6

Graph Drawing Algorithms

Peter Eades
University of Sydney

Carsten Gutwenger
Dortmund University of Technology

Seok-Hee Hong
University of Sydney

Petra Mutzel
Dortmund University of Technology

6.1 Introduction **6**-1
6.2 Overview **6**-3
Drawing Conventions • Aesthetic Criteria • Drawing Methods
6.3 Planar Graph Drawing **6**-8
Straight Line Drawings • Orthogonal Drawings
6.4 Planarization **6**-16
Edge Insertion with Fixed Embedding • Edge Insertion with Variable Embedding • Remarks
6.5 Symmetric Graph Drawing **6**-19
Two-Dimensional Symmetric Graph Drawing • Three-Dimensional Symmetric Graph Drawing • Geometric Automorphisms and Three-Dimensional Symmetry Groups • Algorithm for Drawing Trees with Maximum Symmetries
6.6 Research Issues **6**-24
6.7 Further Information **6**-24
Defining Terms **6**-24
References **6**-25

6.1 Introduction

Graphs are commonly used in computer science to model relational structures such as programs, databases, and data structures. For example:

- Petri nets are used extensively to model communications protocols or biological networks.
- Call graphs of programs are often used in CASE tools.
- Data flow graphs are used widely in Software Engineering; an example is shown in Figure 6.1a.
- Object oriented design techniques use a variety of graphs; one such example is the UML class diagram in Figure 6.1b.

One of the critical problems in using such models is that the graph must be drawn in a way that illuminates the information in the application. A good graph drawing gives a clear understanding of a structural model; a bad drawing is simply misleading. For example, a graph of a computer network is pictured in Figure 6.2a; this drawing is easy to follow. A different drawing of the same graph is shown in Figure 6.2b; this is much more difficult to follow.

A graph drawing algorithm takes a graph and produces a drawing of it. The graph drawing problem is to find graph drawing algorithms that produce good drawings. To make the problem more precise, we define a graph $G = (V, E)$ to consist of a set V of vertices and a set E of edges, that

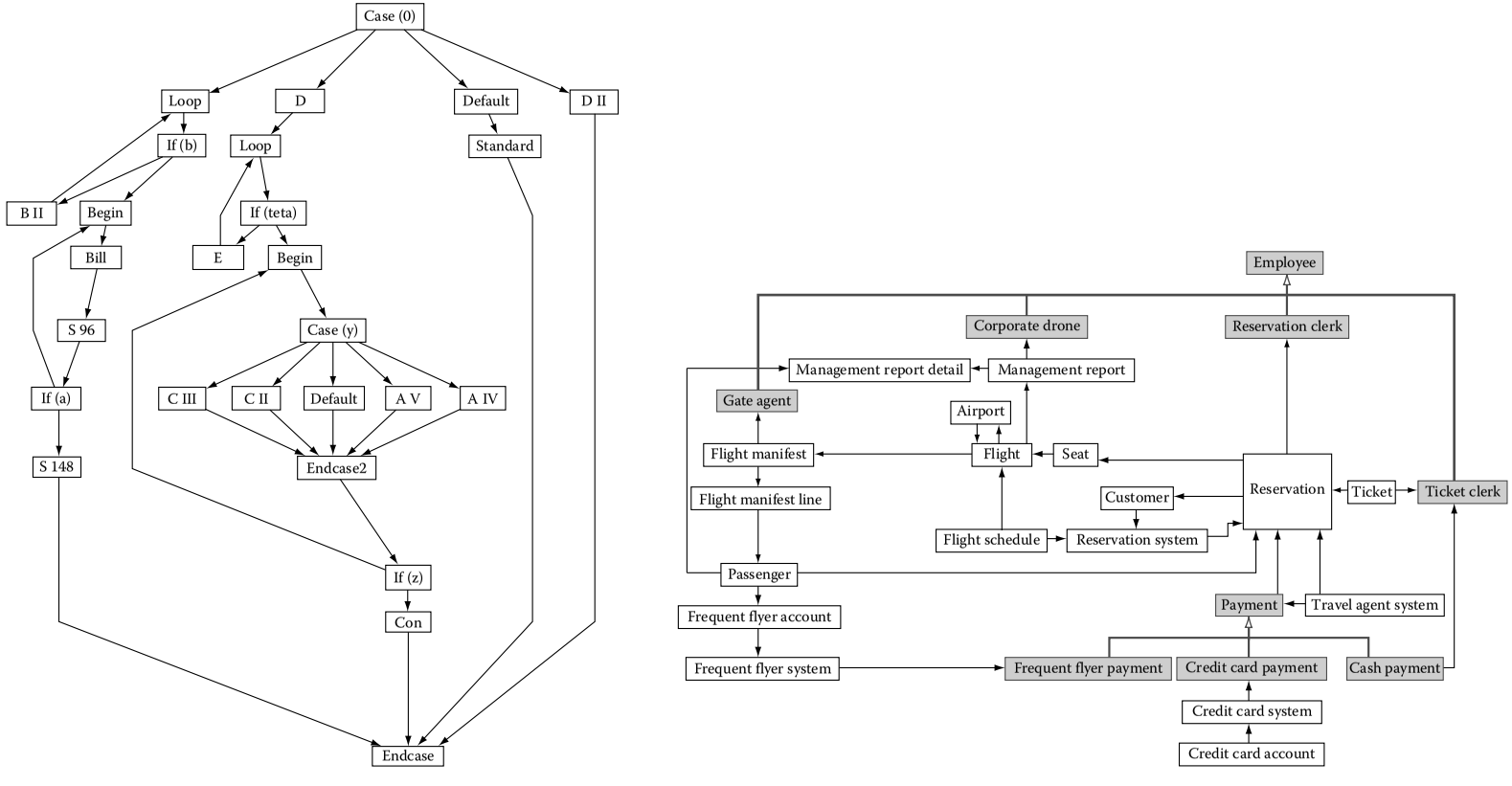

FIGURE 6.1 Two real-world graphs: (a) data flow diagram and (b) UML class diagram.

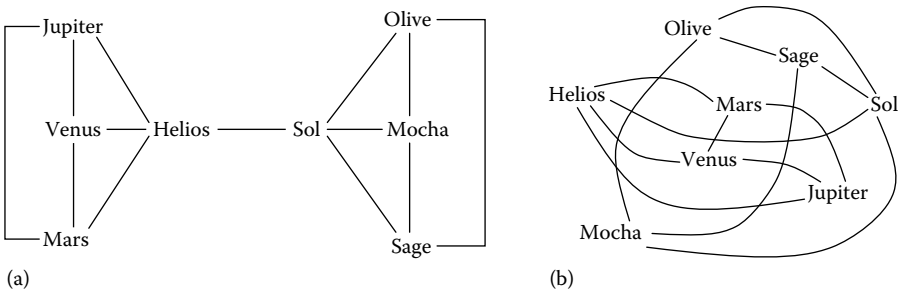

FIGURE 6.2 Two drawings of the same computer network.

is, unordered pairs of vertices. A drawing of G assigns a location (in two or three dimensions) to every vertex of G and a simple curve c_{uv} to every edge (u, v) of G such that the endpoints of c_{uv} are the locations of u and v. Notation and terminology of graph theory is given in [8].

The remainder of this chapter gives an overview of graph drawing (Section 6.2) and details algorithms for obtaining planar drawings (straight-line as well as orthogonal, see Sections 6.3.1 and 6.3.2) and symmetric drawings (Section 6.5). Since not every graph is planar, Section 6.4 describes methods for planarizing a nonplanar graph. A list of research problems is given in Section 6.6, and a review of the defining terms is given in Section 6.7.

6.2 Overview

In this section, we provide a brief overview of the graph drawing problem. Section 6.2.1 outlines the main conventions for drawing graphs, and Section 6.2.2 lists the most common "aesthetic criteria," or optimization goals, of graph drawing. A brief survey of some significant graph drawing algorithms is given in Section 6.2.3.

6.2.1 Drawing Conventions

Drawing conventions for graphs differ from one application area to another. Some of the common conventions are listed below.

- Many graph drawing methods output a grid drawing: the location of each vertex has integer coordinates. Some grid drawings appear in Figure 6.3.
- In a polyline drawing, the curve representing each edge is a polyline, that is, a chain of line segments. Polyline drawings are presented in Figures 6.3b and 6.1a. If each polyline is just a line segment, then the drawing is a straight-line drawing, as in Figure 6.3c.

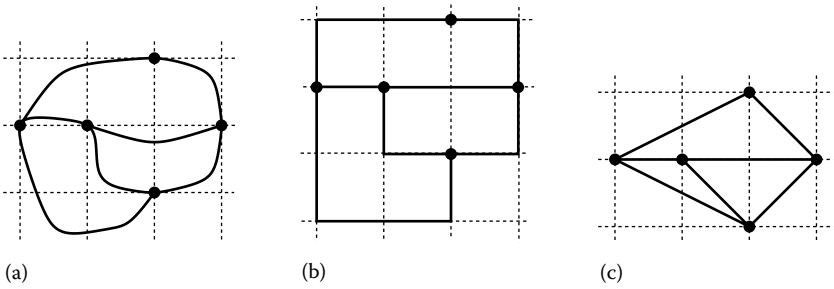

FIGURE 6.3 Examples of drawing conventions: (a) a grid drawing, (b) an orthogonal grid drawing, and (c) a straight-line drawing.

- In an orthogonal drawing, each edge is a polyline composed of straight line segments parallel to one of the coordinate axes. Orthogonal drawings are used in many application areas because horizontal and vertical line segments are easy to follow. Figures 6.3b and 6.1b are orthogonal drawings.

6.2.2 Aesthetic Criteria

The main requirement of a graph drawing method is that the output should be readable; that is, it should be easy to understand, easy to remember, and it should illuminate rather than obscure the application. Of course it is difficult to model readability precisely, since it varies from one application to another, and from one human to another; these variations mean that there are many graph-drawing problems. The problems can be classified roughly according to the specific optimization goals that they try to achieve. These goals are called aesthetic criteria; a list of some such criteria is given below.

- Minimization of the number of edge crossings is an aesthetic criterion, which is important in many application areas. The drawing in Figure 6.2a has no edge crossings, whereas the one in Figure 6.2b has ten.

  A graph that can be drawn in the plane with no edge crossings is called a planar graph. Methods for drawing planar graphs are described in Section 6.3.

  In general, visualization systems must deal with graphs that are not planar. To exploit the theory of planar graphs and methods for drawing planar graphs, it is necessary to planarize nonplanar graphs, that is, to transform them to planar graphs. A planarization technique is described in Section 6.4.
- Bends: In polyline drawings, it is easier to follow edges with fewer bends. Thus, many graph drawing methods aim to minimize, or at least bound, the number of edge bends. In the drawing in Figure 6.3b, there are six edge bends; the maximum number of bends on an edge is two. Algorithms for straight-line drawings (no bends at all) are given in Sections 6.3.1 and 6.5.

  Very few graphs have an orthogonal drawing with no bends, but there are a number of methods that aim to keep the number of bends in orthogonal drawings small. A method for creating two-dimensional planar orthogonal drawings of a planar graph (with a fixed embedding) with a minimum number of bends is described in Section 6.3.2.
- The vertex resolution of a drawing is the minimum distance between a pair of vertices. For a given screen size, we would like to have a drawing with maximum resolution.

  In most cases, the drawing is a grid drawing; this guarantees that the drawing has vertex resolution at least one. To ensure adequate vertex resolution, we try to keep the *size* of the grid drawing bounded. If a two dimensional grid drawing lies within an isothetic rectangle of width w and height h, then the vertex resolution for a unit square screen is at least $\max(1/w, 1/h)$. All the methods described in Section 6.3 give grid drawings with polynomially bounded size.
- Displaying symmetries in a drawing of a graph is an important criterion, as symmetry clearly reveals the hidden structure of the graph. For example, the drawing in Figure 6.2a displays one reflectional (or axial) symmetry. It clearly shows two isomorphic subgraphs with congruent drawings. Symmetric drawings are beautiful and easy to understand.

Maximizing the number of symmetries in a drawing is one of the desired optimization criteria in graph drawing. Formal models and algorithms for symmetric graph drawings are described in Section 6.5.

6.2.3 Drawing Methods

In Sections 6.3 through 6.5, we describe some graph drawing methods in detail. However, there are many additional approaches to graph drawing. Here, we briefly overview some of the more significant methods.

6.2.3.1 Force-Directed Methods

Force-directed methods draw a physical analogy between the layout problem and a system of forces defined on drawings of graphs. For example, vertices may be replaced with bodies that repel each other, and edges have been replaced with Hooke's law springs. In general, these methods have two parts:

The model: This is a "force system" defined by the vertices and edges of the graph. It is a physical model for a graph. The model may be defined in terms of "energy" rather than in terms of a system of forces; the force system is just the derivative of the energy system.

The algorithm: This is a technique for finding an equilibrium state of the force system, that is, a position for each vertex such that the total force on every vertex is zero. This state defines a drawing of the graph. If the model is stated as an energy system then the algorithm is usually stated as a technique for finding a configuration with locally minimal energy.

Force directed algorithms are easy to understand and easy to implement, and thus they have become quite popular [20,25]. Moreover, it is relatively easy to integrate various kinds of constraints. Recently, multilevel methods in combination with spatial data structures such as quad trees have been used to overcome the problems with large graphs. These methods are fairly successful with regular graphs (e.g., meshed-based), highly symmetric graphs, and "tree-like" graphs (see Figure 6.4). They work in both two and three dimensions. Comparisons of the many variations of the basic idea appear in [10,32].

6.2.3.2 Hierarchical Methods

Hierarchical methods are suitable for directed graphs, especially where the graph has very few directed cycles.

Suppose that $G = (V, E)$ is an acyclic directed graph. A layering of G is a partition of V into subsets $L_1, L_2, \ldots, L_h$, such that if $(u, v) \in E$ where $u \in L_i$ and $v \in L_j$, then $i > j$. An acyclic directed graph with a layering is a hierarchical graph.

Hierarchical methods convert a directed graph into a hierarchical graph, and draw the hierarchical graph such that layer L_i lies on the horizontal line $y = i$. A sample drawing is shown in Figure 6.1a. The aims of these methods include the following.

a. Represent the "flow" of the graph from top to bottom of the page. This implies that most arcs should point downward.
b. Ensure that the graph drawing fits the page; that is, it is not too high (not too many layers) and not too wide (not too many vertices on each layer).
c. Ensure that arcs are not too long. This implies that the y extent $|i - j|$ of an arc (u, v) with $u \in L_i$ and $v \in L_j$ should be minimized.
d. Reduce the number of edge crossings.
e. Ensure that arcs are as straight as possible.

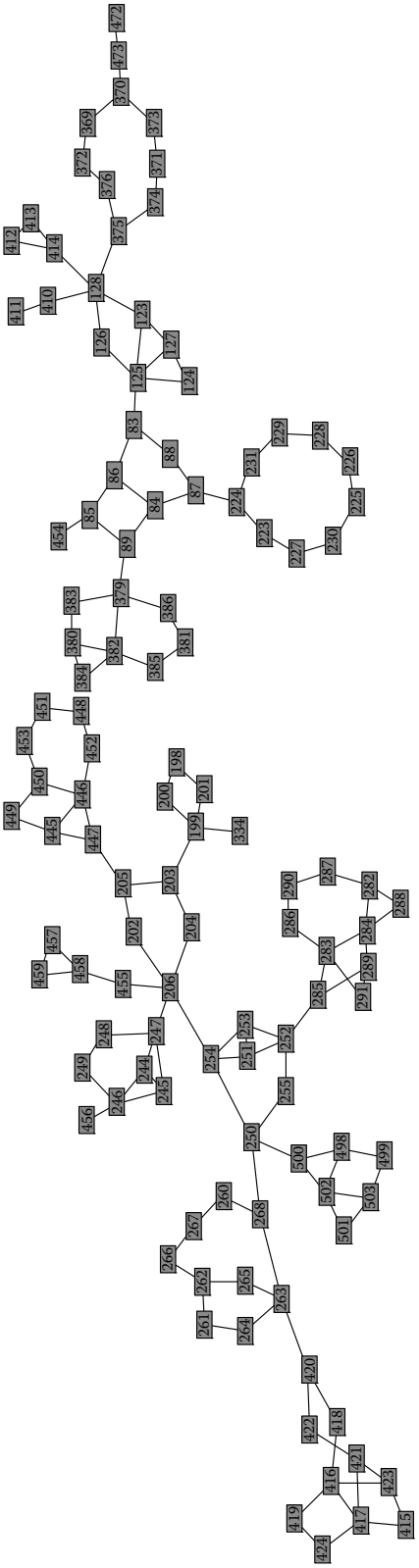

FIGURE 6.4 A force-directed drawing.

There are many hierarchical methods, but the following four steps are common to most.

1. Directed cycles are removed by temporarily reversing some of the arcs. The arcs that are reversed will appear pointing upward in the final drawing, and thus, the number of arcs reversed should be small to achieve (a) above.
2. The set of vertices is partitioned into layers. The partition aims to achieve (b) and (c) above.
3. Vertices within each layer are permuted so that the overall number of crossings is small.
4. Vertices are positioned in each layer so that edges that span more than one layer are as straight as possible.

Each step involves heuristics for NP-hard optimization problems. A detailed description of hierarchical methods appears in [22]. An empirical comparison of various hierarchical drawing methods appears in [7].

6.2.3.3 Tree Drawing Methods

Rooted trees are special hierarchical graphs, and hierarchical methods apply. We can assign vertices at depth k in the tree to layer $h - k$, where h is the maximum depth. The convention that layer i is drawn on the horizontal line $y = i$ helps the viewer to see the hierarchy represented by the tree. However, for trees, there are some simpler approaches; here, we outline one such method.

Note that the edge crossing problem is trivial for trees. The main challenge is to create a drawing with width small enough to fit the page. The Reingold–Tilford algorithm [48] is a simple heuristic method designed to reduce width. Suppose that T is an oriented binary rooted tree. We denote the left subtree by T_L and the right subtree by T_R. Draw subtrees T_L and T_R recursively on separate sheets of paper. Move the drawings of T_L and T_R toward each other as far as possible without the two drawings touching. Now, center the root of T between its children. The Reingold–Tilford method can be extended to nonbinary trees [50,12]. A typical tree drawing based on these ideas is shown in Figure 6.5.

6.2.3.4 3D Drawings

Affordable high quality 3D graphics in every PC has motivated a great deal of research in three-dimensional graph drawing over the last two decades. Three-dimensional graph drawings with a variety of aesthetics and edge representations have been extensively studied by the graph drawing community since early 1990. The proceedings of the annual graph drawing conferences document these developments.

Three-dimensional graph drawing research can be roughly divided into two categories: grid drawing and nongrid drawing. Grid drawings can be further divided into two categories based on edge representations: straight-line drawing and orthogonal drawing. Further, for each drawing

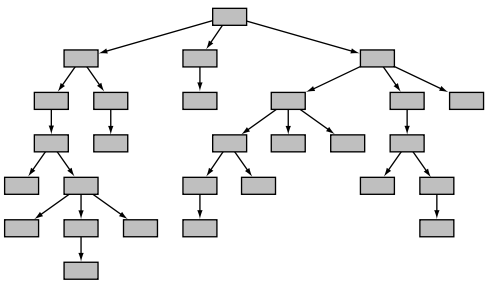

FIGURE 6.5 Tree drawn with an algorithm based on the ideas by Reingold–Tilford.

convention, one can define the following aesthetic (i.e., optimization) criteria: minimize volume for straight-line grid drawings and orthogonal grid drawings, minimize the number of bends in orthogonal grid drawings, or maximize symmetry in nongrid straight-line drawings.

Note that there is a recent survey on three-dimensional graph drawing [19], which covers the state of art survey on three dimensional graph drawing, in particular straight-line grid drawing and orthogonal grid drawing.

6.3 Planar Graph Drawing

A great deal of the long history of graph theory has been devoted to the study of representations of graphs in the plane. The concepts and results developed over centuries by graph theorists have proved very useful in recent visualization and VLSI layout applications. This section describes some techniques used in finding good drawings in the plane.

A representation of a planar graph without edge crossings in the plane is called a plane representation or a planar embedding. The graph is then called a plane graph. A planar embedding divides the plane into regions, which we call faces. The unbounded face is called the outer face. A planar embedding can be represented by a planar map, that is, a cyclic ordered list of the edges bounding each face. There are linear time algorithms for testing planarity of a given graph and, in the positive case, computing a planar map (see, e.g., [9]).

6.3.1 Straight Line Drawings

In this section, we consider the problem of constructing straight-line planar drawings of planar graphs. This problem predates the applications in information visualization and was considered by a number of mathematicians (e.g., [23]). The problem with these early approaches was that they offered poor resolution; that is, they placed vertices exponentially close together. The breakthrough came with the following theorem from [16].

THEOREM 6.1 *[16] Every n-vertex planar graph has a straight-line planar grid drawing which is contained within a rectangle of dimensions $O(n) \times O(n)$.*

The purpose of this section is to outline a constructive proof of Theorem 6.1. Roughly speaking, it proceeds as follows.

1. Dummy edges are added to the graph to ensure that it is a triangulated planar graph, that is, each face is a triangle.
2. The vertices are ordered in a certain way defined below.
3. The vertices are placed one at a time on the grid in the order defined in step 2.

The first step is not difficult. First we find a planar embedding, and then add edges to every face until it becomes a triangle. We present steps 2 and 3 in the following two subsections.

6.3.1.1 Computing the Ordering

Let $G = (V, E)$ be a triangulated plane graph with $n > 3$ vertices, with vertices u, v, and w on the outer face. Suppose that the vertices of V are ordered $v_1, v_2, \ldots, v_n$. Denote the subgraph induced by $v_1, v_2, \ldots, v_\ell$ by G_ℓ, and the outer face of G_ℓ by C_ℓ. The ordering $v_1 = u, v_2 = v, \ldots, v_n = w$ is a canonical ordering if for $3 \leq \ell \leq n-1$, G_ℓ is two-connected, C_ℓ is a cycle containing the edge (v_1, v_2), $v_{\ell+1}$ is in the outer face of G_ℓ, and $v_{\ell+1}$ has at least two neighbors in G_ℓ, and the neighbors are consecutive on the path $C_\ell - (v_1, v_2)$.

LEMMA 6.1 [16] Every triangulated plane graph has a canonical ordering.

PROOF 6.1 The proof proceeds by reverse induction. The outer face of G is the triangle v_1, v_2, v_n. Since G is triangulated, the neighbors of v_n form a cycle, which is the boundary of the outer face of $G - \{v_n\} = G_{n-1}$. Thus, the lemma holds for $\ell = n - 1$.

Suppose that $i \leq n - 2$, and assume that $v_n, v_{n-1}, \ldots, v_{i+2}$ have been chosen so that G_{i+1} and C_{i+1} satisfy the requirements of the lemma. We need to choose a vertex w as v_{i+1} on C_{i+1} so that w is not incident with a chord of C_{i+1}. It is not difficult to show that such a vertex exists. □

6.3.1.2 The Drawing Algorithm

The algorithm for drawing the graph begins by drawing vertices v_1, v_2, and v_3 at locations $(0, 0)$, $(2, 0)$, and $(1, 1)$, respectively. Then, the vertices $v_4, \ldots, v_n$ are placed one at a time, increasing in y coordinate. After v_k has been placed, we have a drawing of the subgraph G_k. Suppose that the outer face C_k of the drawing of G_k consists of the edge (v_1, v_2), and a path $P_k = (v_1 = w_1, w_2, \ldots, w_m = v_2)$; then, the drawing is constructed to satisfy the following three properties:

1. The vertices v_1 and v_2 are located at $(0, 0)$ and $(2k - 4, 0)$, respectively.
2. The path P_k increases monotonically in the x direction; that is, $x(w_1) < x(w_2) < \cdots < x(w_m)$.
3. For each $i \in \{1, 2, \ldots, m - 1\}$, the edge (w_i, w_{i+1}) has slope either $+1$ or -1.

Such a drawing is illustrated in Figure 6.6.

We proceed by induction to show how a drawing with these properties may be computed. The drawing of G_3 satisfies the three properties. Now, suppose that $k \geq 3$, and we have a drawing of G_k that satisfies the three properties. The canonical ordering of the vertices implies that the neighbors of v_{k+1} in G_k occur consecutively on the path P_k; suppose that they are $w_p, w_{p+1}, \ldots, w_q$. Note that the intersection of the line of slope $+1$ through w_p with the line of slope -1 through w_q is at a grid point $\mu(p, q)$ (since the Manhattan distance between two vertices is even). If we placed v_{k+1} at $\mu(p, q)$, then the resulting drawing would have no edge crossings, but perhaps the edge (w_p, v_{k+1}) would overlap with the edge (w_p, w_{p+1}), as in Figure 6.6. This can be repaired by moving all $w_{p+1}, w_{p+2}, \ldots, w_m$ one unit to the right. It is also possible that the edge (v_{k+1}, w_q) overlaps the edge (w_{q-1}, w_q), so we move all the vertices $w_q, w_{q+1}, \ldots, w_m$ a further unit to the right. This ensures that the newly added edges do not overlap with the edges of G_k. Now, we can place v_{k+1} safely at $\mu(p, q)$ as shown in Figure 6.7 (note that $\mu(p, q)$ is still a grid point). The three required properties clearly hold.

But there is a problem with merely moving the vertices on P_k: after the vertices $w_p, w_{p+1}, \ldots$ w_m are moved to the right, the drawing of G_k may have edge crossings. We must use a more comprehensive strategy to repair the drawing of G_k; we must move even more vertices. Roughly speaking, when moving vertex w_i to the right, we will also move the vertices to the right below w_i.

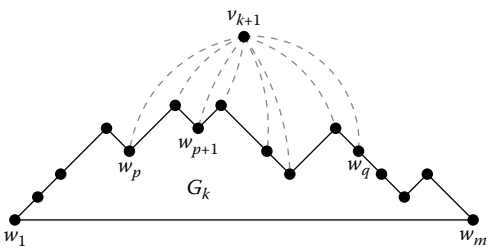

FIGURE 6.6 The subgraph G_k with v_{k+1} placed at $\mu(p, q)$.

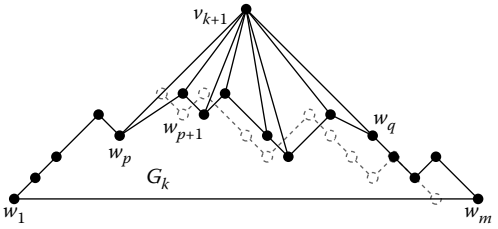

FIGURE 6.7 The subgraph G_k with v_{k+1} placed at $\mu(p, q)$ after moving the vertices $w_{p+1}, \ldots, w_{q-1}$ one unit and the vertices $w_q, \ldots, w_m$ two units to the right.

To this end, we define a sequence of sets $M_k(w_1), M_k(w_2), \ldots, M_k(w_m)$ below. For a given sequence $\alpha(w_1), \alpha(w_2), \ldots, \alpha(w_m)$ of nonnegative integers, the vertices of G_k are moved successively as follows: first, all vertices of $M_k(w_1)$ are moved by $\alpha(w_1)$, then all vertices of $M_k(w_2)$ are moved by $\alpha(w_2)$, and so on.

The sets M_k are defined recursively as follows: $M_3(v_1) = \{v_1, v_2, v_3\}, M_3(v_2) = \{v_2, v_3\}, M_3(v_3) = \{v_3\}$. To compute M_{k+1} from M_k, note that if v_{k+1} is adjacent to $w_p, w_{p+1}, \ldots, w_q$, then P_{k+1} is $(w_1, w_2, \ldots, w_p, v_{k+1}, w_q, \ldots, w_m)$. For each vertex u on this path, we must define $M_{k+1}(u)$. Roughly speaking, we add v_k to $M_k(w_i)$ if w_i is left of v_{k+1}, otherwise $M_k(w_i)$ is not altered. More precisely, for $1 \leq i \leq p$, we define $M_{k+1}(w_i)$ to be $M_k(w_i) \cup \{v_{k+1}\}$, and $M_{k+1}(v_{k+1})$ is $M_k(w_{p+1}) \cup \{v_{k+1}\}$. For $q \leq j \leq M$, we define $M_{k+1}(w_j)$ to be $M_k(w_j)$. It is not difficult to show that the sets satisfy the following three properties:

(a) $w_j \in M_k(w_i)$ if and only if $i \leq j$.
(b) $M_k(w_m) \subset M_k(w_{m-1}) \subset \cdots \subset M_k(w_1)$.
(c) Suppose that $\alpha(w_1), \alpha(w_2), \ldots, \alpha(w_m)$ is a sequence of nonnegative integers and we apply algorithm *Move* to G_k; then the drawing of G_k remains planar.

These properties guarantee planarity.

6.3.1.3 Remarks

The proof of Theorem 6.1 constitutes an algorithm that can be implemented in linear time [15]. The area of the computed drawing is $2n - 4 \times n - 2$; this is asymptotically optimal, but the constants can be reduced to $(n - \Delta_0 - 1) \times (n - \Delta_0 - 1)$, where $0 \leq \Delta_0 \leq \lfloor (n-1)/2 \rfloor$ is the number of cyclic faces of G with respect to its minimum realizer (see [51]).

A sample drawing appears in Figure 6.8, which also shows some of the problems with the method. If the input graph is relatively sparse, then the number of "dummy" edges that must be added can be significant. These dummy edges have considerable influence over the final shape of the drawing but do not occur in the final drawing; the result may appear strange (see Figure 6.8). Using a different ordering, the algorithm also works for triconnected and even for biconnected planar graphs [41,27]. This reduces the number of dummy edges considerably.

Although the drawings produced have relatively good vertex resolution, they are not very readable, because of two reasons: the angles between incident edges can get quite small, and the area is still too big in practice. These two significant problems can be overcome by allowing bends in the edges. Kant [41] modifies the algorithm described above to obtain a "mixed model algorithm;" this algorithm constructs a polyline drawing of a triconnected planar graph such that the size of the minimal angle is at least $\frac{1}{d}\pi$, where d is the maximal degree of a vertex. Figure 6.9 shows the same graph as in Figure 6.8 drawn with a modification of the mixed model algorithm for biconnected planar graphs (see [28]). The grid size of the drawing in Figure 6.8 is 38×19, whereas the size for the mixed model drawing is 10×8.

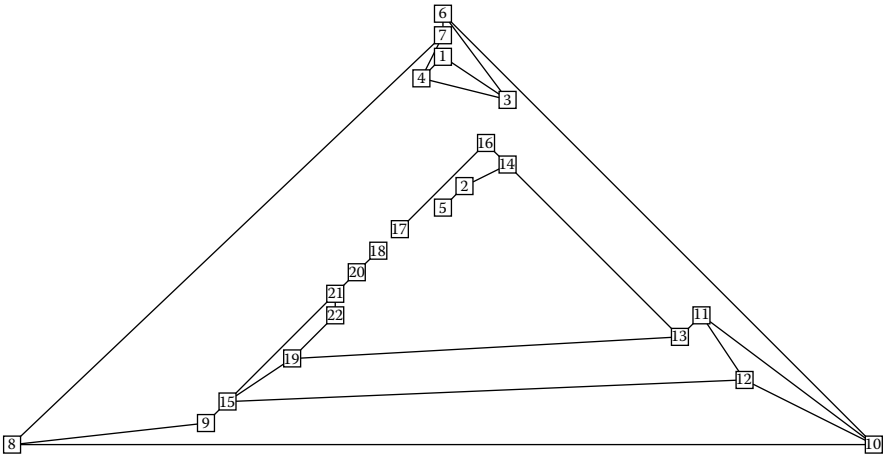

FIGURE 6.8 Sparse graph drawn with straight line edges.

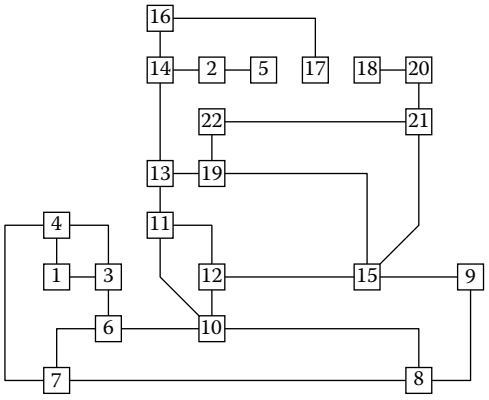

FIGURE 6.9 Sparse graph drawn with the mixed model algorithm.

6.3.2 Orthogonal Drawings

In polyline drawings, one aesthetic is to minimize the total number of bends. Computing an orthogonal drawing of a planar graph with the minimum number of bends is an NP-hard problem. However, for the restricted problem wherein a fixed planar embedding is part of the input, there is a polynomial time algorithm [49]. In this subsection, we will describe the transformation of this restricted bend minimization problem into a network flow problem.

6.3.2.1 Mathematical Preliminaries

Intuitively, the orthogonal representation of an orthogonal planar drawing of a graph describes its "shape;" it does not specify the length of the line-segments but determines the number of bends of each edge. The algorithm described in this section takes as input a planar embedding, represented as a planar map (that is a cyclic ordered list of the edges bounding each face). It produces as output an orthogonal representation with the minimum number of bends. From this, an actual drawing can be constructed via well-known compaction algorithms (see, e.g., [49,45] or Chapter 8).

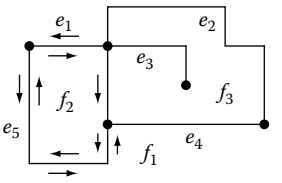

FIGURE 6.10 A graph and its orthogonal representation.

Formally, an orthogonal representation is a function H assigning an ordered list $H(f)$ to each face f. Each element of $H(f)$ has the form (e, s, a), where e is an edge adjacent to f, s is a binary string, and $a \in \{90, 180, 270, 360\}$. If r is an element in $H(f)$, then e_r, s_r, and a_r denote the corresponding entries, that is, $r = (e_r, s_r, a_r)$. The list for an inner face f is ordered so that the edges e_r appear in clockwise order around f; thus, H consists of a planar map of the graph extended by the s- and a-entries. The kth bit in $s_r \in H(f)$ represents the kth bend while traversing the edge in face f in clockwise order: the entry is 0 if the bend produces a 90° angle (on the right hand side) and 1 otherwise. An edge without bend is represented by the zero-string ϵ. The number a_r represents the angle between the last line segment of edge e_r and the first line segment of edge e'_r, where $r' \in H(f)$ follows r in the clockwise order around f. For an outer face f_0, the element $r = (e_r, s_r, a_r)$ is defined similarly, just by replacing "clockwise order" by "counterclockwise order."

An orthogonal representation of Figure 6.10 is

$$H(f_1)=((e_1, \varepsilon, 270), (e_5, 11, 90), (e_4, \varepsilon, 270), (e_2, 1011, 90))$$
$$H(f_2)=((e_1, \varepsilon, 90), (e_6, \varepsilon, 180), (e_5, 00, 90))$$
$$H(f_3)=((e_2, 0010, 90), (e_4, \varepsilon, 90), (e_6, \varepsilon, 90), (e_3, 0, 360), (e_3, 1, 90))$$

Given an orthogonal representation H, the number of bends is obviously given by

$$B(H) = \frac{1}{2} \sum_f \sum_{r \in H(f)} |s_r|,$$

where $|s|$ is the length of string s. Geometrical observations lead to the following lemma that characterizes orthogonal representations.

LEMMA 6.2 The function H is an orthogonal representation of an orthogonal planar drawing of a graph if and only if the properties (P1) to (P4) below are satisfied.

(P1) There is a plane graph with maximal degree four, whose planar map corresponds to the e-entries in $H(f)$.

(P2) Suppose that $r = (e_r, s_r, a_r)$ and $r' = (e_{r'}, s_{r'}, a_{r'})$ are two elements in the orthogonal representation representing the same edge, that is, $e_r = e_{r'}$. Then, reversing the order of the elements in the string s_r and complementing each bit gives the string $s_{r'}$.

(P3) Suppose that ZEROS(s) denotes the number of 0's in string s, ONES(s), the number of 1's in string s, and

$$\text{ROT}(r) = \text{ZEROS}(s_r) - \text{ONES}(s_r) + (2 - a[r]/90).$$

Then, the faces described by H build rectilinear polygons, that is, polygons in which the lines are horizontal and vertical, if and only if

$$\sum_{r \in H(f)} \text{ROT}(r) = \begin{cases} +4 & \text{if } f \text{ is an inner face} \\ -4 & \text{otherwise.} \end{cases}$$

(P4) For all vertices v in the plane graph represented by H, the following holds: the sum of the angles between edges with end vertex v given in the a-entries of the corresponding elements is 360.

6.3.2.2 The Transformation into a Network Flow Problem

Suppose that $G = (V, E)$ is a plane graph and P is a planar map with outer face f_0, that is, for each face f of the embedding, $P(f)$ is a list of the edges on f in clockwise order (respectively counterclockwise order for f_0). Denote the set of faces of G defined by P by F.

We define a network $N(P) = (U, A, b, l, u, c)$ with vertex set U, arc set A, demand/supply b, lower bound l, capacity u, and cost c, as follows. The vertex set $U = U_V \cup U_F$ consists of the vertices in V and a vertex set U_F associated with the set of faces F in G. The vertices $i_v \in U_V$ supply $b(i_v) = 4$ units of flow. The vertices $i_f \in U_F$ have supply respectively demand of

$$b(i_f) = \begin{cases} -2|P(f)| + 4 & \text{if } f \text{ is inner face} \\ -2|P(f)| - 4 & \text{otherwise.} \end{cases}$$

The arc (multi-)set A consists of the two sets A_V and A_F, as follows:

- A_V contains the arcs (i_v, i_f), $v \in V, f \in F$, for all vertices v that are endpoints of edges in $P(f)$ (v is associated with an angle in face f). Formally, we define $E(v,f) := \{e \in P(f) \mid e = (u, v)\}$ and $A_V = \{(i_v, i_f) \mid e \in E(v,f)\}$. Each arc in A_V has lower bound $l(i_v, i_f) = 1$, upper bound $u(i_v, i_f) = 4$, and zero cost $c(i_v, i_f) = 0$. Intuitively, the flow in these arcs is associated with the angle at v in face f.
- A_F contains two arcs (i_f, i_g) and (i_g, i_f) for each edge $e = (f, g)$, where f and g are the faces adjacent to e. Arcs in A_F have lower bound $l(i_f, i_g) = 0$, infinite upper bound, and unit cost $c(i_f, i_g) = c(i_g, i_f) = 1$.

A graph and its network are shown in Figure 6.11. The value of the flow through the network is

$$B(U) = \sum_{v \in V} b(i_v) + \sum_{f \in F} b(i_f) = 4|V| + \sum_{f \neq f_0} (-2|P(f)| + 4) - 2|P(f_0)| - 4 = 4(|V| - |E| + |F| - 2) = 0.$$

The transformation defined above seems to be quite complex, but it can be intuitively interpreted as follows. Each unit of flow represents an angle of 90°. The vertices distribute their 4 flow units among their adjacent faces through the arcs in A_V; this explains the lower and upper bounds of 1 and 4, respectively. Flow in these arcs does not introduce any bends; this explains the zero cost. In general, the flow arriving from vertices $i_v \in U_V$ at the face vertices $i_f \in U_F$ is not equal to the

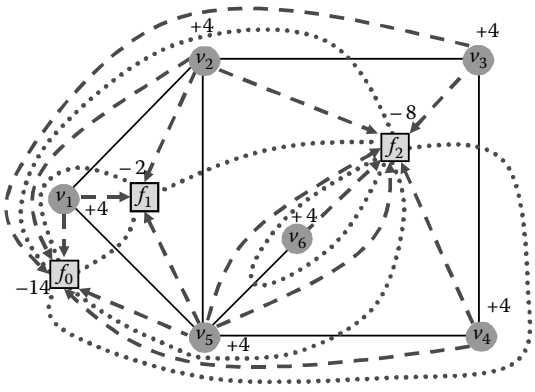

FIGURE 6.11 A graph and its network. The graph consists of vertices $v_1, \ldots, v_6$ and the solid edges. The nodes of the network are labeled with their supply and demand. The arc sets A_V and A_F are drawn dashed and dotted, respectively.

demand or supply of i_f. For adjusting the imbalances, flow will be sent through arcs in A_F. Each flow unit through such arcs will produce one bend. The flow conservation rule on vertices U_V forces the sum of the angles around the vertices to be 360° (see property (P4)). The conservation rule for vertices U_F forces the created faces to be rectilinear polygons (see (P3)). Most importantly, the cost of a given flow in the network $N(P)$ is equal to the number of bends in the constructed orthogonal representation.

LEMMA 6.3 [49] For any orthogonal representation H with planar map P, there exists an integer flow x in $N(P)$ with cost equal to the number of bends $b(H)$.

PROOF 6.3 The flow x in $N(P)$ can be constructed in the following way: With every arc (i_v, i_f) in A_V, we associate an element $r \in R(v,f)$, where $R(v,f) = \{r \in H(f) \mid e_r = (u,v)\}$. The flow value $x(i_v, i_f)$ arises from the a-entries in $r : x(i_v, i_f) = a_r/90$. For every arc (i_f, i_g) in A_F, we associate an element $r \in R(f,g)$, where $R(f,g) = \{r \in H(f) \mid e_r \in P(g)\}$. The flow is defined by the s-entries of r: $x(i_f, i_g) = \text{ZEROS}(s_r)$. The flow $x(i_f, i_g)$ represents the 90° angles within the region f along the edge represented by r separating f and g. The corresponding 270° angles in the face f are represented by their 90° angles in face g through $x(i_g, i_f)$.

We show that the function x is, indeed, a feasible flow with cost equal to the number of bends in H. Obviously, the lower and upper bounds of the arcs in A_V and A_F are satisfied by x. In order to show that the flow conservation rules at vertices in U are satisfied, we first consider the flow balance at a vertex $i_v \in U_V$ (using (P4)):

$$\sum_f x(i_v, i_f) - 0 = \sum_f \sum_{r \in R(v,f)} \frac{a_r}{90} = \sum_f \sum_{\{r \in H(f) \mid e_r = (u,v)\}} \frac{a_r}{90} = \sum_{e_r = (u,v)} \frac{a_r}{90} = 4 = b_{i_v}.$$

For vertices $i_f \in U_F$ (using (P2) and (P3)), we have

$$\begin{aligned}
&\sum_g x(i_f, i_g) - \sum_{v \in V} x(i_v, i_f) - \sum_{h \in F} x(i_h, i_f) \\
&\quad = -\sum_{r \in H(f)} \frac{a_r}{90} - \sum_{r \in H(f)} \text{ONES}(s_r) + \sum_{r \in H(f)} \text{ZEROS}(s_r) \\
&\quad = -2|P(f)| + \sum_{r \in H(f)} \text{ROT}(r) = -2|P(f)| \pm 4 = b_{i_f}.
\end{aligned}$$

The costs of the flow are equal to the number of bends in H:

$$B(H) = \frac{1}{2} \sum_f \sum_{r \in H(f)} |s_r| = \sum_{f \in F} \sum_{g \in F} x(i_f, i_g) = \sum_{a \in A_F} x_a = \sum_{a \in A} c_a x_a.$$

□

LEMMA 6.4 [49] For any integer flow x in $N(P)$, there exists an orthogonal representation H with planar map P. The number of bends in H is equal to the cost of the flow x.

PROOF 6.4 We construct the orthogonal representation H as follows: the edge lists $H(f)$ are given by the planar map $P(f)$. Again, we associate an element $r \in R(v,f)$ with an arc (i_v, i_f) in A_V, and set

$a_r = 90 \cdot x(i_v, i_f)$. For the bit strings s, we associate with each $r \in R(f,g)$ a directed arc $(i_f, i_g) \in A_F$. Then, we determine the element $q \in H(g)$ with $q \neq r$ and $e_q = e_r$. We set

$$s_r = 0^{x(i_f,i_g)}1^{x(i_g,i_f)} \quad \text{and} \quad s_q = 0^{x(i_g,i_f)}1^{x(i_f,i_g)}.$$

We claim that the properties (P1) to (P4) are satisfied by the above defined e, s- and a-entries. Obviously, (P1) and (P2) are satisfied by the construction. The sum of the angles between adjacent edges to v given by the a-entries is

$$\sum_{e_r=(u,v)} a_r = \sum_f x(i_v, i_f) \cdot 90 = 360,$$

thus showing (P4). Finally, it is not hard to show that the faces defined by H build rectilinear polygons (P3), since

$$\sum_{r \in H(f)} \text{ROT}(r) = 2|P(f)| + (-2|P(f)| \pm 4) = 0 \pm 4$$

The number of bends in the orthogonal representation is given by

$$B(H) = \frac{1}{2} \sum_f \sum_{r \in H(f)} |s_r| = \frac{1}{2} \sum_{(i_f,i_g) \in A_F} x(f,g) = \sum_{a \in A} c(a)x(a).$$

□

In a network with integral costs and capacities, there always exists a min-cost flow with integer flow values. Hence we have the following theorem.

THEOREM 6.2 *[49] Let P be a planar map of a planar graph G with maximal degree 4. The minimal number of bends in an orthogonal planar drawing with planar map P is equal to the cost of a min-cost flow in N(P). The orthogonal representation H of any orthogonal planar drawing of G can be constructed from the integer-valued min-cost flow in N(P).*

The algorithm for constructing an orthogonal drawing is as follows. First, construct the network $N(P)$ in time $O(|U| + |A|)$. A min-cost flow on a planar graph with n vertices can be computed in time $O(n^2 \log n)$ (see Chapter 28 for the theory of network algorithms). In this special case, it is even possible to compute the min-cost flow in time $O(n^{\frac{7}{4}}\sqrt{\log n})$ [26]. The orthogonal representation can be constructed easily from the given flow. The length of the line segments can be generated by any compaction algorithm (see [49,45] or Chapter 8).

6.3.2.3 Extensions to Graphs with High Degree

The drawings produced by the bend minimization algorithm look very pleasant (see, e.g., Figures 6.13d and 6.17). Since the algorithm is restricted to planar graphs with maximal degree four, various models have been suggested for extending the network-flow method to planar graphs with high degree vertices. Tamassia et al. [4] have suggested the big nodes model where they decompose the high degree vertices into certain patterns of low degree vertices. Thereby, each edge is assigned to one of the new low degree vertices. This leads to layouts, in which the corresponding high degree nodes are drawn much bigger than the original ones (see Figure 6.12a). This can be overcome by placing vertices of original size into the centers of the big nodes. The quasiorthogonal drawing approach by Klau and Mutzel does not introduce such patterns, but simply integrates the high degree vertices

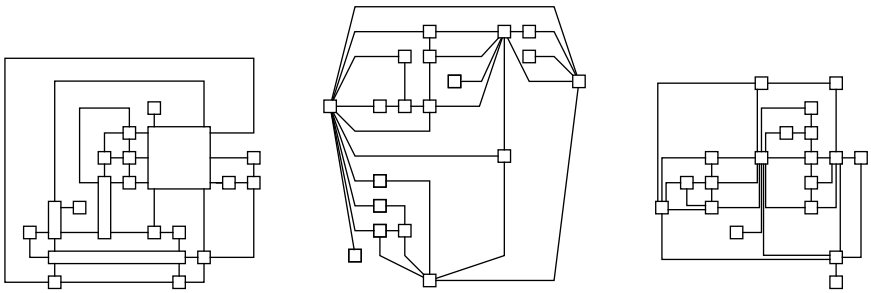

FIGURE 6.12 A graph with high degree vertices drawn with three different approaches: (a) big-nodes model, (b) quasiorthogonal drawing, and (c) Kandinsky layout.

into the network [44]. Also here, the high degree vertices expand their size, which is repaired again by a final replacement step (see Figure 6.12b).

Fößmeier and Kaufmann [24] extend the new network model so that it can deal with a coarse grid for the vertices and a fine grid for the edges (see Figure 6.12c). Unfortunately, the network model needs additional arcs and constraints leading to negative cycles. It is open if the min-cost flow in this Kandinsky model can be found in polynomial time. However, it can be formulated as an integer linear program, which can be solved to optimality for many practical instances using commercial ILP-solvers. There also exist approximation algorithms that theoretically guarantee a two-approximation of the optimal solution. Practical experiments have shown that the solutions found by these algorithms are very close to the optimal solutions [3].

6.4 Planarization

This section describes methods for transforming a nonplanar graph G into a planar graph. This transformation yields a planarized representation G' in which crossings are represented as additional nodes (called dummy nodes) with degree four. We then apply any planar drawing algorithm to G' and obtain a drawing of the original graph by simply replacing the dummy nodes in the layout of G' with edge crossings.

Planarization is a practically successful heuristic for finding a drawing with the minimum number of crossings [30]. The crossing minimization problem is NP-hard. For this problem, Buchheim et al. [11] presented an exact branch-and-cut algorithm that is able to solve small to medium size instances to optimality; cf. [13]. However, the algorithm is very complex in theory and practice so that it will not replace the planarization method in the future.

The planarization approach via planar subgraph was introduced by Batini et al. [5] as a general framework consisting of two steps: The first step computes a large planar subgraph, which serves as a starting point for the following step. The second step reinserts the remaining "nonplanar" edges, while trying to keep the number of crossings small. This is usually done by processing the edges to be inserted successively. Each edge insertion step needs to determine which edges will be crossed; replacing these crossings by dummy nodes with degree four results again in a planar graph that is taken as input for inserting the next edge. Finally, we obtain a planarized representation P of G. Figure 6.13 illustrates the application of this approach by an example, where just a single edge needs to be reinserted.

Finding a planar subgraph of maximum size is an NP-hard optimization problem, which can be solved using branch-and-cut algorithms, as has been shown by Jünger and Mutzel [39]. On the other

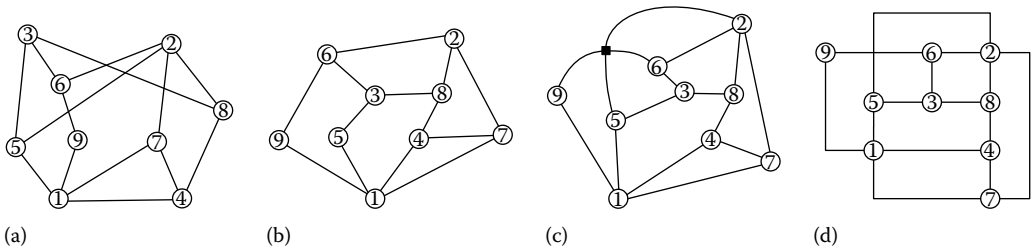

FIGURE 6.13 Example for the application of the planarization method: (a) input graph, (b) planar graph, (c) planarized representation, and (d) final drawing.

hand, we can resort to efficient heuristics, e.g., a maximal planar subgraph (i.e., a planar subgraph such that adding any further edge results in a nonplanar graph) can be computed in linear time as shown by Djidjev [18]. However, this algorithm is very complex and so far no implementation exists; a much easier approach is to use a planarity testing algorithm. The easiest way is to start with a spanning tree of the graph, and add the remaining edges successively; if adding an edge makes the graph nonplanar, we remove it again and proceed with the next edge.

6.4.1 Edge Insertion with Fixed Embedding

The standard technique for reinserting an edge into a planar graph works as follows. Let $e = (v, w)$ be the edge we want to insert into the planar graph P. We first compute a planar embedding Π of P, which fixes the order of edges around each vertex and thus the faces of the drawing. Finding a planar embedding works in linear time; see [9]. Next, we construct the geometric dual graph $\Pi^* = (V^*, E^*)$ of Π. Its vertices V^* are exactly the faces of Π, and it contains an edge (f, f') for each edge of P, where f and f' are the faces of Π separated by e.

Observe that crossing an edge corresponds to traversing an edge in Π^*; if we want to connect vertices v and w, we simply have to find a path in Π^* starting at a face adjacent to v and ending at a face adjacent to w. Therefore, we add two more vertices v^* and w^* (representing v and w, respectively) to Π^* and connect v^* with all faces adjacent to v, and analogously w^* with all faces adjacent to w, resulting in a graph $\Pi^*_{v,w}$. Then, connecting v and w with a minimum number of crossings in a drawing of Π corresponds to finding a shortest path from v^* to w^* in $\Pi^*_{v,w}$, which can be found in linear time by applying breadth-first search. Figure 6.14a shows an example for a geometric dual graph extended in this way, given by the squared vertices (plus vertex 2 and 5) and the dashed edges. The shortest path for inserting edge (2,5) is highlighted in (a), and the corresponding drawing is shown in (b).

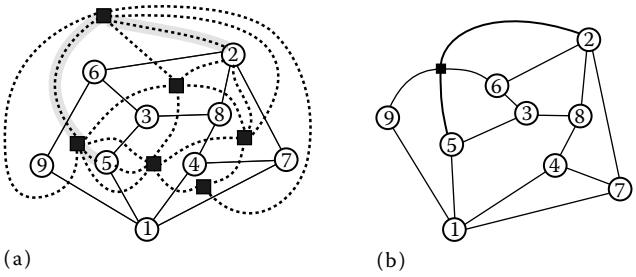

FIGURE 6.14 Inserting an edge $e = (2, 5)$ into a planar graph P with fixed embedding Π: (a) the (extended) geometric dual graph of Π (dashed edges) and (b) substituting the crossing with a dummy node.

6.4.2 Edge Insertion with Variable Embedding

The drawback of edge insertion with fixed planar embedding is that the quality of the solution heavily depends on the choice of the planar embedding. Fixing an unfavorable embedding may result in arbitrarily many unnecessary crossings, which could be avoided by choosing a different embedding. Edge insertion with variable embedding overcomes this weakness by also finding an "optimal" embedding. This requires that we are able to efficiently represent all possible planar embeddings of the graph; the challenging part here is that the possible number of a graph's planar embeddings can be exponential in its size. In the following section, we outline the necessary foundations and an efficient algorithm to solve the problems. For simplicity, we only consider biconnected graphs.

A compact and efficiently computable representation of all planar embeddings of a biconnected graph is given by decomposing the graph into its triconnected components, similar to the decomposition of a graph into its biconnected components. The triconnected components consist of three types of structures: serial, parallel, and triconnected components. All the components of a graph are related in a tree-like fashion. Figure 6.15 gives examples; the shaded regions represent subgraphs that share exactly two vertices (a separation pair) with the rest of the graph. The actual structure of a component—its skeleton—is obtained by shrinking the shaded parts to single, the so-called virtual edges; the part that was shrunk is also called the expansion graph of the virtual edge. The key observation is that we can enumerate all planar embeddings of a graph by permuting the components in parallel structures and mirroring triconnected structures; obviously, we cannot afford explicit enumeration in an algorithm. Typically, triconnected components are represented by a data structure called SPQR-tree introduced by Di Battista and Tamassia [17], which can be constructed in linear time and space; see [29].

The optimal edge insertion algorithm by Gutwenger et al. [31] proceeds as follows. Assume that we want to insert edge (v, w) into the planar and biconnected graph G. First, we compute the SPQR-tree of G and identify the (shortest) path from a component whose skeleton contains v to a component whose skeleton contains w. For each component on this path, we can compute the crossings contributed by that particular component individually. Starting with the component's skeleton, we replace all virtual edges whose expansion graph does not contain v or w (except as an endpoint of the virtual edge) by their expansion graphs; other virtual edges are split creating a representant for v or w, respectively. We observe that only the triconnected structures contribute required crossings (see Figure 6.16): For serial structures, this is trivial, and parallel structures can always be arranged such that the affected parts are neighbored. Triconnected structures can be handled analogously as in the fixed embedding case, since their skeletons themselves are triconnected and thus admit unique embeddings (up to mirroring), and it can be shown that it is permitted to choose arbitrary embeddings for the expansion graphs. Finally, the obtained crossings are linked together in the order given by the computed path in the SPQR-tree. The overall runtime of the algorithm is linear, thus even matching the algorithm for the fixed embedding case.

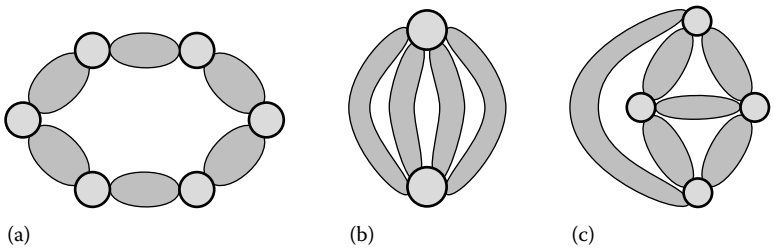

FIGURE 6.15 Triconnected components: (a) serial, (b) parallel, and (c) triconnected structures.

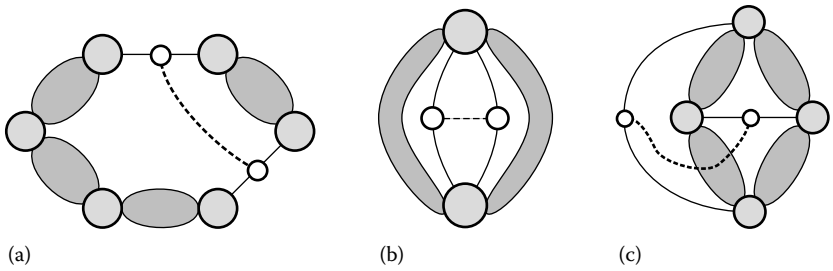

FIGURE 6.16 Edge insertion: serial (a) and parallel (b) components do not contribute any crossings; triconnected structures can be handled as the fixed embedding case.

6.4.3 Remarks

A graph drawn (by the software library OGDF [47]) using the planarization approach and network flow techniques (for drawing the planarized graph) is shown in Figure 6.17.

The question arises whether the insertion of an edge into a planar graph G using the optimal edge insertion algorithm will lead to a crossing minimum drawing. However, Gutwenger et al. [31] have presented a class of almost planar graphs for which optimal edge insertion does not lead to an approximation algorithm with a constant factor. An almost planar graph G is a graph containing an edge $e \in G$ such that $G - e$ is planar. On the other hand, Hliněný and Salazar [33] have shown that optimal edge insertion approximates the optimal number of crossings in almost planar graphs by a factor of Δ, where Δ denotes the maximum node degree in the given graph.

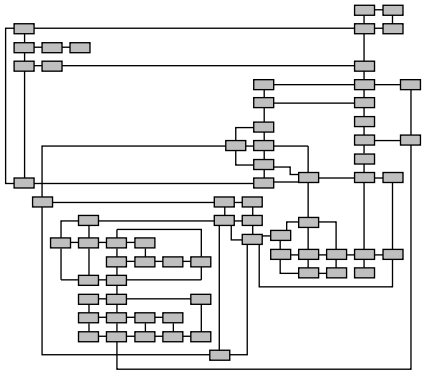

FIGURE 6.17 A graph drawn using the planarization approach and network flow techniques.

6.5 Symmetric Graph Drawing

Symmetry is a much admired property in visualization. Drawings of graphs in graph theory textbooks are often symmetric, because the symmetry clearly reveals the structure of the graph.

The aim of symmetric graph drawing is to take as input a graph G and construct a drawing of G that is as symmetric as possible. In general, constructing a symmetric graph drawing has two steps:

1. Detection of symmetries in an abstract graph
2. Display of the symmetries in a drawing of the graph

Note that the first step is much harder than the second step. As a result, most research in symmetric graph drawing has been devoted to the first step. In this survey, we also focus on symmetry detection algorithms.

We first start with a short review on symmetric graph drawing in two dimensions for a brief comparison.

6.5.1 Two-Dimensional Symmetric Graph Drawing

A symmetry of a two-dimensional figure is an isometry of the plane that fixes the figure. There are two types of two dimensional symmetries, *rotational* symmetry and *reflectional (or axial)* symmetry. Rotational symmetry is a rotation about a *point* and reflectional symmetry is a reflection with respect to an *axis*.

Symmetric graph drawing in two dimensions has been widely investigated by a number of authors for the last two decades. For details, see a recent survey on symmetric graph drawing in two dimensions (see, e.g., [21]). Here, we briefly summarize fundamental results.

The problem of determining whether a given graph can be drawn symmetrically is NP-complete in general [46]; however, exact algorithms are presented based on group-theoretic approaches [1]. Linear-time algorithms for constructing maximally symmetric drawings are available for restricted classes of graphs. A series of linear time algorithms developed for constructing maximally symmetric drawings of planar graphs, including trees, outerplanar graphs, triconnected planar graphs, biconnected planar graphs, one-connected planar graphs, and disconnected planar graphs (see, e.g., [46,36]).

6.5.2 Three-Dimensional Symmetric Graph Drawing

Symmetry in three dimensions is much richer than that in two dimensions. For example, a maximal symmetric drawing of the icosahedron graph in two dimensions shows six symmetries. However, the maximal symmetric drawing of the icosahedron graph in three dimensions shows 120 symmetries.

Recently, symmetric graph drawing in three dimensions has been investigated. The problem of drawing a given graph with the maximum number of symmetries in three dimensions is NP-hard [34]. Nevertheless, fast exact algorithms for general graphs based on group theory are available [1]. Also, there are linear time algorithms for constructing maximally symmetric drawings of restricted classes of graphs in three dimensions. Examples include trees, series parallel digraphs, and planar graphs [34,35,37].

6.5.3 Geometric Automorphisms and Three-Dimensional Symmetry Groups

A symmetry of a drawing D of a graph $G = (V, E)$ induces a permutation of the vertices, which is an automorphism of the graph; this automorphism is displayed by the symmetry. We say that the automorphism is a *geometric* automorphism of a graph G [1]. The symmetry group of a drawing D of a graph G induces a subgroup of the automorphism group of G. The subgroup P is a *geometric automorphism group* if there is a drawing D of G, which displays every element of P as a symmetry of the drawing [1]. The size of a symmetry group is the number of elements of the group. The aim of symmetric graph drawing is to construct a drawing of a given graph with a maximum size geometric automorphism group.

The structure of three-dimensional symmetry groups can be quite complex. A complete list of all possible three dimensional symmetry groups can be found in [2]. However, all are variations on just three types: regular pyramid, regular prism, and Platonic solids, which are described below.

A *regular pyramid* is a pyramid with a regular g-gon as its base. It has only one g-fold rotation axis, called the *vertical rotation* axis, passing through the apex and the center of its base. It has g rotational symmetries, called *vertical rotations*, each of which is a rotation of $2\pi i/g$, $i = 0, 1, \ldots, g-1$. Also, there are g *vertical reflections* in reflection planes, each containing the vertical rotation axis. In total, the regular pyramid has $2g$ symmetries.

A *regular prism* has a regular g-gon as its top and bottom face. There are $g+1$ rotation axes, and they can be divided into two classes. The first one, called the *principal* axis or vertical rotation axis, is a g-fold rotation axis that passes through the centers of the two g-gon faces. The second class, called *secondary* axes or *horizontal rotation* axes, consists of g two-fold rotation axes that lie in a plane perpendicular to the principal axis. The number of rotational symmetries is $2g$. Also, there are g reflection planes, called vertical reflection planes, each containing the principal axis, and another reflection plane perpendicular to the principal axis, called *horizontal reflection*. Further, it has $g-1$ *rotary reflections*, composition of a rotation and a reflection. In total, the regular prism has $4g$ symmetries.

Platonic solids have more axes of symmetry. The tetrahedron has four three-fold rotation axes and three two-fold rotation axes. It has 12 rotational symmetries and in total 24 symmetries. The octahedron has three four-fold rotation axes, four three-fold axes, and six two-fold rotation axes. It has 24 rotational symmetries and a full symmetry group of size 48. The icosahedron has six 5-fold rotation axes, ten 3-fold rotation axes, and fifteen 2-fold rotation axes. It has 60 rotational symmetries and a full symmetry group of size 120. Note that the cube and the octahedron are dual solids, and the dodecahedron and the icosahedron are dual.

Other symmetry groups are derived from those of the regular pyramid, the regular prism, and the Platonic solids. In fact, the variations have *less* symmetries than the three basic types. For example, a variation of regular pyramid is a pyramid with only vertical rotations, without the vertical reflections. For more details of three-dimensional symmetry groups and their relations, see [2].

6.5.4 Algorithm for Drawing Trees with Maximum Symmetries

Here, we sketch an algorithm for finding maximally symmetric drawings of trees in three dimensions [35].

Fortunately, maximally symmetric three-dimensional drawings of trees have only a few kinds of groups. First, note a trivial case: every tree has a planar drawing in the xy-plane; a reflection in this plane is a symmetry of the drawing. This kind of symmetry display, where every vertex is mapped to itself, is *inconsequential*.

LEMMA 6.5 [35] The maximum size three-dimensional symmetry group of a tree T is either inconsequential or one of the following three types:

1. Symmetry group of a regular pyramid
2. Symmetry group of a regular prism
3. Symmetry group of one of the Platonic solids

The symmetry finding algorithm for trees constructs all three possible symmetric configurations (i.e., pyramid, prism, and Platonic solids), and then chooses the configuration that has the maximum number of symmetries.

A vertex is *fixed* by a given three-dimensional symmetry if it is mapped onto itself by that symmetry. Note that the center of a tree T is fixed by every automorphism of T; thus, every symmetry of a drawing of T fixes the location of the center.

We first place the center of the input tree at the apex for the pyramid configuration, at the centroid for the prism and the Platonic solids configuration. Then, we place each subtree attached to the center to form a symmetric configuration.

The symmetry finding algorithm uses an auxiliary tree called the Isomorphism Class Tree. This tree is a fundamental structure describing the isomorphisms between subtrees in the input tree. Suppose that T is a tree rooted at r, and that $T_1, T_2, \ldots, T_k$ are the subtrees formed by deleting r. Then, the ICT tree I_T of T has a root node n_r corresponding to r. Each child of n_r in I_T corresponds to an isomorphism class of the subtrees $T_1, T_2, \ldots, T_k$. Information about each isomorphism class is stored at the node. The subtrees under these children are defined recursively. Using a tree isomorphism algorithm that runs in linear time, we can compute the ICT tree in linear time.

Thus, we can state the symmetry finding algorithm as follows.

1. Find the center of T and root T at the center.
2. Construct the Isomorphism Class Tree (ICT) of T.
3. Find three-dimensional symmetry groups of each type:
   a. Construct a pyramid configuration.
   b. Construct a prism configuration.
   c. Construct Platonic solids configuration.
4. Output the group of the configuration that has the maximum size.

There are some variations for each configuration, based on the number of subtrees fixed by the three dimensional symmetry group.

For the pyramid configuration, we construct a pyramid-type drawing by placing the center of the tree at the apex of a pyramid, some fixed subtrees about the rotation axis, and g isomorphic subtrees in the reflection planes that contain the side edges of the pyramid. The resulting drawing has the same symmetry group of the g-gon based pyramid.

The pyramid configuration can have up to two fixed subtrees, one on each side of the center on the rotation axis. For example, Figure 6.18a shows two fixed isomorphic subtrees, and Figure 6.18b shows two fixed nonisomorphic subtrees. Both drawings display four rotational symmetries and four reflectional symmetries. Note that if we do not fix the two subtrees on the rotation axes, the tree in Figure 6.18a can display only two rotational symmetries in three dimensions.

In order to find the three-dimensional pyramid symmetry group of maximum size, we need to consider each subtree as a possible candidate for a fixed subtree. Further, we need to compute the size of the rotational symmetry group of each subtree, as it affects the size of the rotational symmetry group of each configuration as a whole. For example, if the fixed subtrees in Figure 6.18b were

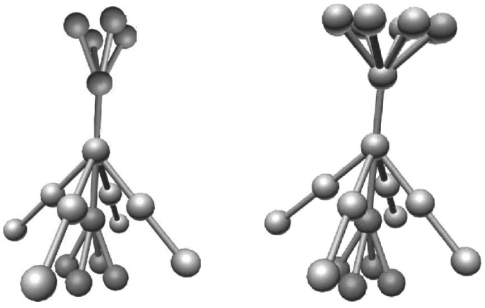

FIGURE 6.18 Pyramid configuration with two fixed subtrees.

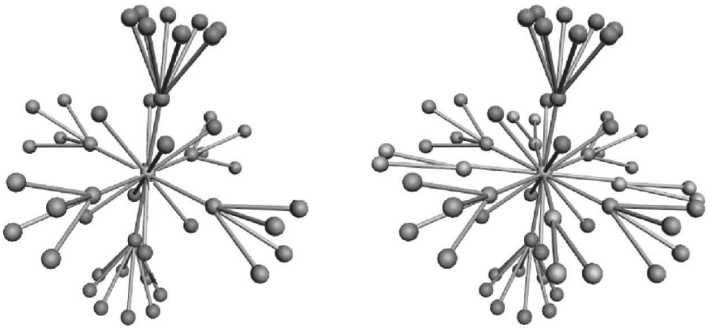

FIGURE 6.19 Prism configuration with fixed subtrees on the *xy*-plane and the *z*-axis.

redrawn to display only two rotational symmetries, then the size of the rotational symmetry group of the whole drawing would be merely two. The ICT can be used to find the fixed subtrees efficiently, and thus to compute a maximum size rotational group for the pyramid configuration in linear time.

The symmetries of a tree drawn in the prism configuration can be more complex than those of the pyramid configuration. The main reason is that a prism has two kinds of rotation axis: the first, called the principal axis, passes through the regular polygons at each end of the prism.

Suppose that the prism has a g-fold rotation about the principal axis. We assume that the principal axis is the z-axis. There are also g two-fold secondary axes in the xy-plane, each at an angle of $\pi i/g$ to the x-axis, for $i = 0, 1, \ldots, g-1$. The prism has a rotational symmetry by an angle of π about each of these axes. The complexity of prism drawings of trees comes from the fact that each of these $g+1$ rotation axes may fix subtrees. For example, see Figure 6.19. The drawing in Figure 6.19a shows two isomorphic fixed subtrees on the z-axis, and four isomorphic subtrees on the secondary axis. The drawing in Figure 6.19b shows two different types of four isomorphic subtrees on the secondary axis, in addition to the two isomorphic fixed subtrees on the z-axis.

Again, using the ICT, we can compute a maximum size rotational group for prism configuration in linear time.

The Platonic solids have many rotation axes. However, the symmetry groups of the Platonic solids are fixed, and we only need to test whether we can construct a three-dimensional drawing of a tree that has the same symmetry group as one of the Platonic solids. Using a method similar to the previous cases, we can test this in linear time.

For example, Figure 6.20 shows the fixed subtree on the four-fold axes of the cube configuration.

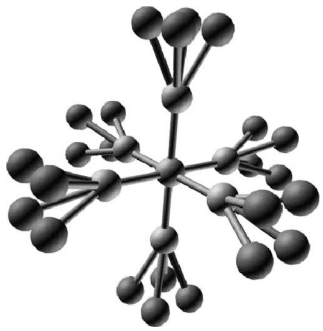

FIGURE 6.20 The cube configuration with fixed subtrees on four-fold axes.

The main thrust of the algorithms for the pyramid, prism, and the Platonic solids is the use of the ICT to determine the fixed subtrees.

Once we compute the maximum size three-dimensional symmetry group, one can use a linear time straight-line drawing algorithm that arranges subtrees in *wedges* and *cones* in [35].

The following theorem summarizes the main results of this section.

THEOREM 6.3 *[35] There is a linear time algorithm that constructs symmetric drawings of trees with maximum number of three-dimensional symmetries.*

6.6 Research Issues

The field of graph drawing is still developing and there are many fundamental questions to be resolved. Here, we list a few related to the above-mentioned drawing methods.

- The grid size for planar straight-line drawings is bounded below; there is an n-vertex plane graph G such that, for any straight-line grid drawing of G, each dimension of the grid is at least $\lfloor \frac{2(n-1)}{3} \rfloor$ even if the other dimension is allowed to be unbounded [16,14]. It is conjectured that this lower bound is almost achievable; more precisely, every planar graph with n vertices has a two-dimensional straight-line drawing on a grid of size $\lceil \frac{2n}{3} \rceil \times \lceil \frac{2n}{3} \rceil$. So far, the best known upper bound is $(n - \Delta_0 - 1) \times (n - \Delta_0 - 1)$, where $0 \leq \Delta_0 \leq \lfloor (n-1)/2 \rfloor$ is the number of cyclic faces of G with respect to its minimum realizer (see [51]).
- The network-flow approach by Tamassia for achieving a bend-minimum orthogonal planar drawing works only for plane graphs with maximum degree 4. So far there is no perfect model or method for extending the network-flow approach to plane graphs with maximum vertex degree > 4. A specific open research question is the problem of finding a polynomial time algorithm for computing bend-minimum drawings in the Kandinsky model for high-degree plane graphs.
- A lot of research has been done in the field of crossing minimization, but there are still open fundamental problems. Though there is a polynomial time algorithm to approximate the crossing number for almost planar graphs with bounded degree, it is unknown if there is such a polynomial time approximation (or even exact) algorithm for almost planar graphs in general. Furthermore, if we consider apex graphs, i.e., graphs containing a vertex whose removal leaves a planar graph, we do not even know if there is a polynomial time approximation algorithm in the case of bounded degree.
- Every triconnected planar graph G can be realized as the one-skeleton of a convex polytope P in $\mathbb{R}^3$ such that all automorphisms of G are induced by isometries of P (see, e.g., [21], Theorem by Mani). Is there a linear time algorithm for constructing a symmetric convex polytope of a triconnected planar graph?

6.7 Further Information

Books on graph drawing appeared in [6,42]. The proceedings of the annual international Graph Drawing Symposia are published in the Lecture Notes in Computer Science Series by Springer; see, for example, [43,38]. There exist various software packages for graph drawing (e.g., [40]). The *Graph Drawing E-Print Archive* (GDEA) is an electronic repository and archive for research materials on the topic of graph drawing (see, http://gdea.informatik.uni-koeln.de).

Defining Terms

Graph drawing: A geometric representation of a graph is a graph drawing. Usually, the representation is in either 2-or 3-dimensional Euclidean space, where a vertex v is represented by a point $p(v)$ and an edge (u, v) is represented by a simple curve whose endpoints are $p(u)$ and $p(v)$.

Edge crossing: Two edges cross in a graph drawing if their geometric representations intersect. The number of crossings in a graph drawing is the number of pairs of edges that cross.

Planar graph: A graph drawing in two-dimensional space is planar if it has no edge crossings. A graph is planar if it has a planar drawing.

Almost planar graph: A graph is almost planar if it contains an edge whose removal leaves a planar graph.

Planarization: Informally, *planarization* is the process of transforming a graph into a planar graph. More precisely, the transformation involves either removing edges (planarization by edge removal) or replacing pairs of nonincident edges by four-stars, as in Figure 6.21 (planarization by adding crossing vertices). In both cases, the aim of planarization is to make the number of operations (either removing edges or replacing pairs of nonincident edges by four-stars) as small as possible.

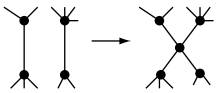

FIGURE 6.21 Planarization by crossing vertices.

Grid drawing: A grid drawing is a graph drawing in which each vertex is represented by a point with integer coordinates.

Straight-line drawing: A straight-line drawing is a graph drawing in which each edge is represented by a straight line segment.

Orthogonal drawing: An orthogonal drawing is a graph drawing in which each edge is represented by a polyline, each segment of which is parallel to a coordinate axis.

References

1. D. Abelson, S. Hong, and D. E. Taylor. Geometric automorphism groups of graphs. *Discrete Appl. Math.*, 155(17):2211–2226, 2007.
2. M. A. Armstrong. *Groups and Symmetry*. Springer-Verlag, New York, 1988.
3. W. Barth, P. Mutzel, and C. Yildiz. A new approximation algorithm for bend minimization in the kandinsky model. In D. Kaufmann and M. und Wagner, editors, *Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pp. 343–354. Springer-Verlag, Berlin, Germany, 2007.
4. C. Batini, E. Nardelli, and R. Tamassia. A layout algorithm for data flow diagrams. *IEEE Trans. Softw. Eng.*, SE-12(4):538–546, 1986.
5. C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity-relationship diagrams. *J. Syst. Softw.*, 4:163–173, 1984.
6. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.
7. G. Di Battista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara. Drawing directed acyclic graphs: An experimental study. In S. North, editor, *Graph Drawing (Proceedings of the GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pp. 76–91. Springer-Verlag, Berlin, Germany, 1997.
8. J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Macmillan, London, U.K., 1976.
9. J. M. Boyer and W. Myrvold. On the cutting edge: Simplified o(n) planarity by edge addition. *J. Graph Algorithms Appl.*, 8(2):241–273, 2004.
10. F. J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In F. J. Brandenburg, editor, *Graph Drawing (Proceedings of the GD '95)*, volume 1027 of *Lecture Notes Computer Science*, pp. 76–87. Springer-Verlag, Berlin, Germany, 1996.
11. C. Buchheim, M. Chimani, D. Ebner, C. Gutwenger, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. A branch-and-cut approach to the crossing number problem. *Discrete Optimization*, 5(2):373–388, 2008.
12. C. Buchheim, M. Jünger, and S. Leipert. Drawing rooted trees in linear time. *Softw. Pract. Exp.*, 36(6):651–665, 2006.

13. M. Chimani, C. Gutwenger, and P. Mutzel. Experiments on exact crossing minimization using column generation. In *Experimental Algorithms (Proceedings of the WEA 2006)*, volume 4007 of *Lecture Notes in Computer Science*, pp. 303–315. Springer-Verlag, Berlin, Germany, 2006.
14. M. Chrobak and S. Nakao. Minimum width grid drawings of planar graphs. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proceedings of the GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pp. 104–110. Springer-Verlag, Berlin, Germany, 1995.
15. M. Chrobak and T. H. Payne. A linear time algorithm for drawing a planar graph on a grid. *Inf. Process. Lett.*, 54:241–246, 1995.
16. H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.
17. G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.
18. H. N. Djidjev. A linear algorithm for the maximal planar subgraph problem. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures, Lecture Notes Computer Science*, pp. 369–380. Springer-Verlag, Berlin, Germany, 1995.
19. V. Dujmović and S. Whitesides. Three dimensional graph drawing. In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*. CRC Press, Boca Raton, FL, to appear.
20. P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
21. P. Eades and S. Hong. Detection and display of symmetries. In R. Tamassia, editor, *Handbook of Graph Drawing and Visualisation*. CRC Press, Boca Raton, FL, to appear.
22. P. Eades and K. Sugiyama. How to draw a directed graph. *J. Inf. Process.*, 13:424–437, 1991.
23. I. Fary. On straight lines representation of planar graphs. *Acta Sci. Math. Szeged*, 11:229–233, 1948.
24. U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F. J. Brandenburg, editor, *Graph Drawing (Proceedings of the GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pp. 254–266. Springer-Verlag, Berlin, Germany, 1996.
25. T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exp.*, 21(11):1129–1164, 1991.
26. A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In S. North, editor, *Graph Drawing (Proceedings of the GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pp. 201–216. Springer-Verlag, Berlin, Germany, 1997.
27. C. Gutwenger and P. Mutzel. Grid embedding of biconnected planar graphs. Technical Report, Max-Planck-Institut Informatik, Saarbrücken, Germany, 1998.
28. C. Gutwenger and P. Mutzel. Planar polyline drawings with good angular resolution. In S. Whitesides, editor, *Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pp. 167–182. Springer-Verlag, Berlin, Germany, 1998.
29. C. Gutwenger and P. Mutzel. A linear time implementation of SPQR trees. In J. Marks, editor, *Graph Drawing (Proceedings of the GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pp. 77–90. Springer-Verlag, Berlin, Germany, 2001.
30. C. Gutwenger and P. Mutzel. An experimental study of crossing minimization heuristics. In B. Liotta, editor, *Graph Drawing (Proceedings of the GD 2003)*, volume 2912 of *Lecture Notes in Computer Science*, pp. 13–24. Springer-Verlag, Berlin, Germany, 2004.
31. C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005.
32. S. Hachul and M. Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In P. Healy and N. S. Nikolov, editors, *Graph Drawing*, volume 3843 of *Lecture Notes in Computer Science*, pp. 235–250. Springer-Verlag, Berlin, Germany, 2006.
33. P. Hliněný and G. Salazar. On the crossing number of almost planar graphs. In M. Kaufmann and D. Wagner, editors, *Graph Drawing (Proceedings of the GD 2006)*, volume 4372 of *Lecture Notes in Computer Science*, pp. 162–173. Springer-Verlag, Berlin, Germany, 2007.

34. S. Hong. Drawing graphs symmetrically in three dimensions. In *Proceedings of the Graph Drawing 2001*, volume 2265 of *Lecture Notes in Computer Science*, pp. 189–204. Springer, Berlin, Germany, 2002.
35. S. Hong and P. Eades. Drawing trees symmetrically in three dimensions. *Algorithmica*, 36(2):153–178, 2003.
36. S. Hong and P. Eades. Drawing planar graphs symmetrically. II. Biconnected planar graphs. *Algorithmica*, 42(2):159–197, 2005.
37. S. Hong, P. Eades, and J. Hillman. Linkless symmetric drawings of series parallel digraphs. *Comput. Geom. Theor. Appl.*, 29(3):191–222, 2004.
38. S.-H. Hong, T. Nishizeki, and W. Quan, editors. *Graph Drawing 2007*, volume 4875 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2008.
39. M. Jünger and P. Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica, Special Issue on Graph Drawing*, 16(1):33–59, 1996.
40. M. Jünger and P. Mutzel. *Graph Drawing Software*. Springer-Verlag, Berlin, Germany, 2003.
41. G. Kant. Drawing planar graphs nicely using the *lmc*-ordering. In *Proceedings of the 33th Annual IEEE Symposium on Foundation of Computer Science*, pp. 101–110, Pittsburgh, PA, 1992.
42. M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science Tutorial*. Springer-Verlag, Berlin, Germany, 2001.
43. M. Kaufmann and D. Wagner, editors. *Graph Drawing 2006*, volume 4372 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2007.
44. G. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report, Max-Planck-Institut Informatik, Saarbrücken, Germany, 1998.
45. G. W. Klau, K. Klein, and P. Mutzel. An experimental comparison of orthogonal compaction algorithms. In J. Marks, editor, *Graph Drawing (Proceedings of the GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pp. 37–51. Springer-Verlag, Berlin, Germany, 2001.
46. J. Manning. Geometric symmetry in graphs. PhD thesis, Purdue University, West Lafayette, IN, 1990.
47. OGDF–Open Graph Drawing Framework, 2008. Open source software project, available via http://www.ogdf.net.
48. E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Trans. Softw. Eng.*, SE-7(2):223–228, 1981.
49. R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
50. J. Q. Walker II. A node-positioning algorithm for general trees. *Softw. Pract. Exp.*, 20(7):685–705, 1990.
51. H. Zhang and X. He. Compact visibility representation and straight-line grid embedding of plane graphs. In F. Dehne, J.-R. Sack, and M. Smid, editors, *Algorithms and Data Structures (WADS 2003)*, volume 2748 of *Lecture Notes in Computer Science*, pp. 493–504. Springer-Verlag, Berlin, Germany, 2003.