6. Orthogonal Graph Drawing

Markus Eiglsperger, Sándor P. Fekete, and Gunnar W. Klau

6.1 Introduction

There are various criteria to judge the quality of a drawing of a graph. From a human point of view, one of the most important issues is the readability of a drawing: ideally, it should be easy to understand the structure of a graph with just a few glances, and the chance of confusion over connections between different vertices should be small. From an algorithmic point of view, it is necessary to capture this quality by means of an objective function. Various objective functions have been studied, with a great deal of effort put into their optimization by means of combinatorial algorithms.

An undesired property of a drawing that may impede its legibility is the presence of edges that are too close together. Keeping different edges apart may be particularly difficult in the vicinity of vertices, where several adjacent edges have to meet. Clearly, there is some correlation between the involved angles and the optical distinctiveness of the drawn edges. This motivates a particular objective function that is considered at the beginning of this chapter: find a drawing of a graph such that the minimum angle between adjacent edges is maximized. The first section discusses upper and lower bounds on angles in straight-line drawings.

There is a particularly nice way to guarantee maximal distinctiveness of adjacent edges in a drawing: when forcing all angles between adjacent edges to be multiples of $\frac{\pi}{2}$, edges will correspond to axis-parallel paths. The price we may have to pay for this type of clarity is to admit bends in the path representing an edge. In order to avoid confusion by too complicated paths, it is desirable to minimize the number of these bends. This setup has given rise to the area of orthogonal graph drawing – probably one of the most prolific in all of graph drawing, with scores of methods, heuristics, and sophisticated algorithms like KANDINSKY, others extending to mainly theoretical research areas like three-dimensional drawings. Over the years, orthogonal graph drawing has become far more important than the issues of angles in drawings, but both types of problems have their own motivation and have been studied independently.

It is the main objective of this chapter to combine a description of the key ideas (like the flow methods in the landmark paper of Tamassia 1987) with an overview of some of the main consequences and applications.

This chapter is organized as follows: after discussing angles in drawings in Section 6.2, Section 6.3 characterizes the correspondence between orthogonal drawings and combinatorial descriptions: how can we give a compact combinatorial encoding of the orthogonal shape of a drawing (called an "orthogonal representation" in the literature), and how can we realize a given combinatorial encoding as an orthogonal drawing?

Section 6.4 describes a number of heuristics that have been developed for finding a good orthogonal drawing without employing orthogonal representations.

When trying to find orthogonal drawings with few bends, we concentrate on the space of orthogonal representations. Optimizing over this space is the subject of Section 6.5, where we describe an efficient combinatorial algorithm for this task. Extensions to planar graphs of possibly high degree are sketched.

The final Sections 6.6 and 6.7 deal with improving orthogonal drawings. In many cases, the output of orthogonal drawing algorithms can be compacted further by assigning different – but still consistent – lengths to the edge segments. Section 6.6 gives an overview of compaction techniques, ranging from efficient heuristics to optimal techniques. The following Section 6.7 presents some efficient postprocessing methods that operate directly on the orthogonal drawings and try to improve aesthetic criteria like the number of bends or the number of crossings. We conclude with Section 6.8 and present some open problems in orthogonal graph drawing.

In many algorithms presented in this chapter, flow algorithms play the key role. Many problems can be reduced to a maximum or minimum cost flow problem. A good overview of network flow problems, modeling, and algorithms can be found in Ahuja et al. (1993) or in Bertsekas (1998).

6.2 Angles in Drawings

As described above, the following optimization problem comes up naturally when trying to create drawings with high resolution:

Problem 6.1 (Angular Resolution). Given a graph $G = (V, E)$ with n vertices and m edges, how can we draw the vertices as points in the plane, and the edges as straight lines between adjacent vertices, such that the angular resolution, i.e., the smallest angle between adjacent edges, is as large as possible?

Over the years, a number of researchers have given various kinds of answers to this question. One type of result is to establish the complexity of Angular Resolution. It was shown by Formann et al. (1990) that it is $\mathcal{NP}$-hard to check whether a planar graph with maximum degree 4 can be drawn with angular resolution at least $\frac{\pi}{2}$. Before discussing how to relax the requirements on drawings, such that a drawing with resolution $\frac{\pi}{2}$ is always possible, we describe some lower and upper bounds for straight-line drawings.

If d is the maximum degree of a vertex in G, it is clear that the angular resolution cannot exceed $\frac{2\pi}{d}$. It was shown in (Formann et al., 1990) that for planar graphs, a resolution of $\Omega(\frac{1}{d})$ can indeed be achieved. For general graphs, a resolution of $\Omega(\frac{1}{d^2})$ can be guaranteed. The key is to use a coloring of $G^2 = (V, E^2)$ of G. (Recall that two vertices in V are adjacent in G^2 if and only if they have distance at most 2 in G.) It can be shown that there is a coloring of G with $O(d)$ colors for planar graphs G and with $O(d^2)$ colors for general graphs G. Then points of a color class are drawn as a cluster of points on a unit circle, with different clusters distributed at equal distance around the circle. This guarantees that any angle between adjacent edges in G involves points from three different color classes of G^2, implying the claimed bounds.

The method by Formann et al. suffers from a very serious drawback for practical purposes. The number of crossings in the straight-line drawing may be much higher than necessary. In particular, the resulting drawing of a planar graph may not be free of edge crossings. Thus, it remained open whether the resolution of a planar drawing of a planar graph could be bounded from below in a satisfactory manner.

A first partial answer to this problem was given by Malitz and Papakostas (1992, 1994) who described a method to guarantee a lower bound of $\frac{1}{7^d}$ for planar straight-line drawings of planar graphs. Their approach relies on so-called "disc-packings" or "coin graph representations" of a planar graph $G = (V, E)$, where vertices $v \in V$ are represented by disjoint discs, and two vertices $v_1, v_2 \in V$ are adjacent if and only if the discs corresponding to v_1 and v_2 touch. The existence of this type of representation has been proved independently by a number of researchers, including Koebe (1936), Andreev (1970a,b), Colin de Verdière (1989), and Thurston (unpublished). The lower bound arises from the fact that in certain subsets of adjacent discs, the radii can vary by at most a factor of $\frac{1}{7}$. Malitz and Papakostas also conjectured that a lower bound of $\Omega(\frac{1}{d})$ for the angular resolution of planar straight-line drawings of planar graphs might be achievable. As partial evidence for this conjecture, they showed that a particular relaxation of the problem ANGULAR RESOLUTION has an optimum of $O(\frac{1}{d})$:

Any set of angles in a feasible drawing has to satisfy a set of linear equalities – see Figure 6.1. Around every vertex v, the sum of the angles $\Phi(v)$ at v must be 2π; for every interior cycle the sum of the angles $\Phi(f)$ must be $(|\Phi(f)| - 2)\pi$, and $(|\Phi(f_0)| + 2)\pi$ for the exterior cycle f_0:

$$\sum_{\phi_i \in \Phi(v)} \phi_i = 2\pi \text{ for all } v \in V. \tag{6.1}$$

$$\sum_{\phi_i \in \Phi(f)} \phi_i = \pi(|\Phi(f)| - 2) \text{ for all } f \in F \setminus \{f_0\}. \tag{6.2}$$

$$\sum_{\phi_i \in \Phi(f_0)} \phi_i = \pi(|\Phi(f_0)| + 2). \tag{6.3}$$

If we only impose these necessary conditions while maximizing the minimum angle, we get a linear program that can be solved efficiently. By using linear programming duality, it can be shown that there is always an optimum of value $\Omega(\frac{1}{d})$.

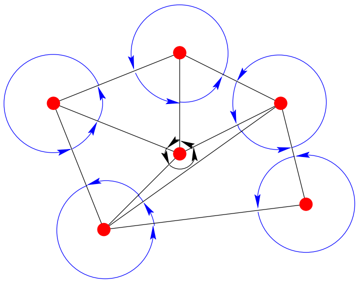


Fig. 6.1. Angles in a drawing must satisfy certain conditions.

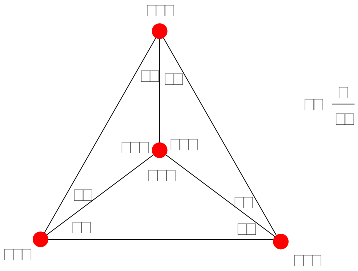


Fig. 6.2. A solution to the linear system that does not correspond to a feasible drawing.

It can be seen from Figure 6.2 that not every solution to the linear programming relaxation corresponds to a set of angles that allow a drawing. After drawing the bottom triangle with the given angles, the three edges incident to the top vertex cannot intersect in one point. It was shown by Di Battista and Vismara (1993, 1996) that for triangulated triconnected planar graphs with a designated external face f_0, the conditions on angle sums can be amended by the following requirement to get a set of necessary and sufficient conditions – see Figure 6.3:

Around each vertex v of degree $d(v)$, we have angles γ_i, $i = 1, \ldots, \deg(v)$. Each γ_i lies in the same triangle as the angles α_i and β_i, as shown in the figure. In any feasible drawing, the angles α_i, β_i satisfy

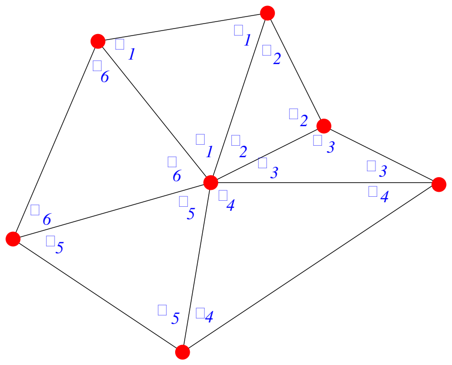


Fig. 6.3. Angles around a vertex in a drawing.

$$\prod_{i=1}^{d(v)} \frac{\sin \alpha_i}{\sin \beta_i} = 1.$$

The necessity of these conditions is a consequence of planar trigonometry; conversely, it can be shown that this additional condition suffices to get a feasible drawing for each "wheel", as shown in Figure 6.3, since any triple of edges that are incident to the same vertex will indeed meet at a single point. By induction, it follows that we get a feasible drawing for the full graph. Adding these conditions results in a nonlinear program for optimizing the angular resolution.

Garg and Tamassia (1994) managed to disprove the conjecture of Malitz and Papakostas (1994) by giving a family of graphs with an upper bound of $O(\sqrt{\log d/d^3})$. See Figure 6.4. In the same paper, they showed that good angular resolution may come at the expense of high numerical resolution, i.e., large angles require high precision in the coordinates. In other terms, they showed that there is a family of graphs, such that if all n vertices are drawn at integer coordinates, a drawing of angular resolution ρ requires area $\Omega(c^{\rho n})$. See Figure 6.5.

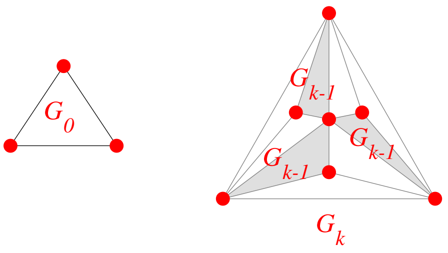


Fig. 6.4. A family of graphs with small angular resolution.

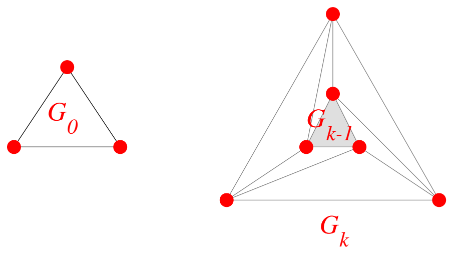


Fig. 6.5. A family of graphs with tradeoff between angular resolution and area of a drawing.

6.3 Orthogonal Drawings and Their Encoding

6.3.1 Why Orthogonal Drawings?

When trying to come up with a drawing of a graph that makes it easy to distinguish different edges, there is an alternative to using straight edges at arbitrary angles: if edges are allowed to be drawn as a path consisting of several line segments, it is possible to let all edges be represented by axis-parallel paths, called an *orthogonal drawing*. The price we may have to pay is the introduction of additional nodes where changes in direction occur in a path, i.e., a number of *bends* of the edges.

See Figure 6.6 for an example. A formal definition of an orthogonal drawing can be given as follows:

Definition 6.2. *An* orthogonal grid embedding *Γ of a graph $G = (V, E)$ is a mapping into the plane, which maps vertices $v \in V$ to integer grid points $\Gamma(v)$ and edges in $(v, w) \in E$ to non-overlapping paths in the grid such that the images of their endpoints $\Gamma(v)$ and $\Gamma(w)$ are connected. A grid embedding is* simple *if its number of bends is zero. A simple embedding induces a partition of the edge set E into a horizontal set E_h and a vertical set E_v. For simplicity, we assume that edges in E_h are directed from left to right and edges in E_v from bottom to top.*

Another issue is the numerical resolution, i.e., the difference of the involved coordinates for a drawing of given size. If we scale the final drawing such that all coordinates are integers, this translates to trying to find a drawing that uses small area.

The arrangement of edges in an orthogonal drawing is well-structured (there are only two classes of line segments, and the segments for each individual edge interchange between horizontal and vertical), so it is conceivable that drawings with good visual and structural properties are possible. Moreover, there is a close relationship to problems of VLSI layout. The components

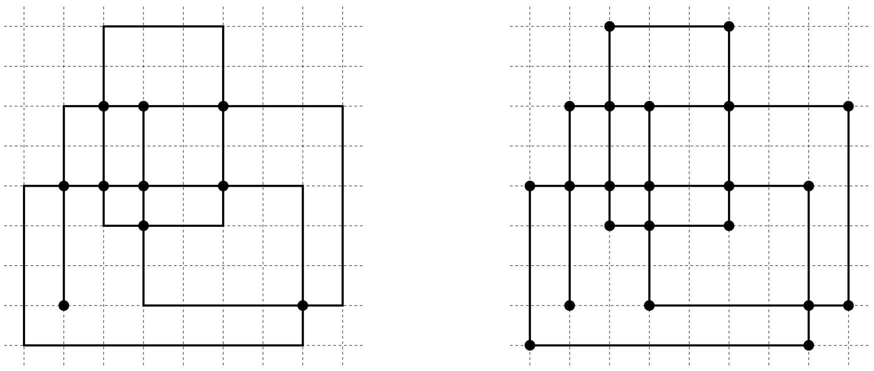


Fig. 6.6. An orthogonal grid embedding (left) and its simple counterpart (right).

in a chip layout correspond to the vertices, their connecting wires to the edges of a graph. This allows it to make use of existing methods; see Sections 6.4 and 6.6. In VLSI, however, research has concentrated on very efficient methods due to the large sizes of the instances. Often, superlinear running time is too slow for practical application, whereas in the area of graph drawing higher running time is tolerated to get better drawings[1]. However, the presence of crossing edges is highly undesirable in well-structured drawings of a graph as well as in routing wires of a chip; thus, planar orthogonal graph drawing deals almost exclusively with plane drawings of planar graphs. Since it is impossible to avoid overlap between edges if a graph has degree more than 4, we start by focusing on planar graphs with maximum degree 4 – so-called *4-planar graphs*. It will be discussed later how to deal with planar graphs of higher degree.

6.3.2 Encoding Planarity

Before dealing with heuristic and exact algorithms for orthogonal drawings and their optimization in the following sections, we now describe a way to encode a graph and a drawing of a graph, such that we can use these encodings for input and output of our algorithms. A graph $G = (V, E)$ can be simply described as a list of vertices V, and the edges E connecting them. A plane drawing of a planar graph contains additional topological information: if the edges of a graph are represented by a set of curves, the plane is subdivided into a number of open regions. These regions are called *faces*. For a given embedding, the structure of these faces is characterized by the cycles of edges surrounding them, so that it is also legitimate to speak of these cycles as faces.

[1] Another difference (at least compared to problems within the topology-shape-metrics paradigm that will be discussed in Section 6.3.4) is that the order of wires connected to a component is not necessarily fixed – this corresponds to a scenario with an arbitrary embedding.

It is not hard to see that faces and their adjacencies are determined by the circular order of edges around each vertex. Thus, we can describe a particular (topological) embedding of a planar graph by a list of its vertices V, a list of its edges E, a list of faces F, and a list $P(f)$ of edges for each face f. See Figure 6.7 for an example; the corresponding list of faces and list of edges around each face is as follows:

$$\begin{aligned}
&F = \{f_1, \ldots, f_5\}\\
&P(f_1) = (e_1, e_3, e_5, e_9, e_{10}, e_4, e_3, e_2)\\
&P(f_2) = (e_2, e_1)\\
&P(f_3) = (e_5, e_6, e_8)\\
&P(f_4) = (e_6, e_4, e_7)\\
&P(f_5) = (e_8, e_7, e_{10}, e_9).
\end{aligned}$$

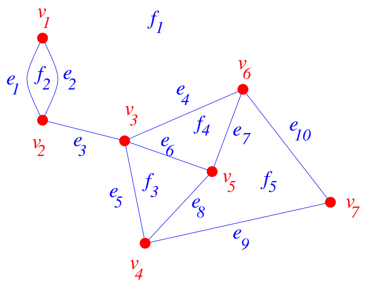


Fig. 6.7. A plane drawing of a graph G.

6.3.3 Encoding Orthogonality

When considering an orthogonal drawing of a planar graph, we need to provide even more information – in particular, we need to describe the bends along an edge, and the type of turn that an edge takes at a bend. This can be done as follows: for each edge in the edge list of a face, we describe the sequence of bends encountered while traveling along the edge by a 0-1 string. A “1” in the string indicates a left-hand turn taken at the bend, while a right-hand turn is indicated by a “0”. If an edge has no bends, this is indicated by the empty string ε. Finally, each edge is assigned an angle (as a multiple of $\frac{\pi}{2}$) that is enclosed by its last line segment and the first line segment of the next edge. See Figure 6.8 for an example with the following encoding of the edges:

$$H(f_1) = ((e_1, \varepsilon, 3), (e_5, 11, 1), (e_4, \varepsilon, 3), (e_2, 1011, 1)),$$
$$H(f_2) = ((e_1, \varepsilon, 1), (e_6, \varepsilon, 2), (e_5, 00, 1)),$$
$$H(f_3) = ((e_2, 0010, 1), (e_4, \varepsilon, 1), (e_6, \varepsilon, 1), (e_3, 0, 4), (e_3, 1, 1)).$$

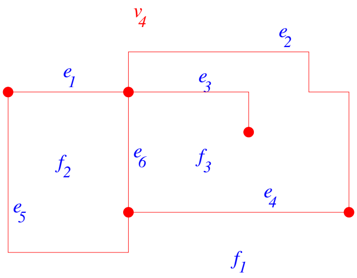


Fig. 6.8. An orthogonal drawing of a graph G.

This type of encoding of the orthogonal shape of a drawing has been called an *orthogonal representation* of the drawing; for historical reasons, we will use this term, even though it can be argued that in the true meaning of the word, the orthogonal drawing itself is a representation of the graph. (Strictly speaking, the code should be called an "encoding of the representation".)

The orthogonal representation carries only the combinatorial and some of the geometric information of the drawing; in particular, there is no information about the edge lengths. As described above, there is a well-defined orthogonal representation for each drawing of a graph, and it is not hard to see that there are a number of necessary conditions on an orthogonal representation:

1. There is a 4-planar graph corresponding to the lists for V and E.
2. Each edge is encoded twice, once for each of the two faces it bounds. Both these encodings must be consistent.
3. The sum of angles along the perimeter of a face f_i described by $H(f_i)$ must be consistent with the fact that f_i is a simple rectilinear polygon.
4. For each vertex v, the sum of angles between consecutive edges around v must sum to 4.

All four of these conditions are easy to check. As it turns out, they are also sufficient. We describe in the next section how to find an orthogonal drawing that realizes a given orthogonal representation.

6.3.4 Getting a First Drawing

We now discuss how to assign consistent integer lengths to the edge segments that are contained in an orthogonal representation. This task is also referred to as the third phase in the *topology-shape-metrics approach* as presented in (Di Battista et al., 1999). The first phase deals with fixing the topology of the eventual drawing by determining a combinatorial embedding and an outer face of the planar input graph. In the second phase the shape of the orthogonal drawing is fixed. Its output is an orthogonal representation H as described in Section 6.3.2. The best-known member of this class of algorithms is the bend-minimization algorithm by Tamassia (1987). Section 6.5 describes these flow-based algorithms. For now, we concentrate on the third task: fixing the metrics of the drawing resulting in an orthogonal grid embedding.

We present an algorithm which first finds an orthogonal grid embedding for H by describing a simplified variant of the approach presented in Tamassia (1987). This grid embedding can be further improved by applying compaction and postprocessing algorithms described in Section 6.6.

The idea of this method is to add artificial vertices and edges to H so that it is easy to find a drawing Γ' for the resulting orthogonal representation H'. Note that the insertion of artificial objects still happens on the level of the orthogonal representation and does not involve geometric operations. The removal of the artificial vertices and edges leads to an orthogonal grid embedding Γ for the original representation H. The algorithm presented in this section runs in time $O((n+b)^{7/4}\sqrt{\log(n+b)})$ and guarantees a drawing of $O((n+b)^2)$ area, where b is the number of bends.

In a first step, each bend in H is replaced by a virtual vertex, resulting in a simple orthogonal representation with $n+b$ vertices. Consider for every face f in H the circular bit string $S(f)$ that is built by traversing the edges of f in clockwise order as follows. Depending on the angle $a(e)$ (expressed in multiples of $\frac{\pi}{2}$) that an edge e forms with its succeeding edge, we add a "0" (if $a(e) = 1$), a "1" (if $a(e) = 3$) or a "11" (if $a(e) = 4$) to the bit string. Angles of 2 do not contribute to $S(f)$, since they correspond to two collinear line segments that do not form a bend. The resulting string describes the shape of face f as in Figure 6.9 (a). Now the algorithm looks for occurrences of the substring "100" in $S(f)$, which corresponds to a rectangular "ear". Whenever such a string is found, the corresponding part of the face is removed and "100" is replaced by "0" (see Figure 6.9 (b)). Clearly, this operation affects only the shape of face f, other faces remain untouched. The procedure finds all such substrings in time $O(|S(f)|)$ for each face f by a circular traversal of $S(f)$ and stacking the encountered "1"s. The bit string of a face f in the resulting orthogonal representation H' is either $S(f) =$"0000" – in this case f is an internal face – or it does not contain two consecutive zeros – then we are dealing with the external face. (See Figures 6.9 (c) and (d). The example will be continued in Section 6.6.)

An orthogonal grid embedding for H' is found by assigning the same length to opposite sides in the rectangular faces. This can be done optimally by the following algorithm; its key step is the computation of two minimum cost flows. Build two networks N_h and N_v-one for each direction, as in Fig-

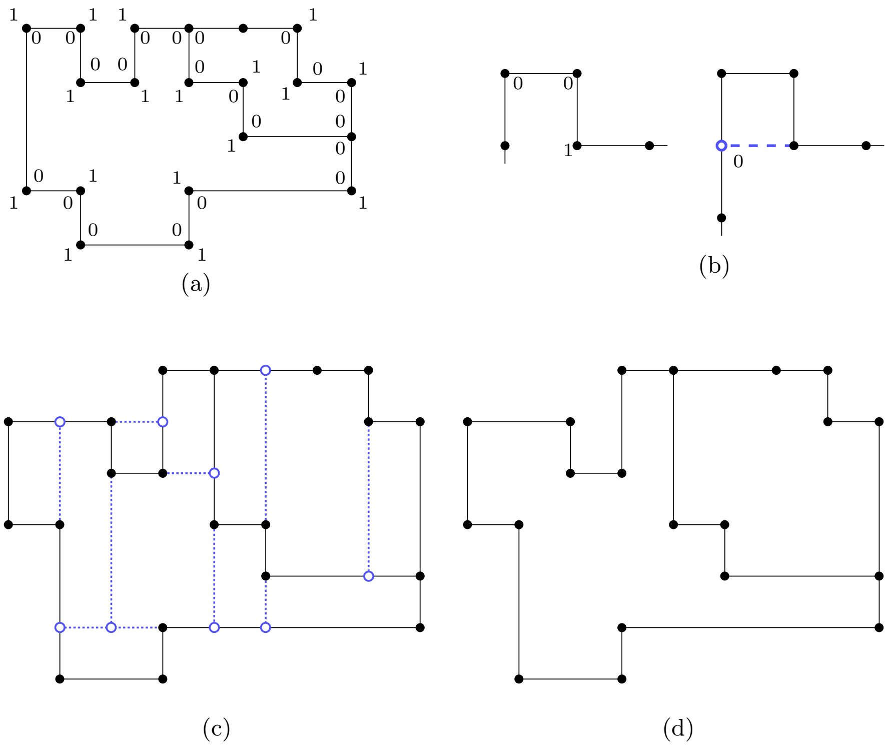


Fig. 6.9. (a) A sketch of an orthogonal representation H with bit strings $S(f)$ for each face f. (b) Substring "100" and the corresponding cut. (c) A sketch of the dissected representation H'. (d) Deleting the artificial objects in a drawing for H' yields a drawing for H.

ure 6.10. The union of N_h and N_v is the dual graph of H', the arcs in N_h are directed from bottom to top, those in N_v from left to right. Each arc a corresponds to an edge e in H', and the flow through a is interpreted as the length of e. Intuitively, this implies a lower bound of one and unit cost for the flow through a. Flow conservation ensures that opposite sides are of equal length. The cost of the flows add up to the total edge length, and – since they are minimized – lead to an optimal drawing for H'. Note however, that the artificially introduced vertices and edges have to be removed from the drawing in order to get an orthogonal grid embedding for H. As Figure 6.9(d) shows, the initial solution is not optimal for the original input.

An optimal flow can be computed in time $O(n^{7/4}\sqrt{\log n})$ with the algorithm described in Garg and Tamassia (1996b). This corresponds to a drawing for H' with minimum total edge length, width, height, and area. The networks are similar to the ones used for optimal one-dimensional compaction, which are introduced in Section 6.6.2. Furthermore, there is a linear-time method for

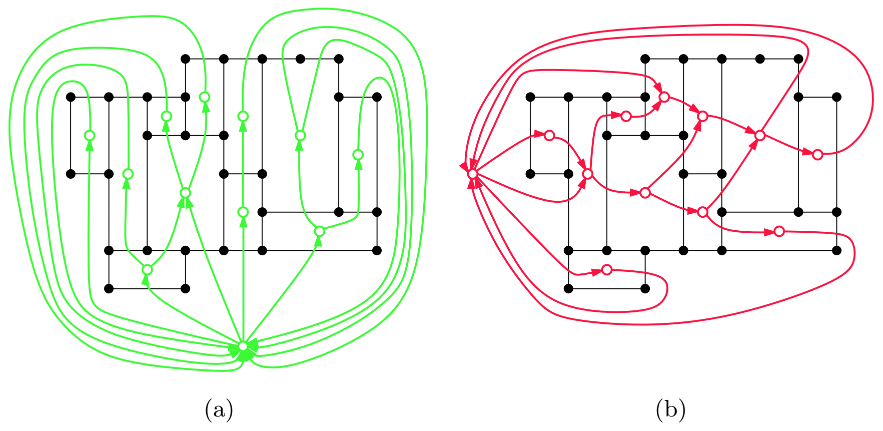


Fig. 6.10. The two networks N_h and N_v serve to construct a first drawing for an orthogonal representation.

computing a feasible flow in the networks. This method for finding a drawing for H' is optimal with respect to width, height and area, but not necessarily to total edge length. It is based on finding a topological numbering of special directed acyclic graphs and will be explained in Section 6.6.2, because it can also be used as a heuristic for compaction.

6.4 Heuristics

We have seen in the last two sections that there is a close connection between orthogonal drawings of planar graphs and a certain type of combinatorial description. Before we proceed to describe a combinatorial algorithm that uses a network flow approach based on this characterization to optimize the number of bends for a fixed embedding, we discuss a number of heuristic methods that construct good layouts. A main advantage of these methods is their fast running time: typically, it is linear. This makes them more suitable for large problems than the network flow approach, which produces almost quadratic running time. Furthermore, they provide easy procedures for local improvements of a drawing, yielding worst-case bounds for the number of bends and the area, depending on the size of the graph. A final advantage of the heuristic approaches is the fact that some of them work on non-planar graphs.

Typically, the methods work best on 2-connected graphs. A graph is called *2-connected* if removing any vertex and its incident edges leaves a connected graph. The *2-connected components* (or *blocks*) of a connected graph are (a) its maximal 2-connected subgraphs, and (b) its bridges together with their endpoints. If removing $\{v_1, v_2\}$ disconnects the graph we call $\{v_1, v_2\}$ a *cutting pair* of G.

The following result is folklore, see the textbook Sedgewick (1988) for details:

Lemma 6.3. *Testing 2-connectivity and finding all cutting pairs can be done in linear time.*

A natural approach to drawing a graph is to proceed by adding one vertex at a time to an existing drawing. To make sure that any new vertex has the necessary space, a special type of order is chosen:

Definition 6.4. *Let G be a graph. An st-*order *of G is an ordering $\{v_1, v_2, \ldots, v_n\}$ of the vertices of G such that every v_j $(2 \leq j \leq n-1)$ has at least one "predecessor" v_i and at least one "successor" v_k that are neighbors of v_k with $i < j < k$. The edges from v_i to its predecessors (successors) are called* incoming (outgoing) *edges of v_i. Their number is the in-degree indeg(v_i) (out-degree outdeg(v_i)) of v_i.*

If the graph is not 2-connected, there may not be an *st*-order; otherwise, the following theorem holds:

Theorem 6.5 (Lempel et al. 1967, Even and Tarjan 1976).
Let G be a 2-connected graph and $s, t \in V$. Then there exists an st-order such that s is the first and t is the last vertex. It can be computed in $O(m)$ time.

6.4.1 Visibility Representations

A *visibility representation* (Rosenstiehl and Tarjan, 1986) Γ for a graph G maps every vertex v of G to a horizontal segment $\Gamma(v)$ (called *vertex segment*), and each edge (u, v) of G to a vertical segment $\Gamma(u, v)$ (called *edge segment*), such that for each edge (u, v), the edge segment $\Gamma(u, v)$ has its endpoints on the vertex segments $\Gamma(u)$ and $\Gamma(v)$, and does not intersect any other vertex segment. A vertex segment $\Gamma(s)$ is called *source*, if all of its incident edges are above $\Gamma(s)$. A vertex segment $\Gamma(t)$ is called a *sink*, if all its incident edge segments are below $\Gamma(t)$. Figure 6.11 shows a visibility representation of a graph. Visibility representations were introduced by Otten and van Wijk (1978) and are often used as a starting point for drawing a planar graph. The following lemma was proven independently by Rosenstiehl and Tarjan (1986), and Tamassia and Tollis (1986):

Lemma 6.6. *Let G be a 2-connected planar graph with n vertices. Then for any two vertices s and t on the same face of G, there is a visibility representation Γ for G such that:*

(a) Γ has exactly one source, $\Gamma(s)$, and exactly one sink, $\Gamma(t)$.
(b) all the remaining vertex segments $\Gamma(v), v \neq s, t$, have two edge segments incident to $\Gamma(v)$ at its left endpoint (one from below and one from above).

A representation with these properties can be constructed in $O(n)$ time.

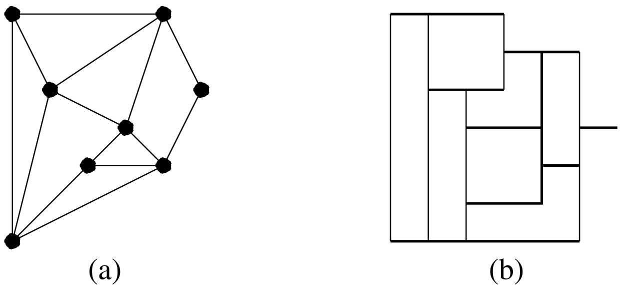


Fig. 6.11. A graph G and a visibility representation for G.

6.4.2 The Algorithm by Tamassia and Tollis

The following algorithm constructs an orthogonal grid embedding as described in Definition 6.2 for a 4-planar graph G. It is due to Tamassia and Tollis (1989) and has linear running time; it produces drawings with at most $2.4n + 2$ bends, and no edge has more than four bends. The length of each edge is $O(n)$, and the total area of the orthogonal grid embedding is $O(n^2)$. If G is 2-connected, we get even better bounds: the number of bends is bounded by $2n + 4$, and only two edges have more than two bends.

The algorithm has four phases. In the first phase, a visibility representation of the graph is constructed. In the second phase, this is transformed into an orthogonal grid embedding. In the third phase, a number of modifications are applied to the orthogonal representation of the grid embedding in order to reduce the number of bends. The last phase computes an orthogonal grid embedding for the orthogonal representation. We start by presenting the individual phases, then summarize the overall algorithm.

Visibility Representation. In Section 6.4.1 it was stated that for each planar 2-connected graph a visibility representation with one source and one sink can be computed in linear time. For planar graphs that are not 2-connected, it cannot be guaranteed that there exists a visibility representation with exactly one source and exactly one sink. However, the number and the degree of the sources and sinks of the visibility representation are crucial for the quality of the drawings produced by the algorithm.

Therefore, the algorithm constructs a visibility representation of a connected graph G with only one source and a low number of sinks. This is done by decomposing the graph into 2-connected blocks that are separated by cut vertices. For each of these blocks a visibility representation is computed according to Lemma 6.6. A cut vertex is chosen as the source, and if possible a cut vertex is chosen as a sink. The visibility representations of the distinct blocks can then be merged, such that one visibility representation for the entire graph is created. For details see Tamassia and Tollis (1989).

Lemma 6.7. *Let $G = (V, E)$ be a connected planar 4-graph. Then a visibility representation Γ for G can be constructed in $O(n)$ time, such that:*

(a) Γ has exactly one source
(b) all the non-source and non-sink vertex segments $\Gamma(v)$ have two edge segments incident to $\Gamma(v)$ at its left endpoint (one from below and one from above).

Creating an Orthogonal Embedding. In the second phase, the transformation of the visibility representation into an orthogonal embedding is performed. This is done by substituting every vertex segment $\Gamma(v)$ of Γ by a structure consisting of a single vertex v and some vertical and horizontal segments. Figure 6.12 shows the corresponding structure for every possible shape of v. Symmetric cases are omitted. It is not hard to see that this can be done in $O(n)$ time, and only a constant number of bends is inserted per edge, for a total of $O(n)$ bends.

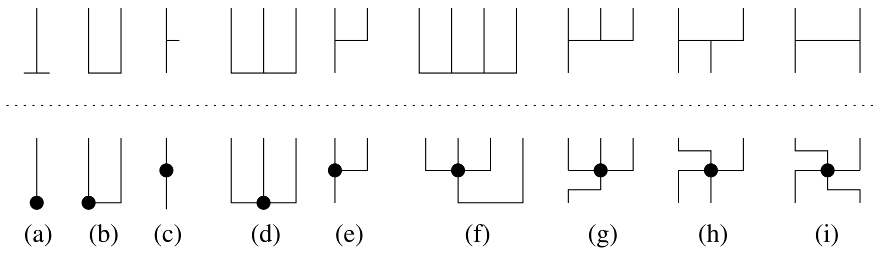


Fig. 6.12. Substitutions of vertex segments with structures.

Bend-Stretching Transformations. Because only one vertex at a time is treated during the substitution process, the embedding may contain a considerable number of artificial bends. During the third phase we try to remove these bends by performing a series of bend-stretching transformations, which are local optimization steps. There are three types of bend stretching transformations, each working on the orthogonal representation H of the embedding. H can be obtained from the orthogonal embedding that is computed in the second phase. A bend on an edge is called *convex* if it forms an angle of $\frac{\pi}{2}$ and *concave* if it forms an angle of $\frac{3\pi}{2}$. The transformations are as follows – see Figure 6.13:

Transformation T1. If an edge (u, v) has both convex and concave bends, remove one bend of each type until the edge has only bends of one type.
Transformation T2. If all edges around a vertex have bends of the same type, these bends can be removed.
Transformation T3. If two edges e_1, e_2, following each other in the clockwise order around a vertex v, form an angle of π, and e_2 has a convex bend with respect to v, then the bend can be removed.

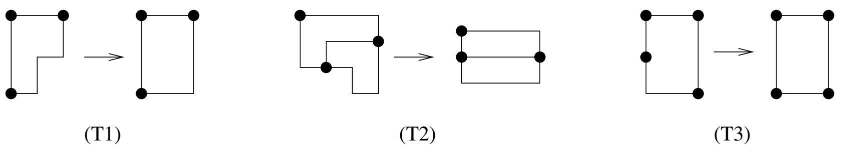


Fig. 6.13. Examples of bend-stretching transformations T1, T2, and T3.

The above transformations generate a new orthogonal representation H' with fewer bends than H. In particular, any transformation of type T1 reduces the number of bends by at least 2, while transformations of type T2 and T3 reduce it by at least 1. Each transformation can be executed in constant time. Since each bend-stretching transformation removes at least one bend, and in the previous step only $O(n)$ bends are introduced, performing these transformations takes only time $O(n)$.

Construction of a Grid Embedding. In the last phase, an orthogonal grid embedding is computed from the orthogonal representation. Details are as described in Section 6.3. Here we only note that for an orthogonal representation with $O(n)$ bends, the drawing can be computed in $O(n)$ time, and the area is $O(n^2)$.

Algorithm 17: Visibility-Grid-Embedding

```
Input: a 1-connected planar 4-graph $G$
Output: an orthogonal planar grid embedding of $G$

construct a visibility representation $\Gamma$ of $G$ using algorithm *Visibility*;
create an orthogonal embedding $\hat{G}$ of $G$ by substituting each vertex-segment
of $\Gamma$ with the appropriate structure listed in Figure 6.12;
calculate the orthogonal shape $H$ of $\hat{G}$;
apply $T_1$ on every edge of $H$ (if possible);
apply $T_2$ on every vertex of $H$(if possible);
apply $T_3$ on every vertex of $H$ with degree $\leq 3$ while possible;
use $H$ to construct an orthogonal planar grid embedding of $G$;
```

Analysis of the Algorithm.

Lemma 6.8. *If the visibility representation constructed in Step 1 of algorithm* Visibility Grid Embedding *has one source, t_i sinks, and n_i non-source vertex segments of degree i, $i = 2, 3, 4$, then the number of bends is at most $n_3 + 2n_4 + t_2 + t_3 + 2t_4 + \delta$, where $\delta = 0, 1, 2, 4$, depending on whether the source has degree 1, 2, 3, or 4.*

Proof. Let n'_4 be the number of vertices resulting from the substitutions of Figure 6.12 (g), (h), and let n''_4 (i) be the number of vertices resulting from substitutions of type (i). Notice that $n'_4 + n''_4 \leq n_4 + t_4$. From the substitutions of Figure 6.4.2, the number of bends after Step 2 is given by

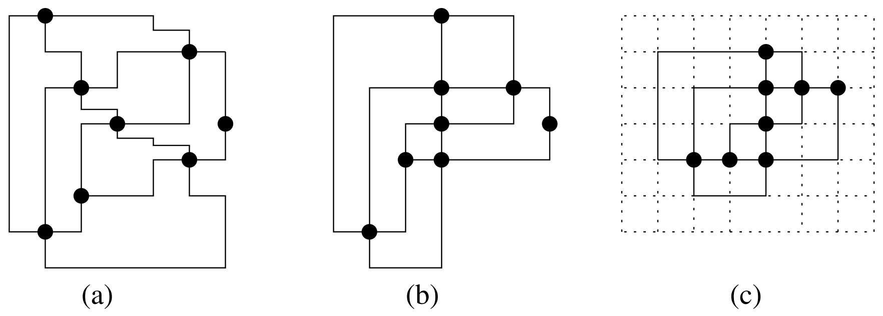


Fig. 6.14. (a) shows the graph of Figure 6.11 (b) after Step 2 of the Algorithm *Visibility Grid Embedding*, (b) after Step 6, and (c) after Step 7.

$n_3 + 4n'_4 + 6n''_4 + t_2 + t_3 + 4t_4 + \delta$. In Step 4, transformation T_1 is applied at least once for each vertex in case of Figure 6.12 (g), (h), and at least twice for each vertex in the case of Figure 6.12 (i). This means that in Step 4 at least $2n'_4 + 4n''_4$ bends are eliminated. Thus, the above equation yields a number of bends of $n_3 + 2n'_4 + 2n''_4 + t_2 + t_3 + 4t_4 + \delta \leq n_3 + 2n_4 + t_2 + t_3 + 2t_4 + \delta$. During steps 5 and 6, this number may only improve.

Theorem 6.9. *Let G be a connected 4-graph with n vertices. Then algorithm* Visibility Grid Embedding *produces a grid embedding of G with at most $2n+4$ bends if G is 2-connected, and $2.4n + 2$ bends if G is connected. The running time is $O(n)$.*

Sketch of Proof. For the case of a 2-connected graph, the visibility representation computed by algorithm *Visibility* has only one sink, and the bound follows immediately from Lemma 6.8. For other connected graphs, it can be shown that the bound $t_3 + 2t_4 \leq m/5$ holds, unless the graphs belong to a certain family of graphs. With the help of this bound, Lemma 6.8 implies the claim. For the exceptions to this bound it is possible to use their special structure for showing that algorithm *Visibility Grid Embedding* needs not more than $2.4n + 2$ bends to draw them. The linear running time follows directly from the above discussion. See Tamassia and Tollis (1989) for the complete proof.

6.4.3 The Algorithm by Biedl and Kant

The algorithm by Tamassia and Tollis relies on the planarity of the graph. It is natural to investigate heuristics that also work for non-planar graphs. Biedl and Kant (1994) have designed an algorithm for constructing an orthogonal grid-embedding of a 2-connected 4-graph G. It starts by computing an *st*-order of the input graph (as defined in Definition 6.4), and then embeds the vertices consecutively in the grid, according to their order in the *st*-order. For each vertex a new row is added to the layout. For each uncompleted edge, i.e., an edge with exactly one endpoint embedded in the grid, a column on the left or right boundary of the existing layout is added.

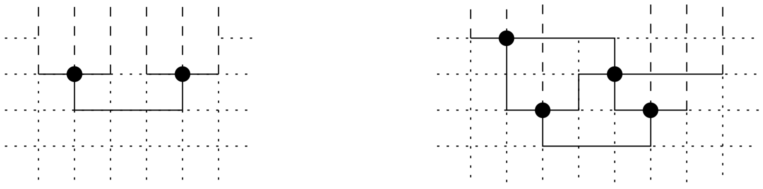


Fig. 6.15. Embedding of the first two vertices and the layout at a later stage.

Algorithm 18: Grid-Embedding

```
Input: a 2-connected 4-graph $G$
Output: an orthogonal grid-embedding of $G$
obtain an $st$-order $\{v_1, v_2, \ldots, v_n\}$ for $G$;
place vertices $v_1$ and $v_2$ on the grid and connect them;
allocate one column in the grid for each edge of $v_1$ and $v_2$, except for the
edge connecting $v_1$ and $v_2$;
for $3 \leq i \leq n$ do
    place $v_i$ on a new row;
    place $v_i$ on a column that is allocated to an incoming edge of $v_i$; if pos-
    sible, do not take the leftmost or rightmost column;
    draw all its incoming edges using the columns allocated to it;
    allocate columns to the outgoing edges of $v_i$ on the left or right bound-
    ary.
```

Lemma 6.10. *The grid size is at most* $(m - n + 1) \times n$.

Proof. Observe that the height of the constructed layout is one less than the number of rows, and the width is one less than the number of columns. In order to embed vertices v_1 and v_2, two rows are used as shown in Figure 6.15. Every following vertex increases the height by one, the last vertex by at most two. Thus, the height is bounded by n. When embedding v_1 and v_2, we use a width of $outdeg(v_1) + outdeg(v_2) - 2$. Every following vertex v increases the width by $outdeg(v) - 1$, except for the last vertex, which increases it by $0 = outdeg(v_n) - 1 + 1$. Thus, the width is $\sum_{v \in V}(outdeg(v) - 1) + 1 = m - n + 1$.

Lemma 6.11. *At most one edge has three bends, all other edges have at most two bends. Overall, there are at most* $2m - 2n + 4$ *bends in the drawing.*

Proof. Every edge (v_i, v_j), $i < j$, bends at most once when v_i is embedded. Completing the edge needs at most one additional bend if $v_j \neq v_n$.

Embedding v_n bends one edge twice, all others at most once, thus only this edge can have three bends. With the embedding of $v \notin \{v_1, v_2, v_n\}$, there are $indeg(v) - 1$ and $outdeg(v) - 1$ new bends, hence $deg(v) - 2$ new bends. Embedding v_1 and v_2 gives $outdeg(v_1) + outdeg(v_2) - 1$ bends, and v_n requires $indeg(v_n)$ bends if $indeg(v_n) = 4$, and $indeg(v_n) - 1$ bends otherwise. As $indeg(v_0) = 0$, $indeg(v_1) = 1$, and $outdeg(v_n) = 0$, we have $\sum_{v \in V}(deg(v) - 2) + 4 = 2m - 2n + 4$ bends if $deg(v_n) = 4$ and $2m - 2n + 3$ bends otherwise.

In (Biedl and Kant, 1994) it is shown that edges with three bends can be avoided unless G is the *octahedron*, i.e., the unique planar 4-regular graph with six vertices. The area bound proven in Lemma 6.10 can be improved even further. The authors show that if G has at most one vertex of degree two, one column and two bends can be saved. This is done by using a special st-order. For $m \geq 2n - 1$, there is at most one vertex of degree two. So we have a width of at most n if $m = 2n$, and a height of $n - 1$ if $m = 2n - 1$. If in addition G is not 4-regular, a row can be saved by choosing a vertex of degree less than four as v_n. This leads to the following theorem:

Theorem 6.12. *Let $G = (V, E)$ be a 2-connected 4-graph. Then G can be embedded in an $n \times n$ grid with at most $2n + 2$ bends. If G is not 4-regular, an $(n-1) \times (n-1)$ grid and $2n - 1$ bends suffice. Each edge is bent at most twice, unless G is the octahedron.*

There is also a variant of the algorithm that produces planar orthogonal drawings of planar graphs with the same bounds on area and number of bends. The above algorithm works only for 2-connected graphs, but it can be extended to connected graphs as follows. Break the graph into its blocks and compute the *block cut vertex tree* of G. The block cut vertex tree of a graph G has a *B-node* for each block of G and a *C-node* for each cut vertex of G. Edges in the block cut vertex tree connect each B-node to the C-nodes associated with the cut vertices of the block. Now the graph is drawn inductively: in the base case, execute the algorithm for a 2-connected graph. In the induction step, consider a subtree of the block cut vertex tree of G and split the subtree into a block G_0 (i.e., the root of the subtree) and the connected subgraphs $G_1, G_2, \ldots, G_q$. By induction hypothesis, each G_i already has a drawing. Hence, the process of drawing G reduces to drawing G_0 and merging each G_i appropriately. The merging process can be done such that the area bounds for the 2-connected case hold also for connected graphs. For details see Biedl and Kant (1994).

6.4.4 Pairing Technique

The algorithm in the previous section made generous use of new columns and rows for additional vertices and edges. The following algorithm tries to reuse as many rows and columns as possible for placing new vertices. In this way, a better area bound is achieved. The algorithm requires 2-connectivity of the input graph. Its authors Papakostas and Tollis (1997d) show that the algorithm can be extended to the simply connected case in a way similar to the algorithm by Biedl and Kant, though, unlike the latter, it does not necessarily produce planar drawings of planar graphs.

The central idea of the algorithm is to form *pairs* of vertices. There are two different kinds of pairs:

Row pairs. The two vertices of a pair are placed in a way that reuses a row in the final drawing of G, i.e., at least two vertices are placed in the same row.

Column pairs. The two vertices of a pair are placed in a way that reuses a column in the final drawing of G, i.e., at least two vertices are placed in the same column.

In order to obtain the pairs, an *st*-order of G is computed. As a next step, each vertex is assigned a type. A vertex with a incoming edges and b outgoing edges is called a vertex of type a-b, or an *a-b vertex* ($1 \leq a, b \leq 4$). If there are 1-1 vertices whose outgoing edges are entering a 1-2 or a 1-3 vertex, we remove these 1-1 vertices and create a new edge between its predecessor and its successor. These removed vertices can be inserted in the drawing at the end of the algorithm without affecting the bounds for area and the number of bends. The graph obtained by removing these vertices is called the *reduced* graph G', and the number of its vertices is denoted by n'. For forming the pairs, the vertices are considered in reverse order of the *st*-order. If a vertex of type 1-2 or 1-3 is encountered, it is paired with its immediate predecessor in the *st*-order. If a vertex of type 2-2 is encountered, the vertex is paired with the next vertex in the *st*-order that is not a 1-1, 2-1, or 3-1 vertex, or a predecessor of the 2-2 vertex. After this step, all 1-2, 1-3, and 2-2 vertices v_i for $3 \leq i \leq n$ are paired. Paired vertices are called *assigned*, vertices not belonging to a pair are *unassigned.* After the pairs are computed, the vertices are embedded into the grid according to the *st*-order. Consider a vertex v that has not yet been embedded. If v is not paired, it is embedded in the grid like in the algorithm by Biedl and Kant. If v is paired, it is embedded together with the second vertex in the pair either as a column pair or as a row pair. The concrete embedding of the pairs is rather technical, for a description see Papakostas and Tollis (1997d). Algorithm 19 summarizes the above steps.

An analysis of the algorithm shows that only for unpaired vertices of type 4-0, 0-4, and 3-1 both a new column and a new row must be allocated when these vertices are embedded. For vertices that do not fulfill these conditions either a new row or a new column must be added to the drawing, but not both. By solving a system of linear equations it can be shown that there can be at most $\lfloor \frac{n+2}{2} \rfloor$ unpaired vertices of type 4-0, 0-4, and 3-1. This leads us to the following theorem:

Algorithm 19: Pair-Orthogonal

```
Input: a 2-connected 4-graph $G$
Output: an orthogonal grid drawing of $G$

compute an $st$-order of $G$;
construct the reduced graph $G'$ from $G$;
obtain a pairing for $G'$;
place $v_1$ and $v_2$ on the grid;
$i := 3$;
while $i < n'$ do
    if $v_i$ has not already been placed then
        if vertex $v_i$ is unassigned then
            place vertex $v_i$ in a new row;
            connect $v_i$ with the incoming edges;
            allocate columns for the outgoing edges of $v_i$;
        else
            place vertex $v_i$ along with the other vertex in the pair, following
            the placement rules described above for the specific type of pair;
    $i := i + 1$;
```

Theorem 6.13. *Let G be a 2-connected 4-graph with n vertices. Algorithm* Pair-Orthogonal *constructs an orthogonal grid drawing of G in $O(n)$ time with area $0.77n^2$. The total number of bends of the drawing is at most $2n+4$, and no edge has more than two bends.*

Proof. Let k_1 and k_2 denote the number of vertices that do not allocate a new row or column when they are embedded. It follows from our above discussion that there are at most $\lfloor \frac{n-2}{2} \rfloor$ vertices contributing neither to k_1 nor to k_2. Thus, $k_1 + k_2 \geq \lceil \frac{n-2}{4} \rceil$. The area is maximized when $k_1 = k_2 = \frac{n-2}{8}$. The claim is now verified by simple calculation. The analysis of bend costs is similar to Lemma 6.11; in addition, the construction allows it to avoid edges with three bends. Using the data structure by Dietz and Sleator (1987), the algorithm can be implemented in $O(n)$ time.

6.4.5 Algorithms for Drawing High-Degree Graphs

So far we have only considered graphs with maximum degree 4, i.e., *4-graphs.* When considering graphs of higher degree, we cannot avoid overlap of edges if we continue to draw vertices as points, so it makes sense to draw vertices as boxes with a sufficient number of grid lines for adjacent edges. In order to use the existing machinery, a straightforward approach is to split high-degree vertices into chains or cycles of vertices before applying an algorithm for 4-graphs like the algorithm by Tamassia (1987) or the algorithm by Biedl and Kant (1994). From this layout, the boxes for the vertices are created. The GIOTTO system (Tamassia et al., 1988) and the quasi-orthogonal drawing algorithm by Klau and Mutzel (1998) follow this approach. Unfortunately, this concept allows no control over the box size of the vertices, so the generated layouts may contain vertices of unrestricted size. Other examples are algorithms for visibility representations (Rosenstiehl and Tarjan, 1986; Tamassia and Tollis, 1986).

A different approach is the KANDINSKY framework, where each vertex is represented by a square. As shown in Figure 6.21, these squares are aligned in a square grid, and edges are routed along an edge grid. Section 6.5.4 will sketch this approach; it extends the network flow technique by Tamassia (1987) for 4-planar graphs that is described in Sections 6.5.2 and 6.5.3.

Papakostas and Tollis (1997c) present an algorithm where the size of any box for a vertex is less than twice the degree of the vertex. Their approach is a generalization of the pairing algorithm presented in the previous section, with boxes instead of vertices being placed on the grid. Outgoing edges leave a box on the top side, incoming edges enter a box on the left or right side. No edges are leaving or entering at the bottom side. Each edge has exactly one bend and the area bound for the algorithm is $m \times \frac{m}{2}$.

In the context of this section, we concentrate on a general framework for generating orthogonal drawings for graphs of high degree that was presented in Biedl et al. (1997a) and Biedl (1997). This *three-phase method* distinguishes the phases vertex placement, edge routing, and port assignment; in addition, there are preprocessing and postprocessing steps. In the preprocessing step, the graph is transformed into a *normalized graph*, i.e., a connected graph without reflex edges and without vertices of degree one. If the input graph is not connected, the connected components are drawn separately. In the first phase, vertices are represented as points and are placed on grid positions. In the second phase, edges are routed between vertices. The drawing obtained after these two phases is called a *sketch*; at this stage, it is not a valid drawing because the edges are routed with overlaps. From this sketch a drawing is produced by adding rows and columns to the drawing, such that vertices are enlarged to boxes. Furthermore, each edge is assigned a port at its vertices such that there is no intersection between any two edges on the same side. Since reflex edges and vertices of degree one were removed in the preprocessing step, they are reinserted in a postprocessing step. As a last step, drawings of the connected components are combined. It should be noted that this framework allows it to handle various kinds of constraints, like constraints on the position of vertices.

An example for this approach is the following algorithm for directed graphs: Place the n vertices of the input graph in *general position* on an $n \times n$ grid, i.e., no two vertices are placed on the same grid line. Edges are routed such that they always leave a vertex on the left or right side, and that they always enter a vertex at its top or bottom. Since the vertices are in general position, exactly one bend is needed to draw any edge. So if $e = (v_i, v_j)$ is an edge directed from v_i to v_j, then we place a temporary bend in the row of v_i and the column of v_j. Next, generate the boxes for the vertices and consider a row r. This row contains one vertex v, and some number of bends. Let $l(v)$ denote the number of edges emerging from the left side of v, and $r(v)$ the number of edges emerging from the right side of v. Analogously, let $b(v)$ and $t(v)$ denote the number of edges leaving the bottom or top side of v. We add $\max\{r(v), l(v), 1\} - 1$ bends above the row of v. We distribute these bends among these rows such that no two edges on one side cross as shown in Figure 6.16. It is clear that the drawings generated by this algorithm have height and width at most m, leading to an area bound of $m \times m$.

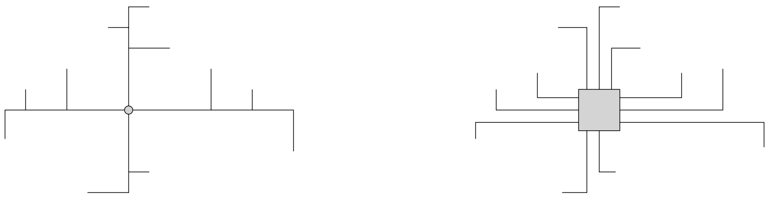


Fig. 6.16. An example for port assignment.

The worst case for the above area bound arises when all edges of the given graph leave the vertex on the same side. The algorithm by Biedl and Kaufmann (1997) achieves a better area bound by balancing the edges across the sides of a vertex box. This can be done by an appropriate placement in the initial phase of placing vertices. For a description, we use the following notation:

Definition 6.14. *Let G be a directed graph and let $\{v_1, \dots, v_n\}$ be an arbitrary vertex order. An edge directed from v_i to v_j is called* good *if $i < j$ and* bad *otherwise. A predecessor (successor) v_i of v_j is* good *if the edge connecting v_i and v_j is good. We denote by $indeg^{good}(v_j)$ and $indeg^{bad}(v_j)$ the number of good and bad incoming edges of v_j. Similarly, we define $outdeg^{good}(v_j)$ and $outdeg^{bad}(v_j)$.*

Now the rows for the vertices are computed in ascending vertex order. If a vertex v has no good predecessor, then we create a new row at an arbitrary place. Otherwise, we add a row close to the median of the good predecessor rows. Vertex v is placed in this new row. Similarly, a column for each vertex is computed, proceeding in reverse order, and considering good successors instead of good predecessors.

The following lemma shows how vertex order and edge orientation affect the number of edges connected to each side.

Lemma 6.15. *For each vertex, we have the bounds $\lfloor outdeg(v)/2 \rfloor \leq r(v)$, $l(v) \leq \lceil outdeg^{good}(v)/2 \rceil + outdeg^{bad}(v)$ and $\lfloor indeg(v)/2 \rfloor \leq t(v)$, $b(v) \leq \lceil indeg^{good}(v)/2 \rceil + indeg^{bad}(v)$.*

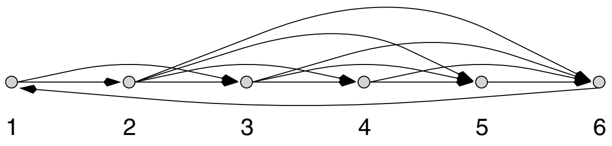
Proof. Consider $b(v)$. By the placement of bends, any bend at the top of v belongs to an incoming edge of v. By the placement of vertices, at most half (rounded up) and at least half (rounded down) of the good predecessors are below v. Nothing can be said about the place of the bad predecessors. The result follows for $b(v)$. The proofs for the other three sides are similar.

It follows from the above lemma that the number of bad predecessors and successors of a vertex v should be minimized in order to reduce the size of a vertex box. The next lemma shows that there always exists a vertex order and an edge orientation such that there is a low number of bad predecessors and successors.

Definition 6.16. *A vertex order together with an edge orientation is called* polar-free almost acyclic, *if*
(a) $\mathit{indeg}(v) \geq 1$ *and* $\mathit{outdeg}(v) \geq 1$ *for all* $v \in V$, *and*
(b) $\mathit{indeg}^{bad}(v) \leq 1$, *if* $\mathit{indeg}^{good}(v) \geq 0$ *then* $\mathit{indeg}^{bad}(v) = 0$, *and*
(c) $\mathit{outdeg}^{bad}(v) \leq 1$, *if* $\mathit{outdeg}^{good}(v) \geq 0$ *then* $\mathit{outdeg}^{bad}(v) = 0$.

Lemma 6.17. *Let G be a simple graph without vertices of degree zero or one. Then G has a polar-free almost acyclic order and orientation. It can be found in $O(m)$ time.*

For a proof of this lemma see Biedl and Kaufmann (1997).



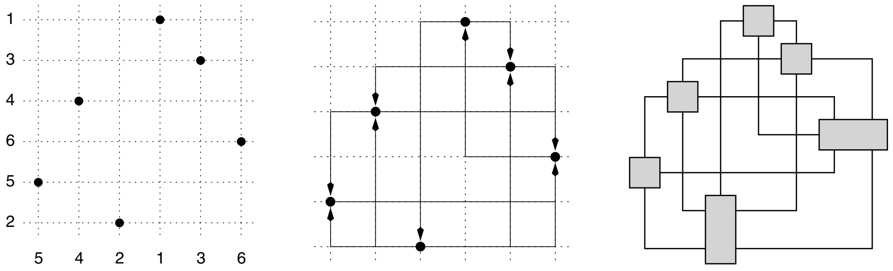


Fig. 6.17. Example of a run of the algorithm with polar-free almost acyclic order and orientation of the input graph.

Theorem 6.18 (Biedl and Kaufmann 1997). *Let G be a simple and connected graph. Then G has an orthogonal drawing in an $\frac{n+m}{2} \times \frac{n+m}{2}$-grid with one bend per edge. The drawing can be found in $O(m)$ time.*

Proof. We only show the claim for the height, the claim for the width is similar. Suppose that G has no vertices of degree one. After the vertex placement, we have n rows. For each vertex v, we add $\max\{r(v), l(v), 1\} - 1$ rows. Thus, the height is $\sum_{v \in V} \max\{1, r(v), l(v)\}$. By Lemma 6.15 and the conditions on the polar-free almost acyclic order, we have $r(v), l(v) \leq \lceil \frac{outdeg(v)}{2} \rceil$. Thus, $\sum_{v \in V} \max\{1, r(v), l(v)\} \leq \sum_{v \in V} \frac{outdeg(v)}{2} + 1 = \frac{m+n}{2}$.

It remains to be shown that vertices of degree one can be inserted in the drawing without violating the area bound. Consider a vertex v of degree 1 in the postprocessing phase. First, one row is added above the top side of the neighbor vertex of v. Then a new column is generated, such that the width of the neighbor vertex increases by one and no adjacent edges of the neighbor vertex cross this new column. Then v is placed in the new row and column. No bend is needed to connect this vertex to its neighbor.

The vertex order and edge orientation can be found in $O(m)$ time as shown in Lemma 6.17. By using the data structure of Dietz and Sleator (1987), the algorithm can be implemented in $O(m)$ time. Thus, the overall complexity of the algorithm is $O(m)$.

An interactive version of the algorithm can be found in Biedl and Kaufmann (1997). A version of this algorithm that considers constraints can be found in Wiese and Kaufmann (1998).

6.4.6 A Divide-and-Conquer Approach

Now we describe a divide-and-conquer approach that originates from VLSI design. We already noted in Section 6.3 that there is a close relationship between graph drawing and VLSI design. While the first considers vertices and edges, the latter deals with transistors and wires. Clearly, there is a correspondence between the respective objects, but it is not without any problems, as wires and transistors need a certain amount of area. Thus, an important parameter for VLSI design is the *minimum feature size* λ, which is the width of the narrowest wire that can be manufactured. The smallest transistor that can be manufactured is a square with edge length λ and area λ^2. Further difficulties may arise from the fact that a general graph may have arbitrary degree, whereas a transistor can only have a limited number of connections. We resolve these difficulties by restricting us to graphs with vertices of a degree bounded by a constant, and by further assuming that vertices occupy only a constant area of silicon.

The VLSI model used here is similar to that of Thompson (1980), where wires have unit width and only two wires may cross at a point. Vertices are represented by little boxes that are placed on a rectangular grid, so that each box lies within a grid square. Edges run horizontally and vertically, one per grid square, except that an edge running horizontally may cross one running vertically. See Figure 6.18 for an example.

Layouts in this model are *sliceable*. That is, a horizontal or vertical line can be used to bisect the layout, the pieces can be moved apart, and the severed wires can be reconnected to realize the original graph connections. Slicing can be used to introduce a new edge in an existing layout as follows: perform two vertical and two horizontal cuts through the layout to separate the vertices. Separate the pieces by a grid unit, and reconnect the severed edges across the gaps in order to connect the vertices. If the length of the original layout was L and the width W, the new layout has length at most $L + 2$ and width at most $W + 2$.

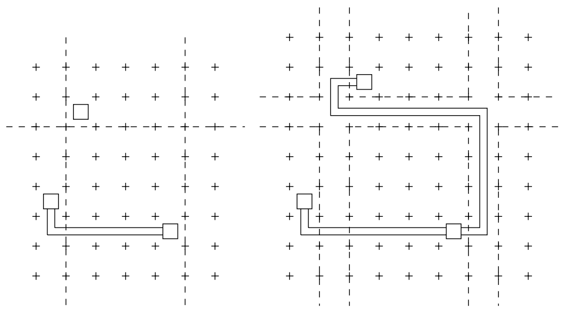


Fig. 6.18. An example for slicing and edge routing.

This property can be exploited to generate layouts for graphs in a divide-and-conquer approach. First, the graph is divided into unconnected components by removing edges. Then these components are laid out separately and the removed edges are inserted by slicing. Note that there may be crossings inserted in the slicing process, which implies that the produced drawing may be non-planar.

The quality of the layout produced by this approach depends on how many edges have to be removed to obtain unconnected components that differ in size by only a constant. In the worst case we have to remove $O(n)$ edges to obtain two unconnected components with this property, but there exist classes of graphs that can be separated by removing fewer edges. These classes of graphs are characterized by *separator theorems*. It is crucial that the classes are closed under the subgraph relation, i.e., each subgraph of a graph in such a class is again in the class. An example is the class of planar graphs, as it is closed under taking subgraphs. On the other hand, the class of trees is not closed under the subgraph relation.

Definition 6.19 ($f(n)$-separator). *Let S be a class of graphs that is closed under the subgraph relation. An $f(n)$-separator Theorem for S is a theorem of the following form: There exist constants α_s and c_s where $0 < \alpha_s \leq \frac{1}{2}$ and $c_s > 0$, such that if G is a graph with n vertices in S, then by removing at most $c_s f(n)$ edges, G can be partitioned into disjoint subgraphs G_1 and G_2*

having αn and $(1-\alpha)n$ vertices respectively, where $\alpha_s \leq \alpha \leq 1-\alpha_s$. The set of removed edges is called the cut set *of the bisection.*

Lipton and Tarjan (1970) showed that any planar graph with n vertices can be divided into two subgraphs of approximately the same size by removing $O(\sqrt{n})$ vertices in $O(n)$ time. When removing k nodes to split a graph with maximum degree d, at most $d \cdot k$ edges are removed from it. Since we are only considering graphs with bounded degree, and planar graphs are closed under the subgraph relation, this means that planar graphs have a $\sqrt{n}$-separator theorem. It is easy to see that forests have a 1-separator, see Valiant (1981) for a proof.

The analysis of the algorithm is quite complicated and results in solving recurrence equations. Leiserson (1980) and Valiant (1981) showed independently that graphs belonging to a graph class with a $\sqrt{n}$-separator can be drawn using $O(n \log^2 n)$ area and that graphs belonging to a graph class with a 1-separator can be drawn using $O(n)$ area.

Corollary 6.20. *Let G be a planar graph with n vertices and bounded degree. There is a layout of G, such that the area occupied by G is $O(n \log^2 n)$.*

For trees with maximum degree 4 one can use a method that is different from slicing for edge routing, which avoids crossings, but guarantees the same area bound (Valiant, 1981).

Corollary 6.21. *Let T be a tree with n vertices and bounded degree. There is a layout of T, such that the area occupied by T is $O(n)$. If T has maximum degree less than or equal to 4, there exists a planar layout with the same area bound.*

6.5 Flow-Based Methods

6.5.1 Drawing Graphs with Few Bends

We have seen in the preceding sections that over the years, orthogonal drawings of graphs have received a large amount of attention: there have been many different approaches to finding such drawings with desirable properties, like a small number of bends in the rectilinear paths representing edges. Unfortunately, one of the results by Formann et al. (1990) shows that it is $\mathcal{NP}$-complete to decide whether a planar graph with maximum degree 4 (a "4-planar graph") has an orthogonal drawing without any bends. The crux of the reduction is the fact that finding the right order of edges around each vertex in a drawing is difficult.

This difficulty arises when we are only given the information provided by vertices and edges, but have to find an optimal embedding – as in the hardness proof by Formann et al. It should be noted that for the special case of 3-planar graphs, it was shown by Di Battista et al. (1998a) that it is possible to optimize the number of bends in a drawing in polynomial time. This implies that it is the existence of degree 4 vertices in a graph that makes the problem difficult. As it was shown by Didimo and Liotta (1998), there are algorithms with polynomial complexity if the number of these vertices is bounded; the running time is exponential in the number of degree 4 vertices.

For many graphs that need to be drawn, the embedding is fixed, so the $\mathcal{NP}$-hardness result does not apply. In fact, it was shown by Tamassia (1987) that for a fixed embedding of a 4-planar graph, there is a nice combinatorial algorithm that computes a drawing with the smallest possible number of bends. We spend the rest of this section describing the idea of Tamassia's algorithm, and sketch some implications and extensions. For simplicity of notation, we do not distinguish between combinatorial objects (e.g., edges in a graph) and the geometric objects representing them (e.g., edge segments in a drawing).

6.5.2 A Network for Angles

Suppose we are given a planar graph $G = (V, E)$, and a fixed embedding of G, described by a clockwise order $\langle e_1, \dots, e_{d(v)} \rangle$ of edges around any vertex $v \in V$. Let F be the set of faces in this fixed embedding, with f_0 being the exterior face. In any orthogonal drawing of G, the angles $\phi(e_1, e_2)$ between adjacent edge segments e_1 and e_2 are multiples of $\frac{\pi}{2}$. We use the notation $\psi(e_1, e_2) = 2\phi(e_1, e_2)/\pi$, write $E(v)$ for the (ordered) set of edge segments adjacent to a vertex $v \in V$, and $\Psi(f)$ for the set of angles in a face f. Then we can write the following conditions on these angles:

$$\sum_{e_i \in E(v)} \psi(e_i, e_{i+1}) = 4 \qquad \text{for all } v \in V. \tag{6.4}$$

$$\sum_{p(e_i, e_{i+1}) \in \Psi(f)} \psi(e_i, e_{i+1}) = 2|\Psi(f)| - 4 \quad \text{for all } f \in F \setminus \{f_0\}. \tag{6.5}$$

$$\sum_{p(e_i, e_{i+1}) \in \Psi(f_0)} \psi(e_i, e_{i+1}) = 2|\Psi(f_0)| + 4. \tag{6.6}$$

$$\psi(e_i, e_{i+1}) \geq 1. \tag{6.7}$$

Note the correspondence of these conditions to the linear relaxation described in Section 6.2.

Each angle p_i occurs exactly twice in this system of conditions, once at a vertex or bend in an edge, and once as an angle of a face. If we think of the size of these angles as amount of flow of some entity, we can introduce a network as follows:

1. There is a "source" node n_v for each vertex $v \in V$.
2. There is a "sink" node n_f for each face $f \in F$.
3. There is an arc $a_{v,f}$ from node n_v to node n_f if vertex v is incident to face f.
4. There is a source s, connected to all nodes n_v, and a sink t, connected from all nodes n_f.
5. For any two adjacent faces $f_1, f_2 \in F$, there are two arcs a_{f_1,f_2} and a_{f_2,f_1}.

Flow among these arcs has the following significance:

The sink node allocates angles to the vertices; as described, any vertex has a total sum of angles summing up to 4, so we fix the flow $x_{s,v}$ to this amount. A flow of $x_i \geq 1$ units on arc $a_{v,f}$ indicates that the angle p_i incident to vertex v and face f has size x_i. By requiring flow conservation at each node n_v, we make sure that condition (6.4) is valid. The lower bound (6.7) guarantees that each angle is positive. An example is shown in Figure 6.19.

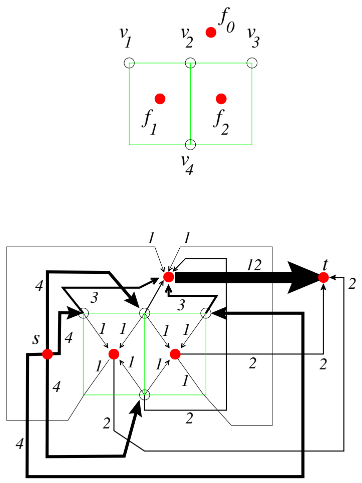


Fig. 6.19. An orthogonal drawing of a graph (top); flows in the corresponding network, with the amount of flow for each arc indicated by numbers and arc thickness (bottom). At the bottom, original edges are grey, and only arcs with positive flow are shown.

Furthermore, a flow of x_i units on arc $a_{v,f}$ indicates that the angle p_i incident to vertex v and face f has size x_i. A bend in an edge between two faces f_1 and f_2 creates a "reflex" angle of 3 on one side, and a "convex" angle of 1 on the other. In conditions (6.5) and (6.6), any angle in $P(f)$ contributes an angle of 2 to the face balance. The difference is accounted for by a flow of one unit from the face with the reflex angle to the face with the convex angle. This leaves a total amount of $2|V(f)| - 4$ for the angles at the set of vertices $V(f)$ incident to faces $f \in F \setminus \{f_0\}$, and $2|V(f_0)| - 4$ for the set of angles $V(f_0)$ incident to the exterior face f_0. By requiring this flow on the arcs $a_{f_i,t}$, and requiring flow conservation, conditions (6.5) and (6.6) are kept valid.

In addition to the above nodes and arcs, the following capacities and costs are defined:

1. Any arc $a_{v,f}$ from node n_v to node n_f gets capacity 4, and cost 0.
2. Any arcs a_{f_1,f_2} gets unbounded capacity, and cost 1 per unit of flow.

These capacities arise from the fact that no angle is larger than 4, but any edge can have an unbounded number of bends. Since our objective function is the total number of bends, and any unit of flow in an arc a_{f_1,f_2} corresponds to one bend, the cost function is chosen this way.

6.5.3 Optimal Flow in the Network and Implications

It is clear from our above discussion that any feasible drawing of G corresponds to a feasible flow in the network. Furthermore, if there is a feasible flow, the integrality of the arc capacities implies that there is a feasible solution where the flow on each arc is integral. Using the methods described in Section 6.3, it is straightforward to derive a feasible orthogonal representation for this flow. We have shown in Section 6.3 that any feasible orthogonal representation can be used to construct a feasible drawing in linear time. This leaves it to find a minimum cost feasible flow in the network, a classical problem of combinatorial optimization. See the book by Ahuja et al. (1993) for an overview over the basic algorithmic approaches. We summarize:

Theorem 6.22 (Tamassia 1987). *For a fixed embedding of a planar graph G with n vertices, an orthogonal drawing with a minimum number of bends can be found in time that is required for finding a minimum cost flow on a network with $O(n)$ arcs and $O(n)$ vertices.*

In 1987, the resulting complexity was $O(n^2 \log n)$. Currently, the best running time is $O(n^{\frac{7}{4}}\sqrt{\log n})$, using an improved network flow algorithm by Garg and Tamassia (1997).

There are a number of consequences and extensions. Any modification of a feasible flow that leads to a flow of reduced cost can be interpreted as a local improvement of a drawing. See Figure 6.20 for a number of examples, where the improvement of the flow is performed by identifying a cycle of negative cost in a reduced cost network. (Considering these types of local improvements in flow networks is a standard approach.) Any unit of flow along an arc in a negative cycle implies that we should increase an angle by a single multiple of $\frac{\pi}{2}$ at the expense of another. In the figure, such flow is indicated by the places where the cycle crosses an edge, a vertex, or a bend in the drawing. Performing these changes along the full cycle reduces the total number of bends along the encountered places, while all parts inside and outside of the cycle keep the same angles. As shown in the figure, this corresponds to a “rotation” of the inside against the outside.

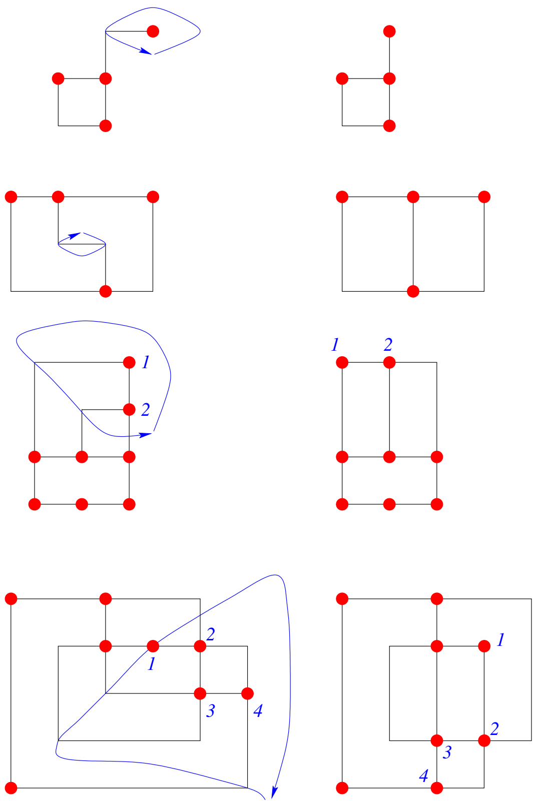


Fig. 6.20. Four examples: orthogonal drawings with an improving cycle in the corresponding flow (left); improved drawings (right).

Furthermore, it is straightforward to extend the ideas of Tamassia to flow networks for other types of grids. One example is the treatment of drawings in a hexagonal grid, where angles are multiples of $\frac{\pi}{3}$. However, these type of drawings allow angles of size $\frac{\pi}{3}$ or $\frac{2\pi}{3}$ at a bend; in order to use the network flow approach, angles of the first type have to be considered to carry twice the cost of angles of the latter type. Another issue is the question of finding a feasible drawing for a given flow (treated in Section 6.3 for the orthogonal case), which is unresolved, as the number of degrees of freedom in a hexagonal grid is different from the geometric dimension. See the paper by Tamassia (1987).

6.5.4 Kandinsky

If we need to draw planar graphs with maximum degree beyond 4, drawing vertices as points must create overlap, as the degree of a vertex exceeds the number of different orthogonal directions. As we described in Section 6.4.5, one possible remedy is to draw nodes not as points, but as boxes. A variant that allows it to make use of flow optimization techniques is given by the KANDINSKY model, which was first introduced by Fößmeier and Kaufmann (1995). Here, all vertices are given as identical $k \times k$ squares, with the size k determined appropriately. These squares are aligned on a square grid of size $(2k-1) \times (2k-1)$, and edges are routed as axis-parallel paths along the grid lines running through boxes. See Figure 6.21 for an illustration.

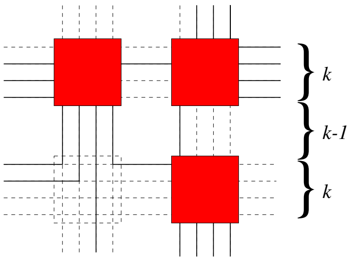


Fig. 6.21. The basic layout of vertices and edges in the KANDINSKY model.

Obviously, there are a number of technical issues that have to be taken care of, among them the choice of grid size, and more generally compaction methods that are modified from the steps described in Section 6.3. Details can be found in the original paper by (Fößmeier and Kaufmann, 1995), and in the thesis (Fößmeier, 1997b). Here we concentrate on the modifications of the flow network that have to be performed.

The key modification of the flow network for bend minimization arises from the fact that for a vertex of degree larger than four, there have to be edges leaving in the same direction. Clearly, neighboring edges of this type enclose an interior angle of 0; for the network described above, only positive angles are feasible. This can be fixed by allowing a flow of -1 to represent a zero angle, which can be interpreted as a flow in the opposite direction.

While this fix takes care of the angles, it creates another problem: since these flows do not incur any cost, it is possible to shift flow until the overall cost is zero. Thus, a minimum cost flow does not correspond to a feasible drawing.

However, if we exclude parallel edges (which can be done in a preprocessing step), any pair of edges enclosing an angle of 0 at one vertex must connect this vertex to two different vertices. This means that at some point, the two edges cannot continue to run in parallel. Because of the underlying grid in the Kandinsky model, this can only occur when one of the two edges bends. Thus, an angle of 0 forces a bend in one of the enclosing edges. We can charge this forced bend for the zero angle to the vertex – see Figure 6.22. If f is the face with the zero angle at vertex v, with neighboring faces g and h, then the forced bend can be interpreted as a unit of flow from g or h to v, at a cost of 1.

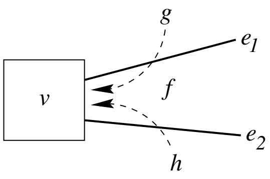


Fig. 6.22. Auxiliary arcs at a vertex.

After this modification, there are new issues that need to be taken care of: we only want to charge the cost for a zero angle once, and if we charge the arc from g to v, we must not charge the arc from h to v. This can be resolved by using the modified network for the flow between v, f, g, and h as shown in Figure 6.23, using additional edges with cost $2c + 1$ and $-c$. (Note that for clarity in Figure 6.23, the reference to vertex v in the labeling of the auxiliary nodes H is omitted.) In particular, the arc from a face f to an incident vertex v with edges e_i and e_j is represented by a path formed by the following arcs:

– Arcs with capacity 1 and cost $2c + 1$ from f to an auxiliary node $H_i^{v,fg}$ and $H_j^{v,fh}$. Here, g and h are the faces separated from f by e_i and e_j.
– Arcs with capacity 1 and cost $-c$ from $H_i^{v,fg}$ to $H_i^{v,gf}$, and vice versa.
– Arcs with capacity 1 and cost 0 from $H_i^{v,fg}$ to an auxiliary node H_g^v.
– Arcs with capacity 1 and cost 0 from H_g^v if v lies on the boundary of g.

This introduces cycles with negative cost into the network, while a flow that is feasible for a drawing must be decomposable into partial flows from s to t. Thus, we are no longer dealing with a classical minimum cost flow problem. However, using minimum cost flow algorithms based on augmentations along shortest paths from s to t still yields the desired result that an optimal flow corresponds to a feasible drawing with a minimum number of bends.

6.5.5 Constraints and Extensions

There are many algorithms that arise from the basic ideas described in the previous section. Some of them are able to consider a variety of constraints by using integer programming methods. See (Eiglsperger et al., 2000) for a description.

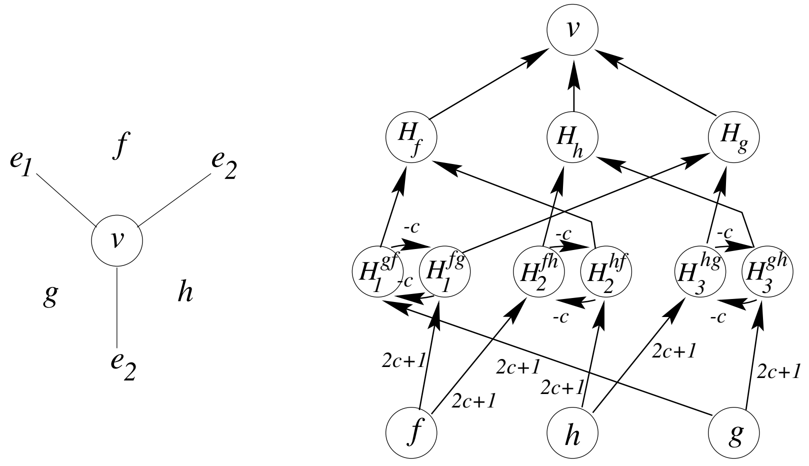


Fig. 6.23. The neighborhood of a graph (left); the auxiliary nodes and arcs in the flow network (right).

There are other graph drawing problems where the objective is closely related to what we described in the previous sections, but may yet have some differences. We give a few pointers to and descriptions of these issues.

While flow models minimize the total number of bends, it may very well be that the objective is to minimize the maximum number of bends in any edge. As Fößmeier et al. (1996) demonstrated, it is possible to construct a drawing of a planar graph in the Kandinsky-model, such that no edge has more than one bend. This allows it to minimize the total number of bends under this restriction: the constraint can be enforced by imposing an upper limit of 1 on the capacities of the face-to-face arcs in the flow network. Similarly, other upper bounds on the number of bends for individual edges can be handled.

There have been efforts to use flow models for bend minimization if the embedding is not fixed. Clearly, serious difficulties are to be expected, as we pointed out at the beginning of this section: since the problem for a non-fixed embedding is $\mathcal{NP}$-hard, such approaches can only be expected to lead to heuristics, or we may have to accept a worst-case running time that is not polynomial. As we already mentioned above, there is an algorithm by Didimo and Liotta (1998) for finding a drawing with a minimum number of bends for non-fixed embedding that is only exponential in the number of degree four nodes. Extending earlier work by Bertolazzi et al. (1997), the algorithm proceeds by a branch-and-bound search. Branching steps correspond to local modifications of the current graph embedding; the full set of possible embeddings is maintained by using a special data structure called an SPQ*R-tree. The four different types of tree nodes (S, P, Q*, and R) correspond to different types of triconnected components that allow a limited number of local modifications. The flow method by Tamassia is used as a subroutine. Experiments seem to indicate that the algorithm may be practical for test graphs of up to 200 nodes, since the distribution of degree four nodes in the test graphs tends to keep running times significantly below the worst case estimates.

It has been attempted to use flow techniques even for nonplanar graphs, by making use of sufficient a priori knowledge of the location of edge crossing. Details are quite tricky and technical; see the thesis (Fößmeier, 1997b).

6.6 Compaction

Compaction is the process of changing a given orthogonal layout, so that either the area, the total edge length, or the maximum edge length decreases. In this section, we will focus on compaction techniques that maintain the topology and shape properties of the input.

Most of the algorithms described in this section have their roots in VLSI design, but have been adapted to solve the compaction problems in graph drawing. In addition, we present two recent developments originating in the area of graph drawing. An overview of the techniques in VLSI layout is given in Lengauer (1990, Chap. 10) and in LaPaugh (1998, Sect. 23.3). Before describing the compaction techniques we state the underlying compaction problems in a formal way, and we discuss their complexity in Section 6.6.1. One-dimensional algorithms attack the problems by dividing them into two separate subproblems for the horizontal and vertical direction. They are covered in Section 6.6.2: we focus on the compression-ridge technique and the graph-based compaction strategies. Finally, Section 6.6.3 is dedicated to optimal methods.

6.6.1 Problems and Their Complexity

Depending on the various aesthetic criteria we want to optimize, there are several versions of compaction problems originating in the topology-shape-metrics approach. The input of the third phase within this paradigm is an orthogonal representation H. Such a representation may have been produced by the flow-based methods of Section 6.5. By introducing auxiliary vertices, we may assume that H is simple. The task is to find an orthogonal grid embedding respecting the shape of H with either minimum total edge length, minimum area, or minimum length of the longest edge. We will refer to these problems as $\mathrm{COMP}_{\mathrm{sum}}$, COMP_{A}, and $\mathrm{COMP}_{\mathrm{max}}$, respectively. It is also possible to state the problems for existing orthogonal grid embeddings, especially as a postprocessing step for drawings as produced in Section 6.3. In this case the task is to change the coordinates of vertices and bends, but not the angles formed by the edge segments. These formulations may seem somewhat restrictive in the case of existing drawings (in a more general scenario, it may be possible to introduce or remove bends), but even then, they may serve as a local improvement step.

Almost all variants of the two-dimensional compaction problem in VLSI design are $\mathcal{NP}$-hard. In most cases, the corresponding proofs are reductions of the problem 3Partition (see Garey and Johnson (1991)). They exploit the fact that in VLSI problem formulations, wires may be allowed to swap their connection points at components. Such a swap corresponds to a change of the embedding and is not allowed in graph drawing problem formulations. These changes are crucial for the reductions and cannot be used in a setting with fixed topology.

For a long time it was conjectured that the compaction problems (i.e., COMP_{sum}, COMP_A, and COMP_{max}) for a fixed embedding are also $\mathcal{NP}$-hard. Recently this was proven by Patrignani (1999a). His proof for COMP_A is based on a reduction of the satisfiability problem SAT (see Garey and Johnson (1991)). Given a formula ϕ, there is an orthogonal representation $H_A(\phi)$ with n_A vertices and m_A edges, and area at most $(9n_A+2)(9m_A+7)$, if and only if ϕ is satisfiable. The reductions for COMP_{sum} and COMP_{max} are similar.

6.6.2 One-Dimensional Compaction Methods

Due to the large sizes of instances, research in VLSI design has focused on one-dimensional methods. Only one dimension may be changed at a time; the other dimension is fixed. We will refer to the restricted, one-dimensional compaction problems as $\text{COMP}^1_{\text{sum}}$, COMP^1_A, and $\text{COMP}^1_{\text{max}}$, respectively.

After a compaction step, the layout is changed: alternating the direction and performing another step results in an iterative process. However, at each step the decisions are purely local, and compaction in one direction may prevent greater progress in the other direction. Furthermore, the layout may be blocked in both dimensions, but still be far away from an optimal solution (see Figure 6.24 for an example).

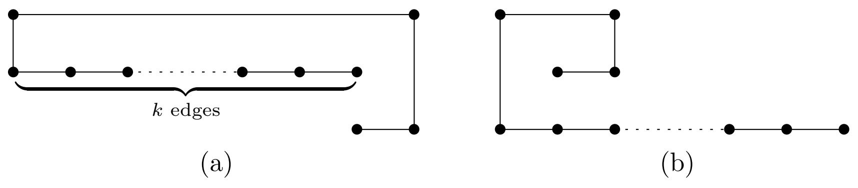


Fig. 6.24. (a) Both directions are blocked, the total edge length is $2k + 5$. (b) An optimal layout with edge length $k + 6$.

Originating in VLSI design, the *compression-ridge method* searches the layout for cuts that divide it into two parts and pass through regions of empty space. For fixed embedding the “empty space” corresponds to edges that are longer than the minimum length of one unit. If such a cut has been found, its edges can be shortened by at least one unit and the resulting grid embedding is still feasible.

We sketch the method from (Dai and Kuh, 1987), adapted to different scenarios in the area of graph drawing. All cuts are found as an interpretation of a maximum flow in a network N that depends on the initial drawing. A compaction step in x-direction for the example introduced in Section 6.3 (page 130) is shown in Figure 6.25. Compaction in y-direction can be explained similarly. First the layout is dissected into horizontal stripes. This corresponds to the dissection process described in Section 6.3 with the restriction that only artificial edges of horizontal direction are allowed. A modification of the dissection method still runs in linear time; the result is a drawing with internal faces of rectangular shape. Now the network N can be constructed as follows: each rectangular face f corresponds to a node $n(f)$ in N. In addition, there are two nodes s and t for the outer face; s at the top of the drawing, t at the bottom. Arcs are directed downwards: for each horizontal edge e separating an upper face f from a lower face g, there is an arc $a_e^+ = (n(f), n(g))$ and an arc $a_e^- = (n(g), n(f))$. The capacity of a_e^+ is the length of e minus one. This corresponds to the maximal possible shortening of e. A capacity of ∞ is assigned to the opposite arc a_e^-, accounting for possible elongations of e. The maximum flow from s to t in this network corresponds to the shortening that can be applied to obtain a minimum width drawing; thus, we get a layout of optimal horizontal width. Each compaction step has running time $O(n \log n)$, the bottleneck being the computation of a maximum flow problem in N. (By construction, the network N is planar and linear in size of the original graph.)

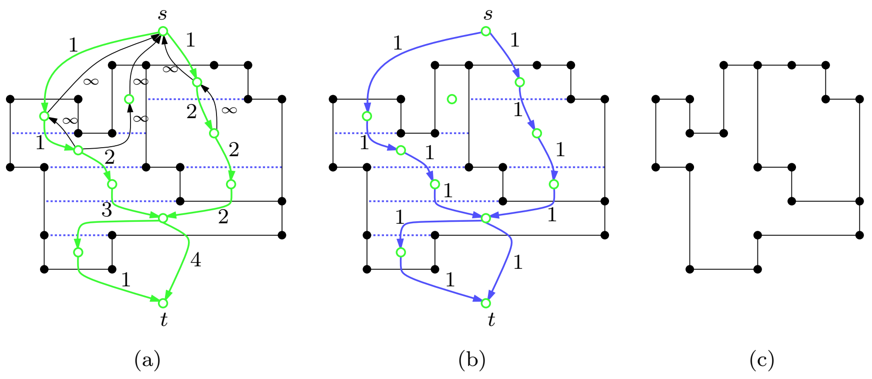


Fig. 6.25. The compression-ridge method in graph drawing: (a) the network N for x-compaction of Figure 6.9 (d) (only some of the unbounded upward arcs are shown); (b) the maximum flow in N; (c) the drawing after the horizontal compaction step.

So-called *graph-based compaction methods* represent a different and more efficient approach: Two *layout graphs* – one for each direction of the compaction – encode the visibility properties between maximally connected vertical and horizontal paths in the given grid embedding. These paths are also referred to as *bars* in Di Battista et al. (1999), *maximal chains* in Bridgeman et al. (1999), or *segments* in Klau and Mutzel (1999b).

Definition 6.23. *A horizontal* segment *is a maximally connected component in (V, E_h), the subgraph of G containing only the horizontal edges. Similarly, we define vertical segments in (V, E_v). The sets S_h and S_v refer to the horizontal and vertical segments, and the set $S = S_h \cup S_v$ refers to all segments. A vertex v lies on the two unique segments* $\mathrm{hor}(v) \in S_h$ *and* $\mathrm{ver}(v) \in S_v$*.*

The directed layout graphs $D_x = (V_x, A_x)$ and $D_y = (V_y, A_y)$ are built as follows: the node set V_x of the horizontal graph D_x corresponds to the set of vertical segments S_v. A similar construction applies to D_y, here $V_y = S_h$.

For an arc set A let $\mathrm{trans}(A)$ be the transitive hull of A. Geometric relations between the segments define the arc sets in the digraphs: whenever a horizontal segment s_i is to the left of another horizontal segment s_j, we want to find a directed path between s_i and s_j. We characterize the vertical relationships analogously. More formally, we want to have

$$\mathrm{trans}(A_x) = \{(s_i, s_j) \mid s_i \text{ is to the left of } s_j\} \quad \text{and} \tag{6.8}$$

$$\mathrm{trans}(A_y) = \{(s_i, s_j) \mid s_i \text{ is below } s_j\} \ . \tag{6.9}$$

Any sets with properties (6.8) and (6.9) can be used as the arc sets of the layout graphs. Figure 6.26 shows two layout graphs for the running example in this section with arc sets produced by a sweep-line method.

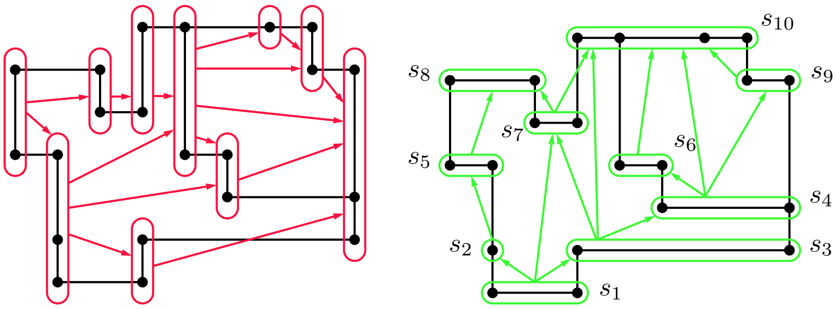


Fig. 6.26. The directed layout graphs D_x (left) and D_y (right).

Each arc corresponds to a distance constraint for a pair of segments. Since the visibility properties must be maintained in one-dimensional compaction (recall that the coordinates of the other direction are fixed) an arc (s_i, s_j)

describes the fact that all the vertices of s_j must be assigned a greater coordinate than the one for vertices of s_i. Hence, the task of one-dimensional compaction in x-direction reduces to computing a topological numbering for the nodes in D_x. Similarly, a vertical compaction step corresponds to a topological numbering in D_y. Since the layout graphs are acyclic by construction, such an order Φ can be computed in time $O(|V_x| + |A_x|)$ or $O(|V_y| + |A_y|)$, e.g., with the longest path method. It is easy to show that D_x and D_y are planar; thus, $|V_x|$, $|V_y|$, $|A_x|$, and $|A_y|$ are in $O(n)$. Therefore, the running time for computing the topological numbering Φ is linear in the size of the original graph.

We illustrate the method by performing vertical compaction on the example graph. Consider D_y in Figure 6.26. The longest path method results in the following topological numbering $\Phi : S_h \rightarrow \mathrm{Z}$.

$$\begin{array}{lllll} \Phi(s_1) = 0 & \Phi(s_2) = 1 & \Phi(s_3) = 1 & \Phi(s_4) = 2 & \Phi(s_5) = 2 \\ \Phi(s_6) = 3 & \Phi(s_7) = 2 & \Phi(s_8) = 3 & \Phi(s_9) = 3 & \Phi(s_{10)} = 4 \ . \end{array}$$

The new vertical coordinate of a vertex v is just the topological number of hor(v). Setting all y-coordinates in this manner results in the compacted drawing with minimum possible height in a one-dimensional setting, as shown in Figure 6.27 (a).

Note, however, that this method tends to push vertices as far to the bottom as possible (or to the left when performing a horizontal compaction step). Each segment gets its minimum topological number. For segments lying on the longest path tree, this is the optimal assignment; other segments should rather be placed closer to their neighbors than to the bottom or left margin. In Figure 6.27 (a) this has no negative influence on the aesthetics, but Figure 6.27 (b) shows an example where this is the case. Though the drawing has minimum width and height, the bottom edge is drawn longer than its one-dimensional minimum length of one grid unit. In addition to the unpleasant drawing, these edges might prevent the following compaction step in the other direction from a better performance.

Minimizing the total one-dimensional edge length corresponds to minimizing the difference between the topological numbers. In the area of VLSI this problem is known as *wire balancing*. For vertical compaction, the corresponding optimization problem is

$$\begin{aligned} \min \quad & \sum_{(v,w)\in E_v} \Phi(\mathrm{hor}(w)) - \Phi(\mathrm{hor}(v)) \qquad (6.10) \\ \text{s.t.} \quad & \Phi(s_j) - \Phi(s_i) \geq 1 \qquad \text{for all } (s_i, s_j) \in A_y \ . \end{aligned}$$

Note that (6.10) is the same problem as the layer assignment for layered drawings of graphs (see Section 5.3). The optimization problem can be seen as the dual of a flow problem, as shown by the following steps. Let $\Delta_{\mathrm{ver}}(s)$ denote the *vertical degree* of a horizontal segment $s \in S_h$, defined as

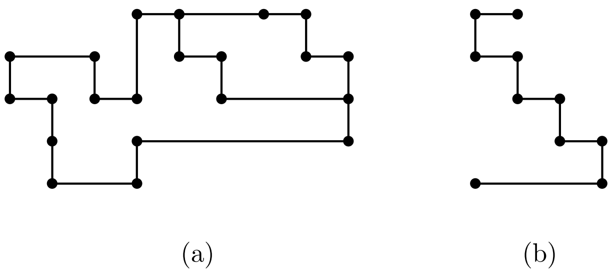


Fig. 6.27. The graph-based compaction method with longest-path computations: (a) the example from Figure 6.9 after a vertical compaction step; (b) the method does not lead to optimal layouts with respect to the one-dimensional compaction problem $\mathrm{COMP}^1_{\mathrm{sum}}$.

$$\Delta_{\mathrm{ver}}(s) = |\{(v,w) \in E_v \mid \mathrm{hor}(v) = s\}| - |\{(u,v) \in E_v \mid \mathrm{hor}(v) = s\}|. \quad (6.11)$$

Then $\sum_{s \in S_h} \Delta_{\mathrm{ver}}(s) = 0$, and the optimization problem (6.10) becomes

$$\begin{aligned} \min \quad & \sum_{s \in S_h} \Delta_{\mathrm{ver}}(s)\Phi(s) & & (6.12)\\ \text{s.t.} \quad & \Phi(s_j) - \Phi(s_i) \geq 1 \qquad \text{for all } (s_i, s_j) \in A_y. \end{aligned}$$

The dual of (6.12) is

$$\begin{aligned} \max \quad & \sum_{a \in A_y} \Psi(a) & & (6.13)\\ \text{s.t.} \quad & \sum_{a=(s,t)} \Psi(a) - \sum_{a=(r,s)} \Psi(a) = \Delta_{\mathrm{ver}}(s) & & \text{for all } s \in S_h,\\ & \Psi(a) \geq 0 & & \text{for all } a \in A_y. \end{aligned}$$

The objective function of this problem can be written as

$$\min \sum_{a \in A_y} -\Psi(a),$$

leading to the standard form of a minimum cost flow problem. This implies that there is a polynomial-time method for finding an optimal solution for the one-dimensional compaction problems COMP^1_A and $\mathrm{COMP}^1_{\mathrm{sum}}$.

6.6.3 Optimal Compaction Methods

There are several methods for finding an optimal solution for two-dimensional compaction problems of the types $\mathrm{COMP}_{\mathrm{sum}}$, COMP_A, and $\mathrm{COMP}_{\max}$ that we have introduced in Section 6.1. Both the algorithms by Kedem and Watanabe (1984); Watanabe (1984), and by Schlag et al. (1983) are based on a branch-and-bound approach and originate in VLSI design. Recently, there have been developments in the area of graph drawing: Bridgeman et al. (1999) specify a class of orthogonal representations for which optimal drawings with respect to area and total edge length can be found employing the methods from Section 6.3. Klau and Mutzel (1999b) present an optimal branch-and-cut approach for minimizing the total edge length for a given orthogonal representation; they also characterize classes of representations for which their algorithm runs in polynomial time.

The algorithm by Kedem and Watanabe (1984); Watanabe (1984) is based on a translation of the two-dimensional compaction problem COMP_A into a nonlinear mixed integer programming formulation that is solved by a branch-and-bound algorithm. They express the problem as minimizing the nonlinear area function under a set of linear and nonlinear constraints. Their formulation, however, sacrifices the general statement of COMP_A as defined in Section 6.1 and considers only a subset of the feasible solutions in order to achieve a better running time. In the following we sketch the branch-and-bound algorithm.

A vector of decision variables d determines the interaction of components (remember that components correspond to vertices of a graph drawing instance). In the given formulation, only two positions are possible for a pair of components, coded as an entry in the 0/1-vector d. Each combination corresponds to a different relative placement of the component pair – either a horizontal or a vertical constraint is active. The entries in d correspond to arcs in the layout graphs, i.e., a fixed d specifies two (possibly infeasible) one-dimensional compaction problems. This formulation has the drawback of being able to handle only two-way choices instead of four possible relative placements. Though the area of the computed layout is optimal for a given partial order, it may not be optimal for an instance of COMP_A as formulated in Section 6.1. The authors propose a postprocessing step to determine where the partial order of elements has to be swapped, but they do not present a method that guarantees an overall optimal solution.

A fixed set of relative positioning decisions – corresponding to a node in the branch-and-bound tree – results in two one-dimensional problems. They are solved using the graph–based longest path method described in Section 6.6.2. If the subproblem is infeasible, the tree of problems can be cut at this node. If the node is a leaf and a feasible solution is found that is better than the previous global upper bound, the bound is updated. Otherwise, a solution may cause an update of the local lower bound. If the latter becomes greater than the global upper bound, an optimal solution cannot be found below the current node, and again the tree is cut at this point.

In general, an optimal solution for the restricted problem can be found in short time by using this method. However, the proof of optimality may be very time-consuming.

A different branch-and-bound approach was proposed by Schlag et al. (1983). They give a characterization of feasible layouts in terms of satisfiability of a special Boolean expression.

If the distance constraints between two components i and j in the layout process are not fulfilled, the pair is called a *violation*. Four constraints $c_{ij}^1, c_{ij}^2, c_{ij}^3$, and c_{ij}^4 define the relative placement of elements i and j. A violation can be seen as the nonemptiness of the intersection between the two rectangles R_{ij} and j, shown in Figure 6.28. The rectangle R_{ij} contains element i; at each of the four sides it is enlarged by the appropriate minimum distance to element j.

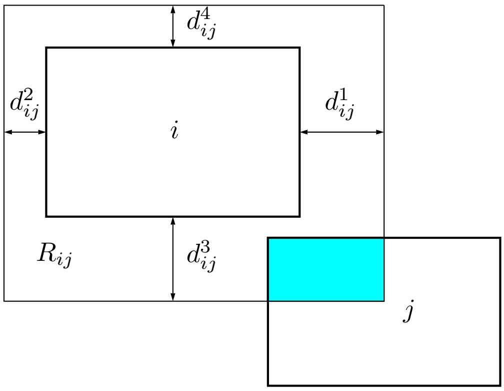


Fig. 6.28. A violation formed by elements i and j.

With each constraint $c \in \cup_{1 \le i < j \le n} \{c_{ij}^1, c_{ij}^2, c_{ij}^3, c_{ij}^4\}$ and each layout P, we also associate a logical variable. The variable is "true" if the constraint is fulfilled in P and "false" otherwise. Then a legal layout, i.e., a feasible orthogonal grid embedding, is characterized by the following properties:

– It satisfies the base constraints determining the sizes of the elements.
– For every c the formula F is "true", where

$$F = \bigwedge_{i,j} \left(c_{ij}^1 \vee c_{ij}^2 \vee c_{ij}^3 \vee c_{ij}^4 \right) \quad . \tag{6.14}$$

For a practical application, the size of set F is too big. The basic idea of the two-dimensional compaction algorithm is to start with $F = \{\}$, and to obtain a so-called *smashing* by solving the system of inequalities with the longest path method from Section 6.6.2. A smashing is a possibly illegal layout that respects only the set of constraints in the current set F. Then the algorithm determines a violation (i, j) in the smashing by means of a rectangle intersection algorithm, searching for situations like in Figure 6.28. Once such a pair (i, j) is found, a branching step is performed: in each of the four subproblems, a different constraint c_{ij}^k ($k \in \{1, \ldots, 4\}$) is added to the set of active constraints F. Then the algorithm calls the procedure recursively with each of the four different sets $F + \{c_{ij}^1\}, \ldots, F + \{c_{ij}^4\}$.

An optimal solution is obtained like in the algorithm by Kedem and Watanabe. If the subproblem is a leaf, the generated constraints are either inconsistent, or the layout is legal and may become the new upper bound. Computations at inner nodes in the branch-and-bound tree have the following effects: illegal subproblems and problems exceeding the global upper bound cause the algorithm to cut the tree at the current node; otherwise, the objective value of the smashing becomes the new local lower bound.

This concludes the overview of optimal VLSI methods for two-dimensional compaction problems. Though not applicable to the typically huge instances of VLSI problems, the algorithms can be useful for the compaction of orthogonal grid embeddings. The rest of this section is dedicated to recent developments in the area of orthogonal graph drawing.

As seen in Section 6.3, there is a class of orthogonal representations where the two-dimensional compaction problems can be solved to optimality–these are representations with faces not containing forbidden sub-shapes, resulting in inner faces of rectangular shape. The work by Bridgeman et al. (1999) studies the class of orthogonal representations and introduces so-called *turn-regular orthogonal representations* to devise polynomial-time heuristics for the compaction problems in graph drawing.

An orthogonal representation H is *turn-regular*, if it does not contain opposite angles inside of a face. A pair of angles is *in opposition*, if it forms one of the 18 configurations shown in Figure 6.29.

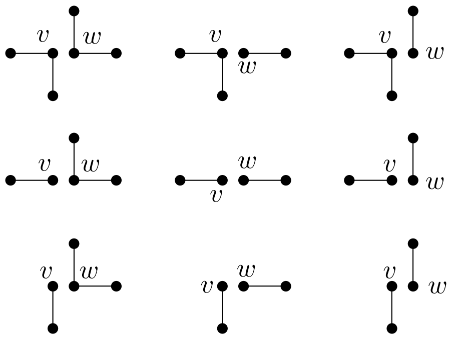


Fig. 6.29. Nine configurations for a pair of opposite angles inside of a face. The other nine can be obtained by 90° rotation.

Turn-regularity of an orthogonal representation can be tested in linear time. The authors show that the relative position of every pair of vertices is defined, if and only if the underlying orthogonal representation is turn-regular. This implies that a drawing that respects all the relative positions is a feasible orthogonal grid embedding.

For a turn-regular representation, the networks introduced in Section 6.3 can be used in order to obtain a minimum area drawing in linear time. Furthermore, the $O(n^{7/4}\sqrt{\log n})$ time algorithm introduced in Garg and Tamassia (1995a) can compute the drawing with minimum total edge length within this optimal area.

Based on this theoretical background, the following compaction heuristic can be applied to any orthogonal representation H: First, H is tested for turn-regularity with a linear-time algorithm. If the test is positive, an optimal drawing can be computed in polynomial time. Otherwise, the heuristic turns the non-regular faces into regular ones. Techniques similar to the dissection method presented in Section 6.3 can be used for this purpose, e.g., it is possible to insert straight artificial edges between pairs of opposite vertices. In general, the drawings resulting from this dissection method are better than the ones from Section 6.3, but are still far away from an optimal solution. Figure 6.30 shows a drawing for the example from Section 6.3, constructed with this heuristic method.

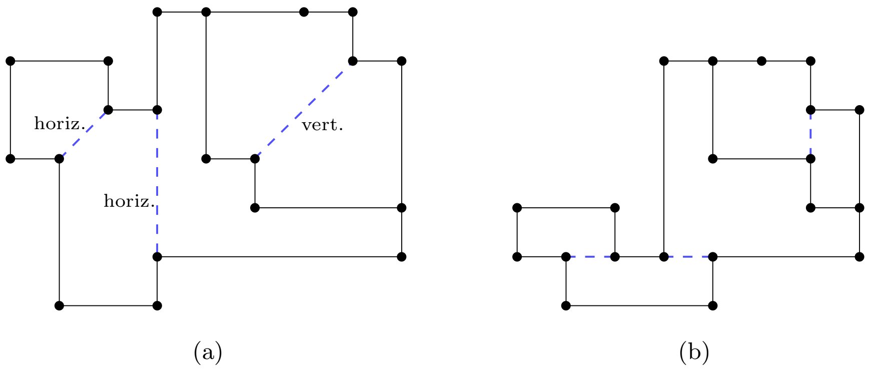


Fig. 6.30. The heuristic based on turn-regularity applied to the example from Figure 6.9: (a) pairs of opposite angles have been linked by either horizontal or vertical artificial edges. The resulting representation is turn-regular and compactable in polynomial time.

The method by Klau and Mutzel (1999b) solves the problem of minimizing the total or maximal edge length for a given orthogonal representation. It makes use of a necessary and sufficient condition for all feasible solutions of a given instance of the compaction problem. This condition is based on existing paths in so-called *constraint graphs*. This pair of graphs is similar to the layout graphs defined in Section 6.6.2. As in one-dimensional graph-based compaction, nodes in these graphs represent the segments (see Definition 6.23), and arcs characterize relative positioning relations.

Figure 6.31 shows an example of a pair of constraint graphs. The arcs specify exactly the relative relationships known from the given simple orthogonal representation H. Each edge in H determines the relative position of two segments in every feasible orthogonal grid embedding for H. Pairs of constraint graphs whose arc sets consist of all such arcs are also called *shape descriptions*.

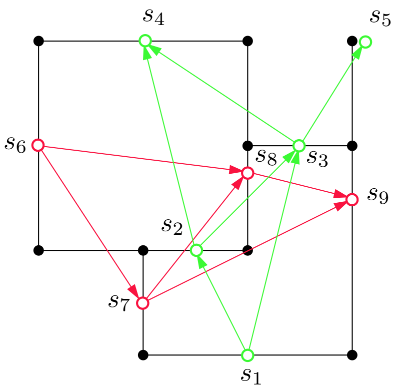


Fig. 6.31. A pair of constraint graphs that is a shape description. Each segment is limited by two horizontal and two vertical segments. The left limit of segment s_3 is $l(s_3) = s_8$, its right limit is $r(s_3) = s_9$. The bottom and top limits of s_3 are the segment itself, i.e., $b(s_3) = t(s_3) = s_3$.

The optimal compaction method is based on the following observations:

- The arcs of a shape description are contained in the layout graphs of every drawing that reflects the given shape.
- Most frequently, the information in a shape description σ is not enough to produce a feasible orthogonal grid embedding. Respecting only the relative positioning constraints encoded in σ may lead to crossings and overlapping edges. If this is not the case, however, we call such a pair of constraint graphs *complete*.
- In general, there are many possibilities for extending a shape description to a complete pair of constraint graphs.

Let $u \xrightarrow{*} v$ denote the existence of a directed path from u to v. The following is a precise characterization of complete pairs of constraint graphs in terms of paths that must be contained in the arc sets: a pair of graphs is complete if and only if both arc sets are acyclic and for every pair of segments $(s_i, s_j) \in S \times S$, one of the following four conditions holds:

$$\begin{array}{llll} 1. & r(s_i) \xrightarrow{*} l(s_j), & 3. & t(s_j) \xrightarrow{*} b(s_i), \\ 2. & r(s_j) \xrightarrow{*} l(s_i), & 4. & t(s_i) \xrightarrow{*} b(s_j). \end{array} \tag{6.15}$$

In this definition, $l(s), r(s), b(s)$, and $t(s)$ denote the limits of a segment s as introduced in Figure 6.31. If one of the conditions applies we also call the pair of segments *separated*.

We can now express a one-to-one correspondence between these complete extensions and feasible orthogonal grid embeddings. For each simple orthogonal drawing with shape description σ, there exists a complete extension τ of σ and vice versa: every complete extension τ of a shape description σ corresponds to a simple orthogonal drawing with shape description σ.

Hence, the compaction task can be seen as the search for a complete extension of the given shape description leading to minimum total edge length (or minimum maximal edge length). One way to characterize the set of complete extensions is by means of an integer linear program (ILP). We introduce a binary variable x_{ij} for each arc (s_i, s_j) that may be part of some extension of the given shape description $\sigma = \langle (S_v, A_h), (S_h, A_v) \rangle$. If (s_i, s_j) is contained in the extension, the corresponding variable x_{ij} is one; otherwise, it is zero. We refer to the set of arcs in σ by $A = A_h \cup A_v$ and to the set of potential additional arcs by A^+. In addition, there is a variable $c_s \in Z$ for each segment $s \in S$ denoting the coordinate of s. This yields the following ILP:

$$\min \sum_{e \in E_h} c_{r(e)} - c_{l(e)} + \sum_{e \in E_v} c_{t(e)} - c_{b(e)} \tag{6.16}$$

subject to

$$\begin{aligned} x_{r_i,l_j} + x_{r_j,l_i} + x_{t_j,b_i} + x_{t_i,b_j} &\geq 1 & \forall (s_i, s_j) &\in S \times S & (6.16.1)\\ c_j - c_i &\geq 1 & \forall (s_i, s_j) &\in A & (6.16.2)\\ c_j - c_i - (M+1)x_{ij} &\geq -M & \forall (i,j) &\in A^+ & (6.16.3)\\ x_{ij} &\in \{0,1\} & \forall (i,j) &\in A^+ & (6.16.4) \end{aligned}$$

Inequalities (6.16.1) model the characterization of separation, i.e., the existence of necessary paths in an extension as required by conditions (6.15). In this formulation, r_i is short for the segment $r(s_i)$; the same abbreviation applies to all other limits. Inequalities (6.16.2) force the coordinates to obey the distance rules coded by the arcs in the underlying shape description. The same must hold true for the potential additional arcs: whenever a variable x_{ij} has value 1, we want an inequality of type (6.16.2); otherwise, there should be no restriction on the coordinate variables. This situation is modeled by inequalities (6.16.3) with the help of a big constant M. The authors show that in a feasible solution, the corresponding arc sets are acyclic and the entries of the coordinate vector c integral.

Like the one-to-one correspondence between complete extensions and feasible orthogonal grid embeddings, there is a one-to-one correspondence between feasible solutions of the ILP and complete extensions of the given shape description.

To formulate $\text{COMP}_{\max}$, i.e., the minimization of the longest edge, the ILP has to be slightly modified. Only a linear number of inequalities have to be added and the objective function must be changed for that purpose.

For the class of turn-regular orthogonal representations defined in Bridgeman et al. (1999), there is only one complete extension of the corresponding shape description. In a preprocessing phase, the algorithm extends the given shape description as far as possible by adding arcs when there is only one possibility of meeting the four conditions (6.15). In case of complete constraint graphs, the integer linear program decomposes into two separate one-dimensional compaction problems that can be solved with the algorithm from Section 6.6.2, which is optimal in the one-dimensional case. Figure 6.32 shows a drawing for the example graph that is optimal with respect to total edge length, constructed with the branch-and-cut algorithm. It should be noted that the algorithm performs quite well on medium-sized instances, despite of its exponential worst-case time complexity.

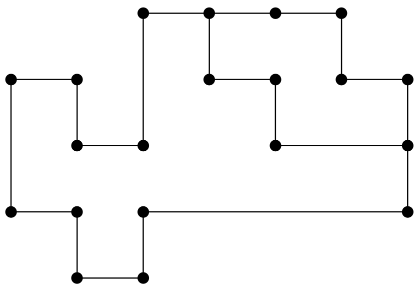


Fig. 6.32. An optimal solution of the two-dimensional compaction problem produced by the branch-and-cut algorithm (example from Figure 6.9).

To conclude this section, observe that the aesthetic criteria “area” and “edge length” may contradict each other, even when the corresponding orthogonal representation is turn-regular and the underlying shape description is complete. Unlike the rectangular case, optimality of the two criteria does not always coincide (see Figure 6.33).

6.7 Improving Other Aesthetic Criteria

In this section we present efficient postprocessing routines for improving other aesthetic criteria. Here the focus is on efficiency rather than optimality. In addition to decreasing area and edge length, these techniques aim at reducing the number of bends, the number of crossings, and the sizes of vertices.

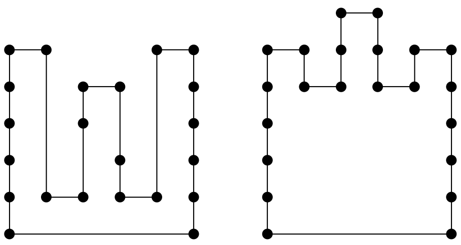


Fig. 6.33. Underlying shape description and orthogonal representation are complete and turn-regular, respectively. Nevertheless, minimum area (left) and minimum total edge length (right) exclude each other. (The example is taken from Patrignani (1999a).)

Few efforts have been made in this direction. The bend-stretching transformations by Tamassia and Tollis (1989) as presented in Section 6.4 fall in this category. They may reduce the number of bends in an orthogonal drawing. The *refinement algorithm* by Six et al. (1998) provides an additional set of elementary transformations to reduce the number of crossings. A more complicated and less efficient approach is given by Fößmeier et al. (1998). They also consider changing the sizes of vertices in order to get smaller drawings and fewer bends.

The goal of the algorithm by Six et al. (1998) is to find an efficient way for obtaining a better drawing in terms of area, number of bends, number of crossings, and total edge length. To improve area and edge length, they use the linear-time one-dimensional compaction method described in Section 6.6.2. The number of bends is reduced by using the bend-stretching transformations introduced by Tamassia and Tollis (1989) that are described in Section 6.4.2.

In addition, Six et al. (1998) consider the following configurations that can be removed in order to increase readability of the orthogonal drawing.

1. U-Turns are three consecutive edge segments forming two 90° angles (as shown in Figure 6.34 (a)).
2. Poorly placed degree two vertices are those which are neither on a bend nor distributed evenly in the drawing (Figure 6.34 (b)).
3. Self-crossings occur between two edges that are incident to the same vertex. The authors distinguish between near and far self-crossings (Figure 6.34 (c)).
4. A stranded vertex has only one neighbor that is placed far away (as shown in Figure 6.34 (d)).

A preprocessing phase constructs a so-called *abstracted graph* G' by deleting vertices with degree at most two. In some cases, the following simple procedures can repair the configurations of Figure 6.34: if a U-turn is found, the algorithm checks whether the middle segment can be moved towards the ends of the “U”. If the necessary space is available, this operation can save cross-

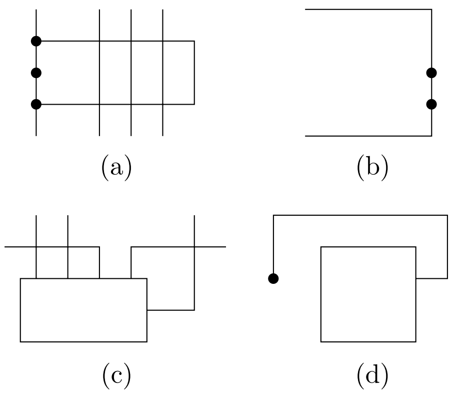


Fig. 6.34. The four additional configurations considered in Six et al. (1998): (a) a U-turn; (b) poorly placed degree two vertices; (c) self-crossings (near and far); (d) a stranded vertex.

ings and reduce total edge length. The bends on edges e that represent chains of degree two vertices are redistributed, so that either they lie on bends of e, or they are in the middle of an edge segment. Another operation removes near self-crossings by swapping the affected edges. For far self-crossings, the procedure tries to reroute the edges. Finally, stranded nodes are placed as close as possible to their neighbors. The authors give an $O(n+m)$ time bound for their refinement algorithm.

Another approach to postprocessing is the *4M-algorithm* by Fößmeier et al. (1998). It consists of the four operations *moving*, *matching*, *morphing*, and *merging*.

The *moving* operation is a one-dimensional compaction method similar to the compression ridge method presented in Section 6.6.2. The authors introduce a *moving line* (corresponding to an s-t flow) to cut the drawing into two parts. They propose a depth-first search to find this line more efficiently. Moving is shown in Figure 6.35 (a).

Matching resembles the third bend-stretching transformation in Tamassia and Tollis (1989). The aim is to save bends by moving vertices to the geometric places of bends. The technique, however, is different. Analogous to the moving line in the preceding operation, a *matching line* is used for finding these configurations. A theoretical characterization of matching lines can be found in Tamassia (1987) and in Di Battista et al. (1999). Figure 6.35 (b) illustrates the matching operation.

The *morphing* procedure saves bends by changing the size of a vertex v, drawn as a box. This operation is the inverse to shrinking vertices by introducing bends, which is used to get an orthogonal drawing from a visibility representation, described in Section 6.4.2. The basic idea of morphing is to expand v in direction of a close bend b so that the geometric representation of v covers b. Then the operation changes the box of v to its smallest possible size. Figure 6.35 (c) demonstrates an application of a morphing step. There are many cases, however, where this operation is not applicable: vertices may grow too big or overlap other parts of the drawing. Again, a *morphing line* is used for finding configurations where the operation can be performed successfully.

The last operation in the 4M-algorithm is *merging*. This operation aims at reducing the sizes of vertices drawn as a box and is illustrated in Figure 6.35 (d). Merging is a combination of inverse morphing and matching. First it introduces a bend b by resizing a vertex v in order to place a neighbor w of v on the position of b. As a result, either the width or the height of vertex v decreases by one grid unit.

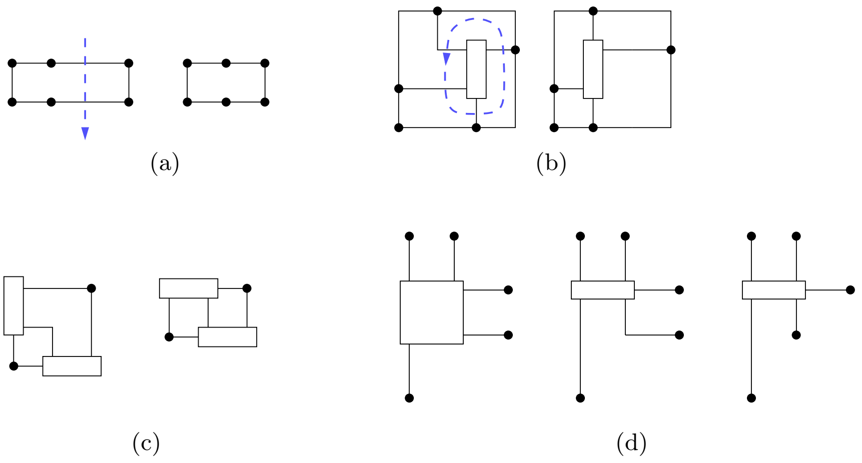


Fig. 6.35. The 4M-algorithm: (a) moving; (b) matching; (c) morphing; (d) merging.

Different variants of the 4M-algorithm run in $O(n^2)$ or $O(n \log n)$ time.

6.8 Conclusions and Open Problems

We have described a number of models and methods for orthogonal drawings of graphs. As we saw in the beginning, these problems are somewhat related to the issue of angles in drawings.

Drawing edges as axis-parallel paths makes it relatively easy to give combinatorial descriptions of these drawings. This allows it to use combinatorial arguments for getting a first drawing, as well as methods from mathematical programming for improving it. This basic approach of discretization can also be applied by using other grids; however, the resulting combinatorial issues may be harder to resolve.

Many of the performance guarantees of the presented heuristics still leave a gap between the number of bends in a drawing that can be achieved and the number of bends that may be necessary. It is conceivable that some of these gaps will be narrowed or closed.

Another elegant application of methods from mathematical programming is given by the polynomial flow-based algorithms for bend minimization. Like in some other cases, an immediate application is restricted to the relatively small class of planar graphs with maximum degree 4, but there are some extensions to cope with other models of orthogonality. A possible approach is to draw vertices as boxes; doing this in a specific manner leads to the KANDINSKY model, but others are possible.

The same applies to local improvement of the metric quality of a drawing, i.e., compaction. If we are dealing with a graph that has vertices of degrees exceeding 4, it easy to perform graph-based compaction on orthogonal drawings where vertices of high degree are represented as boxes. Crossings can be modeled as virtual vertices of degree four. The algorithms can be changed so that they can also process input drawings in Kandinsky-style, i.e., drawings with different grids for vertices and edges. There are many more variations of the problems that can be formulated elegantly so that a solution is found using the methods presented in this chapter.

We conclude by listing some of the open problems concerning the quality of orthogonal drawings:

– Can we give good approximation algorithms for drawing a 4-planar graph with few bends if the embedding is *not fixed*?
– Can we extend these approximation algorithms to general planar graphs?
– Are there approximation algorithms for classes of non-planar graphs?
– Are there more classes of orthogonal representations for which appropriate compaction algorithms find the optimal drawings in polynomial time?
– Several algorithms in this chapter operate in a fixed embedding or fixed shape setting. How do the optimal drawings with respect to the aesthetic criteria change if the embedding and/or the shape may be changed?
– To date, no approximation algorithms exist for the compaction problems. It would be very interesting to have efficient heuristics with a good performance guarantee.
– Can some of the ideas for planar drawings be extended to three dimensions?

Some aspects of the last question are discussed in Chapter 7.