Drawing Graphs for Cartographic Applications

Elena Mumford

Drawing Graphs for Cartographic Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 8 september 2008 om 16.00 uur

door

Elena Mumford

geboren te Slonim, Wit-Rusland

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.T. de Berg

Copromotor:
dr. B. Speckmann

Contents

ii

Chapter 1

Introduction

A graph is a simple but powerful abstraction used to model connectivity and relations between objects. A graph consists of nodes (vertices) and arcs (edges) that connect pairs of nodes. Graphs have applications in a wide variety of areas which includes automated cartography, chemistry, biology, VLSI design, and software engineering. They usually either represent an actual network or are used to describe the relations between entities. In automated cartography, for example, graphs represent different types of transportation networks, flow maps (maps that depict the flow of goods from one place in the world to another), and subdivisions (for instance, a subdivision of a map into countries can be seen as a graph, whose nodes are the points where more than 2 regions of the map meet). In chemistry graphs are used to model molecules—with vertices representing the atoms and the edges representing the chemical bonds between them. In biology the relations between different species can be modeled as a graph. In VLSI design the connections between the entities of an electronic chip are often represented by a graph.

For a graph to be able to serve its purpose as an information carrier it has to be visualized. Moreover, it has to be visualized such that it conveys the information in the best possible way. Graph drawing [2] is a branch of computer science that concentrates on developing algorithms for visualizing graphs. That is, graph drawers develop algorithms that produce drawings of graphs that satisfy certain aesthetic criteria. These criteria are often expressed in properties like edge complexity, number of edge crossings, angular resolution, number of edge directions, shapes of faces, graph symmetries, or the aspect ratio of a drawing. Some of these criteria are application-specific. For example, graphs that represent metro maps are usually drawn using a limited number of edge directions—see, for example, [40, 51, 54]. When drawing such a map one has to balance between drawing the map exactly as it looks in reality (which would produce a map cluttered with unnecessary details) and simplifying the map beyond recognition. Limiting the number of edge directions allows us to depict a metro map such that the map looks simple enough and is easy to follow, but still resembles the original layout of the network.

Other criteria are more general—they aim at improving the readability of any graph drawing. An example of such a criterion is the number of edge crossings of a graph. Unless crossings of a graph are part of the information conveyed by it—such as a crossing in a map representing a highway overpass—we would like to have as few crossings as possible, because edge crossings in general make a graph cluttered and unreadable. Another example of a general criterion is edge complexity—edges with too many bends are hard to follow. Most of the graph-drawing problems have to deal with satisfying a combination of criteria. For instance, in VLSI design the drawings are required to use two perpendicular directions for their edges and have as few edge bends and edge crossings as possible.

Problems in graph drawing are often of combinatorial and/or geometric nature, hence the methods for solving these problems come from a variety of areas such as computational geometry, graph theory, and (computational) topology.

In the remainder of this chapter we introduce the problems we consider in this thesis. We also give background and report related work for each of them. We start with a problem of representing a plane graph as a rectilinear cartogram in Section 1.1. Then we consider cased drawing optimization problems in Section 1.2. Finally in Section 1.3 we consider a problem in computational topology that deals with drawings of curves in the plane.

1.1   Drawing planar graphs: rectilinear duals with applications to cartograms

As we already mentioned edge crossings have a negative effect on the readability of a drawing of a graph. That is why plane graphs and planar graphs are especially popular with graph drawers. A *plane graph* is a graph that is drawn in the plane without crossings. A *planar graph* is a graph that can be drawn in the plane without edge crossings. In some application areas graphs are actually plane—for example, a graph presenting a subdivision of a map. Planarity testing has received a lot of attention in the graph theory and graph-drawing community (see, for example [41, 20, 67]). Every planar graph has a *straight line drawing* [30, 62, 66]—a crossing-free drawing where every edge is presented by a straight line segment. Thus when drawing a planar graph one has the freedom of concentrating on more "sophisticated" aesthetic criteria such as the shapes of the faces, graph symmetries, the size of the grid on which a graph can be drawn, or the aspect ratio of the drawing.

Traditionally the vertices of a graph are depicted as points, disks, or boxes and the edges as open Jordan arcs. However, sometimes alternative ways to represent graphs are used. Our first example of an alternative representation is the *visibility representation* of a graph. In the visibility representation the nodes of a graph are represented by horizontal segments such that each two vertices that are connected by an edge are "visible" to each other. More formally, the vertices of the graph are represented by horizontal segments (called *vertex-segments*) and the edges of the graph are represented by vertical segments (called *edge-segments*) in such a way that the following conditions hold: The vertex-segments do
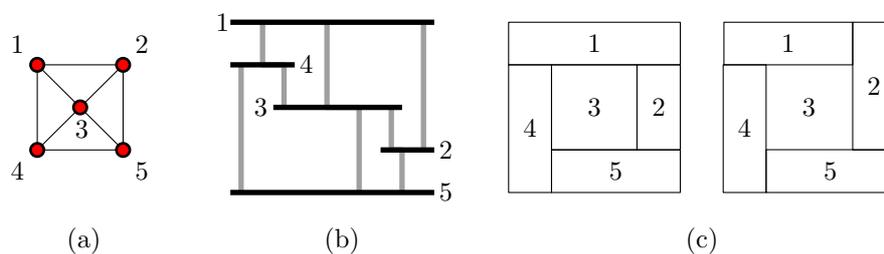
Figure 1.1 (a) a graph $\mathcal{G}$; (b) a visibility representation of $\mathcal{G}$; (c) a representation of $\mathcal{G}$ as a sliceable (left) and a non-sliceable (right) rectangular dual.

not overlap; the edge-segments do not overlap; and an edge-segment representing an edge (u, v) starts on the vertex-segment u, ends on the vertex-segment v, and does not intersect any other vertex-segment—see Figure 1.1(a,b) for an example.

Our next example is about plane triangulated graphs. These are graphs whose every face (except for maybe the outer face) is a triangle. Any plane graph can be turned into a triangulated graph by adding as many edges to it as possible while keeping it planar. We would like to present such a graph as rectangular dual. A rectangular dual of a graph $\mathcal{G}$ is a rectangular layout—a partition of a rectangle into rectangular regions—such that its dual graph is $\mathcal{G}$. Figure 1.1(c) presents a rectangular dual of the graph in Figure 1.1(a).

The question of rectangular dual existence for a plane graph has been extensively studied by the VLSI design community and has applications in computing floorplans of integrated circuits. A *floorplan* is a rectangular layout where each rectangle represents an entity of an integrated circuit. The adjacencies of the rectangles in such a layout are dictated by the connections between the blocks of the circuit. In 1985 Kozminski and Kinnen [45] and two years later Bhasker and Sahni [5] have shown that a plane triangulated graph with four vertices on the outer boundary admits a rectangular dual if and only if it does not contain separating triangles. A *separating triangle* is a 3-cycle in a graph that contains graph vertices inside it. One year later Bhasker and Sahni [6] developed a linear time algorithm for finding such a dual for a given graph. Eight years later a parallel $O(\log^2(n))$ time algorithm for finding such a dual has been developed by He [36] and two years after that another linear time algorithm was developed by He and Kant [43].

In this thesis we study a generalization of the rectangular-dual graph representation. That is, we would like to represent a plane triangulated graph as a rectilinear dual—a partition of a rectangle into a set of correctly adjacent rectilinear polygons. Additionally, our graph has positive weights assigned to its vertices. We would like to represent such a graph as a partition of a rectangle into rectilinear regions in such a way that (a) every region corresponds to exactly one vertex of the graph and vice versa; (b) no four regions meet in one point; (c) two regions share a boundary if and only if the corresponding nodes of the graph are adjacent; (d) the area of each region is equal to the weight of the corresponding vertex. Furthermore, we would like regions in such a layout to have as few vertices

as possible. A rectilinear layout itself can be seen as a rectilinear drawing of a cubic planar graph where the faces are the regions of the layout, the vertices are the points of the layout where three regions meet, and the edges are the sequences of horizontal and vertical segments of the boundaries of the regions. Then the problem can be formulated as constructing orthogonal drawings of planar graphs with prescribed face areas [57].

The idea for representing a graph in such a way has originated in automated cartography, in particular in cartogram construction, which we discuss in our next section.

1.1.1 Cartograms

Automated cartography [19] is a research area that concerns itself with visualizing geographical data with the aid of computers. The problems that fall into the scope of automated cartography include cartographic generalization, map overlay, label placement, map schematization, and specialized map construction. In this work we are interested in the type of specialized maps called a *cartogram*.

A *cartogram*, also known as *value-by-area map*, is a technique used by cartographers to visualize statistical data over a set of geographical regions like countries, states or counties. The regions of a cartogram are deformed such that the area of a region corresponds to a particular geographic variable [19]. The most common variable is population: In a population cartogram, the areas of the regions are proportional to their population. The shape of the regions depends on the type of cartogram.

There are different types of cartograms. The first type of cartogram is the *non-contiguous cartogram*. In a non-contiguous cartogram each region is shrunk such that the region preserves its original shape until its area has a required value. Then the regions are placed such that their relative positions are preserved and the regions do not touch each other. Often in order to improve the readability of the cartogram the regions are placed on top of the original map of the area. Another type of cartograms is the *Dorling cartogram* [23]. In a Dorling cartogram every region has the same shape. The shape is usually a circle,
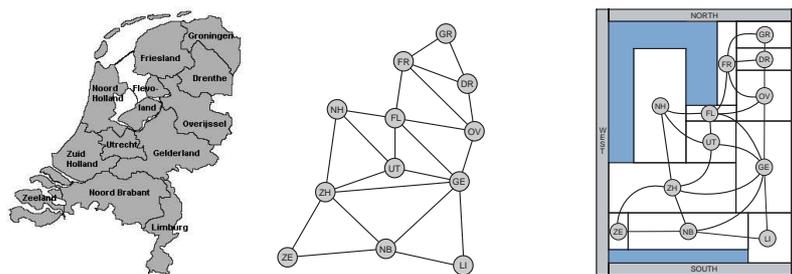
Figure 1.2 The provinces of the Netherlands, their adjacency graph, and a population cartogram—here additional "sea rectangles" were added to preserve the outer shape.

but can also be, for example, a square. The next type of cartograms is the *contiguous cartogram*. In a contiguous cartogram the regions are positioned such that they connected to each other without any gaps in between. The shapes of the regions are deformed, but we desire that they still resemble the original ones as much as possible.

Whether a cartogram is good is determined by how closely the cartogram resembles the original map and how precisely the area of its regions describe the associated values. More formally, the quality of a cartogram can be expressed using the following criteria: (a) correct adjacencies of the regions; (b) cartographic error; (c) relative positions of the regions; (d) how close the shape of a region in the cartogram resembles the shape of the corresponding region of the original map. The first criterion requires that the dual graph of the cartogram is the same as the dual graph of the original map. Here the *dual graph* of a map—also referred to as *adjacency graph*—is the graph that has one node per region and connects two regions if they are adjacent, where two regions are considered to be adjacent if they share a 1-dimensional part of their boundaries (see Figure 1.2). The second criterion, the *cartographic error* [42], is defined for each region as $|A_c - A_s| / A_s$, where A_c is the area of the region in the cartogram and A_s is the specified area of that region, given by the geographic variable to be shown. Relative positions of the cartogram regions should be not too different from the original relative positions of the regions in the map. Finally, we would like the regions to resemble the regions of the original map.

The first two chapters of this thesis are dedicated to a special type of contiguous cartograms, namely, rectilinear cartograms. Of particular relevance to rectilinear cartograms are *rectangular cartograms* introduced by Raisz in 1934 [58], which we discuss next.

Rectangular cartograms. A *rectangular cartogram* is a cartogram where each region is represented by a rectangle. This has the advantage that the areas (and thereby the associated values) of the regions can be easily estimated by visual inspection.

We consider the problem of construction a rectangular cartogram that has to satisfy only the first two criteria—cartographic error and correct adjacencies of the regions. From a graph-theoretic point of view, constructing rectangular cartograms with correct adjacencies and zero cartographic error translates to the following problem. We are given a plane graph $G = (V, E)$ (the dual graph of the original map) and a positive weight for each vertex (the required area of the region for that vertex). We would like to construct a rectangular dual of G where the area of each region is the weight of the corresponding vertex. We assume that the input graph G is plane and triangulated, except for possibly the outer face; this means that the original map did not have four or more countries whose boundaries share a common point and that G does not have inner nodes (i.e. nodes that do not belong to the outer face of the graph) of degree two. A graph that does contain inner nodes of degree two cannot be represented by a rectangular cartogram, since in a rectangular dual each rectangle that is not adjacent to the outer boundary of the layout needs to have at least four neighbors.

It is obvious that for a graph to be represented as a rectangular cartogram it is necessary
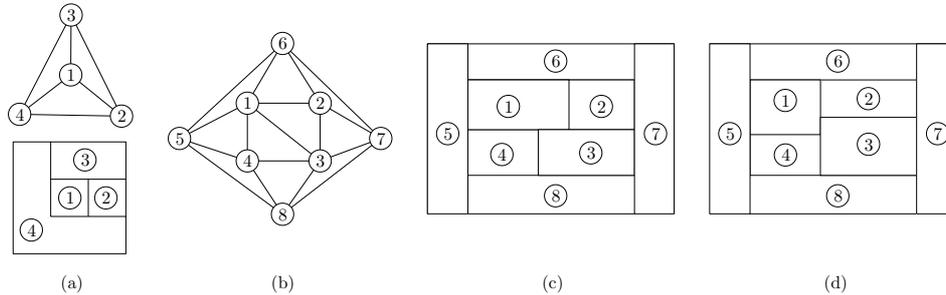
Figure 1.3 (a) a graph that does not admit a rectangular dual and its rectilinear dual; (b) the graph does have a rectangular dual but for certain weights no rectangular cartogram can be constructed; (c,d) two different rectangular duals for the graph in (b).

that it admits a rectangular dual. For example, the graph in Figure 1.3(a) does not have a rectangular dual and hence does not admit a rectangular cartogram for any set of weights assigned to its vertices. However, even when a graph does admit a rectangular dual it cannot always be represented as a rectangular cartogram. Consider, for instance, the graph in Figure 1.3(b). In any rectangular dual of that graph its four inner rectangles are arranged either as in Figure 1.3(c) or as in Figure 1.3(d). Thus if, for example, the weights of vertices 1 and 3 is 10 and the weights of vertices 2 and 4 is 100, then neither of its two duals can be turned into a rectangular cartogram.

The problem of deciding in polynomial time whether a graph admits a rectangular cartogram remains open. However, Van Kreveld and Speckmann [46] have shown that when a graph admits a special type of a rectangular dual, we can decide in polynomial time, whether that particular dual can be turned into a rectangular cartogram for a given set of weights. These special types are *sliceable* layouts and *L-shape destructible* layouts. A layout is *sliceable* if it can be obtained by recursively partitioning a rectangle by horizontal and vertical lines. An *L-shape destructible* layout is, basically, a layout from which we can remove its rectangles one by one in such a way that the remaining rectangles always form an L-shaped polygon.

When we cannot construct a rectangular cartogram with correct adjacencies and zero cartographic error, there are several ways to tackle this problem. One is to relax the strict requirements on the adjacencies and areas. For example, Van Kreveld and Speckmann [46] gave an algorithm that constructs rectangular cartograms that in practice have only a small cartographic error and mild disturbances of the adjacencies. Heilmann et al. [37] gave an algorithm that always produces regions with the correct areas; unfortunately the adjacencies can be disturbed badly. Another possibility is to use different shapes for the regions.

For example, we can allow the regions to be polygons of constant complexity. Such a cartogram can always be constructed using the algorithm by Thomassen [63]. However, we would like to consider a different generalization for rectangular cartograms, namely, *rectilinear cartograms*.
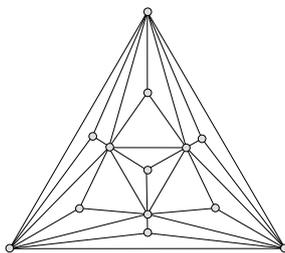
Figure 1.4 A graph that does not admit a rectilinear dual with less than eight vertices per region.

Rectilinear cartograms. A cartogram where each region is a rectilinear polygon is called a *rectilinear cartogram*. Thus a rectangular cartogram is special case of a rectilinear cartogram, where every region is a rectangle. In order to keep the cartogram as readable as possible we require the rectilinear regions to have as small complexity (i.e. number of vertices) as possible.

Just as in the case of rectangular cartograms we first need to make sure we can represent our graph G as a partition of a rectangle into a set of correctly adjacent rectilinear polygons. In other words, we need to find a *rectilinear dual*. A rectilinear dual of G is a partition L of a rectangle into rectilinear polygons such that (a) no four regions of L meet in one point; (b) each vertex of G corresponds to a region of L and vice versa; (c) two regions are adjacent if and only if the corresponding vertices of the graph are connected with an edge. Figure 1.3(a) depicts the smallest triangulated graph that does not admit a rectangular dual together with its rectilinear dual. Yeap and Sarrafzadeh [70] have shown that every plane triangulated graph has a rectilinear dual where every region has at most eight vertices. They have also shown that this bound is tight, that is there exists a graph—Figure 1.4—for which a rectilinear dual where every region has at most six vertices does not exist. Ten years after Yeap and Sarrafzadeh published their result, Liao *et al.* [12] demonstrated that every triangulated graph has a rectilinear dual where every region is represented by either a rectangle, an *L*-shape, or a *T*-shape, whereas rectilinear layouts created by the algorithm of [70] also contained *S*-shapes—see Figure 1.5. Moreover, the *L*- and *T*-shapes in the layouts created by the algorithm of Liao *et al.* are always oriented as shown in Figure 1.5, and every *T* shape can be decomposed into two rectangles. The latter is a very nice property, since the complexity of a rectilinear polygon is often measured in the number of rectangles it can be partitioned into.
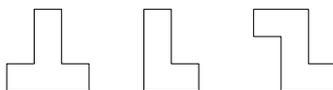
Figure 1.5 An *L*-shape, a *T*-shape, and an *S*-shape.

Previous work.   The problem of representing a graph as a rectilinear cartogram was studied by Rahman et al. [57] for so-called "good slicing graphs". Namely, for graphs that admit a sliceable dual $\mathcal{L}$ with the following property: for any horizontal slice line ℓ in $\mathcal{L}$ on one of the sides of ℓ there is exactly one region of $\mathcal{L}$ adjacent to it. See Figure 1.6 for an example. They have shown that by fixing the positions of some of the corners of the rectangles of such a layout one can give the regions correct areas by "bending" the edges of the rectangles. Each region in the resulting cartogram has at most eight vertices.
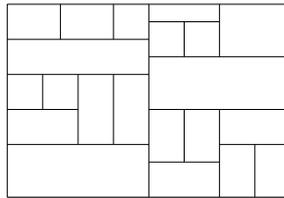
Figure 1.6 A "good slicing graph" [57].

Biedl and Genc [7] showed that it is NP-hard to decide whether a rectilinear cartogram that uses regions with at most 8 vertices exists for a given graph.

1.1.2   Our results

In Chapter 2 we present an $O(n \log n)$ algorithm that constructs a rectilinear cartogram for any triangulated vertex-weighted graph. The complexity of the cartogram is at most 12 for graphs that admit a sliceable dual, at most 20 for graphs that admit a rectangular dual, and at most 40 for all other plane-triangulated graphs. Chapter 3 describes how the algorithm can be used to construct a cartogram for a given geographical map and a set of weights for its regions. It also presents heuristics aiming at reducing the complexity of produced cartograms. We present experimental results that show that in practice the algorithm works a lot better than the provable complexity bounds tell us. Namely, our experiments have demonstrated that one can always construct a cartogram where the average number of vertices per region does not exceed five. Since a rectangle has four vertices, this means that most of the regions of our rectilinear cartograms are in fact rectangles. Moreover, the maximum complexity of the regions in such a cartogram never exceeds ten.

The results in Chapters 2 and 3 are joint work with Mark de Berg and Bettina Speckmann. The results in Chapters 2 appeared in the *Proceedings of the 13th International Symposium on Graph Drawing* in 2005 [17]. The full version of the paper has been accepted for publication in *Journal of Discrete Mathematics*. The results in Chapter 3 appeared in the *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems* in 2006 [18].

1.2 Drawing graphs with edge crossings: optimizing cased drawings

A non-planar graph cannot be drawn in the plane without edge crossings. Since edge crossings impede the readability of a graph, graph-drawing problems for non-planar graphs often aim at reducing the effect of edge crossings on the quality of a drawing of a graph. There are many ways to tackle the non-planarity. Maybe one of the most obvious ways is to try to draw a graph with as few crossings as possible. The lowest number of edge crossings that a drawing of a given graph can have is called the *crossing number* of the graph. Unfortunately, the problem of finding the crossing number of a graph is NP-complete [33] and remains so even for cubic graphs [38]. Another approach is to use a different representation of a graph. For example, we could separate a graph into planar subgraphs and draw each subgraph on a two-dimensional layer in three-dimensional space. The minimum number of layers necessary to draw a graph in such a way is called the *thickness* of the graph. Unfortunately, finding the thickness of a graph is an NP-complete problem as well [49]. Another interesting representation of a graph that targets non-planarity is a *confluent* drawing of a graph [21, 27]. Confluent drawings are also known as the drawings using *hierarchical edge bundles* [39] in the information-visualization community. In a confluent drawing the edges of a graph are merged into so-called "tracks" such that any pair of vertices of the graph that share an edge is connected by a smooth path in the drawing—see Figure 1.7.

Sometimes the output of a graph-drawing algorithm has to not only satisfy the given aesthetic criteria but also has to adhere some additional constraints. For example, the order of the edges around each vertex of a graph might be already given [54, 40], the edge lengths can be predefined [25, 9], the point set to which the vertex set has to be mapped might be fixed [44] or even the position of every vertex can be predefined [56, 8]. In that case some of the methods described above cannot be applied.

In Chapter 4 of this thesis we consider a graph drawing problem that deals with the problem of visualizing a graph whose (non-planar) drawing is fixed. In the following section we show that there is still some choices to be made when drawing such a graph. Namely, we show how one can produce better drawings by using so-called casings for the edge crossings of a graph and by carefully choosing which edge overpasses at each crossing.
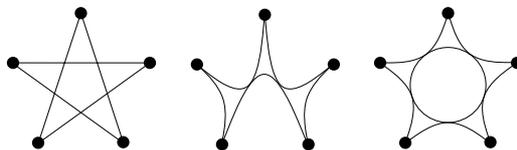
Figure 1.7 K_5 and its two confluent drawings [21].

1.2.1   Cased drawings

The vertices of a drawing are commonly marked with a disk, but it can still be difficult to differentiate between vertices and edge crossings in a dense graph. *Edge casing* is a well-known method—used, for example, in electrical drawings, when depicting knots, and, more generally, in information visualization—to alleviate this problem and to improve the readability of a drawing. A *cased drawing* orders the edges of each crossing and interrupts the lower edge in an appropriate neighborhood of the crossing. One can also envision that every edge is encased in a strip of the background color and that the casing of the upper edge covers the lower edge at the crossing. See Fig. 1.8 for an example. If there are no application-specific restrictions that dictate the order of the edges at each crossing, then we can in principle choose freely how to arrange them. However, certain orders will lead to a more readable drawing than others.

Quality of a drawing.   Globally speaking, two factors may influence the readability of a cased drawing in a negative way. Firstly, if there are many "switches" along an edge—the edge alternates being above and below other edges often—then it might become difficult to follow that edge. Drawings that have many switches can appear somewhat chaotic. Secondly, if an edge is frequently below other edges, then it might become hardly visible. We use these two considerations to formulate several optimization criteria that try to capture the concept of a "good" cased drawing. These optimization criteria include

(1) minimizing the maximum number of switches an edge can have,

(2) minimizing the total number of switches for all edges of the drawing,

(3) minimizing the maximum number of times an edge is covered by other edges,

(4) minimizing the maximum length of the portion of an edge that is covered with the casings around the edges that overpass it,

(5) maximizing the minimum distance between two consecutive underpasses of an edge,

(6) maximizing the total number of switches in the drawing.
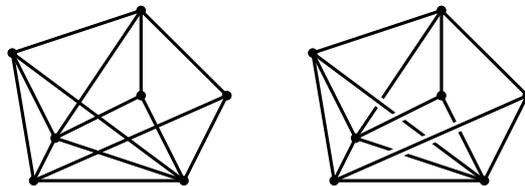
Figure 1.8 Normal and cased drawing of a graph.
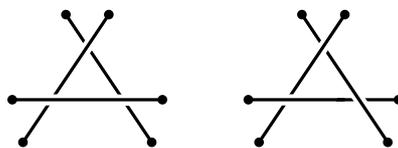
Figure 1.9 Stacking and weaving.

The last optimization problem actually aims at making the edges of a graph in the drawing harder to follow, which has possible applications in puzzle production for children.

Stacking and weaving.   When we turn a given drawing into a cased drawing, we need to define a drawing order for every edge crossing. We can choose to either establish a global top-to-bottom order on the edges, or to treat each edge crossing individually. We call the first option the *stacking model* and the second one the *weaving model*, since cyclic overlap of three or more edges can occur (see Fig. 1.9).

1.2.2   Our Results

We consider the optimization criteria for both stacking and weaving schemes. Further, we address the algorithmic question of how to turn a given drawing into an optimal cased drawing. For the weaving model we provide polynomial algorithms for minimizing and maximizing the total number of switches in a drawing. We also give a polynomial algorithm for maximizing the minimum distance between two consecutive underpasses of an edge. We show that minimizing the maximum length of the portion of an edge that is covered by casings is NP-hard. For the stacking model we provide polynomial time algorithms for constructing an optimal casing for problems (3)-(5). The remaining three problems remain open.

These results are joint work with David Eppstein, Marc van Kreveld, and Bettina Speckmann. The results have appeared in the *Proceedings of 10th Workshop on Algorithms and Data Structures* in 2007 [28]. The full version of the paper is accepted for publication in *Computational Geometry: Theory and Applications*.

1.3   Curve drawings: realizability of closed curves as surface boundaries

In the previous two sections of this chapter we considered two different graph-drawing problems. The problem we introduce in this section (and study in Chapter 5) is of a different nature. Namely, in this section we consider a problem in computational topology concerning drawings of closed curves in the plane, which we would like to test for being

projections of boundaries of certain special surfaces in 3D. When solving this problem we use the concept of *casings* that we introduced in our previous section. Next we define the problem more precisely.

We consider a problem concerning the relations between two types of topological objects: curves and special types of manifolds which we call *generalized terrains*. A *generalized terrain* is a surface embedded in space, in such a way that each point has a neighborhood such that every vertical line intersects that neighborhood at most once. A generalized terrain projects to an immersed surface in the plane, the boundary of which is a self-intersecting curve. We study under what circumstances we can reverse these mappings. Roughly speaking, we look at a 3D drawing of a non-intersecting curve from above. Based on what we see (namely, a 2D curve in the viewing plane) we would like to detect whether this curve is a boundary of some surface in three- or two-dimensional space. To be able to define the problem precisely we need to start with a few definitions.

1.3.1 Realizability of 2D drawings of curves as surface boundaries

A *surface* or *two-dimensional manifold with boundary* is a compact Hausdorff topological space M such that every point p has a neighborhood homeomorphic to a closed disk. If this homeomorphism maps p to a boundary point of the disk, we call p a *boundary point* of M; the set of boundary points is represented by ∂M. An *immersion* or *local homeomorphism* is a continuous function $i : M \to T$ that, restricted to some neighborhood of every point in M, is a homeomorphism. Here we will be concerned only with the case that $T = \mathbb{R}^2$, in which case we say that the surface M is *immersed in the plane*. If M is topologically a disk, we call i an *immersed disk*, but immersions of other types of manifold are also possible.

An *embedding* of a surface M into some space S is a closed subspace of S that is the image of M under a one-to-one continuous function $e : M \to S$, the inverse of which is also continuous. A *terrain* is a surface embedded in $\mathbb{R}^3$ such that every vertical line $\{(x,y,z) \mid x = c_1, y = c_2\}$ intersects it at most once. We are interested here in a localized version of this property: a *generalized terrain* is a surface M embedded in space $\mathbb{R}^3$ such that every point of M has a neighborhood the image of which is a terrain. Intuitively, such a surface is embedded in such a way that its inner points have no vertical tangent lines, so that it has a consistent up-down orientation at every point. We will also call such a
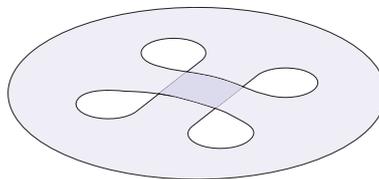
Figure 1.10 A generalized terrain, as viewed from above.

surface an *embedded surface*, when it does not introduce any confusion—see Figure 1.10 for an example. Intuitively, every generalized terrain can be constructed by gluing terrains along their boundaries. As with immersions, if M is topologically a disk, we call $e(M)$ an *embedded disk*.

If i is an immersion, $i(\partial M)$ is a curve in the plane, which we call the *boundary of the immersion*; with a suitable general-position assumption on i, this curve intersects itself only at proper pairwise crossings [50]. And if e is an embedding of a generalized terrain, we may project it into the plane to form an immersion: let $\pi_z(x, y, z) = (x, y)$, then $i(p) = \pi_z(e(p))$ is a local homeomorphism from M to $\mathbb{R}^2$. We are interested in the conditions under which these transformations can be reversed: if we are given an immersed surface, when is it the projection of a generalized terrain? If we are given a curve in the plane, when is it the boundary of an immersed surface, or of the projection of a generalized terrain?

The study of this subject goes back to a paper by Whitney [68] in 1937. In 1992 Shor and Van Wyk [60] first considered problems of this type from the algorithmic point of view. In our terminology, they showed that it is possible in polynomial time to determine whether a given curve (with proper pairwise crossings) is the boundary of an immersed disk. However, the possibilities of non-disk manifolds, of curves with multiple connected components, and of space embeddings as well as of plane immersions left many similar types of problems unsolved.

A *hole* in a surface M with an immersion i is a component C of the boundary of M such that $i(C)$ is a simple curve and such that i maps a neighborhood of C to the outside of $i(C)$. It is tempting to imagine that every immersed surface, and therefore every generalized terrain, must be topologically equivalent to a disk with holes, and that every immersed surface with a single boundary component must be topologically equivalent to a disk, but this is not true. For instance, Figure 1.11 shows a single curve that bounds an immersed surface which is topologically equivalent to a punctured torus.
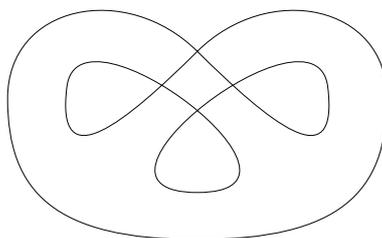
Figure 1.11   This curve is the boundary of a unique immersed surface, topologically equivalent to a punctured torus.
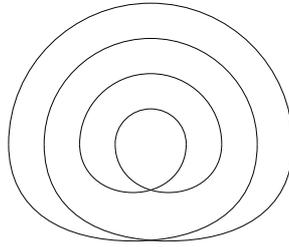
Figure 1.12 These two curves form the boundary of an immersed surface in two different ways, neither of which can be seen as a projection of a surface embedded in $\mathbb{R}^3$.

Although every generalized terrain projects to an immersed surface and every immersed surface has a self-intersecting curve as a boundary, this correspondence does not always work in the opposite direction. Some immersed surfaces cannot be lifted into space; Figure 1.12 shows an example of an immersed disk with a hole. Any embedding of this disk into $\mathbb{R}^3$ would intersect itself at some curve connecting the self-intersection points of its boundary curve. Some curves, such as the curve defined by the ∞ symbol, cannot be embedded as boundaries of any immersed surfaces. Even more problematically, some curves can be a boundary of more than one immersed surface. Bennequin in [4] has given an example of a curve (see Figure 1.13) that can be viewed as a boundary of an immersed disk in five different ways. Two of these ways involve a single central component with three lobes hanging off it symmetrically, while the other three have a shape that is more like a single spiral strip. The three spiral disks can be embedded into space but the two three-lobed disks cannot.
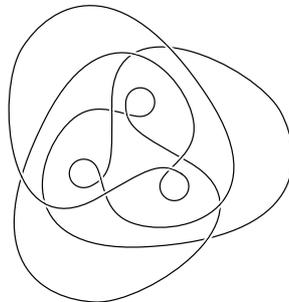
Figure 1.13 Bennequin's curve bounding five different immersed disks.

1.3.2 Our results

We show that it is NP-complete to determine whether an immersed disk is the projection of a surface embedded in space, or whether a curve is the boundary of an immersed surface in the plane that is not constrained to be a disk. However, when a casing is supplied

with a self-intersecting curve, describing which component of the curve lies above and which below at each crossing, we may determine in time linear in the number of crossings whether the cased curve forms the projected boundary of a surface in space. Furthermore, we show that the problem of testing whether an immersed surface is a projection of a generalized terrain is fixed-parameter tractable. The definition of the parameter in the algorithm is somewhat technical and can be found in Chapter 5.

These results are joint work with David Eppstein.

Chapter 2

Rectilinear duals for vertex weighted graphs

2.1 Introduction

Motivation. *Cartograms* is a technique used by cartographers to visualize statistical data about a set of regions like countries, states or counties. The regions of a cartogram are deformed such that the area of a region corresponds to a particular geographic variable [19]. The most common variable is population: In a population cartogram, the areas of the regions are proportional to their population.

In this chapter we restrict our attention to so-called *rectilinear cartograms*, which use rectilinear polygons as regions—see [19, 53] for some examples from the cartography community. Figure 2.1 shows an example of a rectilinear cartogram depicting the population of Europe.

Whether a cartogram is good is determined by several factors. In this chapter we focus on two important criteria, namely the correct adjacencies of the regions of the cartogram and the *cartographic error* [42]. The first criterion requires that the dual graph of the cartogram is the same as the dual graph of the original map. Here the *dual graph* of a map—also referred to as *adjacency graph*—is the graph that has one node per region and connects two regions if they are adjacent, where two regions are considered to be adjacent if they share a 1-dimensional part of their boundaries . The second criterion, the cartographic error, is defined for each region as $|A_c - A_s|/A_s$, where A_c is the area of the region in the cartogram and A_s is the specified area of that region, given by the geographic variable to be shown.

Figure 2.1 A rectilinear cartogram of Europe, theme population.

From a graph-theoretic point of view constructing rectilinear cartograms with correct adjacencies and zero cartographic error translates to the following problem. We are given a plane graph $G = (V, E)$ (the dual graph of the original map) and a positive weight for each vertex (the required area of the region for that vertex). Then we want to construct a partition of a rectangle into rectilinear regions whose dual graph is G—such a partition is called a *rectilinear dual* of G—and where the area of each region is the weight of the corresponding vertex.

We prove that any plane triangulated vertex-weighted graph admits a rectilinear cartogram with regions that have a constant complexity. Before we describe our results in more detail we first define the terms we use more precisely.

Terminology. A *layout* L is a partition of a rectangle R into a finite set of interior-disjoint regions. We consider only *rectilinear layouts*, where every region is a simple rectilinear polygon whose sides are parallel to the edges of R. We define the *complexity* of a rectilinear polygon as the total number of its vertices and the *complexity* of a rectilinear layout as the maximum complexity of any of its regions. A rectilinear layout is called *rectangular* if all its regions are rectangles. Thus, a rectangular layout is a rectilinear layout of complexity 4. Finally, a rectangular layout is called *sliceable* if it can be obtained by recursively slicing a rectangle by horizontal and vertical lines, which we call *slice lines*. (In computational geometry, such a recursive subdivision is called a (rectilinear) *binary space partition*, or *BSP* for short.)

We denote the dual graph (also called connectivity graph) of a layout L by $G(L)$. Given a graph G, a layout L such that $G = G(L)$ is called a *dual layout* (or simply a *dual*)
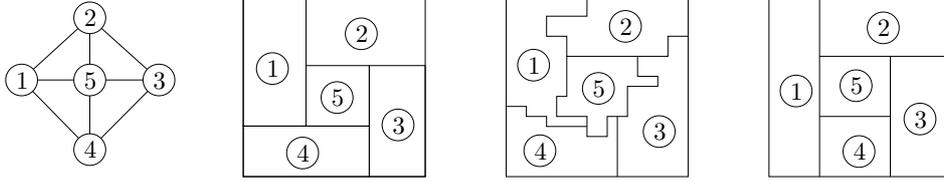
Figure 2.2 A graph $\mathcal{G}$ with a rectangular, rectilinear, and sliceable dual.

for $\mathcal{G}$. $\mathcal{G}(\mathcal{L})$ is unique for any layout $\mathcal{L}$. Every graph has a rectilinear dual layout, but not necessarily a rectangular dual layout. If it does have a rectangular dual, then the rectangular dual layout is not necessarily unique. Figure 2.2 depicts a graph and its three rectilinear duals, two of which (the first one and the last one in the figure) are rectangular.

Every vertex v of a vertex-weighted graph $\mathcal{G}$ has a positive weight $w(v)$ associated with it. Given a vertex-weighted plane graph $\mathcal{G}$ that admits a dual $\mathcal{L}$, we say that $\mathcal{L}$ is a *cartogram* if the area of each region of $\mathcal{L}$ is equal to the weight of the corresponding vertex of $\mathcal{G}$. The cartogram is called *rectangular (rectilinear, sliceable)* if the corresponding layout is rectangular (rectilinear, sliceable).

A *k*-cycle of a graph that has vertices both inside and outside of the cycle is called *separating*. A separating 3-cycle is called *separating triangle*.

Organization. In Section 2.2 we show how to construct a cartogram of complexity 12 for any vertex-weighted plane triangulated graph that has a sliceable dual. We extend our results in Sections 2.3 and 2.4 to general vertex-weighted plane triangulated graphs. Specifically, if a graph $\mathcal{G}$ admits a rectangular dual then we can construct a cartogram of complexity at most 20, otherwise we can still construct a cartogram of complexity at most 40. In Section 2.5 we analyze the running time of our algorithm and in Section 2.6 we discuss the characterization of graphs that admit a sliceable dual.

2.2   Graphs that admit a sliceable dual

Let $\mathcal{G} = (V, E)$ be a vertex-weighted plane triangulated graph with n vertices that admits a sliceable dual. The exact characterization of such graphs is still unknown, but Yeap and Sarrafzadeh [69] proved that every triangulated plane graph without separating triangles and without separating 4-cycles has a sliceable dual, which can be constructed in $O(n^2)$ time. Without loss of generality we assume that the vertex weights of $\mathcal{G}$ sum to 1, and that the rectangle R that we want to partition is the unit square.

Let $\mathcal{L}_1$ be a sliceable dual for $\mathcal{G}$. We scale and stretch $\mathcal{L}_1$ such that it becomes a partition of the unit square R—Figure 2.2 depicts an example of a graph $\mathcal{G}$ and its sliceable dual $\mathcal{L}_1$. We will transform $\mathcal{L}_1$ into a cartogram for $\mathcal{G}$ in three steps. In the first step we transform $\mathcal{L}_1$ into a layout $\mathcal{L}_2$—see Figure 2.4—where every region has the correct area.

In doing so, however, we may loose some of the adjacencies, that is, $\mathcal{L}_2$ may no longer be a dual layout for $\mathcal{G}$—for instance, the regions 2 and 7 in Figure 2.4 are not adjacent anymore. This is remedied in the second step, where we transform $\mathcal{L}_2$ into a layout $\mathcal{L}_3$—see Figure 2.5 for an example—whose dual is $\mathcal{G}$. In this step we re-introduce some errors in the areas. But these errors are small, and we can remove them in the third step, which produces the final cartogram, $\mathcal{L}_4$—see Figure 2.6. Below we describe each of these steps in more detail.

Step 1: Setting the areas correctly

The first step is relatively easy. Recall that a sliceable layout is a recursive partition of R into rectangles by vertical and horizontal slice lines. This recursive partition can be modelled as a BSP tree $\mathcal{T}$. Each node v of $\mathcal{T}$ corresponds to a rectangle $R(v) \subseteq R$ and the interior nodes additionally store a slice line $\ell(v)$. The rectangles $R(v)$ are defined recursively, as follows. We have $R(\text{root}(\mathcal{T})) = R$. Furthermore, $R(\text{leftchild}(v)) = R(v) \cap \ell^-(v)$ and $R(\text{rightchild}(v)) = R(v) \cap \ell^+(v)$, where $\ell^-(v)$ and $\ell^+(v)$ denote the half-space to the left and right of $\ell(v)$ (or, if $\ell(v)$ is horizontal, below and above $\ell(v)$). The rectangles $R(v)$ corresponding to the leaves are precisely the regions of the sliceable layout. See for example Figure 2.3—the shaded rectangle corresponds to the shaded node. The BSP tree for a sliceable layout is not necessarily unique, because different recursive partition processes may lead to the same layout.

The point where two or maximally three slice lines meet is called a *junction (point)*. We distinguish between T- and X-junctions. A T-junction involves two slice lines while an X-junction involves three slice lines, two of which are aligned. Note that when we start our layout we have T-junctions only. However, X-junctions might appear later when the layout goes through further steps of our algorithm, including the step we are describing in this section.

Now, let $\mathcal{T}$ be a BSP tree that models the sliceable layout $\mathcal{L}_1$. We will transform $\mathcal{L}_1$ into $\mathcal{L}_2$ by changing the coordinates of the slice lines used by $\mathcal{T}$ in a top-down manner. We maintain the following invariant: When we arrive at a node v in $\mathcal{T}$, the area of $R(v)$ is
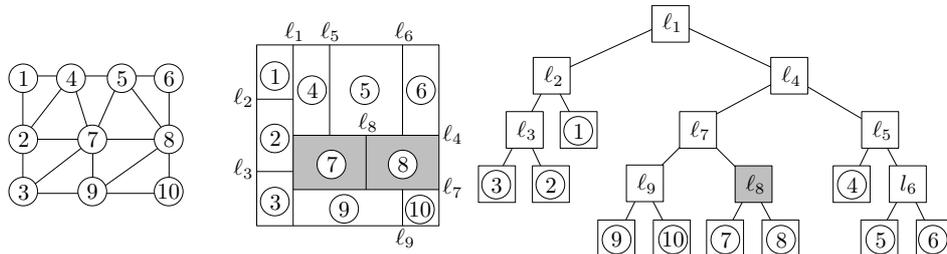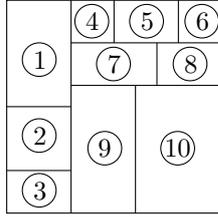

Figure 2.3 A graph $\mathcal{G}$, the layout $\mathcal{L}_1$, and the BSP tree $\mathcal{T}$.
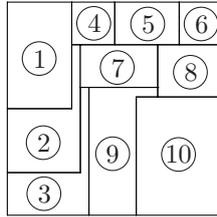
Figure 2.4 Layout $\mathcal{L}_2$. **Figure 2.5** Layout $\mathcal{L}_3$. **Figure 2.6** Layout $\mathcal{L}_4$.
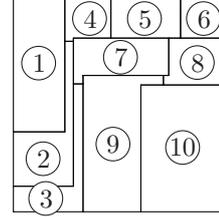
equal to the sum of the required areas of the regions represented by the leaves below v. Clearly this is true when we start the procedure at the root of $\mathcal{T}$. Now assume that we arrive at a node v which stores a slice line $\ell(v)$. We simply sum up all the required areas in the left subtree of v and adjust the position of the $\ell(v)$ in the unique way that assigns the correct areas to $R(\text{leftchild}(v))$ and $R(\text{rightchild}(v))$. When we reach a leaf there is nothing to do; the rectangle it represents now has the required area. See, for example, Figure 2.4 that shows the layout $\mathcal{L}_2$ for the example in Figure 2.3 and the weights $[w(1), \ldots, w(10)] = [0.15, 0.09, 0.06, 0.04, 0.06, 0.04, 0.08, 0.06, 0.18, 0.24]$.

Step 2: Setting the adjacencies right

The movement of the slice lines in Step 1 may have changed the adjacencies between the regions. To remedy this, we will use the BSP tree $\mathcal{T}$ again.

Before we start, we define two strips for each slice line $\ell(v)$. These strips are centered around $\ell(v)$ and are called the *tail strip* and the *shift strip*. The width of the tail strip is $2\varepsilon_v$ and the width of the shift strip is $2\delta_v$, where $\varepsilon_v < \delta_v$ and ε_v and δ_v are sufficiently small. The exact values of ε_v and δ_v will be specified in Step 3. At this point it is relevant only that we can choose them in such a way that the shift strips of two slice lines are disjoint except when two slice lines meet—see Figure 2.7 for an illustration.

We will make sure that the changes to the layout in Step 2 all occur within the tail strips and that the changes in Step 3 all occur within the shift strips. Due to the choice of the δ_v's all the junction points within the shift strip will lie on the slice line $\ell(v)$.

To restore the correct adjacencies, we traverse the BSP tree bottom-up. We maintain the invariant that after handling a node v, all adjacencies between regions inside $R(v)$ have been restored. Now suppose that we reach a node v. The invariant tells us that all adjacencies inside $R(\text{leftchild}(v))$ and $R(\text{rightchild}(v))$ have been restored. It remains to restore the correct adjacencies between regions on different sides of the slice line $\ell(v)$. We will describe how to restore the adjacencies for the case where $\ell(v)$ is vertical; horizontal slice lines are handled in a similar fashion, with the roles of the *x*- and *y*-coordinates exchanged.
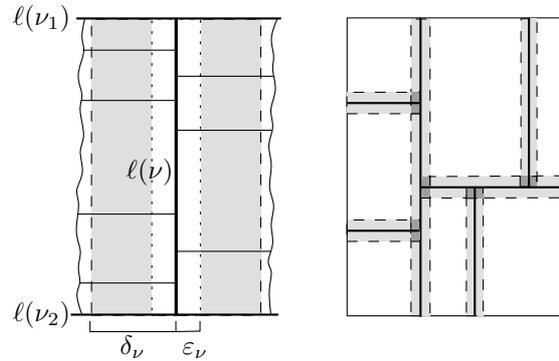
Figure 2.7 The shift and tail strips for ℓ_ν (left), ℓ_ν has two external junctions where it meets ℓ_{ν_1} and ℓ_{ν_2}, all other junctions are internal; the intersection pattern of shift-strips (right).

Let $A_1, A_2, \ldots, A_k$ be the set of regions inside $R(\nu)$ bordering $\ell(\nu)$ from the left, and let $B_1, B_2, \ldots, B_m$ be the set of regions inside $R(\nu)$ bordering $\ell(\nu)$ from the right. Both the A_i's and the B_j's are numbered from top to bottom—see Figure 2.8. We write $A_i \prec A_j$ to indicate that A_i is above A_j; thus $A_i \prec A_j$ if and only if $i < j$. The same notation is used for the B_j's. Now consider the tail strip centered around $\ell(\nu)$. All slice lines ending on $\ell(\nu)$ are straight lines within the tail strip (and, in fact, even within the shift strip). This is true before Step 2, but as we argue later, it is still true when we start to process $\ell(\nu)$.

In Step 1 (and when Step 2 was applied to $R(\text{leftchild}(\nu))$ and $R(\text{rightchild}(\nu))$), the slice lines separating the A_i's from each other and the slice lines separating the B_j's from each other may have shifted, thus disturbing the adjacencies between the A_i's and B_j's. For each A_i, we define $\text{top}(A_i) := B_k$ if B_k is the highest region (among the B_j's) adjacent to A_i in the original layout $\mathcal{L}_1$. Similarly, $\text{bottom}(A_i)$ is the lowest such region. This means that in $\mathcal{L}_1$, the region A_i was adjacent to all B_j with $\text{top}(A_i) \preceq B_j \preceq \text{bottom}(A_i)$. We restore
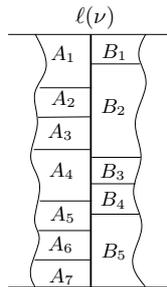
Figure 2.8 Left and right neighbors.

these adjacencies for A_i by adding at most two so-called *tails* to A_i, as described below. This is done from top to bottom: We first handle A_1, then A_2, and so on. During this process the slice line $\ell(v)$ will be deformed—it will no longer be a straight line, but it will become a rectilinear poly-line. However, the part of $\ell(v)$ bordering regions we still have to handle will be straight. More precisely, we maintain the following invariant: When we start to handle a region A_i, the part of $\ell(v)$ that lies below the bottom edge of top(A_i) is straight and the right borders of all $A_j \succeq A_i$ are collinear with that part of $\ell(v)$.

Next we describe how A_i is handled. There are two cases, which are not mutually exclusive: Zero, one, or both of them may apply. When both cases apply, we treat first (a) and then (b).

(a) If A_i is not adjacent to top(A_i) and top(A_i) is higher than A_i in $\mathcal{L}_2$ (that is, the layout after Step 1 before Step 2), then we add a tail from A_i to top(A_i). (If A_i is not adjacent to top(A_i) and top(A_i) is lower than A_i, then case (b) will automatically connect A_i to top(A_i).) More precisely, we add a rectangle to the right of A_i whose bottom edge is collinear with the bottom edge of A_i and whose top edge is contained in the bottom edge of top(A_i). The width of this rectangle is $\frac{\varepsilon_v}{n}$. Moreover, we shift the part of $\ell(v)$ below top(A_i) by $\frac{\varepsilon_v}{n}$ to the right. Observe that this will make all the B_j below top(A_i) smaller and all A_j below A_i larger—see the second picture in Figure 2.9.

Note, that the tail can be of positive (Figure 2.9), zero (Figure 2.10(a)-(b)) or negative (Figure 2.10(c)-(d)) length. The zero-tail occurs when the line along A_i's north border and the line along top(A_i)'s south border form an X-junction at the moment of handling—see Figure 2.10(a). The negative tail occurs when the line along A_i's north border and the line along top(A_i)'s south border formed an X-junction in $\mathcal{L}_2$, but the end of the line along A_i's north border moved up when we handed that line earlier in Step 2—see Figure 2.10(c).
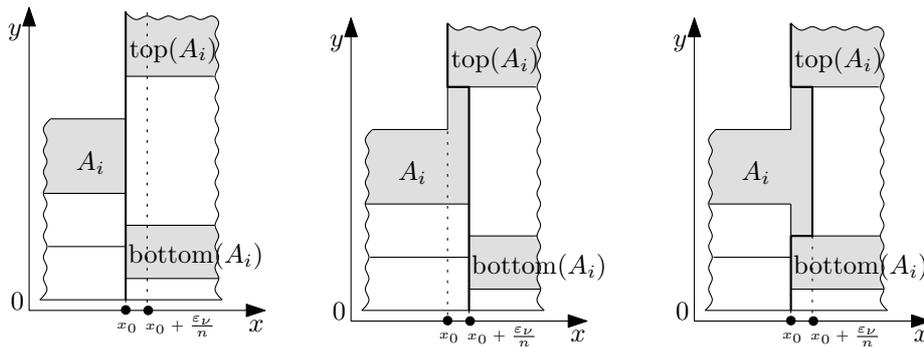
Figure 2.9 Both case (a) and case (b) apply.

(b) If A_i is not adjacent to bottom(A_i) and bottom(A_i) is lower than A_i in $\mathcal{L}_2$, then we
also add a tail, as follows. (If A_i was not adjacent to bottom(A_i) and bottom(A_i)
was higher than A_i, then necessarily case (a) has already been treated and in fact A_i
is now adjacent to bottom(A_i).) First, we shift the part of the slice line below the
top edge of bottom(A_i) by $\frac{\varepsilon_v}{n}$ to the left. Observe that this will enlarge bottom(A_i)
and all the B_j below it, and make all $A_j \succ A_i$ smaller. Next, we add a rectangle of
width $\frac{\varepsilon_v}{n}$ to A_i, which connects A_i to bottom(A_i). Its top edge is contained in the
bottom edge of A_i, its right edge is collinear to A_i's right edge, and its bottom edge
is contained in the top edge of bottom(A_i)—see the third picture in Figure 2.9.

Note that every tail "ends" on some B_j, that is, no tail extends all the way to the slice lines
on which $\ell(v)$ ends. This implies that

- (as we already claimed earlier) no bends are introduced inside the shift strips of the
  two slice lines on which $\ell(v)$ ends.

- the bordering sequence (the sets of countries along each side of a slice line and their
  order) of any other slice line remains unchanged.

Lemma 2.1 *The layout $\mathcal{L}_3$ obtained after Step 2 has the following properties:*

 (i) *If two regions are adjacent in $\mathcal{L}_1$, then they are also adjacent in $\mathcal{L}_3$.*

 (ii) *The tails that are added when handling a slice line ℓ all lie within the tail strip of ℓ.*

(iii) *Each region gets at most three tails.*
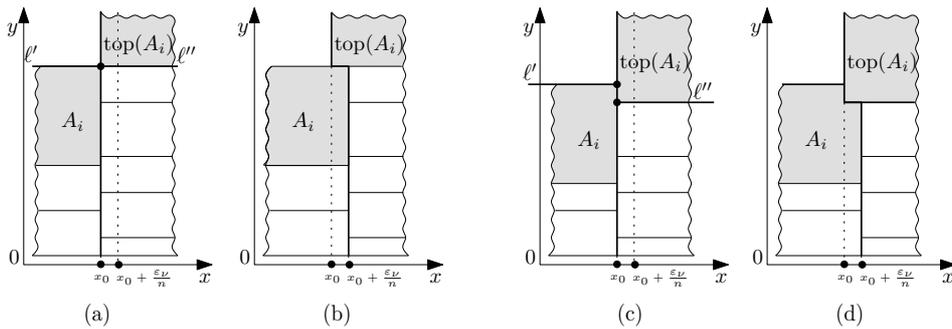
Figure 2.10 (a),(b) zero-tail up; (c), (d) negative tail up.
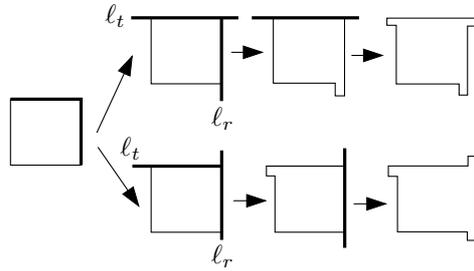
Figure 2.11 Tailing a region

Proof.

(i) It follows from the construction that each region A_i along a slice line $\ell(v)$ has the required adjacencies after $\ell(v)$ has been handled. Hence, the construction maintains the invariant that all adjacencies within $R(v)$ are restored after $\ell(v)$ has been handled. Therefore, after the slice line that is stored at the root of $\mathcal{T}$ is handled, all adjacencies have been restored.

(ii) A tail inside a tail strip of width $2\varepsilon_v$ has width $\frac{\varepsilon_v}{n}$ and is always adjacent to the current slice line. A slice line is shifted every time when a region grows a tail along it. Hence the slice line is shifted at most the number of tails that is grown along it. Next we analyze the maximum number of tails along a slice line.

Let k_1 be the number of regions growing one tail, and k_2 be the number of regions growing two tails. Let n_t be the number of tails. First of all, in order for the regions to have the need to grow any tails at all, there should be at least 2 regions on the opposite side of the line. Beside those two regions, for every region that grows 2 tails there is at least one region on the opposite side of the line, that is located between the regions the tails are reaching out for. Thus we have at least $k_2 + 2$ regions on the "non-growing-tails" side of the line. That means that the total number of regions on both sides of the line is at least $k_1 + 2k_2 + 2$, and at the same time it is at most n. The number of tails is $k_1 + 2k_2$. We have $k_1 + 2k_2 \leq n - 2$. Hence a line is shifted at most $n - 2$ times. Hence the tails lie within the tail strip, as claimed.

(iii) A region can get tails only when the slice line ℓ_r on its right or the slice line ℓ_t along its top are handled. Since a region must be either the topmost region along ℓ_r or the rightmost region along ℓ_t it can only get a double tail along one of these slice lines. Thus each region receives at most 3 tails. Note that since the tails along the same slice line are aligned, a region does not get more than three concave vertices (see Figure 2.11).

$\square$

Note that if $\mathcal{G}$ is triangulated then Lemma 2.1 (i) implies that two regions in $\mathcal{L}_3$ are adjacent if and only if they are adjacent in $\mathcal{L}_1$: All required adjacencies are present and in a plane triangulated graph there is no room for additional adjacencies.

The result of applying Step 2 to the layout of Figure 2.4 is shown in Figure 2.5.

Step 3: Repairing the areas

When we repaired the adjacencies in Step 2, we re-introduced some errors in the areas of the regions. We now set out to remedy this. In Step 2, the slice lines actually became rectilinear poly-lines. These poly-lines, which we will keep on calling slice lines for convenience, are monotone: A horizontal (resp. vertical) line intersects any vertical (resp. horizontal) slice line in a single point, a segment, or not at all. We will repair the areas by moving the slice lines in a top-down manner, similar to Step 1. But because we do not want to loose any adjacencies again, we have to be more careful in how we exactly move a slice line. This is described next.

Assume that we wish to move a horizontal slice line ℓ; vertical slice lines are treated in a similar manner. Let ℓ_1 and ℓ_2 be the slice lines to the left and to the right of ℓ, that is, the slice lines on which ℓ ends. We define a so-called *container* for ℓ, denoted by $C(\ell)$. The container $C(\ell)$ is a rectangle containing most of ℓ, as well as parts of the other slice lines ending on ℓ. Instead of moving the slice line ℓ we will move the container $C(\ell)$ and its complete contents.

We first define the container $C(\ell)$ more precisely. The top and bottom sides of $C(\ell)$ are contained in the boundary of the tail strip of ℓ. The position of the right side of $C(\ell)$ is determined by what happened at the junction between ℓ and ℓ_2 when ℓ_2 was processed during Step 2. Let A_i and A_{i+1} be the regions above and below ℓ and bordering ℓ_2, and let 2ε be the width of the tail strip of ℓ_2.

Next we describe the way the position of the right side of $C(\ell)$ is determined. There are two mutually exclusive cases depending on the configuration A_i and A_{i+1} have after Step 2.

(i) *A_i did not get a downward tail and A_{i+1} did not get an upward tail* (see Figure 2.12(a))

We set the right side of the container $C(\ell)$ to be collinear with the part of ℓ_2 lying within ℓ's shift strip (see Figure 2.12(a), (b)).

Note that there could be an extra junction on ℓ_2 within $C(\ell)$, formed by a line ℓ' that meets ℓ_2 from the other side—see Figure 2.12. Then moving $C(\ell)$ in the direction of that junction could move ℓ past the junction, thus creating a new adjacency and destroying an existing adjacency. In Figure 2.12, for instance, the adjacency between A_{i+1} and B_j is destroyed and the adjacency between A_i and B_{j+1} is created. We claim that this can only happen if ℓ_2, ℓ, and ℓ' formed an X-junction before Step 1. Indeed, suppose they did not form an X-junction. If they still do not form an X-junction after Step 1, then by definition of the tail-strip width, the junction
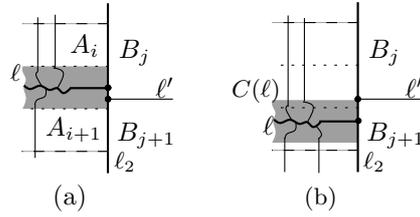
Figure 2.12   (a) ℓ and ℓ_2 form a T-junction and there is another junction on ℓ_2 within $C(\ell)$; (b) $C(\ell)$ is moved down and makes A_i adjacent to B_{j+1}.

of ℓ_2 and ℓ' is outside the tail strip of ℓ. If, on the other hand, they do form an X-junction after Step 1, then A_{i+1} would have received a zero-tail in Step 2. Hence, ℓ_2, ℓ, and ℓ' formed an X-junction before Step 1, as claimed. So if the input graph is triangulated, this situation in fact does not arise. (If the input graph was not triangulated, then moving ℓ past the junction does not destroy any required adjacency, it just replaces one diagonal in a 4-cycle by the other.)

(ii) *A_i got a downward tail or A_{i+1} got an upward tail.*
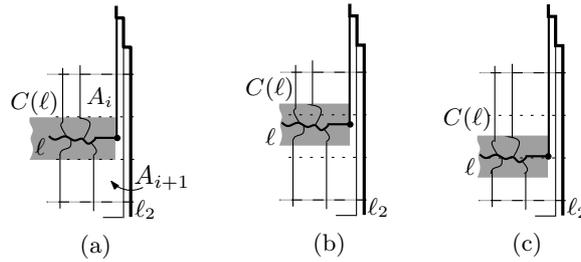
Figure 2.13 (a) A_{i+1} has an upward tail; (b) moving $C(\ell)$ up; (c) moving $C(\ell)$ down.

In this case the right side of $C(\ell)$ will go through the leftmost edge of the tail of $A_i(A_{i+1})$—see Figure 2.13.

Note that in this case more tails may have entered the tail strip of ℓ. For example, if A_{i+1} got an upward tail then some other regions below A_{i+1} possibly got an upward tail as well. Figures 2.14, 2.15 and 2.16 illustrate the case when ℓ and ℓ_2 were involved in an X-junction in $\mathcal{L}_1$—hence A_{i+1} could have a ("normal", zero- or negative) tail within ℓ's tail strip.

The position of the left side of $C(\ell)$ is determined in a similar fashion (with 2ε being the width of the tail strip of ℓ_1), as follows. Let B_j and B_{j+1} be the regions above and below ℓ and bordering ℓ_1. As before, we have are two mutually exclusive cases depending on the configuration B_j and B_{j+1} have after Step 2.
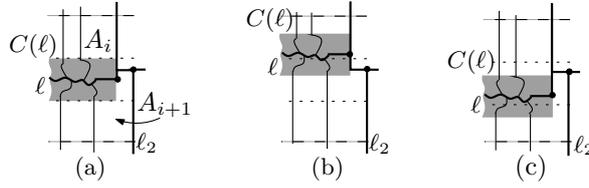
Figure 2.14   (a) A_{i+1} got an upward tail with its end inside ℓ's tail strip; (b) moving $C(\ell)$ up; (c) moving $C(\ell)$ down.
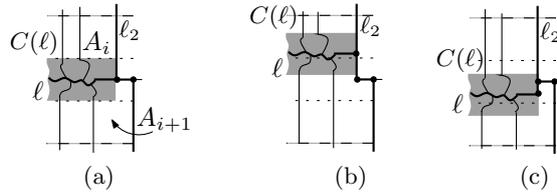
Figure 2.15   (a) A_{i+1} got an upward zero-tail; (b) moving $C(\ell)$ up; (c) moving $C(\ell)$ down.
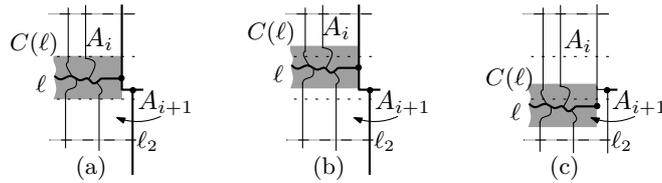
Figure 2.16 (a) A_{j+1} has a negative tail upward; (b) moving $C(\ell)$ up; (c) moving $C(\ell)$ down.
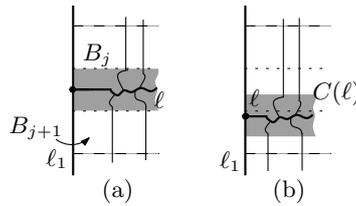
Figure 2.17 (a) ℓ and ℓ_1 form a T-junction; (b) $C(\ell)$ is moved down.

(i) *B_j did not become a destination of an upward tail and B_{j+1} did not become a destination of a downward tail.*

This situation is symmetric to case (ii) of the right side of $C(\ell)$—see Figure 2.17.

(ii) B_j *became the destination of a downward tail or* B_{j+1} *became the destination of an upward tail.*

The left side of $C(\ell)$ will go through the rightmost part of the slice line ℓ_1—see Figure 2.18.
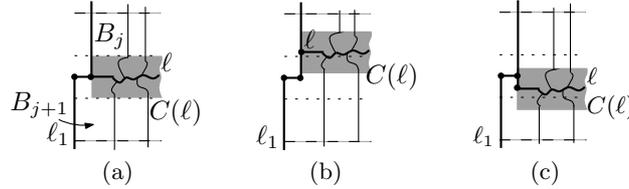
Figure 2.18 (a) B_{j+1} is a destination of a downward tail; (b) moving $C(\ell)$ up; (c) moving $C(\ell)$ down.

Note that in all cases the required adjacencies—that is, the adjacencies in $\mathcal{L}_1$—are preserved when $C(\ell)$ is moved.

Recall that we are repairing the areas in a top-down manner. When we get to slice line ℓ, we need to make sure that the total area to the left of ℓ—or rather the total area of the regions corresponding to the left subtree of the node corresponding to ℓ in the BSP tree—is correct. (Note that since the total area of the regions corresponding to the subtree rooted at the node ℓ is already correct, correcting the total area of the regions in the left subtree will automatically correct the total area of the regions corresponding to the right subtree.) We do this by moving the container $C(\ell)$. We will show below that the error we have to repair is so small that it can be repaired by moving $C(\ell)$ within the shift strip of ℓ. The parts of the slice lines ending on ℓ that are inside the shift strip and outside the tail strip are all straight segments; this follows from Lemma 2.1 (ii). Hence, when we move $C(\ell)$ we can simply shrink or stretch these segments, and the topology does not change.

We first analyze what happens to the complexity of the regions when we move the containers.

Lemma 2.2 *After Step 3 a region has at most 4 concave vertices in total.*

Proof. We might only "bend" a slice line ℓ, ending on slice lines ℓ_1 and ℓ_2, when moving its container $C(\ell)$. Thus we can introduce concave vertices to two regions adjacent to ℓ and ℓ_1 (ℓ_2), denoted above as B_j and B_{j+1} (A_i and A_{i+1}).

The shape of a region after Step 3 depends on the configuration and behavior of the four slice lines bounding it. The possible configurations are depicted in Figure 2.19. Here we present the complexity analysis of the region A where the lines around form the configuration depicted by Figure 2.19(i).

Step 3 only has effect on A when the lines around it are moved. We denote these lines $\ell_1, \ell_2, \ell_3,$ and ℓ_4 and by C_{ij} we denote the corner of A where the lines l_i and l_j meet,
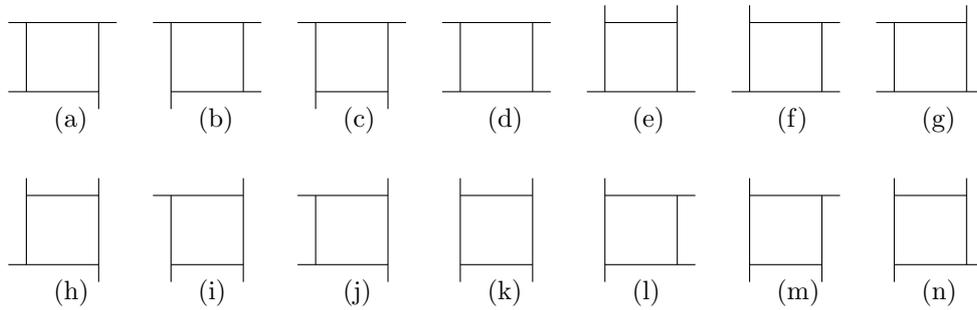
Figure 2.19 All possible configuration of slice lines surrounding a region.

$1 \leq i, j \leq 4$, $i \neq j$—see Fig 2.20. Step 2 might have given A tails grown at corners C_{14}, C_{24} and C_{13}. Every tail that A gained in Step 2 can only get longer or shorter after moving lines ℓ_1, ℓ_2, and ℓ_3 in Step 3 (see Figures 2.13, 2.14, and 2.16). Thus, every one of the three corners mentioned above that had a non-zero tail still contributes at most one concave vertex to the complexity of A after Step 3 (see Figures 2.13, 2.14, and 2.16). Every one of these three corners that had a zero-tail did not contribute a convex vertex after Step 2 but might contribute one after Step 3 (see Figure 2.15). Thus in total corners C_{14}, C_{24}, and C_{13} contribute at most 3 concave vertices after Step 3. As for the corner C_{23} it has not contributed any concave vertices in Step 2, but might have become a destination of a tail. In this case in Step 3 it might gain a concave vertex (see Figure 2.18). Thus in total after Step 3 the region A has at most 4 concave vertices, hence its complexity is at most 12.

Figure 2.21 depicts all possible shapes that a region can have after Step 3.

It is easy to verify—see Figures 2.12–2.18—that a similar kind of reasoning can be applied to prove that for all other configurations of junctions around a region the total number of concave vertices after Step 3 is bounded by four—at most one for each corner of the region in $\mathcal{L}_1$.   □
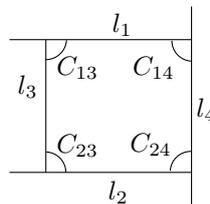
Figure 2.20 Lines and corners around a region.
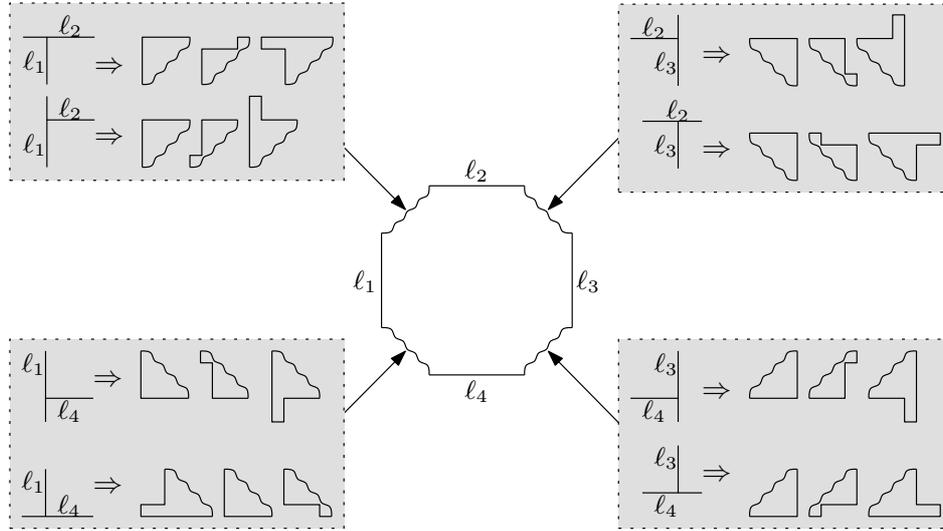
Figure 2.21 All possible shapes of a region after Step 3.

It remains to prove that we can choose the widths of the tail strip and shift strip appropriately. The two properties that we require are as follows.

Requirement 1 *The shift-strips of slice lines do not intersect if the slice lines do not form a junction after Step 1.*

Requirement 2 *The shift strip of each slice line ℓ is wide enough so that, when handling ℓ in Step 3, moving the container $C(\ell)$ can repair the areas while staying within the shift strip.*

For the first requirement it is sufficient to take the width of the shift strip to be smaller than $\Delta/2$, where $\Delta := \min(\Delta_x, \Delta_y)$ and Δ_x (Δ_y) is the minimum difference between any two distinct x-coordinates (y-coordinates) of the vertical (horizontal) slice lines after Step 1. As for the second requirement, we provide very rough upper bounds on the values for the width of the shift and tail strips, just to show that suitable values exist. Number the slice lines $\ell_1, \ldots \ell_{n-1}$ in the same order in which we handle them in Step 3. (For example, the slice line at the root of the BSP tree will be ℓ_1.)

Lemma 2.3 *If the width of the shift strip of slice line ℓ_k is set to $\delta_k := \Delta/4 * ((\Delta(1-\Delta))/10)^{n-k-1}$ and the width of the tail strip is set to $\varepsilon_k := \delta_k * \Delta/2$, for $1 \leq k \leq n-1$, then Requirements 1 and 2 are fulfilled.*

Proof. We have to prove that if the width of the shift strip of slice line ℓ_k is set to $\delta_k = \Delta/4 * ((\Delta(1-\Delta))/10)^{n-k-1}$ and the width of the tail strip is set to $\varepsilon_k = \delta_k * \Delta/2$, for

$1 \leq k \leq n-1$, then—when we move the container $C(\ell_k)$ of a slice line ℓ_k—there is enough area within its shift strip to compensate for the error introduced between its children. The proof is by induction on the slice line index.

Induction basis: ℓ_1**.** The error is introduced only when tailing during Step 2, because ℓ_1 is the first slice line handled in Step 3. Since the length of the slice line (here and later in the proof by the length of a slice line we mean its original length in $\mathcal{L}_1$) is 1, the error is less than half of the tail strip area, which is ε_1, and the available "maneuvering" area is $\delta_1 - \varepsilon_1 = (1 - \Delta/2)\delta_1$ which is clearly greater than ε_1.

Induction hypothesis: When the slice lines $\ell_1, ..., \ell_{k-1}$ were handled in Step 3, the container of each slice line was only moved within its shift strip.

Induction step: Consider a line ℓ_k at some node ν in the BSP. Let it be vertical. Denote by R_{left} and R_{right} the union of the regions in the left and right subtrees of ν. Consider the error for R_{right} (the error for R_{left} is the same with the sign reversed). Let ℓ_i, ℓ_j, and ℓ_m be the slice lines around R_{right}. They are higher up in the hierarchy. Hence $i, j, m < k$ and the lines have already been processed. This implies that the error induced by each of these lines is not more than the area of the shift strips they are in.
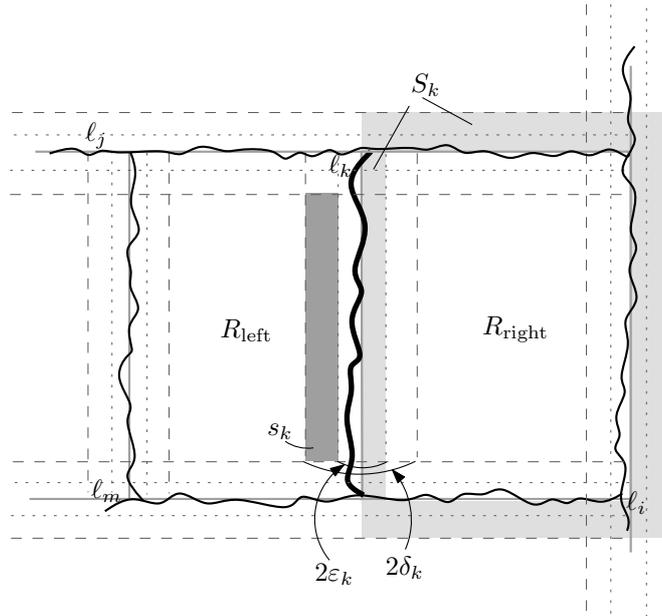
Figure 2.22 The common area of R_{left} and R_{right} is correct. s_k is the minimal area we have available to correct the error between R_{left} and R_{right}. The error itself is bounded from above by S_k.

The absolute value of $error(R_{\text{right}})$ is the sum of errors introduced by changes along ℓ_i, ℓ_j, and l_m (tailing in Step 2 and shifting in Step 3) and by tailing along ℓ_k itself. By the induction hypothesis $|error(R_{\text{right}})| < S_k$, where S_k is the sum of areas of shift strips of ℓ_i, ℓ_j and ℓ_m around R_{right} and the area of the part of ℓ_k's tails strip within R_{right}—see Figure 2.22. The area for "maneuvering" inside ℓ_k's shift strip is bounded from below by the area s_k of the part of the ℓ_k's shift strip on the left side of ℓ_k outside the tail strip between the shift strips of l_j and l_m—see Figure 2.22.

We need to show that $s_k - S_k > 0$. Since the lengths of the slice lines ℓ_i, ℓ_j, ℓ_m, and ℓ_k are less than 1, we have $S_k < \widetilde{S}_k$, where

$$\widetilde{S}_k = \delta_i + \delta_j + \delta_m + \varepsilon_k$$

and $s_k > \widetilde{s}_k$, where

$$\widetilde{s}_k = (\Delta - \delta_j - \delta_m) \cdot (\delta_k - \varepsilon_k)$$

Hence

$$s_k - S_k > \widetilde{s}_k - \widetilde{S}_k = (\Delta - \delta_j - \delta_m) \cdot (\delta_k - \varepsilon_k) - (\delta_i + \delta_j + \delta_m + \varepsilon_k)$$

Assume without loss of generality that $m > i$ and $m > j$, then $\delta_m = \max(\delta_i, \delta_j, \delta_m)$ and

$$\widetilde{s}_k - \widetilde{S}_k > (\Delta - 2\delta_m) \cdot (\delta_k - \varepsilon_k) - (3\delta_m + \varepsilon_k)$$

Substituting $\varepsilon_k = \Delta \delta_k / 2$ on the right side we get

$$
\begin{aligned}
\widetilde{s}_k - \widetilde{S}_k \quad &> \quad (\Delta - 2\delta_m)(1 - \Delta/2)\delta_k - 3\delta_m - \Delta\delta_k/2 \\
&= \quad \delta_k \cdot ((\Delta - 2\delta_m)(1 - \Delta/2) - 3\delta_m/\delta_k - \Delta/2) \\
&> \quad \delta_k(\Delta(1 - \Delta)/2 - 5\delta_m/\delta_k)
\end{aligned}
$$

Since $k > m$ and therefore $\delta_m/\delta_k = (\Delta(1-\Delta)/10)^{k-m} \le \Delta(1-\Delta)/10$ we have

$$
\begin{aligned}
\widetilde{s}_k - \widetilde{S}_k \quad &\ge \quad \delta_k(\Delta(1 - \Delta/2) - 5(\Delta(1-\Delta)/10)) \\
&= \quad 0
\end{aligned}
$$

Hence $s_k - S_k > 0$ as claimed.

$$\square$$

We conclude this section with the following theorem:

Theorem 2.4 *Let $\mathcal{G}$ be a vertex-weighted plane triangulated graph that admits a sliceable dual. Then $\mathcal{G}$ admits a rectilinear cartogram of complexity at most 12.*

Next we consider more general graphs, namely graphs that admit a rectangular dual and arbitrary triangulated plane graphs. These more general classes of graphs are handled by adding an extra step before the three steps described in the previous section.

We begin with graphs that admit a rectangular dual, that is, plane triangulated graphs without separating triangles.

2.3   Graphs that admit rectangular duals

Given a graph without separating triangles, its rectangular dual can be constructed, for example, by the algorithm of Kant and He [43]. Let now G be a plane triangulated graph without separating triangles and L_0 a rectangular dual of G. We construct a rectilinear BSP on L_0, that is, we recursively partition L_0 using horizontal or vertical splitting lines until each cell in the partitioning intersects a single rectangle from L_0. This can be done in such a way that each rectangle in L_0 is cut into at most four rectangles [15]. The resulting layout of these subrectangles, L_1, is sliceable by construction.

We then assign weights to the subrectangles. If a rectangle in L_0 representing a vertex v of G was cut into k subrectangles in L_1 then each subrectangle is assigned weight $w(v)/k$. (In practice it may be better to make the weight of each subrectangle proportional to its area. We discuss the subrectangle area assignment in more detail in our next chapter.) Next, we perform Step 1–3 of the previous section on the layout L_1 with these weights. Each rectilinear region in the layout L_4 obtained after Step 3 corresponds to a subrectangle in L_1. Finally, we merge the regions corresponding to subrectangles coming from the same rectangle in L_0—and, hence, from the same vertex of G—thus obtaining a layout L_5 with one region per vertex of G. The next lemma guarantees the correctness of our approach.

Lemma 2.5 *The algorithm described above produces a layout where each region has the correct area and adjacencies.*

Proof. This follows directly from the correctness of the algorithm of the previous section, except for one subtlety: In the previous section the layout L_1 only contained T-junctions— this is true since the input graph was triangulated—but this is no longer the case. L_1 is now obtained by cutting the layout L_0 with a BSP, which means that it can have X-junctions. Hence, the faces of the graph $G(L_1)$ dual to L_1 are not only triangles, but also 4-cycles— see Figure 2.23. Because the original layout L_0 does not have X-junctions, all X-junctions in L_1 are caused by edges of L_0 being cut by a splitting line of the BSP. Here we assume for simplicity and without loss of generality that no two vertical (resp. horizontal) splitting lines used by the BSP have the same x-coordinate (resp. y-coordinate). This means that of
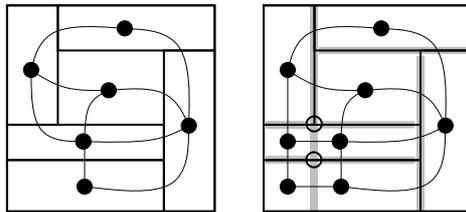
Figure   2.23   Cutting L_0 (left) with a BSP produces a layout L_1 (right) with two X-junctions (circled).

the four subrectangles incident to an X-junction, two neighboring ones must belong to the same rectangle R_1 in $\mathcal{L}_0$; the other two subrectangles belong to other rectangles R_2 and R_3, possibly with $R_2 = R_3$. Lemma 2.1 (i) states that after Step 2, all adjacencies present in $\mathcal{L}_1$ are also present in $\mathcal{L}_3$. In the proof we did not use that $\mathcal{G}(\mathcal{L}_1)$ only has triangular faces, so the statement is still true. However, if $\mathcal{G}(\mathcal{L}_1)$ also has faces that are 4-cycles, then Step 2 might introduce new adjacencies between opposite vertices of such a 4-cycle. Because, by construction, every 4-cycle has two neighboring vertices that correspond to the same rectangle in $\mathcal{L}_0$, these added adjacencies do not pose a problem. They simply represent an existing adjacency between two rectangles in $\mathcal{L}_0$ for the second time. Recall that in the cases depicted in Figure 2.12 and Figure 2.17 the adjacencies may have been changed in Step 3 if there was an X-junction. But by the same argument as above, this does not pose a problem. Finally, note that the adjacencies between the subrectangles that belong to the same rectangle in $\mathcal{L}_0$ ensure that the regions of $\mathcal{L}_5$ are connected. We conclude that the algorithm does indeed produce a valid layout with the correct adjacencies. It follows immediately from the construction that it also gives each region the correct area. □

It remains to analyze the complexity of the regions in the final layout. Of course we can just multiply the bound from the previous section by four, since each vertex in $\mathcal{G}$ is represented by four rectangles in $\mathcal{L}_1$. This results in a bound of 48. The next lemma shows that things are not quite that bad.

Lemma 2.6 *The algorithm described above produces regions of complexity at most 20.*

Proof. A region is cut into at most four subregions A, B, C, and D when a rectilinear BSP is constructed on the rectangular dual $\mathcal{L}_0$. (When a region is cut into less than 4 subregions its complexity after Steps 1–3 can only be smaller than the worst-case complexity of a region cut in four.) W.l.o.g. we can assume that the first cut (by the line ℓ_1) is vertical—see Figure 2.24.

The four regions jointly have 16 corners, which can be classified as follows. There are four *original corners* which correspond to the corners of P (marked with ■). The remaining corners are *induced corners* which can be further subdivided into four *internal corners* (marked with ×), four *horizontal side corners* (marked with ○), and four *vertical side corners* (marked with □).
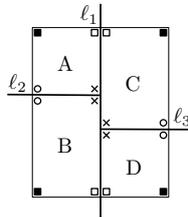
Figure 2.24 Corners.

We denote by *P* the region formed by the union of regions *A-D* after Step 3. The internal corners and any vertices they might have gained in Steps 1-3 lie inside *P* or on its edges and do not contribute to its complexity. In other words, they disappear when the subregions are merged. Hence, we only have to worry about what happens at the original and side corners.

According to the proof of Lemma 2.2 each original corner can gain one concave vertex in Steps 1-3 which implies that *P* has complexity at least $4 + 8 = 12$.
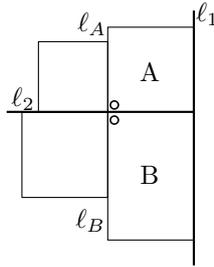
Figure 2.25 Horizontal side corners after Step 1.

Now consider the horizontal side corners of *A* and *B*. By construction of the BSP, the slice line ℓ_A along *A*'s west border and the slice line ℓ_B along *B*'s west border both end on ℓ_2. In other words, ℓ_2 extends further to the west than the horizontal side corners. (Indeed, a slice line that just connects two sides of the same rectangle is useless and will not be used in the BSP.) Hence, in Step 2 the horizontal side corners of *A* and *B* are tailed, if at all, when ℓ_2 is handled—see Figure 2.25. Next we argue that in fact there will be no tails to or from *A* and *B* at these side corners, and moreover they do not gain a concave vertex in Step 3. Before Step 1 the only north neighbor of *B* is *A*, and *A* and *B* are still adjacent after Step 1 because they both end on ℓ_1. This implies that *B* will not grow a tail in Step 2. Similarly, *B* is the only south neighbor of *A*, so no other region will grow a tail along ℓ_2 to reach *A*. This means that ℓ_2 must be straight at the horizontal side corners of *A* and *B*. It follows—see Figures 2.12 and 2.17—that no concave vertices are introduced in Step 3. Thus no extra vertices are introduced at the horizontal side corners of *A* and *B*. A similar argument applies to the horizontal side corners of *C* and *D*. Thus, the horizontal side corners contribute at most themselves to the list of vertices of *P*, hence at most 4 in total to *P*'s complexity, which is now $4 + 8 + 4 = 16$.

It remains to analyze what happens at the vertical side corners. If *A* and *C* have no neighbors along ℓ_1 but *B* and *D*, then only one of the vertical sides corners of *A* and *C* can get a tail during Step 2—see Figures 2.26(a)-(b) for an example. By construction of the BSP, *B* and *D* must have additional neighbors along ℓ_1. It follows from the analysis below that in this case the vertical side corners contribute a total of 4 vertices to *P*'s complexity which then becomes $4 + 8 + 4 + 4 = 20$. Similar reasoning applies to the case when *B* and *D* have no neighbors along ℓ_1 but *A* and *C*. In this case, at most one of the vertical side corners can be reached by a tail along ℓ_0—see Figure 2.26(c)-(d) for an example. Now assume
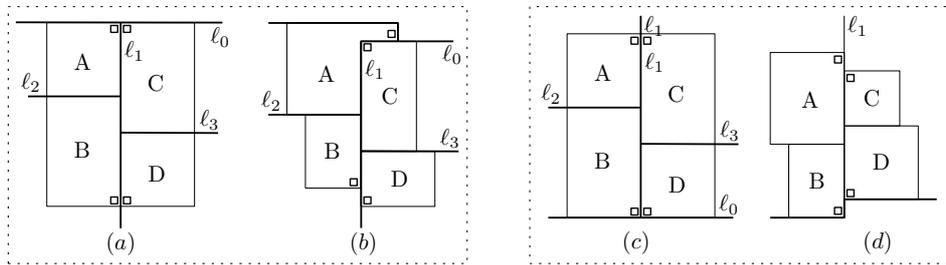
Figure 2.26 (a) Both *A* and *B* have only one neighbor along ℓ_1 ,(b) after Step 2 ; (c) Both *C* and *D* have only one neighbor along ℓ_1 , (d) after Step 2.

that all of *A–D* have neighbors along ℓ_1 that are not part of *P*. For what happens in Step 2, we resort to a complete case analysis.

We distinguish cases according to the location of the three corners *a*, *b*, and *c* of *C* and *D* with respect to the four intervals 1-4 on ℓ_1 which are defined by the corresponding corners of *A* and *B*—see Figure 2.27. We denote each case with a triple (a,b,c), where, for example, $a = 1$ denotes a case where the corner *a* is contained in interval 1. Only the cases when corners do not coincide with the endpoints of the intervals are depicted in Figure 2.28, and in every case the vertical side corners contribute 4 vertices to *P*'s complexity which becomes $4 + 8 + 4 + 4 = 20$, as claimed. The cases when one or more of the corners do coincide with the interval endpoints produce either the same or smaller complexity outer shapes.

Note that only the location of corners *a*, *b*, and *c* after Step 1 is relevant for *P*'s outer shape, the relation between ℓ_2 and ℓ_3 before Step 1 influences only the interior of *P* which has no impact on *P*'s complexity. All cases in Figure 2.28 are drawn for the situation where ℓ_2 is above ℓ_3. As an example, in Figure 2.29 we illustrate the interior of *P* for both relative positions of ℓ_2 and ℓ_3.
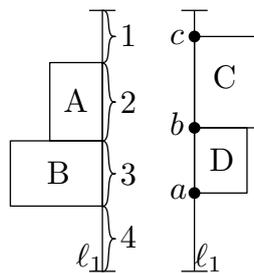
Figure 2.27 Case distinction.
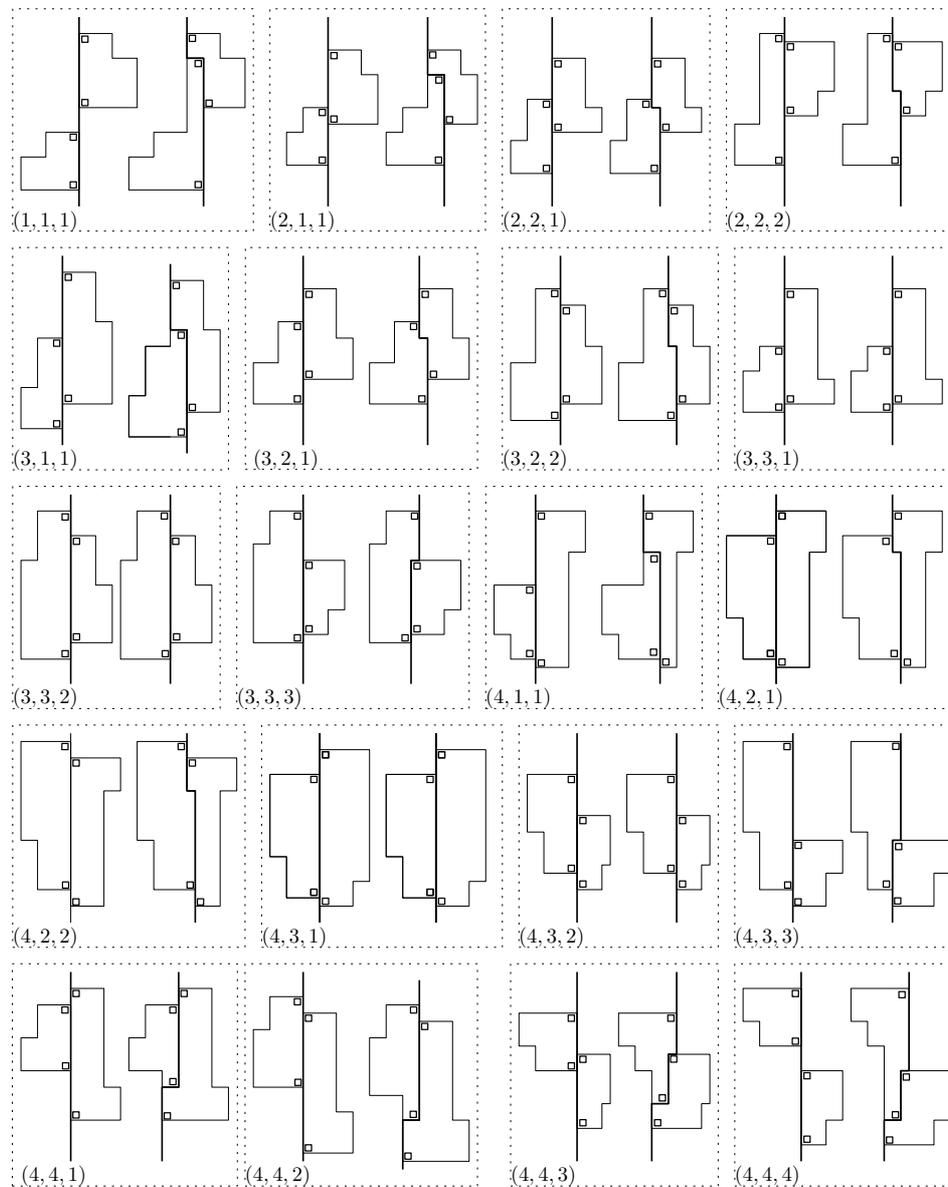
Figure 2.28 Corners charged to the vertical side vertices after Step 2.

It remains to argue that Step 3 does not introduce any extra vertices at the vertical side corners. This can be done by a careful inspection of each of the cases in Figure 2.28. For instance, consider case (1,1,1). Since only *A* and possibly *B* were adjacent to *C*, no

other tails end on *C*. Hence, ℓ_1 is straight near *C*'s vertical side corner, and the container of the slice line through *C*'s top edge extends all the way to ℓ_1. Hence, no extra vertices are introduced when that container is moved. Similarly, the container of the slice line through the bottom edge of *B* extends all the way to ℓ_1, so no extra vertices are introduced there either. Finally, it is easy to see that no extra vertices are introduced in Step 3 at the vertical side corners of *A* and *D*. Similar reasonings can be applied to all other cases in Figure 2.28; we omit easy but tedious details. Hence, the total complexity remains 20, as claimed. □
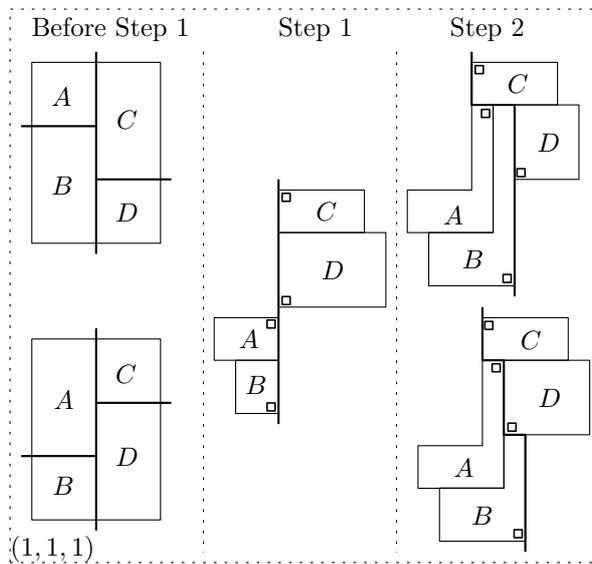
Figure 2.29 The interior of *P* depending on the relative position of ℓ_2 and ℓ_3 in case $(1,1,1)$.

The next theorem summarizes our result for graphs that admit a rectangular dual.

Theorem 2.7 *Let $\mathcal{G}$ be a vertex-weighted plane triangulated graph that admits a rectangular dual, i.e., $\mathcal{G}$ has no separating triangles. Then $\mathcal{G}$ admits a cartogram of complexity at most 20.*

We next show that for some graphs that admit a rectangular dual we can do better. Then in Section 2.4 we now turn our attention to general plane triangulated graphs.

2.3.1   Rectilinear cartograms for pseudo-sliceable layouts

As we already mentioned, a sliceable layout can be turned into a cartogram of complexity 12 for any values assigned to its regions. We would like to show that the same result can be achieved for a generalization of sliceable layouts called *pseudo-sliceable* layouts. Before we can define a pseudo-sliceable layout we need a few intermediate definitions. A *windmill-layout* is one of the two layouts depicted in Figure 2.30(a). A layout $\mathcal{L}$ that is obtained from a windmill-layout by replacing the middle-rectangle with some layout we call a *generalized windmill-layout*—see Figure 2.30(c) for an example. We call the sublayout in the middle the *heart* of the windmill-layout. The rectangle below the heart we call the *bottom wing* of the windmill layout. In a similar fashion we call the rectangles to the left, to the right and above the heart *left, right*, and *top wing* of the generalized windmill-layout. A *kinderdijk-layout* is a layout that can be obtained by replacing k rectangles in a sliceable layout with a windmill-layout for some $k \geq 0$—see Figure 2.30(b) for an example. [1] Thus any sliceable layout is a degenerate case of a kinderdijk-layout for $k = 0$.
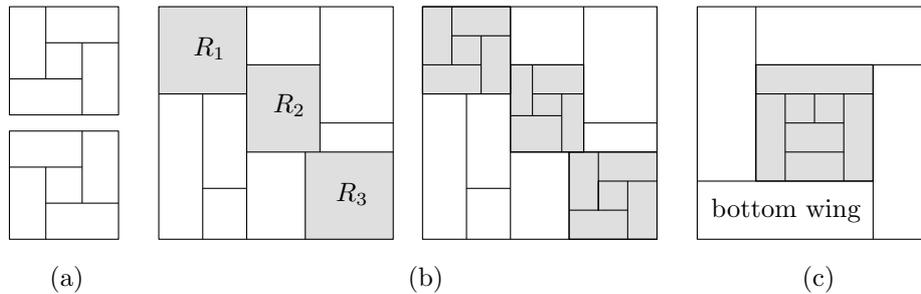
Figure 2.30 (a) Windmill-layouts; (b) a sliceable layout $\mathcal{L}$ and a kinderdijk-layout obtained from $\mathcal{L}$ by replacing rectangles R_1, R_2 and R_3 with windmill layouts; (c) a generalized windmill layout with its heart shaded gray.

A *pseudo-sliceable layout* is defined recursively as follows:

(i)   A kinderdijk-layout is pseudo-sliceable;

(ii) Let $\mathcal{L}_w$ be a windmill sublayout in a pseudo-sliceable layout $\mathcal{L}_1$. The layout $\mathcal{L}$ obtained from $\mathcal{L}$, by replacing the middle rectangle in $\mathcal{L}_w$ with a pseudo-sliceable layout $\mathcal{L}_2$, is pseudo-sliceable—see Figure 2.31 for an example.

We say that a graph $\mathcal{G}$ admits a *pseudo-sliceable dual* if it admits a rectangular layout $\mathcal{L}$ and $\mathcal{L}$ is pseudo-sliceable.

---

[1] The windmills of Kinderdijk are a popular tourist attraction in the Netherlands.
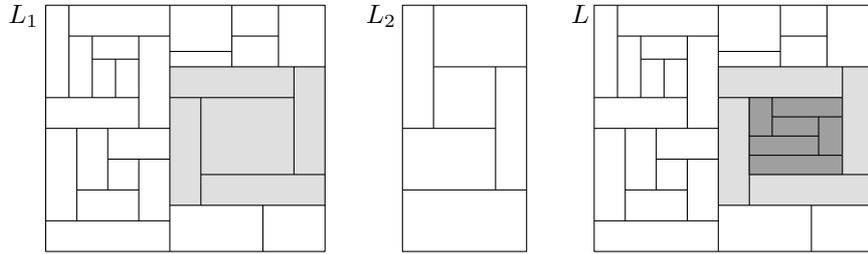
Figure 2.31 Pseudo-sliceable layouts $\mathcal{L}_1$ and $\mathcal{L}_2$ and a pseudo-sliceable layout $\mathcal{L}$ obtained by replacing the middle rectangle in the sublayout $\mathcal{L}_w$ of $\mathcal{L}_1$ (shaded gray) with $\mathcal{L}_2$.

Lemma 2.8 *Any vertex-weighted plane triangulated graph $\mathcal{G}$ that admits a pseudo-sliceable dual admits a cartogram of complexity at most 12.*

Proof. Let $\mathcal{L}_0$ be a pseudo-sliceable dual of $\mathcal{G}$, and let $\mathcal{G}$ contain at least two vertices. It follows from the definition of a pseudo-sliceable layout that $\mathcal{L}_0$ either has a sliceline that partitions $\mathcal{L}_0$ into two pseudo-sliceable sublayouts or it is a generalized windmill-layout. We are going to use this property to construct a very special BSP of $\mathcal{L}_0$ in the following way.

If $\mathcal{L}_0$ does have a slice, we partition $\mathcal{L}$ using that slice. Otherwise $\mathcal{L}_0$ is a generalized windmill. In that case let $\mathcal{L}_h$ be the heart of $\mathcal{L}_0$, let R_b be the bottom wing of $\mathcal{L}_0$, and let ℓ be the edge of the bounding rectangle of $\mathcal{L}_h$ that forms a T-junction with the top edge of R_b. We partition $\mathcal{L}$ into two two layouts by extending the edge ℓ until it reaches the bottom edge of R_b as depicted in Figure 2.32. The obtained sublayouts are pseudo-sliceable so we can repeat the procedure recursively. We denote the obtained sliceable layout by $\mathcal{L}_1$. Note that the only rectangles that get partitioned are the bottom wings of generalized windmill-layouts in $\mathcal{L}_0$. And, in particular, each such wing is partitioned exactly once. Next we reassign the weights of rectangles and perform the Steps 1-3 of Section 2.2 on $\mathcal{L}_1$ with these weights and merge the cut regions back together as we have done in Section 2.3 to obtain the final rectilinear cartogram $\mathcal{L}_4$.

According to the proof of Lemma 2.2, the regions of $\mathcal{L}$ that did not get partitioned by a sliceline become regions of complexity at most 12 in $\mathcal{L}_4$. Next we show that by tweaking Step 2 of the algorithm in Section 2.2 slightly, we can make sure that the regions in $\mathcal{L}_0$, that were halved by a line during the BSP construction (as described above) have complexity at most 12 in the final cartogram. Let R be a region in $\mathcal{L}_0$ that is partitioned into regions A and B in $\mathcal{L}_1$. This means that R is the bottom wing of some generalized windmill-layout $\mathcal{L}_w$, which is a sublayout in $\mathcal{L}_0$. We consider the case when the wings of the windmill layout are arranged as in the top layout of Figure 2.30(a). The other case is symmetric. Let $\mathcal{L}_h$ be the heart of $\mathcal{L}$ and let ℓ be the line that partitions R. Let ℓ_{right} be the slice line collinear with the right edge of R, and let ℓ_{btm} be the slice line collinear with
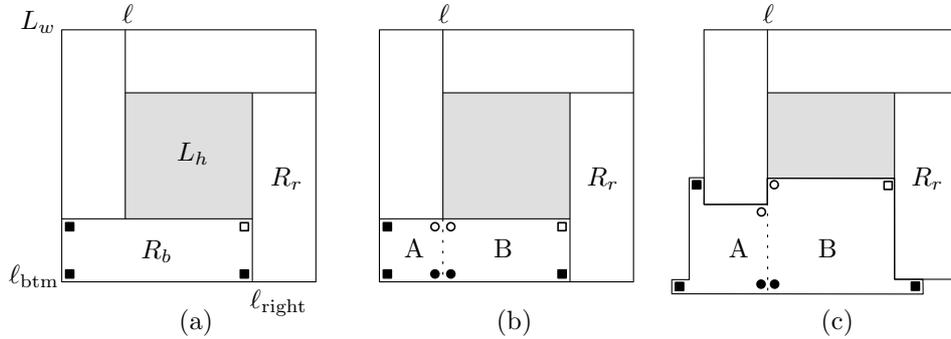
Figure 2.32 (a) Partitioning a generalized windmill-layout; (b) corners of the bottom wing; (c) a possible worst case complexity region in $\mathcal{L}_4$.

its bottom edge—see Figure 2.32(a) for an example. The regions A and B jointly have 8 corners. As in Section 2.3, we classify these corners into four *original corners* (marked with ■ or □) which correspond to the corners of R and four *induced corners* (marked with ○ and ●—see Figure 2.32(a),(b). Each of the three original corners marked with ■ in Figure 2.32(b) can gain a concave vertex after Steps 1-3 of the algorithm (by Lemma 2.2). However, the remaining original corner (marked with □) cannot. This corner can only get a concave vertex when the line ℓ_{right} is handled in Steps 2 and 3. Which only happens if the adjacency between B and its topmost neighbor along ℓ_{right} (that is the right wing R_r of $\mathcal{L}_w$) is broken during Step 1. This cannot occur, because B and R_r are the bottommost regions on each side of ℓ_r.

We subdivide the induced corners of A and B into *upper* (marked with ○) and *lower* (marked with ●) induced corners—see Figure 2.32(b). Consider the upper induced corners of A and B. The only chance to gain convex vertices they have is to do that when the line ℓ is handled in Steps 2 and 3. However, only region A needs to be adjacent to along ℓ is B (and vice versa). Since A and B are the bottommost regions on each side of ℓ, the adjacency between them is never broken during Step 1. This means none of the two upper corners are going to gain a concave vertex during Steps 2 and 3 by the proof of Lemma 2.2. Consider the lower induced corners. Here we are going to tweak the Step 2 of Section 2.2 in the following way. Note that the lower induced corners can only get convex vertices when the adjacencies of A and B are fixed along ℓ_{btm}. Also recall that A and B represent the same region of $\mathcal{L}_0$. When we handle the line ℓ_{btm} in Step 2 we treat A and B together, as one region. That is, if some region needs to reach B by growing a tail right along ℓ_{btm}, we let it grow that tail to A instead. Symmetrically, if some region needs to reach A by growing a tail left along ℓ_{btm}, we let it grow that tail to B instead. This way the bottom induced corners do not get any concave vertices and stay horizontally aligned. And, more importantly, the region R will be adjacent to everything it has to be adjacent along ℓ_{btm} in the final cartogram $\mathcal{L}_4$. To summarize: three of the original

corners contribute at most one concave vertex to the union of A and B in $\mathcal{L}_4$, plus one more concave vertex is contributed if the top edges of the top induced corners are not collinear after Step 1. The remaining original corner never gets a concave vertex and the T-junction formed by the bottom induced corners is not changed during the construction of the cartogram. Thus the total complexity of R in $\mathcal{L}_4$ is at most $4 \times 2 + 4 = 12$—see Figure 2.32 for an example. □

Since pseudo-rectangular layouts produce cartograms of nice complexity, we are interested to know when a graph admits a pseudo-sliceable dual. In particular, we would like to propose the following conjecture:

Conjecture 2.9 *Let $\mathcal{G}$ be a plane triangulated graph that admits a rectangular dual. Then $\mathcal{G}$ admits a pseudo-sliceable dual.*

Another useful property of a pseudo-sliceable layout is that it is *decidable*—that is, given a set of positive weights for its regions we can decide in polynomial time whether we can turn it into a rectangular cartogram for this set of weights. We are actually going to prove an even stronger property about pseudo-sliceable layouts, but before we do that let us introduce some definitions. Two rectangular layouts $\mathcal{L}$ and $\mathcal{L}'$ are said to be *equivalent* if (a) they have a common dual graph $\mathcal{G}$; (b) they correspond to the same regular edge labeling of $\mathcal{G}$. The latter means that for any edge (u,v) in $\mathcal{G}$ the pair of rectangles $R(u)$ and $R(v)$ in $\mathcal{L}$ has the same type of adjacency in $\mathcal{L}$ as the pair of rectangles $R'(u)$ and $R'(v)$ in $\mathcal{L}'$, that is, for example, if $R(u)$ is vertically above $R(v)$, then we have the same situation for $R'(u)$ and $R'(v)$ in $\mathcal{L}'$ and vice versa. We say that a rectangular layout $\mathcal{L}$ is *area-rigid* if for a given bounding rectangle R and any given set of positive weights for its faces there exists at most one rectangular layout $\mathcal{L}'$ such that (a) $\mathcal{L}'$ is equivalent to $\mathcal{L}$; (b) the bounding rectangle of $\mathcal{L}'$ is R; (c) the area of each region of $\mathcal{L}'$ is equal to the weight assigned to it.

We are going to use the following lemma to prove that any pseudo-sliceable layout is area-rigid. The proof follows directly from the definition above.

Lemma 2.10 *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two area-rigid layouts. The layout $\mathcal{L}$ obtained by replacing a rectangular region in $\mathcal{L}_1$ with $\mathcal{L}_2$ is area-rigid.*

Theorem 2.11 *A pseudo-sliceable layout is area-rigid and decidable.*

Proof. Van Kreveld and Speckmann [46] have shown that a sliceable layout and a windmill-layout are area-rigid and decidable. Thus by construction and by Lemma 2.10 any pseudo-sliceable layout is area-rigid and decidable. □

The same result is valid for a more general class of layouts, namely, *vortex-layouts*. A *vortex-layout* is recursively defined as follows: (a) a kinderdijk layout is a vortex-layout; (b) a layout obtained from a vortex-layout by replacing one of its rectangles with another vortex-layout is a vortex-layout.

Corollary 2.12 *A vortex-layout is area-rigid and decidable.*

We conclude with the following conjecture.

Conjecture 2.13 *Any rectangular layout is area-rigid.*

2.4   General plane triangulated graphs

As mentioned earlier, Liao et al. [12] showed that any plane triangulated graph has a rectilinear dual that uses L- and T-shapes—that is, regions of maximal complexity 8—in addition to rectangles. We cut each region into at most two rectangles and then proceed as in the previous case: We cut the collection of rectangles with a BSP to obtain a sliceable layout $\mathcal{L}_1$, we assign weights to the rectangles in $\mathcal{L}_1$, run Steps 1–3, and merge regions belonging to the same vertex in $\mathcal{G}$. This leads to the following result.

Theorem 2.14 *Any vertex-weighted plane triangulated graph $\mathcal{G}$ admits a cartogram of complexity at most 40.*

Proof. We preprocess the rectilinear layout created by the algorithm of Liao et al. [12] by cutting each T-shaped region A (L-shapes can be considered to be degenerated T-shapes) into two rectangles A_1 and A_2 as it is shown in Figure 2.33(a). This is always possible because the algorithm presented in [12] always produces layouts where each T-shaped region has the following properties

(i) it is oriented as a letter "T" written upside down

(ii) the heights of its 2 horizontal branches are identical.
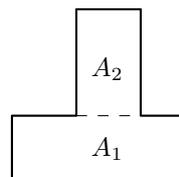
Figure 2.33 Cutting the T-shape.

Each of the regions A_1 and A_2 after Step 3 has at most 20 vertices. Hence the complexity of their union is at most 40.   □

2.5 Runtime analysis

Our algorithm uses three data structures. The first one stores the input graph G. The second one stores the adjacency graph $G(\mathcal{L}_1)$ of $\mathcal{L}_1$, that is, the adjacency graph of the sliceable subdivision which we obtain after constructing a BSP on the rectangular dual $\mathcal{L}_0$ of G. The third structure represents the layout that is being transformed into the cartogram. G and $G(\mathcal{L}_1)$ are stored as a set of vertices, where with each vertex we store four lists of pointers that refer to its north, east, south and west neighbors. Furthermore, each vertex v of G stores pointers to the at most 12 nodes in $G(\mathcal{L}_1)$ that correspond to v. Also, each vertex y of $G(\mathcal{L}_1)$ has a pointer to the corresponding region in the layout. The layout itself is stored in a BSP tree $\mathcal{T}$, as described in Section 2. Recall that each internal node of $\mathcal{T}$ stores a slice line. With each slice line ℓ we also store two sorted lists of regions that are bordering ℓ from the left and the right (or the bottom and the top), respectively. Each leaf of $\mathcal{T}$ contains a pointer to the corresponding node in $G(\mathcal{L}_1)$.

Computing a rectangular dual $\mathcal{L}_0$ of G takes linear time [43]. If G does not allow a rectangular dual, then we compute a rectilinear dual using L- and T-shapes in addition to rectangles, also in linear time [12]. We cut each region into at most two rectangles to construct a rectangular subdivision $\mathcal{L}_0$. Constructing a BSP on $\mathcal{L}_0$ takes $O(n \log n)$ time, using the algorithm by d'Amore and Franciosa [15]. At the same time we can also construct $G(\mathcal{L}_1)$ at no additional cost since each region is split at most three times. The lists of adjacent regions for each slice line ℓ in $\mathcal{T}$ can be created by traversing $\mathcal{T}$ bottom up in linear time.

In Step 1 we first calculate the weight for each internal node, traversing $\mathcal{T}$ bottom up, and then move each slice line to meet the weight requirements, traversing $\mathcal{T}$ top down. Both weight calculation and moving the lines takes linear time in total. Step 2 takes $O(k)$ time per slice line ℓ, where k is the number of ℓ's neighbors. Since each region is a neighbor of at most 4 lines, updating all lines takes linear time in total. Moving the slice lines in Step 3 also takes linear time. Finally, combining the at most 12 regions in the cartogram that correspond to a vertex v of G can be done in linear time as well.

Theorem 2.15 *Any vertex-weighted plane triangulated graph G with n vertices admits a cartogram of complexity at most 40, which can be constructed in $O(n \log n)$ time.*

Recall that for graphs that admit a sliceable or pseudo-sliceable dual we can construct a cartogram of much smaller complexity than in the general case. Hence it is very interesting and important to be able to decide when graph admits such a dual. The characterization of such graphs remains an open question. Testing a graph for sliceability in polynomial time is an open problem as well. In the next section we discuss a possible approach to arrive at a characterization of graphs that admit a sliceable dual.

2.6   Characterization of graphs that admit a sliceable dual

Thanks to Koźmiński and Kinnen [45] we have a characterization of graphs that admit a rectangular dual. The characterization of the class of graphs that admit a sliceable dual remains open. In this section we address some partial results in this area and discuss some ideas about how we think this problem can be approached. We start with some terminology.

Terminology. As we mentioned before, any plane triangulated graph without separating triangles and with four vertices on the outer boundary admits a rectangular dual. The class of graphs admitting a rectangular dual is slightly more general. Namely (see [45]), any triangulated graph that can be extended with four additional vertices placed on the outer face of G to form a new graph $E(G)$ such that

(a) the additional vertices form the boundary of the outer face of $E(G)$;

(b) $E(G)$ is a triangulated graph (except for the outer face) without separating triangles.

These four additional vertices are labeled $b(G)$, $t(G)$, $l(G)$, and $r(G)$ to correspond to four rectangles that bound a rectangular dual of G from below, above, left, and right. $E(G)$ is called an *extended graph* of G. The rectangular dual of G is obtained by removing the rectangles corresponding to the external vertices from the dual of $E(G)$.

A vertex of G that is connected to more than one vertex in $\{b(G), t(G), l(G), r(G)\}$ is called a *corner vertex* because it corresponds to a rectangle in a corner of a dual layout of G. The choice of corner vertices uniquely defines the extended graph $E(G)$ for G and vice versa, hence $E(G)$ is often referred to as a *corner assignment* of G. A corner assignment $E(G)$ of a graph G such that $E(G)$ admits a rectangular dual is called *valid*. We say that $\mathcal{L}$ is a *dual layout for G with corner assignment $E(G)$* if the vertices adjacent to more than one vertex of the outer cycle of $E(G)$ are exactly the dual vertices of the corner rectangles of $\mathcal{L}$. Figure 2.34(a) shows an example of a graph G, its extended graph $E(G)$ (Figure 2.34(c)) and the corresponding rectangular dual (Figure 2.34(d)). The corner vertices of G are shaded.

Any rectangular layout containing at least two rectangles has at least two and at most four rectangles occupying its four corners. Hence, for any corner assignment a graph G has at least two and at most four corner vertices. The corner vertices naturally decompose the outer cycle of G into four *outer paths* $p_{\text{left}}(G)$, $p_{\text{top}}(G)$, $p_{\text{right}}(G)$, and $p_{\text{btm}}(G)$. Each path starts at one corner vertex and ends at the next one. Namely, the vertices adjacent to $l(G)$ form the path $p_{\text{left}}(G)$, the vertices adjacent to $t(G)$ form $p_{\text{top}}(G)$, the vertices adjacent to $r(G)$ form $p_{\text{right}}(G)$, and the vertices adjacent to $b(G)$ form $p_{\text{btm}}(G)$). These paths represent the rectangles of a rectangular dual of G that are adjacent to left, top, right and bottom sides of the boundary rectangle of the dual. In Figure 2.34(c) the path $p_{\text{top}}(G)$ is indicated by a dotted line.

For a graph to have a rectangular dual any vertex of degree two has to become a corner vertex, otherwise the extended graph would gain a separating triangle—see Figure 2.34,

where the vertex labeled 6 in G is connected to only one external vertex in graph (b). As a result its degree in the extended graph is only three which means that it lies inside a separating triangle (shaded gray).

Thus any triangulated (except for the outer face) graph G without separating triangles and at most four vertices of degree two admits a rectangular dual. A plane triangulated graph that admits a rectangular dual is called *sliceable* if it admits a sliceable dual.
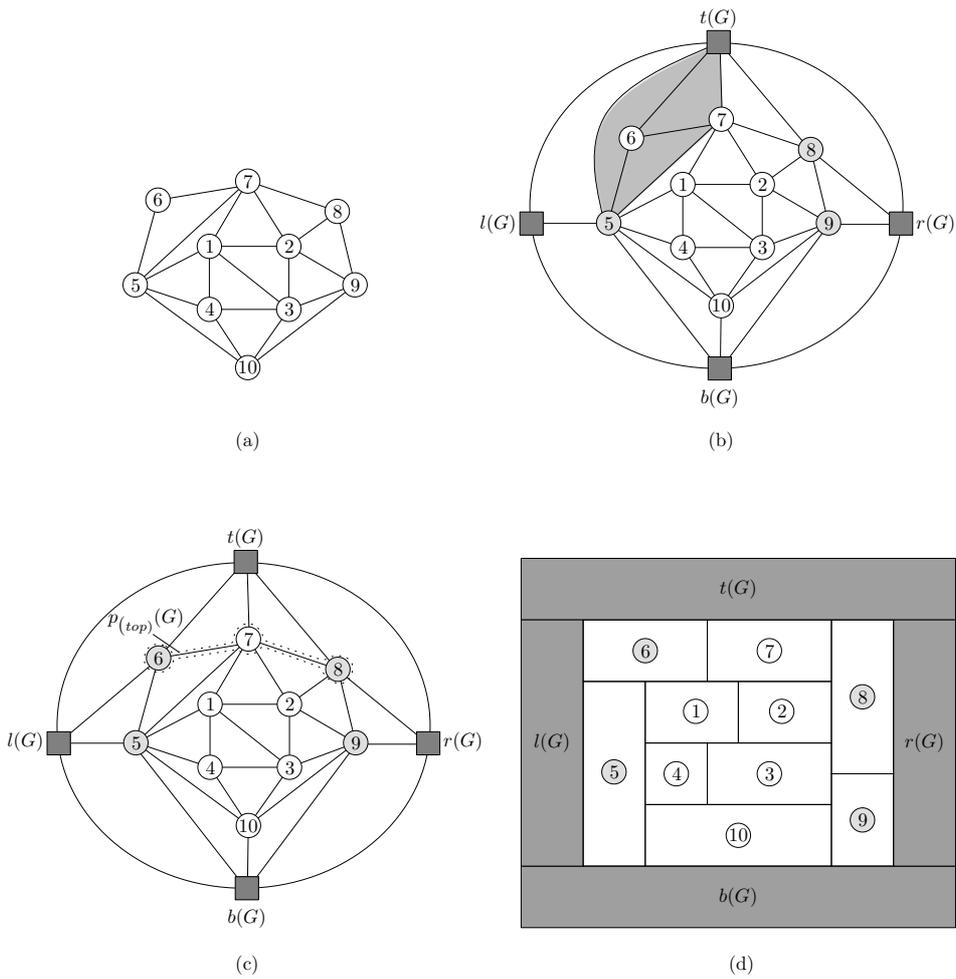
Figure 2.34 (a) A triangulated graph G; (b) An extended graph for G that does not admit a rectangular dual—the shaded area illuminates a separating triangle; (c) an extended graph $E(G)$ that admits the rectangular dual depicted in (d); the dotted line indicates the path $p_{\text{top}}(G)$.

           (a)   (b)   (c)
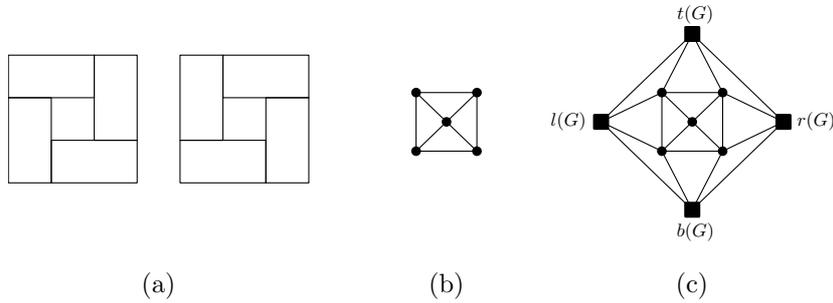
Figure 2.35 (a) Two windmill-layouts; (b) their dual graph; (c) the corresponding corner assignment.

Figure 2.35(a) depicts the two smallest non-sliceable rectangular layouts. Figure 2.35(b) depicts the dual graph of a windmill-layout and Figure 2.35(c) depicts the corresponding extended graph (the corner assignment). We call the graph in Figure 2.35(c) a *windmill-graph*. Any graph G that contains a windmill-graph as a subgraph is not sliceable, because in any dual layout of G the five inner vertices of the windmill-graph would have to be represented by one of the layouts in Figure 2.35(a) bounded by four rectangles representing the four outer vertices of the windmill-graph.

Approach. Let G be a plane triangulated graph without separating triangles that has a valid corner assignment $E(G)$. Let C be a separating four-cycle in G and let G_C be the subgraph of G that is induced by all vertices inside C.

For any dual layout $\mathcal{L}$ of $E(G)$ the part of the layout that represents the subgraph G_C is bounded by the four rectangles of the cycle C. In other words, in any rectangular dual $\mathcal{L}$ of G, the subgraph G_C is represented by its rectangular dual $\mathcal{L}_C$ with the corner assignment defined by (the adjacencies to) the 4-cycle C. This means in particular, that for all rectangular duals of G the rectangular duals presenting G_C have the same corner assignment. C defines the corner assignment for G_C; we define the corresponding extended graph of G_C by $E(G_C)$. (Since this is the only corner assignment for G_C we are interested in here, this does not introduce any ambiguity.)

Thus if $E(G)$ admits a sliceable dual, then so do the graph $E(G_C)$ and the graph obtained from G by *reducing* the subgraph G_C to a single vertex—i.e. replacing the subgraph G_C with a vertex v_C, connected to all vertices of C—we denote this graph $E(G) \setminus G_C$. Trivially, the inverse of the statement is true too: if the graphs $E(G_C)$ and $E(G) \setminus G_C$ admit a sliceable dual, so does the graph $E(G)$. Given a BSP tree $\mathcal{T}_0$ of a sliceable dual $\mathcal{L}_0$ of $E(G) \setminus G_C$ and a BSP tree $\mathcal{T}_C$ of a sliceable dual $\mathcal{L}_C$ of G_C (with corner assignment $E(G_C)$) we can obtain the BSP tree $\mathcal{T}$ of a sliceable dual $\mathcal{L}$ of G by replacing the leaf corresponding to rectangle R_v representing the node v_C in $G \setminus G_C$ with the tree $\mathcal{T}_C$.

Let C be a separating 4-cycle in $E(G)$. We say that C is *maximal* in $E(G)$ if its interior is not contained in any other separating 4-cycle of $E(G)$ except for the outer cy-

cle $(t(\mathcal{G}), r(\mathcal{G}), b(\mathcal{G}), l(\mathcal{G}))$. Let $MQ(E(\mathcal{G}))$ denote the set of all maximal separating 4-cycles in $E(\mathcal{G})$. Using the fact that every two maximal separating 4-cycles are interior-disjoint ([69]) we arrive at the following theorem:

Theorem 2.16 *Let $\mathcal{G}$ be a graph that admits a rectangular dual for the corner assignment $E(\mathcal{G})$. Let $MQ(\mathcal{G})$ be the set of all maximal separating 4-cycles in $\mathcal{G}$ and let $\tilde{\mathcal{G}}$ be a graph obtained by replacing the subgraph $\mathcal{G}_C$ for every $C \in MQ(\mathcal{G})$ by a single vertex connected to all vertices of C. $\mathcal{G}$ is sliceable if and only if*

(1) $E(\mathcal{G}_C)$ is sliceable for all $C \in MQ(\mathcal{G})$.

(2) The graph $\tilde{\mathcal{G}} = \mathcal{G} \setminus \bigcup_{\forall C \in MQ(E(\mathcal{G}))} \mathcal{G}_C$ is sliceable.

Thus to be able to test an arbitrary triangulated graph (that admits a rectangular dual) for sliceability, it is sufficient to be able to test graphs without separating four-cycles and graphs with only maximal separating four-cycles, where each separating four-cycle has exactly one vertex in its interior.

Yeap and Sarrafzadeh ([69]) have show that any graph without separating 4-cycles admits a sliceable dual.

As for graphs with only maximal 4-cycles, Dasgupta and Sur-Kolay [16] have published a paper where they have adapted the approach of Yeap and Sarrafzadeh and claimed that any graph that has only maximal separating 4-cycles (except for the graph whose extended layout is a windmill) is sliceable. It is important to note that Dasgupta and Sur-Kolay use a different definition of a maximal 4-cycle—they consider the 4-cycles in the graph $\mathcal{G}$ and not in $E(\mathcal{G})$—thus, their maximal separating 4-cycle is a separating 4-cycle in $\mathcal{G}$ whose interior is not contained in any other 4-cycle of $\mathcal{G}$. The set of maximal separating 4-cycles by our definition is different from the set defined by Dasgupta and Sur-Kolay—see Figure 2.36. However, it is not difficult to construct a simple counter-example to disprove this claim. As we pointed out earlier, any graph $\mathcal{G}$ with corner assignment $E(\mathcal{G})$ such that $E(\mathcal{G})$ contains a windmill-graph as a subgraph is not sliceable. Thus all we need to do is construct a graph $\mathcal{G}$ with a unique separating 4-cycle and assign its corners such that
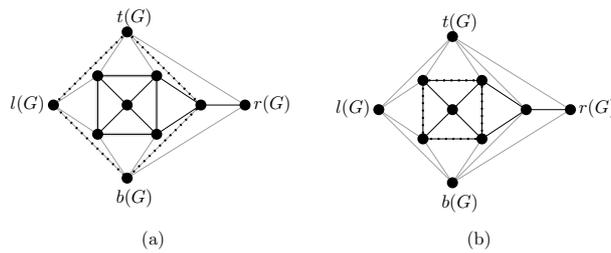
Figure 2.36 (a) The thick dotted line marks a maximal separating 4-cycle in $E(\mathcal{G})$; (b) the thick dotted line marks a maximal separating 4-cycle in $\mathcal{G}$.

(a)   (b)
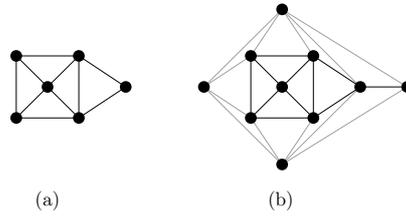
Figure 2.37 (a) graph G; (b) graph $E(G)$ that has a windmill-graph as a subgraph.

the corresponding extended graph $E(G)$ contains the windmill-graph as a subgraph—see Figure 2.37 for an example.

Similarly we can construct a graph with only maximal separating 4-cycles that does not admit a sliceable dual for any corner assignment. The graph G in Figure 2.38(a) contains five instances $G_1, \ldots G_5$ of the same graph G_0 (see Figure 2.39(a)) as subgraphs on its boundary. Consider any corner assignment $E(G)$ of G. If vertices labeled u_i and v_i of G_i are both not corner vertices, then G_i "inflicts" a windmill-graph as a subgraph of $E(G)$—see Figure 2.39(b). Graph G can have at most four corner vertices. Hence there exists i such that both u_i and v_i are not corner vertices. Thus any extended $E(G)$ graph of G contains a windmill-graph as a subgraph—Figure 2.38(b) depicts an example where G_i inflicts a windmill graph as a subgraph in $E(G)$.

(a)   (b)
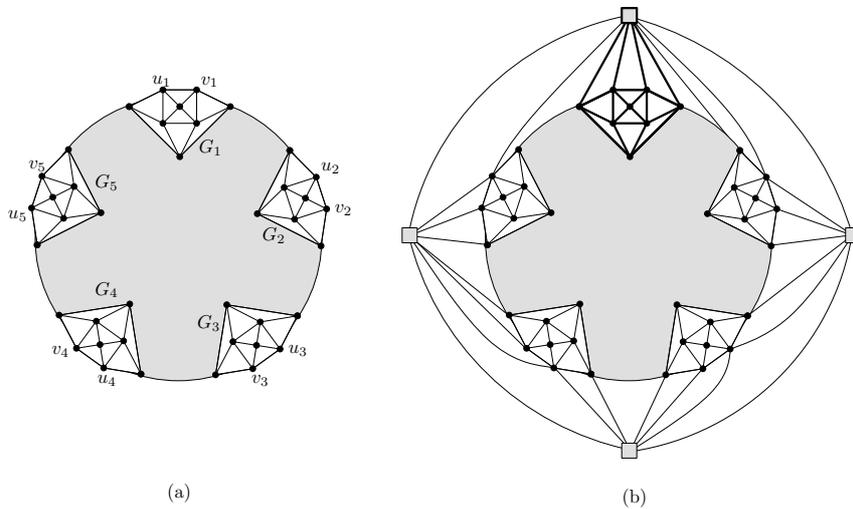
Figure 2.38 (a) A graph G that is not sliceable for any corner assignment; (b) An arbitrary corner assignment $E(G)$. A windmill-subgraph in $E(G)$ is drawn with thick edges.
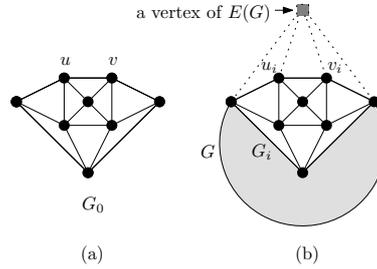
Figure 2.39 (a) The graph G_0; (b) A subgraph G_i in $E(G)$, its vertices u_i and v_i are not corner vertices.

In order to be able to explain where Dasgupta's and Sur-Kolay's proof in [16] went wrong we first need to outline the method of Yeap and Sarrafzadeh used by the authors of [16]. We describe this method in Section 2.6.1. We analyze the flaws in the proof of [16] in Section 2.6.2.

2.6.1 Graphs without separating 4-cycles.

A *path* in a graph is a sequence of vertices $v_1, ..., v_k$, where (v_i, v_{j+1}) is an edge of G for all $0 < i < k$. A path is called *simple* if $v_i \neq v_j$ for all $0 < i, j \leq k$. A path is *chordless* if it does not contain any "shortcuts", that is (v_i, v_j) is an edge of G if and only if $|i - j| = 1$, for all $0 < i, j < k, i \neq j$. A *cut* $E(V_1, V_2)$ in a graph G is defined as the set of all edges $e = (v_1, v_2)$ such that $v_1 \in V_1$, $v_2 = V_2$, where(V_1, V_2) is a partition of the vertex set V of the graph (see [22] for this and other graph theoretic definitions). A *bond* in a graph G is a minimal non-empty cut in G. A cut is *minimal* if it does not contain another cut of G as a subset. If G is connected, a cut $E(V_1, V_2)$ in G is a bond if and only if the graphs G_1 and G_2 induced by vertex sets V_1 and V_2 are connected.

Let G be a graph that admits a rectangular dual L for corner assignment $E(G)$—see Figure 2.40(a). Let ℓ be a vertical slice line in L that decomposes L into two sublayouts L_{left} and L_{right}—see Figure 2.40(b). Let $A_1, ..., A_m$ be the rectangles adjacent to ℓ on its left side in a top-down order, and let $B_1, ..., B_k$ be the rectangles adjacent to ℓ on its right side in a top-down order. The vertices of the dual graph of L corresponding to $A_1, ... A_m$ form a path $p_{\text{left}} = v_{A_1}, ..., v_{A_m}$. Similarly, the vertices of G corresponding to $B_1, ..., B_k$ form a path $p_{\text{right}} = v_{B_1}, ..., v_{B_k}$. The paths p_{left} and p_{right} are shaded gray in Figure 2.40(b).

The graph G is naturally decomposed into two subgraphs G_{left} (the dual graph of L_{left}), G_{right} (the dual graph of L_{right}) and the bond $S = E(V_{\text{left}}, V_{\text{right}})$, where V_{left} and V_{right} are the vertex sets of the graphs G_{left} and G_{right}—the edges of S are depicted in Figure 2.40(b) by dotted lines. More precisely, $S = \{e = (v_1, v_2) \in G : v_1 \in p_{\text{left}}, v_2 \in p_{\text{right}}\}$ is the set of edges connecting the vertices of the paths p_{left} and p_{right}.
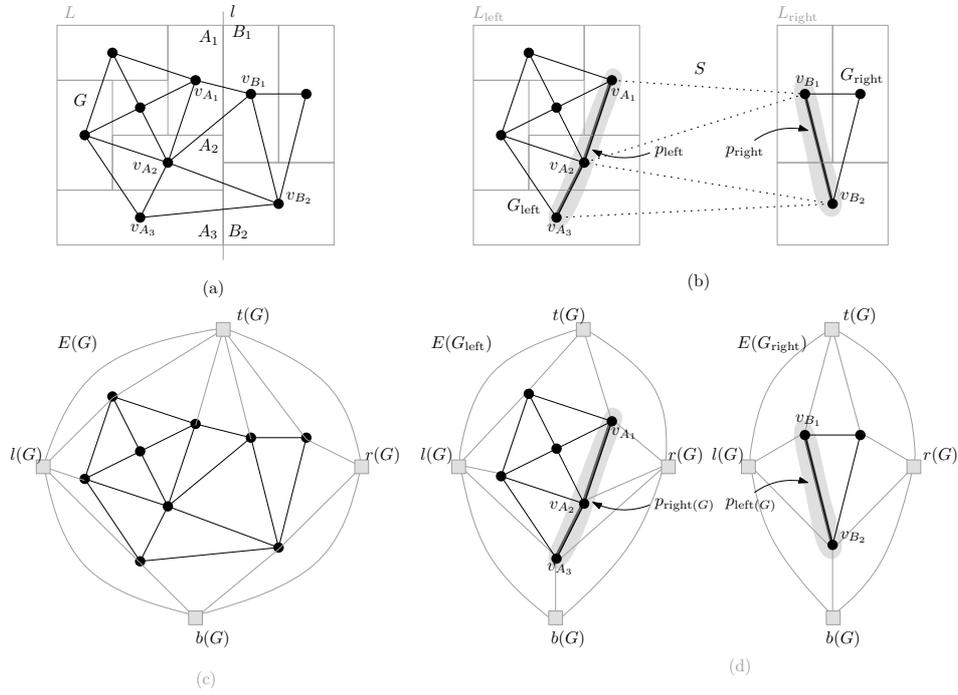
Figure 2.40 (a) Graph G (in black), its dual layout L (in gray) that has a slice line ℓ; (b) ℓ decomposes L into two layouts L_{left} (with the dual graphs G_{left}) and L_{right} (with the dual graph G_{right}), S is the cut in G of edges representing the adjacensies; (c) The corner assignment $E(G)$ corresponding to L; (d) The corner assignments $E(G_{\text{left}})$ and $E(G_{\text{right}})$ corresponding to the layouts L_{left} and L_{right}.

The corner assignment for the graphs G_{left} (corresponding to the dual layout L_{left}) and Gright (corresponding to the dual layout L_{right}) corresponding corner assignments for these graphs are given as follows (see Figure 2.40)(c), (d):

(1) $p_{\text{left}}(G_{\text{left}}) = p_{\text{left}}(G)$, $p_{\text{left}}(G_{\text{right}}) = p_{\text{right}}$

(2) $p_{\text{top}}(G_{\text{left}}) = p_{\text{top}}(G) \bigcap G_{\text{left}}$, $p_{\text{top}}(G_{\text{right}}) = p_{\text{top}}(G) \bigcap G_{\text{right}}$,

(3) $p_{\text{btm}}(G_{\text{left}}) = p_{\text{btm}}(G) \bigcap G_{\text{left}}$, $p_{\text{btm}}(G_{\text{right}}) = p_{\text{btm}}(G) \bigcap G_{\text{right}}$,

(4) $p_{\text{right}}(G_{\text{left}}) = p_{\text{left}}$, $p_{\text{right}}(G_{\text{right}}) = p_{\text{right}}(G)$

Furthermore, the bond $S = (V_{\text{left}}, V_{\text{right}})$ has following properties:

(5) the paths $(t(G), p_{\text{left}}(S), b(G))$ and $(t(G), p_{\text{right}}(S), b(G))$ are chordless;

(6) *S* contains exactly one edge e_{btm} of $p_{btm}(G)$ and exactly one edge e_{top} of $p_{top}(G)$. These are the only two edges of the outer boundary of G that belong to *S*.

Note that since *S* is a bond, property (5) guarantees that e_{top} and e_{btm} are the only two edges of the outer boundary of G that belong to *S*. A similar construction applies to a horizontal slice in $\mathcal{L}$. Thus, a rectangular dual for G is constructed by putting together the rectangular duals of its subgraphs G_{left} and Gright.

Can we do this trick the other way round? Namely, let G be a graph that admits a rectangular dual for a corner assignment $E(G)$. Let $S = (V_{left}, V_{right})$ be a bond in G such that such that V_{left} contains $p_{left}(G)$ and V_{right} contains $p_{right}(G)$. Moreover, the paths p_{left} formed by the vertices of $V_{left} \cap V(S)$ in G_{left} and p_{right} formed by the vertices of $V_{Rht} \cap V(S)$ in G_{right} have properties $(5) - (6)$. Let G_{left} and G_{right} be the graphs induced by V_{left} and V_{right} in G. Assume that their corner assignments $E(G_{left})$ and G_{right} are given by (1)-(4). Let $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$ be rectangular duals of G_{left} and G_{right} corresponding to the corner assignments. Can we construct a rectangular dual $\mathcal{L}$ of G by attaching $\mathcal{L}_{right}$ to the right of $\mathcal{L}_{left}$?

Yeap and Sarrafzadeh have proven that this is indeed the case. In other words, they have demonstrated that a rectangular dual $\mathcal{L}$ for G can be constructed by (roughly speaking) gluing the layout $\mathcal{L}_{right}$ to the right side of $\mathcal{L}_{left}$. (The "roughly speaking" refers to some transformations needed to be applied to the layouts to make the adjacencies between the regions of $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$ correct). The line along which the layouts are glued becomes a vertical slice line in $\mathcal{L}$. A bond that satisfies only the condition (5) is called a *vertical slice*. The path $p_{left}(S)$ is called the *left boundary path* of *S* and $p_{right}(S)$ is called the *right boundary path*. A vertical slice that satisfies condition (6) is called a *proper vertical slice* in G. A *horizontal slice* and a *proper horizontal slice* are defined similarly. A *(proper) slice* is a (proper) vertical or horizontal slice.

Moreover, Yeap and Sarrafzadeh have shown that given a graph G that admits a rectangular dual and contains no separating four-cycles one can always find a a proper slice in G and the graphs obtained from G by removing the edges of *S* admit a rectangular dual for the corner assignments given by (1)-(4) and do not contain separating four-cycles. Which means we can recursively find bonds in the graphs we decompose G into and this way construct a sliceable dual of G. They have formulated their results as the following theorem:

Theorem 2.17 *[69] Let G be a plane triangulated graph without separating 4-cycles and let $E(G)$ be its corner assignment such that $E(G)$ does not contain any separating triangles. Then*

(I) *If G contains at least two vertices, then there exists a proper slice *S* in G.*

(II) *The graphs obtained from G by removing the slice *S* with corner assignments given by (1)-(4) are plane triangulated graphs without separating four-cycles that admit a rectangular dual.*

Next we give a rough outline of the proper slice construction in their proof.

Assume without loss of generality that both $p_{\text{top}}(\mathcal{G})$ and $p_{\text{btm}}(\mathcal{G})$ contain at least two vertices—otherwise $p_{\text{left}}(\mathcal{G})$ and $p_{\text{right}}(\mathcal{G})$ have this property and we can "rotate" the procedure: We simply replace every "top" with "left", and every "bottom" with "right"). The proper vertical slice can be constructed in two steps:

(S1) Let $p_{\text{right}} = p_{\text{right}}(\mathcal{G})$. $(t(\mathcal{G}), p_{\text{right}}, b(\mathcal{G}))$ is chordless because $\mathcal{G}$ contains no separating triangles and contains only one vertex of $p_{\text{top}}(\mathcal{G})$. Consider the path p formed by all vertices of $\mathcal{G}$ adjacent to the vertices of p_{right}. If the path $(t(\mathcal{G}), p, b(\mathcal{G}))$ is chordless too, then the edges connecting the vertices of the paths p_{right} and p form a proper vertical slice in $\mathcal{G}$. If it is not, proceed to the next step.

(S2) Adjust both paths such that they do form a proper vertical slice.

2.6.2 Graphs with only maximal separating four-cycles

Dasgupta and Sur-Kolay attempted to adapt this approach to construct a sliceable dual for a larger class of graphs. They claimed to: *"... prove a stronger condition for slicibility, namely, even if a rectangular graph contains complex 4-cycles, it is necessarily slicible, provided all complex 4-cycles in the graph are maximal, and there does not exist any complex 4-cycle in the graph whose all four vertices are assigned to be corner blocks of the corresponding floorplan."* (They use the word "slicible" instead of sliceable in their paper, as quite e few people in the floorplanning community do.) They formulated their main result in the following theorem:

(Flawed) Claim [16] *A rectangular graph $\mathcal{G}$ with n vertices, $n > 4$, is slicible if it satisfies either of the following two conditions:*

- *its outermost cycle is the only complex 4-cycle in G and at least one of its four vertices is a non-distinct corner;*

- *all the complex 4-cycles of G are maximal.*

Here "rectangular graph" means a graph that admits a rectangular dual, and "non-distinct corner" means a corner vertex that represents a rectangle that occupies more than one corner of the layout, in other words, it is a corner vertex adjacent to more than two vertices of $t(\mathcal{G})$, $r(\mathcal{G})$, $b(\mathcal{G})$, $l(\mathcal{G})$. We find the use of the word "either" a bit confusing here, but based on the proof of the theorem we have chosen to read the claim such that both conditions are necessary for sliceability of a graph. The case when a graph needs to satisfy only one of the conditions clearly does not produce a correct theorem either. In other words, we interpret the statement of the claim as follows: Let $\mathcal{G}$ be a plane triangulated graph without separating triangles, such that any separating 4-cycle in $\mathcal{G}$ is maximal and let $E(\mathcal{G})$ be its valid corner assignment. If $E(\mathcal{G})$ is not a generalized windmill, then $E(\mathcal{G})$ has a sliceable rectangular dual.

The authors used the general approach of Yeap and Sarrafzadeh to find a proper slice in a graph that fulfills the conditions of their theorem. They extended the last step (the adjustments of the two paths) to be able to handle graphs with separating four-cycles.

However, they do not follow the proof outline of the previous group of authors. After describing how to construct a proper slice S in $\mathcal{G}$ they do not check whether the graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, obtained from $\mathcal{G}$ by removing the slice, satisfy the conditions of the theorem. The graph in Figure 2.37 presents a graph where the first slice would decompose the graph into graphs $\mathcal{G}_{\text{left}}$ and $\mathcal{G}_{\text{right}}$. $\mathcal{G}_{\text{left}}$ with the corner assignment given by the corner assignment of $\mathcal{G}$ and the slice forms a windmill-graph and hence does not satisfy the conditions of the theorem.

If that was the only flaw in the proof, the paper would prove a slightly weaker result. Namely, the paper would show that every graph that admits a rectangular dual and has only maximal separating 4-cycles would admit a *kinderdijk-layout* as a dual. This in its turn would mean that every graph that admits a rectangular dual has a pseudo-sliceable dual and hence admits a rectilinear cartogram of complexity 12 for any set of positive weights assigned to its vertices.

However, there is a second flaw in the proof of their theorem that prevents them from achieving even this weaker result. Namely, at the end of their slice construction proof they say: "*An interesting observation follows. If the boundary vertices of the C_4s are such that the vertices v_p^{i+1} and v_p^{i+2} are both adjacent to $t(\mathcal{G})$ in $E(\mathcal{G})$, then bipartitioning the graph using S_0' yields an $E(\mathcal{G}_l)$ with a C_3. In this case, slicibility of $\mathcal{G}$ cannot be guaranteed. For a slicible realization of the rectangular graph, in this case, any one of the vertices v_p^{i+1} and v_p^{i+2} must be a corner.*"

In fact their slice construction procedure only works if the corner assignment of the graph satisfies certain conditions. But in order to be able to slice a graph recursively, one has to be able to construct a slice for any given corner assignment—we only have freedom in choosing the corner assignments for the initial graph $\mathcal{G}$. Once we have performed the slice, the corner assignments of the subgraphs of $\mathcal{G}$ obtained by the slice are fixed. This means that the slicing procedure cannot be performed recursively and the theorem is completely invalid.

Furthermore, the authors say that the converse of their claim remains an open problem. We would like to note that the converse of the claim is trivially not true. Namely, there exists a graph $\mathcal{G}$ that contains separating four-cycles that are not maximal in $\mathcal{G}$ such that $\mathcal{G}$ admits a sliceable dual. Such a graph can be constructed as follows. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two graphs such that every separating 4-cycle in $\mathcal{G}_1$ is maximal in $\mathcal{G}_1$, and every separating 4-cycle in $\mathcal{G}_2$ is maximal in $\mathcal{G}_2$; $\mathcal{G}_1$ with corner assignment $E(\mathcal{G}_1)$ and $\mathcal{G}_2$ with corner assignment $E(\mathcal{G}_2)$ are sliceable.

Take an arbitrary separating 4-cycle C in $\mathcal{G}_1$ and replace the subgraph $\mathcal{G}_C$ in its interior by $\mathcal{G}_2$. Connect the vertices of $\mathcal{G}_2$ to C according to the corner assignment given by $E(\mathcal{G}_2)$. Then in the obtained graph $\mathcal{G}$ every separating 4-cycle of the subgraph $\mathcal{G}_2$ is not maximal in $\mathcal{G}$, but $\mathcal{G}$ is sliceable. The BSP tree of its sliceable dual can be obtained from

(a)   (b)
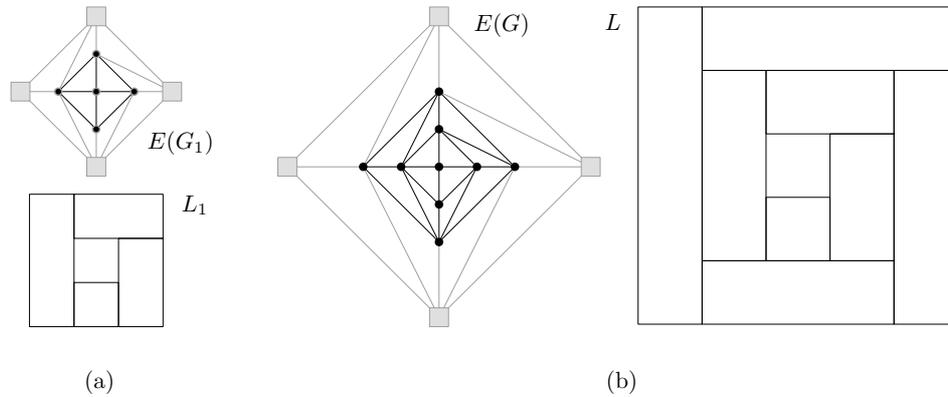
Figure 2.41 (a) A graph G_1 (depicted in black) that admits a sliceable dual L_1 for the depicted corner assignment; (b) A graph G that has a non-maximal separating four-cycle and its sliceable dual.

the BSP tree T_1 of the sliceable dual of G_1 by replacing the subtree T_C corresponding to the layout of the subgraph G_C by the BSP tree T_2 of the sliceable dual of the graph G_2. Figure 2.41 depicts an example of such a construction for the case when $G_1 = G_2$. The graph G_1 at the top of Figure 2.41(a) has a unique rectangular dual L_1 (Figure 2.41(a), bottom) for the given corner assignment. L_1 is sliceable. Figure 2.41(b) depicts the graph obtained by replacing the interior of the only separating four-cycle by another instance of G_1 connected to the vertices of the four-cycle such that the corner assignment given by $E(G_1)$ (Figure 2.41(a), top) is preserved.

Thus the problem of characterizing plane triangulated graphs that admit a sliceable rectangular dual remains open.

Chapter 3

Rectilinear duals with applications to cartograms

3.1 Introduction

In this chapter we discuss how the algorithm from the previous chapter can be used to construct rectilinear cartograms in practice. That is given a *input map*, i.e. a geographical map that is partitioned into regions and a positive value assigned to each region, we would like to construct a rectilinear cartogram for that map, such that the area of every region in the cartogram is equal to the assigned value. The (constructive) proof presented in the previous chapter guarantees the existence of a rectilinear cartogram for any input map and any set of area values. However, the resulting regions can be quite complex, with thin "tails" that facilitate correct adjacencies. Here we develop a more practical variant of the proposed approach. This new algorithm follows the general strategy of our previous method, but we introduce substantial algorithmic modifications in every step. We implemented and tested the algorithm on various data sets. It produces regions of very small complexity—in fact, most regions are rectangles—while still ensuring both exact areas and correct adjacencies for all regions.

One of the steps for which we develop a new technique is the computation of a *binary space partition* for a rectilinear layout, which we now discuss in more detail.

Binary Space Partitions. Suppose we have a collection S of objects in the plane. A *binary space partition*, or *BSP* for short, for S is a recursive subdivision of the plane by splitting lines, until each cell of the final subdivision is intersected by a single, or perhaps a small number, of objects from S. BSPs are well known indexing structures [59, 64] that can be used to do point location, to answer range queries, and so on. When the objects in

S are the regions of a layout, as it will be the case in our application, we thus require that each cell be contained in a unique region of the layout—see Figure 3.1 for an example. When we are dealing with BSPs for a rectilinear layout, it is natural to also make the BSP rectilinear, that is, to require that the splitting lines be either horizontal or vertical. From now on, we limit our discussion to rectilinear BSPs for rectilinear layouts.

The splitting lines of a BSP may cut the rectangles into fragments. When this happens often, the performance deteriorates: the size of the BSP—and, hence, the storage needed to store it—increases, and algorithms working on the BSP slow down. Hence it is desirable to limit the fragmentation as much as possible and indeed several papers present algorithms to construct BSPs of small size. For example, D'Amore and Franciosa [15] show that any layout consisting of *n* rectangles admits a BSP whose size (measured as the number of cells in the final subdivision) is at most 4*n*.

However, experimental evidence shows that many rectangular layouts admit BSPs of size close to *n*. Thus the question arises: is it possible, given a rectilinear layout, to construct a BSP whose size is optimal for that particular layout? We answer this question in the affirmative: we give a polynomial-time algorithm to construct a BSP of minimum size for any given rectilinear layout. Our algorithm is quite general—it can compute optimal BSPs for a wide class of cost functions. When computing cartograms we make use of this generality. We apply a dedicated cost function leading to BSPs amenable to the construction of high-quality cartograms.

Organization.   In Section 3.2 we present our algorithm for computing optimal BSPs for rectilinear layouts. In Section 3.3 we recap the general outline of the cartogram construction algorithm and describe the modifications and new techniques introduced at each step of the algorithm to improve the quality of its output. We report on experimental results in Section 3.4.

3.2   Optimal BSPs for rectilinear layouts

A *layout* is a partition of a rectangle into a finite set of interior-disjoint regions. A *rectilinear layout* is a layout where every region is a rectilinear polygon. Let $\mathcal{L}$ be a rectilinear layout with *n* edges in total. A BSP for $\mathcal{L}$ can be modeled as a BSP tree $\mathcal{T}$. Each internal node of $\mathcal{T}$ stores a splitting line and each leaf corresponds to a cell in the final BSP subdivision (see for example Figure 3.1).

Our algorithm to compute optimal BSPs can handle different optimality criteria. For example, it can be used to compute a minimum-size or a minimum-depth BSP. Before we present the algorithm, we first describe the type of cost functions that our algorithm can handle.

Let $\mathcal{T}$ be a BSP tree for $\mathcal{L}$. We define the cost of a node in $\mathcal{T}$ as follows.
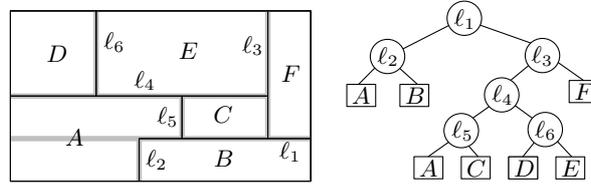
Figure 3.1 A BSP and the corresponding BSP tree. The leaves in the tree are labeled with the name of the region that contains the corresponding cell.

- We assume that each region r of the input map $\mathcal{M}$ has a non-negative cost associated to it, denoted $\text{cost}(r)$. We assign the cost to each region of the layout equal to the cost of the map region it represents. The costs of the regions in the layout determine the costs of the leaf nodes of $\mathcal{T}$: for a leaf μ we define $\text{cost}(\mu) := \text{cost}(r_\mu)$, where r_μ is the unique region of $\mathcal{L}$ that contains the cell in the BSP subdivision corresponding to μ.

- The cost of an internal node ν is determined by the costs of its children and a function $F : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$: if ν_1 and ν_2 denote the children of ν then we have $\text{cost}(\nu) := F(\text{cost}(\nu_1), \text{cost}(\nu_2))$.

The cost of a tree $\mathcal{T}$ is simply defined as the cost of its root. We call a BSP tree $\mathcal{T}$ for $\mathcal{L}$ *optimal* if its cost is minimal over all possible BSPs for $\mathcal{L}$. The goal of our algorithm is to compute such an optimal BSP tree, given the layout $\mathcal{L}$, the cost function on the regions of $\mathcal{L}$ and a function F. In order for our algorithm to work, the function F needs to be monotone in the following sense:

Monotonicity: For any a, a', b, b' with $a \leq a'$ and $b \leq b'$ we have
$F(a, b) \leq F(a', b)$ and $F(a, b) \leq F(a, b')$.

There are many natural optimality criteria that can be modelled like this. Suppose, for instance, that we want to compute a BSP of minimum size. Then we set the cost of each region to 1 and we define $F(a, b) = a + b$. To obtain a BSP of minimum depth we can also set the cost of each region to 1 but take $F(a, b) = \max(a, b) + 1$. Note that in both cases F is monotone. The possibilities of assigning different costs to different regions allows us to favor certain regions to be cut over other regions; in the application to cartograms we will make use of this flexibility.

We are now ready to describe our algorithm for computing an optimal BSP for $\mathcal{L}$. Let $x_1, x_2, \ldots, x_{n_x}$ be the sorted sequence of distinct x-coordinates of vertical edges in $\mathcal{L}$, and let $y_1, y_2, \ldots, y_{n_y}$ be the sorted sequence of distinct y-coordinates of horizontal edges in $\mathcal{L}$. We first *normalize* the layout: we replace the coordinates $x_1, x_2, \ldots, x_{n_x}$ by their ranks $1, \ldots, n_x$ and we replace $y_1, y_2, \ldots, y_{n_y}$ by $1, \ldots, n_y$. Note that an optimal (rectilinear) BSP for the original layout corresponds to an optimal BSP for the normalized layout; this is

true because the cost of a leaf depends only on the cost of the layout region that contains the leaf cell.

From now on, we use $\mathcal{L}$ to denote the normalized layout. Observe that there exists an optimal BSP for $\mathcal{L}$ such that each splitting line contains (a part of) an edge of $\mathcal{L}$. Indeed, if there is a splitting line that does not contain a part of an edge it can be shifted until it does; it is not difficult to prove that this cannot increase the cost of the BSP.

Now consider an optimal BSP tree $\mathcal{T}^*$ all of whose splitting lines contain a part of some edge. The splitting line at the root cuts $\mathcal{L}$ into two "sublayouts". These sublayouts are again cut into smaller sublayouts, and so on. Because all splitting lines contain a part of an edge of $\mathcal{L}$, the sublayouts that arise during the process are always rectangles of the form $[x_1 : x_2] \times [y_1 : y_2]$, where x_1 and x_2 are x-coordinates of vertical edges in the layout, and y_1 and y_2 are y-coordinates of horizontal edges. This leads us to define for $1 \leq x_1 < x_2 \leq n_x$ and $1 \leq y_1 < y_2 \leq n_y$ the following quantity:

$$\text{Opt}(x_1, x_2, y_1, y_2) := \text{the minimum cost of a BSP tree for the sublayout of } \mathcal{L} \text{ inside the rectangle } [x_1 : x_2] \times [y_1 : y_2].$$

Now consider an optimal BSP $\mathcal{T}^*$ that cuts a layout into two smaller sublayouts. Because of the monotonicity of F, the two subtrees of the root of $\mathcal{T}^*$ must be optimal BSP trees for these two sublayouts. Thus we can get an optimal BSP by trying the different ways to cut $\mathcal{L}$ into sublayouts along an edge, and then for each such cut compute the optimal BSP tree for the two sublayouts.

Lemma 3.1 *If $x_2 = x_1 + 1$ and $y_2 = y_1 + 1$ then $\text{Opt}(x_1, x_2, y_1, y_2) = \text{cost}(r)$, where r is the region of $\mathcal{L}$ containing the rectangle $[x_1 : x_2] \times [y_1 : y_2]$. Otherwise, we have*

$$\text{Opt}(x_1, x_2, y_1, y_2) = \min \begin{cases} \min_{x_1 < x < x_2} F(\text{Opt}(x_1, x, y_1, y_2), \text{Opt}(x, x_2, y_1, y_2)) \\ \min_{y_1 < y < y_2} F(\text{Opt}(x_1, x_2, y_1, y), \text{Opt}(x_1, x_2, y, y_2)) \end{cases}$$

Using Lemma 3.1 we can compute an optimal BSP for a layout $\mathcal{L}$ by dynamic programming. We have a 4-dimensional table $\text{Opt}[1..n_x - 1, \ 2..n_x, \ 1..n_y - 1, \ 2..n_y]$ that we fill in as follows: we first determine $\text{Opt}[x, x+1, y, y+1]$ for each $1 \leq x < n_x$ and $1 \leq y < n_y$ by finding for each rectangle $[x : x+1] \times [y, y+1]$ the layout region containing it, and then we fill in the rest of the table in a bottom-up manner, using Lemma 3.1. We spend $O(n)$ time to compute the value for each entry in the table. At the end the optimal cost of a BSP tree for $\mathcal{L}$ is stored in $\text{Opt}(1, n_x, 1, n_y)$. Once the table is filled, we can make a second (top-down) pass through the table to produce a BSP tree that leads to the optimal cost; how to do this is standard in dynamic-programming algorithms, and so we omit the details. We obtain the following result.

Theorem 3.2 *Let $\mathcal{L}$ be a rectilinear layout with n edges in total, and suppose we have a cost function on BSP trees determined by costs on the regions in $\mathcal{L}$ and a monotone function F, as described above. Then we can compute an optimal BSP for $\mathcal{L}$ in $O(n^5)$ time.*

Practical improvements. Our algorithm to compute optimal BSPs works fine for small layouts—that is, layouts with not too many regions—but for large layouts it becomes quite slow and needs a lot of storage. Next we describe some heuristics that improve its performance.

Recall that there is an optimal BSP where each splitting line contains (a part of) an edge of $\mathcal{L}$. Hence, when we compute $\text{Opt}(x_1, x_2, y_1, y_2)$ according to Lemma 3.1, we need to try only those values of x for which there is a vertical edge of the layout $\mathcal{L}$ that lies at least partially within the rectangle $[x_1 : x_2] \times [y_1 : y_2]$. We can reduce the number of different y-values to be tested similarly.

When $F(a, b) = a + b$ then we can speed things up even further by using the following observation. Suppose we have a sublayout for which we want to compute an optimal BSP, and suppose that there is a region that has an edge cutting completely through the sublayout. Then a splitting line containing that edge will not cut *any* region of the sublayout. We call this a *free split*. In Figure 3.1, for instance, after we have split along ℓ_1, there is a free split in the sublayout above ℓ_1, namely along ℓ_3. When $F(a, b) = a + b$, we can always take such a free split without losing optimality; we do not have to try any other options for the splitting line. (This is not the case in general. If we want to optimize the depth, for example, a free split can be unbalanced and not lead to an optimal result.)

If we apply these two heuristics, then in many cases we do not have to test all x with $x_1 < x < x_2$ and all y with $y_1 < y < y_2$. This reduces the computation time significantly. We can also use this to reduce the amount of storage. If there are cases where we do not try all possibilities according to Lemma 3.1, then the corresponding entry in the table $\text{Opt}[1..n_x - 1, 2..n_x, 1..n_y - 1, 2..n_y]$ is apparently not needed. Indeed, in our experiments we observed that only 21% or less of the table was actually used.

Hence, we can reduce the storage substantially if instead of using the table $\text{Opt}[1..n_x - 1, 2..n_x, 1..n_y - 1, 2..n_y]$, we store the values of the entries in a hash table. More precisely, we have a hash table $T[0..m]$ and a hash function h that maps a four-tuple (x_1, x_2, y_1, y_2) with $1 \leq x_1 \leq n_x - 1$, $2 \leq x_2 \leq n_x$, $1 \leq y_1 \leq n_y - 1$, and $2 \leq y_2 \leq n_y$ to an index in the range $0..m$. The value for $\text{Opt}(x_1, x_2, y_1, y_2)$ will then be stored in $T[h(x_1, x_2, y_1, y_2)]$. To make this work, we need to use a top-down version of the dynamic-programming algorithm that uses memoization [14]. This also allows us to perform *pruning* to avoid solving subproblems that they cannot lead to optimal results.

As mentioned earlier, we will use the optimal BSP-construction algorithm as an intermediate step in our algorithm to construct rectilinear cartograms. To obtain high-quality cartograms we have tried several dedicated costs functions, which will be described in detail in Section 3.3 (see Step 3). Section 3.4 reports on the experimental comparison of the optimal BSP-construction algorithm with the algorithm by d'Amore and Franciosa [15] and a greedy algorithm, in the context of our cartogram application.

3.3 Computing rectilinear cartograms

Recall that a map is a partition of a rectangle into a finite set of interior-disjoint regions. The *dual graph* $\mathcal{G}(\mathcal{M})$ of a map $\mathcal{M}$ is the graph that has one node per region and connects two regions if they are adjacent. Thus a cartogram has correct adjacencies if and only if its dual graph is the same as the dual graph of the original map.

Algorithmic outline.   As mentioned before our algorithm follows the general outline of the algorithm we presented in the previous chapter.

The algorithm roughly works as follows: we construct a suitably modified version $\mathcal{G}$ of the dual graph $\mathcal{G}(\mathcal{M})$ of $\mathcal{M}$, assign to each vertex in this graph $\mathcal{G}$ a weight that equals the required area of the corresponding region, and run the algorithm from the previous chapter on the resulting vertex-weighted graph. This will lead to a correct rectilinear cartogram with regions of bounded complexity. To obtain good results in practice, however, we need to modify the algorithm in several places. We now first describe the various steps of our algorithm and then we discuss each of these steps in more detail.

> **Algorithm** CARTOGRAM CONSTRUCTION($\mathcal{M}$)
> *Input.* A map $\mathcal{M}$ and the required area for each region.
> *Output.* A rectilinear cartogram $\mathcal{C}$.
> 1.   Compute the dual graph $\mathcal{G}(\mathcal{M})$ of $\mathcal{M}$ and augment it with sea and pole vertices to obtain the graph $\mathcal{G}$. If necessary, modify the graph $\mathcal{G}$ so that it has a rectangular dual.
> 2.   Construct a rectangular dual $\mathcal{L}_1$ of $\mathcal{G}$.
> 3.   Construct a BSP tree $\mathcal{T}$ of $\mathcal{L}_1$.
> 4.   Move the BSP lines to correct the areas.
> 5.   Grow tails to correct any broken adjacencies.
> 6.   Move the BSP lines to repair the area errors re-introduced in Step 5.
> 7.   Produce the final cartogram $\mathcal{C}$.

Step 1: Connectivity graph construction

To convert the input map into a suitable weighted graph, we follow the same approach as van Kreveld and Speckmann [46]. We first compute the dual graph $\mathcal{G} = \mathcal{G}(\mathcal{M})$ of the input map and then assign to each vertex a weight that is the required area of the corresponding region in the map. For the subsequent steps of the algorithm it is convenient to work with a graph that has a rectangular dual. A planar graph has a rectangular dual with four rectangles on the boundary if and only if the following three conditions are met [5, 45]: (i) every interior face is a triangle, (ii) the exterior face is a quadrangle, and (iii) the graph has no separating triangles, that is, no 3-cycles for which there are vertices outside and inside the cycle.

To enforce condition (i), we simply triangulate any interior face that is not a triangle. (In most cases G is in fact already triangulated except for its outer face and for inner seas that are adjacent to several regions.) To enforce (ii), we add NORTH, EAST, SOUTH, and WEST vertices—we call these vertices *poles*—to form the boundary of the graph. There are several possibilities to enforce (iii). Here we add additional neighbors to all vertices of degree three or less. For example, when constructing the connectivity graph for the map of Europe, we split Belgium into two parts and make Luxemburg adjacent to both of them.

Furthermore, to make the future cartogram resemble the original map more closely, we add so-called sea vertices to our graph, representing the main bodies of water of the map. Some bodies of water are represented by more than one vertex to make the cartogram more recognizable or to increase the degree of a vertex where needed. After Step 1 we have a vertex-weighted graph G that admits a rectangular dual.

Step 2: Constructing a rectangular dual

We use the algorithm by Kant and He, but—as proposed in [46]—we make use of the fact that it has the possibility to specify for any two neighboring vertices u and v whether the region corresponding to u should be north, east, south, or west of the region corresponding to v. This is done by labeling each edge with a direction: north, east, south, or west. In the previous chapter it was not relevant how this was done since the input was a graph and not a map, but we want the edge labeling to correspond to the geographic situation as much as possible. For example, the label of the edge connecting the vertex for the Netherlands to the vertex for Germany should indicate that the Netherlands lies west of Germany. To determine how to label an edge, we employ a simple heuristic: we consider the relative positions of the centers of mass of the two regions in the original input map M. Often the relative positions clearly indicate how to label an edge, but sometimes two options are possible. For example, the center of mass of Germany lies north-west of the center of mass of Austria. This gives us a so-called *layout option*. If the input map M has m layout options, then there are 2^m different directed edge labeling possible, each potentially leading to a different rectangular dual. However, not every directed edge labeling corresponds to a layout that realizes that labeling. A directed edge labeling such that can be realized as a rectangular layout is called *regular edge labeling* ([43]). It is a labeling that satisfies the following conditions ([43]):

(1) Every internal node has at least one north, one south, one east, and one west directed edge.

(2) The edges around each internal vertex of the graph come in the following order clockwise around the vertex: first all edges directed north, then all edges directed east, next all the edges directed south and finally all the edges directed west.

Our algorithm tries all possible layout options and generates a rectangular dual for each regular edge labeling. For every rectangular dual we then execute the remaining steps of

the algorithm, leading to a number of different cartograms. From these, we can choose the best based on various quality criteria.

Step 3: Constructing a BSP

Step 3 computes a rectilinear BSP for the rectangular layout $\mathcal{L}_1$ that results from Step 2. In general, we would like the BSP to avoid cutting regions as much as possible, because regions that are cut by the BSP have more complicated shapes in the final cartogram. That is, we would like to cut as little number of regions as possible, into as little number of pieces as possible.

The BSP-construction algorithm by d'Amore and Franciosa [15] which we referred to in the previous chapter guarantees that each rectangle in $\mathcal{L}_1$ is cut into at most four pieces. Although this algorithm is the one that provides the best known theoretical bound for the complexity of the output, it is not the one that works best in practice. We are going to show here that using the optimal BSP construction algorithm described in Section 3.2 brings us closer to our "cutting as little as possible" goal.

We use the optimal BSP construction algorithm described in Section 3.2 and we experiment with several different optimality criteria. Recall that the cost of a BSP tree is determined by the costs assigned to each of the regions—this determines the costs of each leaf in the BSP tree—and a function F that determines how to compute the cost of a node from the costs of its children. In our experiments we always use $F(a,b) = a + b$. This means that the cost of a BSP is simply the sum over all regions of the number of pieces into which the region is cut, weighted by the cost of that region. For the cost of cutting a region, we try several alternatives:

- The obvious choice is to set the cost of each land region to 1. This minimizes the total number of cuts over all regions, leading to cartograms with regions that have simple shapes. Because the final shape of the sea regions is less important, we give them cost zero.

Observe that a region that is cut by the BSP has more flexibility to change its shape in the remaining steps. This suggests to give "difficult" regions a lower cost, which ensures they are cut first if any regions need to be cut at all. This can be done in two ways:

- Regions whose required area deviates greatly from their current area are more difficult to handle, so we can give such regions a lower cost. More precisely, if A_r is the required area of a land region and $A_{\mathcal{L}_1}$ is the area of that region in $\mathcal{L}_1$ then we set its cost to be

$$\min\left(\frac{A_r}{A_{\mathcal{L}_1}}, \frac{A_{\mathcal{L}_1}}{A_r} \right) \, .$$

- Regions with many neighbors might need some extra flexibility. Thus we set the cost of a land region to

$$\frac{1}{1 + \text{number of land neighbors}}.$$

We compare these BSP construction strategies experimentally to the following other strategies.

- The algorithm by D'Amore and Franciosa [15], as used in the previous chapter. We also tried a variant of this algorithm that applied free splits (see the previous section) whenever possible.

- A simple greedy strategy that always uses a splitting line for which the total cost of the cut regions is minimal. The greedy strategy automatically uses free splits if they are possible, because a free split has zero cost. As in our optimal BSP algorithms, we define the cost of cutting a sea region to be zero and we try different costs for the land regions: cost 1 for each region, a cost that depends on the deviation of the original and specified area of the region, and a cost that depends on the number of neighbors of the region. When there are more splitting lines with minimum cost, we take the one that is most balanced (in terms of the number of regions on both sides of the line).

Every BSP construction method might cut some regions into two or more subregions and then the weight w of the region has to be distributed between its k subregions. In other words, we must assign positive weights $w_1, \ldots, w_k$ to the subregions such that $\sum_{i=1}^{k} w_i = w$. We do this proportionally to the areas: if a rectangle r from $\mathcal{L}_1$ is cut by the BSP into subrectangles $r_1, \ldots, r_k$, then the specified area for a subrectangle r_i is (required area of r)· $(\text{area}(r_i)/\text{area}(r))$.

Step 4: Getting the areas right

The input to Step 4 is a BSP tree $\mathcal{T}$ for the rectangular dual $\mathcal{L}_1$. Recall that each internal node v in a BSP tree stores a splitting line; we denote this line by $\ell(v)$. Also recall that each leaf in the BSP corresponds to a cell in the BSP subdivision (which in our case is contained in one of the regions of $\mathcal{L}_1$). For an internal node v we define $R(v)$ to be the union of all the cells corresponding to leaves in the subtree of v. This implies that $R(\text{root}(\mathcal{T}))$ is simply the whole layout area, and that the splitting line $\ell(v)$ splits $R(v)$ into $R(\text{leftchild}(v))$ and $R(\text{rightchild}(v))$.

We traverse the $\mathcal{T}$ top down. At each node v the splitting line $\ell(v)$ is repositioned while maintaining the following invariant: when a node v is handled, the current area of $R(v)$ is equal to the sum of the weights of the required areas of the leaf cells in the subtree $\mathcal{T}(v)$ rooted at v. We start at the root and scale $\mathcal{L}_2$ for R to have the required area. Then for

each internal node v we position $\ell(v)$ such that $R_{\text{rightchild}(v)}$ and $R_{\text{leftchild}(v)}$ have correct areas. When we arrive at the leaves the corresponding cells have correct areas.

The repositioning of the splitting lines may destroy some of the adjacencies. This will be remedied in the later steps.

In order to reduce the impact of this step on the correctness of the adjacencies, we would like to move each splitting line as little as possible. But unfortunately, the location of each splitting line seems completely determined by the required areas. We observe, however, that this is not completely true: we can make use of the fact that sea regions are introduced artificially and do not have specified area requirements. Initially, we set the required areas for the sea rectangles such that the total sea area will sum up to some fixed percentage (for example, 20%) of the total layout area, and then we distribute this total sea area over the various sea rectangles in a more or less arbitrary manner. Now, instead of preserving these initial settings, we do the following. Suppose that we are repositioning a vertical splitting line $\ell(v)$ and that we have to move it to the right. If there are sea rectangles to the left and to the right of $\ell(v)$, then we can decrease the required area for the sea to the left of $\ell(v)$ and increase the required area for the sea to the right of $\ell(v)$. The total sea area stays the same, but $\ell(v)$ has to be moved less, or maybe not at all. Hence, we reduce the movement of the BSP lines and lower the number of broken adjacencies. However, if we transfer too much sea area, then the final layout will not look good. Therefore we use a threshold parameter that gives an upper bound on the percentage of the sea area that we are allowed to transfer.

Step 5: Tailing

Repositioning the splitting lines during the previous step may have caused some of the adjacencies to be broken, as illustrated in Figure 3.2. Step 5 fixes these adjacencies.

The algorithm fixes the adjacencies using so-called *tails*. These are very thin rectangles that are added to a region to connect it to some other region. The region then becomes an L-shape—see Figure 3.2—or something more complicated if a region gets several tails. Every tail introduces area errors. Therefore the tails created by the algorithm are extremely thin, to guarantee that the errors can be repaired in Step 6 without destroying any adjacencies.
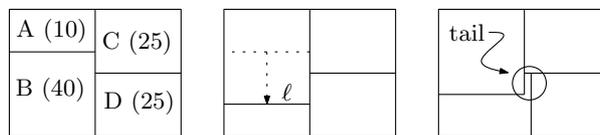
Figure 3.2 A rectangular dual $\mathcal{L}_1$ with specified areas. The splitting line ℓ is moved down in Step 4 and the adjacency between B and C is broken; it is restored by adding a tail to B.

Here we add two heuristics to our algorithm: one is aimed at reducing the number of tails, the other at increasing the width of the tails. We reduce the number of tails by ignoring some of the broken adjacencies without compromising the correctness of the cartogram. This is possible when some of the sea areas of the map are represented by more than one sea region of the cartogram, because it is not important which one of these sea regions a land region is bordering. Thus we can relax the adjacency requirements as follows:

- The sea regions forming the same sea area on the map have to form a connected region, but which sea rectangles border each other is not important.

- Each land region must have the correct adjacencies with the other land regions, but when it is adjacent to a sea region then it does not matter which sea region (of the same sea area) it is adjacent to.

Our second heuristic tries to increase the width of the tails. This is important because the tails that are produced by the algorithm are so extremely thin that they easily become invisible. We start with fairly wide tails and run the algorithm. If it finishes with a correct cartogram then we are done. But the tails might be too wide to be able to repair the areas in Step 6 without breaking adjacencies. When this happens then we decrease the tail width and try again. In fact, our implementation is a bit more careful and reduces the tail width only locally (namely, at the place in the layout where the algorithm could not repair the area).

Step 6: Correcting the areas again

The errors in the areas are corrected by moving the splitting lines—which have by now become polylines—of the BSP, similar to Step 4. This time, however, the errors to be repaired are so small that one can prove that the splitting lines have to be moved only so little that no adjacencies are destroyed. We follow the approach from the previous chapter, but we use the extra flexibility that the sea areas give us, as described in Step 4.

Step 7: Producing the final cartogram.

Some of the regions may have been split into parts when constructing the connectivity graph or the BSP. We merge these pieces so that they form one region again.

3.4 Implementation and test results

Recall that rectangular cartograms can not always achieve both zero cartographic error and correct adjacencies, while our rectilinear cartograms always achieve both. Hence our main quality criterion is the shape of the regions. We want to use as many rectangular

regions as possible, and only a few L-shapes or other regions of higher complexity. Furthermore, we would like to avoid very thin regions. We measure the complexity of a region by the number of corners and we generalize aspect ratio to rectilinear polygons by defining the fatness of a rectilinear polygon R as

$$\text{area}(R)/\text{area}(\text{bounding square of } R).$$

Note that fatness is the inverse of aspect ratio when R is a rectangle. We also measure the width of the tails that are added in Step 5 of our algorithm: the algorithm we introduced in the previous chapter uses extremely thin tails—so thin that they are invisible when printed—and we want to see whether our modification leads to better results. Finally, we also report on the size of the BSP produced in Step 3 of our algorithm.

We use two different maps: the countries of Europe and the states of the US. Our data sets are area, population, total highway length, gross domestic product (GDP; Europe only), native population (US only), number of cars (US only), and number of farms (US only). Thus in total we performed 10 different experiments. As mentioned earlier our algorithm tries a number of different layout options in Step 2. The results we report below are always for the layout with the largest minimum tail width. For each experiment we report the following measures:

- maximum complexity (MC) and average complexity (AC) of the regions in the final cartogram.
- minimum fatness (MF) and average fatness (AF) of the regions in the final cartogram.
- minimum tail width (TW) of the tails added in Step 5.
- maximum number (MP) and average number (AP) of rectangular pieces per map region after Step 3.

We tried several variants of our algorithm, which only differ in the way the BSP is constructed in Step 3. They are:

- The optimal algorithm from Section 3.2, with three different cost functions for the regions: unweighted (CN), weighted based on area deviation (AD), and weighted based on the degree (DG).
- A greedy algorithm that always uses a splitting line for which the total cost of the cut regions is minimal, with the same three cost functions.
- The algorithm by D'Amore and Franciosa [15], as used in the previous chapter. We use a variant of this algorithm that applies free splits whenever possible; otherwise the results are much worse.

Figures 3.3-3.6 show the exact results for some of our experiments. In all cases the total sea area was set to 20%. The cartogram in Figure 3.3 shows Europe with the theme

population. Figure 3.4 is another cartogram of Europe, with the theme gross domestic product. Both Europe cartograms were produced with the unweighted optimal BSP algorithm. Figure 3.5 and Figure 3.6 show two cartograms of the US with the theme population and number of highways, respectively. Both were also produced by the optimal BSP algorithm, here based on the number of neighbors. Finally, Figure 3.7 shows all intermediate layouts for the construction of the EU population cartogram.

EU population							
QM	D'Am	Greedy			Optimal		
		CN	AD	DG	CN	AD	DG
MC	18	10	14	8	8	8	10
AC	5.65	5.1	5.05	4.9	4.6	4.6	4.6
MF	0.03	0.02	0.02	0.03	0.04	0.04	0.04
AF	0.45	0.45	0.47	0.47	0.45	0.44	0.45
TW	1.26	1.99	1.76	2.18	3.07	3.07	3.07
MP	5	3	3	3	3	3	3
AP	1.45	1.13	1.15	1.15	1.13	1.13	1.13

Figure 3.3 Europe, theme population.

EU GDP							
QM	D'Am	Greedy			Optimal		
		CN	AD	DG	CN	AD	DG
MC	10	12	10	12	10	10	10
AC	5.2	4.95	4.65	4.85	4.6	4.55	4.55
MF	0.03	0.03	0.02	0.01	0.06	0.05	0.05
AF	0.40	0.37	0.42	0.40	0.41	0.44	0.44
TW	1.66	1.96	0.82	1.02	2.32	2.20	2.20
MP	3	3	3	3	3	3	3
AP	1.28	1.13	1.10	1.13	1.13	1.08	1.08

Figure 3.4 Europe, theme gross domestic product (GDP).

US population							
QM	D'Am	Greedy			Optimal		
		CN	AD	DG	CN	AD	DG
MC	8	8	8	8	8	8	8
AC	5.04	5.29	5.17	4.79	5.17	5.17	5.25
MF	0.04	0.02	0.02	0.03	0.02	0.02	0.02
AF	0.41	0.37	0.38	0.40	0.37	0.37	0.39
TW	1.01	1.20	1.35	1.37	1.36	1.36	1.36
MP	2	2	2	2	2	2	1
AP	1.04	1.02	1.08	1.02	1.02	1.02	1.00
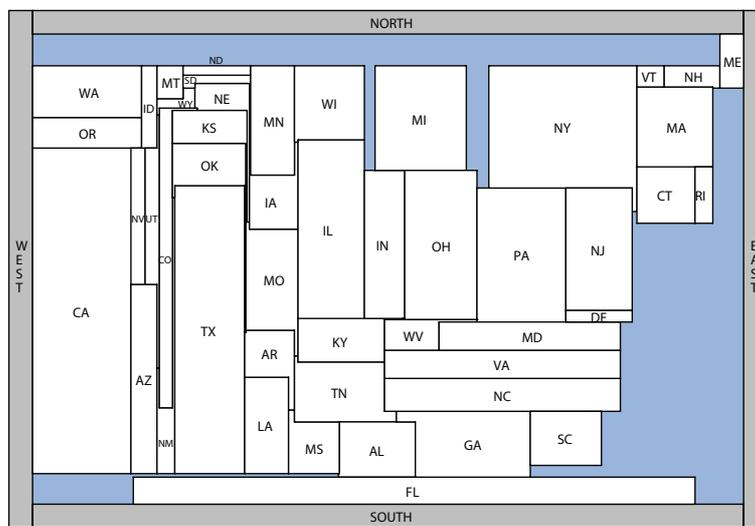
Figure 3.5 USA, theme population.

US highways							
QM	D'Am	Greedy			Optimal		
		CN	AD	DG	CN	AD	DG
MC	12	8	8	10	8	8	10
AC	5.29	4.58	4.67	4.79	4.71	4.67	4.67
MF	0.06	0.06	0.10	0.09	0.08	0.10	0.12
AF	0.47	0.50	0.49	0.50	0.48	0.45	0.47
TW	1.40	2.98	2.09	2.45	2.45	1.84	2.75
MP	2	1	2	2	2	2	2
AP	1.17	1.00	1.04	1.15	1.04	1.04	1.06
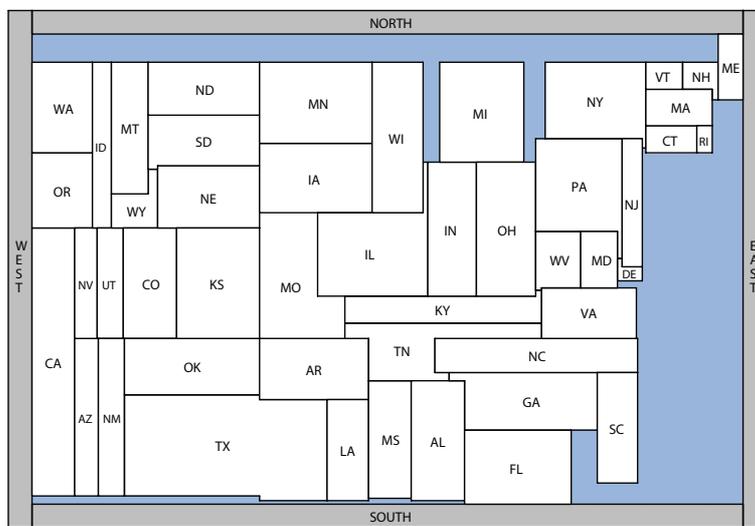
Figure 3.6 USA, theme highway length.

Discussion. In 8 out of our 10 experiments, a variant (usually the one based on area deviation) of the optimal BSP algorithm produces a cartogram with the lowest average complexity of the regions. In the two other experiments the greedy BSP algorithm is the best.

The average complexity of the regions in the best cartogram never exceeds 5, and is often below 4.6. Since a rectangle has complexity 4, this means that most of the regions of our rectilinear cartograms are in fact rectangles. The maximum complexity of the regions in the best cartogram never exceeds 10. This is significantly better than the maximum complexity of 20 guaranteed by the Lemma 2.6.

A major drawback of the algorithm in Chapter 2 from a practical point of view are the incredibly thin tails that are a consequence of the correctness proof. Our improved algorithm produces tails that are clearly visible and allow the user to extract correct adjacency information from the cartograms.

The BSP algorithms generally cut only a few rectangles into pieces; most rectangles are never cut. More precisely, in all experiments there was a BSP such that no rectangle was cut into more than two pieces. Note however, that that was not necessarily the BSP that gave the optimal cartogram. Although each region of the cartogram might be represented by several rectangles in the rectangular dual $\mathcal{L}_1$ (for example, Belgium is represented by 2 rectangles), the average number of rectangular pieces per map region after Step 3 is always 1.13 or less—much better than the average of 2 guaranteed by the algorithm of D'Amore and Franciosa [15].

When studying the tables of results it might appear that sometimes the unweighted optimal BSP algorithm seems to produce a higher number of pieces than the weighted optimal BSP algorithm, which would contradict the optimality of the unweighted algorithm. However, this is an artifact caused by the use of layout options. For each variant of the algorithm we always try a number of layouts and report the data for the option with the widest tails. Thus the data reported for the weighted version can be based on a different layout than the data for the optimal algorithm.

Finally, we did not try to optimize the running time of our algorithm, but it is reasonably fast nevertheless: on the US data sets the algorithm typically took about half a minute, while on the Europe data sets it was less then 10 seconds. Both numbers are wall clock times on a PC with the specifications: Intel(R) Pentium(R) 4. CPU 2.80 GHz, 1.00 GB of RAM running MS Windows XP Version 2002, Service Pack 2. 10 seconds is the time spent for running the algorithm once, that is, with one layout option (i.e. one edge labeling for the complete graph) and one BSP algorithm. (Trying all layout options makes the algorithm a lot slower, since the number of different layouts we tried is fairly large: 192 for the US data set, and 72 for the Europe data sets.)

(a) Rectangular dual.

(b) Rectangular dual cut by BSP.

(c) Correct area.

(d) Correct adjacencies by tailing.

(e) Correct areas and adjacencies.

(f) Final cartogram.
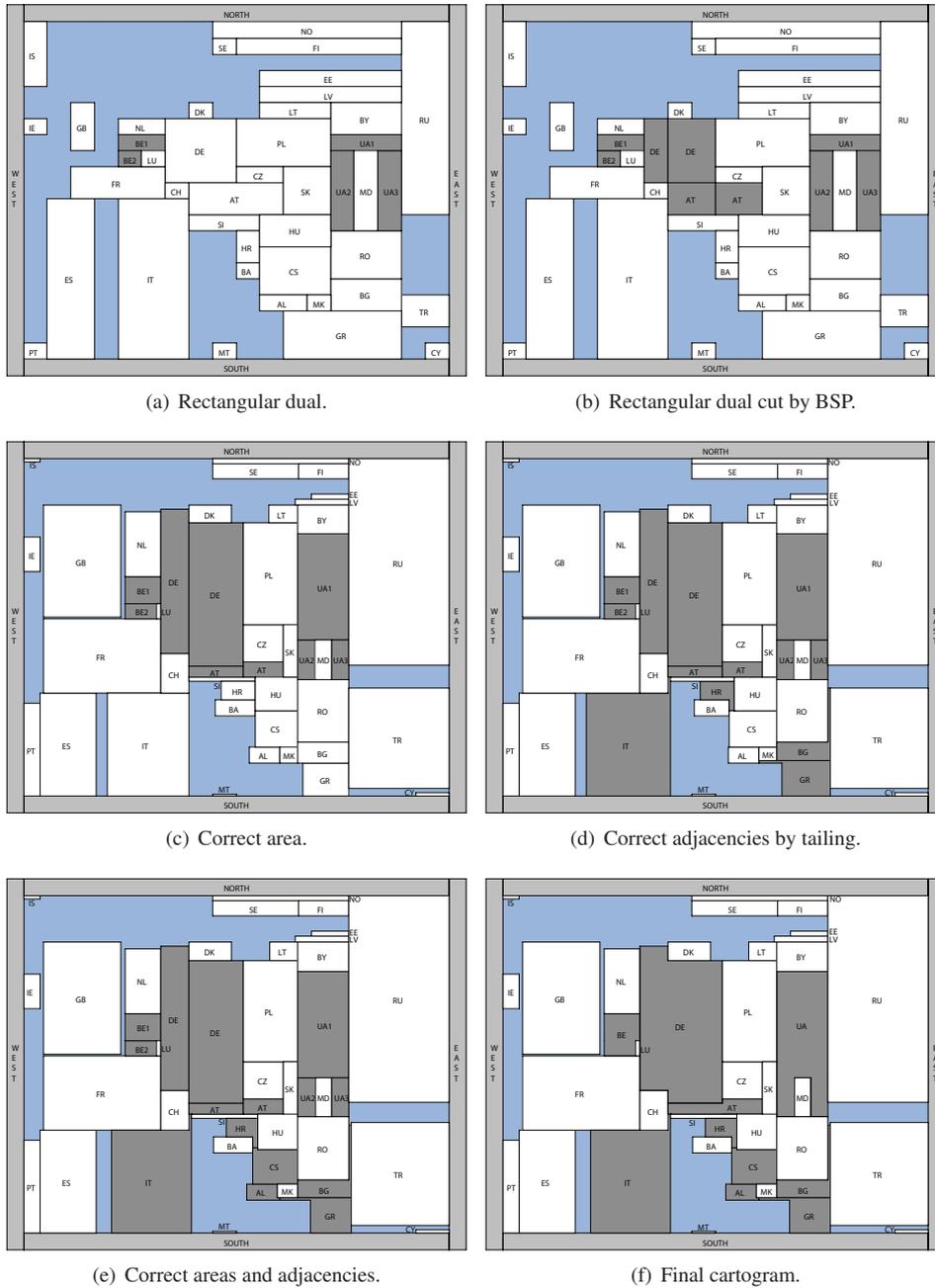
Figure 3.7 Constructing a rectilinear cartogram for Europe with the theme population.

Chapter 4

Optimizing cased drawings of graphs

4.1   Introduction

Drawings of non-planar graphs necessarily contain edge crossings. The vertices of a drawing are commonly marked with a disk, but it can still be difficult to detect a vertex within a dense cluster of edge crossings. *Edge casing* is a well-known method—used, for example, in electrical drawings, when depicting knots, and, more generally, in information visualization—to alleviate this problem and to improve the readability of a drawing. A *cased drawing* orders the edges of each crossing and interrupts the lower edge in an appropriate neighborhood of the crossing. One can also envision that every edge is encased in a strip of the background color and that the casing of the upper edge covers the lower edge at the crossing. See Fig. 4.1 for an example. In graphics, this also referred to as the haloed line effect [1].

If there are no application specific restrictions that dictate the order of the edges at each crossing, then we can in principle choose freely how to arrange them. Certain orders
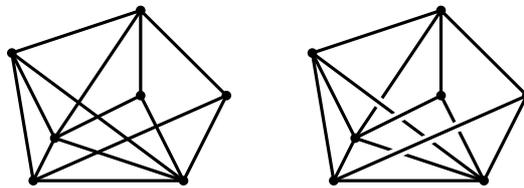
Figure 4.1 Normal and cased drawing of a graph.

will lead to a more readable drawing than others. In this chapter we formulate several optimization criteria that try to capture the concept of a "good" cased drawing. Further, we address the algorithmic question of how to turn a given drawing into an optimal cased drawing.

Definitions.   Let G be a graph with n vertices and m edges and let D be a drawing of G with k crossings. We want to turn D into a cased drawing where the width of the casing is given in the variable *casingwidth*. To avoid covering a vertex with the casing of an edge we assume that no vertex v of D lies on (or very close to) an edge e of D unless v is an endpoint of e. Further, no more than two edges of D cross in one point and any two crossings are far enough apart so that the casings of the edges involved do not interfere. With these assumptions we can consider crossings independently. Without these restrictions the problem changes significantly—optimization problems that are solvable in polynomial time can become NP-hard. In Section 4.5 we discuss in detail the effects of removing the restrictions on the input.

We define the *edge crossing graph G_{DC}* for D as follows. G_{DC} contains a vertex for every edge of D and an edge for any two edges of D that cross. Let C be a crossing between two edges e_1 and e_2. In a cased drawing either e_1 is drawn on top of e_2 or vice versa. If e_1 is drawn on top of e_2 then we say that C is a *bridge* for e_1 and a *tunnel* for e_2. In Fig. 4.2, C_1 is a bridge for e_1 and a tunnel for e_2. The *length* of a tunnel is *casingwidth*$/\sin\alpha$, where $\alpha \leq \pi/2$ is the angle of the edges at the crossing. A pair of consecutive crossings C_1 and C_2 along an edge e is called a *switch* if C_1 is a bridge for e and C_2 is a tunnel for e, or vice versa. In Fig. 4.2, (C_1, C_2) is a switch.

Stacking and weaving.   When we turn a given drawing into a cased drawing, we need to define a drawing order for every edge crossing. We can choose to either establish a global top-to-bottom order on the edges, or to treat each edge crossing individually. We call the first option the *stacking model* and the second one the *weaving model*, since cyclic overlap of three or more edges can occur (see Fig. 4.3).

Quality of a drawing.   Globally speaking, two factors may influence the readability of a cased drawing in a negative way. Firstly, if there are many switches along an edge then it might become difficult to follow that edge. Drawings that have many switches can
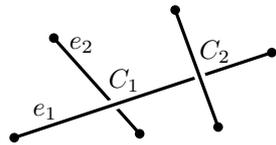
Figure 4.2 Tunnels and bridges.   **Figure 4.3** Stacking and weaving.
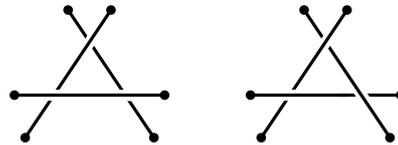
appear somewhat chaotic. Secondly, if an edge is frequently below other edges, then it might become hardly visible. These two considerations lead to the following optimization problems for a drawing *D*.

MINTOTALSWITCHES   Minimize the total number of switches.

MINMAXSWITCHES   Minimize the maximum number of switches for any edge.

MINMAXTUNNELS   Minimize the maximum number of tunnels for any edge.

MINMAXTUNNELLENGTH   Minimize the maximum total length of tunnels for any edge.

MAXMINTUNNELDISTANCE   Maximize the minimum distance between any two consecutive tunnels.

Cased drawings are also used in certain types of mazes where paths may cross (see, for example, Fig. 4.4). For this application the paths between junctions should be difficult to follow and the whole picture should look somewhat chaotic. This motivates the problem of maximizing the number of switches of a drawing *D*.

MAXTOTALSWITCHES   Maximize the total number of switches.

Every stacked drawing is clearly a weaving drawing. However, the weaving model is stronger than the stacking model for MINTOTALSWITCHES in the sense, that there exists a drawing *D* with a weaving drawing that has fewer switches than any stacked drawing of *D*. Fig. 4.5 shows an example of such a drawing. No cased drawing of this graph in the stacking model can be drawn with less than five switches. For, the thickly drawn bundles of $c > 4$ parallel edges must be cased as shown (or its mirror image) else there would be at least c switches in a bundle, the four vertical and horizontal segments must cross the bundles consistently with the casing of the bundles, and this already leads to the four switches that occur as drawn near the midpoint of each vertical or horizontal segment.
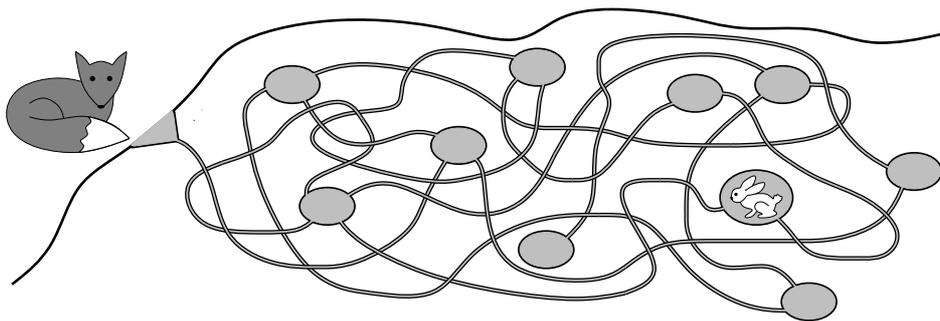
Figure 4.4 "Help the fox eat the rabbit."
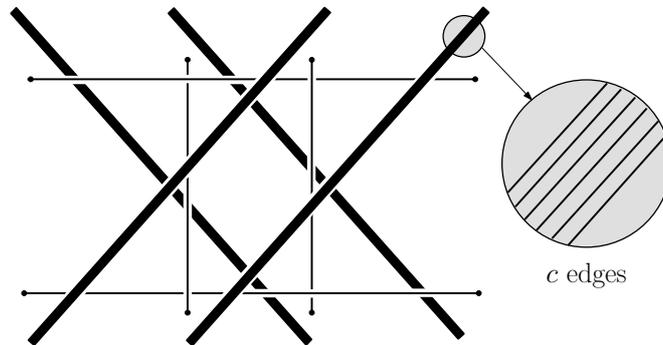
Figure 4.5 Optimal drawing with four switches in the weaving model for MINTOTAL-SWITCHES.

Thus, any deviation from the drawing in the casing of the four crossings between vertical and horizontal segments would create additional switches. However, the drawing shown is not a stacked drawing.

Related work.   If we consider only simple arrangements of line segments in the plane (i.e. arrangements of segments in the plane such that the segments intersect only in interior points and no three segments pass through the same point) as our initial drawing, then there is a third model to consider, an intermediate between stacking and weaving: drawings which are plane projections of line segments in three dimensions. We call this model the *realizable model*. Clearly every cased drawing in the stacking model is also a drawing in the realizable model, but not every cased drawing in the weaving model can be realized (see [55]). The optimal drawing in Fig. 4.5 can be realized, hence the realizable model is stronger than the stacking model.

Fig. 4.6 shows that the weaving model is stronger than the realizable model for MINTO-TALSWITCHES—no cased drawing of this graph in the realizable model can be drawn with less than 13 switches. The reasoning is quite similar to the one employed for the construction in Fig. 4.5. The thickly drawn bundles of $c > 12$ parallel edges must be cased as shown (or its mirror image) else there would be at least c switches in each bundle. The 8 vertical and horizontal "normal" edges must cross the bundles consistently with the casing of the bundles, and this already leads to the 12 switches as drawn in this figure. Thus, any deviation from the drawing in the casing of the 16 crossings between the normal edges would create additional switches. However, the normal edges as drawn here constitute a perfect 4×4 weaving which can not be realized (see [55]).

Organization.   For many of the problems described above, we either find polynomial time algorithms or NP-hardness results in both the stacking and weaving models. We summarize our results in Table 4.1. All our bounds assume that $m = \Omega(n)$ and that $k = \Omega(m)$. Furthermore, we assume that the input drawing is given as an arrangement, in particular,
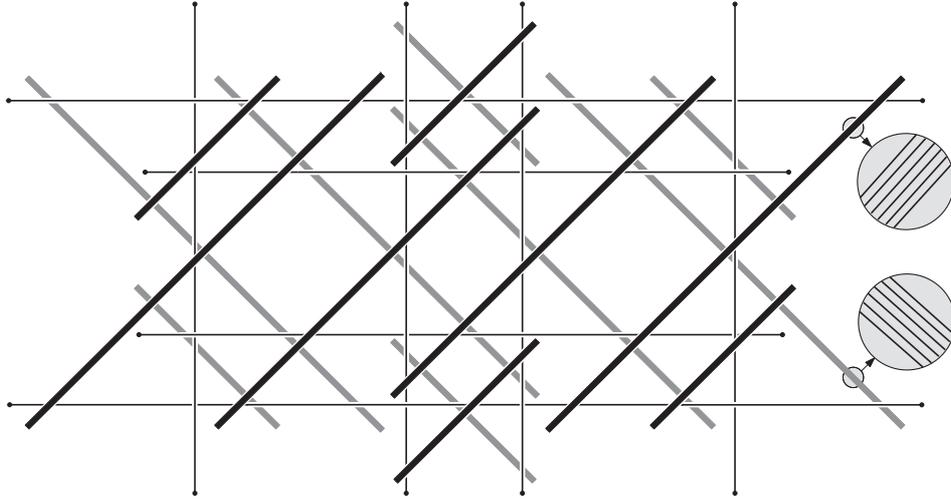
Figure 4.6 Optimal drawing with 12 switches in the weaving model for MINTOTAL-SWITCHES.

all crossings are given as vertices of this arrangement. Our results are derived for straight line input drawings, but several of them also generalize to curved drawings. Section 4.2 presents the results concerning the optimization problems that seek to minimize the number of switches, Section 4.3 gives our results on maximizing the number of switches, and Section 4.4 discusses our solutions to the optimization problems that concern the tunnels. In Section 4.5 we show that MINTOTALSWITCHES becomes NP-hard in both the weaving and the stacking model if we allow more than three edges to cross in one point. We conclude with some open problems.

Model	Stacking	Weaving
MINTOTALSWITCHES	*open*	$O((q+1)k+q^{5/2}\log^{3/2}k)$
MAXTOTALSWITCHES	*open*	$O((t+1)k+t^{5/2}\log^{3/2}k)$
MINMAXSWITCHES	*open*	*open*
MINMAXTUNNELS	$O(m\log m+k)$ *exp.*	$O(m^4)$
MINMAXTUNNELLENGTH	$O(m\log m+k)$ *exp.*	NP-hard
MAXMINTUNNELDISTANCE	$O((m+k)\log m)$ *exp.*	$O((m+K)\log m)$ *exp.*

Table 4.1 Table of results: n is the number of vertices, m is the number of edges, $k = O(m^2)$ is the number of crossings, $K = O(m^3)$ is the total number of pairs of crossings on the same edge, $q = O(k)$ is the number of *odd face polygons*, and $t = O(k)$ is the number of *s-odd face polygons*.

4.2   Minimizing switches

In this section we discuss results related to the MINTOTALSWITCHES and MINMAX-SWITCHES problems. We first discuss some combinatorial results giving simple bounds on the number of switches needed, and recognition algorithms for graphs needing no switches. As we know little about these problems for the stacking model, all results stated in this section will be for the weaving model.

Lemma 4.1 *Given a drawing D of a graph we can turn D into a cased drawing without any switches if and only if the edge crossing graph G_{DC} is bipartite.*

Corollary 4.2 *Given a drawing D of a graph we can decide in $O((n+m)\log(n+m))$ time whether D can be turned into a cased drawing without any switches.*

Proof. We apply the bipartiteness algorithm of [26]. Note that this does not construct the arrangement, so the time bound does not involve k, the number of crossings in the drawing.   □

Define a *vertex-free cycle* in a drawing of a graph G to be a face f formed by the arrangement of the edges in the drawing, such that there are no vertices of G on the boundary of f. An *odd vertex-free cycle* is a vertex-free cycle composed of an odd number of segments of the arrangement.

Lemma 4.3 *Let f be an odd vertex-free cycle in a drawing D. Then in any casing of D, there must be a switch on one of the segments of f.*

Proof. Unless there is a switch, the segments must alternate between those that cross above the previous segment, and those that cross below the previous segment. However, this alternation cannot continue all the way around an odd cycle, for it would end up in an inconsistent state from how it started.   □

Lemma 4.4 *Given a drawing D of a graph the minimum number of switches of any cased drawing obtained from D is at least half of the number of odd vertex-free cycles in D.*

Proof. Let o be the number of odd vertex-free cycles in D. By Lemma 4.3, each odd vertex-free cycle must have a switch on one of its segments. Choose one such switch for each cycle; then each segment belongs to at most two vertex-free cycles, so these choices group the odd cycles into pairs of cycles sharing a common switch, together with possibly some unpaired cycles. The number of pairs and unpaired cycles must be at least $o/2$, so the number of switches must also be this large.   □

Lemma 4.5 *For any n large enough, a drawing of a graph G with n vertices and $O(n)$ edges exists for which any crossing choice gives rise to $\Omega(n^2)$ switches.*

Proof. A construction with three sets of parallel lines, each of linear size, gives $\Omega(n^2)$ vertex-free triangles, and each triangle gives at least one switch (see Fig. 4.7). □

Lemma 4.6 *For any n large enough, a drawing of a graph G with n vertices and $O(n^2)$ edges exists for which any crossing choice gives rise to $\Omega(n^4)$ switches.*

Proof. We build our graph as follows: make a very elongated rectangle, place $n/6$ vertices equally spaced on each short edge, and draw the complete bipartite graph. This graph has $(n/6)^2$ edges. One can prove that there is a strip parallel to the short side of the rectangle, such that the parts of the edges inside the strip behave in the same way as parallel ones do with respect to creating triangles when overlapped the way it is described in the previous lemma. This gives us the desired graph with $\Omega(n^4)$ triangles, and hence with $\Omega(n^4)$ switches. □

We define a *degree-one graph* to be a graph in which every vertex is incident to exactly one edge; that is, it must consist of a collection of disconnected edges (see Fig. 4.8).

Lemma 4.7 *Let D be a drawing of a graph G. Then there exists a drawing D' of a degree-one graph G', such that the edges of D correspond one-for-one with the edges of D', casings of D correspond one-for-one to casings of D', and switches of D correspond one-for-one with switches of D'.*

Proof. Form G' by placing a small circle around each vertex of G. Given an edge $e = (u, v)$ in G, let u_e be the point where e crosses the circle around u and similarly let v_e be the point where e crosses the circle around v. Form D' and G' by replacing each edge $e = (u, v)$ in G by the corresponding edge (u_e, v_e), drawn as the subset of edge e connecting those points. As these replacements do not occur between any two crossings along any edge, they do not affect the switches on the edge. Both drawings have the same set of crossings, and any switch in a casing of one drawing gives rise to a switch in the corresponding casing of the other drawing. □
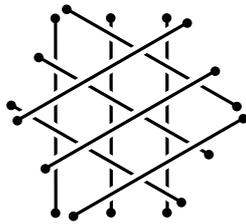
Figure 4.7 $O(n)$ edges and $\Omega(n^2)$ triangles.
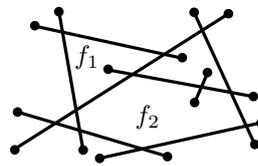
Figure 4.8 A degree-one graph, f_1 is an odd polygon and f_2 is an even polygon.

In a drawing of a degree-one graph, define a *polygon* to be a sequence of segments of the arrangement formed by the drawing edges that forms the boundary of a simple polygon in the plane. Define a *face polygon* to be a polygon that forms the boundary of the closure of a face of the arrangement; note that there may be edges drawn in the interior of this polygon, as long as they do not separate it into multiple components.

Lemma 4.8 *In a drawing of a degree-one graph, there can be no vertex on any segment of a polygon.*

Proof. We have already required that no vertex can lie on an edge unless it is the endpoint of an edge. And, if a segment contains the endpoint of an edge, it cannot continue past the endpoint to form the boundary of a polygon.   □

Note, however, that a polygon can contain vertices in its interior. Define the *complexity* of a polygon to be the number of segments forming it, plus the number of graph vertices interior to the polygon. We say that a polygon is *odd* if its complexity is an odd number, and *even* if its complexity is an even number (see Fig. 4.8).

Lemma 4.9 *Let p be a polygon in a drawing of a degree-one graph. Then, modulo two, the complexity of p is equal to the sum of the complexities of the face polygons of faces within p.*

Proof. Each segment of p contributes one to the complexity of p and one to the complexity of some face polygon. Each vertex within p contributes one to the complexity of p and one to the complexity of the face that contains it. Each segment within the interior of p either separates two faces, and contributes two to the total complexity of faces within p, or does not separate any face and contributes nothing to the complexity. Thus in each case the contribution to p and to the sum of its faces is the same modulo two.   □

Lemma 4.10 *Let p be an odd polygon in a drawing of a degree-one graph. Then there exists an odd face polygon in the same drawing.*

Proof. By Lemma 4.9, the complexity of p has the same parity as the sum of the complexities of its faces. Therefore, if p is odd, it has an odd number of odd faces, and in particular there must be a nonzero number of odd faces.   □

Lemma 4.11 *Let D be a drawing of a degree-one graph. Then D has a casing with no switches if and only if it has no odd face polygon.*

Proof. As we have seen, D has a casing with no switches if and only if the edge crossing graph is bipartite. This graph is bipartite if and only if it has no odd cycles, and an odd cycle in the edge crossing graph corresponds to an odd polygon in D. For, if C is an odd cycle in the edge crossing graph, it must lie on a polygon p of D. Each crossing

in C contributes one to the complexity of this polygon. Each edge of D that crosses p without belonging to C either crosses it an even number of times (contributing that number of additional segments to the complexity of p) and has both endpoints inside p or both outside p, or it crosses an odd number of times and has one endpoint inside p; thus, it contributes an even amount to the complexity of p. Thus, p must be an odd polygon. By Lemma 4.10, there is an odd face polygon in D. Conversely, any odd face polygon in D can be shown to form an odd cycle in the edge crossing graph.   □

Theorem 4.12 *A solution for* MINTOTALSWITCHES *in the weaving model can be found in time* $O((q+1)k + q^{5/2}\log^{3/2}k)$, *where k denotes the number of crossings in the input drawing and q denotes the number of its odd face polygons.*

Proof. Let D be the drawing which we wish to case for the minimum number of switches. By Lemma 4.7, we may assume without loss of generality that each vertex of D has degree one.

We apply a solution technique related to the Chinese Postman problem, and also to the problem of via minimization in VLSI design [11]: form an auxiliary graph G^o, and include in G^o a single vertex for each odd face polygon in D. Also include in G^o an edge connecting each pair of vertices, and label this edge by the number of segments of the drawing that are crossed in a path connecting the corresponding two faces in D that crosses as few segments as possible. We claim that the minimum weight of a perfect matching in G^o equals the minimum total number of switches in any casing of D.

In one direction, we can case D with a number of switches equal to or better than the weight of the matching, as follows: for each edge of the matching, insert a small break into each of the segments in the path corresponding to the edge. The resulting broken arrangement has no odd face cycles, for the breaks connect pairs of odd face cycles in D to form larger even cycles. Therefore, by Lemma 4.11, we can case the drawing with the breaks, without any switches. Forming a drawing of D by reconnecting all the break points adds at most one switch per break point, so the total number of switches equals at most the weight of the perfect matching.

In the other direction, suppose that we have a casing of D with a minimum number of switches; we must show that there exists an equally good matching in G^o. To show this, consider the drawing formed by inserting a small break in each segment of D having a switch. This eliminates all switches in the drawing, so by Lemma 4.11, the modified drawing has no odd face polygons. Consider any face polygon in the modified drawing; by Lemma 4.10 it must include an even number of odd faces in the original drawing. Thus, the odd faces of D are connected in groups of evenly many faces in the modified drawing, and within each such group we can connect the odd faces in pairs by paths of breaks in the drawing, giving a matching in G^o with total weight at most equal to the number of switches in D.

The number of vertices of the graph G^o is $O(q)$, where q is the number of odd face polygons in D. We can construct G^o in time $O(k+qk)$ where k is the number of crossings in

D by using breadth-first search in the arrangement dual to *D* to find the distances from each vertex to all other vertices. A minimum weight perfect matching in a complete weighted graph with integer weights bounded by k can be found in time $O(q^{5/2} \log^{3/2} k)$ using the algorithm of Gabow and Tarjan [32]. Therefore the time for this algorithm is $O((q+1)k + q^{5/2} \log^{3/2} k)$. $\qquad\square$

For MINMAXSWITCHES, we have the following weaker result.

Theorem 4.13 *If the edge crossing graph G_{DC} is planar, then we can assure at most four switches along any edge in the weaving model.*

Proof. Find an edge e that intersects at most five others, take e out, and recursively treat the rest. Put e back in such a way that it does not increase the switch count of any of the edges it intersects. Edge e itself will have at most four switches. $\qquad\square$

4.3   Maximizing switches

In this section we describe a solution for the MAXTOTALSWITCHES problem in the weaving model.

We define a *hitch* along an edge e in a cased drawing to be a pair of consecutive crossings along e such that the crossings are either both bridges or both tunnels for e. Hence every pair of consecutive crossings along an edge e in a cased drawing forms either a hitch or a switch for e. Let n_p be the number of all pairs of consecutive crossings along all the edges of a drawing D, and let n_t and n_s be the number of hitches and switches, respectively, in a given casing of D. Clearly $n_p = n_s + n_t$. Thus the problem of finding the maximal number of switches for a cased drawing is equivalent to the problem of finding the minimal number of hitches. We call a cased drawing without hitches a *hitch-free casing* or a *perfect weaving*. The cased drawing obtained from D_c by turning bridges into tunnels and vice versa is the *inversion* of D_c and denoted by $D_{\bar{c}}$. If D_c is a perfect weaving then so is $D_{\bar{c}}$.

Thanks to Lemma 4.7 we can again limit ourselves to considering degree-one graphs only. Let p be a vertex of a polygon (which is defined as in the previous section). We say that p is a *side vertex* of the polygon if the edges of the polygon form a $180°$ angle at p, and p is called a *corner vertex* otherwise (see Fig. 4.9: side vertices are depicted as white squares, corner vertices as black squares). We define the *s-complexity* of a polygon as the number of its side vertices. A polygon of odd s-complexity is called *s-odd*.

Lemma 4.14 *Let f be an s-odd polygon of a degree-one graph drawing D, then for any casing of D one of the edges forming f has a hitch.*

Proof. Let $e_1, \ldots, e_k$ be the edges of D forming f in clockwise order. Then the first crossing of e_i along f in clockwise order is $c_{i,1} = e_i \cap e_{i-1}$, for $1 < i \le k$, $c_{1,1} = e_1 \cap e_k$,

and the last crossing of e_i along f in clockwise order is $c_{i,2} = e_i \cap e_{i-1}$ for $1 \le i < k$, $c_{k,2} = e_k \cap e_1$.

Assume that a cased drawing D_c of D exists where none of e_i, $1 \le i \le k$, has a hitch. Then if an edge e_i has an odd number of side vertices between $c_{i,1}$ and $c_{i,2}$, then $c_{i,1}$ and $c_{i,2}$ are cased the same way, otherwise they are cased differently. By symmetry we may assume that $c_{1,1}$ is a bridge. Since in total f has an odd number of side vertices, $c_{k,2}$ must be a bridge as well. But this is impossible because $c_{1,1}$ and $c_{k,2}$ are the same crossing. Hence for any cased drawing of D there is an edge in f that has a hitch. □

Lemma 4.15 *Let f be a polygon in a drawing of a degree-one graph. Then the parity of the number of side vertices of f is equal to the parity of the number of graph vertices inside f.*

Proof. If an edge e of D contains part of the boundary of f we call it a *boundary edge* for f, otherwise we say that e is an *external edge* for f. Let e be an external edge for f. Each crossing of e with the boundary of f is a side vertex of f—see Fig. 4.9. If e intersects the boundary of the polygon an odd number of times, then it has only one vertex inside it. If it intersects the boundary of f an even number of times, then it either has both vertices inside f or both outside f. Hence the parity of the number $n_{s,1}$ of times that edges external for f cross f's boundary is equal to the parity of the number $n_{v,1}$ of the end vertices these edges contribute to the interior of f.

Now let e be a boundary edge for f. We say that a side vertex v is charged to e if $v \in e$ but the segments of e incident to v are not part of f's boundary, see Fig. 4.9 for an example. This way each side vertex formed by a pair of boundary edges is only charged to one of the edges.

If both ends of e lie outside f then an even number of side vertices of f is charged to e. The same happens if both ends of e lie inside f. If f contains only one end of e then e is charged by an odd number of side vertices of f. Hence the number $n_{s,2}$ of side vertices of f formed by pairs of boundary edges and the number $n_{v,2}$ of vertices contributed by
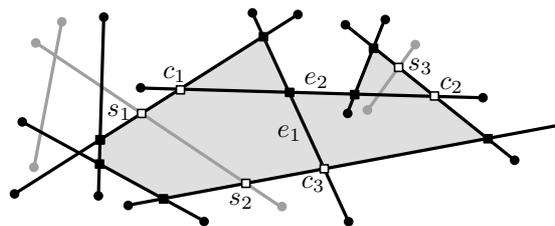
Figure 4.9 Gray edges do not contribute to the boundary of the gray polygon f; s_1, s_2 and s_3 are the side vertices formed by external edges that cross the boundary of f, and c_1, c_2 and c_3 are the side vertices formed by pairs of boundary edges for f. c_1 and c_2 are charged to e_2 and c_3 is charged to e_1.

boundary edges to the interior of f have the same parity. The total number of side vertices of f is $n_{s,1} + n_{s,2}$, and the total number of vertices inside f is $n_{v,1} + n_{v,2}$. These numbers have the same parity. $\square$

Lemma 4.16 *If a drawing has an s-odd polygon, then it also has an s-odd face polygon.*

Proof. Let f be an s-odd polygon in a drawing D, and assume it has k vertices inside it. Then k is the sum of the numbers of the vertices inside all face polygons inside f. Since k is odd, there is a face polygon of odd s-complexity inside f. $\square$

Consider an arrangement of closed or infinite curves in the plane. We can naturally extend the notion of a cased drawings to such an arrangement.

Lemma 4.17 *Let A be an arrangement of closed or infinite curves in the plane where every vertex has even degree. Then A has a perfect weaving.*

Proof. It is well known (see [34]) that the faces of the arrangement A can be two-colored such that no two edge-adjacent faces have the same color. Consider such a two-coloring of the faces into red and blue faces. At any crossing c of two curves, choose the order such that the curve that has red to its left when going away from c is above the curve that has red to its right. Then any edge bounding any face of the arrangement has a tunnel and a bridge. $\square$

Lemma 4.18 *A drawing D of a degree-one graph where all vertices lie on the outer face has a perfect weaving.*

Proof. We can see such a graph as an arrangement of lines in the plane (by extending the edges beyond their vertices). Then D has a perfect weaving by the previous lemma. $\square$

Lemma 4.19 *A drawing D of a degree-one graph has a hitch-free cased drawing if and only if it has no s-odd polygons.*

Proof. Let D be a degree-one graph drawing without s-odd face polygons. For each internal face of D we pairwise connect the vertices inside it. The edges that have the vertices on the outer face we extend infinitely beyond their outer face vertices. The obtained arrangement of curves in the plane has a perfect weaving by Lemma 4.17. It is easy to see that removing the added connections leaves us with a perfect weaving of D. In the other direction, a hitch-free drawing has no s-odd polygons by Lemma 4.14. $\square$

Given a drawing, the casing with maximal number of switches can be found in almost exactly the same way as the drawing with minimal number of switches.

Theorem 4.20 MAXTOTALSWITCHES *in the weaving model can be solved in time $O((t + 1)k + t^{5/2} \log^{3/2} k)$, where k denotes the number of crossings in the input drawing and t denotes the number of its s-odd face polygons.*

Proof. We apply the same technique used in the proof of Theorem 4.12. We construct an auxiliary graph G^s that contains a vertex for each s-odd face polygon. Then we connect each pair of vertices with an edge and label them in exactly the same way as the edges of graph G^o. Then the minimum total number of hitches in any casing of D is equal to the minimum weight perfect matching of G^s.

First we show that we can always have a casing with at most as many hitches as the weight of the matching. For each edge of the matching, consider its path in the arrangement and the edges that it crosses. For each crossed edge e of the arrangement we add a short edge that crosses only edge e, see Fig. 4.10. We refer to these little edges and their end vertices as *helper* edges and vertices. The resulting graph has no s-odd faces and can be cased as a perfect weaving. We remove the helper edges and vertices, getting at most one hitch per removed edge. The resulting casing has as many hitches as there were helper edges, hence the number of hitches in the resulting graph is at most the weight of the matching.

Now assume we have a casing with minimum number of hitches. We place a helper edge between the edges forming the hitch along an edge e such that the helper edge crosses only e and case the crossing along e opposite with respect to the hitches along e. The obtained drawing is cased and has no more hitches. Hen ce it has no s-odd face polygons by Lemma 4.19.

We added an even number of helper vertices in total: an odd number of helper vertices to each s-odd face and an even number of helper vertices to each s-even face. Now for each face that has more than one helper vertex we pair the helper vertices arbitrarily and connect each pair with a so-called *connector* edge. As a result we have a set of paths of helper vertices, where helper and connector edges alternate in each path. Each s-odd face of our arrangement contains exactly one end point of such a path. Each path is non-trivial (it contains at least two vertices). Hence the set of paths forms a perfect matching with weight equal to the number of hitches in the casing. □

(a) (b) (c)
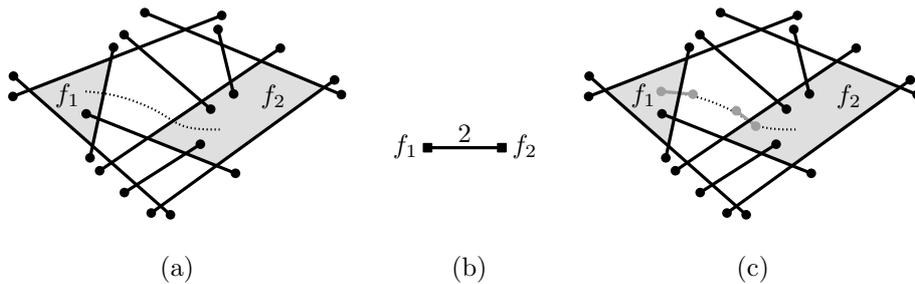
Figure 4.10 (a) A degree-one graph drawing D with two s-odd faces f_1 and f_2. The dotted line represents the path connecting them that crosses minimum number of edges possible. (b) The auxiliary graph G^s for D. (c) D with (gray) helper edges added along the path connecting f_1 and f_2.

4.4 Minimizing tunnels

In this section we present three algorithms that solve MINMAXTUNNELS, MINMAX-TUNNELLENGTH, and MAXMINTUNNELDISTANCE in the stacking model. We also present algorithms for MINMAXTUNNELS and MAXMINTUNNELDISTANCE in the weaving model. MINMAXTUNNELLENGTH is NP-hard in the weaving model.

4.4.1 Stacking model

In the stacking model, some edge e has to be bottommost. This immediately gives the number of tunnels of e, the total length of tunnels of e, and the shortest distance between two tunnels of e. The idea of the algorithm is to determine for each edge what its value would be if it were bottommost, and then choose the edge that is best for the optimization to be bottommost (smallest value for MINMAXTUNNELS and MINMAX-TUNNELLENGTH, and largest value for MAXMINTUNNELDISTANCE). The other $m-1$ edges are stacked iteratively above this edge. It is easy to see that such an approach indeed maximizes the minimum, or minimizes the maximum. We next give an efficient implementation of the approach. The idea is to maintain the values of all not yet selected edges under consecutive selections of bottommost edges instead of recomputing them.

We start by computing the arrangement of edges in $O(m\log m + k)$ expected time, for instance using Mulmuley's algorithm [52]. This allows us to determine the values for all edges in $O(k)$ additional time.

For MINMAXTUNNELS and MINMAXTUNNELLENGTH, we keep all edges in a Fibonnacci heap on this value. One selection involves an EXTRACT-MIN, giving an edge e, and traversing e in the arrangement to find all edges it crosses. For these edges we update the value and perform a DECREASE-KEY operation on the Fibonnacci heap. For MIN-MAXTUNNELS we decrease the value by one and for MINMAXTUNNELLENGTH we decrease by the length of the crossing, which is *casingwidth*$/\sin\alpha$, where α is the angle the crossing edges make. For MINMAXTUNNELS and MINMAXTUNNELLENGTH this is all that we need. We perform m EXTRACT-MIN and k DECREASE-KEY operations. The total traversal time along the edges throughout the whole algorithm is $O(k)$. Thus, the algorithm runs in $O(m\log m + k)$ expected time.

For MAXMINTUNNELDISTANCE we use a Fibonnacci heap that allows EXTRACT-MAX and INCREASE-KEY. For the selected edge we again traverse the arrangement to update the values of the crossing edges. However, we cannot update the value of an edge in constant time for this optimization. We maintain a data structure for each edge that maintains the minimum tunnel distance in $O(\log m)$ time under updates. The structure is an augmented balanced binary search tree that stores the edge parts in between consecutive

crossings in its leaves. Each leaf stores the distance between these crossings. Each internal node is augmented such that it stores the minimum distance for the subtree in a variable. The root stores the minimum distance of the edge if it were the bottommost one of the remaining edges. An update involves merging two adjacent leaves of the tree and computing the distance between two crossings. Augmentation allows us to have the new minimum in the root of the tree in $O(\log m)$ time per update. In total this takes $O(m \log m + k \log m)$ expected time.

Theorem 4.21 *Given a straight-line drawing of a graph with n vertices, $m = \Omega(n)$ edges, and k edge crossings, we can solve* MINMAXTUNNELS *and* MINMAXTUNNELLENGTH *in $O(m \log m + k)$ expected time and* MAXMINTUNNELDISTANCE *in $O(m \log m + k \log m)$ expected time in the stacking model.*

4.4.2 Weaving model

In the weaving model, the polynomial time algorithm for MINMAXTUNNELS comes from the fact that the problem of directing an undirected graph, and minimizing the maximum indegree, can be solved in time quadratic in the number of edges [65]. We apply this on the edge crossing graph of the drawing, and hence we get $O(m^4)$ time. For minimizing tunnel length per edge, we can show:

Theorem 4.22 MINMAXTUNNELLENGTH *is NP-hard in the weaving model.*

Proof. The reduction is from PLANAR 3-SAT, shown NP-hard by Lichtenstein [48]. The reduction is similar to the one for maximizing minimum visible perimeter length in sets of opaque disks of unit size [10]. Note that the proof implies that no PTAS exists. The reduction only uses edges that intersect two or three other edges, so restricting the number of intersections per edge to be constant leaves the problem NP-hard. Also, the number of orientations of edges is constant.

A cased drawing of a set of line segments has property (A) if every line segment has at most two tunnels at crossings with a perpendicular segment, or one tunnel at a crossing with a non-perpendicular segment. Our reduction is such that a PLANAR 3-SAT instance is satisfiable if and only if a set of line segments has a cased drawing with property (A).

We arrange a set of line segments of equal length, using only four orientations. The slopes are -4, $-\frac{1}{4}$, $+\frac{1}{4}$, and $+4$. If two perpendicular line segments cross, then one has tunnel length equal to the width w of the casing at the crossing. If two other line segments cross, then one edge has tunnel length $w/\sin(\gamma) = 2,125 \cdot w$ at the crossing, where $\gamma = 2 \cdot \arctan(\frac{1}{4})$ is the (acute) angle between the line segments. Therefore, a cased drawing with property (A) has tunnel length at most $2,125 \cdot w$, whereas a cased drawing that does not satisfy property (A) has an edge that has tunnel length at least $3 \cdot w$. This shows the direct relation between property (A) and MINMAXTUNNELLENGTH, and provides the gap that shows that no PTAS exists.

A Boolean variable x_i is modeled by a cycle of crossing line segments as in Fig. 4.11. Along the cycle, crossings alternate between perpendicular and non-perpendicular, and hence it has even length. The variable satisfies property (A) iff the cycle has cyclic overlap, which can be clockwise or counterclockwise. One state is associated with $x_i = \text{TRUE}$, the other is associated with $x_i = \text{FALSE}$. In each state, the line segments of the cycle alternate in allowing an additional, perpendicular line segment to have a bridge over the line segment of the cycle. In the figure, where the cycle is in the TRUE-state, the line segments with slope $+\frac{1}{4}$ and $+4$ allow such an extra tunnel under a line segment that is not from the cycle. If the cycle is in the FALSE-state, the line segments with slope -4 and $-\frac{1}{4}$ allow the extra tunnel. We use the line segments of slope $-\frac{1}{4}$ to make connections and channels to clauses where $\overline{x_i}$ occurs, and the line segments with slope $+\frac{1}{4}$ for clauses where x_i occurs. Note that the variable can be made larger easily to allow more connections, in case the variable occurs in many clauses.

Channels are formed by line segments that do not cross perpendicularly. So any line segment of the channel can have a tunnel at at most one of its two crossings, or else property (A) is violated. Note that a sequence of crossing line segments with slopes such as $-4, +4, +\frac{1}{4}, -\frac{1}{4}$ gives a turn in the channel. The exact position of the crossing is not essential and hence we can easily reach any part of the plane with a channel, and ending with a line segment of any orientation.
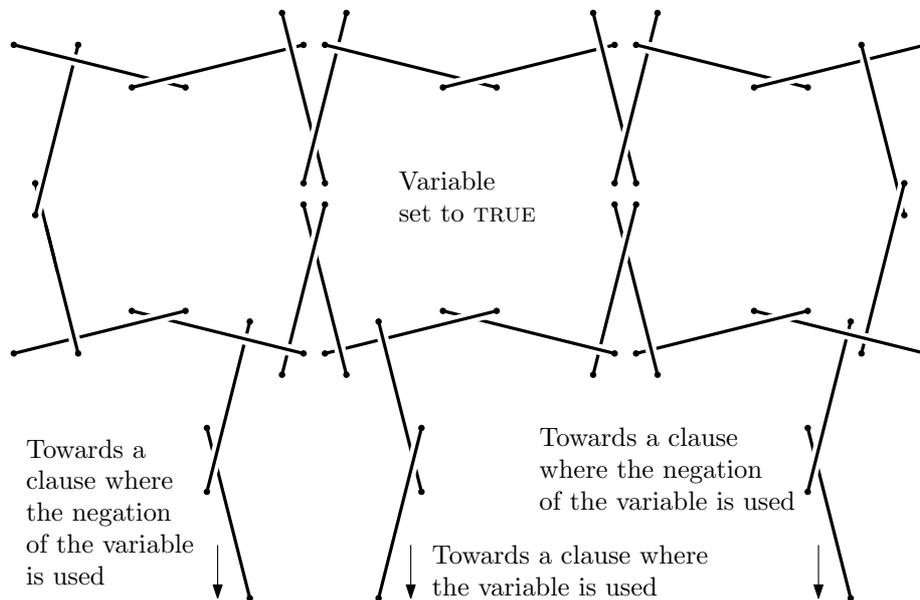
Figure 4.11 Boolean variable and the connection of channels.

From a variable that makes the clause TRUE

From variables that do not make the clause TRUE
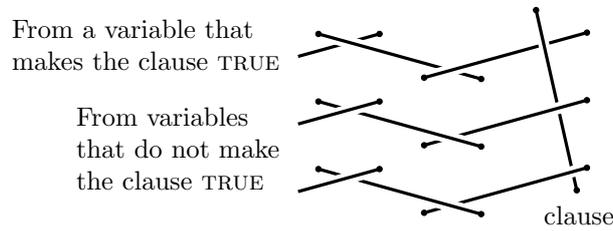
clause

Figure 4.12 A clause construction.

A 3-SAT clause is formed by a single line segment that is crossed perpendicularly by three other line segments, see Fig. 4.12. Property (A) holds if the clause line segment has at most two tunnels. This corresponds directly to satisfiability of the clause.

With this reduction, testing if property (A) holds is equivalent to testing if the PLANAR 3-SAT instance is satisfiable, and NP-hardness follows. □

In the remainder of this section we show how to solve MAXMINTUNNELDISTANCE. We observe that there are polynomially many possible values for the smallest tunnel distance, and perform a binary search on these, using 2-SAT instances as the decision tool.

We first compute the arrangement of the m edges to determine all crossings. Only distances between two—not necessarily consecutive—crossings along any edge can give the minimum tunnel distance. One edge crosses at most $m-1$ other edges, and hence the number of candidate distances, K, is $O(m^3)$. Obviously, K is also $O(k^2)$. From the arrangement of edges we can determine all of these distances in $O(m \log m + K)$ time. We sort them in $O(K \log K)$ time to set up a binary search. We will show that the decision step takes $O(m+K)$ time, and hence the whole algorithm takes $O(m \log m + K \log K) = O((m+K) \log m)$ time.

Let δ be a value for which we wish to decide whether we can set the crossings of edges such that all distances between two tunnels along any edge is at least δ. For every two edges e_i and e_j that cross with $i < j$, we have a Boolean variable x_{ij}. We associate x_{ij} with TRUE if e_i has a bridge at its crossing with e_j, and with FALSE otherwise. Now we traverse the arrangement of edges and construct a 2-SAT formula. Let e_i, e_j, and e_h be three edges such that the latter two cross e_i. If the distance between the crossings is less than δ, then e_i should not have the crossings with e_j and e_h as tunnels. Hence, we make a clause for the 2-SAT formula as follows (Fig. 4.13): if $i < j$ and $i < h$, then the clause is $(x_{ij} \lor x_{ih})$; the other three cases ($i > j$ and/or $i > h$) are similar. The conjunction of all clauses gives a 2-SAT formula that is satisfiable if and only if we can set the crossings such that the minimum tunnel distance is at least δ. We can construct the whole 2-SAT instance in $O(m+K)$ time since we have the arrangement, and satisfiability of 2-SAT can be determined in linear time [29].
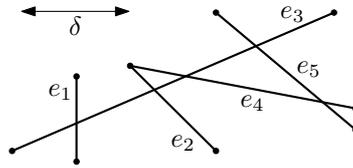
Figure 4.13 The 2-SAT formula $(\bar{x}_{13} \lor \bar{x}_{23}) \land (\bar{x}_{23} \lor x_{34}) \land (\bar{x}_{23} \lor x_{35}) \land (x_{34} \lor x_{35})$.

Theorem 4.23 *Given a straight-line drawing of a graph with n vertices and $m = \Omega(n)$ edges, we can solve* MAXMINTUNNELDISTANCE *in $O((m+K)\log m)$ expected time in the weaving model, where $K = O(m^3)$ is the total number of pairs of crossings on the same edge.*

4.5   Removing the restrictions

The restrictions that we impose on the input drawing—no vertex lies on a or very close to an edge, no more than two edges cross in one point, and crossing are not too close—have a significant influence on the problems that we study. For example, if we allow edges to partly overlap with vertices, then it is natural to always draw the vertex on top with a casing around it. In this case, however, there might be no cased drawing in the stacking model, because three edges that necessarily give cyclic overlap can easily be constructed. Testing if cyclic overlap occurs, and hence, if a cased drawing in the stacking model exists can be done by topological sorting.

If we remove the restriction that edge crossings are not too close, we may have three edges that intersect so closely that it is not possible for the edges to have a tunnel at the one crossing and a bridge at the other. Consequently, three such edges must (locally) be stacked, where one edge has a single bridge over both other edges, and both other edges have a single tunnel. Allowing such "triple crossings" makes the problem MINTOTAL-SWITCHES NP-hard.

Theorem 4.24 *If triple crossings of edges are allowed, then* MINTOTALSWITCHES *is NP-hard in both the weaving and the stacking model.*

Proof. By reduction from PLANAR MAX-2SAT shown NP-hard by [35]. Assume an instance of such a problem which has n Boolean variables and m clauses. The objective is to satisfy as many of the 2SAT clauses as possible. We reduce the instance to an instance of cased drawing such that the number of switches is the same as the number of unsatisfied clauses in the PLANAR MAX-2SAT instance.

A variable is represented by an even cycle of bundles of edges. Each bundle consists of $m+1$ parallel edges that are either horizontal or vertical. A horizontal bundle intersects two vertical bundles, and each vertical bundle intersects two horizontal bundles; see

Fig. 4.14. Since the variables' part of the construction is bipartite, it need not contain any switches. Either all horizontal edges are on top of the vertical ones, or vice versa. This corresponds to the TRUE and FALSE assignment of the variable, respectively.

We tap off a channel from a variable construction with a single vertical edge as in Fig. 4.14. The edge crosses all edges of a horizontal bundle, and to avoid switches, it must be below or above all of them (depending on whether the vertical edges in the variable are below or above). The channel itself is a sequence of single edges that leads from the variable construction to a clause construction. A channel has no switches if the edges alternate in having bridges at both crossings and having tunnels at both crossings. The parity of the number of edges in a channel determines if the channel arrives at a clause with an edge that already has a tunnel, or with an edge that already has a bridge. By using horizontal, vertical, and diagonal edges, we can always get the parity as desired and end with an edge in any orientation. So far, no switches are needed yet in a MINTOTALSWITCHES cased drawing.

A clause consists of a single edge that has a triple crossing with the two channels that come from variables that occur in the clause. The clause edge itself cannot have a switch, because it only has the triple intersection. To avoid switches on the last edges of the
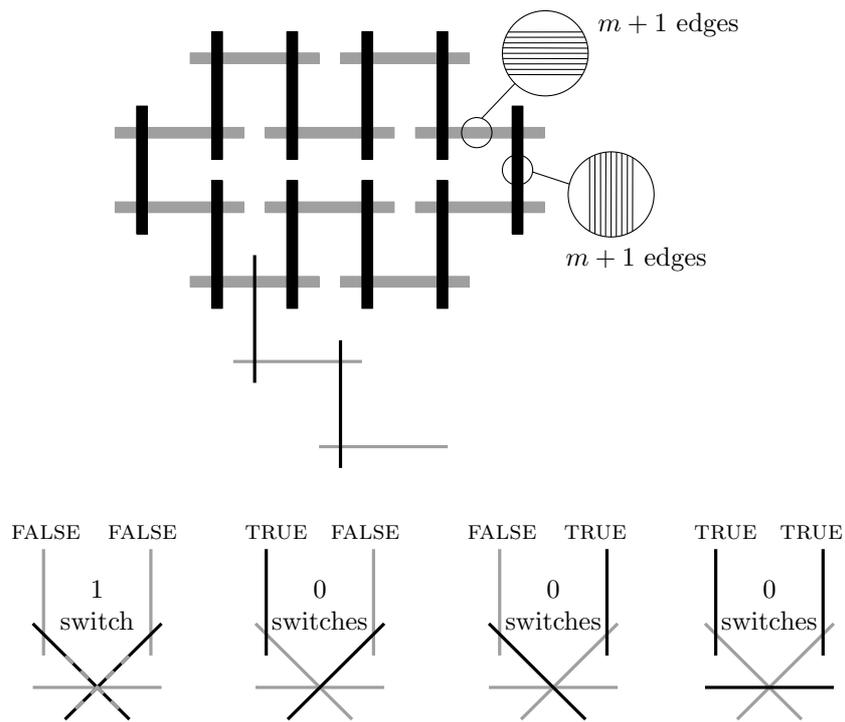
Figure 4.14 A variable, a channel, and four cases of a clause construction.

channels, we can put the clause edge on top if both channel edges already have a tunnel, or we can put the clause edge at the bottom otherwise. If both channel edges already have a bridge, then there is no way to avoid a switch because one of the them will have to go below the other. There will be exactly one switch; in all other cases there are no switches at the clause. So we let the parity of any chain be such that the last channel edge be such that it has a bridge if the variable has a state that gives FALSE in the clause.

The maximum number of switches needed is obviously at most m, the number of clauses of the PLANAR MAX-2SAT instance. We used bundles of $m + 1$ parallel edges in the variables to make sure that a variable cannot have a mixed state without giving at least $m + 1$ switches already. If the bundles were just single edges, then with only two switches, one part of the channels can use the TRUE state and the other part the FALSE state of the variable, possibly satisfying many more clauses. The use of bundles prevents this possibility.

Note that the minimum number of switches will be achieved with a stacked drawing. Hence, the problem is NP-hard in both models.   □

Chapter 5

Realizability of curves as surface boundaries

5.1 Introduction

In this chapter we consider the algorithmic interplay between three types of topological objects: self-crossing curves in the plane, two-dimensional surfaces mapped to the plane in a self-overlapping way, and three-dimensional embeddings of surfaces that generalize the terrains familiar in computational geometry. For more background on the problem that we will study see Section 1.3.

A *surface* or *two-dimensional manifold with boundary* is a compact Hausdorff topological space M such that every point p has a neighborhood homeomorphic to a closed disk. If this homeomorphism maps p to a boundary point of the disk, we call p a *boundary point* of M; the set of boundary points is represented by ∂M. An *immersion* or *local homeomorphism* is a continuous function $i : M \to T$ that, restricted to some neighborhood of every point, is a homeomorphism. We will here be concerned only with the case that $T = \mathbb{R}^2$, in which case we say that the surface M is *immersed in the plane*. If M is topologically a disk, we call $i(M)$ an *immersed disk*, but immersions of other types of manifold are also possible.

An *embedding* of a surface M into some space S is a closed subspace of S that is the image of M under a one-to-one continuous function $e : M \mapsto S$, the inverse of which is also continuous. A *terrain* is a surface embedded in space $\mathbb{R}^3$ such that every vertical line $\{(x,y,z) \mid x = c_1, y = c_2\}$ intersects it at most once. We are interested here in a localized version of this property: an *generalized terrain* is a surface M embedded in space $\mathbb{R}^3$ such that every point of M has a neighborhood the image of which is a terrain. Intuitively, such a surface is embedded in such a way that it has no vertical tangent lines, so that
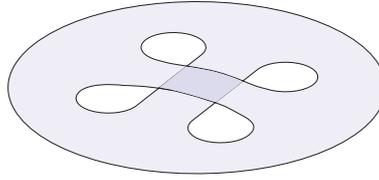
Figure 5.1 A generalized terrain, as viewed from above.

it has a consistent up-down orientation at every point. We will also call such a surface an *embedded surface*, when it does not introduce any confusion—see Figure 5.1 for an example. Intuitively, every generalized terrain can be constructed by gluing terrains along their boundaries. As with immersions, if M is topologically a disk, we call $e(M)$ an *embedded disk*.

If i is an immersion, $i(\partial M)$ is a curve in the plane, which we call the *boundary of the immersion*; with a suitable general-position assumption on i, this curve intersects itself only at proper pairwise crossings [50]. And if e is an embedding of a generalized terrain, we may project it into the plane to form an immersion: let $\pi_z(x, y, z) = (x, y)$, then $i(p) = \pi_z(e(p))$ is a local homeomorphism from M to $\mathbb{R}^2$. We are interested in the conditions under which these transformations can be reversed: if we are given an immersed surface, when is it the projection of a generalized terrain? If we are given a curve in the plane, when is it the boundary of an immersed surface, or of the projection of a generalized terrain?

Figure 5.2 depicts the set of objects whose relationships we consider; a transformation from one type of object to another that loses information about the object is depicted as a downward arc. The algorithmic problems we consider, therefore, correspond to the upward arcs in this figure; for instance, the arc from curves to immersed disks, labeled "P", represents the result of Shor and Van Wyk that one can determine in polynomial time whether a curve is the boundary of an immersed disk; they call a curve that has this property *self-overlapping*. The remaining labels on the arcs of the figure represent our new results.

The cased curves mentioned on the right side of Figure 5.2 require further explanation. If one computes the projected boundary of a generalized terrain, one can obtain not just a set of curves in the plane, but also a "casing" describing the above-below relationship of the two components of boundary curve at each crossing point. Casings are generally depicted graphically by interrupting the lower curve segment at a crossing, and allowing the upper curve segment to pass through the crossing uninterrupted, as we have done with the arcs and crossings of the figure. Casings of this type are standard in the two-dimensional description of mathematical knots. In Chapter 4 we study the casing method from the graph drawing point of view. By throwing away the casing information, we obtain an uncased curve in the plane, but an uncased curve with n crossings has 2^n different casings. As we show, this casing information is crucial: with it we can, in linear time, reconstruct a generalized terrain (if one exists) that has the given cased curve as its projected boundary.
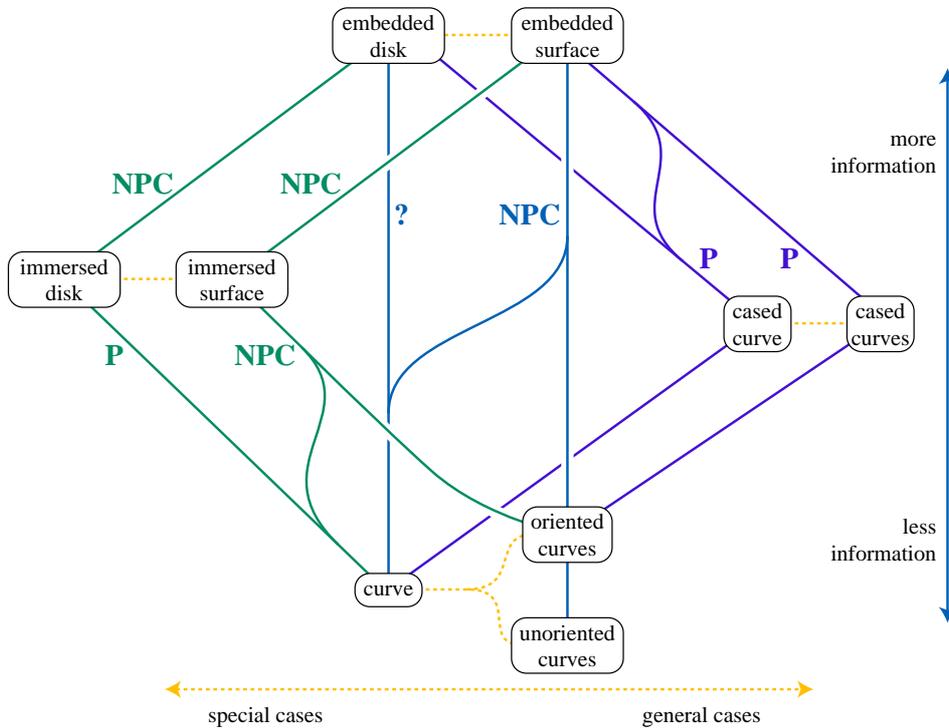
Figure 5.2 The roadmap of the results on embedding curves as boundaries of surfaces.

In the absence of casing, as we show, it is NP-complete to determine whether a curve is the projected boundary of a generalized terrain, or the boundary of an immersed surface. Additionally, we show using a different reduction that it is NP-complete to determine whether an immersed disk is the projection of a generalized terrain. The question mark in Figure 5.2 indicates that the problem of deciding whether a curve is a boundary of a disk embedded in space remains open.

Two spatial embeddings of a curve are called *combinatorially equivalent* if they provide it with the same casing. Thus, every curve with n crossings has at most 2^n combinatorially different spatial embeddings. However, we show that an immersed surface has at most $2^{n/2}$ combinatorially different spatial embeddings. As we show, the ideas behind this combinatorial result lead to a fixed-parameter-tractable algorithm for testing whether an immersed surface can be lifted to an embedded surface.

Let $i : M \to \mathbb{R}^2$ be a surface immersion in the plane. For every point $p \in i(M)$ the *thickness* of $i(M)$ at p is the number of points in the set $i^{-1}(p)$. The boundary of an immersed surface splits $\mathbb{R}^2$ into faces, where the thickness at all points belonging to the same face is the same. In fact, the thickness at p can be obtained from the boundary curve of the

surface and is equal to the *winding number* of the curve around p, defined as the number of times the curve goes around p in clockwise direction; you can see, for example, [13] for a more detailed explanation of this concept.

A *lifting* of an immersion i is an embedding $e : M \to \mathbb{R}^3$ that projects to i: that is, for all $p \in M$, $i(p) = \pi_z(e(M))$, where $\pi_z((x, y, z)) = (x, y)$. Necessarily, e must describe a generalized terrain, for otherwise its projection would not be a local homeomorphism.

A *hole* in a surface M with an immersion i is a component C of the boundary of M such that $i(C)$ is a simple curve and such that i maps a neighborhood of C to the outside of $i(C)$.

In dealing with surfaces that have multiple boundary components, it is important to have the concept of an *orientation* of a curve, an assignment of a consistent cyclic ordering to the points of the curve. We orient the boundary components of a surface M so that (as viewed according to the orientation) the surface itself is to the right of the boundary curve. An immersion or embedding of a surface, having a given set of curves in the plane as boundary, is consistent with an orientation of those curves if this rightwards orientation intrinsic to the surface matches the orientation in the embedding. Thus, a simple closed curve bounding a disk in the plane is oriented clockwise, while the boundary of a hole is oriented counterclockwise.

Organization.   In the next section we present an algorithm that determines whether a cased curve represents the boundary of a generalized terrain. The same algorithm finds a terrain having the curve as boundary if such a terrain exists. In Section 5.3 we show that finding an immersed surface in the plane or embedded surface in space of which a given uncased curve is a boundary is NP-hard. In Section 5.3.3 we show that determining whether a surface immersed in the plane is a projection of a surface embedded in space is NP-hard. Section 5.4 contains a proof that the problem of lifting an immersed surface into space is an NP-complete. It also contains a fixed parameter tractable algorithm for finding such a lifting for an immersed surface with a single component.

5.2   Lifting a cased curve

In this section we present an algorithm that constructs an embedding of a single component curve as a boundary of a generalized terrain if such a terrain exists.

Consider the set of regions the curve splits the plane into. We call them *faces* of the curve. The unbounded face of the curve is called its *exterior*. A *strand s* of an oriented curve is a closed oriented segment $[i, j]$ of the curve such that (a) s starts at a crossing i (i is called the *start* of the strand); (b) s ends at the next crossing j (called the *end* of the strand); (c) if we follow the curve from i to j in the direction of the orientation s does not contain any other crossings of the curve. Thus two strands of a curve intersect only at their start and end points.

Let i be a crossing of a cased curve C and let $s = [i, j]$ and $s' = [i, k]$ be the two strands that start at i. We say that a strand $s = [i, j]$ *overcrosses* and s' *undercrosses* if at the crossing i the strand s is above s'.

Theorem 5.1 *Given a single component curve with a given casing, we can determine whether the cased curve represents the boundary of a generalized terrain, and find a terrain having it as boundary, in time linear in the number of crossings of the curve.*

Proof. We represent the curve using a modified form of Dowker notation [24] as follows:

1. Label the n crossings of the curve by the numbers from 1 to n, arbitrarily.

2. Choose a starting point on a part of the curve that lies on the exterior face of the curve, and orient the curve consistently so that at this starting point the exterior face lies to the left of the oriented curve. For example, in Figure 5.3 the starting point is chosen on strand $[-1, -1]$.

3. At a crossing i, draw two arrows, one on each of the two strands of the curve, outward from the crossing in the direction given by the orientation. Assign the arrow that is clockwise of the other arrow the number $+i$ and the other the number $-i$. Thus each strand $[i, j]$ of the curve is marked by either $+i$ or $-i$, where i is the index of the start of the strand its first crossing along the curve.

4. List the marks constructed in this way in a list C, in the order that they would be found by traversing the curve as oriented from the given starting point. For example, the curve in Figure 5.3 is represented by $C = (-1, +1, -4, +4, -3, +3, -2, +2)$.

Additionally, we specify the casing as an n-bit binary number. It has a nonzero bit in position $i - 1$ if the strand labeled $+i$ crosses above the strand labeled $-i$, and a zero bit otherwise.

We use the sequence of above-below relationships to compute an array MaxHeight and an array Height. MaxHeight$[i]$ indicates the winding number of the strand i of the curve (that is the winding number of the face that lies to the left of the corresponding curve), and Height$[i]$ indicates the number of layers of surface that are supposed to lie below the curve as it heads outwards from the crossing labeled $|i|$. Note that here, i may be either positive or negative, as it denotes the label of the strand.

We first calculate the array MaxHeight by traversing C. At the start of the labeling sequence C the winding number is 0: since we start with the strand that is adjacent to the outer face, the winding number is the winding number of the outer face, which is zero. Every time we encounter a strand that is marked positive, the winding number becomes the winding number of the previously encountered strand plus one. Every time we encounter a negatively marked strand, we set its winding number to the winding number of the previously encountered strand minus one. If we find a strand whose winding number is negative, we terminate, since in that case the curve cannot be embedded as a boundary of
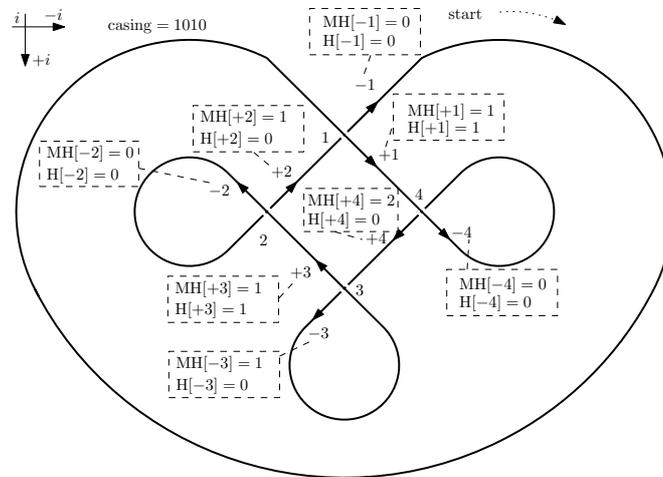
Figure 5.3 A curve $\mathcal{C}$ demonstrating the Dowker notation labeling and the values of the arrays Height and MaxHeight

an surface in space. Furthermore, we check whether the winding numbers of the curve are consistent—that is we check whether the winding number of the last strand in the curve $\mathcal{C}$ is one. We traverse $\mathcal{C}$ once again to calculate the array Height. At the start of the labeling sequence, the height is zero: as it is on the outside face, the surface must be only one layer thick at that point. Subsequent heights can be computed in constant time per value, by traversing the curve: at an undercrossing, the height remains the same as at the previous arrow in the traversal order, while at an overcrossing, the height differs either by $+1$ or -1. For an overcrossing with label $+|i|$, the new height is one larger than the previous height. For example, in Figure 5.3 at crossing 3 we have $\text{Height}[+3] = \text{Height}[-3] + 1$ because positively marked strand $+3$ overcrosses at that crossing. For an overcrossing with label $-|i|$, the new height is one smaller than the previous height. For example, in Figure 5.3 at crossing 2 we have $\text{Height}[-2] = \text{Height}[+3] - 1$ because negatively marked strand -2 overcrosses at that crossing.

From these heights of the boundary curve, we can define a full surface, by stacking sheets of surface above each face of the drawing, the number of sheets equalling the winding number of the curve around the face. The height of the boundary curve tells us which sheets continue from one face to an adjacent face, and which sheet has its boundary there. This describes a valid surface if the following conditions are met:

1. The height of each strand of the curve lies between zero and the winding number of the face adjacent to the strand in the exterior direction.

2. The casing determined by the heights matches the input casing.

Both conditions may easily be checked in linear time.   □

For a curve with multiple components we have a similar result but the complexity of the algorithm is higher.

Theorem 5.2 *Given a cased oriented curve, we can determine whether the cased curve represents the boundary of a generalized terrain and find a terrain having it as a boundary, in $O(min(nk, n + k^3))$ time, where n is the number of self-crossings of the curve and k is the number of its components.*

Proof. We first calculate the winding numbers of the faces our curve C splits the plane into. For that we construct a *dual graph* $D(C)$ of the curve defined as follows. $D(C)$ has a vertex for each face. Two vertices v_1 and v_2 of $D(C)$ are connected by a directed edge $(v_1 \rightarrow v_2)$ if the corresponding faces f_1 and f_2 are separated by a directed strand of C in such a way that f_1 lies to the left as we move along the strand in the given direction. Let v have a winding number w. Then every neighbor u of v has a winding number $w - 1$ if the corresponding edge is oriented from u to v or has the winding number $w + 1$ otherwise. (If we encounter a vertex with a negative winding number, the curve cannot be embedded as a generalized terrain boundary.) The winding number at the vertex representing the outer face is zero and $D(C)$ is connected, so we can find winding numbers for all vertices of $D(C)$ in time linear in n. Next, as in the single-curve algorithm, we pick a starting point for every curve component c_i with as low thickness (the winding number of a face adjacent to the left of c_i) as possible; let b_i denote the height of the curve c_i at that starting point. By tracing around the curve, as before, we can determine the height of the curve at each of its other strands s, as an integer offset γ_s from b_i. To ensure correctness of the casing, we have four types of constraints on these heights:

- (1) $b_i + \gamma_s$ values are non-negative for all strands s of c_i, for all $0 \le i \le k$. This will follow automatically from our choice of start point as long as b_i itself is non-negative.

- (2) Each $b_i + \gamma_s \le w$, where w is the winding number adjacent to s (on its left). We put all constraints of this type for a single component together to find the smallest integer B_i such that b_i must be at most B_i in order to satisfy the given constraints.

- (3) Each crossing between two strands s_1 and s_2 of the same component c_i is cased correctly. This can be tested by comparing their offsets γ_{s_1} and γ_{s_2}, and does not depend on the choice of b_i.

- (4) Each crossing between two different components c_i and c_j is cased correctly. This translates to the inequality $b_i \ge b_j - \delta_{ij}$, for some value δ_{ij} that can be calculated from the two offsets γ_{s^i} and γ_{s^j} at the two crossing strands $s^i \in c_i$ and $s^j \in c_j$.

To handle constraints of types (1), (2), and (3), make a graph G, with one vertex v_i per component c_i, and a special starting vertex s. Draw an edge with length zero from s to

each v_i, and an edge of length δ_{ij} from v_j to v_i. If there exists a negative cycle in this graph, it corresponds to a set of constraints of type (4) that cannot be simultaneously satisfied, and no embedding exists. Otherwise, let b_i be $-d(s, v_i)$ where d is the distance computed by a single source shortest path algorithm (note the minus sign). The edge from s to v_i forces $d(s, v_i)$ to be non-positive, so $b_i \geq 0$. The edge from v_j to v_i forces $d(s, v_i) \leq d(s, v_j) + \delta_{ij}$, the negation of a constraint of type (4), so all such constraints are satisfied. We can test whether this assignment of b_i values satisfies the constraints of type (2) by testing each such inequality; if one of the constraints of this type is violated then it together with the constraints on the shortest path to v_i cannot be simultaneously satisfied and no embedding exists. Therefore, the casing corresponds to an embedding if and only if setting $b_i = -d(s, v_i)$ results in a height assignment that satisfies all constraints. If there are k curve components, the graph has $O(min(n, k^2))$ edges, the negative cycle detection and single source shortest path in graph with negative edge weights can be done in $O(min(nk, k^3))$ time (e.g. by Bellman-Ford [3, 47]) so the total time would be $O(min(nk, n + k^3))$. □

This result immediately leads to an $O(n2^n)$-time algorithm, which we have implemented, for testing whether an uncased curve is the boundary of a generalized terrain: simply apply this linear time test to all possible casings of the curve. However, we believe that dynamic programming based on a separator decomposition of the curve arrangement will lead to improved running times. In such a technique, similarly to dynamic programs for other planar graph algorithms described by Smith and Wormald [61], one would partition the planar graph representing the input curve arrangement along a separator, and maintain a system of dynamic program states describing the heights of points on the input curve at each position at which the separator intersects the input curve. As the heights are at most n, and each dynamic programming state would need to combine the heights of $O(\sqrt{n})$ points, the time for such an approach would be exponential in $O(\sqrt{n}\log n)$, improving the simple algorithm described above which is exponential in n. However, such an algorithm would be significantly more complicated than the one we implemented.

5.3 Hardness of immersion and embedding for uncased curves

As we now show, the absence of a casing makes it much more difficult to determine whether a given curve or set of curves is the boundary of an immersion or of a generalized terrain. The overall reduction is depicted in Figure 5.4. We start with a simplified version of the proof that applies to multiple oriented curves.

5.3.1 Oriented curves

We show that finding a surface immersion for an oriented curve is NP-complete using a reduction from PLANAR 3-COLORABILITY. A PLANAR 3-COLORABILITY instance tests
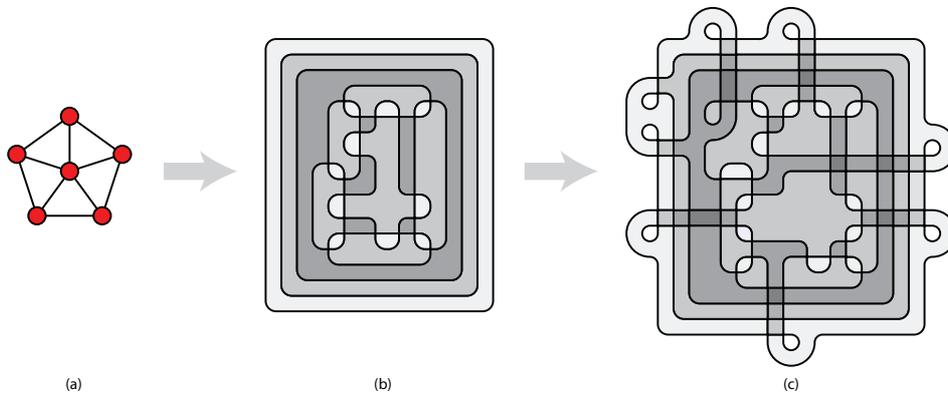
Figure 5.4 Reduction from PLANAR 3-COLORABILITY to immersibility or embeddability of a curve.

whether a planar graph has a vertex-coloring with three colors, such that adjacent vertices have different colors.

We transform a planar graph $G = (V, E)$ into a family C_G of $|V| + 3$ simple closed curves. $|V|$ of these curves, which we call vertex-curves, are oriented counterclockwise and represent the vertices of G. In any immersion or embedding having C_G as boundary, vertex-curves must be hole boundaries due to their orientation. We place these curves in such a way that every pair of vertex-curves is either disjoint or has two points in common, and such that a pair of vertex-curves intersects if and only if the corresponding vertices of the graph are adjacent. The remaining three curves in C_G, denoted c_1, c_2 and c_3, are oriented clockwise and surround all the vertex-curves. These three curves are disk-boundaries. See Figure 5.4 (a), (b) for an example.

Lemma 5.3 *If G has a three-coloring then there exists an immersed surface in the plane of which C_G is a boundary, having the topology of three disks with holes.*

Proof. Form a disk in the plane for each of c_1, c_2, and c_3. For each vertex-curve c_v, corresponding to a vertex v with color i, form a hole with boundary c_v in the disk bounded by c_i. Due to the coloring of G, no two holes in the same disk overlap, so this forms a valid immersion of three punctured disks with boundaries c_1, c_2 and c_3, together with the holes formed as above. ☐

Corollary 5.4 *If G has a three-coloring then C_G can be embedded in space as the boundary of a generalized terrain.*

Proof. Lift the three punctured disks of Lemma 5.3 to distinct heights in $\mathbb{R}^3$. ☐

Next we prove that given a collection of surfaces S immersed in the plane so that the boundaries of S match the reduction C_G of a graph G we can define a 3-coloring of G from S.

To do so, define a relation $\sim$ between hole-boundaries of S: $b_1 \sim b_2$ if and only if there exists a curve in S that starts on b_1, ends on b_2, and does not pass through any points where two holes overlap.

Lemma 5.5 $\sim$ *is an equivalence relation.*

Proof. Reflexivity and symmetry are clear. To prove transitivity, suppose $b_1 \sim b_2$ and $b_2 \sim b_3$, and find a curve from b_1 to b_3 by concatenating the curves $b_1 \sim b_2$, $b_2 \sim b_3$, and a curve around the boundary of b_2. $\square$

Lemma 5.6 $\sim$ *has at most three equivalence classes.*

Proof. We pick a point p of the plane that is not in any of the holes: the immersed surface has three points p_1, p_2, p_3 that map to p. Then every point of the surface can be connected by a path to one of the three points p_i. The path is constructed by first following a straight line segment to p and then detouring around any hole crossed by the line segment. Any two hole boundaries that connect to the same p_i must be equivalent to each other by concatenation of curves. $\square$

Lemma 5.7 *If hole-boundaries b_i and b_j cross each other, then b_i is not related by $\sim$ to b_j.*

Proof. Suppose for a contradiction that $b_i \sim b_j$, let c be a curve connecting b_i and b_j, and let s be a short line segment connecting b_i to b_j near their crossing. We can assume that c and s have the same endpoints, so together they define a (possibly self-intersecting) polygon, containing some other set of boundary holes. We choose c in such a way that this polygon contains as few boundary holes as possible, and then (among curves passing around the other holes in the same pattern) so that it is as short as possible. If c is not equal to s, then it must be stretched tight against some other hole, and we can reduce the number of holes in the polygon by replacing c by a curve that goes the other way around the same hole. Thus c must be equal to s. But then near the crossing b_i and b_j would be on the same layer of the surface, which is not possible. $\square$

Thus $\sim$ corresponds to a partition of V into three subsets, with no adjacent pair of vertices in the same subset; that is, a 3-coloring. G may be transformed into C_G in polynomial time, providing a polynomial-time many-one reduction from the known NP-complete problem of PLANAR 3-COLORABILITY to immersing or embedding a set of oriented curves. These immersion and embedding problems may be solved in NP, in the case of embedding by considering heights of curves as used in the proof of Theorem 5.1, or in the case of immersions by a system of disks and gluings as used by Shor and Van Wyk [60]. We have proven

Theorem 5.8 *The problem of determining whether a set of oriented curves can be seen as a boundary of an immersed surface is NP-complete.*

Corollary 5.9 *The problem of determining whether a set of oriented curves can be seen as a boundary of an embedded surface is NP-complete.*

Proof. Consider a planar graph G and the set of curves C_G described above. Assume we have a generalized terrain M that has C_G as a boundary. M projects to an immersed surface with C_G as a boundary which, as we have shown earlier, defines a three coloring of G. In the other direction, by Corollary 5.4, given a 3-coloring of G, we can embed C_G as a generalized terrain boundary. □

5.3.2   Non-oriented curves

To prove a similar hardness result for unoriented curves, we simply replace each vertex-curve in the reduction by a curve that can only be embedded with one orientation, namely a curve with two self-intersections as depicted in Figure 5.5. If the self-crossing parts of these curves do not cross the other curves, they can only be oriented in such a way that the two outer lobes c_1 and c_2 of the curve act like hole boundaries in whatever surface they bound, i.e. for any immersed surface i the neighborhoods of c_1 and c_2 are mapped outside of $i(c_1)$ and $i(c_2)$ correspondingly. Assume the graph G is not bipartite (if it is PLANAR 3-COLORABILITY has a polynomial solution). If the obtained set of curves can be embedded in space then the curves c_1, c_2 and c_3 are oriented clockwise. Thus there exists a unique orientation of the set of curves such that the set with this orientation can represent a boundary of a surface. Hence,

Corollary 5.10 *It is NP-complete to determine whether a curve can be seen as a boundary of an immersed surface in the plane or of an embedded surface in space.*

5.3.3   Finding a surface embedding for a single component curve

We finish this section by showing that it is NP-complete to determine the existence of an immersed or embedded surface for a curve even when it consist of a single component.
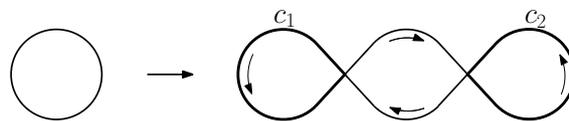
Figure 5.5 A curve that can only be embedded with one orientation.

As before, we reduce the problem from PLANAR 3-COLORABILITY. Our reduction is as before, forming a system C_G of oriented curves that can be the boundary of an immersion or embedding if and only if the given graph G is 3-colorable; by adding an additional step to the reduction we transform C_G into a single curve without changing the existence of an immersion or embedding. The following lemmas make this extra step possible.

Lemma 5.11 *Let C be a curve or set of oriented curves, and let C' be a curve or set of curves formed by breaking C in two points along the boundary of a region of thickness one, and connecting these two breaks by a pair of parallel curves. Then the embedded or immersed surfaces for C are in 1-1 correspondence with those for C'.*

Proof. In terms of the embedded or immersed surface, going from C to C' corresponds to cutting the surface by removing a thin strip between the parallel curves, and going from C' to C corresponds to gluing the cut back together. Cutting and gluing in this way are inverse operations, so both are one-to-one.   □

Lemma 5.12 *Let C be a curve or set of oriented curves, and let R be a region bounded by two segments of curves and two crossing points, such that at least one of the two neighboring regions of R has smaller thickness than R, and such that R contains no crossing of C. Let C' be the curve or set of oriented curves formed by uncrossing the two crossings bounding R. Then C is the boundary of an embedded or immersed surface if and only if C' is.*

Proof. In terms of the embedded or immersed surface, the two curves bounding R must be boundaries of two different layers of the surface, so one can go from C to C' by pulling these layers apart without changing the existence of an immersion or embedding (Figure 5.6). In the other direction, we can push the two layers together without any interaction between them; note however that for embeddings these two operations may not be one-to-one as we may have a choice whether to put one layer above or below the other. □
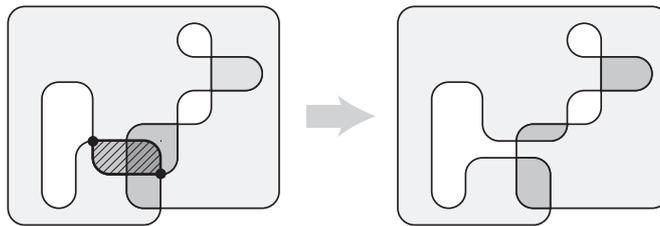
Figure 5.6 Uncrossing the dashed region bounded by two thick segments of curves and two indicated crossing points.

Given an instance G of PLANAR 3-COLORABILITY we construct the curve family C_G as in Section 5.3.1. We then transform C_G into a single curve by attaching the three outer curves together, and the holes to the outer curve, via thin strips that pass across the curve without covering any crossings, go outside the outer boundary, and then connect back to it, as shown in Figure 5.4(c). Each strip connecting two boundary curves can be removed by cutting it and performing a sequence of pulling-apart operations, so by Lemmas 5.11 and 5.12 the single curve is the boundary of an immersed or embedded surface if and only if C_G was. Hence

Theorem 5.13 *The problem of determining the existence of an immersed or embedded surface that has a single component curve as a boundary is NP-complete.*

5.4 Finding an embedding from a surface immersion

As we now show, it is NP-complete to lift an immersion to an embedding. Our reduction is via a graph-theoretic problem, ACYCLIC PARTITION, which we also show to be NP-complete. Define ACYCLIC PARTITION to be the decision problem that takes a directed graph G as input and outputs yes if and only if the vertices of G can be partitioned into two acyclic subsets. We first reduce ACYCLIC PARTITION to our problem by constructing an immersed disk that can be lifted into space if and only the input graph is a yes-instance of ACYCLIC PARTITION. Then we show that ACYCLIC PARTITION is NP-complete.

Lemma 5.14 ACYCLIC PARTITION *can be reduced in polynomial time to disk immersion lifting.*

Proof. Given a directed graph G, we create an immersed disk, in the form of a single central area (a large rectangle) connected to a rectangle for each vertex of G that covers approximately the same region of the plane as the central area (perturbed slightly so the boundaries do not overlap). For each v, the rectangle for v is connected to the central rectangle by a semicircular bridge that extends out from the central rectangle and back in to the rectangle for v, as shown in Figure 5.7; no two of these bridges overlap. Additionally, whenever G has an edge $v \to w$, we extend a rectangular tab out from the rectangle for v so that it covers the bridge for w. See Figure 5.7 showing the complete reduction for a very simple graph.

How can the resulting immersed disk be embedded? Each vertex's rectangle must go either above or below the central rectangle, clearly, and as the rectangles all have a common intersection they can be totally ordered by height in any embedding. If v and w are both above the central rectangle, and there is an edge $v \to w$, then the corresponding tab forces v to be above w. Thus, the total order of the rectangles above the central rectangle is consistent with the edges. This is only possible if the edges connecting pairs of points corresponding to the rectangles above the central rectangle form a directed acyclic graph. Similarly the rectangles below the central rectangle can be embedded in such a way that
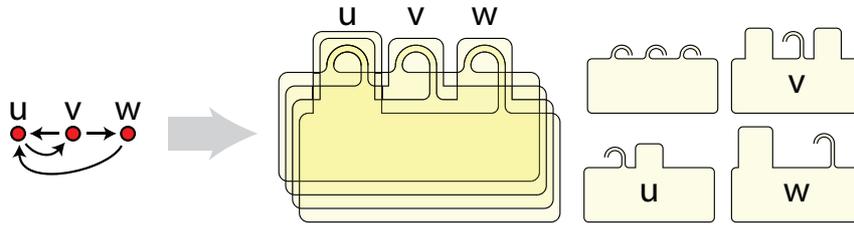
Figure 5.7 Reduction from ACYCLIC PARTITION to liftability of an immersed surface.

their tabs do not block their bridges only if the corresponding vertices form a directed acyclic graph.

In the example in Figure 5.7, for instance, we can embed the rectangles in top-to-bottom order *w*, *u*, central, *v*. The two rectangles above the central rectangle, *w* and *u*, define an acyclic graph as does the rectangle *v* by itself below the rectangle. However it would not work to try to put the rectangles for *u* and *v* on the same side of the central rectangle as each other, as the two of them form a cycle in the graph. □

Next, as promised, we prove that ACYCLIC PARTITION is NP-complete.

Lemma 5.15 ACYCLIC PARTITION *is NP-complete.*

Proof. NOT-ALL-EQUAL-3SAT is a known-NP-complete variation of 3-SAT. A NOT-ALL-EQUAL-3SAT instance consists of a set of clauses, each having three terms (variables or negations of variables). It is satisfied by a truth assignment such that in every clause not all three terms have the same values.

We transform NOT-ALL-EQUAL-3SAT into ACYCLIC PARTITION as follows. Create a pair of graph vertices for each variable and its negation, with edges forming a 2-cycle (Figure 5.8, left); the two vertices must be in different parts of any acyclic bipartition. Create another triple of vertices for each clause, with edges forming a 3-cycle; not all three may be on the same side of any acyclic bipartition. Finally, add 2-cycles connecting the term vertices with corresponding clause vertices (Figure 5.8, right). The resulting graph has an acyclic bipartition if and only if the input NOT-ALL-EQUAL-3SAT instance is satisfiable. □

Combining these two reductions gives us our result:

Theorem 5.16 *It is NP-complete to determine whether an immersed surface can lift to a generalized terrain.*
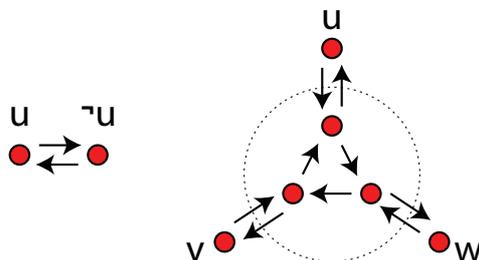
Figure 5.8 Gadgets for the reduction from NOT-ALL-EQUAL-3SAT to ACYCLIC PAR-
TITION.

5.5 The number of embeddings of an immersion

Figure 5.9 shows an immersed surface with $2^{n/2}$ combinatorially different spatial embed-
dings. As we now show, this is the most possible.

Theorem 5.17 *An immersed surface M has at most $2^{n/2}$ combinatorially different spatial
embeddings.*

Proof. We construct an undirected multigraph G on a set of self-intersections of the
boundary curve, as follows. From each self-intersection v, formed by the crossing of
the boundaries of two layers ℓ_i and ℓ_j, trace a curve in ℓ_j along the path formed in the
plane by the boundary of ℓ_i, until reaching another crossing point w that involves a bound-
ary of the layer in which the curve is being traced; add an edge in G connecting v and w.
Figure 5.10 depicts this construction for a curve that is the boundary of a unique immersed
surface.

The obtained graph G is 2-regular: there is one edge for each layer involved in each
crossing. If M is obtained from a generalized terrain, both crossings connected by any
edge e of G must be cased the same way: for, the path traced out by e has at all points of
the path one surface consistently above or below the boundary curve of the other, so there
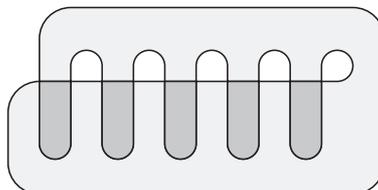
Figure 5.9 An immersed surface with $2^{n/2}$ combinatorially different spatial embed-
dings.
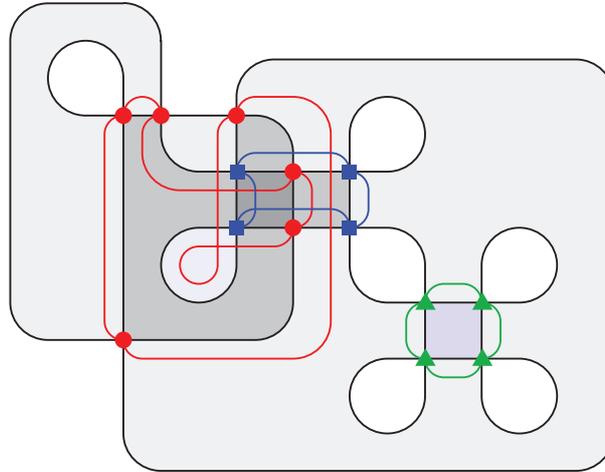
Figure   5.10   The graph G formed from the crossings of an immersed surface. The curve shown in black is the boundary of a unique immersed surface; the corresponding graph consists of two 4-cycles and one 6-cycle.

is no way for the two surfaces to swap heights. Therefore, for a casing coming from an embedding of M, the edges around any cycle of G must alternate between upper and lower, so each cycle must have even length at least equal to two. Choosing one of two casings for each of at most $n/2$ cycles in G, the total number of valid casings is at most $2^{n/2}$.   □

One consequence of this result is a simple $O(n2^{n/2})$-time algorithm for finding a lifting of an immersed surface with a single boundary component, by testing all casings consistent with the graph G. However, in many cases we can do better than this.

Observe that, in Figure 5.10, the cycle on the right (the one with triangular vertices) consists of vertices and boundary paths that all lie on points contained in only two levels of the immersed surface. We call such a cycle an *irrelevant cycle*, because the existence of a three-dimensional embedding does not depend on how it is cased: if a three-dimensional embedding with one of the two possible casings exists, then changing the casing of that cycle while keeping the casing of the rest of the drawing fixed will result in another valid embedding. The other two cycles in the figure are *relevant cycles*, because they are not irrelevant: they both include vertices that are contained within three layers of the immersion. It is possible for a cycle to have all its vertices contained in only two layers, but to have points on the curves forming its edges that are contained in three or more layers; in this case, also, we call it a relevant cycle. Then clearly, we can find a lifting of an immersed surface in time $O(n2^k)$, where k is the number of relevant cycles: try all casings of the relevant cycles, and pick a single fixed casing for each irrelevant one. Thus, the problem of lifting an immersion to an embedding is fixed-parameter tractable, with k as the fixed parameter.

Alternatively, we may use as a parameter the number c of crossings of the input curve that lie within three or more layers of the immersion. It is not difficult to show that $k = O(c)$; for, each relevant cycle either contains such a crossing or contains a two-layer crossing that is adjacent to a three-layer crossing in the arrangement formed by the boundary curves.

It would be of interest to determine whether there is a fixed parameter tractable algorithm for the problem of finding a three-dimensional embedding for a given plane curve, without being given the corresponding immersion. In this case the multigraph G cannot be defined, since it depends on having a fixed immersion, but we may still use c as the parameter for testing fixed parameter tractability.

Chapter 6

Concluding remarks

In this chapter we summarize the results of this thesis. We also discuss some interesting open problems and directions for further research for each of the questions we have considered.

Rectilinear cartograms.   In Chapters 2 and 3 we addressed the problem of constructing a rectilinear cartogram of constant complexity. We looked at the problem from a graph-theoretic point of view and showed that any plane triangulated graph admits a rectilinear cartogram of complexity at most 40. Furthermore, we showed that if a graph admits a rectangular dual it has a cartogram of complexity 20. We also have shown that when a graph lies in a special class it admits a smaller complexity cartogram. That is, if a graph admits a sliceable or a pseudo-sliceable dual it has a cartogram of complexity 12.

However, several problems in the area remain open. First of all, we still have neither a characterization of the class of graphs that admit a sliceable dual nor a polynomial algorithm to test whether a given graph admits a sliceable dual. Technically, one can enumerate all rectangular duals of graphs and layouts by traversing the lattice formed by all possible regular edge labelings of a graph—see [31]. Unfortunately, this can take exponential time. Another approach would be to use dynamic programming to recursively find slices in a graph.

Another class of graphs we are interested in are graphs that admit a pseudo-sliceable dual. So far we have not been able to find a graph that does not admit a pseudo-sliceable dual. If we can prove that any graph without separating triangles indeed admits a pseudo-sliceable dual, we will be able to construct a rectilinear cartogram of complexity at most 12 for any such a graph. Furthermore, in that case we can easily show that any plane-triangulated vertex weighted graph admits a rectilinear cartogram of complexity 24: We simply use the algorithm by Liao *et al.* [12] to preprocess the graph by splitting each vertex into at most two vertices such that the final graph does not contain separating triangles. Next we
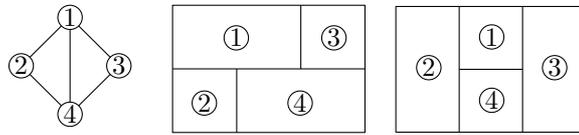
Figure 6.1 A graph and its two rectangular duals

construct a pseudo-sliceable dual for the resulting graph and a cartogram of complexity at most 12 for it. Finally, we merge the regions to obtain a cartogram of complexity at most 24 for the original graph.

Another interesting question is to decide when a graph admits a rectilinear cartogram of minimal complexity—that is, when it has a rectangular cartogram. Here we have two interesting problems. The first one is to describe a class of decidable layouts. A decidable layout is a rectangular layout for which we can decide in polynomial time whether we can set the areas of its regions to the given values to turn it into a rectangular cartogram. Van Kreveld and Speckmann [46] have demonstrated that sliceable and L-shape destructible layouts have this property. We extended the class of decidable layouts by showing that any vortex-layout is area-rigid and decidable. We believe that any rectangular layout is area-rigid. Recall that a graph can have many different rectangular duals. Figure 6.1 depicts an example of a graph and two of its rectangular duals. If both vertices 1 and 4 of the graph have weight 1, and both vertices 2 and 3 have weight 2, then the first layout cannot be turned into a cartogram, and the second one can. Our second open problem is to find a rectangular layout for a graph that can be turned into a rectangular cartogram (if such a layout exists) in polynomial time.

Cased drawings of graphs.   In Chapter 4 we considered cased drawings of graphs. The objective of the work was to try to formulate aesthetic criteria for a cased drawing as optimization problems and solve these problems. We considered two different schemes and formalized a number of criteria as optimization problems. For most of the problems we presented either a polynomial algorithm or demonstrated that the problem is NP-hard. The problem of minimizing the maximum number of switches per edge remains open for both schemes. Also, finding a casing that minimizes (as well as a casing that maximizes) the total number of switches in a graph is still open for the stacking scheme.

So far we only considered casings of graphs whose drawings are fixed. It might be interesting to look at casing problems when the requirements towards an embedding of a given graph are more relaxed. For example, one can look at graphs whose vertices have fixed positions and consider the tradeoffs one has to make while optimizing both the casing of its drawing and the complexity of the graphs edges.

Self-overlapping curves. In Chapter 5 we considered a combinatorial question in computational topology concerning three types of objects: closed curves in the plane, surfaces immersed in the plane, and surfaces embedded in space. In particular, we studied casings of closed curves in the plane to decide whether these curves can be embedded as the boundaries of certain special surfaces. We show that it is NP-complete to determine whether an immersed disk is the projection of a surface embedded in space, or whether a curve is the boundary of an immersed surface in the plane that is not constrained to be a disk. However, when a casing is supplied with a self-intersecting curve, describing which component of the curve lies above and which below at each crossing, we may determine in time linear in the number of crossings whether the cased curve forms the projected boundary of a surface in space. As a related result, we showed that an immersed surface with a single boundary curve that crosses itself n times has at most $2^{n/2}$ combinatorially distinct spatial embeddings and we discussed the existence of fixed-parameter tractable algorithms for related problems. However, we still do not know whether one can decide in polynomial time whether a curve is a projection of the boundary of a disk embedded in space.

References

[1] Arthur Appel, F. James Rohlf, and Arthur J. Stein. The haloed line effect for hidden line elimination. *Proceedings of the 6th International Conference on Computer Graphics and Interactive Techniques*, 13(2):151–157, 1979.

[2] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[3] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[4] Daniel Bennequin. Exemples d'immersions du disque dans le plan qui ne sont pas projections de plongements dans l'espace. *C. R. Acad. Sci. Paris Se'r.*, A-B 281(2-3, AII):A81–A84, 1975.

[5] Jayaram Bhasker and Sartaj Sahni. A linear algorithm to check for the existence of a rectangular dual of a planar triangulated graph. *Networks*, 7:307–317, 1987.

[6] Jayaram Bhasker and Sartaj Sahni. A linear algorithm to find a rectangular dual of a planar triangulated graph. *Algorithmica*, 3:247–278, 1988.

[7] Therese Biedl and Burkay Genc. Complexity of octagonal and rectangular cartograms. In *Proceedings of the 17th Canadian Conference on Computational Geometry*, pages 117–120, 2005.

[8] Prosenjit Bose. On embedding an outer-planar graph in a point set. *Computational Geometry: Theory and Applications*, 23(3):303–312, 2002.

[9] Sergio Cabello, Erik D. Demaine, and Günter Rote. Planar embeddings of graphs with specified edge lengths. *Journal of Graph Algorithms and Applications*, 11:259–276, 2007.

[10] Sergio Cabello, Herman Haverkort, Marc van Kreveld, and Bettina Speckmann. Algorithmic aspects of proportional symbol maps. In *Proceedings of 14th Annual European Symposium on Algorithms*, volume 4168 of *Lecture Notes in Computer Science*, pages 720–731, 2006.

[11] Ruen-Wu Chen, Yoji Kajitani, and Shu-Park Chan. A graph-theoretic via minimization algorithm for two-layer printed circuit boards. *IEEE Transactions on Circuits and Systems*, 30(5):284–299, 1983.

[12] Hsu-Chun Yen Chien-Chih Liao, Hsueh-I Lu. Compact floor-planning via orderly spanning trees. *Journal of Algorithms*, 48:441–451, 2003.

[13] William G. Chinn and Norman E. Steenrod. *First Concepts of Topology*. New Mathematical Library, MAA, 1966.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[15] Fabrizio d'Amore and Paolo Giulio Franciosa. On the optimal binary plane partition for sets of isothetic rectangles. *Information Processing Letters*, 44(5):255–259, 1992.

[16] Parthasarathi Dasgupta and Susmita Sur-Kolay. Slicible rectangular graphs and their optimal floorplans. *ACM Transactions on Design Automation of Electronic Systems*, 6(4):447–470, October 2001.

[17] Mark de Berg, Elena Mumford, and Bettina Speckmann. On rectilinear duals for vertex-weighted plane graphs. In Patrick Healy and Nikola S. Nikolov, editors, *Proceedings of the 13th International Symposium on Graph Drawing*, volume 3843 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2005.

[18] Mark de Berg, Elena Mumford, and Bettina Speckmann. Optimal bsps and rectilinear cartograms. In *GIS '06: Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pages 19–26, New York, NY, USA, 2006. ACM.

[19] Borden D. Dent. *Cartography - thematic map design*. McGraw-Hill, 5th edition, 1999.

[20] Narsingh Deo. Note on Hopcroft and Tarjan's planarity algorithm. *Journal of the ACM*, 23(1):74–75, 1976.

[21] Matthew T. Dickerson, David Eppstein, Michael T. Goodrich, and Jeremy Yu Meng. Confluent drawings: visualizing non-planar diagrams in a planar way. In *Proceedings of the 11th International Symposium on Graph Drawing*, volume 2912 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, September 2003.

[22] Reinhard Diestel. *Graph theory*. Springer-Verlag, New York, 2 edition, 2005.

[23] Daniel Dorling. *Area Cartograms: their Use and Creation*. Number 59 in Concepts and Techniques in Modern Geography. University of East Anglia, Environmental Publications, Norwich, 1996.

[24] Clifford H. Dowker and Morwen B. Thistlethwaite. Classification of knot projections. *Topology and its Applications*, 16:19–31, 1983.

[25] Peter D. Eades and Nick C. Wormald. Fixed edge-length graph drawing is np-hard. *Discrete Applied Mathematics*, 28(2):111–134, 1990.

[26] David Eppstein. Testing bipartiteness of geometric intersection graphs. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 853–861, 2004.

[27] David Eppstein, Michael T. Goodrich, and Jeremy Yu Meng. Confluent layered drawings. In János Pach, editor, *Proceedings of the 12th International Symposium on Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 184–194. Springer-Verlag, 2004.

[28] David Eppstein, Marc J. van Kreveld, Elena Mumford, and Bettina Speckmann. Edges and switches, tunnels and bridges. In *Proceedings of the 10th Workshop on Algorithms and Data Structures*, pages 77–88, 2007.

[29] 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.

[30] Istvan Fary. On straight lines representation of planar graphs. *Acta Scientiarum Mathematicarum Szeged*, 11:229–233, 1948.

[31] Éric Fusy. Transversal structures on triangulations: combinatorial study and straight-line drawing. *Discrete Mathematics*, To appear.

[32] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for general graph-matching problems. *Journal of the ACM*, 38(4):815–853, 1991.

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

[34] Branko Grünbaum. Two-coloring the faces of arrangements. *Periodica Mathematica Hungarica*, 11:181–185, 1980.

[35] Leonidas J. Guibas, J. E. Hershberger, Joseph S. B. Mitchell, and Jack S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *International Journal of Computational Geometry and Applications*, 3(4):383–415, 1993.

[36] Xin He. An efficient parallel algorithm for finding rectangular duals of plane triangular graphs. *Algorithmica*, 13(6):553–572, June 1995.

[37] Roland Heilmann, Daniel A. Keim, Christian Panse, and Mike Sips. Recmap: Rectangular map approximations. In *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS)*, pages 33–40, 2004.

[38] Petr Hliněný. Crossing number is hard for cubic graphs. *Journal of Combinatorial Theory, Series B*, 96(4):455–471, 2006.

[39] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[40] Seok-Hee Hong, Damian Merrick, and Hugo A. D. do Nascimento. The metro map layout problem. In János Pach, editor, *Proceedings of the 12th International Symposium on Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 482–491. Springer, 2004.

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

[42] Nicholas R. Chrisman James A. Dougenik and Duane R. Niemeyer. An algorithm to construct continuous area cartograms. *Professional Geographer*, 3:75–81, 1985.

[43] Goos Kant and Xin He. Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems. *Theoretical Computer Science*, 172:175–193, 1997.

[44] Michael Kaufmann and Roland Wiese. Embedding vertices at points: Few bends suffice for planar graphs. 1731:165–174, 2000.

[45] Krzysztof Koźmiński and Edwin Kinnen. Rectangular dual of planar graphs. *Networks*, 5:145–157, 1985.

[46] Marc van Kreveld and Bettina Speckmann. On rectangular cartograms. *Computational Geometry: Theory and Applications*, 37:175–187, 2007.

[47] Jr. Lester R. Ford and Delbert R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1962.

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

[49] Antony Mansfield. Determining the thickness of graphs is np-hard. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 93, pages 9–23, 1983.

[50] Morris L. Marx. Extensions of normal immersions of S^1 into R^2. *Transactions of the American Mathematical Society*, 187:309–326, 1974.

[51] 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*, volume 4372 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2006.

[52] Ketan Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice Hall, 1994.

[53] NCGIA / USGS. Cartogram Central, 2002. `http://www.ncgia.ucsb.edu/projects/Cartogram_Central/index.html`.

[54] Martin Nöllenburg and Alexander Wolff. A mixed-integer program for drawing high-quality metro maps. In Patrick Healy and Nikola S. Nikolov, editors, *Proceedings of the 13th International Symposium on Graph Drawing*, volume 3843 of *Lecture Notes in Computer Science*, pages 321–333. Springer-Verlag, 2006.

[55] János Pach, Richard Pollack, and Emo Welzl. Weaving patterns of lines and line segments in space. *Algorithmica*, 9(6):561–571, 1993.

[56] János Pach and Rephael Wenger. Embedding planar graphs at fixed vertex locations. In *Proceedings of the 6th International Symposium on Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pages 263–274, London, UK, 1998. Springer-Verlag.

[57] Md. Saidur Rahman, Kazuyuki Miura, and Takao Nishizeki. Octagonal drawings of plane graphs with prescribed face areas. In *Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 3353 of *Lecture Notes in Computer Science*, pages 320–331. Springer, 2004.

[58] Erwin Raisz. The rectangular statistical cartogram. *Geographical Review*, 24:292–296, 1934.

[59] Hanan Samet, editor. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[60] Peter W. Shor and Christopher J. Van Wyk. Detecting and decomposing self-overlapping curves. *Computational Geometry: Theory and Applications*, 2(1):31–50, 1992.

[61] Warren D. Smith and Nicholas C. Wormald. Geometric separator theorems and applications. In *FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, page 232, Washington, DC, USA, 1998. IEEE Computer Society.

[62] Sherman K. Stein. Convex maps. *Proceedings of the American Mathematical Society*, 2:464–466, 1951.

[63] Carsten Thomassen. Plane cubic graphs with prescribed face areas. *Combinatorics, Probability and Computing*, 1:371–381, 1992.

[64] Marc van Kreveld, Jürgen Nievergelt, Thomas Roos, and Peter Widmayer, editors. *Algorithmic Foundations of Geographic Information Systems*, volume 1340 of *Lecture Notes in Computer Science*. Springer, 1997.

[65] Venkat Venkateswaran. Minimizing maximum indegree. *Discrete Applied Mathematics*, 143:374–378, 2004.

[66] Klaus Wagner. Bemerkungen zum Vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 46:26–32, 1936.

[67] Jeffery Westbrook. Fast incremental planarity testing. In *Proceedings of the 19th Intinternational Colloquium on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 342–353. Springer-Verlag, Berlin, 1992.

[68] Hassler Whitney. On regular closed curves in the plane. *Compositio Mathematica*, 4:276–284, 1937.

[69] Gary K.H. Yeap and Majid Sarrafzadeh. Sliceable floorplanning by graph dualization. *SIAM Journal of Discrete Mathematics*, 8(2):258–280, 1995.

[70] Kok-Hoo Yeap and Majid Sarrafzadeh. Floor-planning by graph dualization: 2-concave rectilinear modules. *SIAM Journal on Computing*, 22, 1993.

Acknowledgements

Summary

Drawing Graphs for Cartographic Applications

Graph Drawing is a relatively young area that combines elements of graph theory, algorithms, (computational) geometry and (computational) topology. Research in this field concentrates on developing algorithms for drawing graphs while satisfying certain aesthetic criteria. These criteria are often expressed in properties like edge complexity, number of edge crossings, angular resolutions, shapes of faces or graph symmetries and in general aim at creating a drawing of a graph that conveys the information to the reader in the best possible way. Graph drawing has applications in a wide variety of areas which include cartography, VLSI design and information visualization. In this thesis we consider several graph drawing problems.

The first problem we address is rectilinear cartogram construction. A *cartogram*, also known as *value-by-area map*, is a technique used by cartographers to visualize statistical data over a set of geographical regions like countries, states or counties. The regions of a cartogram are deformed such that the area of a region corresponds to a particular geographic variable. The shapes of the regions depend on the type of cartogram. We consider rectilinear cartograms of constant complexity, that is cartograms where each region is a rectilinear polygon with a constant number of vertices. Whether a cartogram is good is determined by how closely the cartogram resembles the original map and how precisely the area of its regions describe the associated values. The *cartographic error* is defined for each region as $|A_c - A_s|\,/A_s$, where A_c is the area of the region in the cartogram and A_s is the specified area of that region, given by the geographic variable to be shown.

In this thesis we consider the construction of rectilinear cartograms that have correct adjacencies of the regions and zero cartographic error. We show that any plane triangulated graph admits a rectilinear cartogram where every region has at most 40 vertices which can be constructed in $O(n \log n)$ time. We also present experimental results that show that in practice the algorithm works significantly better than suggested by the complexity bounds. In our experiments on real-world data we were always able to construct a cartogram where the average number of vertices per region does not exceed five. Since a rectangle has four vertices, this means that most of the regions of our rectilinear car-

tograms are in fact rectangles. Moreover, the maximum number vertices of each region in these cartograms never exceeded ten.

The second problem we address in this thesis concerns cased drawings of graphs. The vertices of a drawing are commonly marked with a disk, but differentiating between vertices and edge crossings in a dense graph can still be difficult. *Edge casing* is a well-known method—used, for example, in electrical drawings, when depicting knots, and, more generally, in information visualization—to alleviate this problem and to improve the readability of a drawing. A *cased drawing* orders the edges of each crossing and interrupts the lower edge in an appropriate neighborhood of the crossing. One can also envision that every edge is encased in a strip of the background color and that the casing of the upper edge covers the lower edge at the crossing. If there are no application-specific restrictions that dictate the order of the edges at each crossing, then we can in principle choose freely how to arrange them. However, certain orders will lead to a more readable drawing than others. In this thesis we formulate aesthetic criteria for a cased drawing as optimization problems and solve these problems. For most of the problems we present either a polynomial time algorithm or demonstrate that the problem is NP-hard.

Finally we consider a combinatorial question in computational topology concerning three types of objects: closed curves in the plane, surfaces immersed in the plane, and surfaces embedded in space. In particular, we study casings of closed curves in the plane to decide whether these curves can be embedded as the boundaries of certain special surfaces. We show that it is NP-complete to determine whether an immersed disk is the projection of a surface embedded in space, or whether a curve is the boundary of an immersed surface in the plane that is not constrained to be a disk. However, when a casing is supplied with a self-intersecting curve, describing which component of the curve lies above and which below at each crossing, we can determine in time linear in the number of crossings whether the cased curve forms the projected boundary of a surface in space. As a related result, we show that an immersed surface with a single boundary curve that crosses itself n times has at most $2^{n/2}$ combinatorially distinct spatial embeddings and we discuss the existence of fixed-parameter tractable algorithms for related problems.

Curriculum Vitæ

Elena Mumford was born on the 25th of January, in Slonim, USSR. She received her Diploma in Mathematics from the Belarusian State University (Minsk, Belarus) in 1999 and her Professional Doctorate in Engineering from TU Eindhoven (Eindhoven, The Netherlands) in 2004. In September 2004 she started her PhD studies in the Algorithms group in TU Eindhoven (Eindhoven, The Netherlands) and in 2008 she completed her thesis.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in µCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition*

and construction. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies,*

and Business Applications. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures.* Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling.* Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support- .* Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical En-

gineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of

Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementa-*

tion and Composition. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calam. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21