattan-geodesic point-set embeddability with prescribed mapping on the grid—that is, a restricted variant of PO_∞ PSE—is NP-hard even for matchings. In the setting without prescribed mapping, they proved that Manhattan-geodesic PSE is NP-hard even

for subdivisions of cubic graphs. On the other hand, they provided an $O(n \log n)$ decision algorithm for the n-vertex cycle. Without the grid restriction, that is, if bends may be placed anywhere, the problem can be solved efficiently for perfect matchings as the showed.

Di Giacomo et al. [GFF+13] investigated Manhattan-geodesic point-set embeddability of trees. They showed that all caterpillars of maximum degree 3 are PO_1 embeddable on any 1-spaced point set; moreover, they showed that all binary trees are NPO_1 embeddable on any such point set, which was independently proven by us. The planar version PO_1 PSE of trees with one bend per edge was also considered by Kano and Suzuki [KS12] who showed that such an embedding can always be found for some special binary trees if the point set is 1-spaced. Kano and Suzuki further showed that any cycle and any spider of maximum degree 4 admits a PO_1 embedding on any 1-spaced point set.

Chowdhury and Rahman [CR11] considered planar orthogonal PSE without the restriction of bends and points to grid points; in this setting, edges can come arbitrarily close to each other. They also did not restrict the number of bends per edge, that is, they worked with PO_∞ drawings. Both for 3-connected cubic planar graphs with Hamiltonian cycle and for 4-connected planar graphs they gave algorithms for embedding graphs on any point set. In both cases, they could show that the total number of bends is linear.

We also want to mention some work on planar orthogonal drawings without the restriction to a given point set. As a well-known result, Tamassia [Tam87] showed how to efficiently find an orthogonal drawing of a graph with a planar embedding that minimizes the total number of bends with respect to the prescribed embedding. Note that this task is NP-hard if the embedding is not prescribed as Garg and Tamassia showed [GT01]. Tamassia's approach is based on first computing an extended embedding, called *orthogonal representation* in which additional information on the turns in bends is stored. He then transforms the orthogonal representation into an orthogonal drawing on the grid, that is, with our restriction that bends lie on grid points, using a heuristic. Later, Patrignani [Pat01] showed that finding an orthogonal embedding on the grid using minimum area is NP-hard for a planar graph with fixed orthogonal representation; this problem is known as orthogonal compaction.

Bekos et al. [BKK+13] recently suggested improving the readability of nonplanar orthogonal drawings by forcing segments with crossings to run diagonally. They argued that this helps the viewer of such drawings to distinguish more easily between vertices and crossings. They called the resulting drawing style the *slanted orthogonal* drawing style.

Our Contribution. We first focus on the problem versions with prescribed mapping. In this setting, the straight-line versions PO_0 and NPO_0 PSE are trivial. For the versions with one bend per edge, that is, PO_1 PSE and NPO_1 PSE, we develop an algorithm that decides embeddability in quadratic time (Section 7.2). This generalizes a result of Raghavan et al. [RCS86] for matching graphs.

We then move to problem versions with more bends (Section 7.3). We are able to show that NPO_2 PSE and NPO_3 PSE with prescribed mapping are NP-hard even if the graph is a path. In contrast, for 1-spaced point-sets we can always find an NPO_2 embedding for graphs of maximum degree 3. For NPO_3 PSE this is even true for any graph of maximum degree 4. Moreover, We show that finding an embedding of minimum area is NP-hard for paths even if all points lie on the x-axis.

In the planar setting, we show hardness of PO_∞ PSE with prescribed mapping by modifying a hardness proof for Manhattan-geodesic PSE by Katz et al. [KKRW10] (Section 7.4).

As our main result, we show that all problem variants are NP-hard if the mapping is not part of the input, independent of the maximum number of bends and of whether or not crossing are allowed (Section 7.5). For the problem versions with up to one bend per edge, that is, for PO_0, PO_1, NPO_0, and NPO_1 PSE, we can even strengthen this result and show hardness for outerplanar graphs of maximum degree 3.

Finally, we show that, in the version without prescribed mapping, every n-vertex binary tree admits a NPO_1 embedding on any 1-spaced point set (Section 7.6). We slightly extend this result to graphs of maximum degree 3 that arise when replacing the vertices of a binary tree by cycles.

7.2 Orthogonal Point-Set Embeddability with at most One Bend per Edge

We first consider the variant of orthogonal PSE with prescribed mapping, that is, we know the exact position of each vertex. If we do not allow bends in edges, then the drawing is completely determined by the positions of the vertices. Hence, the only possible drawing either is a feasible PO_0 or NPO_0 drawing, or there is no such drawing. For deciding whether the drawing is a feasible NPO_0 drawing, we just have to check that each edge is horizontal or vertical and does not contain a vertex except for its two endvertices. For PO_0 we additionally have to check that no pair of edges crosses. Hence, we can observe that the versions without bends are trivial.

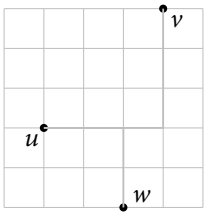
Observation 7.1. *PO_0 and NPO_0 PSE with prescribed mapping are trivial.*

We now move to the versions in which at most one bend per edge is allowed. Suppose that we have an edge $e = (u, v)$. If u and v have either the same x-coordinate or the same y-coordinate, then the only possibility for drawing e is to connect the vertices by a straight-line segment with no bends. If, however, u and v have no common coordinate, we must draw the edge with one bend; there are always two possibilities for connecting two vertices with one bend. Hence, the problem is not trivial. However, we can still decide whether a feasible embedding exists. We first show how to do this for the planar version PO_1 PSE.

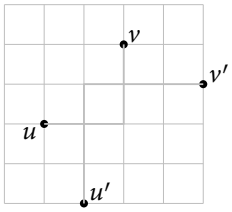
Theorem 7.1. *Let G be an n-vertex graph of maximum degree 4, let S be a set of n points on the grid, and let μ be a vertex–point mapping. We can decide in $O(n^2)$ time whether G admits a μ-respecting PO_1 embedding on S and, if it does, construct such an embedding within the same time bound.*

Proof. We use a 2SAT encoding to solve the problem. A similar approach was used by Raghavan et al. [RCS86] to deal with the special version in which G is a matching. We first assume that the point set is 1-spaced, that is, we never can draw an edge without a bend.

We associate a boolean variable x_{uv} with each edge uv of G. The two possible drawings of the edge uv correspond to the two literals x_{uv} and $\neg x_{uv}$. If S is 1-spaced, only drawings of edges incident to the same vertex can possibly overlap.



(a) Edges overlap in the current configuration.



(b) Edges cross in the current configuration.

Figure 7.1: Overlapping and crossing pairs of orthogonal edges with one bend.

Now, we construct a 2SAT formula ϕ as follows. Consider a pair of drawings of edges uv and uw that overlap; see Figure 7.1a. There are two literals, for instance, x_{uv} and $\neg x_{vw}$, that correspond to the two edge drawings. We add the clause

$$\neg(x_{uv} \wedge \neg x_{uw}) \equiv (\neg x_{uv} \vee x_{uw})$$

to ϕ. This clause ensures that no satisfying truth assignment for ϕ can result in the forbidden combination of drawings for uv and vw. We add such clauses for each pair of overlapping edge drawings. Then, ϕ contains at most $n \cdot \binom{4}{2} \cdot 4 = O(n)$ clauses because G has maximum degree 4.

Next, suppose that the edges $e = uv$ and $e' = u'v'$ do not share a vertex. Clearly, the edges cannot overlap, but they can cross in some of their four possible combinations of drawings; see Figure 7.1b. Suppose that there is a pair of drawings of e and e' that cross. Let, for example, $l_e = \neg x_e$ and $l_{e'} = x_{e'}$ be the literals corresponding to the edge drawings. Then, we add the clause

$$\neg(l_e \wedge l_{e'}) \equiv (\neg l_e \vee \neg l_{e'}) \equiv (x_e \vee \neg x_{e'}).$$

The clause ensures that no satisfying truth assignment can lead to the forbidden combination of drawings of e and e'. We add clauses of this type for each pair of edge drawings that result in a crossing.

Now, we can drop the assumption that the point set is 1-spaced. Assume that there is an edge that must be drawn as a straight-line segment and contains another vertex in its interior. Then, we immediately now that there is no feasible solution. The same holds if there is a pair of edges e and e' that must be drawn as straight-line segments and whose straight-line segments cross. Therefore, we assume that both cases do not occur.

We do not add variables for straight-line edges because their drawing is fixed. However, such edges may still cross or overlap with an edge e with one bend. Furthermore, a drawing of some edge e with a bend may have another vertex in its interior. In both cases, we add a clause $(\neg l_e)$ with a single literal, where l_e is the literal corresponding to the drawing of e with the infeasible configuration, in order to forbid the configuration. Finally, note that when checking whether two edge drawings cross, we must now also check for overlaps instead of just proper intersections.

As we added clauses forbidding each combination of edge drawings if and only if the combination causes an overlap or a crossing, it is easy to see that ϕ is satisfiable if and only if G has a μ-respecting PO_1 embedding on S.

Recall that the maximum degree of G is 4. Hence, there are only $O(n)$ edges. When adding clauses, we checked any pair of edge drawings, and any pair of an edge drawing and a point. This takes $O(n^2)$ time and results in $O(n^2)$ clauses. Since the satisfiability of a 2SAT formula can be decided in time linear in the number of clauses [EIS76], we need $O(n^2)$ time for deciding whether a PO_1 point-set embedding exists. If we find a satisfying truth assignment, we can create a feasible drawing by choosing the drawing described by the truth value of the variable corresponding to the edge, for each edge with a bend. □

Now, we can consider NPO_1 PSE. Here, we can basically reuse the 2SAT model developed for the planar case. The only necessary modification is that we have to drop all clauses that model crossings; we just keep clauses modeling overlaps. Hence, we get the following result.

Theorem 7.2. *Let G be an n-vertex graph of maximum degree 4, let S be a set of n point on the grid, and let μ be a vertex–point mapping. We can decide in $O(n^2)$ time whether G admits a μ-respecting NPO_1 embedding on S and, if it does, construct such an embedding within the same time bound.*

Note that for 1-spaced point sets we just have to add clauses that model overlaps of edges incident to the same vertex. Because there is just a linear number of such clauses, NPO_1 PSE can be solved in linear time on 1-spaced point sets.

7.3 Orthogonal Point-Set Embeddability with Two or Three Bends per Edge

In the previous section, we have seen that both PO_1 PSE and NPO_1 PSE with prescribed mapping are easy. In contrast, we will show that NPO_2 PSE and NPO_3 PSE with prescribed mapping are hard on general point sets. Then, we will see that any graph of maximum degree 4 allows an NPO_3 embedding with prescribed mapping on any 1-spaced point set. For NPO_2 PSE, this is at least true for graphs of maximum degree three.

7.3.1 General Point Sets

We will now show that NPO_2 PSE with prescribed mapping is NP-hard. Furthermore, this holds even if we restrict the problem to graphs that are simple paths.

Theorem 7.3. *NPO_2 PSE with prescribed mapping is NP-hard even for a single path.*

Proof. We first show the result for a collection of paths. Our proof is by reduction from 3SAT (compare Section 2.3). We model each variable of a given 3SAT instance by a horizontal path, called *variable path*, along the x-axis where all edges are drawn with exactly two bends and one horizontal segment above or below the x-axis. We can enforce this drawing style by placing dummy nodes to the left and to the right of each point of the path, see Figure 7.2.

Note that there are exactly two drawing styles for the path depending on whether the leftmost edge is drawn above or below the x-axis. We associate edges below the x-axis with `true` and edges above the x-axis with `false`. For each variable path we number the edges from left to right. Odd edges represent positive literals and even edges represent negated literals. We call

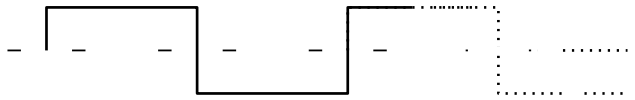
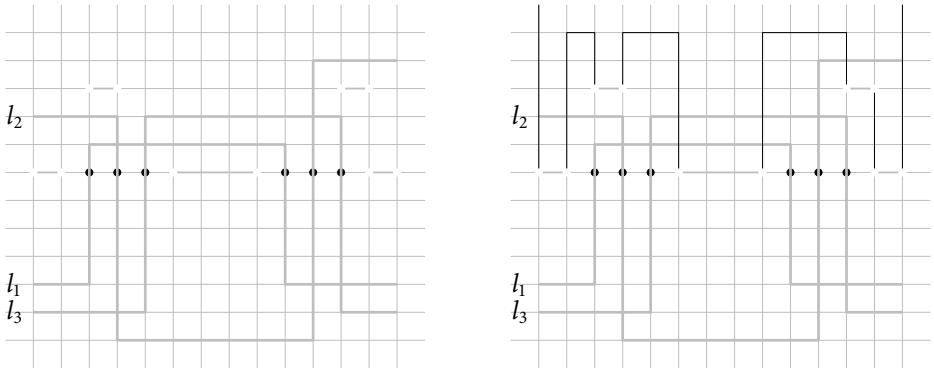


Figure 7.2: Representing a variable as an alternating path.



(a) $l_1 = l_3 =$ `false` and $l_2 =$ `true`

(b) Making the graph connected.

Figure 7.3: Drawing of the clause gadget for clause $C = (l_1 \vee l_2 \vee l_3)$.

the five points used to start a new edge of the path a *turn block*. We place the necessary turn blocks for the variable paths of all variables on the x-axis.

We now want to model a clause $C = (l_1 \vee l_2 \vee l_3)$ with literals l_1, l_2, and l_3 by inserting a gadget around the x-axis. Depending on whether variable v_i of literal l_i is negated or unnegated, we place either one or two turn blocks for v_i before the path of v_i enters the gadget for C. Note that the edge of v_i lying completely inside the gadget is the one modeling the literal, that is, if we need an odd edge (for a positive literal) in the gadget, we have to enter C with an even edge and vice versa. Figure 7.3a shows a clause gadget where two of the three literals are `false`.

Clearly, the gadget cannot be drawn if all three internal edges are above the x-axis, that is, if all three literals are false. On the other hand, it can easily be checked that drawing the clause is always possible if at least one internal edge is below the axis, that is, if the corresponding literal is `true`.

Hence, there is a feasible NPO_2 embedding if and only if there is a satisfying truth assignment for the 3SAT instance. Furthermore, the instance has linear size because we just need a constant number of points and vertices for any clause.

So far, the NPO_2 PSE instance that we constructed consists of a collection of paths. We now show how to modify graph and point set so that we can see that the problem is NP-hard even for a single path. First, we connect the isolated edges used for gadgets and turn blocks as indicated in Figure 7.3b. Second, we extend each variable path by an additional point directly next to the left and right endpoint, respectively. Now, we have a collection of $n + 1$ paths, which we can easily connect to a single path. It is easy to see that all new edges can be drawn above or on the axis if the 3SAT instance is satisfiable. □

Now, consider again the variable paths that we used in the reduction. Recall that we blocked any vertex of a variable path by placing vertices on the next grid points to the left and to the right, so that any edge of the variable path can leave its vertices only vertically. Hence, it is not possible to draw an edge of a variable path with three bends because this would involve leaving one of the vertices to the left or to the right. Therefore, there is no feasible NPO_3 embedding if there is no feasible NPO_2 embedding.

Theorem 7.4. *NPO_3 PSE with prescribed mapping is NP-hard even for a single path.*

7.3.2 Area Minimization

Suppose that we already know that an embedding exists. Using a similar idea as in the previous proof, we can show that finding an area-minimum embedding is NP-hard. Again, this holds even for paths.

Theorem 7.5. *Area-minimization for NPO_2 PSE with prescribed mapping is NP-hard even for a single path and all points lying on the x-axis.*

Proof. Our reduction is from Not-All-Equal 3SAT; compare Section 2.3. Recall that an instance of this version of 3SAT is satisfiable if and only if there exists a truth assignment to the n variables so that each clause contains a literal that evaluates to `true` and a literal that evaluates to `false`. As in the proof of Theorem 7.3, variables are modeled by variable paths, that is, by alternating paths along the x-axis. Note that in our setting, minimizing the area is equivalent to minimizing the height of the drawing.

Again, we can enforce the right combination of odd and even edges of the variable paths—corresponding to unnegated or negated variables—by inserting turn blocks before entering a clause gadget, if necessary. Our clause gadgets consist of two turn blocks for each of the three literals of the clause, positioned symmetrically so that each path can leave the gadget on the same row on which it entered the gadget.

As Figure 7.4 shows, a clause gadget can be drawn using at most five grid rows if and only if not all literals have the same truth value; otherwise, six rows are needed. In the x-interval spanned by the clause gadget there exist also $n-3$ additional edges for the remaining variables. To create a bottleneck for the height of the drawing, we add a matching of $n-2$ edges that all start immediately to the left of the clause gadget and end immediately right of it. Now, the clause gadget always occupies a height of at least $(n-3)+6+(n-2)=2n+1$ if all literals have the same truth value.

To complete the proof, we have to show that a drawing of the graph with height at most $2n$ exists if the given instance of Not-All-Equal 3SAT is satisfiable. Suppose that we have an assignment of truth values to the variables such that there is no clause in which all literals have the same value. The truth value of a variable decides whether the first edge of its variable path is drawn above or below the x-axis. Then, the side is fixed for all other edges, too. For each variable, we reserve one row above the axis, and one row below the axis, using the n rows directly below and the n rows directly above the x-axis. We draw all edges of variable paths that lie not completely inside a clause gadget in the reserved row on the respective side. This can be done without any overlapping.

For an edge representing a literal of a clause, we may reuse the row of another literal to save a row, as shown in Figure 7.4. By doing so there are at most $n-3+5=n+2$ rows occupied inside

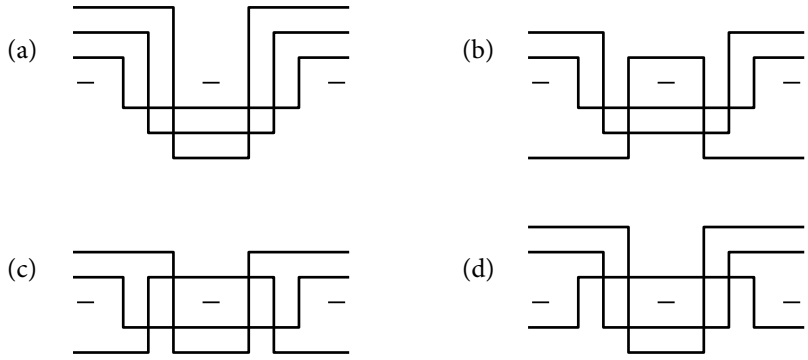


Figure 7.4: Drawings of a clause gadget where (a) all literals have the same truth value or (b–d) one literal has a different truth value than the others. In the given examples, in (a) all literals are `true` and in (b–d) only one is `false`. The cases where two literals are `false` are symmetric to (b–d) and the case where all literals are `false` is symmetric to (a).

each clause gadget. We can add the additional $n-2$ matching edges using only the unoccupied $n-2$ rows. By this construction, we create a drawing where exactly the n rows directly above and the n rows directly below the x-axis are occupied. This completes the reduction.

Again, it is easy to modify the construction so that the graph is a single path, in a similar way as we did in the proof of Theorem 7.3. We just have to be careful when connecting the two blocking dummy vertices inside a gadget to the left and to the right: The two edges for connecting them need one additional row. Hence, they have to replace one of the matching edges. □

Also for this proof, we can observe that allowing an additional bend per edge does not help us when trying to find feasible NPO_3 embeddings. Hence, the problem remains NP-hard.

Theorem 7.6. *Area-minimization for NPO_3 PSE with prescribed mapping is NP-hard even for a single path and all points lying on the x-axis.*

A Tractable Case. We now consider graphs of maximum degree one, that is, perfect matchings, in the same setting; that is, all vertices lie on the x-axis. The big difference to the case with vertices of degree 2 is that we do not have to care about overlapping edge segments incident to a vertex. Suppose that we have a feasible drawing that uses space below the x-axis. Take all edges whose middle segment is in the bottommost row; we can redraw these edges such that the middle segments are drawn one row above the topmost row used, that is, above the x-axis. By repeating this, we get a drawing of the same area with no space used below the x-axis. Especially, we can see that there always is an area-minimum embedding that does not use space below the x-axis. We now show how we can minimize the area of such drawings.

Theorem 7.7. *Let S be a set of n points on the x-axis, let $G=(V,E)$ be a matching consisting of $n/2$ edges, and let μ be a vertex–point mapping. An area-minimum μ-respecting NPO_2 embedding of G on S can be computed in $O(n^2)$ time.*

Proof. If S contains pairs of neighboring points that correspond to edges of the given matching, we connect each of these pairs by a (horizontal) straight-line segment. Furthermore, we can restrict ourselves to drawings in which all middle segments of the remaining edges are placed above the x-axis. Hence, for drawing any of the remaining edges of the matching, we must connect its endpoints by two vertical segments leaving the x-axis to the top and a horizontal middle segment that connects the vertical segments. As G is a matching, only horizontal segments can overlap. In order to minimize the drawing area, we, thus, have to minimize the number of rows, the *layers*, needed for drawing the middle segments of all edges without overlap.

Let $G' = (V', E')$ with $V' = E$; E' contains an edge for each pair of edges of G that cannot use the same layer. Assigning the edges of G to the minimum number of layers is equivalent to coloring the vertices of G' with the minimum number χ' of colors.

Let e_1, e_2 be two edges. The horizontal segment of e_1 cannot be drawn in the same layer as the horizontal segment of e_2 if and only if the x-intervals of e_1 and e_2 intersect. Hence, the graph G' is an *interval graph*: for an edge uv of G—a vertex of G'—the interval is $[x(\mu(u)), x(\mu(v))]$. Two edges must be placed in different layers if their intervals intersect. Thus, a coloring of G' using χ' colors can be computed in $O(|V'| + |E'|) = O(n^2)$ time using the coloring algorithm of Olariu for interval graphs [Ola91]. This coloring yields an assignment of the edges to the minimum number of layers, which in turn corresponds to a minimum-area NPO_2 drawing: we simply use the first χ' horizontal grid lines immediately above the x-axis for the layers of the horizontal edge segments. □

An Approximation. We can use the algorithm for matchings for deriving a 2-approximation for graphs with vertices of degree 2; recall that finding an optimum solution is NP-hard even for a single path. Let $G = (V, E)$ be a *connected* graph of maximum degree 2, let S be a set of $n = |V|$ integer points on the x-axis, and let μ be a vertex–point mapping. The graph G can either be a cycle or a path. Recall that, in order to avoid overlaps we need to place two edges incident to the same vertex on different sides of x-axis. Thus, if G is a cycle of odd length, then G does not have an NPO_2 drawing. On the other hand, if G is a cycle of even length or a path, then G can be drawn by alternately drawing the edges above and below the x-axis just as we did for variable paths in the hardness proof. We can, therefore, decompose G into two matchings, one of which we will draw above and one of which we will draw below the x-axis. Then we apply the interval-graph technique from the proof of Theorem 7.7, draw each matching on its side of the x-axis, and get a minimum-area NPO_2 embedding of G.

However, this does not work for unconnected graphs as the choice of placing edges above or below the x-axis can be made independently for the connected components, although edges of different connected components might interfere with each other regarding the layer assignment. Instead, we suggest using the following 2-approximation algorithm.

Theorem 7.8. *Let $G = (V, E)$ be a graph of n vertices of maximum degree 2 and no cycles of odd length, let S be a set of n grid points on the x-axis, and let $\mu\colon V \to S$ be a mapping. Algorithm 7.1 computes a 2-approximation for a μ-respecting NPO_2 embedding of minimum area.*

Proof. It is easy to see that the algorithm places the edges such that no two edges incident to the same vertex are placed on the same side of the x-axis. As the layering of a solution for the 1-sided problem in the split graph is used, there is also no overlapping of two middle segments. Hence, the algorithm creates a μ-respecting NPO_2 embedding.

foreach node v with $\deg v = 2$ **do**
  Split v into two nodes v_1, v_2 of degree 1.
Solve the 1-sided problem optimally on the split instance.
foreach connected component C (cycle or path) **do**
  Choose some arbitrary edge e of C.
  Draw e above the x-axis; for the rest of the edges of C the side is automatically fixed.
  foreach edge e' of C **do**
    Draw e' in the layer defined by the optimum 1-sided solution on its respective side.

Algorithm 7.1: 2-approximation for 2-sided area minimization.

Let OPT be the height of an optimum solution. We can construct a feasible solution for the 1-sided problem on the split graph by fixing one side of the x-axis, say the part above the axis, and moving all middle segments below the axis to the fixed side above, occupying the same number of new layers above the axis that were used below the axis. By this construction we get a solution of height OPT for the 1-sided problem. The height of this solution cannot be less than the height of the optimum solution we used in the algorithm. As we doubled this solution, we get an approximation factor of 2 for the 2-sided problem. □

7.3.3 1-Spaced Point Sets

We now move to 1-spaced point sets. Consider, for a moment, a specialized NPO_2 drawing convention that requires the first and the last (of the three) segments of an edge to go in the same direction—a *bracket* drawing. If we do not restrict the drawing area, then the problem of finding a bracket embedding of a graph G on a 1-spaced set of n grid points is equivalent to finding a 4-edge coloring of G. The idea is that the four colors encode the direction of the first and last edge segment—going up, down, left, or right—and that the middle segment is drawn sufficiently far away from the input point set. The edge coloring ensures that no two edges incident to the same vertex overlap. By Vizing's theorem [Viz64], we know any graph of maximum degree 3 is 4-edge colorable. Using the algorithm of Skulrattanakulchai [Sku02], a 4-edge coloring can be found in linear time. Hence, we can always find a bracket drawing for graphs of maximum degree 3. Let us summarize.

Theorem 7.9. *Let $G = (V, E)$ be a graph of n vertices with maximum degree 3, let S be a 1-spaced set of n grid points, and let $\mu\colon V \to S$ be a mapping. We can find a μ-respecting NPO_2 embedding of G on S in $O(n)$ time.*

Note that there are graphs of maximum degree 4 that admit neither a 4-edge coloring nor a bracket embedding, but still do admit an NPO_2 embedding, at least for some 1-spaced point sets; see Figure 7.5 for such an embedding of K_5. By checking many instances with the help of a SAT solver, we did not find any example of a graph of maximum degree 4 and a 1-spaced point set such that there is no NPO_2 embedding.

Note that the NPO_2 embeddings on 1-spaced point sets that our algorithm finds will always use space outside of the bounding box of the input points. Furthermore, there are examples of a

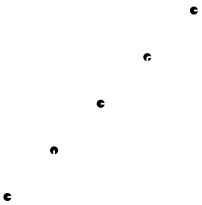


Figure 7.5: NPO_2 drawing of K_5 on a diagonal point set.

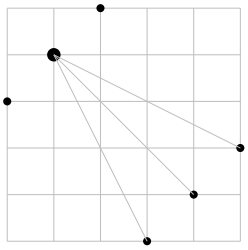


Figure 7.6: Graph and mapping with no NPO_2 embedding on the point set within the bounding box.

graph G, a 1-spaced set S of grid points, and a mapping μ such that G does not admit an NPO_2 point-set embedding on S with mapping μ if we insist that the drawing lies within the bounding box of S; see Figure 7.6. Hence, also here, area minimization is an interesting open problem. Since our hardness proof for area minimization of NPO_2 embeddings used only points of the x-axis, we do not know whether area minimization is still NP-hard for 1-spaced point sets.

Three Bends per Edge. If we allow four bends per edge, then it is clear that, for any graph $G = (V, E)$ of maximum degree 4, any 1-spaced set S of n grid points, and any mapping μ, we can find a μ-respecting NPO_2 embedding of G on S: Any two points can be connected using at most four bends even if we prescribe in which directions the edge leaves the points. If we want to draw an edge with more than one bend, we can use the two segments incident to the vertices to leave the bounding box of the input points since the point set is 1-spaced. The remaining segments can be placed sufficiently far away from the input points to avoid overlaps.

We will now see that just three bends already suffice. To this end, we first partition the edges of the graph into two sets A and $B = E \setminus A$ so that we get two graphs $G_1 = (V, A)$ and $G_2 = (V, B)$ of maximum degree 2; this is always possible as we will see.

First, we can assume that all vertices of G have even degree, that is, degree 2 or degree 4: If this is not already the case, we add edges to G connecting vertices of odd degree until all vertices have even degree; note that there is always an even number of vertices of odd degree. Once there are only vertices of even degree, we can find a Eulerian cycle C in G. We now follow C and alternately assign the edges to A and B. When we traverse a vertex v when following C, one edge incident to v will be assigned to A and another one will be assigned to B. As v can be traversed by C at most twice, its degree in G_1 and in G_2 can be at most two.

Now, we associate two adjacent directions with each subgraph, G_1 and G_2; for example, we can associate "up" and "right" with G_1 and "down" and "left" with G_2. Using these directions for leaving the vertices, we can draw the subgraphs individually; see Figure 7.7. Any connected component of a subgraph is a path or a cycle. For drawing a path, or a cycle of even length, we can, hence, alternately use the two directions assigned to the subgraph for drawing an edge with two bends in the bracket style. If we want to draw a cycle of odd length, we also start the drawing

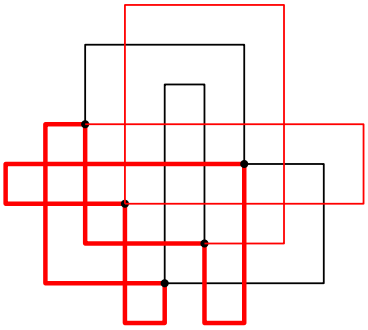


Figure 7.7: The graph K_5 drawn in the NPO_3 style; edges of G_1 and G_2 are drawn in black and in (bold) red, respectively.

like this up to the last edge; the final edge has to use two adjacent directions, for instance up and right. This is always possible with three bends. It is easy to see that, using the above steps, such an embedding can be constructed in linear time. We just have to keep track of the remaining vertices of odd degree when adding edges.

Proposition 7.1. *Let $G = (V, E)$ be a graph of n vertices with maximum degree 4, let S be a 1-spaced set of n grid points, and let $\mu\colon V \to S$ be a mapping. G admits a μ-respecting NPO_3 embedding on S and we can find such an embedding in $O(n)$ time.*

Note that this result is in contrast to general point sets that are not 1-spaced, where the problem is NP-hard; see Theorem 7.4.

7.4 Planar Orthogonal Point-Set Embeddability with Unbounded Bend Numbers

We now turn to the planar case with no bound on the number of bends, that is, to PO_∞ PSE with prescribed mapping; this version allows the most freedom for routing the edges. Katz et al. [KKRW10] have shown that Manhattan-geodesic PO_∞ PSE with prescribed mapping is NP-hard; in this setting, edges must be drawn as shortest orthogonal connections. We show that general PO_∞ PSE with prescribed mapping is also NP-hard by modifying the hardness proof of Katz et al.

Theorem 7.10. *PO_∞ PSE with prescribed mapping is NP-hard even for matchings.*

Proof. We sketch the proof of Katz et al. and show that, with our modifications, we get a feasible reduction for general PO_∞ . The reduction is from 3Partition which is known to be strongly NP-hard; compare Section 2.3. Let $A = \{a_1, \ldots, a_{3n}\}$ be an instance of 3Partition which consists of $3n$ positive integers. The goal is to decide whether there exists a partition of A into n sets $A_1, \ldots, A_n$ of 3 numbers such that all A_i have the same sum $s = 1/n \cdot \sum_{i=1}^{3n} a_i$. Figure 7.8 shows an example for the reduction; our modifications are marked by dashed lines.

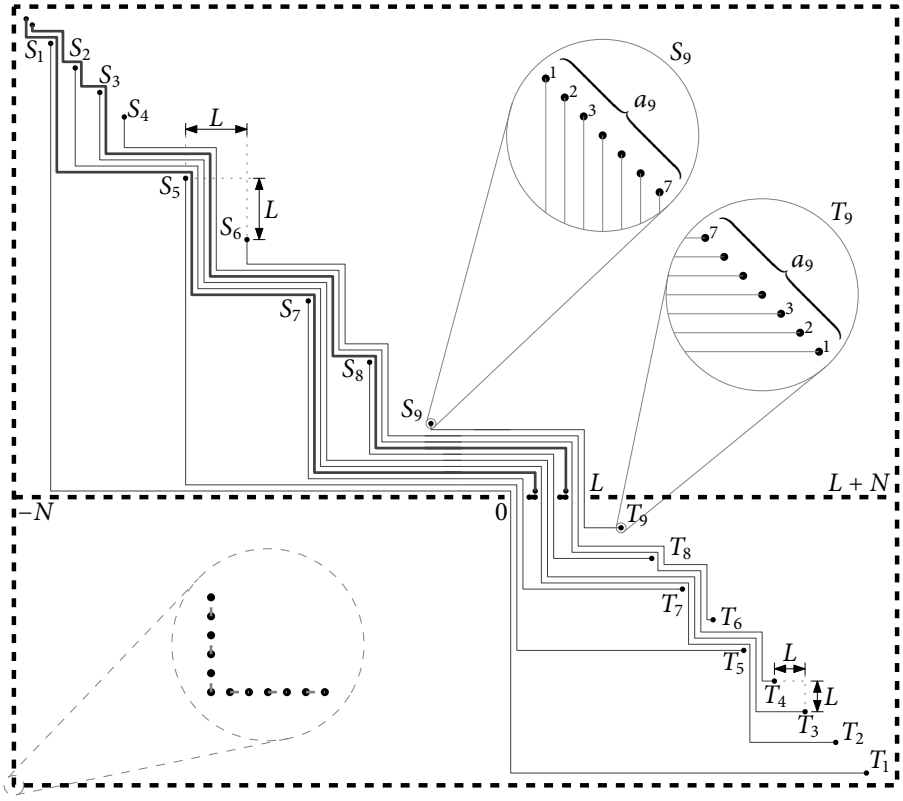


Figure 7.8: Example showing an instance of our reduction from 3Partition to PO_∞ PSE with prescribed mapping with sets $A_1 = \{a_1, a_5, a_7\}$, $A_2 = \{a_2, a_3, a_8\}$, and $A_3 = \{a_4, a_6, a_9\}$. Differences to the construction of Katz et al. [KKRW10] are dashed.

There are two modifications. First, we replaced two single edges between $-N$ and 0 and between L and $L + N$ by consecutive matching edges whose vertices fill all the grid points that previously were spanned by the two edges. Second, we added a frame around the whole construction; the frame is built the same way as the previous modification. It is easy to see that the modified instance still works as desired by Katz et al. The edges between $-N$ and 0 and between L and $L + N$ can only be drawn horizontally in the Manhattan-geodesic style. Furthermore, the geodesic style prevents that any other edge could interfere with the frame around the instance. Hence, if there is a feasible solution to 3Partition, we can find a (Manhattan-geodesic) PO_∞ embedding as shown by Katz et al.

Now, suppose that we have a (not necessarily Manhattan-geodesic) PO_∞ embedding. Even without the Manhattan-geodesic style the dashed frame cannot be crossed because there are no gaps between consecutive vertices. For any number a_i there are two groups S_i and T_i of a_i points placed consecutively in the upper and lower half of the instance, which are connected by a_i matching edges; see Figure 7.8.

The upper and lower half are connected by exactly n gaps of width s each, that is, at most $sn = \sum_{i=1}^{3n} a_i$ edges can leave the upper half. As all edges representing numbers are drawn, the gaps are completely used by them. Therefore, the $n - 1$ black separator edges must completely lie in the upper half and separate it into n regions. Furthermore, any set S_i of points must lie completely in one region; the points of S_i cannot be separated by an edge as they are diagonally contiguous. Hence, the regions define a partition of A into sets $A_1, \ldots, A_n$. As the edges leaving the region of A_i completely fill a gap of width s, it follows that $\sum_{a_j \in A_i} a_j = s$, which completes the proof. □

Note that the ability to use arbitrarily many bends for edges is crucial for the reduction: For creating a Manhattan-geodesic embedding—in the case that a feasible solution for the 3Partition instance exists—a large number of bends (linear in n) is used for routing from the upper to the lower half so that crossings and overlaps with other edges are avoided. Therefore, we cannot conclude anything about the complexity of PO_k PSE for a constant k.

7.5 Orthogonal Point-Set Embeddability without Prescribed Mapping

We now move to the problem variant in which the mapping from vertices to points is not part of the input; that is, we can choose the mapping freely. Recall that there is always an NPO_3 embedding (even with prescribed mapping) if the point set is 1-spaced. For general point sets, however, we will show that all problem variants are NP-hard. We first focus on the versions with up to one bend per edge, for which we can show hardness even for outerplanar graphs of maximum degree 3.

Theorem 7.11. *NPO_0 PSE, NPO_1 PSE, PO_0 PSE, and PO_1 PSE are NP-hard even for outerplanar graphs of maximum degree 3.*

Proof. We proof hardness by reduction from 3Partition and start with NPO_1 PSE. Let $A = \{a_1, \ldots, a_{3n}\}$ be an instance of 3Partition and let $s = 1/n \cdot \sum_{i=1}^{3n} a_i$. We model this instance as an instance of NPO_1 PSE.

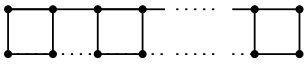


Figure 7.9: Graph G_i representing the number a_i consists of $4a_i$ vertices in $2a_i - 1$ connected 4-cycles.

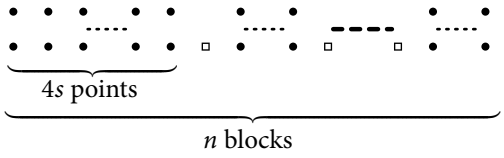


Figure 7.10: Point set S for the reduction from 3PARTITION.

Each number a_i is modeled by a graph $G_i = (V_i, E_i)$ consisting of $4a_i$ vertices in $2a_i - 1$ linked 4-cycles; see Figure 7.9. Note that the unary encoding of the numbers is possible due to the strong NP-hardness of 3PARTITION. Let $G = (V, E)$ be the graph consisting of the union of all G_is and a set V' of $n - 1$ isolated vertices, that is, $V = V' \cup \bigcup_{i=1}^{3n} V_i$ and $E = \bigcup_{i=1}^{3n} E_i$.

Let S be the set of grid points shown in Figure 7.10, that is, S consists of $n - 1 + 4ns$ points on two consecutive horizontal grid lines; the points are arranged in n blocks $S_1, \ldots, S_n$ of $4s$ points where within a block there is no gap, that is, no empty grid point; the blocks are split by isolated points.

Suppose that A is a positive instance, that is, there is a feasible partition into $A_1, \ldots, A_n$. We want to create an NPO_1 embedding of G on S. To this end, we iterate over the numbers a_i with increasing index i. Suppose that $a_i \in A_j$ for some $1 \leq j \leq n$. We want to place the drawing of G_i inside block S_j occupying columns from left to right. We embed G_i as indicated in Figure 7.9, using the $2a_i$ leftmost unoccupied columns of S_j. As $\sum_{a_i \in A_j} a_i = s$ this is possible for all a_i without any overlap within a block. Hence, we get a feasible NPO_1 embedding of G on S (in fact, even a PO_0 embedding).

We now want to see that, if there is no feasible solution of the 3PARTITION instance A, then no feasible embedding of G on S exists. To this end, we show that G_i can only be embedded in such a way that it lies completely inside a single block S_j. Suppose that an embedding of G_i would place a vertex v on an isolated point p. Then it is easy to see that a 4-cycle in which v is contained cannot be completed. On the other hand, it is also not possible to embed a 4-cycle using points of two different blocks. Hence, each G_i has to lie within a unique block S_j and a feasible embedding would yield a solution to the modeled instance of 3SAT. This completes the proof for NPO_1 PSE.

For creating a feasible drawing of a positive instance, we used only straight-line edges without crossings. The reduction, therefore, also works if we restrict the drawing style to straight-line edges and/or planar embeddings. □

Recall that PO_0 PSE is actually NP-hard even for matchings as Rendl and Woeginger [RW93] have shown.

Next, we move to the versions in which more bends are allowed. The basic structure of our reduction remains the same; some additional effort is necessary for making the idea work for larger numbers of bends. Furthermore, we now need general graphs of maximum degree 4.

Theorem 7.12. *NPO_∞ PSE and PO_∞ PSE, as well as NPO_k PSE and PO_k PSE for any constant $k \geq 0$, are NP-hard.*

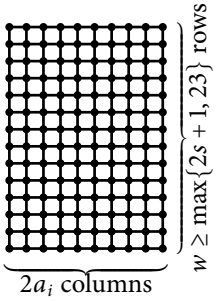


Figure 7.11: Graph G_i representing the number a_i is a grid graph of $2a_i \times w$ vertices.

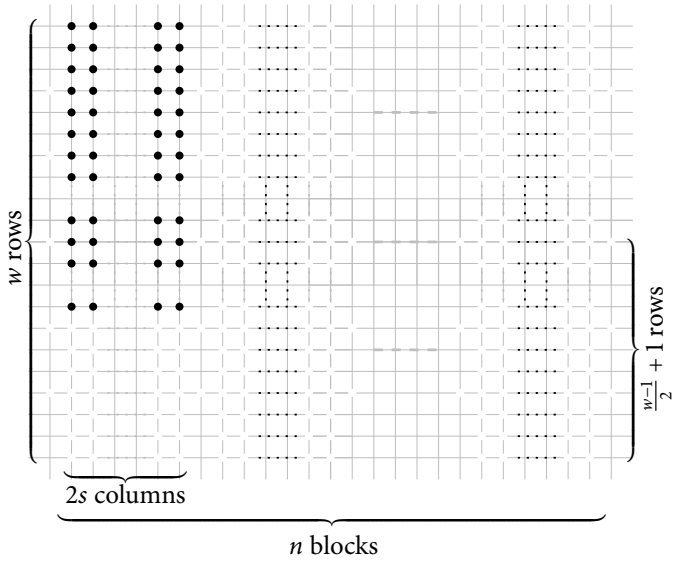


Figure 7.12: Point set for the reduction.

Proof. Again, we prove hardness by reduction from 3Partition. We start with NPO_∞ PSE. The remaining cases will follow as we use neither bends nor crossings in drawings for positive instances of 3Partition. Let $A = \{a_1, \ldots, a_{3n}\}$ be an instance of 3Partition and let $s = 1/n \cdot \sum_{i=1}^{3n} a_i$.

Each number a_i is modeled by a grid graph $G_i = (V_i, E_i)$ consisting of $2a_i \times w$ vertices; see Figure 7.11. Here, $w > 2s$ is an odd number that is as small as possible but at least 23. Again, the unary encoding of the numbers is possible due to the strong NP-completeness of 3Partition. Let $G = (V, E)$ be the graph consisting of the union of all G_is and a set V' of $5(n+1)$ isolated vertices, that is, $V = V' \cup \bigcup_{i=1}^{3n} V_i$ and $E = \bigcup_{i=1}^{3n} E_i$.

Let S be the set of grid points shown in Figure 7.12, that is, S consists of $2wsn + 5(n+1)$ points; the points are arranged in n blocks $S_1, \ldots, S_n$ of $2ws$ points each. The blocks are split by groups of 5 isolated points; similarly, 5 isolated points are placed to the left of S_1 and to the right of S_n, respectively.

Suppose that A is a positive instance, that is, there is a feasible partition into $A_1, \ldots, A_n$. We want to create a feasible NPO_∞ embedding of G on S. To this end, we iterate over the numbers a_i with increasing index i. Suppose that $a_i \in A_j$ for some set A_j. We want to place the drawing of G_i inside block S_j occupying columns from left to right. We embed G_i by drawing it as indicated in Figure 7.11, using the $2a_i$ leftmost unoccupied columns of S_j. As $\sum_{a_i \in S_j} a_i = s$ this is possible for all a_i without any overlap. Hence, we get a feasible NPO_∞ embedding of G on S (in fact, even a planar orthogonal drawing without bends).

We now want to see that, if there is an NPO_∞ embedding of G on S, then there is a feasible solution $A_1, \ldots, A_n$ of the 3Partition instance A. Suppose that an embedding of G_i would place a vertex $v \in V_i$ on an isolated point p. Then it is easy to see that a 4-cycle in which v is contained cannot be completed.

With two bends per edge it is possible to embed a single 4-cycle using points of two different blocks. Due to the grid structure of both the graphs and the blocks, this is, however, still no problem: Almost all 4-cycles will be embedded in the interior of blocks; there, the only possibility for embedding the 4-cycles is following the grid structure. Hence, for example on the middle horizontal line, a graph G_i can be embedded either horizontally (as indicated in Figure 7.11) or vertically.

We show that G_i (or, more precisely, any part of G_i) cannot be embedded vertically on the middle horizontal line. Suppose that a part of G_i is embedded in such a way. Then, we follow the horizontal line to the left and to the right. This means, that we follow G_i to the top and to the bottom, as G_i is embedded vertically. In both directions, G_i cannot leave the block due to the isolated point placed there. However, the block has only width $2s$, while G_i has height $w > 2s$, a contradiction. Hence, the embedding of G_i at the middle horizontal line has to be horizontal. If we follow the embedding of G_i upward and downward, we will reach the upper or lower part of the block of points. Hence, more than half of the height of G_i is contained in the block, which can be the case for only one block. Hence, on the middle line, G_i occurs only in one block S_j. On the other hand, due to the horizontal embedding of G_i, there also has to be at least one block S_j that contains at least half of the height of G_i. In this block S_j, G_i has to occur on the middle horizontal line.

Therefore, the occurrences of the graphs G_i on the middle horizontal line in blocks $S_1, \ldots, S_n$ give rise to a partition of A into sets $A_1, \ldots, A_n$. As the width of each block is $2s$, this partition is a feasible solution for the instance A of 3Partition. This completes the proof.

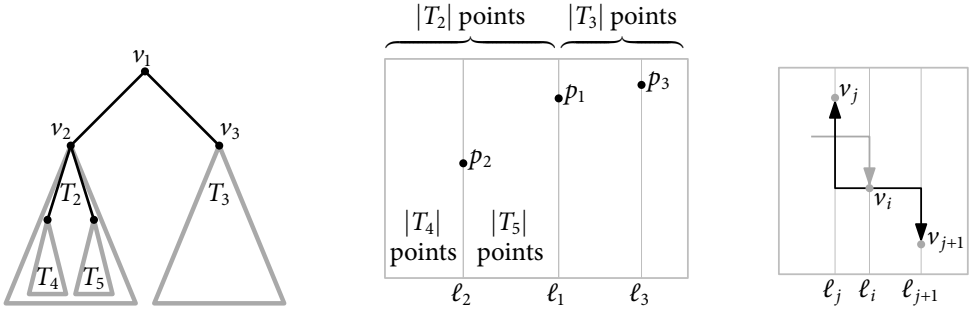
Recall that, as promised in the beginning, we did not use bends or crossings for creating embeddings of positive instances. Hence, the reduction works also for the planar version and for any limited number of bends. □

7.6 NPO_1 Embeddings on 1-Spaced Point Sets without Prescribed Mapping

In Theorem 7.2, we have seen that NPO_1 PSE with prescribed mapping can be solved in quadratic time. In contrast, the problem is NP-hard if the mapping is not part of the input (see Theorem 7.11). If we restrict ourselves to 1-spaced point sets, this does not have to hold; recall that we used many points with identical x- or y-coordinate in the hardness proof. Still, although we do not have a hardness proof, even the restriction to 1-spaced point sets seem quite hard. In this section, we will see that at least for some small classes of graphs there is an embedding for any 1-spaced point set.

First, we can find an NPO_1 embedding for every n-vertex path or cycle on any 1-spaced set of n grid points, even with prescribed mapping: we simply leave each point horizontally and enter the next one vertically in the order prescribed by the mapping.

Without a given mapping, we can see that every binary tree has an NPO_1 embedding on every 1-spaced set of n grid points. The idea is to map the root of the tree to the point that has as many



(a) Input tree. (b) Partition of the point set. (c) Drawing of the edges.

Figure 7.13: Illustrations for embedding binary trees.

points to its left as the number of nodes in the left subtree of the root; see Figure 7.13b. When applying this idea recursively and drawing the first segment of the outgoing edges horizontally (one to the left, one to the right) and the second segment vertically, no two edges overlap; see Figure 7.13c. This was independently observed by Di Giacomo et al. [GFF+13]. We will extend the result to a slightly larger class of graphs; first, we will, however, show the result for binary trees in detail.

Theorem 7.13. *Every binary tree $T = (V, E)$ has an NPO_1 embedding on every 1-spaced set S of n grid points.*

Proof. Assume that T is rooted at an arbitrary vertex r of degree 1 or 2, and let $v_1, \dots, v_n$ be a numbering of the vertices of T given by a breadth-first-search traversal starting from r, that is, $v_1 = r$. For $i = 1, \dots, n$, let T_i be the subtree of T rooted at vertex v_i; see Figure 7.13a.

Let p_1 be the point in S such that the vertical line ℓ_1 through p_1 splits $S_1 = S$ according to $T_1 = T$, that is, we split S_1 into a set S_2 of $|T_2|$ points on its left and a set S_3 of $|T_3|$ points on its right; see Figure 7.13b. Then, we recursively pick points p_2 and p_3 and lines ℓ_2 and ℓ_3 that partition S_2 and S_3 according to T_2 and T_3. We continue until we arrive at the leaves of T. This process determines points $p_1, \dots, p_n$ and lines $\ell_1, \dots, \ell_n$ such that for $i = 1, \dots, n$ point p_i lies on ℓ_i. We simply map vertex v_i to point p_i for $i = 1, \dots, n$.

Consider an index $i \in \{1, \dots, n\}$. Our mapping makes sure that one subtree of T_i is drawn to the left of ℓ_i and the other is drawn to the right of ℓ_i. Let v_j and v_{j+1} be the children of v_i. We draw the edges (v_i, v_j) and (v_i, v_{j+1}) such that their horizontal segments are both incident to v_i; see Figure 7.13c.

Since S is 1-spaced, no two edges can overlap except if they are incident to the same vertex. If we direct the edges of T away from the root, then, by our drawing rule, in any vertex v_i of T the incoming edge arrives in p_i with a vertical segment and the outgoing edges leave p_i with horizontal segments in opposite directions. Hence, the drawing is a feasible NPO_1 embedding. □

An interesting question is whether crossings are actually necessary for binary trees. More specifically, is it possible to find a PO_1 embedding for any binary tree on any 1-spaced point set?

(a) Embedding computed by our algorithm; it cannot be made planar without changing the mapping.

(b) Planar embedding obtained by using a different mapping.

Figure 7.14: Two embeddings of a binary tree on the same 1-spaced point set.

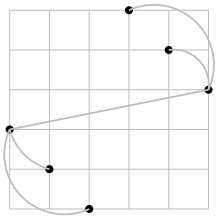


Figure 7.15: Binary tree with no NPO_1 embedding.

Kano and Suzuki proved this for the special class of binary trees in which a simple path containing all degree-3 vertices exists [KS12]; the general question is open. There is no counterexample, but there are examples for which the output drawing of our algorithm cannot be made planar by just modifying the edges without changing the mapping; see Figure 7.14.

In the proof of the previous theorem we exploited the fact that we could choose the vertex–point mapping as needed. Figure 7.15 shows a 6-vertex binary tree that does *not* have an NPO_1 embedding on the given point set if the vertex–point mapping is prescribed as indicated in the drawing.

We will now generalize the result on binary trees to a slightly larger class of graphs. We allow also binary trees whose vertices may be replaced by cycles. We still allow only maximum degree 3. Hence, when replacing a vertex v by a cycle, we have to make sure that the edges connecting v to its parent vertex and to its children are all incident to different vertices of the cycle. The basic idea of the algorithm for finding an embedding stays the same. There will, however, be several cases for embedding a cycle, depending on the configuration of the points.

Theorem 7.14. *Let $G = (V, E)$ be an n-vertex graph of maximum degree 3 that arises when replacing some of the vertices of a binary tree by cycles and let S be a 1-spaced set of n grid points. Then, G admits an NPO_1 embedding on S.*

Proof. The basic idea for extending the construction for binary trees to the new class of graphs is to treat each cycle similar to a single vertex of a binary tree. We do this by reserving the adequate number of consecutive columns for the vertices of the cycle in the middle of the drawing area

for the current subtree when splitting into the drawing areas for the subtrees. The subtrees are connected to the cycle by leaving one point to the right and one point to the left, respectively. The most difficult part is to connect the reserved points to a cycle; we must be able to connect the point representing the vertex that is the connector to the parent vertex (or cycle, respectively), which was embedded before. To this end, we have to make sure that we can enter this point using a vertical segment such that the connections to the left and to the right are still possible.

Let C with $k := |C| \geq 3$ be the cycle representing the root of the current subtree. Let u and v be the vertices of C that connect C to its left and right child, respectively, and let z be the vertex of C that connects C to its parent r. Let $S' = \{p_1, \ldots, p_k\}$ be the set of points reserved for C in consecutive columns ordered from left to right. The edges connecting C to the left and right subtrees leave the points representing u and v to the left and right, respectively, while the edge connecting z to r enters z from above or below, depending on the y-coordinate of the point chosen to represent z. Let y_r be the y-coordinate of r. We analyze the different cases.

1. Vertex z has a neighbor $w \neq u, v$ in C and $k \geq 5$:

   Set $\mu(u) = p_1$ and $\mu(v) = p_k$. Either above or below the horizontal line $y = y_r$ we find two points $p, p' \in S' \setminus \{p_1, p_k\}$. Let p be the one closer to the line $y = y_r$. We set $\mu(z) = p, \mu(w) = p'$ and draw the edge wz such that p is entered vertically. Then we can complete the cycle such that each point is incident to a horizontal and a vertical segment; see Figure 7.16a. It is easy to see that the connections to r and to the children can now be drawn without overlap.

2. Vertex z has a neighbor $w \neq u, v$ in C and $k = 4$:

   Let $C = (u, w, z, v)$; the other case is symmetric. If p_2 and p_3 both lie either below or above $y = y_r$, we can proceed as shown in case 1. If p_2 lies above r and p_3 below we have two subcases depending on the position of p_4:

   a) p_4 lies above p_3: We can draw C as shown in Figure 7.16b.

   b) p_4 lies below p_3: Similarly to case 1, we can draw C such that each point is incident to both a vertical and a horizontal segment as shown in Figure 7.16c.

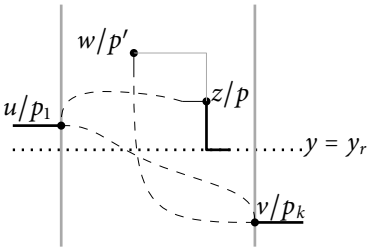
   If p_3 is above r and p_2 below, the cases are symmetric.

3. The two neighbors of z are u and v.

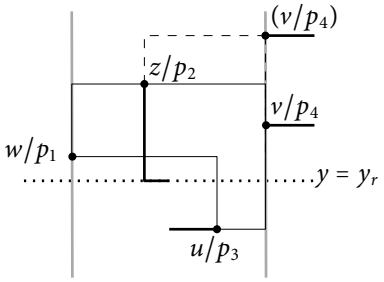
   a) If there is a point $p \in S' \setminus \{p_1, p_k\}$ that is vertically between p_1 and p_k, then we set $\mu(u) = p_1$, $\mu(v) = p_k$ and $\mu(z) = p$ and draw C as in Figure 7.16d, where the second path connecting u and v can be drawn by having a vertical and a horizontal segment incident to each point.

   In the remaining cases, there is no such point vertically between p_1 and p_k.

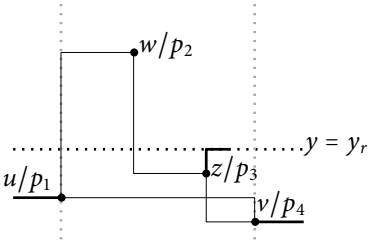
   b) If $k \geq 5$, we find, similar to case 1, two points $p, p' \in S' \setminus \{p_1, p_k\}$ both below or above r such that p is the one closer to the line $y = y_r$. Again we set $\mu(z) = p$; if p' is left of p we set $\mu(u) = p'$ and $\mu(v) = p_k$; see Figure 7.16e. Otherwise, we symmetrically set $\mu(v) = p'$ and $\mu(u) = p_1$. Now we can draw the cycle without overlap such that each point is incident to a vertical and a horizontal segment.



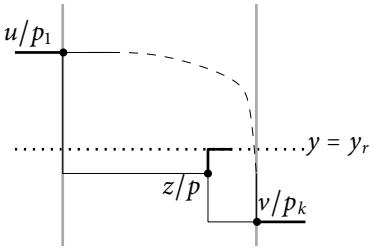
(a) Case 1



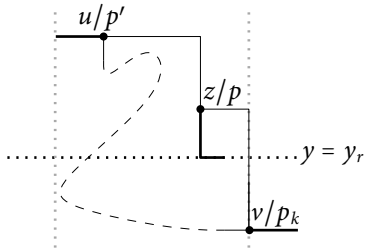
(b) Case 2a



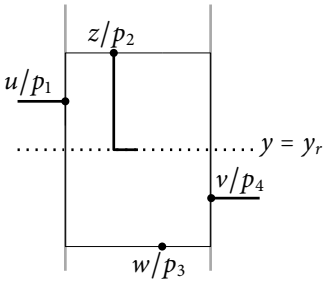
(c) Case 2b



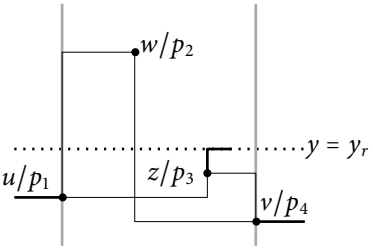
(d) Case 3a



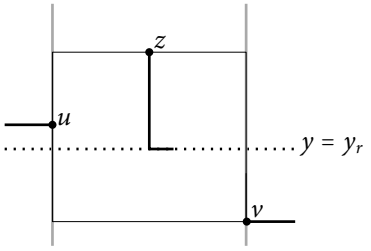
(e) Case 3b



(f) Case 3ci



(g) Case 3cii



(h) Case 3d

Figure 7.16: Embedding of cycle C in the different cases.

c) If $k = 4$, we have $C = (u, z, v, w)$. If p_2 and p_3 lie both above or below r we can proceed as in the previous case. Otherwise, we know that both points are on different sides of $y = y_r$, and that p_1 and p_4 are both vertically between, below, or above p_2 and p_3 (otherwise, we would be in case 3a).
   i) In the first case, we set $\mu(u) = p_1, \mu(v) = p_4, \mu(z) = p_2$ and $\mu(w) = p_3$ and create the drawing of C as shown in Figure 7.16f.
   ii) As above and below are symmetric, the other two cases can be handled as shown in Figure 7.16g.

d) Finally, if $k = 3$, we set $\mu(u) = p_1, \mu(v) = p_3$ and $\mu(z) = p_2$, and simply draw C as shown in Figure 7.16h. This completes the case analysis and, hence, the proof. □

It would, of course, be nice to generalize these embeddability results for binary trees and cycles (without given mapping) to larger classes of graphs, for example, outerplanar graphs of maximum degree 3. This seems, however, quite difficult.

7.7 Concluding Remarks

We have investigated both planar and nonplanar orthogonal point-set embeddability on the grid. For the version without prescribed mapping we have proven NP-hardness of all problem variants.

In the setting with prescribed mapping we have given efficient algorithms for deciding PO_1 PSE and NPO_1 PSE. We have further shown hardness for NPO_2 PSE and NPO_3 PSE with prescribed mapping, and for their area-minimization variants. In contrast, we have seen that any graph of maximum degree 4 can be embedded on any 1-spaced point set in the NPO_3 style. For NPO_2, this is the case at least for any graph of maximum degree 3. We have also shown that PO_∞ PSE, that is, planar orthogonal point-set embeddability, with prescribed mapping is NP-hard even for matchings.

Open Problems. There are still several open problems; here, we give a list of the most interesting ones.

- We know the complexity of NPO_k PSE with prescribed mapping for $k \leq 3$: For $k = 0$ and $k = 1$ we have given efficient solutions; for $k = 2$ and $k = 3$ we have shown hardness. What happens if we allow more than three bends per edge? That is, what is the complexity of NPO_k PSE and NPO_∞ PSE with prescribed mapping for $k \geq 4$?
- We have seen that PO_∞ PSE with prescribed mapping is NP-hard. Is planar orthogonal PSE hard even if we bound the number of bends, that is, is there some constant k such that PO_k PSE with prescribed mapping is NP-hard?
- In the hardness proof for the problem versions without prescribed mapping we made heavy use of vertices of degree 4. Do the problem variants stay NP-hard even for graphs with smaller maximum degree? Recall that Rendl and Woeginger [RW93] showed that PO_0 PSE is NP-hard even for matchings, that is, graphs of maximum degree 1. Are there similar properties for other variants?

- Does every n-node binary tree have a PO_1 embedding, that is, a planar embedding with one bend per edge, on any 1-spaced set of grid points? This was also posed as an open question by Kano and Suzuki [KS12]. We have seen that it is true for the nonplanar version.
- Does every n-node ternary tree have an NPO_1 embedding on any 1-spaced set of grid points? What about outerplanar graphs?
- Can we efficiently decide whether a given graph has an NPO_1 embedding on a given 1-spaced set of grid points (without mapping)?
- Does any graph of maximum degree 4 allow an NPO_2 embedding on any 1-spaced set of grid points? If this is the case, does it even hold for any prescribed mapping? If we cannot always find an NPO_2 embedding, is NPO_2 PSE NP-hard even for 1-spaced point sets?

Part III

Boundary Labeling

Chapter 8

Algorithms for Labeling Focus Regions

When exploring maps or diagrams, there often is a *focus region* in which the user is currently interested. Especially, there can be point sites within the focus region that must be labeled. This problem occurs, for example, when the user of a mapping service wants to see the names of restaurants or other POIs in a crowded downtown area but keep the overview over a larger area.

Our approach for this problem is to place the labels at the boundary of the focus region and connect each site with its label by an edge, which is also called a *leader*. In this way, we move labels from the focus region to the less valuable context region surrounding it. In order to make the leader layout well readable, we present algorithms that rule out crossings between leaders and optimize other characteristics such as total leader length and distance between labels. This yields a new variant of the *boundary labeling* problem. Other than in traditional boundary labeling, where leaders are usually schematized orthogonal or octilinear polylines, we focus on leaders that are either straight-line segments or Bézier curves.

We also present algorithms that, given the sites, find a position of the focus region that optimizes the above characteristics. Moreover, we consider a variant of the problem where we have more sites than space for labels. In this situation, we assume that the sites are prioritized by the user. Alternatively, we take a new facility-location perspective which yields a clustering of the sites. We label one representative of each cluster. If the user wishes, we apply our approach to the sites within a cluster, giving details on demand.

8.1 Introduction

Users of maps normally expect answers to specific queries, for example, where to find a good restaurant or how to reach a certain destination. General-purpose topographic maps do not answer such queries satisfactorily, and, thus, have become almost obsolete. Instead, Internet mapping services such as Google Maps or Bing Maps offer interfaces that allow for user interactions and sophisticated map visualization. Still, the existing systems do not fully support focus-and-context visualization, which generally aims at emphasizing regions and themes of interest while showing overview information for orientation.

In order to emphasize a focus region (for example, the user's vicinity), many researchers have proposed to (locally) increase the map scale in that region, which allows more details to be presented. Different methods have been proposed to define a seamless transition between a large-scale focus region and a small-scale context region, including fish-eye projections [YOT09] and, for network maps, optimization-based graph drawing methods [HS11]. Another common approach is to use *portals* [OW00], that is, windows with detailed information superimposed

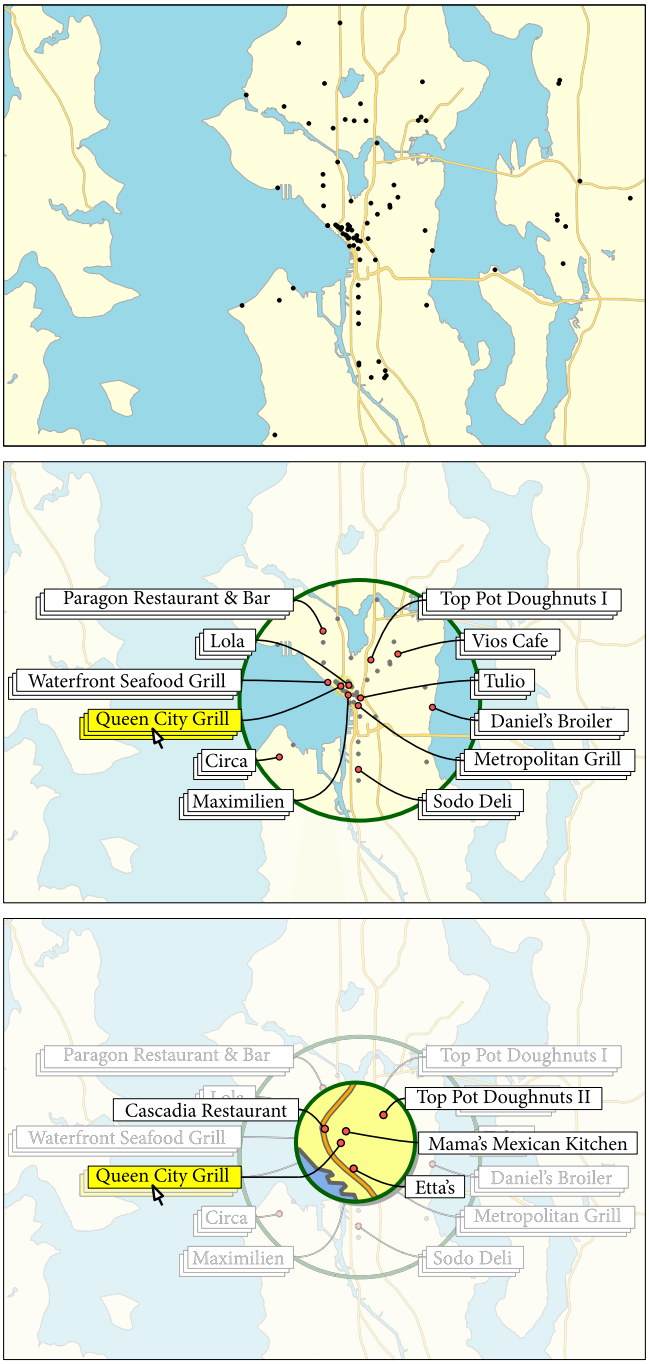
on an overview map. Additionally, emphasis can be put on map objects by appropriately setting their colors [ZR02].

Considering that the focus and the context region of a map serve different purposes, we argue that special labeling techniques are needed for focus-and-context maps. Map space can be regarded as a resource that is more expensive in the focus region; thus, text annotations and iconic labels for points of interest (POIs) or *sites* in the focus region should preferably be moved to the context region. The difficulty herein is that correspondences between labels and sites have to remain clear. One possibility to achieve this is to display such correspondences as linear connections called *visual links* [SWS+11] or *leaders* [BKSW07]. We use the latter term, which is more common in the literature on map labeling.

In this chapter, we apply *boundary labeling* to focus-and-context maps. Boundary labeling commonly means to place labels at the boundary of a *map* and, for each label, to draw a leader that connects a point—called *port*—on the label boundary with the corresponding site in the map. In a focus-and-context map, we suggest placing labels at the boundary of the *focus region*, which may be given explicitly as part of a user's query. This is the case in the scenario shown in Figure 8.1, where a user specifies a circular region in order to query the restaurants in that region. If the user does *not* specify a focus region, however, a visualization system should still be able to produce a good focus-and-context map. As a general rule, focus should be put on regions with many interesting sites. Taking this rule into account, we develop also algorithms that determine a circular focus region of given radius such that the maximum number of sites is labeled given our boundary-labeling model.

The general problem with boundary labeling is that the leaders produce additional clutter and that the correspondence between labels and sites may become unclear, especially if leaders are zig-zagging, crossing each other, long, or close to each other. Therefore, we designed our boundary-labeling algorithms to avoid such unfavorable leader properties. For example, we consider a variant of the labeling problem where the leaders are straight-line segments, crossings are forbidden, and the total length of all leaders is minimized. We also present algorithms that visually improve solutions to this problem variant by transforming the straight-line segments into Bézier curves that have at most one inflection point. This allows us to control the slope of a leader in its site and port, for example, to ensure that a leader connects horizontally or vertically to its port and thus has the same slope as the boundary of its adjoining label. According to the Gestalt criterion of good continuation [Wer38] this is favorable, as it allows map users to understand the label-site correspondences more easily.

Maps often contain too many points of interest to label them all. Therefore, in addition to choosing label positions and drawing leaders, we have to select a subset of the sites that will become labeled. An approach that is common in the literature on map labeling is to search for a maximum-cardinality set of sites that allows for a feasible labeling. If we have space for k labels at the boundary of the focus region, however, we can select *any* subset of k sites to become labeled. Among these solutions we may search for a labeling with short leaders, but this means that mainly sites at the boundary of the focus region will become selected. This is unfavorable, since normally users are particularly interested in the center of the focus region. Moreover, we argue that the selected subset should reflect the spatial distribution of all sites. If the input set contains a dense cluster (for example, a city center with many restaurants), the selected subset should also contain a cluster (which may be less dense) in the same area. In order to take this



(a) A map of Seattle with 86 restaurants (black dots).

(b) A user selects a focus region. At the boundary of that region (green circle), labels are placed for a selection of the restaurants; curved lines connect the labels with the actual sites of the restaurants. Every labeled restaurant represents a cluster of restaurants, which we indicate by drawing a stack of rectangles with the label on top.

(c) When clicking a label, a detailed labeling for the corresponding cluster pops up.

Figure 8.1: Interactive use case of labeling focus regions for exploring restaurants in Seattle.

additional criterion into account, we suggest a novel approach by modeling map labeling as a facility location problem.

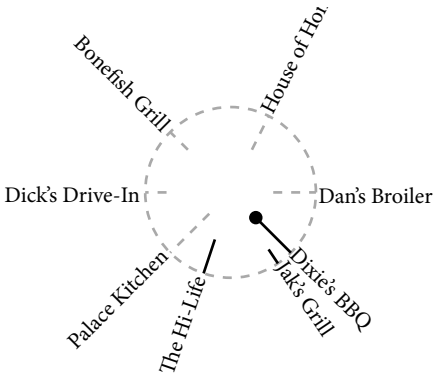
Our Contribution. Our contribution is as follows.

- We discuss two leader models; the *radial-leader model* (see Figure 8.2a) and the *free-leader model* (see Figure 8.2b) [1] and formally state the corresponding layout problems (Section 8.2). While the problems concerning the free-leader model can be reduced to specific matching problems on graphs for which efficient algorithms are known, we present new, efficient algorithmic solutions to our problems in the radial-leader model (Section 8.3). Other than Plaisant and Fekete [FP99] who have used radial leaders before (see Figure 8.3 and the discussion of previous work), our leaders do not bend since we place the labels directly at the boundary of the focus region. In order to strengthen the visual connection between labels and objects, we deliberately decided not to place our labels as blocks of left-aligned text, but rather in the immediate vicinity of the focus region. While the approach of Plaisant and Fekete is meant for interactive exploration, ours also makes sense in a static environment where the aesthetic quality of the leader and label placement is crucial. For example, we optimize the layout in case there are more sites than space for labels. We are particularly interested in the algorithmic challenges behind these optimization problems.
- Among the algorithms for the radial-leader model, we also address a new optimization goal for circular focus regions: given the region's radius, we find a position of the region that maximizes the number of sites whose labels can be placed—without overlap—at the region's boundary; see Section 8.3.3.
- We present two extensions (Section 8.4) that can be applied to our models, namely a facility-location model that allows us to simultaneously cluster and label a set of sites and a postprocessing for (non-crossing) straight-line leaders that transforms them into (non-crossing) Bézier curves; see Figure 8.2 (bottom). To the best of our knowledge, neither clustering nor Bézier curves have been used for (boundary) labeling, so far.
- We use the clustering from extension (i) to partition the focus region into subregions, one for each cluster. If the user clicks on the label that corresponds to the cluster (or into the subregion), we display the subregion enlarged and apply our labeling method to the sites in the subregion; see Figure 8.1c and Section 8.4.1.

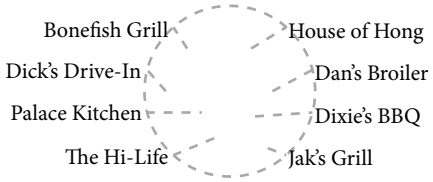
We present results of some experiments that we performed with implementations of our algorithms; the experiments and results are described in various sections of this chapter.

Previous Work. Labeling geographic maps is a central problem in cartography. Labeling maps manually is a tedious task that, in the 1980's, was estimated to consume 50% of a map's production time [Mor80]. Typically, a label should not occlude features of the image and it should not overlap with other labels. In map labeling, due to the small size of labels (usually a

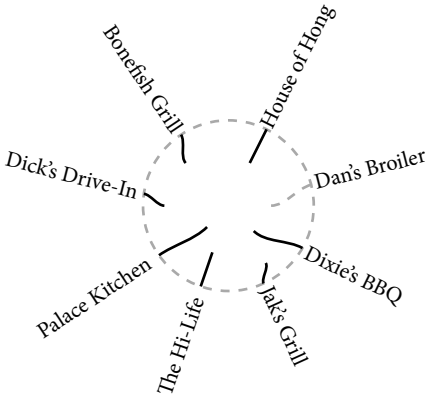
[1] A video demo showing both models is available under `http://www1.informatik.uni-wuerzburg.de/pub/videos/infovis2012.mp4` and `http://vimeo.com/user12598215/circlelab`.



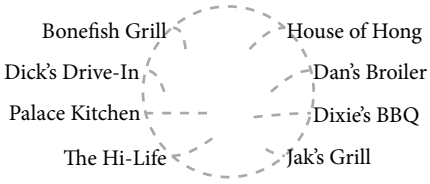
(a) Radial-leader model, straight-line leaders.



(b) Free-leader model, straight-line leaders.

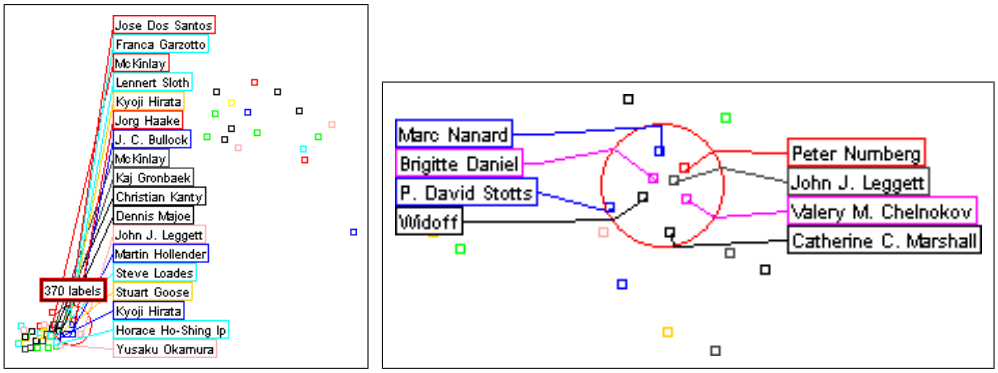


(c) Radial-leader model, Bézier leaders.



(d) Free-leader model, Bézier leaders.

Figure 8.2: Example labelings with our two leader models, top: drawn as straight-line segments, bottom: with Bézier postprocessing applied to the above straight-line solutions.



(a) ordered (b) radial

Figure 8.3: Examples of excentric labeling. Clipped figures from [FP98].

single word/name) and our ability to control the feature density, we usually manage to place the labels on the map so they are in the immediate vicinity of the feature they describe. Map labeling has been studied in computer science for more than two decades [FW91]. A survey on algorithmic map labeling and an extensive bibliography are given by Neyer [Ney01] and Wolff and Strijk [WS], respectively. However, *internal labeling* is not feasible when large labels are employed, a typical situation in technical drawings and medical atlases; this also happens when trying to label sites in a dense focus region.

Traditionally, labels are placed directly on the map; leaders are used very rarely. A simple model to establish a relationship between internal labels and sites is the four-position model [CMS95, WWKS01]. Here a label is represented by a rectangular box and the label is placed such that one of the four rectangle corners coincides with the associated site.

The idea to label data along a circular boundary has been applied before. The *excentric labeling* approach by Fekete and Plaisant [FP99] extends the infotip paradigm to label dense maps interactively. They draw a circular focus region of fixed radius around the current cursor position, and label the sites that fall into the circle. Labels are left-aligned in one or two stacks to the left and/or right of the circle, depending on where space is available. The labels are connected to the sites by leaders. For drawing the leaders, Fekete and Plaisant present two main approaches. In the first approach, they insist on ordering the labels within each stack according to the vertical order of the corresponding sites. As a result, leaders may cross; see Figure 8.3a. In the second approach, a leader goes from a site via its projection on the focus circle and then, in the case of a right stack, to the mid-point of a left label edge. In the case of a left stack, a third segment may be needed in order to reach the right label edge of a very short label without introducing crossings between the leader of that label and other labels [FP99]. The authors call this approach the *non-crossing* or *radial* approach. Obviously, the higher the label stacks are the smaller the angles between the leaders get. If more sites lie in the focus region than can be labeled, an arbitrary subset of representatives is chosen and labeled; additionally, the number of sites in the disk is displayed (see Figure 8.3a).

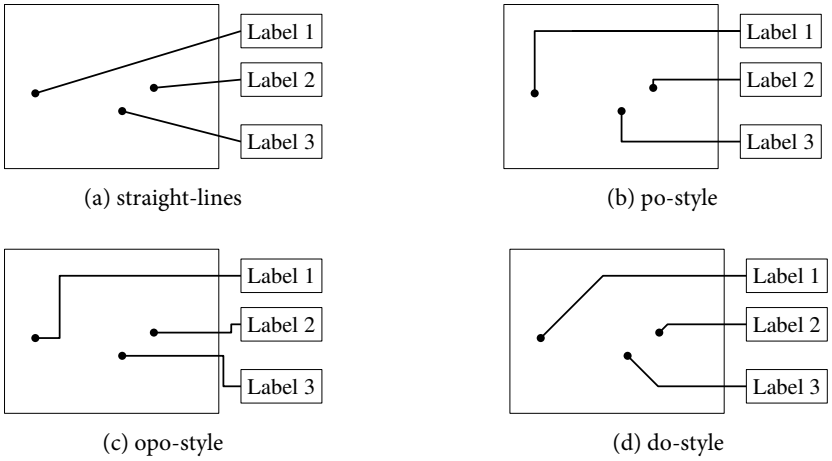


Figure 8.4: Different leader styles for boundary labeling.

Fekete and Plaisant recommend the first approach as a default; the focus of their work was on interactive speed and on a comparison to statically labeled maps where users had to zoom and pan in order to find specific sites. They conducted a user study (with eight subjects) that showed that users completed the task using interactive labeling nearly twice as fast as when using static labeling. Fekete and Plaisant did not specify the asymptotic running times of their approaches, but it is easy to see that they run in $O(n \log n)$ time, where n is the number of points inside the focus region.

Bertini et al. [BRL09] presented extensions for excentric labeling. They added scrolling through lists of labels if the stacks were to large. Furthermore, they developed an automatic adjustment of the size of the focus region based on the density of sites; they also implemented filtering, sorting of labels, and inheritance of visual features. Bertini et al. also conducted a user study whose main result was that users could intuitively use most of the implemented functions; only scrolling through lists of labels caused problems for some participants of the study.

Bekos et al. [BKSW07, BKSW05] introduced *boundary labeling* for labeling static maps. In this model, the sites are contained in a rectangular focus region and labels are placed outside the rectangle. The model supports three types of leaders: straight-line segments and orthogonal polylines with one or two bends per line, called the po- and the opo-style, respectively; see Figure 8.4. Labels are either placed along one, two or all four sides of the boundary rectangle. The authors show how to construct a non-crossing labeling with minimum total leader length or minimum number of bends for some variants of their model.

Later, other variants of boundary labeling have been investigated. Bekos et al. [BKNS10] introduced octilinear leaders, including, for instance, the do-style; see Figure 8.4d. In this style, any leader is drawn as an octilinear edge, consisting of a horizontal segment incident to the label and a diagonal segment incident to the site.

Benkert et al. [BHKN09] presented a polynomial-time algorithm for finding boundary labelings in the po- or do-style (see Figure 8.4b and Figure 8.4d, respectively) that allows to optimize quality measures for the leaders, for example, the total leader length or the number of bends.

Their approach is flexible enough to work with general quality measures for a single leader, but cannot take the interaction between different leaders into account.

Bekos et al. [BKPS11] studied techniques for combining boundary labels with interior labels placed on the map. In this setting, one has to make sure that labels do not intersect interior labels. As, in their model, sites could be labeled with either an interior or a boundary label, intersections could be avoided by moving intersected interior labels to the boundary.

Most of the previous work has been focused on placing the labels either on just one side of a rectangular focus regions, or on two opposite sides. In a recent work, Kindermann et al. [KNR+13] gave algorithms for po-leaders with labels placed on two adjacent sides of the focus region.

Gemsa et al. [GHN11] presented an algorithm targeted to label panorama images using vertical straight-line leaders. They place labels in several layers and present algorithms that minimize the number of layers needed for labeling all sites or that maximize the number of sites that can be labeled using a fixed number of layers.

A survey article by Kaufmann [Kau09] presents the different boundary labeling models that have been studied in the literature.

Hartmann et al. [HGAS05] introduced a more general model for boundary labeling. In particular, they suggested a classification scheme that takes also more complex boundary shapes of the map into account, that is, the boundary can be a circle or more generally an arbitrary silhouette. A first algorithm for the silhouette scenario (based on the force-directed approach) is due to Ali et al. [AHS05].

Recently, Speckmann and Verbeek [SV10] introduced *necklace maps* to visualize statistical data on geographic domains. The size of the label is used to encode the value of the statistical variable and the labels are placed on circles or silhouettes. Necklace maps do not use leaders, but establish relations between geographic objects and labels by matching colors and proximity. This idea is effective if the geographic objects to be labeled have some spacial extent; here, in contrast, we assume that we are given point data.

8.2 Problem Statements and Motivations

We study several incarnations of the problem of labeling focus regions. We distinguish two general models. In the first model, the focus region is a disk and each leader is a section of a ray that emanates from the center of the disk and connects a site to the boundary of the focus region; see Figure 8.2a. We refer to this model as the *radial-leader model* and discuss it in Section 8.2.1. Note that, if no two sites lie on the same ray, leaders are disjoint by construction. In the second model, the *free-leader model* (see Figure 8.2b and Section 8.2.2), every port is placed on a prescribed position on the boundary of the focus region. We do not insist on any specific direction of the leaders; instead, we explicitly require leaders to not cross each other. Note that, in the free-leader model, there is no need to restrict oneself to circular focus regions. In fact, all results in this model hold for convex focus regions.

The choice of the model does not necessarily determine the orientation of the text labels. We think, however, that the radial-leader model suits radially oriented labels particularly well. In contrast, we suggest using the free-leader model in conjunction with horizontally oriented labels. For examples, see Figure 8.2 (top).

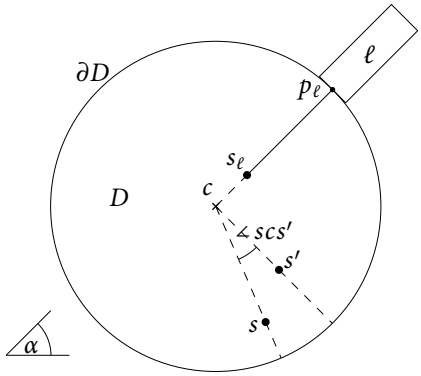


Figure 8.5: In a disk $D = (c, r)$, point s_ℓ is connected by a leader to the port p_ℓ of its label ℓ on the boundary ∂D. The points s and s' are in conflict since $\angle scs' < \alpha$.

8.2.1 Radial Leaders

For the radial-leader model, we assume that the focus region is a disk $D = (c, r)$ with center c and radius r. Given a label ℓ, we define the port p_ℓ of ℓ by radially projecting the site s_ℓ labeled by ℓ onto the boundary ∂D of D, that is, p_ℓ is the intersection of ∂D with the ray that emanates from c and goes through s_ℓ; see Figure 8.5.

Clearly, this model makes it difficult to accommodate labels if the projections of several sites lie in a very small part of ∂D. We model this by saying that two sites s and s' are *in conflict* if the angle $\angle scs'$ is smaller than a predefined value $\alpha > 0$; see Figure 8.5. Our aim is to find (and label) a maximum-cardinality subset of the sites that is *conflict-free*, in the sense that no two sites in the subset are in conflict.

This model makes sense, for example, if (as in Figure 8.2a) each label contains one line of text and has the same orientation as its leader. In this case each label ℓ is a unit-height rectangle (which may contain some text), the lower and upper edges of ℓ have the same slope as the leader of ℓ, and the port of ℓ is the midpoint of either the left or right edge of ℓ. Correspondences between labels and sites are particularly easy to comprehend in this case, since the leaders can be perceived as continuations of their labels. On the downside, the user has to read rotated text. According to a user study of Wigdor and Balakrishnan [WB05], however, rotating a text by not more than 90° leads to only a small decrease in the reading speed of users and only a small increase in the number of reading errors if the text consists of a single word with five to six letters. Therefore, we suggest using this model if the label texts are not much longer than this.

We now formally define the label maximization problem with given center.

Problem 8.1 (Label maximization with given center)**.**
Input: *Disk $D = (c, r)$, set $S \subset D$ of n point sites, angle α.*
Output: *Maximum-cardinality conflict-free subset $S' \subseteq S$, that is, the angle formed by any two rays that emanate from c and go through points in S' is at least α and $|S'|$ is maximized.*

In practice, the radius r of D and the fixed font size determine the angle α.

Next, we take priorities among the sites into account, which occur, for example, if some sites match a user query better than others. In the literature on map labeling, this is commonly modeled by defining weights for the sites and by selecting a maximum-weight set of sites that allows a feasible labeling. The weighted version of the problem is defined as follows.

Problem 8.2 (Weighted label maximization with given center)**.**
Input: *Disk $D = (c, r)$, set $S \subset D$ of n point sites, weight function $w\colon S \to \mathbb{R}^+$, angle α.*
Output: *Conflict-free subset $S' \subseteq S$ such that the total weight $w(S') = \sum_{s \in S'} w(s)$ is maximized.*

For the remaining two variants of the radial-leader model, we support the user by finding a good position of the focus region. In many applications the focus is specified by a fixed set of sites rather than by a fixed focus region. In order to make sure that as much useful information as possible is displayed, we place the focus region such that it admits a maximum-cardinality conflict-free subset of sites.

Problem 8.3 (Label maximization with variable center position)**.**
Input: *Set $S \subset \mathbb{R}^2$ of n point sites, angle α.*
Output: *Center $c \in \mathbb{R}^2 \setminus S$, maximum-cardinality subset $S' \subseteq S$ that is conflict-free with respect to c and α.*

For this problem, we also consider the weighted version, which we define analogously to Problem 8.2.

As the last problem of this section we ask for the disk center c, which maximizes the minimum angular separation of the sites with respect to c.

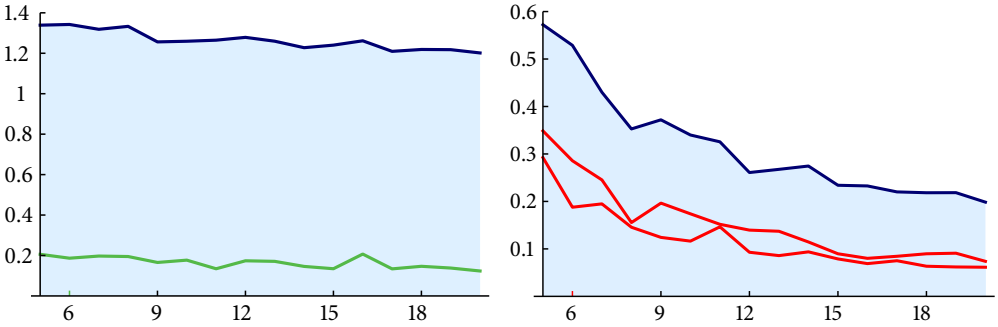
Problem 8.4 (Sector maximization)**.**
Input: *Set $S \subset \mathbb{R}^2$ of n point sites.*
Output: *Center $c \in \mathbb{R}^2 \setminus S$ and angle α such that S is conflict-free with respect to α and α is the largest angle with this property.*

Our main motivation for studying Problem 8.4 stems from the case where *all* sites need to be labeled. Note that such a labeling exists if we make the radius r of the focus disk large enough. Therefore, we can assume that a small separating angle α suffices for having disjoint labels. In practice, however, we cannot arbitrarily increase r since the available space is limited. Therefore, it is reasonable to ask for the smallest disk that allows us to label all sites. If α is the smallest angle formed by two rays emanating from c and going through two sites in S, then r has to be greater than $1/\alpha$—assuming that labels have height 1. Therefore, minimizing the radius of the disk is equivalent to maximizing the smallest angle between any two rays.

Problem 8.4 also occurs when we have a solution to Problem 8.3, but two labels may be unnecessarily close to each other, even though they do not intersect. In this case we could relocate the focus disk to obtain a more balanced spacing between the labels.

Note that maximizing the smallest angle may be in conflict with minimizing the size of the focus disk.

Experiment 8.1. For $n = 5, \ldots, 20$, we selected a sample of n random points under normal distribution and computed the radius of the disk maximizing the smallest angle between leaders and covering all sites—the *smallest-angle focus disk*—and the radius of the smallest enclosing



(a) Radius of the smallest-angle focus disk relative to the smallest enclosing disk as measured in Experiment 8.1. The (upper) blue line shows the average ratio; the (lower) green line depicts the standard deviation in terms of the point set size n.

(b) Average of the minimum angle relative to $360°/n$ of the smallest-angle focus disk (blue, upper), center of gravity disk (black, middle), and minimum enclosing disk (red, lower) in terms of n as recorded in Experiment 8.2.

Figure 8.6: Plots showing the results of Experiments 8.1 and 8.2.

disk. For each instance size, we repeated the experiment 50 times. The result was that the radius for the smallest-angle focus disk was on average only 30% larger than the radius of the smallest enclosing disk, with a standard deviation of about 20% (see Figure 8.6a).

Using the smallest enclosing disk as the focus disk can result in arbitrarily bad angles. To see this, note that we can add a site near the disk center that introduces a small angle between leaders, but will not affect the minimum enclosing disk. Another heuristic for locating the focus disk is the center of gravity. This is motivated by distributing the sites inside the focus disk evenly. We can, however, add two sites that preserve the center of gravity but introduce arbitrarily small angles.

Experiment 8.2. We repeated Experiment 8.1, but this time we computed the smallest angle between leaders of the smallest-angle focus disk, the smallest enclosing disk, and the center of gravity disk (see Figure 8.6b). We observed that, on average, the angle of the smallest-angle disk is twice as large as the angle of the two other disks. This ratio gets even larger for larger point sets.

We repeated Experiments 8.1 and 8.2 with uniformly distributed point sets and skewed point sets in which all points lie close to a straight-line segment (that is, we defined the x-coordinates to be uniformly distributed in a small interval and the y-coordinates to be normally distributed). The results (again 50 samples for $n = 5, \ldots, 20$) were very similar to what we recorded in Experiments 8.1 and 8.2.

Due to our theoretical considerations and the outcome of our experiments we are convinced that the computationally harder sector-maximization technique (see Section 8.3.4) is worth being applied when selecting the focus disk. Figure 8.7 shows a point set with focus disks obtained by the three strategies we just discussed.

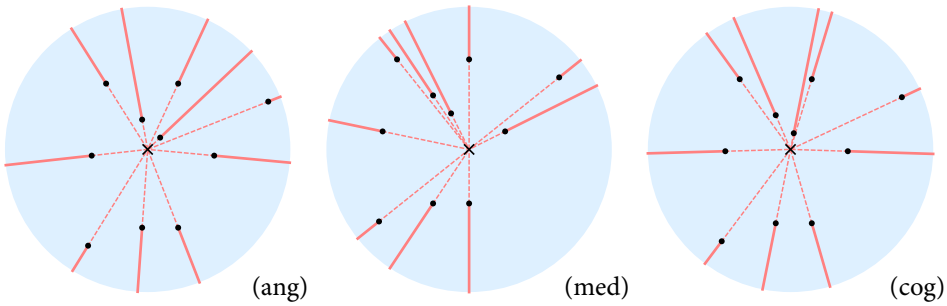


Figure 8.7: An example point set with disk that maximizes the minimum angle between leaders (ang). For comparison: the minimum enclosing disk (med) and the disk centered at the center of gravity (cog). All three disks are drawn with the same radius.

8.2.2 Free Leaders

In the free-leader model we do not require that the focus region is a disk, but we assume that it is a (connected) convex region $\mathcal{F}$. As in the radial-leader model, the focus region contains the sites that we want to label and the leaders are straight-line segments, each of which connects a site with a port on $\partial\mathcal{F}$. The locations of the ports on $\partial\mathcal{F}$ may be given as a set P. Alternatively, in case P is not given, we suggest computing the port locations as shown in Figure 8.8.

The method applied in Figure 8.8 requires a vertical distance $\Delta y \in \mathbb{R}^+$ between two consecutive ports on $\mathcal{F}$ as input, that is, the spacing between two lines of text. Any intersection between a line and the boundary $\partial\mathcal{F}$ of the focus region defines the location of a label port in the set P. Usually, we define the set of lines such that no line intersects the uppermost or lowermost point of $\mathcal{F}$—due to their prominent positions, labels placed at such points would be perceived to dominate other labels, which we normally want to avoid. In any case, setting Δy to a value greater than twice the height of a label and using the classical four-position point-labeling model [BKSW07] that allows any corner of a label to coincide with the label's port, we can always place the labels such that they intersect neither each other nor the interior of $\mathcal{F}$. As the focus region is convex and contains all sites in its interior, every straight-line segment that connects a port with a site lies completely in $\mathcal{F}$. Hence, intersections between leaders and labels (apart from label ports) are impossible. We have to ensure, however, that no two leaders cross.

We first consider the case that the number of sites and the number of locations for label ports are equal, that is, $|S| = |P|$. In this case, the core question is which sites are assigned to which ports. This can be formalized using graph-theoretic terms. We define the graph $G = (S \cup P, S \times P)$ that contains a vertex for each site, a vertex for each port location, and an edge for each pair of a site and a port location. Then a labeling is defined by a *perfect matching* in G, that is, a subset M of $S \times P$ containing exactly one edge incident to each vertex; each edge $\{s, p\} \in M$ defines a leader between a site s and a port p.

Not all perfect matchings in G yield equally good labelings; some may actually imply crossing leaders. To reduce visual clutter, we prefer short leaders. Let $d(s, p)$ be the Euclidean distance of points s and p in the plane. We minimize the sum of the distances over all matched pairs of a site and a port location.

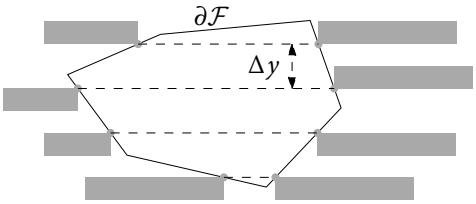


Figure 8.8: Construction of port locations (black dots) for a convex focus region $\mathcal{F}$ based on a set of horizontal lines (dashed) with vertical spacing Δy. The gray rectangles represent labels.

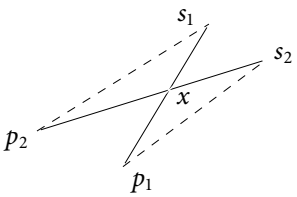


Figure 8.9: A crossing of two leaders (solid edges) can always be resolved (dashed edges) while reducing the total leader length.

Problem 8.5 (Minimum distance port assignment)**.**
Input: *Convex region $\mathcal{F}$, set $S \subset \mathcal{F}$ of n sites, set $P \subset \partial\mathcal{F}$ of n ports.*
Output: *Perfect matching M in the graph $G = (S \cup P, S \times P)$ such that $\sum_{\{s,p\}\in M} d(s,p)$ is minimized and no two leaders defined by edges in M cross.*

To simplify our discussion, we observe that the explicit requirement for non-crossing leaders can be dropped. Even without that requirement, every optimal solution to Problem 8.5 is crossing-free as Bekos et al. [BKSW07] observed.

Observation 8.1 ([BKSW07])**.** *Every perfect matching M in $G = (S \cup P, S \times P)$ that minimizes $\sum_{\{s,p\}\in M} d(s,p)$ defines a crossing-free labeling.*

To see that this is true, we consider two leaders $L_1 = \{p_1, s_1\}$ and $L_2 = \{p_2, s_2\}$ that cross each other in point x; see Figure 8.9. Since $d(p_1, s_1) = d(p_1, x) + d(x, s_1)$, $d(p_2, s_2) = d(p_2, x) + d(x, s_2)$, and d satisfies the triangle inequality, both leaders have total length

$$\begin{aligned} &d(p_1,x) + d(x,s_1) + d(p_2,x) + d(x,s_2) \\ &\quad = (d(p_1,x) + d(x,s_2)) + (d(p_2,x) + d(x,s_1)) \\ &\quad < d(p_1,s_2) + d(p_2,s_1). \end{aligned}$$

Thus, replacing L_1 and L_2 by the leaders $\{p_1, s_2\}$ and $\{p_2, s_1\}$ reduces the total leader length.

Due to Observation 8.1, Problem 8.5 reduces to the problem of finding a minimum-weight perfect matching in a bipartite graph whose nodes represent points in $\mathbb{R}^2$ and whose edges have weights representing Euclidean distances; for any $\varepsilon > 0$, this problem can be solved in $O(n^{2+\varepsilon})$ time [AES99].

As discussed in Section 8.2.1, we often cannot label all sites. Interestingly, if we are given n sites and $k \le n$ port locations, we can find a crossing-free labeling for *any* subset $S' \subseteq S$ of k sites. This also follows from Observation 8.1. If the sites are weighted and we aim at maximizing the total weight of all labeled sites, we simply have to select the k sites of largest weights, which can be done in $O(n)$ time [BFP+73], and can then apply the algorithm for Problem 8.5 to these sites. This requires $O(n + k^{2+\varepsilon})$ time in total. If there are multiple sites of the same weight, however, selecting an arbitrary subset S' of maximum weight can result in unnecessarily long leaders.

Generally, we may even want to relax our requirement for a maximum-weight set of sites to achieve a labeling with shorter leaders, for example, to avoid that clusters of heavy-weighted

sites result in clusters of sites labeled with long leaders. In order to define the trade-off between our preferences for sites of high weights and leaders of short length, we introduce a weight factor $\lambda \in [0,1]$.

Problem 8.6 (Best trade-off between weight and leader length)**.**
Input: *Convex region $\mathcal{F}$, set $S \subset \mathcal{F}$ of n sites with weights $w\colon S \to \mathbb{R}^+$, set $P \subset \partial\mathcal{F}$ of k ports, weight factor $\lambda \in [0,1]$.*
Output: *Matching M with $|M| = \min\{k,n\}$ in the graph $G = (S \cup P, S \times P)$ that maximizes the objective $\lambda \sum_{\{s,p\}\in M} w(s) - (1-\lambda) \sum_{\{s,p\}\in M} d(s,p)$.*

Note that we do not require anymore that the matching is perfect, thus allowing some sites or port locations to remain unmatched. By requiring $|M| = \min\{k,n\}$, however, we ensure that the largest possible number of matches is selected—which is still true if there are fewer sites than locations for ports.

For $\lambda < 1$ we are sure that, if we reduce the leader length and keep the same set of sites labeled, the objective always increases. Hence, by Observation 8.1, every optimal solution to Problem 8.6 is free of crossings. For $\lambda = 1$ we can find an optimal solution without crossings by selecting a set $S' \subseteq S$ of highest weight that contains $\min\{k,n\}$ sites and labeling S' with minimum leader length (by solving Problem 8.6 with $S := S'$ and $\lambda := 0$).

We can solve Problem 8.6 by finding a maximum-weight matching in the bipartite graph $G = (S \cup P, S \times P)$ if we define the weight of an edge $\{s,p\}$ in G to be

$$\lambda \cdot w(s) - (1-\lambda) \cdot d(s,p) + C,$$

where C is a large constant ensuring that all edge weights are positive. For example, we may set C to the diameter of the focus region. This problem can be solved in $O(k^3 + n^3)$ time, using the Hungarian method [Kuh55].

8.3 Algorithms for the Radial-Leader Model

8.3.1 Label Maximization with Given Center

We now present an algorithm for Problem 8.1, that is, we maximize the number of labels in a radial-leader labeling such that, when seen from center c, every two labeled sites are separated by an angle of at least α.

We first select an arbitrary start node $s_1 \in S$ and sort the sites in S lexicographically according to their angles and distances with respect to c. Let the resulting sequence be $\mathcal{S} = \langle s_1, s_2, \ldots, s_n \rangle$. Then, we define the directed graph $G = (S, E)$ that contains the edge (s_i, s_j) with $i < j$ if the angle $\angle s_i c s_j$ between s_i and s_j in c is at least α. We are looking for a longest *closable* path P in G, that is, a path whose start and end vertex also form an angle of at least α.

Let P_{OPT} be a longest closable path in G. Assume that, for two consecutive sites s, s_j in P_{OPT}, there is a site s_i with $i < j$ and $(s, s_i) \in E$. In this case, we can replace s_j with s_i and obtain another longest closable path. Therefore, we can always assume that for every site s in P_{OPT} (except for the last site) the next site t in $\mathcal{S}$ with $(s,t) \in E$ is also contained in P_{OPT}. This allows us to remove many edges from G while still having the guarantee that a longest closable path yields an optimal solution. More precisely, we compute the longest closable path in the reduced

graph $G' = (S, E')$; see Figure 8.10. The set $E' \subseteq E$ contains at most one outgoing edge for each node $s \in S$, that is, the edge $(s, s_i) \in E$ with smallest index i (if such an edge exists). Clearly, the complexity of G' is $O(n)$ and, if we have already computed the sequence $\mathcal{S}$, we can compute G' in $O(n)$ time.

Choosing a dynamic-programming approach, we now compute for $i = n$ down to 1 the length κ_i and the end σ_i of a longest (not necessarily closable) path in G' that starts in s_i. Obviously, for $i = n$, this is the trivial path containing only s_n, which implies $\kappa_n = 1$ and $\sigma_n = s_n$. Similarly, if s_i with $i < n$ does not have a successor, that is, a node s_j with $(s_i, s_j) \in E'$, the longest path starting there consists only of s_i and we can set $\kappa_i = 1$ and $\sigma_i = s_i$. If a site s_i with $i < n$ has a successor s_j in G', we compute κ_i and σ_i based on the values computed before as $\kappa_i = \kappa_j + 1$ and $\sigma_i = \sigma_j$. Hence, computing κ_i and σ_i for $1 \le i \le n$ takes $O(n)$ time in total.

For each of the paths computed with the dynamic program (that is, for $1 \le i \le n$) we can test in $O(1)$ time whether it is closable, simply by testing whether or not $\angle \sigma_i c s_i \ge \alpha$ holds. If the longest path P_i starting at node s_i is closable, then it is obviously longest among all closable paths starting at s_i. On the other hand, if P_i is *not* closable we can be sure that there is no closable path of length κ_i starting at s_i and that we can obtain a closable path of length $\kappa_i - 1$ by removing the last node from P_i. In any case, we obtain a longest closable path starting at s_i, which we denote by P^i_{OPT}. Obviously, a path that is longest among $P^1_{\mathrm{OPT}}, P^2_{\mathrm{OPT}}, \cdots, P^n_{\mathrm{OPT}}$ is a longest closable path in G'.

Summing up, it takes us $O(n)$ time to compute a longest closable path, which is dominated by the time ($O(n \log n)$) needed for sorting S.

Theorem 8.1. *We can solve Problem 8.1 in $O(n \log n)$ time, that is, given a set S of n sites in a disk $D = (c, r)$, and an angle α, we can find a maximum-cardinality conflict-free subset $S' \subseteq S$ with respect to the angle α in $O(n \log n)$ time.*

8.3.2 Weighted Label Maximization with Given Center

We now consider Problem 8.2, that is, the weighted version of the previous problem. In order to find a set S' of maximum weight, we again use the circular order of sites around c, and the graph G' defined in Section 8.3.1 with a small modification. We now have an edge connecting a site s_i to the next conflict-free site s_j in circular order even if $i > j$; see Figure 8.11. We choose some starting site $s' \in S$, and suppose that $s' \in S'$. Then, we go through all sites in counter-clockwise order, starting at s'. During this process we store for each $s \in S$ the maximum weight $T[s]$ of a conflict-free set in the range from s to and including s'. We compute $T[s]$ as follows.

For s', the range contains just s', and, hence, $T[s'] = w(s')$. If a site s does not have a successor t in G' such that t lies in the range from s to and including s', then none of the sites in this range (excluding s') can be labeled together with s'. Hence, a maximum-weight conflict-free set in this range consists just of s', that is, $T[s] = w(s')$. Now, assume we have a site s for which the successor t in G' lies in the range from s to and including s'. Let $\tilde{s}$ be the successor of s in clockwise order; see Figure 8.11. If we select s, then the next conflict-free site is t and an maximum-weight conflict-free solution consists of s and an optimum solution for t. Otherwise, that is, if s is not contained in an optimum solution, an optimum solution for $\tilde{s}$ is also optimum for s. Combining the two cases, we get $T[s] = \max\{w(s) + T[t], T[\tilde{s}]\}$. Hence, we can find a

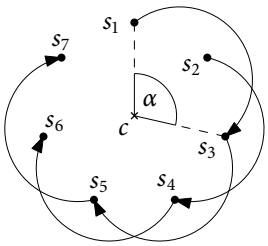


Figure 8.10: A reduced graph G' for sites $\mathcal{S} = \langle s_1, \cdots, s_7 \rangle$. For site s there is an edge to the next site t with $\angle sct \geq \alpha$ (if such a site exists). The longest path in G' is $\langle s_1, s_3, s_5, s_7 \rangle$, which is not closable since $\angle s_7 c s_1 < \alpha$. A longest *closable* path is, for example, $\langle s_1, s_3, s_5 \rangle$.

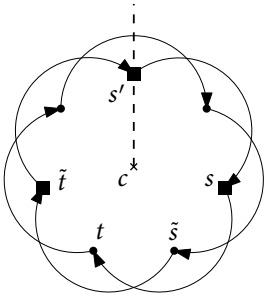


Figure 8.11: A maximum-weight conflict-free set of sites in the range from s to s' (marked by boxes). Here, $T[s] = T[t] + w(s)$. In contrast, $T[t]$ has been computed as $T[t] = T[\tilde{t}]$.

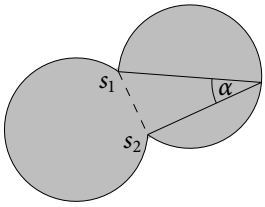


Figure 8.12: A partial double disk $U^{\alpha}_{s_1,s_2}$.

maximum-weight set S' with $s' \in S'$ in $O(n)$ time. By starting this algorithm at each site s', we maximize the total weight of visible labels in $O(n^2)$ time.

Theorem 8.2. *We can solve Problem 8.2 in $O(n^2)$ time, that is, given a set S of n sites in a disk $D = (c, r)$, a weight function $w\colon S \to \mathbb{R}^+$, and an angle α, we can find a maximum-weight conflict-free subset $S' \subseteq S$ with respect to the angle α in $O(n^2)$ time.*

8.3.3 Label Maximization with Variable Center Position

We now present an algorithm for Problem 8.3, that is, we maximize the number of non-conflicting sites separated by an angle α, but now, the center c of the circular focus region is not prescribed.

For each pair of sites $s_1, s_2 \in S$ the points from which the line segment $\overline{s_1 s_2}$ appears at an angle of at least α form a region $U^{\alpha}_{s_1,s_2} = \{c \in \mathbb{R}^2 \mid \angle s_1 c s_2 \geq \alpha\}$. According to the *inscribed angle theorem*, the angle at vertex c of a triangle abc does not change if c moves on the circumcircle of the triangle (while staying on the same side of the straight line supported by a and b). Therefore, $U^{\alpha}_{s_1,s_2}$ is the union of two partial disks, one on each side of $\overline{s_1 s_2}$. We thus call $U^{\alpha}_{s_1,s_2}$ a *partial double disk*; see Figure 8.12.

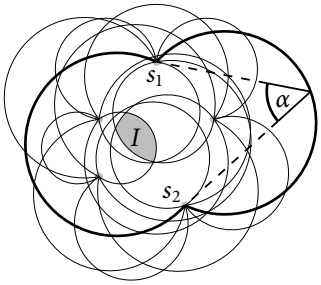


Figure 8.13: Arrangement $\mathcal{A}_S^\alpha$ containing the boundary of each partial double disk $U_{u,v}^\alpha$ with $u, v \in S$. Region I is the intersection of all partial double disks. In this example, for every point $c \in I$, (c, S) is an optimum solution to Problem 8.3.

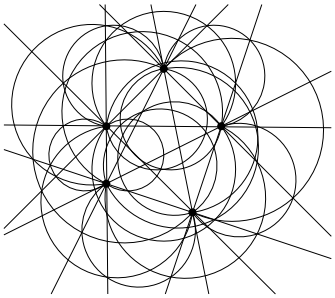


Figure 8.14: Arrangement $\mathcal{A}_S^\alpha$ of Figure 8.13 extended by adding all straight lines formed by pairs of sites.

We denote by $\mathcal{A}_S^\alpha$ the arrangement given by the union of the boundaries of the n^2 partial double disks for all pairs of sites; see Figure 8.13. Since two partial double disks can only intersect $O(1)$ times, the arrangement has a combinatorial complexity of $O(n^4)$. We can traverse the whole arrangement and visit every cell in $O(n^4 \log n)$ time by using a sweep line algorithm. For doing so, we just need to maintain a linked list for the order of disk boundaries that the vertical sweep line intersect, as well as an event queue for the event points in which disks start, end, or intersect; maintaining the event queue causes logarithmic overhead. During the sweep we determine for each cell C a set $S' \subseteq S$ of maximum size such that for every point $c \in C$ the sites in S' are separated by an angle of at least α (when seen from c). With our algorithm from Section 8.3.1 we can do this in $O(n \log n)$ time. Together with the time needed for traversing all cells we get an $O(n^5 \log n)$-time algorithm for finding an optimal center.

We can reduce the running time of the algorithm to $O(n^5)$ by updating the circular order from cell to cell instead of sorting the points $O(n^4)$ times. To this end, we extend the arrangement $\mathcal{A}_S^\alpha$ by adding all straight lines formed by a pair of sites; see Figure 8.14. Note that the complexity of this extended arrangement is still $O(n^4)$ and that the arrangement can be traversed by a sweep line algorithm in $O(n^4 \log n)$ time. When moving a center c inside any of the $O(n^4)$ cells of the extended arrangement, the circular order in which the sites are seen from c does not change. Furthermore, when moving the center from one cell to an adjacent cell, the order can be updated. This is done by a single swap of two sites if the cells are separated by the straight-line supporting the two sites (and the separation is not part of the straight-line segment between the sites).

Hence, we can easily update the circular orders in one step of our sweep line algorithm in $O(1)$ time. As finding an optimum set $S' \subseteq S$ of sites needs only $O(n)$ time for each cell, the total running time for finding an optimum center improves to $O(n^5)$.

Theorem 8.3. *We can solve Problem 8.3 in $O(n^5)$ time, that is, given a set S of n sites and an angle α, we can find a center c and a maximum-cardinality conflict-free subset $S' \subseteq S$ with respect to the center c and the angle α in $O(n^5)$ time.*

In the weighted setting, we can combine the traversal of the disk arrangement with the technique presented in Section 8.3.2. This yields an $O(n^6)$-time algorithm that finds an optimum center position.

8.3.4 Sector Maximization

Let us first address the decision problem derived from Problem 8.4, that is, we want to decide if we can find a center such that all sites are separated by a given angle α. This question can be answered with the help of the arrangement $\mathcal{A}_S^\alpha$ introduced in the previous section. We can find such a center c, if there is a cell in $\mathcal{A}_S^\alpha$ that is covered by all possible partial double disks. In order to find such a cell, or to report that none exists, we use again a sweep line algorithm. We keep track of the number of partial double disks that cover the cells we are observing during the sweep. When traversing from one cell to another we can update this information in $O(1)$ time. Hence the decision problem can be solved in $O(n^4 \log n)$ time.

In order to maximize α such that a center c_α with minimum angle at least α exists, we use the following technique: Consider the arrangement $\mathcal{A}_S^{\alpha'}$ parametrized by α'. As long as there exists a cell with positive volume that is covered by all partial double disks, we can decrease the angle α' and obtain a new arrangement with such a cell. It follows that for an optimal angle α, there has to be at last one degenerated cell, formed by a single point, that is covered by all partial disks. This means, however, that three or two of the partial double disks meet in one point. Hence, it suffices to compute for all triplets and tuples of input point pairs the angle where their induced three (or two, respectively) partial double disks meet in a single point. The smallest such angle determines the value α. The running time of this strategy is $O(n^6)$.

Theorem 8.4. *We can solve Problem 8.4 in $O(n^6)$ time, that is, given a set S of n sites we can find an angle α and a center c such that S is conflict-free with respect to c and α, and α is the largest angle for which we can find a feasible center position.*

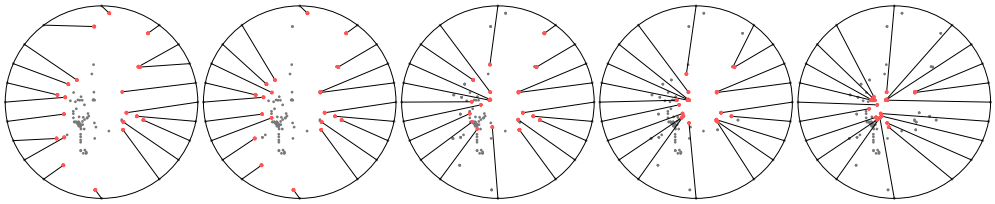
For Experiment 8.1 and 8.2 we have used a prototypical implementation to solve the sector maximization problem. We did not use the proposed (exact) $O(n^6)$-time solution, but solved the sector maximization problem numerically. More precisely, we formulated Problem 8.4 as a minimax problem and solved it by the computer algebra software *Mathematica* via the *differential evolution* method. Our solutions converged for almost all computed instances within reasonable time (see Table 8.1).

#sites	5	15	25	35	45	55	65	75	85
time[s]	1.83	2.10	2.68	3.82	5.26	6.80	10.07	12.59	15.38

Table 8.1: Runtime for the numerical solution of the sector maximization problem on a standard 2-core 3 GHz desktop computer.

8.4 Extensions

In this section we discuss two extensions that can be applied to our leader models. First, we show how to simultaneously cluster sites and label a representative site from each cluster (Section 8.4.1).



(a) $\lambda = 0$ (b) $\lambda = 0.45$ (c) $\lambda = 0.50$ (d) $\lambda = 0.55$ (e) $\lambda = 1$

Figure 8.15: Solutions to Problem 8.6 obtained with different values for λ. Since sites close to the center c of the focus region are assumed to be important, the weight $w(s)$ of a site s was set to the Euclidean distance between s and c. Gray dots show sites that did not become labeled.

Second, we transform, in a post-processing step, our straight-line leaders into well-shaped Bézier curves. For the radial-leader model, this gives us more flexibility in the placement of the ports as compared to straight-line leaders. Hence, we can either place more or larger labels. For the free-leader model, Bézier curves help to lead the user's eye in a natural, bend-free way from the site to the label (or vice versa).

8.4.1 Simultaneous Clustering and Labeling

When discussing the free-leader model (Section 8.2.2) we dealt with the problem that the input map may contain significantly more sites than we have leader ports available. In this case, we had to *select* a subset of sites that are actually labeled and to *assign* these sites to the ports.

In Problem 8.6, we addressed both issues (selection and assignment) by an extension of bipartite matching. Our experiments indicate, however, that the sites selected by optimum solutions to Problem 8.6 do *not* sufficiently reflect the spatial distribution of the sites. Consider Figure 8.15 that shows various solutions for the same input but different values of the weight factor λ. We used a *non-uniform* weight function w where the weight of a site equals its distance to the center of the circle. This can be justified by the assumption that sites closer to the center of the focus region could be considered more important for the user. Simultaneously, this avoids that only sites close to the boundary are labeled, which happens if we use a uniform weight function or if we set $\lambda = 0$ (see solution (a)). Interestingly, this preference of sites close to the boundary persists even if we increase λ to values slightly below 0.5 (see solution (b)). As soon as we increase λ to values greater than or equal to 0.5 (see solutions (c)–(e)) the optimum solution exhibits a preference for sites closer to the center. What we actually want is, however, that the labeled sites are *evenly* distributed thereby reflecting their spatial distribution. This is not sufficiently accomplished by the solutions for Problem 8.6 shown in Figure 8.15.

For example, if the set of sites contains a dense cluster, the selected subset of labeled sites should also contain a (possibly less dense) cluster in the same area. Simultaneously we want to construct a suitable assignment of the labeled sites to the ports, which is represented by the sum of leader lengths.

We now present a novel free-leader model that is based on *facility location*. Our model simultaneously addresses the problem of *selecting* a suitable subset of sites that are labeled and

of obtaining a good assignment of the selected sites to the leader ports. The main benefit of this model in comparison to Problem 8.6 is that the subset of labeled sites reflects the *spatial distribution* of the complete set of sites.

The problem of selecting a subset of given cardinality k from a set of input points so that the selected points "represent" the entire set in a spatial sense is a common problem that arises, for example, in *clustering* and *location theory*. Popular formulations as an optimization problem are the *k-means* and the (Euclidean) *k-median* problem. For the k-median problem the input is a set S of points (sites) and a number $k \leq |S|$. The goal is to find a k-element set of *facilities* and to connect each site to a facility so that the *total connection cost*, that is, the sum of the Euclidean distances from the sites to their respective facilities is minimized. In clustering theory, the facilities opened by an optimum solution are considered as *cluster centers* that reflect the *spatial distribution* of the set S and the clusters are the subsets of sites that are connected to the same cluster center (facility).

A natural idea is to select, in a first stage, the sites that we want to label by computing a solution to the corresponding k-means or k-median problem and choosing the cluster centers as the selected sites. Then, in a second stage, one can compute a good assignment of these cluster centers to ports. This can, for example, be accomplished by computing a minimum weight bipartite perfect matching between the cluster centers and the ports. The drawback of this approach is that it divides the problem into two separate optimization stages (selection and assignment) and exclusively prefers the selection goal to the assignment goal.

In what follows we present a capacitated extension of the Euclidean k-median problem and the closely related facility location problem that incorporates selection and assignment into one neatly formulated optimization problem. We aim at selecting k sites for labeling (considered as facilities) and connecting the remaining sites in S (considered as customers) to a facility. The k-median problem can then be considered as the problem of opening k facilities and connecting each site to a facility such that the *total connection cost* (sum of distances of sites to their facilities) is minimized. Moreover, we need to ensure that each facility (labeled site) is connected to a *port*. This fits nicely into our location model as we can simply consider the ports as additional, special customers that need to be connected to the facilities. The previously conflicting goals of finding a good selection of sites and finding a good assignment of these sites to ports are then subsumed by the single goal of minimizing the total connection cost. We have to require, however, that each facility serves exactly one port in order to ensure a one-to-one correspondence between labeled sites and ports. Problem 8.7 gives a formal description of our model.

Problem 8.7 (Facility-location-based labeling)**.**

Input: *Convex region $\mathcal{F}$, set $S \subset \mathcal{F}$ of n sites with opening costs $c\colon S \to \mathbb{R}_0^+$, set $P \subset \partial\mathcal{F}$ of $k \leq n$ ports, factor $\lambda \in [0,1]$.*

Output: *A* feasible solution *consists of a k-element set $S' \subseteq S$ of* facilities *and an assignment $\sigma\colon S \cup P \to S'$ specifying for each site or port to which facility it is connected. Each facility must be connected to exactly one port and to at most $\lceil |S|/k \rceil$ sites. A feasible solution has a* total opening cost *of $\sum_{f \in S'} c(f)$ and a* total connection cost *of $\lambda \sum_{p \in P} d(p, \sigma(p)) + (1-\lambda) \sum_{s \in S} d(s, \sigma(s))$. The output is a feasible solution that minimizes the sum of total opening and connection cost.*

We assign to each site s a cost $c(s)$ that is incurred if a facility is opened at s. This cost reflects the importance of labeling the site s where a smaller opening cost means higher importance.

The overall goal is to minimize the sum of opening costs and total connection cost. We impose an additional *capacity constraint* on each facility that ensures that each facility is connected to roughly the same number of sites. The reason is that in the *uncapacitated* variant optimum solutions tend to place a comparatively small number of facilities into a spatially dense accumulation of points since all the sites in such an accumulation can efficiently be served by few facilities. By adding the capacity constraint we ensure that dense accumulations lead also to accumulations in the set of facilities.

Another nice property of our model is that in optimum solutions there are no intersections between port-facility connections and no intersections between site-facility connections. This can be shown analogously to Observation 8.1. See also Figure 8.16.

Integer Linear Program

We now present a formulation of Problem 8.7 as an integer linear program (ILP). To this end, we introduce for each site $i \in S$ a binary variable $y_i \in \{0,1\}$ specifying whether a facility is opened at i. For each pair $(i,j) \in S \times (S \cup P)$ we introduce a binary variable $x_{ij} \in \{0,1\}$ indicating whether j is connected to facility i. We can formulate our objective function as a linear function as follows:

$$\text{Minimize } \sum_{i \in S} c(i) y_i + \lambda \sum_{i,j \in S} d(i,j) x_{ij} + (1-\lambda) \sum_{(i,j) \in S \times P} d(i,j) x_{ij} .$$

It remains to formulate linear constraints ensuring that the variables y_i and x_{ij} describe a feasible solution to the input instance. First, we ensure that each $j \in S \cup P$ can only be connected to a site i where a facility is actually opened by means of the constraint

$$x_{ij} \leq y_i \quad \text{for each } i \in S, j \in S \cup P .$$

To make sure that each $j \in S \cup P$ is connected to exactly one facility, we require that

$$\sum_{i \in S} x_{ij} = 1 \quad \text{for each } j \in S \cup P .$$

The requirement that each port is connected to exactly one facility can be described by

$$\sum_{i \in S} x_{ij} = 1 \quad \text{for each } j \in P .$$

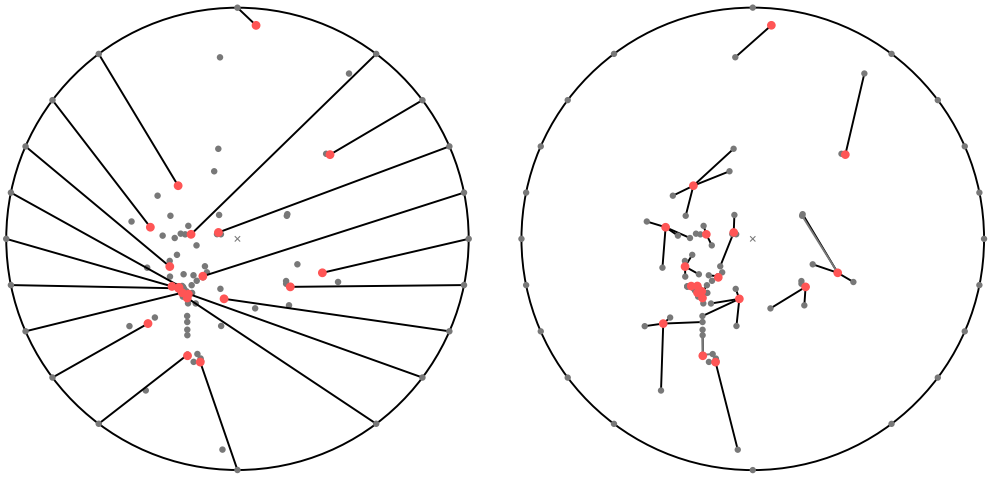
We further require that each facility is connected to only one port, which we can model by

$$\sum_{j \in P} x_{ij} \leq 1 \quad \text{for each } i \in S .$$

The combination of the last two constraints ensures a one-to-one correspondence between ports and opened facilities.

To make sure that the maximum number of sites per facility is not exceeded we require

$$\sum_{j \in S} x_{ij} \leq \lceil |S|/k \rceil \quad \text{for each } i \in S .$$



(a) Selected sites and leaders connecting them to the assigned ports.

(b) Site-facility connections.

Figure 8.16: Optimum solution to Problem 8.7 for the Seattle instance ($n = 95$).

Finally, we ensure that exactly k facilities are opened by means of the constraint

$$\sum_{i \in S} y_i = k\,.$$

Experiment 8.3. We implemented the integer linear program in C++ using the optimizer Gurobi (version 4.6.1). We computed an optimum solution for the same data set that we used in Figure 8.15 for discussing Problem 8.6. It consists of $n = 95$ sites (restaurants on a map excerpt of Seattle) and $k = 20$ ports. The result is shown in Figure 8.16. Observe that the selection of labeled sites much better represents the spatial distribution of all sites than the solutions to Problem 8.6. The computation of the optimum solution took 124s on a PC with an Intel Core2Duo E8400 CPU with 2 cores at 3 GHz each and 4 GB RAM. In comparison, the computation of the optimum solutions to Problem 8.6 with 50 different values of α took less than 1s in total on the same PC.

As the integer programming solution is not suitable for interactive usage, we also tested a heuristic approach for finding good solutions quickly in two steps. First, we use a randomized algorithm for finding k cluster centers. The algorithm chooses new centers one after the other, where, in each step, the probability of choosing a point is proportional to its distance to the closest center chosen so far. This algorithm is known to find a solution to the *k-median* problem with an expected approximation ratio of $O(\log k)$ [AV07]. Next, we compute crossing-free leaders connecting the chosen centers to the ports by using our matching approach. The algorithm for finding the cluster centers is even faster than the matching algorithms, and yields solutions in which labeled points tend to be in dense regions, but not too close to each other. If a complete clustering is needed, the unlabeled points may be assigned to their closest center; see also the section on the uncapacitated relaxation on page 168.

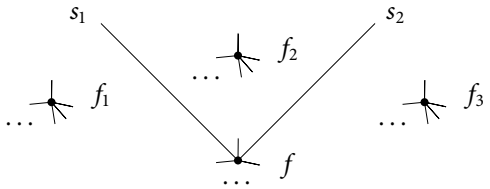


Figure 8.17: Example where the convex hulls of two optimum clusters intersect and where the Voronoi cells of the sites of one cluster are not connected.

Disk-in-Disk Visualization for Labeling Large Point Sets

When dealing with large and dense point sets, we face the problem that only a small fraction of the sites can actually be labeled since labels must be large enough and may not overlap in order to be readable. We present a natural method based on our facility location model that *clusters* the sites into connected, non-overlapping subregions, one for each label. The user can select a subregion—for example, by clicking the corresponding label—which is then scaled up and labeled in more detail; see Figure 8.1. In the new labeling, we keep the selected label fixed, even if this causes crossings with leaders of other labels. We think that this is less distracting than if the selected label jumped to its new position.

We associate each facility f with the set C_f of all sites that are connected to that facility. This set C_f forms a cluster, and the label for C_f is the unique label L_f that is connected to facility f. Note that no cluster contains more than $\lceil |S|/k \rceil$ sites due to the capacity constraints. Our goal is to embed each cluster C_f into a connected subregion $R_f \subseteq \mathcal{F}$ so that no two subregions overlap. A possible use case is to visually highlight the region R_f whenever the cursor hovers over the label L_f. We want to identify subregions that can be easily recognized by the user. Unfortunately, it is not always possible to find *convex* subregions that meet the above requirements.

Figure 8.17 shows an instance with optimum solution for $k = 4$ labels where the convex hulls of the two clusters overlap. The instance contains sites f, f_1, f_2, and f_3 where f_1, f_2, and f_3 are each surrounded by a large number Ω of sites at small distance ε. Similarly, f is surrounded by $\Omega - 2$ many sites at distance ε. Moreover, there are two more distant sites s_1 and s_2. If Ω is large enough the optimum solution must choose f, f_1, f_2, and f_3 as the locations for the facilities. If ε is small enough the sites are connected to the facilities as shown in the picture since all facilities must serve the same number of sites due to the capacity constraints. The convex hull of cluster C_{f_2} is contained in the convex hull of C_f.

The same example also shows that the Voronoi cells of the sites s_1 and s_2 can be disconnected from the Voronoi cells of the remaining sites in cluster C_f. Hence, also using the Voronoi cells does not yield a nice partition of the region $\mathcal{F}$.

Triangulation-Based Partitioning. We now propose a simple approach that partitions $\mathcal{F}$ into a collection of subregions containing one cluster each. Our approach is similar to the computation of skeletons based on constrained Delaunay triangulations [CBB91, BW97] and on the work of Reinbacher et al. [RBvK+08] who deal with the problem of computing polygons that separate *two* given point sets in the plane.

Let $S' \subseteq S$ be the set of cluster centers that are to be labeled. We define the *star* S_f for facility $f \in S'$ as the set of line segments $\overline{fs}$ with $s \in C_f$. Note that no two stars intersect since site-facility connections do not cross. We assume that the boundary $\partial\mathcal{F}$ of $\mathcal{F}$ is given as a polygon. (Otherwise, we can choose a set $\mathcal{F}'$ of sufficiently many points from $\partial\mathcal{F}$ and replace $\mathcal{F}$ with the convex hull of $\mathcal{F}'$.) We compute the union of $\partial\mathcal{F}$ with all stars S_f for $f \in S'$ and complete the resulting set of non-intersecting line segments to a complete triangulation T. For example, we can compute a *constrained Delaunay triangulation* in $O(n \log n)$ time [Che89].

We then partition the region $\mathcal{F}$ into connected subregions R_f for each $f \in S'$ such that C_f is contained in R_f. This is accomplished by partitioning each triangle separately as shown in Figure 8.18a.

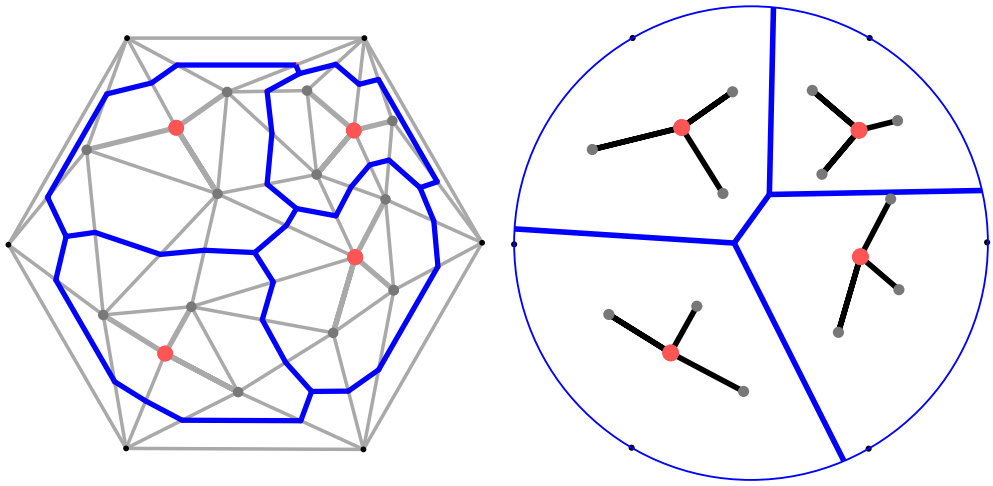
Consider a triangle spanned by three sites. (We treat the vertices on $\partial\mathcal{F}$ as sites of a special cluster.) If all three sites belong to the same cluster, we do not partition the triangle. If the sites belong to two different clusters, we partition the triangle into two *pieces* by cutting along the line segment that connects the midpoints of the two edges whose end points lie in different clusters. In case the three sites belong to three different clusters, we partition the triangle into three pieces by cutting along the three line segments that connect the midpoints of the edges with the center of gravity of the triangle; compare Figure 8.18a for occurrences of the three cases. Note that we partitioned the triangle according to the clustering, that is, sites of the same cluster lie in the same piece and sites of different clusters lie in different pieces. Therefore, we can assign each piece of the triangle to a unique cluster. We perform the above partitioning for every triangle. For each cluster C_f, the region R_f is the union of all pieces that are assigned to the cluster C_f.

The above algorithm ensures that each site of a cluster C_f is contained in R_f and that two distinct regions R_f and $R_{f'}$ are interior-disjoint. Every region R_f is connected as it contains the star S_f. Finally, it is easy to verify that, for each facility f, the boundary ∂R_f of R_f is a simple polygon that consists only of line segments along which we cut triangles.

Uncapacitated Relaxation and Voronoi-Based Partitioning

An interesting relaxation of Problem 8.7 is to drop the capacity constraints for sites, that is, to allow that an arbitrary number of sites is connected to the same facility. Of course, we still insist on exactly one port per facility. This relaxation, gives less incentive to place multiple facilities into spatially dense accumulations of sites since all sites in such an accumulation can efficiently be served by one facility.

A nice property of this relaxed version is that we now can determine a natural decomposition of $\mathcal{F}$ into *convex* and pairwise disjoint regions that cover the clusters of an optimum solution. To this end, consider an optimum solution to the relaxed problem with a set S' of facilities. Now consider the *Voronoi diagram* for the point set S'. Then each facility f lies in a unique Voronoi cell V_f; see Figure 8.18b. These cells form a natural decomposition of $\mathcal{F}$ into interior-disjoint *convex* regions. It is easy to see that for each f the cluster C_f is contained in the Voronoi cell V_f. Assume to the contrary that a site $s \in C_f$ lies in the interior of a cell $V_{f'}$ with $f' \neq f$. Then s would be strictly closer to f' than to f. Therefore, we could obtain a strictly cheaper solution by re-connecting s to f', which would not affect the feasibility since we impose no capacity constraints on f'. This contradicts the optimality of the solution.



(a) Triangulation-based (b) Voronoi-based

Figure 8.18: Partitioning the focus region.

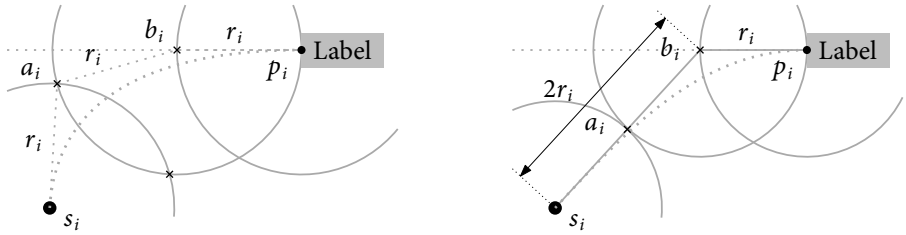
8.4.2 Bézier Post-Processing

In all our previous methods, we used crossing-free straight-line leaders for connecting sites and labels. Now we want to improve these drawings by using more flexible leaders. In previous works on boundary labeling, the only drawing styles for leaders have been straight-line segments and polylines. We, however, use Bézier curves, which are easier to follow than polylines since they do not have sharp bends. More precisely, we use cubic Bézier curves, which are defined by their endpoints and two intermediate control points; see Section 2.2.

For computing the new leader layout, we use the *force-directed approach* similar to the algorithm for drawing metro maps using Bézier curves presented in Chapter 3. Our algorithms start with a straight-line drawing and let forces, defined by physical analogies, iteratively transform the drawing. In contrast to Chapter 3, we have several vertices (the sites) that must not be moved. Again, we define several forces that are applied to the Bézier curves; each force optimizes a certain aspect of the drawing. In each iteration, the desired movement vectors for all curves are computed by summing up the single forces. Before applying these movements to the current drawing, we limit some movements, if necessary, to ensure that the new drawing is crossing-free. Our algorithms terminate when a prespecified number of iterations is reached or the maximum movement per iteration is very small compared to the distances between the input sites.

Horizontal Labels with Given Ports

For the free leader model (Section 8.2.2) our main requirement is that the new leaders enter the labels horizontally from within the focus region. Additionally, the drawing should stay crossing-free. Subject to these constraints, we would like the curves to be as smooth as possible; see Figure 8.1b at the beginning of the chapter (page 147).



(a) Equidistant positions of control points. (b) Minimizing equal distance r_i.

Figure 8.19: Placing the control points of a Bézier leader for a horizontal label.

Problem 8.8 (Bézier leaders for horizontal labels and given ports)**.**

Input: *Disk $D = (c, r)$, sites $s_1, \cdots, s_n \in D$, ports $p_1, \cdots p_n \in \partial D$ such that the segments $\overline{s_1 p_1}, \ldots, \overline{s_n p_n}$ are crossing-free.*

Output: *Bézier curves $B_1, \cdots, B_n$ such that, for $i = 1, \cdots, n$, curve B_i connects s_i to p_i and enters port p_i horizontally, and no two Bézier curves intersect. Curves should not be too close to each other and should not have high curvature.*

During our force-directed algorithm, sites and ports will, of course, stay fixed. For $i = 1, \cdots, n$, we initialize curve B_i by making s_i and p_i its endpoints and by placing the two intermediate control points a_i and b_i on their closer endpoint, that is, on s_i and p_i, respectively. Hence, initially $B_i = \overline{s_i p_i}$ and, according to our assumption, this initial drawing is crossing-free. In order to improve the shapes of our curves, we must move the control points.

It is clear that, if we want the leader to enter port p_i horizontally, b_i must move, but stay on the horizontal line through p_i and inside the focus region. We must also move a_i away from s_i to get a good curve. For a nice-looking curve, it makes sense to place the control points so that the three segments of the polyline $s_i a_i b_i p_i$ have roughly equal length r_i. As Figure 8.19a indicates, there are—under this restriction—two possible positions for a_i if r_i is large enough.

On the other hand, we strive to keep the leader short in order to make it easier to follow. Hence, we try to keep $r_i = d(b_i, p_i)$ as small as possible. In this situation, the only possible position for a_i is the center of $\overline{s_i b_i}$; see Figure 8.19b. To keep visual and computational complexity small, we fix a_i to this position. Our algorithm modifies, therefore, only the parameter $r_i = d(b_i, p_i)$, while a_i will automatically always be in the middle between s_i and b_i. Let r_i^{OPT} be the optimal value of r_i for B_i, that is, the value resulting in $d(b_i, s_i) = 2r_i^{\mathrm{OPT}}$. Then the attracting force on b_i is

$$f_{\mathrm{attr}}(B_i) = r_i^{\mathrm{OPT}} - r_i,$$

that is, the "vector" from the current to the desired position.

An additional criterion for a pleasant-looking leader layout is that two leaders do not come too close. To this end, we add a repelling force between two leaders. Suppose that we have a pair of leaders B_i and B_j, as shown in Figure 8.20. We first compute the minimum distance $d(B_i, B_j)$ between the two curves. Note that this can easily be approximated by a polygonization of the curves. Given the relative position of the curves in Figure 8.20, the control point b_i of B_i

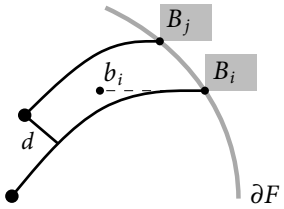


Figure 8.20: Curve B_j repels curve B_i.

should be repelled towards its port p_i. It is reasonable to make this repelling force larger when the distance between the curves gets smaller. Hence, we set

$$f_{\text{rep}}(B_i, B_j) = \frac{-r_i}{d(B_i, B_j)}$$

for the force repelling B_i from B_j. If the relative position of the curves is different, the force is defined analogously but the direction may change.

Finally, the total force on the control point b_i of a curve B_i is

$$f(B_i) = c_{\text{attr}} f_{\text{attr}}(B_i) + c_{\text{rep}} \sum_{B_j \neq B_i} f_{\text{rep}}(B_i, B_j),$$

where the weights c_{attr} and c_{rep} are still flexible. In our tests, $c_{\text{attr}} = c_{\text{rep}} = 0.5$ turned out to be a reasonable choice.

Once all forces are computed they should be applied onto the current drawing, that is, the new value of r_i should be $r_i' = r_i + f(B_i)$. Simply applying the forces could, however, lead to crossings, despite the repelling forces between leaders. Therefore we introduce limitations to the forces, that is, a maximum allowed change of the absolute value of r_i. We set this limitation to

$$f_{\max}(B_i) = 0.45 \min_{B_j \neq B_i} d(B_i, B_j)\,.$$

It is easy to see that, with a maximum movement of d on b_i, all points on the new curve B_i' lie within a distance of at most d from the old curve B_i. Hence, by moving b_i and b_j by at most $0.45d(B_i, B_j)$ we cannot create an intersection of the two leaders. It follows that limiting the absolute value of $f(B_i)$ to $f_{\max}(B_i)$—where necessary—before applying the forces guarantees that the drawing stays crossing-free.

The algorithm terminates when an equilibrium of forces is reached. In practice, it suffices that the maximum change in an iteration is very small, that is, much smaller than the distance between any two sites. Note that it suffices to compute forces just for pairs of leaders that can actually come close. Disregarding all unnecessary pairs gave a huge speedup in our experiments.

Experiment 8.4. We tested a Java prototype implementation of our algorithm on the same machine used for Experiment 8.3. We fixed the port positions at equal distances as described in Section 8.2.2 (see Figure 8.8 in this section). We were mainly interested in the increase of total leader length caused by improving the shape of the leaders. For the test, we used growing subsets of the Seattle instance (see Figure 8.1) where all sites had to be labeled (that is, $k = n$).

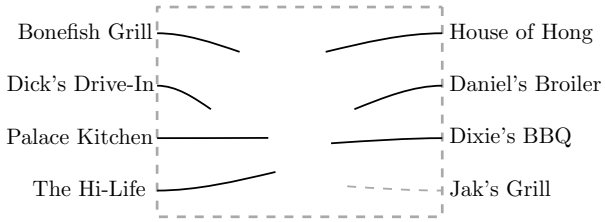


Figure 8.21: Rectangular boundary labeling with leaders drawn as Bézier curves.

In each new subset, we added sites farther from the city center and increased the radius of the focus region accordingly. For each instance, the algorithm did 200 iterations. Table 8.2 shows the results.

#sites (n)	10	20	30	40	50	60	70	80	90
time[ms]	25	169	123	141	121	170	168	253	354
incr. [%]	4.07	3.09	2.69	2.14	2.57	1.51	1.37	1.90	1.83

Table 8.2: Runtime for Bézier postprocessing; increase of leader length.

As we can see, the postprocessing increases the total leader length only by a small percentage while producing nice-looking leaders. For comparison, we refer to the Bézier leader layout in Figure 8.1b: There, the increase of leader length compared to the straight-line version was only 3.1%; the running time was 0.3s. Note that the running time heavily depends on the distribution of sites. If there are very dense regions such as the cluster in the center, the algorithm is slower due to strong repelling forces which increase the number of necessary iterations. If, however, the sites are better distributed, for example, after selecting a subset with the method presented in Section 8.4.1, the postprocessing is faster and produces better-looking leaders.

Traditional Boundary Labeling

Traditional, two-sided boundary labeling for rectangular maps has, so far, only been approached with straight-line or polyline leaders. It is, however, easy to create such a labeling using Bézier curves as leaders by using our techniques. Note that we did not use the shape of the focus region in the force-directed algorithm. Suppose that we are given a rectangular focus region with sites inside and ports on the boundary. Then we can compute a port–label assignment such that straight-line leaders do not cross by the matching technique that solves Problem 8.5. Using this assignment, we can apply the Bézier postprocessing presented above to get a crossing-free layout of Bézier leaders; see Figure 8.21.

Our approach can also be applied to further types of focus regions if we can ensure that the Bézier leaders stay inside the region.

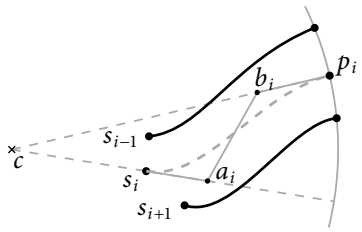


Figure 8.22: Bézier curve B_i between neighbors B_{i-1} and B_{i+1}.

Radial Labels without Given Ports

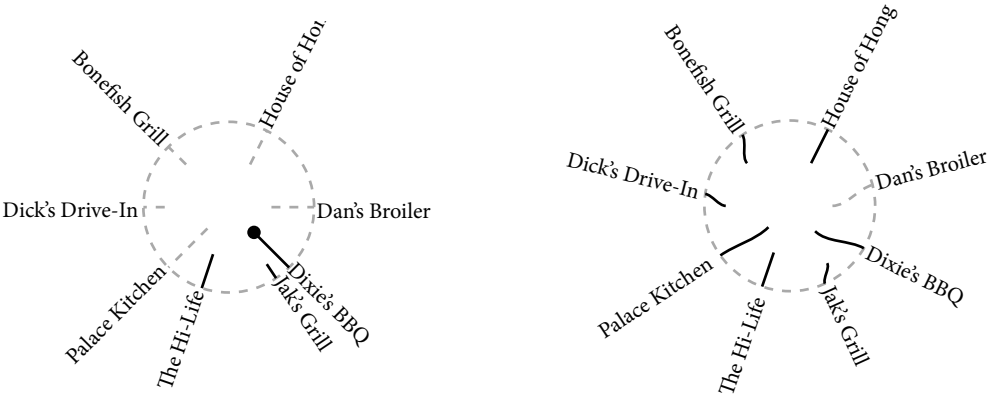
In the radial-leader scenario we cannot improve much by simply rerouting the leaders using Bézier curves: The leaders already leave sites as well as ports radially and have—as straight-line segments—minimum length. There could, however, be pairs of ports forming a small angle at the center compared to the necessary average angle $360°/n$, even after optimizing the smallest angle by using the algorithm presented in Section 8.3.4. We can optimize these angles by moving the ports on the boundary and rerouting the leaders using Bézier curves. We still want the leaders to leave sites and ports radially and insist on crossing-free leaders.

Problem 8.9 (Bézier leaders for radial labels without given ports)**.**
Input: *Disk $D = (c, r)$, sites $s_1, \cdots, s_n \in D$.*
Output: *Ports $p_1, \cdots p_n \in \partial D$ and Bézier curves $B_1, \cdots, B_n$ such that the circular order of the ports is the same as the order of their respective sites, for $i = 1, \ldots, n$, curve B_i leaves site s_i radially and enters port p_i radially, and no two Bézier curves intersect. The gaps between the ports should be of approximately equal length and the ports should be close to their sites.*

This problem can again be tackled using a force-directed algorithm: We start by setting, for $i = 1, \ldots, n$, port p_i to the projection of s_i onto ∂D, and by using the straight-line leader $\overline{s_i p_i}$, which is a special Bézier curve. In each iteration of the algorithm, we try to improve the distribution of the ports on ∂D under the additional requirements of Problem 8.9. We assume that the indices are chosen in such a way that the ports $p_1, \ldots, p_n$ occur in clockwise order on ∂D. We will keep this property during the iterations of the algorithm.

As we want to enter/leave sites and ports radially, the intermediate control points a_i and b_i of the curve B_i connecting site s_i and port p_i must lie on the straight lines cs_i and cp_i, respectively; see Figure 8.22. Thus, our algorithm keeps track of three parameters for each Bézier leader B_i: the position of port p_i and the distances $d(s_i, a_i)$ and $d(b_i, p_i)$. We introduce the following forces:

- A force attracting port p_i to the midpoint of p_{i-1} and p_{i+1} on ∂D, which is in the middle of the circular arc with center c connecting p_{i-1} and p_{i+1} in clockwise order. In an equilibrium, that is, if all ports have equal distance to both of their neighbors, all gaps are of equal length.
- A force attracting each port p_i towards its initial position, that is, the radial projection of s_i, for straightening the Bézier curve.



(a) Straight-line leaders.

(b) Bézier leaders after postprocessing.

Figure 8.23: Improvement of radial leaders by postprocessing with Bézier curves.

- Repelling forces on the control points if two neighboring curves are too close. This should avoid small distances between the leaders. We try to move both control points of B_i to the same direction such that a_i stays closer to the center of the disk than b_i.
- A force that tries to move a_i and b_i such that both distances $d(s_i, a_i)$ and $d(b_i, p_i)$ are 1/3 of the distance of s_i to the original position of p_i, which is $r - d(c, s_i)$.

For any force, it is enough to consider the two neighboring sites s_{i-1} and s_{i+1} in circular order. To avoid crossings between labels we limit forces if necessary. This can, again, be done based on the minimum distances to the two neighboring curves.

Experiment 8.5. The algorithm was implemented in Java and tested on the same machine and the same instances as in Experiment 8.4. We did 200 iterations per instance. As the positions of ports are flexible, we also measured the optimization criterion, that is, the smallest angle, and compared it to the upper bound $360°/n$. Table 8.3 shows the results.

#sites (n)	10	20	30	40	50	60	70	80	90
time [ms]	218	281	324	292	353	400	445	506	574
angle [%]	64.8	26.7	43.3	39.9	48.5	29.3	31.3	37.3	29.2
incr. [%]	11.9	6.4	0.9	1.2	1.0	0.6	0.6	0.4	0.3

Table 8.3: Runtime, minimum angle (% of $360°/n$), increase of length.

For $n \geq 20$, the initial straight-line solution had—due to the dense region of sites in the center—an initial minimum angle of less than (0.001°). Our post-processing always improved this to a reasonable percentage of the upper bound $360°/n$. As in Experiment 8.4, the relative increase of the total leader length was very small. For a visual inspection of the improvement, see Figure 8.23: when using Bézier curves, the minimum angle increased from 11° to 35° (upper bound 45°), whereas the total leader length increased by a mere 6.3%.

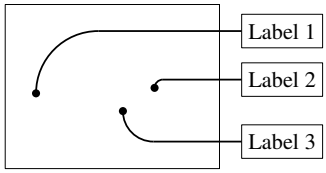


Figure 8.24: Leaders consisting of a straight-line and a circular arc.

8.5 Concluding Remarks

We have investigated the problem of labeling sites in a focus region by placing labels at the boundary of the focus region and connecting sites and their labels by leaders. We have considered the free-leader model and a new radial-leader model. Usually, users will insist on horizontal labels, which we recommend to combine with the free-leader model. If, however, it is crucial to label a lot of sites— especially with short labels—, it is better to place leaders and labels radially. We strongly recommend to take advantage of the Bézier postprocessing, especially for the radial-leader model which otherwise suffers from small port distances. In a dynamic environment, we suggest using straight leaders during user interaction. As soon as an interaction ends, one could turn the straight leaders into Bézier leaders by animating the execution of our force-directed algorithm.

Open Problems. Also for labeling focus regions, there are some open problems. We have presented an algorithm that creates a boundary labeling with curvy leaders. While the results in our tests looked nice and their computation was fast, we stress that the Bézier curves were only computed as a post-processing from an initial straight-line labeling. Potentially, better results could be obtained by directly computing curvy leaders using a new, specialized algorithm. As Bézier curves are not very easy to handle, also other leader styles with curves could be tried, for example, a horizontal line segment combined with a circular arc in smooth transition; see Figure 8.24. Another problem is the evaluation of curvy leaders in a user study. The main questions are whether users prefer the look of curvy leaders over other leader styles and whether curvy leaders make it easier for users to solve tasks like finding a certain site or finding the label for a highlighted site.

Both for the radial and the free-leader model, we presented fast algorithms for finding a labeling that maximizes the weight of labeled sites, if not all sites in the focus region can be labeled at the same time. The algorithms are fast enough for interactive use, that is, if the focus region is moved by the user. In such a move, some new sites can enter the focus region, some old sites leave the focus region, and other sites will remain in the focus region. So far, our solution would be to compute a new labeling independent of the old solution if the focus region is moved. In practice, however, one would like some *stability* of the labeling; that is, if a site is labeled in the first solution, it should preferably stay labeled even if the focus region is moved a bit—at least for a certain time span, so that the user can read the label. We suggest approaching this problem by modifying the weights of sites over time. For instance, if a site is labeled for the first time, its weight should be increased so that it preferably stays labeled in the next solution—unless the site leaves the focus region. After some time, some new sites should also get the opportunity to

be labeled; hence, we can decrease the weight again. The most difficult task is to evaluate the right weight function—over time—and to test the method in an interactive application.

Chapter 9

Many-to-One Boundary Labeling with Backbones

In the previous chapter, we have studied problems in the context of boundary labeling, where one wants to label sites in a focus region so that each point has its individual label that is to be placed at the boundary of the focus region. There are, however, also applications in which several sites must get the same label. For example, one may want to label sites by their class, for example, if different types of restaurants must be labeled. Instead of labeling each site of a class by an individual label and having several labels with the same text, we can also allow that multiple sites of the same class are connected to the same label. This problem setting is known as *many-to-one boundary labeling*, meaning that several sites can be connected to the same label via leaders.

More specifically, we study *many-to-one boundary labeling with backbone leaders*. In this new many-to-one model, a horizontal backbone reaches out of each label into the (rectangular) focus region. Sites that need to be connected to this label are linked via vertical line segments to the backbone; see Figure 9.1. We present dynamic programming algorithms for minimizing the total number of label occurrences and for minimizing the total leader length of crossing-free backbone labelings. When crossings are allowed, we aim at obtaining solutions with the minimum number of crossings. This can be achieved efficiently in the case of fixed label order; however, in the case of flexible label order we show that minimizing the number of leader crossings is NP-hard.

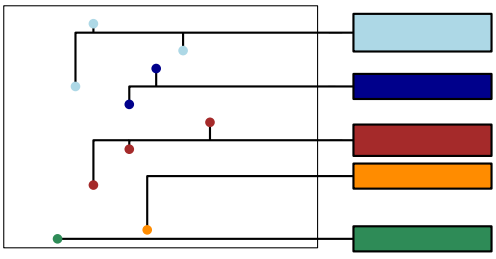
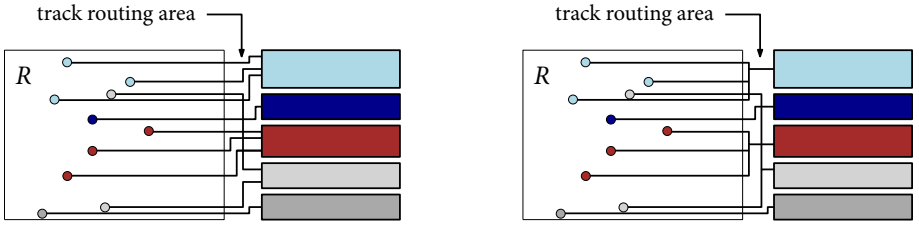


Figure 9.1: A crossing-free many-to-one boundary labeling with backbone leaders.



(a) Individual leaders [LKY08]. (b) Hyperleaders [Lin10].

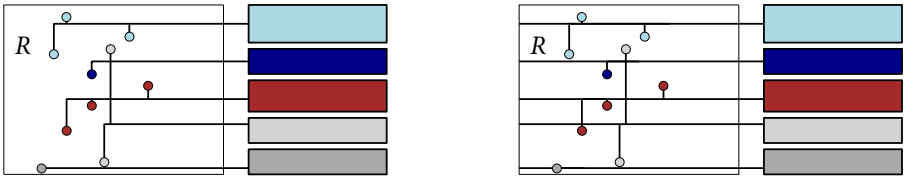
Figure 9.2: Different types of many-to-one labelings.

9.1 Introduction

So far, most work on boundary labeling has been devoted to the case where each label is associated with a single site; see the description in Section 8.1. However, the case where each label is associated with more than one site (the topic of this chapter) is also common in applications. We can think of groups of sites sharing common properties (for example, identical components of technical devices or locations of plants/animals of the same species in a map), which we express as having the same color. Then, we need to connect these identically colored sites via leaders to a label of the same color.

Previous Work. *Many-to-one boundary labeling* was formally introduced by Lin et al. [LKY08]. In their initial definition of many-to-one labeling each label had one port for each connecting site, that is, each point uses an individual leader (see Figure 9.2a). This inevitably lead to (i) tall labels, (ii) a wide track-routing area between the labels and the enclosing rectangle (since leaders are not allowed to overlap), and (iii) leader crossings in the track routing area. Lin et al. [LKY08] examined one and two-sided boundary labeling using so-called opo-leaders; see Fig. 9.2a. They showed that several crossing minimization problems are NP-complete and, subsequently, developed approximation and heuristic algorithms. In a variant of this model, referred to as *boundary labeling with hyperleaders*, Lin [Lin10] resolved the multiple port issue by joining together all leaders attached to a common label with a vertical line segment in the track-routing area; see Figure 9.2b. At the cost of label duplications, leader crossings could be eliminated.

Our Contribution. We study *many-to-one boundary labeling with backbone leaders* (for short, *backbone labeling*). In this new model, a horizontal backbone reaches out of each label into the site-enclosing rectangle. Sites connected to a label are linked via vertical line segments to the label's backbone (see Figure 9.3a). The backbone model does not need a track routing area and thus overcomes several disadvantages of previous many-to-one labeling models, in particular the issues (ii) and (iii) mentioned above. As Figure 9.3 shows, backbone labelings also require much less "ink" in the image than the previous methods and thus are expected to be less disturbing for the viewer. We note that backbone labeling can be seen as a variation of Lin's opo-hyperleaders. Lin [Lin10] posed it as an open problem to study *po*-hyperleaders (which is



(a) Finite Backbones. (b) Infinite Backbones.

Figure 9.3: Different types of many-to-one labelings with backbone leaders.

his terminology for backbones), in particular to minimize the number of duplicate labels in a crossing-free labeling.

We study three aspects of backbone labeling, *label number minimization* (Section 9.2), *total leader length minimization* (Section 9.3), and *crossing minimization* (Section 9.4). The first two aspects require crossing-free leaders. We consider both *finite backbones* (see Figure 9.3a) and *infinite backbones* (see Figure 9.3b). Finite backbones extend horizontally from the label to the furthest point connected to the backbone, whereas infinite backbones span the whole width of the rectangle (thus one could use duplicate labels on both sides). Furthermore, our algorithms vary depending on whether the order of the labels is fixed or flexible and whether more than one label per color class can be used.

For crossing-free backbone labeling we derive efficient algorithms based on dynamic programming to minimize label number and total leader length (Sections 9.2 and 9.3), which solves the open problem of Lin [Lin10]. The main idea is that backbones can be used to split an instance into two independent subinstances. For infinite leaders faster algorithms are possible since each backbone generates two independent instances; for finite backbones the algorithms require more effort since a backbone does not split the whole point set and thus the outermost point connected to each backbone must be considered. For the case where crossings are allowed, we present an efficient algorithm for crossing minimization with fixed label order and show NP-completeness for flexible label order (Section 9.4).

Problem Definition. Before we start investigating the problem variants, we properly define the notation used in this chapter.

In backbone labeling, we are given a set P of n points in an axis-aligned rectangle R, where each point $p \in P$ is assigned a color $c(p)$ from a color set C. Our goal is to place colored labels on the boundary of R and to assign each point $p \in P$ to a label $l(p)$ of color $c(p)$.

All points assigned to the same label will be connected to the label through a single backbone leader. A *backbone leader* consists of a horizontal *backbone* attached to the left or right side of the enclosing rectangle R and vertical line segments that connect the points to the backbone.

Only a single backbone leader can be attached to a label. Hence, we can use the terms *label* and *backbone* interchangeably. Since the backbones are horizontal, we consider labels to be fully described by the y-coordinate of their backbone. Note that, at first sight, this may imply that labels are of infinitely small height. However, by imposing a minimum separation distance between backbones, we can also accommodate labels of finite height.

Let $\mathcal{L}$ be a set of colored labels and consider label $l \in \mathcal{L}$. By $c(l)$, $y(l)$, and $P(l)$ we denote the color of label l, the y-coordinate of the backbone of label l on the boundary of R and the set of points that are connected/associated to label l, respectively.

A *backbone (boundary) labeling* for a set of colored points P in a rectangle R is a set $\mathcal{L}$ of colored labels together with a mapping of each point $p \in P$ to some $c(p)$-colored label in $\mathcal{L}$. The drawing can be easily produced since the backbone leader for label l is fully specified by $y(l)$ and $P(l)$. A backbone labeling is called *legal* if and only if (i) each point is connected to a label of the same color, and (ii) there are no backbone leader overlaps (though crossings are allowed in some cases).

Several restrictions on the number of labels of a specific color may be imposed: The number of labels may be unlimited, effectively allowing us to assign each point to a distinct label. Alternatively, the number of labels may be bounded by $K \geq |C|$. If $K = |C|$, all points of the same color have to be assigned to a single label. We may also restrict the maximum number of allowed labels for each color in C separately by specifying a *color vector* $\vec{k} = (k_1, \ldots, k_{|C|})$. A legal backbone labeling that satisfies all of the imposed restrictions on the number of labels is called *feasible*. Our goal in this chapter is to find feasible backbone labelings that optimize different quality criteria.

A backbone labeling without leader crossings is called *crossing-free*. An interesting variation of backbone labeling concerns the size of the backbone. A *finite backbone* attached to a label at, say, the right side of R extends up to the leftmost point that is assigned to it. An *infinite backbone* spans the whole width of R; see Figure 9.3 for examples of both types of backbones. Note that, in the case of crossing-free labelings, infinite backbones may result in labelings with a larger number of labels and increased total leader length.

In the remaining part of the chapter, we denote the points of P as $\{p_1, p_2, \ldots, p_n\}$ and we assume that no two points share the same x- or y-coordinate. For simplicity, we consider the points to be sorted in decreasing order of y-coordinates, with p_1 being the topmost point in all of our relevant drawings.

9.2 Minimizing the Total Number of Labels

In this section we study the problem of finding a many-to-one boundary labeling that minimizes the total number of labels. If we allow crossings, we can, of course, always find a labeling with just one label per color. In this case, the problem of minimizing the number of crossings arises, which will be covered in Section 9.4.

In this section, we will insist on solutions without crossings. Hence, we try to minimize the total number of labels in a crossing-free solution. We can, therefore, set $K = n$ so that there is effectively no upper bound on the number of labels.

9.2.1 Infinite Backbones

We first investigate the case of infinite backbones. As, in this setting, a backbone cuts the whole instance into two parts, it is clear that the points enclosed by two consecutive backbones can only have the colors of these backbones. Similarly, we can make an important observation on the structure of crossing-free labelings between two consecutive points; see the following lemma.

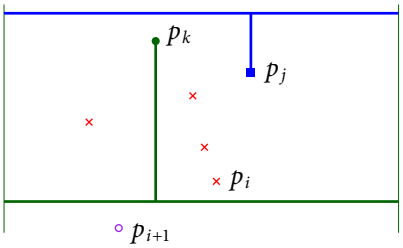


Figure 9.4: Point p_i cannot be labeled.

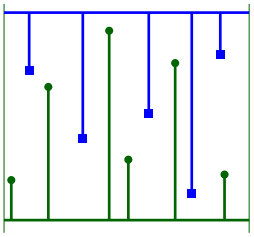


Figure 9.5: Labeling a 2-colored instance with one backbone per color.

Lemma 9.1. *Let p_i and p_{i+1} be two points that are vertically consecutive. Let p_j (with $j < i$) be the first point above p_i with $c(p_j) \neq c(p_i)$, and let $p_{j'}$ (with $j' > i+1$) be the first point below p_{i+1} with $c(p_{j'}) \neq c(p_{i+1})$ if such points exist.*

In any crossing-free backbone labeling p_i and p_{i+1} are vertically separated by at most two backbones. Furthermore, any separating backbone has color $c(p_i)$, $c(p_{i+1})$, $c(p_j)$, or $c(p_{j'})$.

Proof. Suppose that there are three separating backbones. Then the middle one could not be connected to any point. Now, suppose a separating backbone is connected to a point p_k above p_i and has color $c(p_k) \notin \{c(p_j), c(p_i)\}$. Then $k < j < i$. The backbone for p_j has to be above p_k. Hence, point p_i is lying between two backbones of other colors; see Figure 9.4. Its own backbone cannot be placed there without crossing a vertical segment connecting p_k or p_j to their corresponding backbone. Symmetrically, we see that a backbone separating p_i and p_{i+1} that is connected to a point below p_{i+1} can only have color $c(p_{i+1})$ or $c(p_{j'})$. □

Clearly, if all points have the same color, one label always suffices. Even in an instance with two colors, one label per color is enough: We place the backbone of one color above all points, and the backbone of the second color below all points; see Figure 9.5. However, if a third color is involved, then many labels may be required.

We denote the number of labels of an optimal crossing-free solution of P by $\mathrm{NL}(P)$. In the general case of the problem, P may contain several consecutive points of the same color. We proceed by constructing a simplified instance $C(P)$ based on the instance P; in $C(P)$, there are no two consecutive points of the same color. To do so, we identify each maximal set of identically-colored consecutive points of P and we replace all points of such a set by a single point of the same color that lies in the position of the topmost point of the set. Note that in order to achieve this, a simple top-to-bottom sweep is enough. Let $C(P) = \{p'_1, p'_2, \ldots, p'_k\}$ be the *clustered point set*, that we just constructed. For the sake of simplicity, we assume that $f: P \to C(P)$ is a function that computes the representative for a point of P in the simplified instance $C(P)$.

Lemma 9.2. *The number of labels needed in an optimal crossing-free labeling of P with infinite backbones is equal to the number of labels needed in an optimal crossing-free solution of $C(P)$, that is, $\mathrm{NL}(P) = \mathrm{NL}(C(P))$.*

Proof. Since $C(P) \subseteq P$, it trivially follows that $\mathrm{NL}(C(P)) \leq \mathrm{NL}(P)$. So, in order to complete the proof it remains to show that $\mathrm{NL}(P) \leq \mathrm{NL}(C(P))$. Let $S(C(P))$ be an optimal solution

of $C(P)$ with $\mathrm{NL}(C(P))$ labels. If we manage to construct a solution of P that has exactly the same number of labels as the optimal solution of $C(P)$, then obviously $\mathrm{NL}(P) \leq \mathrm{NL}(C(P))$.

Let p'_i with $1 \leq i \leq k$, be an arbitrary point of $C(P)$ and let $\{p_j, p_{j+1}, \ldots, p_{j+m}\}$ be the maximal set of consecutive, identically-colored points of P that has p'_i as its representative in $C(P)$. Let $S(p'_i)$ be the horizontal strip that is defined by the two horizontal lines through p_j and through p_{j+m}, respectively. Clearly, in a legal solution for P, $S(p'_i)$ can accommodate at most one backbone, namely the one of $\{p_j, p_{j+1}, \ldots, p_{j+m}\}$, as we look for crossing-free solutions. Now, observe that $S(p'_1), S(p'_2), \ldots, S(p'_k)$ do not overlap with each other, since we have assumed that our point set P is in general position, and, subsequently, all maximal sets of consecutive, identically-colored points of P are well separated. We proceed to derive a first solution $S(P)$ of P from $S(C(P))$ as follows: We connect each point p_i to the backbone of its representative $f(p_i)$ in $S(C(P))$. Clearly, $S(P)$ is not necessarily crossing-free. However, all potential crossings should appear in horizontal strips $S(p'_1), S(p'_2), \ldots, S(p'_k)$; otherwise $S(C(P))$ is not crossing-free as well.

Let $S(p'_i)$ with $1 \leq i \leq k$, be a horizontal strip that contains crossings. As already stated, $S(p'_i)$ can accommodate at most one backbone, namely the one of $\{p_j, p_{j+1}, \ldots, p_{j+m}\}$. We proceed to move all backbones in $S(p'_i)$ that are above (below, respectively) the one of $\{p_j, p_{j+1}, \ldots, p_{j+m}\}$ to the top of (below, respectively) $S(p'_i)$, without changing their relative order and without influencing the strips above and below $S(p'_i)$; recall that $S(p'_1), S(p'_2), \ldots, S(p'_k)$ do not overlap with each other, which implies that this is always possible. From the above it follows that the constructed solution is crossing-free and has the same number of labels as the one of $C(P)$, which completes the proof. □

With the help of the previous lemmas, we are now ready to present a linear-time algorithm for minimizing the number of infinite backbones.

Theorem 9.1. *Let $P = \{p_1, p_2, \ldots, p_n\}$ be an input point set consisting of n points sorted from top to bottom. Then, a crossing-free labeling of P with the minimum number of infinite backbones can be computed in $O(n)$ time.*

Proof. In order to simplify the proof, we assume that no two consecutive points have the same color, with the help of Lemma 9.2. If this is not already the case, we can first replace P by the simplified instance $C(P)$. After finding a solution for the simplified instance, we can transform this solution into a solution for P as we did in the proof of Lemma 9.2. Note that both transformations can be done in $O(n)$ time.

We will use dynamic programming on simplified instances. For $i = 1, 2, \ldots, n$, colors $\{c_{\mathrm{bak}}, c_{\mathrm{free}}\} \subseteq C$, and $\mathrm{cur} \in \{\texttt{true}, \texttt{false}\}$, let $\mathrm{nl}[i, \mathrm{cur}, c_{\mathrm{bak}}, c_{\mathrm{free}}]$ be the optimal number of backbones above or at p_i in the case where:

- The lowest backbone has color c_{bak}.
- If $\mathrm{cur} = \texttt{true}$, the lowest backbone coincides with p_i; hence, it is $c(p_i)$-colored, that is, $c_{\mathrm{bak}} = c(p_i)$. Otherwise the lowest backbone is above p_i. Note that in the latter case p_i might be unlabeled (for instance if the color of the lowest backbone is not $c(p_i)$, that is, $c_{\mathrm{bak}} \neq c(p_i)$).
- The point that, by Lemma 9.1, may exist between p_i and the lowest backbone has color c_{free}. Obviously, in the case where $\mathrm{cur} = \texttt{true}$ (that is, the lowest backbone coincides with p_i)

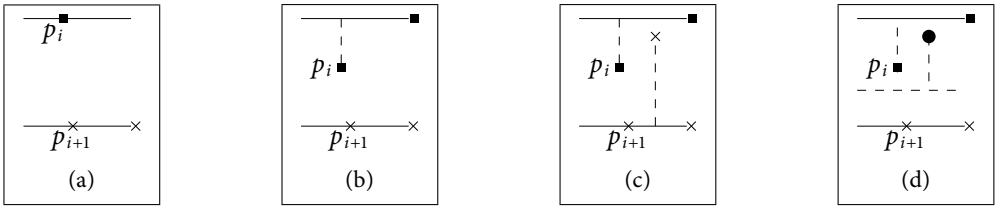


Figure 9.6: Different configurations that arise in case 1.1 of the proof of Theorem 9.1.

this point does not exist. So, in general, if this point does not exists, we assume that $c_{\text{free}} = \varnothing$.

Obviously, $\text{nl}[1, \texttt{true}, c(p_1), \varnothing] = 1$ and $\text{nl}[1, \texttt{false}, \varnothing, c(p_1)] = 0$. Now assume that we have computed all entries of table nl that correspond to different labelings induced by the point p_i. In order to compute the corresponding table entries for the next point p_{i+1}, we distinguish two cases:

1. *The lowest backbone coincides with p_{i+1}:* In this case, the lowest backbone should be $c(p_{i+1})$-colored, cur = `true`, and obviously there is no unlabeled point between the backbone through p_{i+1} and the point p_{i+1}, that is, $c_{\text{free}} = \varnothing$. Hence, we need to compute entry $\text{nl}[i+1, \texttt{true}, c(p_{i+1}), \varnothing]$. To do so, we distinguish the following subcases with respect to the color of the lowest backbone b above or at point p_i.

   1.1 *b is above or at point p_i and $c(p_i)$-colored.* If b is at point p_i (see Figure 9.6a), then trivially there is no unlabeled point below it. Hence, a feasible solution can be derived from $\text{nl}[i, \texttt{true}, c(p_i), \varnothing]$ by adding a new backbone, namely the one incident to p_{i+1}.

   If b is above point p_i, then we distinguish two subcases.

   (a) If there is no unlabeled point below b (see Figure 9.6b), then a feasible solution can, again, be derived from $\text{nl}[i, \texttt{false}, c(p_i), \varnothing]$ by adding a new backbone, namely the one incident to p_{i+1}.

   (b) On the other hand, if there is an unlabeled point below b, then we need to distinguish two subcases based on the color of this point.

   (b.1) If the unlabeled point is colored $c(p_{i+1})$ (see Figure 9.6c), then a single additional backbone incident to p_{i+1} suffices. The corresponding solution is derived from $\text{nl}[i, \texttt{false}, c(p_i), c(p_{i+1})]$.

   (b.2) However, in the case where the unlabeled point is c-colored and $c \notin \{c(p_i), c(p_{i+1})\}$ (see Figure 9.6d), two backbones are required and the corresponding feasible solution is derived from $\text{nl}[i, \texttt{false}, c(p_i), c]$ with $c \notin \{c(p_i), c(p_{i+1})\}$. Note that the case where the unlabeled point below b is of color $c(p_i)$ cannot occur, since we have assumed that consecutive points are not of the same color.

   1.2 *b is above p_i and $c(p_{i+1})$-colored.* Again, we distinguish two subcases.

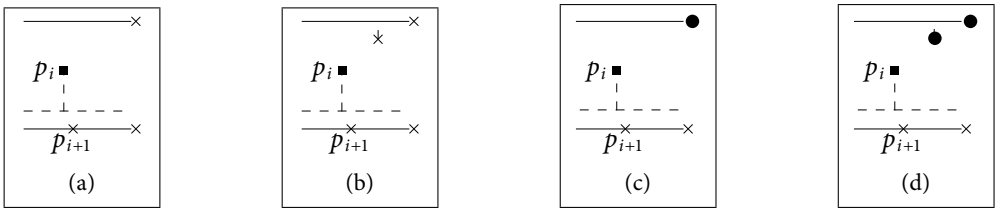


Figure 9.7: Different configurations that arise in cases 1.2 and 1.3 of the proof of Theorem 9.1.

(a) If there is no unlabeled point below b (see Figure 9.7a), then a feasible solution can be derived from $\mathrm{nl}[i, \texttt{false}, c(p_{i+1}), \varnothing]$ by adding two new backbones, that is, the one incident to p_i and the one incident to p_{i+1}.

(b) If there is an unlabeled point below b (see Figure 9.7b), then its color should be $c(p_{i+1})$. If this is not the case, it is easy to see that the backbone above p_i is not $c(p_{i+1})$-colored. Again two backbones are required, that is, the one incident to p_i and the one incident to p_{i+1}. The corresponding solution is derived from $\mathrm{nl}[i, \texttt{false}, c(p_{i+1}), c(p_{i+1})]$.

1.3 *b is above p_i and c-colored, where $c \neq c(p_i)$ and $c \neq c(p_{i+1})$.* In this case, either there is no unlabeled point below b (see Figure 9.7c) or there is one which is c-colored (see Figure 9.7d). In both cases, two backbones have to be placed: one incident to p_i and one incident to p_{i+1}. In the former case, the corresponding feasible solution is derived from $\mathrm{nl}[i, \texttt{false}, c, \varnothing]$ with $c \notin \{c(p_i), c(p_{i+1})\}$, while in the latter it is derived from $\mathrm{nl}[i, \texttt{false}, c, c]$ with, $c \notin \{c(p_i), c(p_{i+1})\}$.

From the above cases, it follows:

$$\mathrm{nl}[i+1, \texttt{true}, c(p_{i+1}), \varnothing] = \min \begin{cases} \mathrm{nl}[i, \texttt{true}, c(p_i), \varnothing] + 1 \\ \mathrm{nl}[i, \texttt{false}, c(p_i), \varnothing] + 1 \\ \mathrm{nl}[i, \texttt{false}, c(p_i), c(p_{i+1})] + 1 \\ \mathrm{nl}[i, \texttt{false}, c(p_i), c] + 2, \ c \notin \{c(p_i), c(p_{i+1})\} \\ \mathrm{nl}[i, \texttt{false}, c(p_{i+1}], \varnothing) + 2 \\ \mathrm{nl}[i, \texttt{false}, c(p_{i+1}], c(p_{i+1})) + 2 \\ \mathrm{nl}[i, \texttt{false}, c, \varnothing] + 2, \ c \notin \{c(p_i), c(p_{i+1})\} \\ \mathrm{nl}[i, \texttt{false}, c, c] + 2, \ c \notin \{c(p_i), c(p_{i+1})\} \end{cases}$$

2. *The lowest backbone is above p_{i+1}:* Again, we distinguish subcases with respect to the color of the lowest backbone b above or at point p_i:

2.1 *b is above or at point p_i and $c(p_i)$-colored.* If b is at point p_i (see Figure 9.8a) or b is above point p_i and either there is no unlabeled point below b (see Figure 9.8b) or

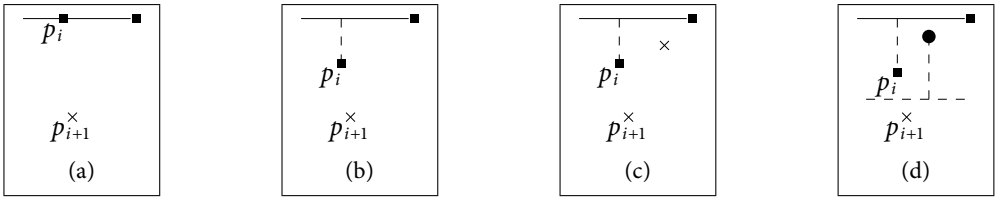


Figure 9.8: Different configurations that arise in case 2.1 of the proof of Theorem 9.1.

the unlabeled point below it is $c(p_{i+1})$-colored (see Figure 9.8c), then no additional backbone is required. Then, the corresponding feasible solutions are as follows:

$$\begin{aligned}\mathrm{nl}[i+1, \texttt{false}, c(p_i), \varnothing] = \min\{\, &\mathrm{nl}[i, \texttt{true}, c(p_i), \varnothing], \\ &\mathrm{nl}[i, \texttt{false}, c(p_i), \varnothing]\}\end{aligned}$$

$$\mathrm{nl}[i+1, \texttt{false}, c(p_i), c(p_{i+1})] = \mathrm{nl}[i, \texttt{false}, c(p_i), c(p_{i+1})]$$

However, in the case where the unlabeled point below b is c-colored, where $c \neq c(p_i)$ and $c \neq c(p_{i+1})$, a new backbone is required (see Figure 9.8d). Hence, the corresponding feasible solution can be derived as

$$\mathrm{nl}[i+1, \texttt{false}, c, \varnothing] = \mathrm{nl}[i, \texttt{false}, c(p_i), c] + 1 \text{ for } c \notin \{c(p_i), c(p_{i+1})\}.$$

2.2 *b is above p_i and $c(p_{i+1})$-colored.* In this case, either there is no unlabeled point below b (see Figure 9.9a) or there is one which is $c(p_{i+1})$-colored (see Figure 9.9b). In both cases no backbone is required. Hence, the corresponding feasible solutions can be derived as follows:

$$\mathrm{nl}[i+1, \texttt{false}, c(p_{i+1}), \varnothing] = \mathrm{nl}[i, \texttt{false}, c(p_{i+1}), \varnothing]$$

$$\mathrm{nl}[i+1, \texttt{false}, c(p_{i+1}), c(p_{i+1})] = \mathrm{nl}[i, \texttt{false}, c(p_{i+1}), c(p_{i+1})]$$

2.3 *b is above p_i and c-colored, where $c \neq c(p_i)$ and $c \neq c(p_{i+1})$.* In this case, if there is no unlabeled point below b (see Figure 9.9c) or there is one which is c-colored (see Figure 9.9d), then one backbone is required for p_i. The corresponding feasible solution can be derived as follows:

$$\begin{aligned}&\mathrm{nl}[i+1, \texttt{false}, c(p_i), \varnothing] \\ &= \min\{\, \mathrm{nl}[i, \texttt{false}, c, \varnothing] + 1, \mathrm{nl}[i, \texttt{false}, c, c]\} + 1 \\ &\qquad \text{with } c \notin \{c(p_i), c(p_{i+1})\}\end{aligned}$$

The most interesting case of our case analysis arises when a forth color is involved, say $c' \notin \{c(p_i), c(p_{i+1}), c\}$. In this case, either the c'-colored point remains unlabeled and p_i is labeled (see Figure 9.9e), or, the c'-colored point is labeled and p_i remains unlabeled (see Figure 9.9f). The corresponding feasible solutions can be described as follows.

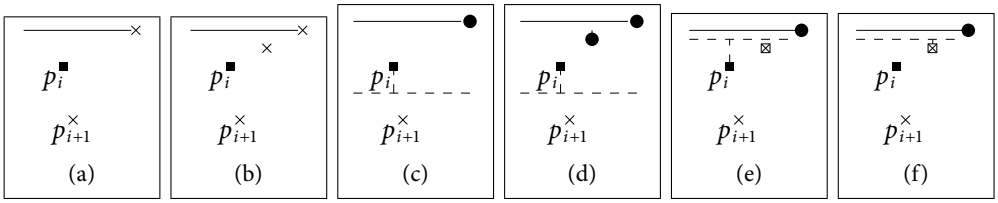


Figure 9.9: Different configurations that arise in cases 2.2 and 2.3 of the proof of Theorem 9.1.

$$\begin{aligned}&\mathrm{nl}[i+1, \texttt{false}, c(p_i), c'] \\ &\quad = \mathrm{nl}[i, \texttt{false}, c, c'] + 1,\ c \notin \{c(p_i), c(p_{i+1})\},\ c' \notin \{c(p_i), c(p_{i+1}), c\}\end{aligned}$$

$$\begin{aligned}&\mathrm{nl}[i+1, \texttt{false}, c', c(p_i)] \\ &\quad = \mathrm{nl}[i, \texttt{false}, c, c'] + 1,\ c \notin \{c(p_i), c(p_{i+1})\},\ c' \notin \{c(p_i), c(p_{i+1}), c\}\end{aligned}$$

Having computed table nl, the number of labels of the optimal solution of P equals the minimum entry of the form $\mathrm{nl}[n, \texttt{false}, \cdot, \varnothing]$. Since the algorithm maintains an $n \times 2 \times |C| \times |C|$ table and each entry is computed in constant time, the time complexity of our algorithm is $O(n|C|^2)$. However, with the help of Lemma 9.2, it can be reduced to $O(n)$ (since there is a constant number of possible colors that surround each point). A solution with the minimum number of labels can be found by backtracking in the dynamic program. □

9.2.2 Finite Backbones

We now consider the problem of minimizing the total number of labels for finite backbones. First, note that we can always slightly shift the backbones in a given solution so that backbones are placed only in gaps between points. We number the gaps from 0 to n where gap 0 is above point p_1, gap n is below point p_n, and gap i is between point p_i and point p_{i+1} for $1 \le i < n$.

Suppose that a point p_ℓ lies between a backbone of color c in gap g and a backbone of color c' in gap g' with $0 \le g < \ell \le g' \le n$ such that both backbones horizontally extend to at least the x-coordinate of p_ℓ; see Figure 9.10. Suppose that all points except the ones in the rectangle $R(g, g', \ell)$, spanned by the gaps g and g' and limited by p_ℓ to the left and by the boundary to the right, are already labeled. An optimum solution for connecting the points in $R(g, g', \ell)$ cannot reuse any backbone except for the two backbones in gaps g and g'; hence, such a partial solution is independent of the rest of the solution.

We use this observation for minimizing the number of backbones by a dynamic program. For $0 \le g \le g' \le n$, $\ell \in \{g+1, \ldots, g'\} \cup \{\varnothing\}$, and two colors c and c' let $T[g, c, g', c', \ell]$ be the minimum number of additional labels that are needed for labeling all points in the rectangle $R(g, g', \ell)$ under the assumption that there is a backbone of color c in gap g, a backbone of color c' in gap g', between these two backbones there is no backbone placed yet, and both backbones extend to the left of p_ℓ as in Figure 9.10. Note that for $\ell = \varnothing$ the rectangle is empty and $T[g, c, g', c', \varnothing] = 0$. Furthermore, also the case $g' = g$ can occur; in this case, as there is no point inside a gap, the relevant entry of table T is $T[g, c, g, c', \varnothing] = 0$.

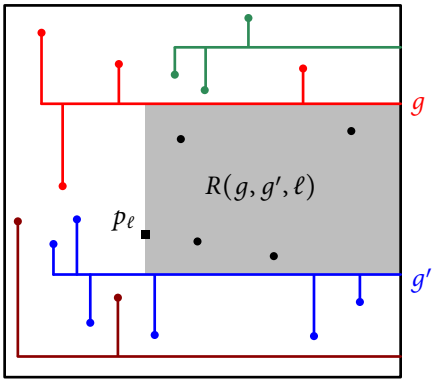


Figure 9.10: A partial instance in the rectangle $R(g, g', \ell)$ bounded by the two backbones in gaps g and g' and the leftmost point p_ℓ.

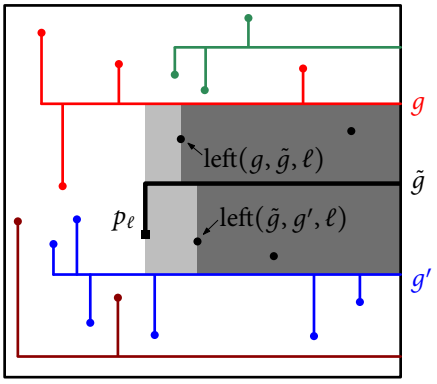


Figure 9.11: The partial instance in the rectangle $R(g, g', \ell)$ is split by a new backbone for p_ℓ into two partial instances in rectangles $R(g, \tilde{g}, \text{left}(g, \tilde{g}, \ell))$ and $R(\tilde{g}, g', \text{left}(\tilde{g}, g', \ell))$.

We distinguish cases based on the connection of point p_ℓ. First, if $c(p_\ell) = c$ or $c(p_\ell) = c'$, it is always optimal to connect p_ℓ to the top or bottom backbone, respectively, as all remaining points will be to the right of the new vertical segment. Hence, in this case,

$$T[g, c, g', c', \ell] = T[g, c, g', c', \text{left}(g, g', \ell)],$$

where $\text{left}(g, g', \ell)$ is the index of the leftmost point in the interior of $R(g, g', \ell)$ or $\text{left}(g, g', \ell) = \varnothing$ if no such point exists.

Otherwise, suppose that $c(p_\ell) \notin \{c, c'\}$. For connecting p_ℓ we need to place a new backbone of color $c(p_\ell)$; this is possible in any gap $\tilde{g}$ with $g \le \tilde{g} \le g'$. Note that reusing gap g or g' is allowed. The backbone splits the instance into two parts, one between gaps g and $\tilde{g}$ and one between gaps $\tilde{g}$ and g'; see Figure 9.11. Hence, we obtain the recursion

$$T[g, c, g', c', \ell] = \min_{g \le \tilde{g} \le g'} \Big(T[g, c, \tilde{g}, c(p_\ell), \text{left}(g, \tilde{g}, \ell)] + T[\tilde{g}, c(p_\ell), g', c', \text{left}(\tilde{g}, g', \ell)] \Big) + 1.$$

Finally, let $\bar{c} \notin C$ be a dummy color, and let $p_{\bar{\ell}} \in P$ be the leftmost point. Then the value $T[0, \bar{c}, n, \bar{c}, \bar{\ell}]$ obtained by using dummy backbones above and below all points yields the minimum number of labels needed for labeling all points. We can compute each of the (n+1) $\times$ $|C| \times (n+1) \times |C| \times (n+1)$ entries of table T in $O(n)$ time. Note that all left$(\cdot, \cdot, \cdot)$-values can easily be precomputed in $O(n^3)$ total time by first sorting the points from left to right and then, for each pair of gaps g and g' with $g < g'$, sweeping once over the points $\{p_{g+1}, \ldots, p_g\}$ in this direction. Summing up, we get the following result.

Theorem 9.2. *Given a set P of n colored points and a color set C, we can compute a feasible labeling of P with the minimum number of finite backbones in $O(n^4|C|^2)$ time.*

Minimum Distances. Our algorithm might place many labels inside a gap, which can result in a solution with very small distances between backbones. In practice, we may want to ensure a minimum distance of Δ between backbones, and between a backbone and a point not connected to this backbone. To this end, in any gap, we insert as many candidate positions for backbones as possible (up to n). Now, instead of using gaps in table T, we use these candidate positions; a position must never be used twice. As there are $O(n^2)$ instead of $n+1$ candidate positions, the number of entries of the table increases by a factor of $O(n^2)$, and we now need $O(n^2)$ time for computing an entry. Hence, the total running time is now $O(n^7|C|^2)$.

9.3 Length Minimization

In the previous section, we have presented algorithms for finding backbone labelings with the minimum number of labels. However, even two labelings with the same number of labels can look quite differently. For making the leaders easy to follow, it is important that they have small length. Hence, the objective is that the total length of leader segments is minimum. In order to avoid that the number of labels is very large in solutions with this new objective, we allow to specify an upper bound for the number of labels as part of the input.

In this section we minimize the total length of all leaders in a crossing-free solution, either including or excluding the horizontal lengths of the backbones. We distinguish between a global bound K on the number of labels or a vector $\vec{k}$ of individual bounds per color.

9.3.1 Infinite Backbones

First, we care about labelings with infinite backbones. We use a parameter λ to distinguish between the two minimization goals, that is, we set $\lambda = 0$ if we want to minimize only the sum of the lengths of all vertical segments, and we set λ to be the width of the rectangle R if we also take the length of the backbones into account.

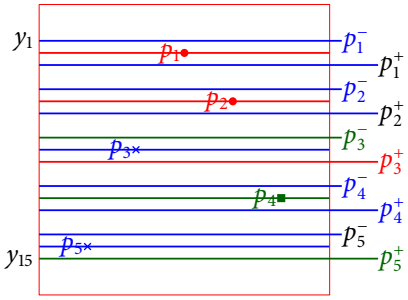


Figure 9.12: Candidates for five points. Red points are circles, blue points are crosses, and the green point is a square. Candidates through a point have the same color as the point. Candidate $p_i^{\pm}$ has the same color as the first point with a different color as p_i that is met when walking from $p_i^{\pm}$ over p_i. Candidates p_1^+, p_2^+, and p_5^- will not be used and have no color.

Single Color. As a first simple case, we assume that all points have the same color. In this case, we have to choose a set S of at most K y-coordinates where we draw the backbones and connect each point to its nearest backbone; this does, of course, not lead to crossings. Hence, we must solve the following problem: Given n points with y-coordinates $y(p_1) > \cdots > y(p_n)$, find a set S of at most K y-coordinates that minimizes

$$\lambda \cdot |S| + \sum_{i=1}^{n} \min_{y \in S} |y - p_i|. \tag{9.1}$$

We can optimize the value in Equation 9.1 by choosing $S \subseteq \{y(p_1), \ldots, y(p_n)\}$, that is, by selecting only backbones that pass through input points: For a backbone position $y \in S \setminus \{y(p_1), \ldots, y(p_n)\}$ let $\{p_i, \ldots, p_j\}$ be the set of points that we would connect to the backbone through y. Let $y(p_i) > \cdots > y(p_{i'}) > y > y(p_{i'+1}) > \cdots > y(p_j)$. If $i' - i + 1 \geq j - i'$, that is, if the majority of sites connected to the backbone at position y lies above the backbone, replace y by $y(p_{i'})$. Otherwise replace y by $y(p_{i'+1})$. Then the objective value in Equation 9.1 can at most improve. Hence, the problem can be solved in $O(Kn)$ time if the points are sorted according to their y-coordinates using the algorithm of Hassin and Tamir [HT91]. Note that the problem corresponds to the 1-dimensional K-median problem if $\lambda = 0$.

Multiple Colors. If the n points have different colors, we can no longer assume that all backbones go through one of the given n points since we have to avoid crossings. However, by Lemma 9.1, it suffices to add between any pair of vertically consecutive points two additional candidates for backbone positions, plus one additional candidate above all points and one below all points. Hence, we have a set of $3n$ candidate lines at y-coordinates

$$p_1^- > y(p_1) > p_1^+ > p_2^- > y(p_2) > p_2^+ > \cdots > p_n^- > y(p_n) > p_n^+ \tag{9.2}$$

where for each i the values p_i^- and p_i^+ are as close to $y(p_i)$ as the label heights allow. Clearly, a backbone through p_i can only be connected to points with color $c(p_i)$. If we use a backbone through p_i^- (or p_i^+, respectively), it will have the same color as the first point below p_i (or

above p_i, respectively) that has a different color than p_i; compare Lemma 9.1. For example, in Fig 9.12, p_1^- is colored blue, since p_3 is the first point below p_1 that has a different color than red, namely blue. Hence, the colors of all candidates are fixed or the candidate will never be used as a backbone. For an easier notation, we denote the y-coordinate of the ith point in Equation 9.2 by y_i and its color by $c(y_i)$.

We minimize the total leader length by using dynamic programming. For each $i = 1, \ldots, 3n$, and for each vector $\vec{k}' = (k'_1, \ldots, k'_{|C|})$ with $k'_1 \leq k_1, \ldots, k'_{|C|} \leq k_{|C|}$, let $L[i, \vec{k}']$ denote the minimum length of a feasible backbone labeling of $p_1, \ldots, p_{\lfloor \frac{i+1}{3} \rfloor}$ using k'_c infinite backbones of color c for $c = 1, \ldots, |C|$ such that the bottommost backbone is at position y_i, if such a labeling exists. Otherwise $L[i, \vec{k}'] = \infty$. In the following, we describe, how to compute the values $L[i, \vec{k}']$.

Assume that we want to place a new backbone at position y_i and that the previous backbone was at position y_j with $j < i$. Then, we have to connect each point p_x with $(j+2)/3 \leq x \leq i/3$ to one of the backbones through y_i or y_j as these points are enclosed between the two backbones. Let $\text{link}(j, i)$ denote the minimum total length of the vertical segments linking these points to their respective backbone. We set $\text{link}(j, i) = \infty$ if there is a point p_x between y_i and y_j with $c(p_x) \notin \{c(y_i), c(y_j)\}$ because p_x cannot be connected to the surrounding backbones. Otherwise, we have

$$\text{link}(j,i) = \begin{cases} \sum\limits_{\frac{j+2}{3} \leq x \leq \frac{i}{3}} \min\big(y_j - y(p_x), y(p_x) - y_i\big) & \text{if } c(y_i) = c(y_j) \\ \sum\limits_{\substack{\frac{j+2}{3} \leq x \leq \frac{i}{3} \\ c(p_x) = c(y_j)}} (y_j - y(p_x)) + \sum\limits_{\substack{\frac{j+2}{3} \leq x \leq \frac{i}{3} \\ c(p_x) = c(y_i)}} (y(p_x) - y_i) & \text{if } c(y_i) \neq c(y_j) \end{cases} \tag{9.3}$$

The base cases are

$$L[i, 0, \ldots, 0, k'_{c(y_i)} = 1, 0, \ldots, 0] = \sum_{0 < x \leq i/3} (y_i - p_x)$$

if all points above y_i have the color $c(y_i)$ and $L[i, 0, \ldots, 0, k'_{c(y_i)} = 1, 0, \ldots, 0] = \infty$ otherwise, as well as $L[i, \vec{0}] = \infty$.

For computing an entry $L[i, k'_1, \ldots, k'_{|C|}]$ we test all candidate positions $y_j > y_i$ for the previous backbone; to the length of the corresponding solution we have to add the connection cost $\text{link}(j, i)$ as well as λ for the new backbone at position y_i. Hence, we get the following recursion:

$$L[i, k'_1, \ldots, k'_{|C|}] = \lambda + \min_{j \leq i} \Big(L[j, k'_1, \ldots, k'_{c(y_i)} - 1, \ldots, k'_{|C|}] + \text{link}(j, i) \Big) \tag{9.4}$$

Note that we need to interpret any entry of table L for which a color bound is negative as ∞.

In order to see that each entry of table L can be computed in $O(n)$ time, we have to show, that, for a fixed index i, all values $\text{link}(j, i)$ with $j < i$ can be computed in $O(n)$ time. Let c' be the first color of a point above y_i that is different from $c(y_i)$. For a fixed i, starting from $j = i - 1$, we scan the candidates twice in decreasing order of their indices until we find the first point that is neither colored c' nor $c(y_i)$.

For color $c \in \{c(y_i), c'\}$, we traverse the points above y_i from bottom to top. For any point $p_{x'}$ that we see, we store two values: the number $n_c(x')$ of points of color c that we have seen so far and the sum of distances of these c-colored points to y_i, that is,

$$l_c(x') = \sum_{x' \le x \le \frac{i}{3}, c(p_x)=c} (y(p_x) - y_i).$$

Note that we can easily compute $l_c(x'-1)$ in constant time from $l_c(x')$: If $p_{x'-1}$ is not c-colored, then $l_c(x'-1) = l_c(x')$; if $c(p_{x'-1}) = c$, then we have to connect the point $p_{x'-1}$ to y_i and, hence, $l_c(x'-1) = l_c(x') + (y(p_{x'-1}) - y_i)$.

With these values we can compute any value $\mathrm{link}(j, i)$ as follows. First, suppose that $c(y_i) \neq c(y_j)$ as in the second case of Equation 9.4. Let $p_{x'}$ be the point immediately below y_j. Then

$$\sum_{\substack{\frac{j+2}{3} \le x \le \frac{i}{3} \\ c(p_x)=c(y_i)}} (y(p_x) - y_i) = l_{c(y_i)}(x').$$

Furthermore, we can also compute the length needed for connecting the points of color $c(y_j)$ to the backbone at position y_j since we know their number $n_{c(y_j)}(p_{x'})$ and $y_j - y = (y_j - y_i) - (y - y_i)$ for $y_j \ge y \ge y_i$:

$$\begin{aligned}\sum_{\substack{\frac{j+2}{3} \le x \le \frac{i}{3} \\ c(p_x)=c(y_j)}} (y_j - y(p_x)) &= \sum_{\substack{x' \le x \le \frac{i}{3} \\ c(p_x)=c(y_j)}} \big((y_j - y_i) - (y(p_x) - y_i)\big) \\ &= n_{c(y_j)}(x') \cdot (y_j - y_i) - l_{c(y_j)}(x')\end{aligned}$$

Hence, we can compute all values $\mathrm{link}(j, i)$ with $\mathrm{c}(y_j) \neq \mathrm{c}(y_i)$ in $O(n)$ total time for fixed i.

Now, assume that $\mathrm{c}(y_i) = \mathrm{c}(y_j)$ and let again be $p_{x'}$ the point immediately below y_j. If $n_{c'}(x') > 0$ there is a point of color $c' \neq c(y_i)$ between the two backbones; as this point cannot be connected, $\mathrm{link}(j, i) = \infty$. If no such point exists, every point connects to the closer backbone, either y_j or y_i. Hence, the points are split into two subsets, where $p_{x''}$ is the topmost point that connects down to y_i and all points $p_{x'}, \ldots, p_{x''-1}$ connect to y_j. Similar to the previous computation, we get that

$$\begin{aligned}\mathrm{link}(j, i) &= \sum_{x' \le x \le \frac{i}{3}} \min\big(y_j - y(p_x), y(p_x) - y_i\big) \\ &= \sum_{x' \le x < x''} (y_j - y(p_x)) + \sum_{x'' \le x \le \frac{i}{3}} (y(p_x) - y_i) \\ &= \big(n_{c(y_i)}(x') - n_{c(y_i)}(x'')\big) \cdot (y_j - y_i) - \big(l_{c(y_i)}(x') - l_{c(y_i)}(x'')\big) + l_{c(y_i)}(x'').\end{aligned}$$

This can be computed in constant time. Note that by simply sweeping once over the backbone positions y_j and the points from y_i to the top in parallel, we can easily find the right x'' for each y_j in $O(n)$ total time.

We have now seen that we can compute all values $\mathrm{link}(\cdot, i)$ in $O(n)$ total time. As a consequence, we know that we can compute any entry of table L in $O(n)$ time. For computing all entries of the table, we need, hence, $O\big(n^2 \prod_{i=1}^{|C|} k_i\big)$ time.

Let S be the set of candidates y_i such that all points below y_i have the same color as y_i. Any solution with y_i as the lowest backbone is a candidate for the optimum solution; we do, however, have to consider the cost of connecting the points below y_i to the backbone through y_i. Note that $y_{3n-1} = y(p_n)$ and y_{3n} are always included in the set S. Summing up, we can compute the minimum total length of a backbone labeling of $p_1, \ldots, p_n$ with at most k_c labels per color $1 \leq c \leq |C|$ as

$$\min_{y_i \in S, k'_1 \leq k_1, \ldots, k'_{|C|} \leq k_{|C|}} \left(L[i, k'_1, \ldots, k'_{|C|}] + \sum_{\frac{i+2}{3} \leq x \leq n} (y_i - p_x) \right).$$

Hence, we get the following theorem.

Theorem 9.3. *A minimum length backbone labeling with infinite backbones for n points with $|C|$ colors can be computed in $O\left(n^2 \cdot \prod_{i=1}^{|C|} k_i\right)$ time if at most k_i labels are allowed for color $i \in C$.*

If we globally bound the total number of labels by K, we can use a similar dynamic program; in the table L, we replace the individual bounds k_c for color $c \in C$ with the global bound K, that is, we compute values $L[i, k]$ with $1 \leq i \leq 3n$ and $k \leq K$. The only difference in the dynamic program is, hence, that we always use the global bound instead of the specific bounds for colors. We get the following result.

Theorem 9.4. *A minimum length backbone labeling with infinite backbones for n points with $|C|$ colors can be computed in $O(n^2 K)$ time if up to K labels in total are allowed.*

Note that our dynamic program can also be used for deciding whether a feasible crossing-free solution subject to the bounds on the numbers of labels exists. If no feasible solution exists, the reported minimum length will be ∞.

9.3.2 Finite Backbones

We now turn to leader length minimization for labeling with finite backbones. Here, the length of a backbone segment may differ heavily; hence, we do not use a parameter λ as we did for infinite backbones in Section 9.3.1, but we always count both horizontal and vertical lengths. Recall that we solved the minimization of the number of backbones with the help of a dynamic program based on rectangular subinstances bounded by two backbones and a leftmost point; see Section 9.2.2. We modify this dynamic program for minimizing the total leader length.

As a first obvious change, we now denote by the T-values the additional length of segments and backbones needed for labeling the points of the subinstance. However, we have to adjust more details. By the case of a single point connected to a backbone, we see that we have to allow backbones passing through input points of the same color for length minimization. Additionally, for computing the vertical length needed for connecting to a backbone placed in a gap, we need to know its actual y-coordinate.

Suppose that there is a set B of backbones that all lie in the same gap between points p_i and p_{i+1}. Let $b^\star$ be the longest of these backbones; see Figure 9.13. The backbone $b^\star$ vertically splits the set B; any backbone $b' \in B$ above $b^\star$ can only connect to points above itself and any backbone $b'' \in B$ below $b^\star$ can only connect to points below itself. By moving b' to the top and b'' to the bottom as far as possible the total leader length decreases. Hence, in any optimum

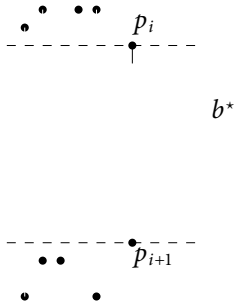


Figure 9.13: The longest backbone $b^\star$ splits the backbones between p_i and p_{i+1}.

solution, the backbones above $b^\star$ will be very close to $y(p_i)$ and the backbones below $b^\star$ will be very close to $y(p_{i+1})$. Furthermore, depending on the numbers of connected points above and below, by either moving $b^\star$ to the top or to the bottom we will find a solution that is not worse, and in which any backbone of B is close to p_i or p_{i+1}.

If we allow backbones to be infinitely close to points or other backbones, we can use backbone positions p_i^- and p_i^+ that lie infinitely close above and below p_i, respectively, and share its y-coordinate. Each of these positions may be used for an arbitrary number of backbones.

Now, in the case distinction, we have to be a bit more careful. When the leftmost point p_ℓ in the subinstance bounded by backbones of color c to the top and of color c' to the bottom, respectively, has color c or c', we can no longer alway connect p_ℓ to the existing backbones. Although such a connection is always possible, opening a new backbone may save leader length in this step or in later steps. Hence, we have to additionally test all positions for placing a new backbone in the same way as we do if p_ℓ has a different color. Note that this does not increase the runtime.

With the new positions as well as the input points as possible label positions and the updated case analysis, we can then find a solution with minimum total leader length in $O(n^4|C|^2)$ time, if the number of labels is not bounded, by adding the length of the newly placed segments in any calculation.

Bounded Numbers of Labels. If we want to integrate an upper bound K on the total number of labels, or, for each color $c \in C$, an upper bound k_c on the number of labels of color c, into the dynamic program—as we did for infinite backbones—, we need an additional dimension for the remaining number of backbones that we can use in the subinstance (or a dimension for each color $c \in C$ for the remaining number of backbones of that color); that is, we now use table entries of the form $T[i, c, i', c', \ell, K']$ (or $T[i, c, i', c', \ell, \vec{k}']$) where y_i and $y_{i'}$ with $i < i'$ are the positions of the upper and lower backbone, respectively, c and c' are their respective colors, p_ℓ with $(i+2)/3 \le \ell \le i'/3$ (or $\ell = \varnothing$) is the leftmost point of the subinstance (if such a point exists), and K' (or $\vec{k}'$) is the number of labels (per color) that we allow for the subinstance. Additionally, when splitting the instance into two parts, we have to consider not only the position of the splitting backbone of color $c(p_\ell)$, but also the different combinations of distributing the allowed numbers of backbones among the subinstances. For a global bound K, we need, hence, $O(nK)$ time for computing an entry of the table. If we have individual bounds k_c for

(a) Backbones placed with the minimum leader length.

(b) Candidate positions for backbones inside the gap.

Figure 9.14: Situation between two consecutive points for finite backbones.

$c \in C$, we need $O\left(n \prod_{c \in C} k_c\right)$ time. Together with the additional dimension(s) of the table, we can minimize the total leader length in $O\left(n^4 |C|^2 K^2\right)$ time if we have a global bound K, and in $O\left(n^4 |C|^2 \left(\prod_{c \in C} k_c\right)^2\right)$ time if we have an individual bound k_c for each color $c \in C$. Note that we can easily detect cases where we have to add a backbone of color $c(p_\ell)$ but the current bound $k_{c(p_\ell)} = 0$ (or $K = 0$) in the subinstance. In such a case, we report $+\infty$ as the total leader length, indicating that no feasible solution with the given bounds exists.

Minimum Distances. So far, we allowed backbones to be infinitely close to unconnected points and other backbones, which will, in practice, lead to overlaps, for instance between consecutive labels. One would rather enforce a small distance between two backbones or a backbone and a point, even if this increases the total leader length a bit. Let $\Delta > 0$ be the minimum allowed distance, which depends, for example, on the font size used for the labels. In an optimum solution, there will be two sequences of backbones on the top and on the bottom of a gap between p_i and p_{i+1}, such that inside a sequence consecutive backbones have distance Δ; see Figure 9.14a. We get all possible backbone positions inside the gap by taking all y-coordinates inside whose y-distance to either p_i or p_{i+1} is an integer multiple of Δ; see Figure 9.14b. Note that n positions of each type suffice in a gap; if the gap is too small, there might even be fewer positions. The two sequences can overlap. In this case, we have to check that we do not combine two positions with a distance smaller than Δ in the dynamic program.

Together with the input points, we get a set of $O(n^2)$ candidate positions for backbones, each of which can be used at most once. This increases the number of entries of table T by a factor of $O(n^2)$, and the running time of computing a single entry by a factor of $O(n)$. The resulting running time of our dynamic program is $O\left(n^7 |C|^2\right)$ if we do not bound the number of labels, $O\left(n^7 |C|^2 K^2\right)$ if we have a global bound K on the number of labels, and $O\left(n^7 |C|^2 \left(\prod_{c \in C} k_c\right)^2\right)$ if we have an individual bound k_c for each color $c \in C$.

Theorem 9.5. *Given a set P of n colored points, a color set C, and a label bound K (or vector $\vec{k}$ of bounds per color), we can compute a feasible labeling of P with finite backbones that minimizes*

the total leader length in $O\big(n^7|C|^2K^2\big)$ *time (or in* $O\Big(n^7|C|^2\big(\prod_{c\in C} k_c\big)^2\Big)$ *time).*

Note that, as in the case of infinite backbones, also for finite backbones we can use the dynamic program for deciding whether a feasible solution for the given bounds on the numbers of labels exists: If no such solution exists, the reported total leader length will be ∞.

9.4 Crossing Minimization

In this section we allow crossings between backbone leaders, which generally allows us to use fewer labels. More precisely, if crossings are allowed, it is trivially possible to label all points using just one label per color. Such a solution may, however, lead to many crossings between backbones and vertical leader segments. Therefore, we are interested in minimizing the number of such crossings. We concentrate on he case that $K = |C|$ labels, that is, one per color, are allowed. We will first consider the case that the relative order of labels for the colors from top to bottom is prescribed. For this case we will present efficient algorithms for minimizing the number of crossings. Then, we will see that without this restriction the problem becomes NP-hard, at least for finite backbones.

9.4.1 Fixed y-Order of Labels

We first assume that the color set C is ordered and we require that for each pair of colors $i < j$ the label of color i is above the label of color j. We will develop a fast algorithm for crossing minimization with infinite backbones. Then, we will show how this algorithm can be modified for the case of finite backbones.

Infinite Backbones

Since the order of the labels is fixed, the order in which the backbones appear from top-to-bottom should also be fixed. This implies that the i-th backbone in the given y-ordering from top to bottom is connected to the points of color i.

Observe that it is always possible to slightly shift the backbones of a solution without increasing the number of crossings such that no backbone contains a point. Thus, the backbones can be assumed to be positioned in the gaps between vertically adjacent points; we number the gaps from 0 to n as in Section 9.2.2.

Suppose that we fix the position of the i-th backbone to gap g. For $1 \le i \le |C|$ and $0 \le g \le n$, let $\mathrm{cross}(i, g)$ be the number of crossings of the vertical segments of the non-i-colored points when the color-i backbone is placed at gap g. Note that this number depends only on the y-ordering of the backbones, which is fixed, and not on their actual positions. So, we can precompute table cross, using dynamic programming, as follows.

All table entries of the form $\mathrm{cross}(\cdot, 0)$ can clearly be computed in $O(n|C|)$ total time because, for color i, $\mathrm{cross}(i, 0)$ is equal to number of points having some color $< i$. Then, $\mathrm{cross}(i, g) = \mathrm{cross}(i, g-1) + 1$, if the point p_g between gaps $g-1$ and g has color j with $j > i$. In the case where p_g has color j with $j < i$, $\mathrm{cross}(i, g) = \mathrm{cross}(i, g-1) - 1$. If p_g has color i, then $\mathrm{cross}(i, g) = \mathrm{cross}(i, g-1)$. From the above, it follows that the computation of the table cross takes $O(n|C|)$ time.

Now, we use another dynamic program for computing the minimum number of crossings. Let $T[i, g]$ denote the minimum number of crossings on the backbones $1, \ldots, i$ in any solution subject to the condition that the backbones are placed in the given ordering and backbone i is positioned in gap g. Clearly $T[0, g] = 0$ for $g = 0, \ldots, n$. For computing an entry $T[i, g]$ with $i > 0$, we test all positions for the previous backbone $i - 1$ in a gap g' above (and including) gap g. In addition to the number of crossings from the entry $T[i - 1, g']$, we also have to take the number $\text{cross}(i, g)$ of crossings of the new backbone into account. Hence, we have

$$T[i, g] = \text{cross}(i, g) + \min_{g' \leq g} T[i - 1, g'].$$

Having precomputed table cross and assuming that for each entry $T[i, g]$, we also store the smallest entry $T[i, g']$ with $g' \leq g$, each entry of table T can be computed in constant time. Hence, table T can be filled in time $O(n|C|)$. Then, the minimum crossing number is $\min_{0 \leq g \leq n} T[|C|, g]$. A corresponding solution can be found by backtracking in the dynamic program. Let us summarize.

Theorem 9.6. *Given a set P of n colored points and an ordered color set C, a backbone labeling with one label per color, labels in the given color order, infinite backbones, and minimum number of crossings can be computed in $O(n|C|)$ time.*

Finite Backbones

We can easily modify the approach used for infinite backbones for minimizing the number of crossings for finite backbones, if the y-order of labels is fixed, as the following theorem shows.

Theorem 9.7. *Given a set P of n colored points and an ordered color set C, a backbone labeling with one label per color, labels in the given order, finite backbones, and minimum number of crossings can be computed in $O(n|C|)$ time.*

Proof. We develop a dynamic program very similar to the one presented for infinite backbones. The only part that we have to change is that the computation of the number of crossings when fixing a backbone at a certain position should take into consideration that the backbones are not of infinite length. Recall that the dynamic program could precompute these crossings, by maintaining an $n \times |C|$ table cross, in which each entry $\text{cross}(i, g)$ corresponds to the number of crossings of the non-i-colored points when the color-i-backbone is placed at gap g, for $1 \leq i \leq |C|$ and $0 \leq g \leq n$. For finite backbones, $\text{cross}(i, g) = \text{cross}(i, g - 1) + 1$, if the point between gaps $g - 1$ and g is right of the leftmost i-colored point and has color j with $j > i$. In the case, where the point p_g between gaps $g - 1$ and g is right of the leftmost i-colored point and has color j with $j < i$, $\text{cross}(i, g) = \text{cross}(i, g - 1) - 1$. Otherwise, $\text{cross}(i, g) = \text{cross}(i, g - 1)$. Again, all table entries of the form $\text{cross}(\cdot, 0)$ can clearly be computed in $O(n)$ time. □

9.4.2 Flexible y-Order of Labels

We now no longer assume that the order of labels is prescribed, that is, we need to minimize the number of crossings over all label orders. While there is an efficient algorithm for a restricted variant of the problem with infinite backbones, the problem is NP-complete for finite backbones.

Infinite Backbones

We give an efficient algorithm for the case where there are $K = |C|$ fixed label positions $y_1, \ldots, y_K$ on the right boundary of R, for instance, uniformly distributed.

Theorem 9.8. *Given a set P of n colored points, a color set C, and a set of $|C|$ fixed label positions, we can compute a feasible backbone labeling with infinite backbones that minimizes the number of crossings in $O(n + |C|^3)$ time.*

Proof. First observe that if the backbone of color k with $1 \leq k \leq |C|$ is placed at position y_i with $1 \leq i \leq |C|$, then the number of crossings created by the vertical segments leading to this backbone is fixed, since all label positions will be occupied by an infinite backbone. Let n_k be the number of points of color k. The crossing number $\mathrm{cr}(k, i)$ can be determined in $O(n_k + |C|)$ time. In fact, by a sweep from top to bottom, we can even determine all crossing numbers $\mathrm{cr}(k, \cdot)$ for backbone k with $1 \leq k \leq |C|$ in $O(n_k + |C|)$ time. Now, we construct an instance of a weighted bipartite matching problem, where for each position y_i with $1 \leq k \leq |C|$ and each backbone k with $1 \leq k \leq |C|$, we establish an edge $\{k, i\}$ of weight $\mathrm{cr}(k, i)$. In total, this takes $O(n + |C|^2)$ time. The minimum-cost weighted bipartite matching problem can be computed in $O(|C|^3)$ time using the Hungarian method [Kuh55] and yields a backbone labeling with the minimal number of crossings. □

Note that the previous approach does not work for finite backbones. In contrast to infinite backbones a crossing of a vertical segment for some color with a backbone depends on the horizontal extend and, hence, on the color of this backbone. Therefore, it is not possible to calculate a simple number $\mathrm{cr}(k, i)$ of crossings for the placement of backbone k on position y_i.

Finite Backbones

Next, we consider the variant with finite backbones and prove that it is NP-hard to minimize the number of crossings. Here, we do not restrict ourselves to candidate positions for backbones. For simplicity, we allow points that share the same x- or y-coordinates. This can be remedied by a slight perturbation. Our arguments do not make use of this special situation and, hence, carry over to the perturbed constructions. We first introduce a number of gadgets that are required for our proof and explain their properties, before describing the hardness reduction.

The first gadget is the *range restrictor gadget*. Its construction consists of a middle backbone, whose position will be restricted to a given vertical range R, and an upper and a lower *guard gadget* that ensure that positioning the middle backbone outside range R creates many crossings; see Figure 9.15. We assume that the middle backbone is connected to at least one point further to the left such that it extends beyond all points of the guard gadgets. Additionally, the middle backbone is connected to two *range points* whose y-coordinates are the upper and lower boundary of the range R. Their x-coordinates are such that they are on the right of the points of the guard gadgets. A *guard* consists of a backbone that connects to a set of M points, where $M > 1$ is an arbitrary number. The M points of a guard lie left of the range points. The upper guard points are horizontally aligned and lie slightly below the upper bound of range R. The lower guard points are horizontally aligned and are placed such that they are slightly above the lower bound of range R. We place M upper and M lower guards such that the guards form pairs for

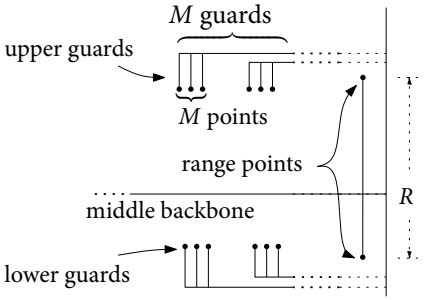


Figure 9.15: The range restrictor gadget.

which the guard points overlap horizontally. The upper (respectively lower) guard gadget is formed by the set of upper (respectively lower) guards. We call M the *size* of the guard gadgets. The next lemma shows the important properties of the range restrictor gadget.

Lemma 9.3. *The backbones of the range restrictor gadget can be positioned such that there are no crossings. If the middle backbone is positioned outside the range R, there are at least $M - 1$ crossings.*

Proof. The first statement is illustrated by the drawing in Figure 9.15. Suppose for a contradiction to the second statement that the middle backbone is positioned outside range R and that there are fewer than $M - 1$ crossings. Assume, without loss of generality, that the middle backbone is embedded below range R; the other case is symmetric.

First, observe that all backbones of guards must be positioned above the middle backbone, as a guard backbone below the middle backbone would create M crossings, namely between the middle backbone and the segments connecting the points of the guard to its backbone. Hence the middle backbone is the lowest. Now observe that any guard that is positioned below the upper range point crosses the segment that connects this range point to the middle backbone. To avoid having $M - 1$ crossings, it follows that at least $M + 1$ guards (both upper and lower) must be positioned above range R. Hence, there is at least one pair consisting of an upper and a lower guard that are both positioned above the range R. This, however, independent of their ordering, creates at least $M - 1$ crossings; see Figure 9.16, where the two alternatives for the lower guard are drawn in black and bold gray, respectively. This contradicts our assumption. □

Let B be an axis-aligned rectangular box and let R be a small interval that is contained in the range of y-coordinates spanned by B. A *blocker gadget* of *width* m consists of a backbone that connects to $2m$ points, half of which are positioned on the top and on the bottom side of B, respectively. Moreover, a range restrictor gadget is used to restrict the backbone of the blocker to the range R. Figure 9.17 shows an example. Note that, due to the range restrictor, this drawing is essentially fixed. We say that a backbone *crosses* the blocker gadget if its backbone crosses the box B. It is easy to see that any backbone that crosses a blocker gadget creates m crossings, where m is the width of the blocker.

We are now ready to show that the crossing minimization problem with flexible y-order of the labels is NP-complete.

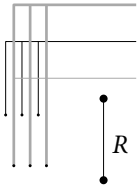


Figure 9.16: Crossings caused by a pair of an upper and a lower guard that are positioned on the same side outside range R.

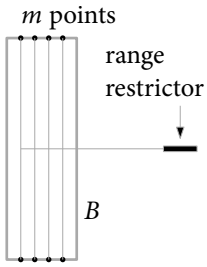


Figure 9.17: The blocker gadget.

Theorem 9.9. *Given a set P of input points in $k = |C|$ different colors and an integer Y it is NP-complete to decide whether a backbone labeling with one label per color and at most Y leader crossings exists.*

Proof. The proof of NP-hardness is by reduction from the NP-complete Fixed Linear Crossing Number problem [MNKF90], which is defined as follows. Given a graph $G = (V, E)$, a bijective function $f: V \to \{1, \ldots, |V|\}$, and an integer Z, one has to decide whether there is a drawing of G with the vertices placed on a horizontal line (the *spine*) in the order specified by f and the edges drawn as semi-circles above or below the spine so that there are at most Z edge crossings. Masuda et al. [MNKF90] showed that the problem is NP-complete even if G is a matching.

Let G be a matching. Then the number of vertices is even and we can assume that the vertices $V = \{v_1, \ldots, v_{2n}\}$ are indexed in the order specified by f, that is, $f(v_i) = i$ for $1 \leq i \leq 2n$. Furthermore, we direct every edge $\{v_i, v_j\}$ with $i < j$ from v_i to v_j. Let $\{u_1, \ldots, u_n\}$ be the ordered source vertices and let $\{w_1, \ldots, w_n\}$ be the ordered sink vertices. Figure 9.18 shows an example graph G drawn on a spine in the specified order.

In our reduction we will create an *edge gadget* for every edge of G. The gadget consists of five blocker gadgets and one *side selector gadget*. Each of the six sub-gadgets uses its own color and thus defines one middle backbone. The edge gadgets are ordered from left to right according to the sequence of source vertices $(u_1, \ldots, u_n)$. Figure 9.19 shows a sketch of the instance I_G created for the matching G with four edges shown in Figure 9.18.

The edge gadgets are placed symmetrically with respect to the x-axis. We create $2n+1$ special rows below the x-axis and $2n+1$ special rows above, indexed by $-(2n+1), -2n, \ldots, 0, \ldots, 2n, 2n+1$. The gadget for an edge (v_i, v_j) uses five blocker gadgets (denoted as *central, upper, lower, upper gap*, and *lower gap* blockers) in two different columns to create two small gaps in rows j and $-j$, see the hatched blocks in the same color in Figure 9.19. The upper and lower blockers extend vertically to rows $2n+1$ and $-2n-1$, respectively. The gaps are intended to create two alternatives for routing the backbone of the side selector. Every backbone that starts left of the two gap blockers is forced to cross at least one of these five blocker gadgets as long as it is vertically placed between rows $2n+1$ and $-2n-1$.

The blockers have width $m = 8n^2$. Their backbones are fixed to lie between rows 0 and -1 for the central blocker, between rows $2n$ and $2n+1$ ($-2n$ and $-2n-1$) for the upper (respectively lower) blocker, and between rows j and $j+1$ ($-j$ and $-j-1$) for the upper (respectively lower)

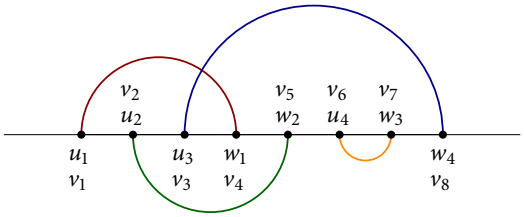


Figure 9.18: An instance of Fixed Linear Crossing Number with four edges.

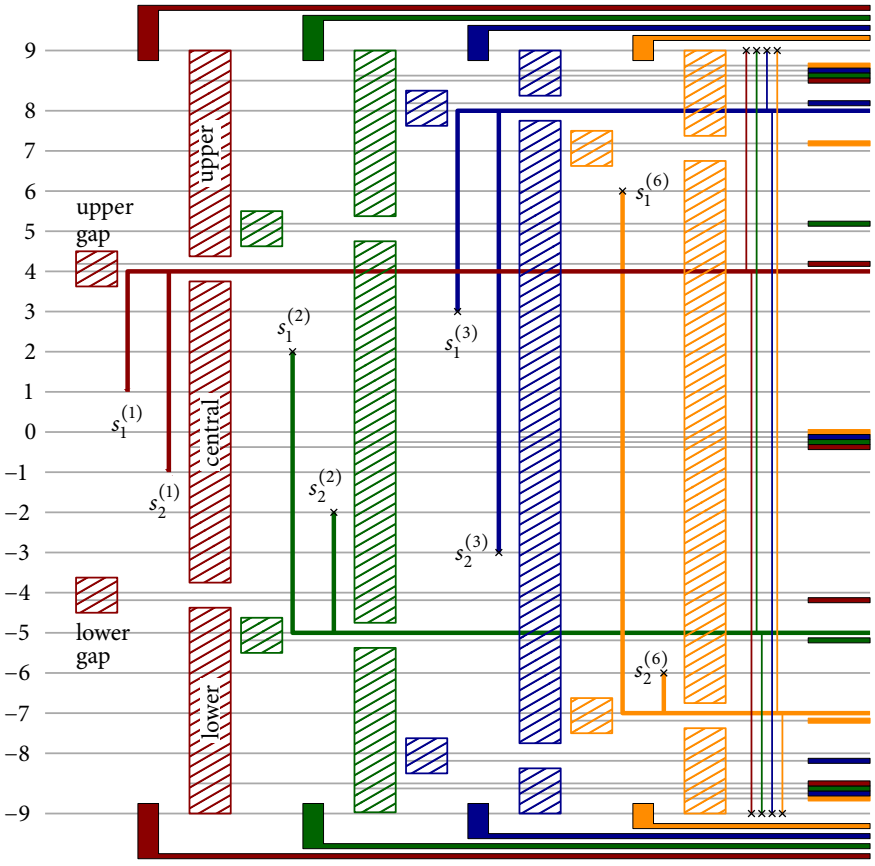


Figure 9.19: Sketch of the reduction of the graph of Figure 9.18 to a backbone labeling instance. Hatched rectangles represent blockers, thick segments represent side selectors, and filled shapes represent guard gadgets or range restrictor gadgets.

gap blocker; recall that this can easily be done by placing the range restrictor gadget of the blocker at the respective position.

The *side selector* consists of two horizontally spaced *selector points* $s_1^{(i)}$ and $s_2^{(i)}$ in rows i and $-i$ located between the left and right blocker columns. They have the same color and thus define one joint backbone, the *selector backbone*, which is supposed to pass through one of the two gaps in an optimal solution. The n edge gadgets are placed from left to right in the order of their source vertices; see Figure 9.19.

The backbone of every selector gadget is vertically restricted to the range between rows $2n+1$ and $-2n-1$ in any optimal solution by augmenting each selector gadget with a range restrictor gadget. This means that we add two more points for each selector to the right of all edge gadgets, one in row $2n+1$ and the other in row $-2n-1$. They are connected to the selector backbone. In combination with a corresponding upper and lower guard gadget of size $M = \Omega(n^4)$ between the two selector points $s_1^{(i)}$ and $s_2^{(i)}$ this achieves the range restriction according to Lemma 9.3.

This completes the backbone labeling instance. We will now show two important properties of optimal solutions for the constructed instance. We first show that the selector backbones do indeed pass through one of their two gaps.

Property 1. In a crossing-minimal labeling the backbone of the selector gadget for every edge (v_i, v_j) passes through one of its two gaps in rows j or $-j$.

Proof. There are basically three different options for placing a selector backbone: (a) outside its range restriction, that is, above row $2n+1$ or below row $-2n-1$, (b) between rows $2n+1$ and $-2n-1$, but not in one of the two gaps, and (c) in rows j or $-j$, that is, inside one of the gaps. In case (a) we get at least $M = \Omega(n^4)$ crossings by Lemma 9.3. So we may assume that case (a) never occurs for any selector gadget; we will see that in this case there are only $O(n^4)$ crossing in total for the selector gadgets. In cases (b) and (c) we note that the backbone will cross one blocker for each edge whose source vertex is right of v_i in the order $(u_1, \ldots, u_n)$. Let k be the number of these edges. Additionally, in case (b), the backbone crosses one of its own blockers. In cases (b) and (c) the two vertical segments of the range restrictor of edge (v_i, v_j) cross every selector and blocker backbone regardless of the position of its own backbone, which yields $6n-1$ crossings. Thus, case (b) causes at least $(k+1) \cdot m + 6n - 1$ crossings.

For giving an upper bound on the number of crossings in case (c) we note that the backbone can cross at most three vertical segments of any other selector gadget: the two segments connected to its selector points and one segment connected to a point in either row $2n+1$ or row $-2n-1$, which is part of the range restrictor gadget. The two vertical segments connected to points $s_1^{(i)}$ and $s_2^{(i)}$ together will cross the backbone of each central blocker at most once, the backbones of each pair of upper/lower gap blockers at most twice, and each selector backbone at most twice. Backbones of upper and lower blockers are never crossed in case (c). So in case (c) the segments of the selector gadget cross at most $km + 8n - 1$ segments, which is less than the lower bound of $(k+1)m + 6n - 1$ in case (b). We conclude that each backbone indeed passes through one of the gaps in an optimal solution. Any violation of this rule would create at least m additional crossings, which is more than what an arbitrary assignment of selector backbones to gaps yields. □

Next, we show how the number of crossings in the backbone labeling instance relates to the number of crossings in the instance of the Fixed Linear Crossing Number problem. There is

a large number of unavoidable crossings regardless of the backbone positions of the selector gadgets. By Property 1 and the fact that violating any range restriction immediately causes $M = \Omega(n^2)$ crossings, we can assume that every backbone adheres to the rules, that is, stays within its range as defined by the range restriction gadgets or passes through one of its two gaps, in the case of selector backbones.

Property 2. An optimal solution of the backbone labeling instance I_G created for a matching G with n edges has $X + 2Z$ crossings, where X is a constant depending on G and Z is the minimum number of crossings of G in the Fixed Linear Crossing Number instance.

Proof. Aside from guard backbones, which never have crossings, there are two types of backbones in our construction, blocker and selector backbones. We argue separately for all four possible types of crossings and distinguish fixed crossings that must occur and variable crossings that depend on the placement of the selector backbones. The types of crossings are

(i) crossings between blocker backbones and vertical blocker segments,

(ii) crossings between blocker backbones and vertical selector segments,

(iii) crossings between selector backbones and vertical blocker segments, and

(iv) crossings between selector backbones and vertical selector segments.

We will analyze the numbers of crossings for these types individually.

Case (i): By construction each blocker backbone must intersect exactly one blocker gadget of width m for each edge gadget to its right. Thus we obtain

$$X_1 = 5m \sum_{i=1}^{n-1} i = 5m \cdot \frac{n^2 - n}{2}$$

fixed crossings in total from Case (i).

Case (ii): Each blocker backbone crosses, for each edge, exactly one vertical selector segment that is part of the range restrictor gadget on the right-hand side of our construction. Each central blocker backbone additionally crosses for each edge gadget to its right one vertical segment incident to one of the selector points, regardless of the selector position. The two gap blocker backbones for gaps in rows j and $-j$ together cause two additional crossings for each edge gadget to its right whose target vertex v_k satisfies $k > j$. To see this we need to distinguish two cases. Let $e = (v_i, v_k)$ be the edge of an edge gadget with $k > j$. If $i < j$, then both vertical selector segments either cross the lower gap blocker backbone or they both cross the upper gap blocker backbone (see edges (v_1, v_4) and (v_2, v_5) in Figure 9.19). If $i > j$, then one of the two vertical selector segments crosses both gap blocker backbones, and the other one crosses none (see edges (v_1, v_4) and (v_6, v_7) in Figure 9.19). The backbones of the upper and lower blockers do not cross any other vertical selector segment.

Let $\kappa = |\{\{(v_i, v_j), (v_k, v_l)\} \in E^2 \mid i < k \text{ and } j < l\}| = O(n^2)$ be the number of pairs of edges causing crossings with gap blocker backbones. Then we obtain

$$X_2 = 5n^2 + \frac{n^2 - n}{2} + 2\kappa$$

fixed crossings from Case (ii).

Case (iii): Each selector backbone that passes through one of its gaps crosses exactly one blocker gadget for each edge gadget to its right. Thus, we obtain

$$X_3 = m \sum_{i=1}^{n-1} i = m \cdot \frac{n^2 - n}{2}$$

fixed crossings in Case (iii).

Case (iv): Let $e = (v_i, v_j)$ and $f = (v_k, v_l)$ be two edges in G, and let $i < k$. Then there are three sub-cases: (a) e and f are *sequential*, that is, $i < j < k < l$, (b) e and f are *nested*, that is, $i < k < l < j$, or (c) e and f are *interlaced*, that is, $i < k < j < l$. For every pair of sequential edges there is exactly one crossing, regardless of the gap assignments (see edges (v_1, v_4) and (v_6, v_7) in Figure 9.19). For every pair of nested edges there is no crossing, regardless of the gap assignments (see edges (v_3, v_8) and (v_6, v_7) in Figure 9.19). Finally, for every pair of interlaced edges there are no crossings if the respective side selector backbones are assigned to opposite sides of the x-axis or two crossings if they are assigned to the same side. Therefore, pairs of interlaced edges do not contribute to the number of fixed crossings. Let $\tau = \left|\left\{\{(v_i, v_j), (v_k, v_l)\} \in E^2 \mid i < j < k < l\right\}\right| = O(n^2)$ be the number of pairs of sequential edges. Then we obtain

$$X_4 = \tau$$

fixed crossings from Case (iv).

From the discussion of the four cases we can immediately see that all crossings are fixed, except for those related to pairs of interlaced edges (see, for example, edges (v_1, v_4) and (v_3, v_8) or (v_2, v_5) in Figure 9.19). These are exactly the edge pairs that create crossings in the Fixed Linear Crossing Number problem if assigned to the same side of the spine. As discussed in Case (iv) the selector gadgets of two interlaced edges create two extra crossings if and only if they are assigned to gaps on the same side of the x-axis. If we create a bijection that maps a selector backbone placed in the upper gap to an edge drawn above the spine, and a selector backbone in the lower gap to an edge drawn below the spine, we see that an edge crossing on the same side of the spine in a drawing of G corresponds to two extra crossings in a labeling of I_G and vice versa. So, if Z is the minimum number of crossings in a spine drawing of G, then $2Z$ is the minimum number of variable crossings in a labeling of I_G. By setting $X = X_1 + X_2 + X_3 + X_4$ this proves Property 2. □

From Property 2 it follows immediately that crossing minimization with finite backbones is NP-hard since the size of the instance I_G is polynomial in n (more precisely, we need only $O(n^5)$ points for I_G).

Furthermore, we can guess an order of the backbones and then, by using the algorithm of Theorem 9.7, compute the minimum crossing number for this order. This shows that crossing minimization is contained in NP. Hence, the problem is NP-complete. □

9.5 Concluding Remarks

We have introduced the new model of many-to-one labeling with backbones; this model generalizes the po-leader model of classic boundary labeling to many-to-one boundary labeling. For both settings, finite and infinite backbones, we have seen that minimizing the total number of labels as well as minimizing the total length of leaders can be achieved in polynomial time by using dynamic programming. On the other hand, only very restricted versions of crossing minimization can be solved efficiently. In general, crossing minimization with a bounded number of labels per color is NP-hard for finite backbones.

Open Problems. In the setting of crossing minimization, we have just seen hardness for finite backbones. In our hardness proof it was essential that backbones do not extend infinitely to the right; hence, the hardness proof we gave cannot simply be modified for showing hardness for infinite backbones. It is, therefore, an open problem whether the simpler structure of many-to-one labelings with infinite backbones allows for efficiently minimizing the number of crossings or the problem is also NP-hard.

The other optimization criteria, that is, minimizing the total number of labels or the total leader length can be solved optimally. We did, however, only consider the case where just one side of the focus region is used for placing the labels. For infinite backbones, using both the left and the right boundary does not make a difference. However, we get much more flexibility in the case of finite backbones. An open question is, hence, whether the 2-sided problem variant is still solvable in polynomial time.

For labeling circular focus regions (see Chapter 8), we developed algorithms that maximize the number (or the weight) of sites that can be labeled subject to constraints such as a fixed number of labels or a minimum gap between two labels. Similar problems can be considered for the backbone labelings discussed in this chapter. For example, subject to upper bounds on the number of labels for the different colors, we want to maximize the number of sites (or the total weight of sites, respectively) that are connected to a label of their color. Both for finite and infinite backbones this problem should be solvable in polynomial time by modifying the respective dynamic programs presented for leader length minimization in Section 9.3. To this end, entries of the tables must represent the maximum weight of sites that can be labeled in subinstances and minimization must be changed to maximization. Some further modifications will be necessary, but should not be too hard to realize.

In the previous chapter, we presented a post-processing step for classic straight-line boundary labeling that replaced the straight-line leaders by Bézier curves. A similar approach could be tried for many-to-one backbone boundary labeling. Any site would be connected to its horizontal backbone via a Bézier curve with a horizontal tangent at the backbone, that is, with a smooth transition. Since many sites may be connected to the same backbone, this can create a nice confluent appearance; see Figure 9.20.

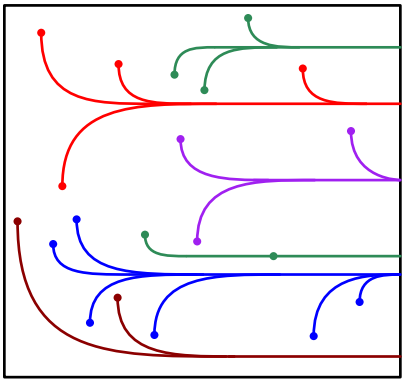


Figure 9.20: Sketch of a backbone labeling with Bézier curves.

Chapter 10

Conclusion

In this thesis, we have investigated three areas of graph drawing: metro maps, point-set embeddability, and boundary labeling. In all three areas, we have presented algorithms for tractable cases, but we have also seen that there are NP-hard subproblems by providing hardness proofs. In the case of metro lines and boundary labeling, minimizing the number of crossings is NP-hard, while for many point-set embeddability problems deciding whether a feasible solution exists—without restricting the number of crossings—is itself NP-hard.

In these results, the problems considered are quite representative for the area of graph drawing. The same holds for the type of results we presented. We have seen very practical algorithms that have been implemented and tested in Chapters 3 and 8, as well as provably optimal algorithms, but also many theoretical results.

Perspectives for Future Work. In the concluding remarks of the individual chapters, we have seen some specific open problems in the context of the respective chapter. In what follows, we will see a few perspectives for possible future work that generalize or extend the considered problems in a broader way.

For metro maps, we have seen methods for creating curvy drawings of the network and for minimizing crossings between metro lines. An interesting direction for future research is the transfer of drawing conventions and algorithms for metro maps to other areas in which some relevant lines—that is, mainly paths—in networks should be visualized. Nesbitt [Nes04] discussed the use of this *metro map metaphor* for different purposes such as navigating through web pages or visualizing business plans. For tours through the internet, the automatic creation of metro-map like drawings has been studied by Svandad et al. [SGSK01]. Stott et al. [SRB+05] presented a method for visualizing project plans in the style of metro maps. However, no general purpose algorithm—independent of specific applications—for creating metro-map like drawings is available. The input of such an algorithm would be just a graph and paths on that graph, that will play the role of metro maps. The most important difference to real-world metro networks is that no geographic positions of vertices are given. On the one hand, this allows more flexibility for creating nice drawings. On the other hand, it makes it also more complicated to find a first feasible drawing, for example, when trying to adapt our method presented in Chapter 3.

There has been previous work pointing out relations between metro-line crossing minimization and edge bundling. More specifically, the edges of the original graph become lines in the graph after bundling. In this setting, there can be edges with a large number of lines—much larger than in real-world metro maps. We think that block crossings as considered in Chapter 5 can help a lot in improving the readability of bundled graph drawings.

For point-set embeddability, there have already been some results, and we have added some more. However, in a practical application, a small deviation between the desired positions (that is, the input points) and the actual vertex positions could often be tolerated because a user will hardly realize such a deviation. This relaxation gives rise to two new variants of embeddability problem. We can either define the largest deviation that is allowed for a vertex, or we can measure all deviations and try to find a feasible embedding that minimizes the deviations. In both cases, relaxing the position constraint a bit can allow us to find a feasible embedding if there has not been one for the stricter version, or it can allow us to find a nicer embedding. Löffler [Löf11] considered the version with maximum deviation for planar straight-line drawings and showed hardness even for cycles. For general graphs and nonplanar drawings, Abellanas et al. [AAPS05] developed a force-directed heuristic by adding a force that tries to keep the vertex in its desired region; this is similar to what we did in Chapter 3 for modeling that metro stations should be drawn close to their geographic position.

In boundary labeling, there are algorithms that find feasible solutions for various styles of leaders. In few cases, even the interaction with the underlying map can already be taken into account. However, the interaction between different leaders—except for intersections—is rarely considered as an optimization criterion. In some drawing styles, algorithms often even use a track routing area which can contain many parallel segments placed close together. In practice, we want to have large gaps between leaders such that it is easy to distinguish between different leaders. Furthermore, we would also like that leaders do not come close to sites—with the exception of their own site.

The open problems we have mentioned here are examples for a general tendency in graph drawing: several problems are solved in theory, while the resulting drawings are, in practices, not very nice.

Bibliography

[AAPS05] Manuel Abellanas, Andrés Aiello, Gregorio Hernández Penalver, and Rodrigo I. Silveira. Network drawing with geographical constraints on vertices. *Actas XI Encuentros de Geometra Computacional*, pages 111–118, 2005. [see page 208]

[ABKS10] Evmorfia N. Argyriou, Michael A. Bekos, Michael Kaufmann, and Antonios Symvonis. On metro-line crossing minimization. *Journal of Graph Algorithms and Applications*, 14(1):75–96, 2010. [see pages 45, 47, 48, 49, 51, 55, 63, and 71]

[ACMM05] Amit Agarwal, Moses Charikar, Konstantin Makarychev, and Yury Makarychev. $O(\sqrt{\log n})$ approximation algorithms for Min UnCut, Min 2CNF deletion, and directed cut problems. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05)*, pages 573–581, New York, 2005. ACM. [see page 59]

[AES99] Pankaj K. Agarwal, Alon Efrat, and Micha Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM Journal on Computing*, 29(3):912–953, 1999. [see page 157]

[AFK+12] Karin Arikushi, Radoslav Fulek, Balázs Keszegh, Filip Morić, and Csaba Tóth. Graphs that admit right angle crossing drawings. *Computational Geometry: Theory and Applications*, 45(4):169–177, 2012. [see pages 109, 113, and 116]

[AGM08] Matthew Asquith, Joachim Gudmundsson, and Damian Merrick. An ILP for the metro-line crossing problem. In James Harland and Prabhu Manyem, editors, *Proceedings of 14th Computing: The Australasian Theory Symposium (CATS'08)*, volume 77 of *CRPIT*, pages 49–56. Australian Computer Society, 2008. [see pages 47, 48, and 55]

[AHS05] Kamran Ali, Knut Hartmann, and Thomas Strothotte. Label layout for interactive 3D illustrations. *Journal of WSCG*, 13(1):1–8, 2005. [see page 152]

[AV07] David Arthur and Sergei Vassilvitskii. `k-means++`: The advantages of careful seeding. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'07)*, pages 1027–1035. SIAM, 2007. [see page 166]

[Bar83] Francisco Barahona. On some weakly bipartite graphs. *Operations Research Letters*, 2(5):239–242, 1983. [see page 61]

[BCDE13] Michael J. Bannister, Zhanpeng Cheng, William E. Devanny, and David Eppstein. Superpatterns and universal point sets. In Stephen Wismath and Alexander Wolff,

editors, *Proceedings of the 21st International Symposium on Graph Drawing (GD'13)*, volume 8242 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 2013. [see page 109]

[BCF+13] Michael A. Bekos, Sabine Cornelsen, Martin Fink, Seok-Hee Hong, Michael Kaufmann, Martin Nöllenburg, Ignaz Rutter, and Antonios Symvonis. Many-to-one boundary labeling with backbones. In Stephen Wismath and Alexander Wolff, editors, *Proceedings of the 21st International Symposium on Graph Drawing (GD'13)*, volume 8242 of *Lecture Notes in Computer Science*, pages 244–255. Springer-Verlag, 2013. [see page 10]

[Ber99] François Bertault. A force-directed algorithm that preserves edge crossing properties. In Jan Kratochvíl, editor, *Proceedings of the 7th International Symposium on Graph Drawing (GD'99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 351–358. Springer-Verlag, 1999. [see page 33]

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. [see page 157]

[BFR12] Laurent Bulteau, Guillaume Fertin, and Irena Rusu. Sorting by transpositions is difficult. *SIAM Journal on Discrete Mathematics*, 26(3):1148–1180, 2012. [see pages 76 and 77]

[BHKN09] Marc Benkert, Herman J. Haverkort, Moritz Kroll, and Martin Nöllenburg. Algorithms for multi-criteria boundary labeling. *Journal of Graph Algorithms and Applications*, 13(3):289–317, 2009. [see page 151]

[BHNP13] Sergey Bereg, Alexander E. Holroyd, Lev Nachmanson, and Sergey Pupyrev. Drawing permutations with few corners. In Stephen Wismath and Alexander Wolff, editors, *Proceedings of the 21st International Symposium on Graph Drawing (GD'13)*, volume 8242 of *Lecture Notes in Computer Science*, pages 484–495. Springer-Verlag, 2013. [see pages 71 and 103]

[BKK+13] Michael A. Bekos, Michael Kaufmann, Robert Krug, Stefan Näher, and Vincenzo Roselli. Slanted orthogonal drawings. In Stephen Wismath and Alexander Wolff, editors, *Proceedings of the 21st International Symposium on Graph Drawing (GD'13)*, volume 8242 of *Lecture Notes in Computer Science*, pages 428–439. Springer-Verlag, 2013. [see page 121]

[BKNS10] Michael A. Bekos, Michael Kaufmann, Martin Nöllenburg, and Antonios Symvonis. Boundary labeling with octilinear leaders. *Algorithmica*, 57(3):436–461, 2010. [see page 151]

[BKPS08] Michael A. Bekos, Michael Kaufmann, Katerina Potika, and Antonios Symvonis. Line crossing minimization on metro maps. In Seok-Hee Hong, Takao Nishizeki, and Wu Quan, editors, *Proceedings of the 15th International Symposium on Graph Drawing (GD'07)*, volume 4875 of *Lecture Notes in Computer Science*, pages 231–242. Springer-Verlag, 2008. [see pages 46, 47, 48, 63, and 90]

[BKPS11] Michael A. Bekos, Michael Kaufmann, Dimitrios Papadopoulos, and Antonios Symvonis. Combining traditional map labeling with boundary labeling. In Ivana Cerná, Tibor Gyimóthy, Juraj Hromkovic, Keith G. Jeffery, Rastislav Královic, Marko Vukolic, and Stefan Wolf, editors, *Proceedings of the 37th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'11)*, volume 6543 of *Lecture Notes in Computer Science*, pages 111–122. Springer-Verlag, 2011. [see page 152]

[BKSW05] Michael A. Bekos, Michael Kaufmann, Antonios Symvonis, and Alexander Wolff. Boundary labeling: Models and efficient algorithms for rectangular maps. In János Pach, editor, *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *Lecture Notes in Computer Science*, pages 49–59. Springer-Verlag, 2005. [see page 151]

[BKSW07] Michael A. Bekos, Michael Kaufmann, Antonios Symvonis, and Alexander Wolff. Boundary labeling: Models and efficient algorithms for rectangular maps. *Computational Geometry: Theory and Applications*, 36(3):215–236, 2007. [see pages 146, 151, 156, and 157]

[BMS97] Prosenjit Bose, Michael McAllister, and Jack Snoeyink. Optimal algorithms to embed trees in a point set. *Journal of Graph Algorithms and Applications*, 1(2):1–15, 1997. [see page 109]

[BNUW07] Marc Benkert, Martin Nöllenburg, Takeaki Uno, and Alexander Wolff. Minimizing intra-edge crossings in wiring diagrams and public transport maps. In Michael Kaufmann and Dorothea Wagner, editors, *Proceedings of the 14th International Symposium on Graph Drawing (GD'06)*, volume 4372 of *Lecture Notes in Computer Science*, pages 270–281. Springer-Verlag, 2007. [see pages 43, 47, and 48]

[Bos02] Prosenjit Bose. On embedding an outer-planar graph in a point set. *Computational Geometry: Theory and Applications*, 23(3):303–312, 2002. [see page 109]

[BP98] Vineet Bafna and Pavel A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, 1998. [see pages 75, 76, 77, and 79]

[BRL09] Enrico Bertini, Maurizio Rigamonti, and Denis Lalanne. Extended excentric labeling. *Computer Graphics Forum*, 28(3):927–934, 2009. [see page 151]

[BW97] Matthias Bader and Robert Weibel. Detecting and resolving size and proximity conflicts in the generalization of polygonal maps. In *Proceedings of the 18th ICA/ACI International Cartographic Conference (ICC'97)*, pages 1525–1532, Stockholm, 1997. [see page 167]

[BW00] Ulrik Brandes and Dorothea Wagner. Using graph layout to visualize train connection data. *Journal of Graph Algorithms and Applications*, 4(3):135–155, 2000. [see page 28]

[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):353–366, 2006. [see pages 108, 109, and 110]

[CBB91] Raju Chithambaram, Kate Beard, and Renato Barrera. Skeletonizing polygons for map generalization. In *Technical papers, ACSM-ASPRS Convention, Cartography and GIS/LIS*, volume 2, pages 44–54, 1991. [see page 167]

[CCG+12] Roman Chernobelskiy, Kathryn I. Cunningham, Michael T. Goodrich, Stephen G. Kobourov, and Lowell Trott. Force-directed lombardi-style graph drawing. In Marc J. van Kreveld and Bettina Speckmann, editors, *Proceedings of the 19th International Symposium on Graph Drawing (GD'11)*, volume 7034 of *Lecture Notes in Computer Science*, pages 320–331. Springer-Verlag, 2012. [see page 28]

[Che89] Paul L. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989. [see page 168]

[CI01] David A. Christie and Robert W. Irving. Sorting strings by reversals and by transpositions. *SIAM Journal on Discrete Mathematics*, 14(2):193–206, 2001. [see pages 76 and 78]

[CL98] Hsiao-Feng Steven Chen and D. T. Lee. On crossing minimization problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:406–418, 1998. [see page 48]

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, third edition, 2009. [see page 11]

[CMS95] Jon Christensen, Joe Marks, and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995. [see page 150]

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71)*, pages 151–158. ACM, 1971. [see page 18]

[CR11] Md. Emran Chowdhury and Md. Saidur Rahman. Orthogonal point-set embeddings of 3-connected and 4-connected planar graphs. In *Proceedings of the 14th International Conference on Computer and Information Technology (ICCIT'11)*, pages 327–332, 2011. [see page 121]

[CZQ+08] Weiwei Cui, Hong Zhou, Huamin Qu, Pak Chung Wong, and Xiaoming Li. Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–1284, 2008. [see page 44]

[DEG+12] Christian A. Duncan, David Eppstein, Michael T. Goodrich, Stephen G. Kobourov, and Martin Nöllenburg. Lombardi drawings of graphs. *Journal of Graph Algorithms and Applications*, 16(1):85–108, 2012. [see page 28]

[DEG+13] Christian A. Duncan, David Eppstein, Michael T. Goodrich, Stephen G. Kobourov, and Martin Nöllenburg. Drawing trees with perfect angular resolution and polynomial area. *Discrete & Computational Geometry*, 49(2):157–182, 2013. [see page 28]

[DEL11] Walter Didimo, Peter Eades, and Giuseppe Liotta. Drawing graphs with right angle crossings. *Theoretical Computer Science*, 412(39):5156–5166, 2011. [see pages 107, 110, and 116]

[DETT99] Guiseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999. [see page 11]

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer-Verlag, 1999. [see page 20]

[DF13] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer-Verlag, 2013. [see page 20]

[dFPP90] Hubert de Fraysseix, János Pach, and Richard Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990. [see page 14]

[DHM08] Tim Dwyer, Nathan Hurst, and Damian Merrick. A fast and simple heuristic for metro map path simplification. In George Bebis, Richard D. Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Fatih Murat Porikli, Jörg Peters, James T. Klosowski, Laura L. Arns, Yu Ka Chun, Theresa-Marie Rhyne, and Laura Monroe, editors, *Proceedings of the 4th International Symposium onAdvances in Visual Computing (ISVC'08)*, volume 5359 of *Lecture Notes in Computer Science*, pages 22–30. Springer-Verlag, 2008. [see page 25]

[Die10] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, 4th edition, 2010. [see page 11]

[DJP+94] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, Paul D. Seymour, and Mihalis Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994. [see pages 19 and 97]

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984. [see page 15]

[EEK07] Alon Efrat, Cesim Erten, and Stephen G. Kobourov. Fixed-location circular arc drawing of planar graphs. *Journal of Graph Algorithms and Applications*, 11(1):145–164, 2007. [see page 109]

[EH06] Isaac Elias and Tzvika Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):369–379, 2006. [see pages 76 and 77]

[EIS76] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976. [see pages 57 and 124]

[Epp13] David Eppstein. Planar Lombardi drawings for subcubic graphs. In Walter Didimo and Maurizio Patrignani, editors, *Proceedings of the 20th International Symposium on Graph Drawing (GD'12)*, volume 7704 of *Lecture Notes in Computer Science*, pages 126–137. Springer-Verlag, 2013. [see page 28]

[FHM+12] Martin Fink, Jan-Henrik Haunert, Tamara Mchedlidze, Joachim Spoerhase, and Alexander Wolff. Drawing graphs with vertices at specified positions and crossings at large angles. In Md. Saidur Rahman and Shin-ichi Nakano, editors, *Proceedings of the 6th Workshop on Algorithms and Computation (WALCOM'12)*, volume 7157 of *Lecture Notes in Computer Science*, pages 186–197. Springer-Verlag, 2012. [see pages 7 and 8]

[FHN+13] Martin Fink, Herman Haverkort, Martin Nöllenburg, Maxwell Roberts, Julian Schuhmann, and Alexander Wolff. Drawing metro maps using Bézier curves. In Walter Didimo and Maurizio Patrignani, editors, *Proceedings of the 20th International Symposium on Graph Drawing (GD'12)*, volume 7704 of *Lecture Notes in Computer Science*, pages 463–474. Springer-Verlag, 2013. [see page 5]

[FHS+12] Martin Fink, Jan-Henrik Haunert, André Schulz, Joachim Spoerhase, and Alexander Wolff. Algorithms for labeling focus regions. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2583–2592, 2012. [see page 8]

[FLR+09] Guillaume Fertin, Anthony Labarre, Irena Rusu, Eric Tannier, and Stéphane Vialette. *Combinatorics of Genome Rearrangements*. MIT Press, 2009. [see page 76]

[FLW14] Martin Fink, Magnus Lechner, and Alexander Wolff. Concentric metro maps. In Maxwell J. Roberts and Peter Rodgers, editors, *Abstracts of the Schematic Mapping Workshop 2014*, 2014. Poster. [see pages 41 and 42]

[FP98] Jean-Daniel Fekete and Catharine Plaisant. Excentric labeling: Dynamic neighborhood labeling for data visualization. Technical Report HCIL-98-09, Department of Computer Science, University of Maryland, 1998. [see page 150]

[FP99] Jean-Daniel Fekete and Catharine Plaisant. Excentric labeling: Dynamic neighborhood labeling for data visualization. In *Proceedings of the ACM CHI 1999 Conference on Human Factors in Computing Systems (CHI'99)*, pages 512–519, 1999. [see pages 148 and 150]

[FP13a] Martin Fink and Sergey Pupyrev. Metro-line crossing minimization: Hardness, approximations, and tractable cases. In Stephen Wismath and Alexander Wolff, editors, *Proceedings of the 21st International Symposium on Graph Drawing (GD'13)*, volume 8242 of *Lecture Notes in Computer Science*, pages 328–339. Springer-Verlag, 2013. [see page 6]

[FP13b] Martin Fink and Sergey Pupyrev. Ordering metro lines by block crossings. In Krishnendu Chatterjee and Jiri Sgall, editors, *Proceedings of the 38th International Symposium on Mathematical Foundations of Computer Science (MFCS'13)*, volume 8087 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 2013. [see page 7]

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. [see pages 15 and 31]

[FT05] Benjamin Finkel and Roberto Tamassia. Curvilinear graph drawing using the force-directed method. In János Pach, editor, *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *Lecture Notes in Computer Science*, pages 448–453. Springer-Verlag, 2005. [see page 28]

[FW91] Michael Formann and Frank Wagner. A packing problem with applications to lettering of maps. In *Proceedings of the 7th Annual Symposium on Computational Geometry (SCG'91)*, pages 281–288. ACM, 1991. [see page 150]

[GDLM11] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, and Henk Meijer. Area, curve complexity, and crossing resolution of non-planar graph drawings. *Theory of Computing Systems*, 49(3):565–575, 2011. [see pages 109, 110, 114, and 118]

[GFF+13] Emilio Di Giacomo, Fabrizio Frati, Radoslav Fulek, Luca Grilli, and Marcus Krug. Orthogeodesic point-set embedding of trees. *Computational Geometry: Theory and Applications*, 46(8):929–944, 2013. [see pages 108, 121, and 137]

[GHN11] Andreas Gemsa, Jan-Henrik Haunert, and Martin Nöllenburg. Boundary-labeling algorithms for panorama images. In Isabel F. Cruz, Divyakant Agrawal, Christian S. Jensen, Eyal Ofek, and Egemen Tanin, editors, *Proceedings of the 19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (ACM-GIS'11)*, pages 289–298. ACM, 2011. [see page 152]

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. [see pages 11, 18, 59, and 60]

[GJ83] Michael R. Garey and David S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983. [see pages 3 and 13]

[GKO+09] Xavier Goaoc, Jan Kratochvíl, Yoshio Okamoto, Chan-Su Shin, Andreas Spillner, and Alexander Wolff. Untangling a planar graph. *Discrete & Computational Geometry*, 42(4):542–569, 2009. [see page 109]

[GMPP91] Peter Gritzmann, Bojan Mohar, János Pach, and Richard M. Pollack. Embedding a planar triangulation with vertices at specified positions. *The American Mathematical Monthly*, 98:165–166, 1991. [see pages 107 and 109]

[GP81] Martin Grötschel and William R. Pulleyblank. Weakly bipartite graphs and the Max-Cut problem. *Operations Research Letters*, 1(1):23–27, 1981. [see page 61]

[Gro89] Patrick Groeneveld. Wire ordering for detailed routing. *IEEE Design & Test of Computers*, 6(6):6–17, 1989. [see pages 44 and 47]

[GT01] Ashim Garg and Roberto Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM Journal on Computing*, 31(2):601–625, 2001. [see page 121]

[Hal91] John H. Halton. On the thickness of graphs of given degree. *Information Sciences*, 54(3):219–238, 1991. [see page 109]

[HGAS05] Knut Hartmann, Timo Götzelmann, Kamran Ali, and Thomas Strothotte. Metrics for functional and aesthetic label layouts. In Andreas Butz, Brian Fisher, Antonio Krüger, and Patrick Olivier, editors, *Proceedings of the 5th International Symposium on Smart Graphics (SG'05)*, volume 3638 of *Lecture Notes in Computer Science*, pages 115–126. Springer-Verlag, 2005. [see page 152]

[HHE08] Weidong Huang, Seok-Hee Hong, and Peter Eades. Effects of crossing angles. In *Proceedings of the IEEE Pacific Visualisation Symposium 2008 (PacificVis'08)*, pages 41–46, 2008. [see pages 4 and 107]

[HMdN06] Seok-Hee Hong, Damian Merrick, and Hugo A. D. do Nascimento. Automatic visualisation of metro maps. *Journal of Visual Languages and Computing*, 17(3):203–224, 2006. [see pages 24 and 25]

[Hol06] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006. [see page 44]

[HS11] Jan-Henrik Haunert and Leon Sering. Drawing road networks with focus regions. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2555–2562, 2011. [see page 145]

[HT74] John E. Hopcroft and Robert E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974. [see page 14]

[HT91] Refael Hassin and Arie Tamir. Improved complexity bounds for location problems on the real line. *Operations Research Letters*, 10(7):395–402, 1991. [see page 189]

[HV98] Lenwood S. Heath and John Paul C. Vergara. Sorting by bounded block-moves. *Discrete Applied Mathematics*, 88(1–3):181–206, 1998. [see page 76]

[Kau09] Michael Kaufmann. On map labeling with leaders. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 290–304. Springer-Verlag, 2009. [see page 152]

[KKRW10] Bastian Katz, Marcus Krug, Ignaz Rutter, and Alexander Wolff. Manhattan-geodesic embedding of planar graphs. In David Eppstein and Emden R. Gansner, editors, *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *Lecture Notes in Computer Science*, pages 207–218. Springer-Verlag, 2010. [see pages 120, 122, 131, and 132]

[KNR+13] Philipp Kindermann, Benjamin Niedermann, Ignaz Rutter, Marcus Schaefer, André Schulz, and Alexander Wolff. Two-sided boundary labeling with adjacent sides. In Frank Dehne, Roberto Solis-Oba, and Jörg-Rüdiger Sack, editors, *Proceedings of the 13th International on Algorithms and Data Structures (WADS'13)*, volume 8037 of *Lecture Notes in Computer Science*, pages 463–474. Springer-Verlag, 2013. [see page 152]

[Kob13] Stephen G. Kobourov. Force-directed drawing algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, volume 81 of *Discrete Mathematics and Its Applications*, chapter 12. Chapman & Hall/CRC, 2013. [see page 16]

[KS12] Mikio Kano and Kazuhiro Suzuki. Geometric graphs in the plane lattice. In Alberto Márquez, Pedro Ramos, and Jorge Urrutia, editors, *Proceedings of the XIVth Spanish Meeting on Computational Geometry (ECG'11)*, volume 7579 of *Lecture Notes in Computer Science*, pages 274–281. Springer-Verlag, 2012. [see pages 121, 138, and 142]

[Kuh55] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1–2):83–97, 1955. [see pages 158 and 197]

[Kur04] Maciej Kurowski. A 1.235 lower bound on the number of points needed to draw all n-vertex planar graphs. *Information Processing Letters*, 92(2):95–98, 2004. [see page 109]

[KW01] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. [see page 11]

[KW02] Michael Kaufmann and Roland Wiese. Embedding vertices at points: Few bends suffice for planar graphs. *Journal of Graph Algorithms and Applications*, 6(1):115–129, 2002. [see pages 107 and 109]

[Lin10] Chun-Cheng Lin. Crossing-free many-to-one boundary labeling with hyperleaders. In *Proceedings of the IEEE Pacific Visualization Symposium 2010 (PacificVis'10)*, pages 185–192. IEEE, 2010. [see pages 178 and 179]

[LKY08] Chun-Cheng Lin, Hao-Jen Kao, and Hsu-Chun Yen. Many-to-one boundary labeling. *Journal of Graph Algorithms and Applications*, 12(3):319–356, 2008. [see page 178]

[LN13] Maarten Löffler and Martin Nöllenburg. Planar Lombardi drawings of outerpaths. In Walter Didimo and Maurizio Patrignani, editors, *Proceedings of the 20th International Symposium on Graph Drawing (GD'12)*, volume 7704 of *Lecture Notes in Computer Science*, pages 561–562. Springer-Verlag, 2013. [see page 28]

[Löf11] Maarten Löffler. Existence and computation of tours through imprecise points. *International Journal of Computational Geometry & Applications*, 21(1):1–24, 2011. [see page 208]

[MG07] Damian Merrick and Joachim Gudmundsson. Path simplification for metro map layout. In Michael Kaufmann and Dorothea Wagner, editors, *Proceedings of the 14th International Symposium on Graph Drawing (GD'06)*, volume 4372 of *Lecture Notes in Computer Science*, pages 258–269. Springer-Verlag, 2007. [see page 25]

[MM96] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996. [see page 14]

[MNKF90] Sumio Masuda, Kazuo Nakajima, Toshinobu Kashiwabara, and Toshio Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Transactions on Computers*, 39(1):124–127, 1990. [see page 199]

[Mor80] Joel L. Morrison. Computer technology and cartographic change. In David R.F. Taylor, editor, *The Computer in Contemporary Cartography*. Johns Hopkins University Press, 1980. [see page 148]

[MS95] Malgorzata Marek-Sadowska and Majid Sarrafzadeh. The crossing distribution problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):423–433, 1995. [see pages 47 and 76]

[Nes04] Keith V. Nesbitt. Getting to more abstract places using the metro map metaphor. In *Proceedings of the 8th International Conference on Information Visualisation (IV'04)*, pages 488–493. IEEE Computer Society, 2004. [see page 207]

[Ney01] Gabriele Neyer. Map labeling with application to graph drawing. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs*, volume 2025 of *Lecture Notes in Computer Science*, pages 247–273. Springer-Verlag, 2001. [see page 150]

[Nie06] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006. [see page 20]

[Nöl09] Martin Nöllenburg. *Network Visualization: Algorithms, Applications, and Complexity*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe (TH), 2009. [see pages 48 and 65]

[Nöl10] Martin Nöllenburg. An improved algorithm for the metro-line crossing minimization problem. In David Eppstein and Emden R. Gansner, editors, *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *Lecture Notes in Computer Science*, pages 381–392. Springer-Verlag, 2010. [see pages 45, 47, and 74]

[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. [see pages 24, 25, 36, and 37]

[Ola91] Stephan Olariu. An optimal greedy heuristic to color interval graphs. *Information Processing Letters*, 37(1):21–25, 1991. [see page 128]

[O'R88] Joseph O'Rourke. Uniqueness of orthogonal connect-the-dots. *Computational Morphology*, pages 97–104, 1988. [see page 120]

[OTU13a] Yoshio Okamoto, Yuichi Tatsu, and Yushi Uno. Exact and fixed-parameter algorithms for metro-line crossing minimization problems. In Stephen Wismath and Alexander Wolff, editors, *Proceedings of the 21st International Symposium on Graph Drawing (GD'13)*, volume 8242 of *Lecture Notes in Computer Science*, pages 520–521. Springer-Verlag, 2013. [see pages 47, 58, and 68]

[OTU13b] Yoshio Okamoto, Yuichi Tatsu, and Yushi Uno. Exact and fixed-parameter algorithms for metro-line crossing minimization problems. ArXiv e-print abs/1306.3538, 2013. [see page 47]

[Ove03] Mark Ovenden. *Metro maps of the world*. Harrow Weald: Capital Transport Publishing, 2nd edition, 2003. [see page 24]

[OW00] Chris Olston and Allison Woodruff. Getting portals to behave. In Jock D. Mackinlay, Steven F. Roth, and Daniel A. Keim, editors, *Proceedings of the 6th IEEE Symposium on Information Visualization 2000 (InfoVis'00)*, pages 15–25. IEEE Computer Society, 2000. [see page 145]

[Pat01] Maurizio Patrignani. On the complexity of orthogonal compaction. *Computational Geometry*, 19(1):47–67, 2001. [see page 121]

[PBP02] Hartmut Prautzsch, Wolfgang Boehm, and Marco Paluszny. *Bézier and B-Spline Techniques*. Mathematics and Visualization. Springer-Verlag, 2002. [see page 16]

[PNBH12] Sergey Pupyrev, Lev Nachmanson, Sergey Bereg, and Alexander E. Holroyd. Edge routing with ordered bundles. In Marc J. van Kreveld and Bettina Speckmann, editors, *Proceedings of the 19th International Symposium on Graph Drawing (GD'11)*, volume 7034 of *Lecture Notes in Computer Science*, pages 136–147. Springer-Verlag, 2012. [see pages 44, 47, 48, 71, and 74]

[PT00] Achilleas Papakostas and Ioannis G. Tollis. Efficient orthogonal drawings of high degree graphs. *Algorithmica*, 26(1):100–125, 2000. [see page 116]

[PW01] János Pach and Rephael Wenger. Embedding planar graphs at fixed vertex locations. *Graphs and Combinatorics*, 17(4):717–728, 2001. [see pages 107, 108, and 109]

[Rap86] David Rappaport. On the complexity of computing orthogonal polygons from a set of points. Technical report, Technical Report SOCS-86.9, McGill University, Montréal, Canada, 1986. [see page 120]

[RBvK+08] Iris Reinbacher, Marc Benkert, Marc van Kreveld, Joseph S.B. Mitchell, Jack Snoeyink, and Alexander Wolff. Delineating boundaries for imprecise regions. *Algorithmica*, 50(3):386–414, 2008. [see page 167]

[RCS86] Raghunath Raghavan, James Cohoon, and Sartaj Sahni. Single bend wiring. *Journal of Algorithms*, 7(2):232–257, 1986. [see pages 120, 121, and 122]

[RNL+13] Maxwell J. Roberts, Elizabeth J. Newton, Fabio D. Lagattolla, Simon Hughes, and Megan C. Hasler. Objective versus subjective measures of Paris metro map usability: Investigating traditional octolinear versus all-curves schematics. *International Journal of Human-Computer Studies*, 71:363–386, 2013. [see pages 5 and 24]

[RO09] Igor Razgon and Barry O'Sullivan. Almost 2-SAT is fixed-parameter tractable. *Journal of Computer and System Sciences*, 75(8):435–450, 2009. [see page 59]

[Rob12] Maxwell J. Roberts. *Underground maps unravelled: Explorations in information design*. Published by the author, Wivenhoe, 2012. [see pages 24 and 27]

[RRL12] João Tiago Ribeiro, Rui Rijo, and António Leal. Fast automatic schematics for public transport spider maps. *Procedia Technology*, 5:659–669, 2012. [see page 25]

[RW93] Franz Rendl and Gerhard Woeginger. Reconstructing sets of orthogonal line segments in the plane. *Discrete Mathematics*, 119:167–174, 1993. [see pages 108, 109, 120, 134, and 141]

[Sch90] Walter Schnyder. Embedding planar graphs on the grid. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'90)*, pages 138–148. SIAM, 1990. [see page 14]

[Sch02] Falk Schreiber. High quality visualization of biochemical pathways in BioPath. *In Silico Biology*, 2(2):59–73, 2002. [see page 44]

[SGSK01] Elmer S. Sandvad, Kaj Grønbæk, Lennert Sloth, and Jørgen Lindskov Knudsen. A metro map metaphor for guided tours on the web: The webvise guided tour system. In *Proceedings of the 10th International Conference on World Wide Web (WWW'01)*, pages 326–333, New York, NY, USA, 2001. ACM. [see page 207]

[Sku02] San Skulrattanakulchai. 4-edge-coloring graphs of maximum degree 3 in linear time. *Information Processing Letters*, 81(4):191–195, 2002. [see page 129]

[SRB+05] Jonathan M. Stott, Peter Rodgers, Remo Aslak Burkhard, Michael Meier, and Matthias Thomas Jelle Smis. Automatic layout of project plans using a metro map metaphor. In *Proceedings of the 9th International Conference on Information Visualisation (IV'05)*, pages 203–206. IEEE Computer Society, 2005. [see page 207]

[SRMW11] Jonathan M. Stott, Peter Rodgers, Juan Carlos Martínez-Ovando, and Stephen G. Walker. Automatic metro map layout using multicriteria optimization. *IEEE Transactions on Visualization and Computer Graphics*, 17(1):101–114, 2011. [see pages 24 and 25]

[SV10] Bettina Speckmann and Kevin Verbeek. Necklace maps. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):881–889, 2010. [see page 152]

[SWS+11] Markus Steinberger, Manuela Waldner, Marc Streit, Alexander Lex, and Dieter Schmalstieg. Context-preserving visual links. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2249–2258, 2011. [see page 146]

[Tam87] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987. [see pages 15 and 121]

[Tam13] Roberto Tamassia, editor. *Handbook of Graph Drawing and Visualization*, volume 81 of *Discrete Mathematics and Its Applications*. Chapman & Hall/CRC, 2013. [see page 11]

[TL14] Peng Ti and Zhilin Li. Generation of schematic network maps with automated detection and enlargement of congested areas. *International Journal of Geographical Information Science*, 28(3):521–540, 2014. [see page 28]

[VB06] Oswald Veblen and William H. Bussey. Finite projective geometries. *Transactions of the American Mathematical Society*, 7(2):241–259, 1906. [see page 95]

[Viz64] Vadim G. Vizing. On an estimate of the chromatic class of a p-graph (in Russian). *Metody Diskretnogo Analiza*, 3:25–30, 1964. [see page 129]

[vK10] Marc van Kreveld. The quality ratio of RAC drawings and planar drawings of planar graphs. In Ulrik Brandes and Sabine Cornelsen, editors, *Proceedings of the 18th International Symposium on Graph Drawing (GD'10)*, volume 6502 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010. [see page 110]

[WB05] Daniel Wigdor and Ravin Balakrishnan. Empirical investigation into the effect of orientation on text readability in tabletop displays. In Hans Gellersen, Kjeld Schmidt, Michel Beaudouin-Lafon, and Wendy E. MacKay, editors, *Proceedings of the 9th European Conference Computer-Supported Cooperative Work (ECSCW'05)*, pages 205–224. Springer-Verlag, 2005. [see page 153]

[WC11] Yu-Shuen Wang and Ming-Te Chi. Focus+context metro maps. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2528–2535, 2011. [see page 28]

[Wer38] Max Wertheimer. Laws of organization in perceptual forms. In Willis D. Ellis, editor, *A Source Book of Gestalt Psychology*, pages 71–88. Routledge & Kegan Paul, London, 1938. [see page 146]

[WS] Alexander Wolff and Tycho Strijk. The Map-Labeling Bibliography (1996). Online: http://i11www.ira.uka.de/map-labeling/bibliography. [see page 150]

[WWKS01] Frank Wagner, Alexander Wolff, Vikas Kapoor, and Tycho Strijk. Three rules suffice for good label placement. *Algorithmica*, 30(2):334–349, 2001. [see page 150]

[YOT09] Daisuke Yamamoto, Shotaro Ozeki, and Naohisa Takahashi. Focus+glue+context: an improved fisheye approach for web map services. In Divyakant Agrawal, Walid G. Aref, Chang-Tien Lu, Mohamed F. Mokbel, Peter Scheuermann, Cyrus Shahabi, and Ouri Wolfson, editors, *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM-GIS'09)*, pages 101–110. ACM, 2009. [see page 145]

[ZR02] Alexander Zipf and Kai-Florian Richter. Using focus maps to ease map reading: Developing smart applications for mobile devices. *Künstliche Intelligenz*, 16(4):35–37, 2002. [see page 146]

The visualization of data is an important topic in computer science; with an increasing amount of data available, making data sets well-readable for users is a frequent task. A lot of data that contains connections between persons or objects can be represented as a graph. Hence, graph drawing, that is, the visualization of graphs, is a specialized research area.

This book covers research results in different areas of graph drawing. Its focus is on drawing metro maps and on labeling maps with external labels that are connected to the points of interest by lines. In both areas, the use of curves and the effects of crossings are investigated.

ISBN 978-3-95826-002-3

Würzburg University Press