Technical Foundations*

Michael Jünger[1] and Petra Mutzel[2]

[1] University of Cologne, Department of Computer Science, Pohligstraße 1, D-50969 Köln, Germany
[2] Vienna University of Technology, Institute of Computer Graphics and Algorithms, Favoritenstraße 9–11, A-1040 Wien, Austria

1 Introduction

Graph drawing software relies on a variety of mathematical results, mainly in *graph theory*, *topology*, and *geometry*, as well as computer science techniques, mainly in the areas *algorithms and data structures*, *software engineering*, and *user interfaces*. Many of the core techniques used in automatic graph drawing come from the intersection of mathematics and computer science in combinatorial and continuous optimization.

Even though automatic graph drawing is a relatively young scientific field, a few generic approaches have emerged in the graph drawing community. They allow a classification of layout methods so that most software packages implement variations of such approaches.

The purpose of this chapter is to lay the foundations for all subsequent chapters so that they can be read independently from each other while referring back to the common material presented here. This chapter has been written based on the requirements and the contributions of all authors in this book. This chapter is *not* an introduction to automatic graph drawing, because it is neither complete nor balanced. In order to avoid repetitions, we only explain subjects that are basic or used in at least two subsequent chapters. The following chapters contain a lot of additional material. For introductions into the field of automatic graph drawing we recommend the books "Graph Drawing" by Di Battista, Eades, Tamassia, and Tollis [23] and "Drawing Graphs" edited by Kaufmann and Wagner [52]. Nevertheless, this book is self-contained in the sense that after this chapter has been read, every subsequent chapter can be read without referring to external sources.

* We gratefully acknowledge the many contributions of the authors of the subsequent chapters. In particular, Gabriele Barbagallo, Andrea Carmignani, Giuseppe Di Battista, Walter Didimo, Carsten Gutwenger, Sebastian Leipert, Maurizio Patrignani, and Maurizio Pizzonia have contributed text fragments and figures that were very helpful to us. In addition, the constructive remarks on earlier drafts by David Auber, Vladimir Batagelj, Ulrik Brandes, Christoph Buchheim, Tim Dwyer, Michael Kaufmann, Gunnar W. Klau, Stephen C. North, Merijam Percan, Georg Sander, and Ioannis G. Tollis were very helpful for corrections and improvements.

Section 2 contains basic notions and notations from graph theory concerning graphs and their representations including undirected and directed graphs, layered graphs, and hierarchical and clustered graphs. It closes with some remarks on the storage of graphs in computer memory. Section 3 discusses concepts of graph planarity and graph embeddings including planar graphs, upward planarity, and cluster planar graphs. Section 4 introduces generic layout styles: tree-, layered-, planarization-, orthogonal-, and force-directed-layout.

2 Graphs and Their Representation

2.1 Undirected Graphs

A *graph* $G = (V, E, \lambda)$ consists of a finite set $V = V(G)$ of *vertices* or *nodes*, a finite set $E = E(G)$ of *edges*, and a function λ that maps each edge to a subset $V' \subseteq V$ with $|V'| \in \{1, 2\}$. An edge e for which $|\lambda(e)| = 1$ is called a *loop* and if for two edges $e_1, e_2 \in E$ we have $\lambda(e_1) = \lambda(e_2)$ we say that e_1 and e_2 are multi-edges. Figure 1 shows a graph with a loop and a pair of multi-edges.

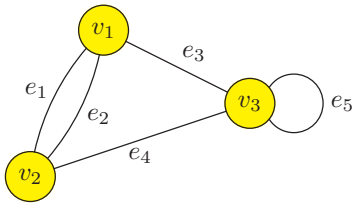


Fig. 1. A graph with a loop and two multi-edges.

A graph with no loops and no multi-edges is characterized by a finite set V of vertices and a finite set $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ of edges and called a *simple graph*. In the sequel, we deal mostly (unless stated otherwise) with simple graphs, but non-simple graphs are important in automatic graph drawing and for ease of notation, we use the simplified $G = (V, E)$ notation with the understanding that multi-edges and loops are distinguishable elements of the multi-set E. E.g., for the non-simple graph in Figure 1, the notation $G(V, E, \lambda) = (\{v_1, v_2, v_3\}, \{e_1, e_2, e_3, e_4, e_5\}, \lambda(e_1) = \{v_1, v_2\}, \lambda(e_2) = \{v_1, v_2\}, \lambda(e_3) = \{v_1, v_3\}, \lambda(e_4) = \{v_2, v_3\}, \lambda(e_5) = \{v_3, v_3\})$ becomes simplified to $G(V, E) = (\{v_1, v_2, v_3\}, \{e_1, e_2, e_3, e_4, e_5\}) = (\{v_1, v_2, v_3\}, \{\{v_1, v_2\}, \{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_3\}\})$.

For an edge $e = \{u, v\}$, the vertices u and v are the *end-vertices* of e, and e is *incident* to u and v. An edge $\{u, v\} \in E$ *connects* the vertices u and v. Two vertices $u, v \in V$ are *adjacent* if $\{u, v\} \in E$. By $\text{star}(v) = \{e \in E \mid v \in e\}$ we denote the set of edges incident to a vertex $v \in V$ and $\text{adj}(v) = \{u \in V \mid$

$\{u, v\} \in E\}$ is the set of vertices adjacent to a vertex $v \in V$. By $\deg(v) = |\operatorname{star}(v)| + |\operatorname{loop}(v)|$, where $\operatorname{loop}(v)$ is the set of edges of the form $\{v, v\}$, we denote the *degree* of a vertex $v \in V$, $\operatorname{mindeg}(G) = \min\{\deg(v) \mid v \in V\}$ is the *minimum degree* and $\operatorname{maxdeg}(G) = \max\{\deg(v) \mid v \in V\}$ is the *maximum degree* of G. E.g., in Figure 1, $\operatorname{star}(v_1) = \{e_1, e_2, e_3\}$, $\operatorname{star}(v_3) = \{e_3, e_4, e_5\}$, whereas $\operatorname{adj}(v_1) = \{v_2, v_3\}$ and $\operatorname{adj}(v_3) = \{v_1, v_2, v_3\}$. The degrees of these two vertices are $\deg(v_1) = 3$ and $\deg(v_3) = 4$, the minimal degree of the graph is $\operatorname{mindeg}(G) = 3$ and the maximum degree is $\operatorname{maxdeg}(G) = 4$. A vertex v with $\deg(v) = 0$ is called an *isolated vertex*.

For $W \subseteq V$ let $E[W] = \{\{u, v\} \in E \mid u, v \in W\}$ and for $F \subseteq E$ let $V[F] = \{v \in V \mid v \in e \text{ for some } e \in F\}$. A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ or *contained* in G if $V' \subseteq V$ and $E' \subseteq E$. For a vertex set $W \subseteq V$ we call $G[W] = (W, E[W])$ a *vertex-induced subgraph* of G and for an edge set $F \subseteq E$ we call $G[F] = (V[F], F)$ an *edge-induced subgraph* of G.

A *walk* W of *length* k in a graph G is an alternating sequence of vertices and edges $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$, *beginning* and *ending* with the vertices v_0 and v_k, respectively, and $e_i = \{v_{i-1}, v_i\}$ for $i = 1, 2, \ldots, k$. This walk *connects* v_0 and v_k and may also be denoted by $W = (v_0, v_1, \ldots, v_k)$, when the edges are evident by context. A walk W is called a *path* if all vertices are distinct, and it is called a *trail* if all edges are distinct. The *distance* of two vertices u and v in $G = (V, E)$, denoted by $\operatorname{dist}(u, v)$, is the number of edges in a shortest path connecting u and v in G, and the *diameter* of G is defined by $\operatorname{diam}(G) = \max\{\operatorname{dist}(u, v) \mid u, v \in V\}$. A walk is called a *cycle* if all vertices are distinct except for $v_0 = v_k$ and $k \geq 2$. A graph that does not have any cycles is called a *forest*.

A graph G is *connected* if every pair of vertices is connected by a path, otherwise it is called *disconnected*. A *component* of G is a maximal connected subgraph of G. Thus a disconnected graph has at least two components. A connected forest G is called a *tree*.

A graph $G = (V, E)$ is *k-connected* if at least k vertices must be removed from V in order to make the resulting vertex-induced subgraph disconnected. By $\kappa(G) = \max\{k \mid G \text{ is } k\text{-connected}\}$ we denote the *(vertex-)connectivity* of G.

Of special interest in automatic graph drawing are 1, 2, and 3-connected graphs, also called *connected*, *biconnected*, and *triconnected* graphs, respectively. A vertex whose removal disconnects the graph is called a *cut-vertex*, i.e., a graph is biconnected if it has no cut-vertex. The maximal biconnected components of a graph G are called the *blocks* of G. The blocks intersect in cut-vertices, see Figure 2 for an illustration.

Two vertices whose removal disconnects a biconnected graph are called a *separating vertex pair*, i.e., a graph is triconnected if it has no separating vertex pair.

An edge whose removal disconnects the graph is called a *bridge*. The graph in Figure 2 contains exactly one bridge.

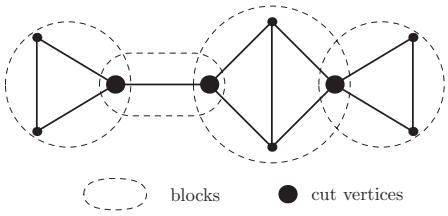


Fig. 2. The cut-vertices and the blocks of a graph.

A graph $G = (V, E)$ is *k-edge-connected* if at least k edges must be removed from E in order to make the resulting edge-induced subgraph disconnected. By $\lambda(G) = \max\{k \mid G \text{ is } k\text{-edge-connected}\}$ we denote the *edge-connectivity* of G.

A *vertex-k-coloring* of a loop-less graph $G = (V, E)$ is an assignment $c : V \to \{1, 2, \ldots, k\}$ such that $c(u) \neq c(v)$ whenever $\{u, v\} \in E$. By $\chi(G) = \min\{k \mid G \text{ has a vertex-}k\text{-coloring}\}$ we denote the *chromatic number* of G. Graphs G with $\chi(G) \leq 2$ are called *bipartite graphs*. Their vertex set can be partitioned into two subsets (corresponding to the two color classes, one of them possibly empty) such that all edges connect vertices of different subsets of the bipartition. A graph is bipartite if and only if it does not contain a cycle of odd length.

2.2 Directed Graphs

A *directed graph* or *digraph* $G = (V, E)$ consists of a finite set $V = V(G)$ of *vertices* and a finite multi-set $E \subseteq V \times V = \{(u, v) \mid u, v \in V\}$ of (*directed*) *edges* or *arcs* that are ordered pairs of vertices. Ignoring for every edge the order of its vertices, we get an undirected graph that is called the *underlying graph* of G. Thus, concepts like subgraph, walk, path, trail, cycle, forest, component, or tree, naturally carry over to directed graphs. In addition, for a *directed walk* we require that the involved edges are ordered pairs (v_{i-1}, v_i) and so we get *directed trails*, *directed paths*, and *directed cycles*. If a digraph $G = (V, E)$ is *simple*, i.e., contains no loops or multi-arcs, it determines a relation $R \subseteq V \times V$ defined by $uRv \iff (u, v) \in E$.

A digraph G is *strongly connected* if each pair of vertices $u, v \in V$ is connected by a directed path from u to v. An *acyclic digraph (directed acyclic graph: dag)* is a digraph with no directed cycle or loop. If $e = (u, v)$ then e is an *outgoing* or *leaving* edge of u and an *incoming* or *entering* edge of v. By $\text{instar}(v) = \{(u, v) \in E \mid u \in V\}$ we denote the set of incoming edges of a vertex $v \in V$ and by $\text{outstar}(v) = \{(v, u) \in E \mid u \in V\}$ we denote the set of outgoing edges of a vertex $v \in V$. Accordingly, we define $\text{inadj}(v) = \{u \in V \mid (u, v) \in E\}$ and $\text{outadj}(v) = \{u \in V \mid (v, u) \in E\}$. Then $\text{indeg}(v) = |\text{instar}(v)|$ is the *in-degree* and $\text{outdeg}(v) = |\text{outstar}(v)|$ is the *out-degree* of a vertex $v \in V$.

A *source* is a vertex with no incoming edges and a *sink* is a vertex with no outgoing edges. An acyclic digraph G with exactly one source is called a *single source directed graph* digraph. If, in addition, its underlying graph is connected and has no loop and no (undirected) cycle, the graph is called a *rooted tree* whose *root* is the only vertex $v = \text{root}(T) \in V$ with $\text{indeg}(v) = 0$ and whose *leaves* are vertices $v \in V$ with $\text{outdeg}(v) = 0$. The *depth* $\text{depth}(v)$ of a vertex v in a rooted tree $T = (V, E)$ is the length of the (unique) directed path from the root of T to v. All vertices of depth k constitute *tree level* k. Furthermore, for each $v \in V$ that is not a leaf, the vertices in $\text{outadj}(v)$ are called *children* of v, and for each $v \in V$ other than the root, the vertex in $\text{inadj}(v)$ is called *parent* of v. Children of the same parent are called *siblings*. An acyclic digraph with exactly one sink is called a *single sink* digraph. An acyclic digraph with exactly one source s and exactly one sink t and an edge (s, t) is called an *st-digraph*.

A *topological numbering* of G is an assignment of numbers $\text{topnumber}(v)$ to the vertices v of G such that for every edge (u, v) of G the number assigned to v is greater than the one assigned to u (i.e., $\text{topnumber}(v) > \text{topnumber}(u)$). A *topological sorting* of G is a topological numbering of G such that every vertex is assigned a distinct integer between 1 and $|V|$. It is easy to see that G admits a topological numbering or sorting if and only if G is acyclic.

2.3 Representation of Graphs

There are several ways to represent an (undirected or directed) graph. Here, we restrict our attention to the classical representation in graph drawing. A graph $G = (V, E)$ is generally visualized by a *drawing* in 2 or 3-dimensional space with the vertices drawn as points or boxes of a pre-specified width and height, and the edges drawn as closed Jordan curves, connecting their incident vertices. Layouts in which the coordinates of the vertex representations are restricted to integer values are called *grid layouts*.

In this book, we describe software for generating graph drawings in 2 or 3-dimensional space. In this chapter, however, we restrict our attention mainly to drawings in 2-dimensional space.

2.4 Layered Graphs

Let $\langle L_1, L_2, \ldots, L_h \rangle$ denote an ordered set of elements, called the *layers* of the graph. A *layered graph* $H = (G, \Lambda)$ consists of a (directed or undirected) graph $G = (V, E)$ and a function $\Lambda : V \to \langle L_1, L_2, \ldots, L_h \rangle$ assigning each vertex $v \in V$ to exactly one layer L_i, $i \in \{1, \ldots, h\}$.

Layered graphs are often represented "top-to-bottom" as follows: For $i = 1, \ldots, h$, the vertices belonging to layer L_i are drawn on a horizontal line with y-coordinate y_i, satisfying the condition $y_1 > y_2 > \cdots > y_h$. A popular

alternative is a "left-to-right" representation with vertical lines at x_i and $x_1 < x_2 < \cdots < x_h$.

In graph drawing, layered graphs occur in the context of directed acyclic graphs. For these graphs, a layering is generated based on a topological numbering, i.e., the function Λ satisfies $\Lambda(v) > \Lambda(u)$ for each edge $(u, v) \in E$. In this context it is common to draw all the edges as curves monotonically decreasing in vertical direction in a top-to-bottom representation and monotonically increasing in horizontal direction in a left-to-right representation.

2.5 Clustered Graphs

Clustered graphs are graphs with recursive clustering structures over the vertices. A *clustered graph* $C = (G, T)$ consists of an undirected graph $G = (V, E)$ and a rooted tree T such that the leaves of T are exactly the vertices of G. Each vertex ν of T represents a *cluster* $V(\nu)$ of the vertices of G that are the leaves of the subtree rooted at ν. The root of T is called *root cluster*. The tree T is called the *inclusion tree* of C because it describes an inclusion relation between clusters. The graph G is called the *underlying graph* of C. The tree $T(\nu)$ represents the subtree of T rooted at the vertex ν, and $G(\nu)$ denotes the subgraph of G induced by the cluster associated with vertex ν. We define $C(\nu) = (G(\nu), T(\nu))$ to be the *sub-clustered graph* associated with vertex ν. An edge $\{v, w\} \in E$ with $v \in V(G(\nu))$ and $w \in V \setminus V(G(\nu))$ is said to be *incident* to cluster ν. Figure 3 shows a drawing of a clustered graph $C = (G, T)$ and the corresponding tree T.

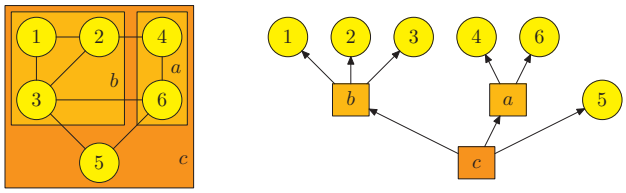


Fig. 3. A drawing of a clustered graph and its defining tree.

In a *drawing of a clustered graph* $C = (G, T)$, the graph G is drawn with points and curves as usual. For each vertex ν of T, the cluster is drawn as a simple closed region R (i.e., a region without holes) that contains the drawing of $V(G(\nu))$, such that the following three conditions hold.

(i) The regions for all sub-clusters of ν are completely contained in the interior of R.
(ii) The regions for all other clusters are completely contained in the exterior of R.
(iii) If there is an edge e between two vertices of $V(\nu)$ then the drawing of e is completely contained in R.

2.6 Compound Graphs

Compound graphs have been introduced for representing graphs with both inclusion and adjacency relationships [63]. A *compound graph* $C = (G, T)$ is defined as an (undirected or directed) graph $G = (V, E_G)$ and a rooted tree $T = (V, E_T)$ that share the same vertex set V. There is a one to one correspondence between the structure of the tree and the set of inclusions between the vertices, namely, a vertex u is in direct inclusion relation to v if and only if u is a child of v in the tree. If the end-vertices u and v of all edges $\{u, v\} \in E_G$ belong to different root-leaf paths in T, C is called a *simple compound graph*. In a simple compound graph, a pair of vertices (u, v) cannot be in an adjacency and in an inclusion relation at the same time. Figure 4 shows an example of a simple compound graph.

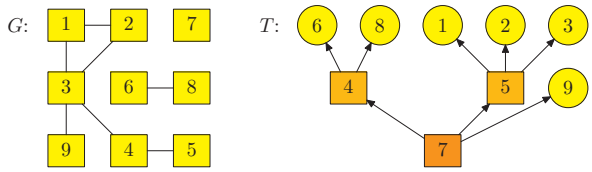


Fig. 4. A compound graph defined by a graph and a tree.

Edges connecting vertices of different tree levels are called *inter-level edges*. If a compound graph does not contain inter-level edges, we call it a *nested graph*.

A further restriction allowing only edges between the leaves of the tree leads to an alternative definition of clustered graphs (see Section 2.5).

In a *drawing of a compound graph* $C = (G, T)$, the vertices of the graph G are drawn as closed regions so that a vertex u is included in the region representing the vertex parent(u) in T, and the edges in E_G are drawn as curves connecting the regions associated with its end-vertices. Figure 5 shows a drawing of the compound graph defined in Figure 4.

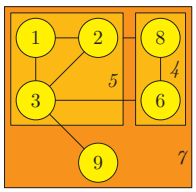
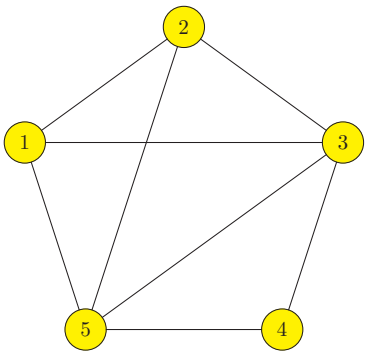


Fig. 5. A drawing of the compound graph defined in Figure 4.

2.7 Storage of Graphs and Digraphs

Common data structures for storing graphs and digraphs $G = (V, E)$ are adjacency matrices and adjacency lists. An *adjacency matrix* for an undirected graph is a $|V|$ by $|V|$ matrix $A(G) = (a_{uv})_{u,v \in V}$ where a_{uv} is the number of edges connecting the vertices u and v, i.e., $a_{uv} \in \{0, 1\}$ for simple graphs. For a digraph, a_{uv} is the number of edges leaving u and entering v, again, $a_{uv} \in \{0, 1\}$ for simple digraphs. Adjacency matrices are, due to their storage requirement of $|V|^2$, independently of $|E|$, unattractive for *sparse graphs*, i.e., graphs in which E contains only a small subset of all possible edges.

In automatic graph drawing, usually sparse graphs are considered. In fact, we often have $|E| \leq c|V|$ for some constant c. Therefore a different data structure is usually more appropriate. Most common are *star-* and/or *adjacency-lists* that give (in-/out-)star(v) and/or (in-/out-)adj(v) for each $v \in V$ as linear, linked, or doubly linked lists, depending on the application. For digraphs, the in- and out-lists may be merged and equipped with an in-/out-flag. In addition to the adjacency lists, a list of the edges with their end-vertices is useful in most cases. In Figure 6, we illustrate the different storage formats.

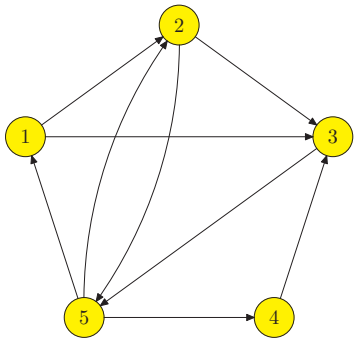


Adjacency matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Adjacency lists

$\text{adj}(1) = \langle 2, 3, 5 \rangle$
$\text{adj}(2) = \langle 3, 5, 1 \rangle$
$\text{adj}(3) = \langle 4, 5, 1, 2 \rangle$
$\text{adj}(4) = \langle 5, 3 \rangle$
$\text{adj}(5) = \langle 1, 2, 3, 4 \rangle$



Adjacency matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Adjacency lists

$\text{inadj}(1) = \langle 5 \rangle$ $\quad \text{outadj}(1) = \langle 2, 3 \rangle$
$\text{inadj}(2) = \langle 1, 5 \rangle$ $\quad \text{outadj}(2) = \langle 5, 3 \rangle$
$\text{inadj}(3) = \langle 2, 1, 4 \rangle$ $\quad \text{outadj}(3) = \langle 5 \rangle$
$\text{inadj}(4) = \langle 5 \rangle$ $\quad \text{outadj}(4) = \langle 3 \rangle$
$\text{inadj}(5) = \langle 2, 3 \rangle$ $\quad \text{outadj}(5) = \langle 1, 2, 4 \rangle$

Fig. 6. Storage of graphs and digraphs.

3 Graph Planarity and Embeddings

This section deals with drawings and embeddings of a graph onto the plane unless otherwise stated. A drawing of a graph G on the plane yields an *embedding* Π of G, i.e., a clockwise ordering of the incident edges for every vertex with respect to the drawing. Such an embedding can be conveniently stored by ordering the star- or adjacency-lists of Section 2.7 accordingly.

3.1 Planar Graphs

A graph $G = (V, E)$ is called *planar* if it can be drawn in the plane such that no two edges cross each other except at common endpoints. An intersection of two edges in a drawing other than at their endpoints is called a *crossing*. A *planar* or *combinatorial embedding* Π of a planar graph G is an embedding with respect to a planar drawing. A graph with a given fixed planar embedding Π is also called a *plane graph*. Given a drawing of a plane graph G, a *face* of G is a topologically connected region in the drawing bounded by the (Jordan curves corresponding to the) edges of G. A face of a plane graph is uniquely described by its boundary edges. The degree $\deg(f)$ of a face f is defined as the number of its boundary edges, where each boundary edge with both sides on the boundary of f is counted twice. The faces of a plane graph are already described by the planar embedding. Two faces are *adjacent* if their boundaries share an edge. The one unbounded face of a plane graph is called the *outer face* or *exterior face*. All other faces are called *interior faces*. An equivalent definition of a planar embedding is an ordered list of the boundary edges for each face, clockwise for interior faces and counter-clockwise for the exterior face.

A famous result of Euler [34] for polytopes relates the number of vertices, edges, and faces in any planar embedding of a connected planar graph:

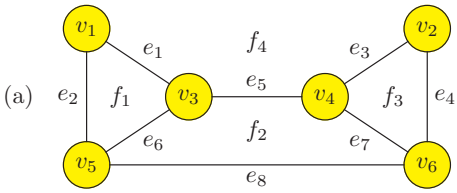
Theorem 1 (Euler's Formula [34]). *Let Π be a planar embedding of a connected planar graph $G = (V, E)$ and let F be the set of faces in Π. Then $|V| - |E| + |F| = 2$.*

From Euler's formula, an upper bound on the number of edges of a planar graph with a given number of vertices is easily derived:

Theorem 2. *For any simple planar graph $G = (V, E)$ with at least 3 vertices we have $|E| \leq 3|V| - 6$.*

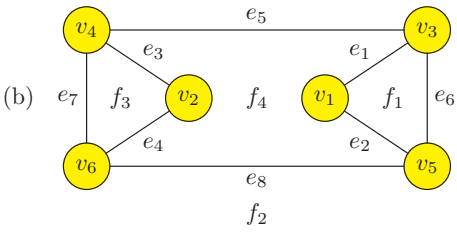
The bound is attained for *triangulated* planar graphs, i.e., planar graphs in which every face is a triangle.

While, in general, the number of different planar embeddings of a planar graph is exponential in $|V|$, a triconnected planar graph has only two different planar embeddings, which are mirror-images of each other, see Figures 7 and 8 for illustrations.

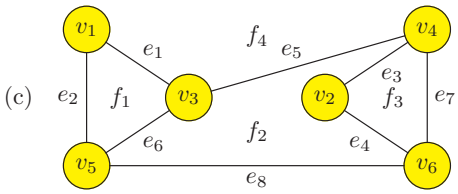


planar embedding

v_1: $\langle v_3, v_5\rangle$	f_1: $\langle e_1, e_6, e_2\rangle$
v_2: $\langle v_4, v_6\rangle$	f_2: $\langle e_6, e_5, e_7, e_8\rangle$
v_3: $\langle v_1, v_4, v_5\rangle$	f_3: $\langle e_3, e_4, e_7\rangle$
v_4: $\langle v_2, v_6, v_3\rangle$	f_4: $\langle e_1, e_2, e_8, e_4, e_3, e_5\rangle$
v_5: $\langle v_1, v_3, v_6\rangle$	
v_6: $\langle v_2, v_5, v_4\rangle$	



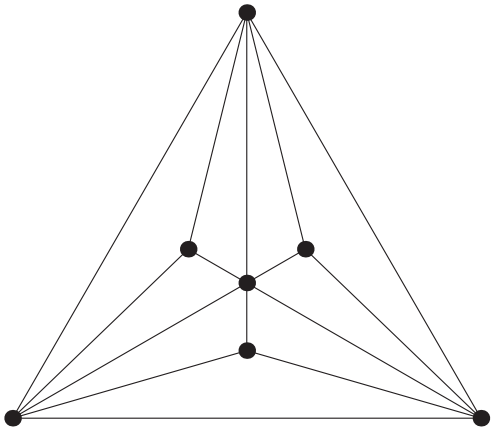
same planar embedding as (a), i.e., combinatorially equivalent to (a)



different planar embedding

v_1: $\langle v_3, v_5\rangle$	f_1: $\langle e_1, e_6, e_2\rangle$
v_2: $\langle v_4, v_6\rangle$	f_2: $\langle e_3, e_4, e_8, e_6, e_5\rangle$
v_3: $\langle v_1, v_4, v_5\rangle$	f_3: $\langle e_4, e_3, e_7\rangle$
v_4: $\langle v_2, v_3, v_6\rangle$	f_4: $\langle e_1, e_2, e_8, e_7, e_5\rangle$
v_5: $\langle v_1, v_3, v_6\rangle$	
v_6: $\langle v_2, v_4, v_5\rangle$	

Fig. 7. Combinatorial embeddings.

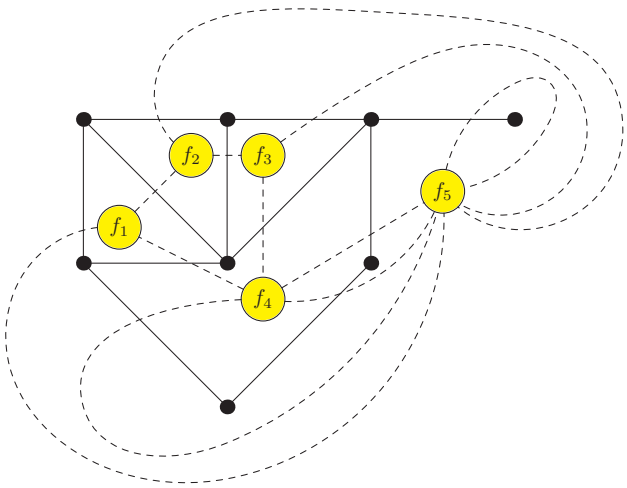


$|V| = 7, \quad |E| = 3|V| - 6 = 15$

Fig. 8. A triconnected graph has a unique embedding up to reflection.

Given a planar embedding $\Pi(G)$ of a planar graph $G = (V, E)$ with face set F, the *dual graph* $G' = (V', E')$ is constructed as follows: $V' = F$ and E' contains an edge $\{f_i, f_j\}$ for each $e \in E$ such that e is on the boundary of

both f_i and f_j. (f_i and f_j may be identical.) By definition, the degree $\deg(f)$ of a face f of G agrees with the degree of f as a vertex of G'. The dual graph of a planar graph is in general a non-simple graph (i.e., contains loops and multi-edges) as can be seen in the example shown in Figure 9. Loops in G' correspond to bridges in G.



f_5 has a loop and there are three copies of $\{f_4, f_5\}$

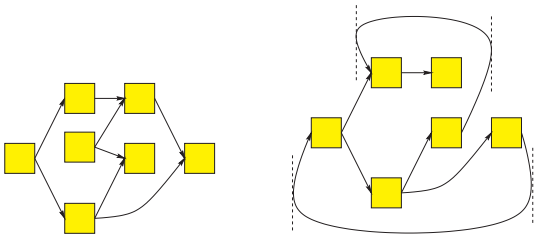
Fig. 9. A planar graph and its non-simple dual graph.

Planarity of a graph $G = (V, E)$ can be tested in $O(|V|)$ time by, e.g., the algorithm of Hopcroft and Tarjan [45], or an approach of Lempel *et al.* [55] using the special data structure PQ-tree introduced by Booth and Lueker [7]. For a planar graph G, an embedding Π of G can be determined in linear time by, e.g., the algorithms of Chiba *et al.* [17] or Mehlhorn and Mutzel [57].

3.2 Upward Planarity

Let G be an embedded digraph. A vertex v of G is called *bimodal* if the circular list of the edges incident to v can be partitioned into two (possibly empty) linear lists of edges, one consisting of the incoming edges and the other consisting of the outgoing edges. An embedding is called a *bimodal embedding* if every vertex is bimodal. A planar graph is said to be *bimodal* if it admits a planar bimodal embedding.

A drawing of G such that all the edges are curves monotonically increasing in a given direction is known as an *upward drawing*. Figure 10(a) shows an example of an upward planar drawing in the left-to-right direction. An *upward embedding* $\mathcal{U}$ is a representation of G that consists of the clockwise



(a) (b)

Fig. 10. (a) An upward planar drawing; (b) A quasi-upward planar drawing with 4 bends; the dashed lines indicate the tangents to the bend-points.

orderings of the incoming edges for every vertex with respect to an upward drawing. Necessary conditions for the existence of an upward planar drawing of an embedded graph G_Π are the acyclicity and the bimodality of G_Π itself [5]. However, these conditions are not sufficient. A polynomial time algorithm to test the existence of upward planar drawings of a planar embedded digraph is given in [5]. The problem is $\mathcal{NP}$-complete in a variable embedding setting [41].

The quasi-upward drawing convention extends the upward drawing convention [3]. A *quasi-upward drawing* of a digraph in the left-to-right direction is such that the vertical line through each vertex v "locally" splits the incoming edges from the outgoing edges of v. The term locally is used to identify a sufficiently small connected region properly containing v.

A *bend* of a quasi-upward drawing in the left-to-right direction is a point on an edge such that the vertical line through this point is tangent to the edge. Intuitively, a bend is a point in which an edge inverts its left-to-right direction. In Figure 10(b) a quasi-upward planar drawing with four bends is shown. In [3] it is proven that a quasi-upward planar drawing of a digraph exists if and only if the digraph is planar bimodal, and a polynomial time algorithm for computing quasi-upward planar drawings with the minimum number of bends of an embedded bimodal digraph is described.

A directed acyclic graph $G = (V, E)$ is called *upward planar* if it has an upward drawing without edge crossings. An *upward planar embedding* is an upward embedding with respect to an upward planar drawing.

Upward planarity testing of directed acyclic graphs is $\mathcal{NP}$-complete as has been shown by Garg and Tamassia [41]. Acyclic digraphs with a single source can be tested for upward planarity:

Theorem 3 (Bertolazzi *et al.* [6]). *There is an $O(|V|)$ time algorithm using SPQR-trees to test whether a single source acyclic digraph $G = (V, E)$ is upward planar, and if so, it outputs an upward planar embedding.*

3.3 Cluster Planarity

In Section 2.5 we have already discussed clustered graphs and their representation. Here, we adapt the concept of planarity to clustered graphs.

In a drawing of a clustered graph, an edge e and a region R have an *edge-region crossing* if the drawing of e crosses the boundary of R more than once. A drawing of a clustered graph is *c-planar* if there are no edge crossings or edge-region crossings. A graph that admits a c-planar drawing is called *c-planar*. Notice that the planarity of the underlying graph does not imply the existence of a c-planar drawing of a clustered graph, see Figure 11.

A c-planar drawing of C induces a c-planar embedding. A *c-planar embedding* of C fixes the planar embedding of the underlying graph G and contains the circular ordering of all edges crossing the boundary of each non-trivial cluster region.

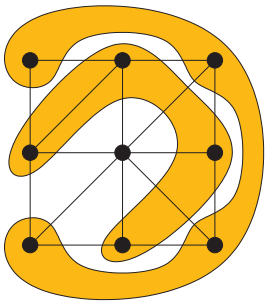


Fig. 11. A clustered graph that is not c-planar.

Unfortunately, so far no polynomial time algorithm is known for c-planarity testing. However, c-planarity can be tested for a subclass of clustered graphs. A clustered graph $C = (G, T)$ is called *c-connected* if each cluster induces a connected subgraph of G.

Theorem 4 (Feng *et al.* [35], Dahlhaus [21]). *The c-planarity of a c-connected clustered graph $C = (G, T)$ can be tested in linear time.*

The algorithm of [35] is based on the following theorem that gives a necessary and sufficient condition for c-planarity of c-connected graphs.

Theorem 5 (Feng *et al.* [35]). *A c-connected clustered graph $C = (G, T)$ is c-planar if and only if G is planar, and there exists a planar drawing of G such that for each vertex ν of T, all vertices and edges of $G \setminus G(\nu)$ are in the outer face of the drawing of $G(\nu)$.*

4 Graph Drawing Methods

4.1 Tree Layout

In automatic graph drawing, the notion "Tree Layout" refers to drawing rooted trees. Before a general (undirected) tree can be processed, a root must be chosen and all edges must be directed away from the root. Forests are processed by drawing each connected component separately. Since for a tree $T = (V, E)$, we have $|E| = |V| - 1$, running times are given as functions of $|V|$. For a typical tree layout, see Figure 12.

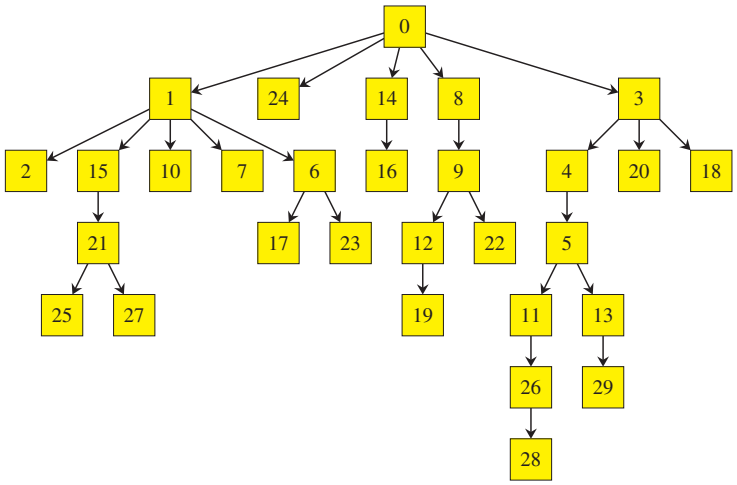


Fig. 12. A typical tree layout.

We start by treating an important special case, namely *binary trees*, in which each vertex has 0, 1, or 2 children. If a vertex has two children, one is a *left* and the other is a *right* child, and if it has one child, this is either a left or a right child. We describe a beautiful $O(|V|)$ algorithm of Reingold and Tilford [60] whose grid layout satisfies the following æsthetic criteria:

(A1) All vertices $v \in V$ of the same depth are drawn on a straight horizontal line whose y-coordinate is $-\text{depth}(v)$.
(A2) A left child is placed to the left (smaller x-coordinate), a right child is placed to the right (bigger x-coordinate) of its parent.
(A3) If a parent has two children, it is centered above its children.
(A4) A tree and its mirror image are drawn identically up to reflection.
(A5) Isomorphic subtrees are drawn identically up to translation.

For a vertex $v \in V$ let $T_l(v)$ be the subtree rooted at the left child of v, if it exists, else $T_l(v) = (\emptyset, \emptyset)$, and let $T_r(v)$ be the subtree rooted at the right child of v, if it exists, else $T_r(v) = (\emptyset, \emptyset)$. Basic traversal orders for the vertices of a binary tree $T = (V, E)$ are defined by the following recursive functions:

```
preorder(T)
{
  if V(T) ≠ ∅  {
    v = root(T);
    visit(v);
    preorder(T_l(v));
    preorder(T_r(v));
  }
}
```

```
inorder(T)
{
  if V(T) ≠ ∅  {
    v = root(T);
    inorder(T_l(v));
    visit(v);
    inorder(T_r(v));
  }
}
```

```
postorder(T)
{
  if V(T) ≠ ∅  {
    v = root(T);
    postorder(T_l(v));
    postorder(T_r(v));
    visit(v);
  }
}
```

The algorithm of Reingold and Tilford follows the divide and conquer principle implemented in the form of a postorder traversal of $T = (V, E)$. Namely, for each $v \in V$ the algorithm computes layouts for $T_l(v)$ and $T_r(v)$ up to horizontal translation, and when v is visited, the two drawings are horizontally shifted together up to a minimum vertex separation of 2 or 3 grid points so that v can be centered above the roots of $T_l(v)$ and $T_r(v)$ at an integer grid coordinate. If one of $T_l(v)$ and $T_r(v)$ is empty, v is placed one grid unit to the left or right, respectively, of the root of the other.

For a tree T, the left contour of T consists of the vertices with minimum x-coordinate at each depth in the tree, and the right contour is defined analogously. The contour information can be stored and updated efficiently with additional flags at each vertex. Whenever the subtrees $T_l(v)$ and $T_r(v)$ are shifted together, the amount of shift is calculated by traversing the right contour of $T_l(v)$ and the left contour of $T_r(v)$ in order to determine the first point of contact. See Figure 13 for an illustration.

Linear running time is achieved by delaying all shifts of subtrees to a second phase. Rather than performing the shifts directly, the necessary displacements for subtrees are stored at their respective roots. In a second phase, this information is processed in a preorder traversal of T in order to compute the final x-coordinates of all vertices.

While the Reingold-Tilford algorithm is very efficient and delivers æsthetically pleasing drawings, the width of the drawing may be arbitrarily far from the minimum width subject to the five æsthetic criteria, more precisely, there is a family of binary trees $T = (V, E)$ that can be drawn on a grid of width 2, yet the Reingold-Tilford algorithm delivers a drawing of width $(|V| + 2)/3$. It has been shown by Supowit and Reingold [65] that achieving minimum grid width is $\mathcal{NP}$-hard, and, even worse, that, unless $\mathcal{P} = \mathcal{NP}$, there is no polynomial time algorithm for achieving a width that is smaller than $\frac{25}{24}$ times the minimum width. On the other hand, if continuous coordinates (rather

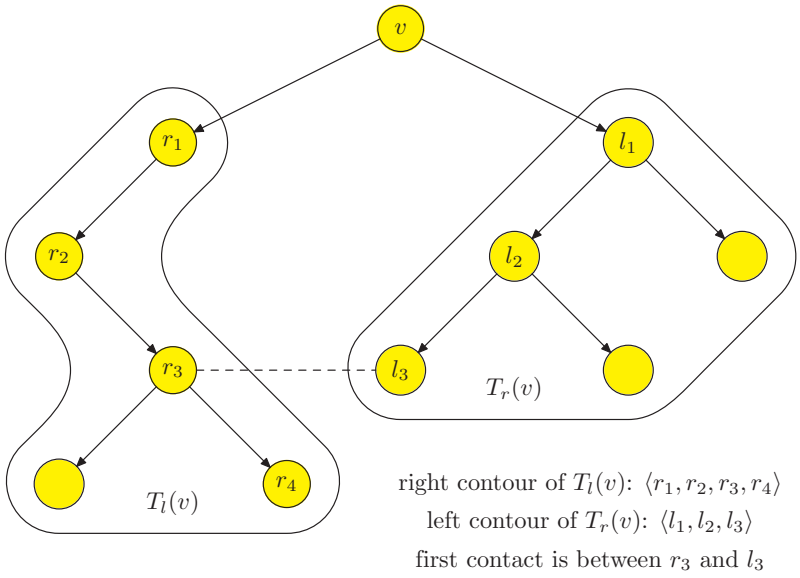


Fig. 13. The Reingold-Tilford algorithm for binary tree drawing.

than integral grid coordinates) are allowed, Supowit and Reingold [65] have shown that minimum width drawings can be found with the help of a linear programming technique in polynomial time.

In practice, the Reingold-Tilford algorithm is generally accepted as the method of choice for drawing binary trees. Walker [70] has generalized this algorithm to general rooted trees, and Buchheim *et al.* [15] have improved the running time of Walker's algorithm to $O(|V|)$ time. An easy modification involving basic trigonometry allows for planar tree drawings in which the vertices are placed on concentric circles around the root rather than on parallel lines, see Eades [29].

4.2 Layered Layout

In the previous section, we have already seen a special case of layered layout: all vertices of a rooted tree $T = (V, E)$ were drawn on parallel horizontal lines, i.e., assigned to parallel horizontal layers, and all directed tree edges $(u, v) \in E$ were drawn as straight lines between two consecutive layers such that the y-coordinate of u was one grid unit bigger than the y-coordinate of v. Conceptually, this drawing style easily extends to general acyclic digraphs $G = (V, E)$ by stipulating that, again, all vertices are drawn on parallel horizontal lines such that for each directed edge $(u, v) \in E$ the y-coordinate of u is bigger than the y-coordinate of v. This idea has been worked out in an automatic graph drawing cornerstone paper by Sugiyama *et al.* [62]. There-

fore, this drawing style is commonly referred to as *Sugiyama-style layout*, see Figure 14 for a typical Sugiyama-style layout on five layers.

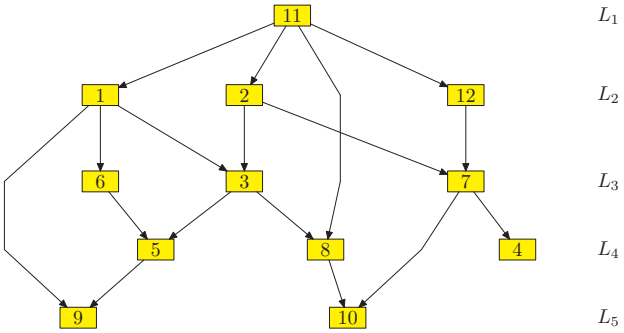


Fig. 14. A typical Sugiyama-style layout.

The great popularity of Sugiyama-style layout is boosted by the fact that any directed or undirected graph can be converted into an acyclic digraph either by reversing directed edges in the case of a digraph or assigning appropriate directions to the edges in the case of an undirected graph. We will first describe layered layout for acyclic digraphs and then discuss the conversion of any graph into an acyclic digraph.

Sugiyama-style layout for an acyclic digraph $G = (V, E)$ decomposes into three phases:

(S1) **Layer Assignment:** Compute a layering of the graph via a topological numbering (see Section 2.4). I.e., assign all vertices to disjoint nonempty subsets $L_1, L_2, \ldots, L_h$ of V called *layers* such that for each edge (u, v) the following holds: If the end-vertex u is assigned to layer L_i and the end-vertex v is assigned to layer L_j then $j > i$. If $j > i+1$, i.e., the edge traverses intermediate layers, we call e a *long* edge and replace it by a directed path from u to v with $j - i - 1$ artificial intermediate vertices for each traversed layer. In Figure 14, the artificial vertices coïncide with the edge bends.

(S2) **Crossing Minimization:** For each layer L, determine permutations of the vertices in L with the goal of obtaining few crossings under the following assumptions: (1) The layers are parallel horizontal lines. (2) Each vertex is drawn on the line corresponding to its layer with x-coordinate compatible with its position in the layer permutation. (3) All edges are drawn as straight line segments between consecutive layers.

(S3) **Coordinate Assignment:** Turn the topological layout of (S2) into a geometric layout by assigning to each $v \in V$ the y-coordinate $-i$ if $v \in L_i$ and an x-coordinate compatible with the vertex permutation of L_i. Finally, suppress the artificial vertices in the drawing.

For each of the three phases, there is a variety of possibilities for implementation. We restrict ourselves to complexity considerations and the discussion of a selection of ideas that are widely used in the practice of automatic graph drawing.

Layer Assignment. In phase (S1) we would like to avoid too many artificial vertices, because they induce long edge drawings and have a negative influence on the running times of the later phases as we will see. Call h the *height* and $\max_{1 \leq i \leq h} |L_i|$ the *width* of the layer assignment. When we postulate a compact final drawing as an æsthetic requirement, we must deal with the tradeoff of keeping both height and width small or reasonably related in order to obtain some desired aspect ratio.

The minimization of the number of artificial vertices has been successfully addressed by Gansner *et al.* [40]. If y_v denotes the vertical coordinate of vertex $v \in V$, then the problem can be formulated as the integer linear programming problem

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v) \in E} y_u - y_v \\ \text{subject to} \quad & y_u - y_v \geq 1 \text{ for all } (u,v) \in E \\ & y_v \geq 0 \text{ and integral for all } v \in V \end{aligned}$$

that can be solved efficiently in polynomial time with network flow techniques. (For network flow algorithms, see, e.g., Cook *et al.* [18].)

It is quite simple to compute a layer assignment with minimum height by observing that the length of a longest directed path in G is a lower bound on the height of the layer assignment, and that an easy modification of a topological sorting algorithm, called *longest path layering* can be used to find a layer assignment with this height. The method uses a first-in-first-out-queue Q of vertices initialized with all vertices $v \in V$ with $\text{indeg}(v) = 0$. Starting with $i = 1$, each vertex $v \in Q$ is removed from Q, assigned to L_i, and all edges in $\text{outstar}(v)$ are deleted. When a deletion operation leads to $\text{indeg}(w) = 0$ for a vertex $w \in \text{outadj}(v)$, then w is inserted as a new vertex at the end of Q. Finally, i is increased by one as soon as all old vertices have been processed and then the new vertices in Q become old vertices. This procedure takes $O(|V| + |E|)$ time and space. Its drawback is that it has no control over the width of the layer assignment.

Unfortunately, it is $\mathcal{NP}$-hard to minimize the height for a given width $w \geq 3$. This can be proved via a simple transformation from a well-known $\mathcal{NP}$-hard multiprocessor scheduling problem in which $|V|$ unit-time jobs with $|E|$ precedence constraints between pairs of jobs must be processed on w parallel machines so as to minimize the completion time. This relation suggests the application of a very popular polynomial time approximative algorithm by Coffman and Graham [19] for this multiprocessor scheduling problem to

the layer assignment problem. We refrain from explaining the method but only state that if h is the height attained by the algorithm and $h_{\min}$ is the minimum possible height given width w, then we have $h \leq (2 - \frac{2}{w})h_{\min}$, i.e., the algorithm computes the optimum solution for $w = 2$ and has a decent performance guarantee for larger w.

From now on, let $G = (V, E)$ be the acyclic digraph *after* the addition of artificial vertices and their incident edges.

Crossing Minimization. The crossing minimization problem is $\mathcal{NP}$-hard even when restricted to 2-layer instances. Nevertheless, it has been attacked with a *branch-and-cut algorithm* that produces an optimum solution in exponential running time, see Healy and Kuusik [44]. (For an introduction to branch-and-cut algorithms see, e.g., Elf *et al.* [33].) However, such methods have not yet reached the maturity and practical efficiency to be used in software systems for automatic graph drawing. Instead, most systems apply a *layer by layer sweep* as follows: Starting from some initial permutation of the vertices on each layer, such heuristics consider pairs of layers $(L_{\text{fixed}}, L_{\text{free}}) = (L_1, L_2), (L_2, L_3), \ldots, (L_{h-1}, L_h), (L_h, L_{h-1}), \ldots, (L_2, L_1), (L_1, L_2), \ldots$ and try to determine a permutation of the vertices in L_{free} that induces a small bilayer cross count for the subgraph induced by the two layers, while keeping L_{fixed} temporarily fixed. These down and up sweeps continue until no improvement is achieved.

Thus the problem is reduced to the *2-layer crossing minimization problem* in which a vertex permutation on one layer is fixed and the other is computed so as to induce the minimum (or at least a small) number of pairwise interior edge crossings among the edges connecting vertices of the two layers. Eades and Wormald [31] have shown that the 2-layer crossing minimization problem with one fixed layer is $\mathcal{NP}$-hard as well, nevertheless, in this case, the optimum can be found efficiently in practice for instances with up to about 60 vertices on the free layer, albeit with a rather complicated branch-and-cut algorithm by Jünger and Mutzel [49]. Experimental studies in the same article have shown that certain efficient heuristics perform very well in practice. We describe two of them: the *barycenter heuristic* of Sugiyama *et al.* [62] and the *median heuristic* by Eades and Wormald [31].

In the 2-layer crossing minimization problem with one fixed layer we have a bipartite graph $G = (V, E)$ with bipartition $V = N \dot{\cup} S$ such that all edges $(u, v) \in E$ have $u \in N$ (the "northern layer") and $v \in S$ (the "southern layer"). Given a permutation $\langle n_1, n_2, \ldots, n_p \rangle$ of all $n_i \in N$, $i \in \{1, 2, \ldots, p\}$ we wish to find a permutation $\langle s_1, s_2, \ldots, s_q \rangle$ of all $s_j \in S$, $j \in \{1, 2, \ldots, q\}$ that induces a small number of interior edge crossings when the edges are drawn as straight line segments connecting the positions of their end-vertices which are placed on two parallel lines according to the permutations. We use the example in Figure 15 for illustration. The permutations given in this figure induce 12 crossings.

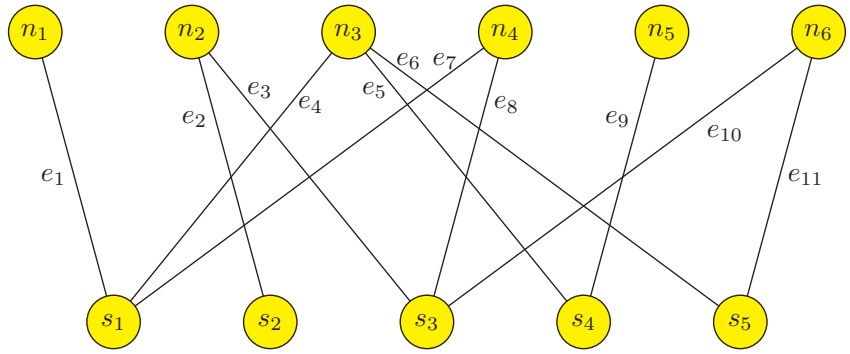


Fig. 15. A two layer graph.

For ease of exposition, we assume without loss of generality that there are no isolated vertices. For each vertex $s \in S$, let

$$\text{barycenter}(s) = \frac{1}{\text{indeg}(s)} \sum_{n_i \in \text{ inadj}(s)} i$$

$$\text{median}(s) = \text{med}\{i \mid n_i \in \text{inadj}(s)\}$$

where $\text{med}(M)$ denotes the *median*, i.e., the element in position $\lfloor \frac{|M|}{2} \rfloor$ when the finite multi-set $M \subseteq \mathbb{N}$ is sorted in ascending order.

The medians and barycenters can be computed for all $s \in S$ in $O(|E|)$ time and space. The two heuristics return S sorted by barycenters or medians, respectively, as the southern vertex permutation. In our example, the barycenter permutation is $\langle s_2, s_1, s_3, s_4, s_5 \rangle$ with barycenter values $\langle 2, 2.\bar{6}, 4, 4, 4.5 \rangle$ and the median permutation is $\langle s_2, s_1, s_4, s_5, s_3 \rangle$ with median values $\langle 2, 3, 3, 3, 4 \rangle$. In this case the barycenter permutation induces 11 crossings and the median permutation stays at 12 crossings.

The sorting step takes $O(|S| \log |S|)$ time in the barycenter heuristic and $O(|N|)$ time in the median heuristic, so that the total running time is $O(|E| + |S| \log |S|)$ for the former and $O(|E|)$ for the latter, with $O(|E|)$ space for both.

The heuristics have been analyzed theoretically and an interesting result (shown by Eades and Wormald [31]) is that for any given northern permutation, if c is the number of crossings induced by the result of the median heuristic and $c_{\min}$ is the minimum possible number of crossings, then $c \leq 3\, c_{\min}$ when a certain tie breaking rule in the sorting step is obeyed.

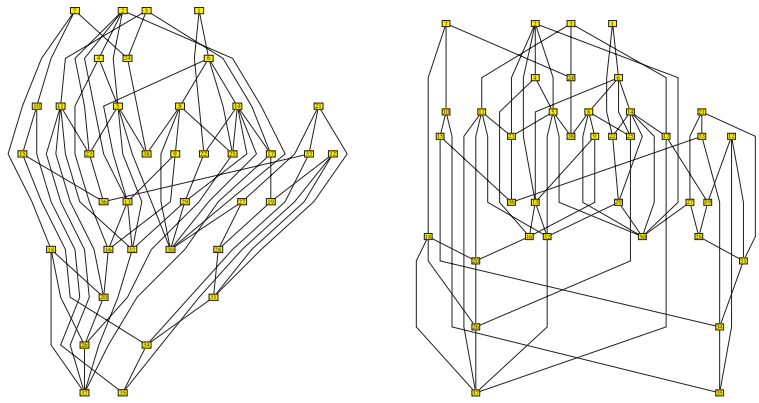
Finally, we discuss an innocent looking problem, namely counting the number of crossings once crossing minimization heuristics have been performed, in order to decide if the layer by layer sweep should be continued or terminated. Since all crossings occur between consecutive layer pairs, the problem reduces to the *2-layer cross counting problem*: Given permutations

π_N of N and π_S of S, determine the number of pairwise interior edge crossings in the above setting.

Of course, it is easy to determine if two given edges in a 2-layer graph with given permutations π_N and π_S cross or not by simple comparisons of the relative orderings of their end vertices on L_N and L_S. This leads to an obvious algorithm with running time $O(|E|^2)$. This algorithm can even output the crossings rather than only count them, and since the number of crossings is $\Theta(|E|^2)$ in the worst case, there can be no asymptotically better algorithm. However, we do not need a list of all crossings, but only their number.

The best known approaches to the 2-layer cross counting problem, both in theory and in practice, are by Waddle and Malhotra [69] and by Barth *et al.* [1] and run in $O(|E| \log |E|)$ and $O(|E| \log(\min\{|N|, |S|\}))$ time, respectively. The former is a sweep line algorithm and the latter uses a reduction of the cross counting problem to the counting of the inversions of a certain sequence. We refrain from a detailed description and only mention that an experimental evaluation by Barth *et al.* [1] shows that a combination of the median heuristic for crossing minimization and cross counting with any of the $O(|E| \log |E|)$ algorithms can be performed for very large graphs very fast.

Coordinate Assignment. After the topology of the layout has been fixed by the previous two phases (S1) and (S2), the purpose of phase (S3) is the assignment of coordinates to the vertices. Since each artificial vertex introduced in phase (S1) gives rise to a possible edge bend in the final layout, a careless implementation of phase (S3) usually leads to the so-called "spaghetti effect" as shown in Figure 16(a).



(a) (b)

Fig. 16. The spaghetti effect and a remedy.

The goal is to avoid this effect by "straightening" the long edges that traverse layers in the layout so as to obtain results as shown in Figure 16(b). A landmark paper on layered layout by Gansner *et al.* [40] treats this problem in much detail. In addition to a layer by layer sweep heuristic similar in spirit to the one described in the previous subsection for crossing reduction, the authors give an integer linear programming formulation as follows.

For each $e \in E$, let $\Omega(e)$ be a priority for edge e to be drawn as a vertical line segment. We wish to assign integer x-coordinates within a grid drawing in which the vertices are separated by at least one grid unit. If x_v is the x-coordinate to be assigned to $v \in V$, then the spaghetti avoidance problem can be formulated as the integer non-linear program

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v)\in E} \Omega((u,v))\,|x_v - x_u| \\ \text{subject to} \quad & x_j - x_i \geq 1 \text{ for all pairs } i \text{ and } j \text{ of vertices} \\ & \qquad \text{within a layer permutation, where} \\ & \qquad i \text{ is immediately followed by } j \\ & x_v \geq 0 \text{ and integral for all } v \in V \end{aligned}$$

which can be transformed via additional variables to the integer linear program

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v)\in E} \Omega((u,v))\,z_{uv} \\ \text{subject to} \quad & z_{uv} \geq x_v - x_u \text{ for all } (u,v) \in E \\ & z_{uv} \geq x_u - x_v \text{ for all } (u,v) \in E \\ & x_j - x_i \geq 1 \text{ for all pairs } i \text{ and } j \text{ of vertices} \\ & \qquad \text{within a layer permutation, where} \\ & \qquad i \text{ is immediately followed by } j \\ & x_v \geq 0 \text{ and integral for all } v \in V \end{aligned}$$

that can be solved efficiently in polynomial time using network flow techniques. The authors recommend choosing

$$\Omega((u,v)) = \begin{cases} 1 & \text{if both } u \text{ and } v \text{ are non-artificial,} \\ 2 & \text{if exactly one of } u \text{ and } v \text{ is artificial,} \\ 8 & \text{if both } u \text{ and } v \text{ are artificial.} \end{cases}$$

Ideally, a long layer traversing edge has at most two bends, one at its first and one at its last artificial vertex, and is drawn vertically in between. The optimum of the above optimization problem cannot guarantee this. With certain additional requirements on the outcome of the crossing minimization phase (S2), two fast heuristics overcome this problem by trying to obtain near optimum solutions to the above optimization problem under the additional restriction that all long edges indeed have at most two bends and the

internal parts are drawn vertically, the first by Buchheim *et al.* [14] runs in $O(|E| \log^2 |E|)$ time, and the second by Brandes and Köpf [10] runs in $O(|E|)$ time. Both produce visually pleasing results very efficiently.

Making any Graph Acyclic and Directed. If we want to apply Sugiyama-style layout to a non-acyclic digraph, a natural way to make it acyclic is reversing the directions of edges in order to obtain an acyclic digraph. In many cases, such as the drawing of flowcharts, the input data can be expected to determine the choice of such reversals. In the absence of such input data, we would like to reverse as few edges as possible. Thus we can guarantee that, in the final layout, a minimum number of edges point upward rather than downward. This problem is equivalent to the *feedback arc set problem*, also known as the *acyclic subdigraph problem*. It is $\mathcal{NP}$-hard yet can be solved in many reasonably sized cases to optimality by branch-and-cut, see Jünger *et al.* [50]. When more than the minimum number of edges are reversed, the equivalence is lost. Fortunately, there are fast heuristics for finding a small number of edges whose reversal makes the digraph acyclic, most notably a heuristic by Eades *et al.* [30] that runs in $O(|E|)$ time and guarantees a solution in which at most $\frac{|E|}{2} - \frac{|V|}{6}$ edges must be reversed in order to obtain an acyclic digraph.

If we want to apply a Sugiyama-style method to an undirected graph, various application-dependent considerations may guide the assignment of directions to the edges so as to obtain an acyclic digraph. In the absence of such guidance, or in addition to it, it is reasonable to assign the directions to the edges with the goal of obtaining a compact final layout. Sander [61] discusses various heuristics, among them a force-directed layout (see Section 4.5) from which the layer assignment, and thus the edge direction assignment, is extracted. This practically successful idea saves phase (S1) and tends to produce uniform edge lengths. See [24] for an experimental study of the many alternative ways to draw directed graphs.

4.3 Planarization

There are many interesting drawing methods for planar graphs that yield plane drawings as we will discuss in Section 4.4. Such a method can be applied to a non-planar graph G after transforming G into a planar graph G'. The basic idea of the planarization method was introduced by Tamassia *et al.* [67].

There are different ways of *planarizing* a given non-planar graph. The most widely used method in graph drawing is to construct an embedding of G with a small number of crossings, and then to substitute each crossing and its involved pair of edges $(\{u_1, v_1\}, \{u_2, v_2\})$ by an artificial vertex w and four incident edges $\{w, u_1\}, \{w, u_2\}, \{w, v_1\}$, and $\{w, v_2\}$. We call the resulting planar graph $G_P = (V_P, E_P)$ a *planarized* graph from G.

After the *planarization phase*, the resulting planar graph G' is drawn using a planar drawing algorithm, and then the artificial vertices are re-substituted by crossings.

The *crossing minimization problem* searches for a drawing with the minimum number of crossings. Unfortunately, this problem is $\mathcal{NP}$-hard [39]. A classical approach for generating a drawing with a small number of crossings is to compute a planar subgraph P of G by temporarily removing edges. Then the removed edges are re-inserted while trying to keep the number of crossings small. In practice, this method usually leads to drawings with few crossings.

Further restrictions of planarity, such as upward or c-planarity, lead to similar planarization methods. Under the restriction that upward planarity and c-planarity can only be tested for a subset of directed and clustered graphs, respectively, approaches for *upward planarization* and *cluster planarization* can be developed by using the corresponding testing algorithms.

In the sequel we will explain the two steps, namely edge removal and edge re-insertion, in more detail.

Edge Removal. We consider the *maximum planar subgraph problem* which is the problem of removing the minimum number of edges of a non-planar graph so that the resulting graph P is planar. This problem is $\mathcal{NP}$-hard [56]. However, Jünger and Mutzel suggest a branch-and-cut algorithm which is able to solve small problem instances to provable optimality in short computation time [48]. Since this algorithm has exponential running time in the worst case, heuristics are often used for solving this problem in practice.

A subgraph $P = (V, E')$ of a graph $G = (V, E)$ is called a *maximal planar subgraph* of G if there is no edge $e \in E \setminus E'$ so that the graph $(V, E' \cup e\}$ is planar.

A simple algorithm for computing a maximal planar subgraph of a given graph G is to start with the empty graph and successively add the edges of G if their addition results in a planar graph. Edges destroying the planarity are discarded (see Algorithm 1). The incremental algorithm can be implemented in time $O(|V||E|)$ by simply calling a linear planarity testing algorithm. Di Battista and Tamassia have suggested an incremental planarity testing algorithm which can test in $O(\log |V|)$ time whether an edge can be added to a graph without destroying planarity, thus leading to a total running time of $O(|E| \log |V|)$. The best theoretical algorithm for incremental planarity testing has been suggested by Djidjev [27] and runs in time $O(|E| + |V|)$. It is based on the SPQR-tree data structure and special dynamic data structures that allow union and splits for sets in constant amortized time.

Another method based on the planarity testing algorithm by Hopcroft and Tarjan has been suggested by Cai *et al.* [16] and runs in time $O(|E| \log |V|)$.

An algorithm based on PQ-trees has been suggested by Jayakumar *et al.* [46] and runs in time $O(|V|^2)$. The peculiarities of this algorithm are

Algorithm 1: Incremental maximal planar subgraph

```
Input  : Graph G = (V, E)
Output: Maximal planar subgraph P = (V, F) of G
Set F = ∅
for all edges e ∈ E do
    if P = (V, F ∪ {e}) is planar then
        Set P = (V, F ∪ {e})
    end
end
```

discussed in various subsequent papers, see, e.g., [47], and it turns out that algorithms based on PQ-trees are not appropriate for finding maximal planar subgraphs in general. Despite the fact that this algorithm does not compute a maximal planar subgraph, it leads to larger subgraphs than the naïve method in general. Ziegler has shown that the results improve if the algorithm is applied several times to the same graph with perturbed input data [71].

Edge Re-Insertion. Given a planar subgraph P of $G = (V, E)$, our task is to re-insert the removed edges so that the number of edge crossings is small. More formally, the problem is to create a drawing with the minimum number of crossings in which the planar subgraph is drawn crossing-free. Without the latter restriction, the problem would be equivalent to the crossing minimization problem.

The most widely used edge re-insertion method is shown in Algorithm 2. It chooses a planar embedding of P and successively inserts the edges of F. After each re-insertion step, the crossings are substituted by artificial vertices, thus providing a planar graph after each step. For a fixed planar embedding, an edge $e = \{u, v\}$ can be inserted with the minimum number of crossings by computing a shortest path in an extended dual graph. The extension is necessary to connect the primal vertices u and v with the dual graph. This is done by adding u^* and v^* inside the faces F_u and F_v that correspond to u and v in the primal graph, respectively, and by adding artificial edges from the new vertices to the dual vertices bounding the faces F_u and F_v. The new edge creates a crossing in G_P whenever a (real) dual edge in G_P^* is used, which corresponds to crossing the boundary of two adjacent faces. Figure 17 shows a plane graph, its extended dual graph, and an edge e to be inserted. The algorithm can be implemented in time $O(|F|(|E'| + |C|))$, where $|C|$ is the number of generated crossings.

Clearly, the number of generated crossings highly depends on the chosen embedding Π at the beginning. Gutwenger *et al.* [43] have presented an algorithm which solves the one-edge insertion problem optimally over the set of all embeddings of G in linear time, thus overcoming this problem. Since the embedding of G_P is changed after substituting the crossings dur-

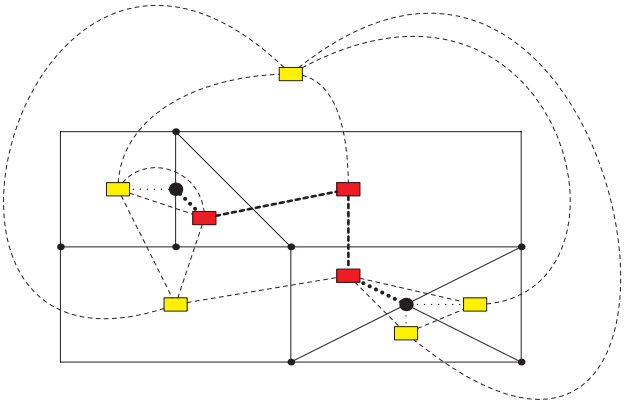


Fig. 17. The edge re-insertion step for one edge.

Algorithm 2: Classical edge re-insertion algorithm

Input : Planar graph $P = (V, E')$ of $G = (V, E)$ and edge set $F = E \setminus E'$
Output: Planarized graph $G_P = (V_P, E_P)$ of G

Set $G_P = (V, E')$
Compute a planar embedding Π of G_P
Compute the dual graph G_P^* of G_P with respect to Π
for *all edges* $e = \{u, v\} \in F$ **do**
  Extend the dual graph G_P^* at the end-vertices of e
  Compute a shortest path from u to v in G_P^*
  Update $G_P = (V_P, E_P)$ by substituting all crossings by new vertices
  Update the dual graph G_P^*
end

ing the run of the algorithm, it may happen that some of the crossings are not needed anymore in the final drawing, and this leads to further crossing reduction. The edge re-insertion is specified in Algorithm 3, whose running time is $O(|F|(|E'| + |C|))$.

Algorithm 3: Optimal embedding re-insertion algorithm

Input : Planar graph $P = (V, E')$ of $G = (V, E)$ and edge set $F = E \setminus E'$
Output: Planarized graph $G_P = (V_P, E_P)$ of G

Set $G_P = (V, E')$
for *all edges* $e = \{u, v\} \in F$ **do**
  Call the optimal 1-edge re-insertion algorithm for e and G_P
  Update $G_P = (V_P, E_P)$ by substituting all crossings by new vertices
end
Reduce superfluous crossings in the planarized graph G_P

The number of generated crossings can be further decreased by a remove-and-reinsert post-processing step (see, e.g., Ziegler [71] for experimental results). A typical drawing using planarization in combination with a planar orthogonal layout method is shown in Figure 18.

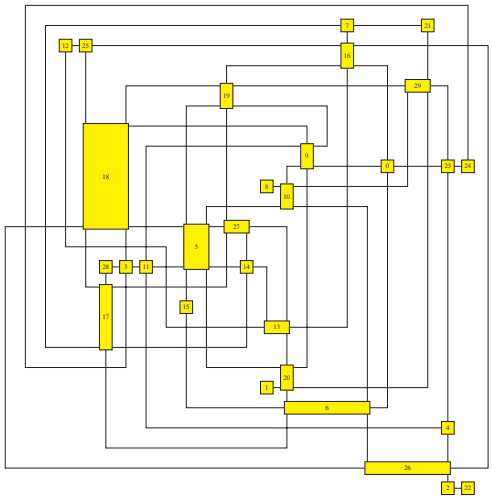


Fig. 18. A typical drawing in planarization style.

4.4 Orthogonal Layout

A popular style in graph drawing is orthogonal drawing. In an *orthogonal drawing* each edge is represented as a chain of horizontal and vertical segments. A point where a horizontal and a vertical segment of an edge meet is called a *bend*.

A popular orthogonal drawing method is the *topology-shape-metrics approach* of Batini *et al.* [2], which we will discuss now. The topology-shape-metrics method focuses on the æsthetic criteria crossings, bends, and area of the drawing and tries to keep these numbers small while fixing the topology, the shape, and the edge lengths, in this order. It deals with topology, shape, and geometry of the drawing separately, allowing for each æsthetic criterion to be addressed in the corresponding step, avoiding the complexity of a global optimization. Experimental results have shown that minimization in all these steps does in fact lead to better drawings (see, e.g., [25]). Algorithm 4 gives an overview of the topology-shape-metrics method.

The topology is fixed using the planarization method based on planar subgraphs (see Section 4.3). Then, a planar orthogonal drawing method is used for planar graphs, which we will discuss in the sequel.

Algorithm 4: The topology-shape-metrics method

Input : Graph $G = (V, E)$;

Output: Orthogonal drawing of $G = (V, E)$

Planarization. If G is planar, then a planar embedding for G is computed. If G is not planar, a set of artificial vertices is added to replace crossings.

Orthogonalization. During this step, an orthogonal representation H of G is computed within the previously computed embedding.

Compaction. In this step a final geometry for H is determined. Namely, coordinates are assigned to vertices and bends of H.

Orthogonalization. Initially, we consider only *4-planar* graphs, i.e., planar graphs G with $\text{maxdeg}(G) = 4$. Given a 4-planar graph G with planar embedding Π, the algorithm by Tamassia [66] computes a planar orthogonal grid embedding with the minimum number of bends in polynomial time by transforming it into a minimum-cost flow problem in a network. The network is based on the dual graph given by Π and contains $O(|V|+|F|)$ vertices and $O(|V|+|E|)$ edges for a planar embedded graph with $|F|$ faces.

The algorithm is *region preserving*, i.e., the underlying topological structure given in Π is not changed by the algorithm.

Each feasible flow in the network corresponds to a possible shape of G. In particular, the minimum cost flow leads to the orthogonal shape with the lowest number of bends since each unit of cost is associated with a bend in the drawing. The flow is used to build a so-called *orthogonal!representation* H that describes the shape of the final drawing in terms of bends occurring along the edges and angles formed by the edges. Formally, H is a function from the set of faces F to lists of triples $r = (e_r, s_r, a_r)$ where e_r is an edge, s_r is a bit string, and a_r is the angle formed with the following edge inside the appropriate face. The bit string s_r provides information about the bends along edge e_r, and the kth bit describes the kth bend on the right side of e_r where a zero indicates a 90° bend and a one a 270° bend. The empty string ε is used to characterize straight line edges. Figure 19 shows an example.

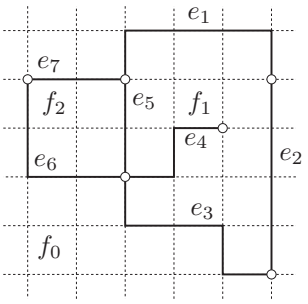
There are four necessary and sufficient conditions for an orthogonal representation H to be a valid shape description of some 4-planar graph:

(P1) There is a 4-planar graph whose planar embedding Π is identical to that given by H restricted to the e-fields. We say that H *extends* Π.

(P2) Let r and r' be two elements in H with $e_r = e_{r'}$. Since each edge is contained twice in H these pairs always exist. Then string $s_{r'}$ can be obtained by applying bitwise negation to the reversion of s_r.

(P3) Let $|s|_0$ and $|s|_1$ denote the numbers of zeroes and ones in string s, respectively. Define for each element r in H the value

$$\rho(r) = |s_r|_0 - |s_r|_1 + (2 - \frac{a_r}{90}).$$



$$H(f_1) = \big((e_1, 00, 180), (e_2, \varepsilon, 90), (e_3, 010, 90), (e_4, 10, 360), (e_4, 10, 90), (e_5, \varepsilon, 180)\big)$$
$$H(f_2) = \big((e_6, 0, 90), (e_7, \varepsilon, 90), (e_5, \varepsilon, 90)\big)$$
$$H(f_0) = \big((e_7, \varepsilon, 270), (e_6, 1, 90), (e_3, 101, 270), (e_2, \varepsilon, 180), (e_1, 11, 90)\big)$$

Fig. 19. Orthogonal grid drawing with corresponding orthogonal representation of a 4-planar graph

Then for each face f

$$\sum_{r \in H(f)} \rho(r) = \begin{cases} +4 & \text{if } f \text{ is an internal face} \\ -4 & \text{if } f \text{ is the external face } f_0. \end{cases}$$

(P4) For each vertex $v \in V$ we have

$$\sum_{e_r = (u,v)} a_r = 360 \qquad \forall u \in V,$$

i.e., the angles around v given by the a-fields sum up to 360°.

We say that a drawing Γ *realizes* H if H is a valid description for the shape of Γ. Figure 19 shows an orthogonal representation H and a grid embedding realizing H. Note that the number of bends in any drawing that realizes H is

$$b(H) = \frac{1}{2} \sum_{f \in F} \sum_{r \in H(f)} |s_r|.$$

Let $G = (V, E)$ be the input graph with planar embedding Π defining the face set F. The construction of the underlying network N follows [42]: Let $U = U_F \cup U_V$ denote its vertex set. Then for each face $f \in F$ there is a vertex in U_F and for each vertex $v \in V$ there is one in U_V. Vertices $u_v \in U_V$ supply $b(u_v) = 4$ units of flow and vertices $u_f \in U_F$ consume

$$-b(u_f) = \begin{cases} 2\deg(f) - 4 & \text{if } f \text{ is an internal face} \\ 2\deg(f) + 4 & \text{if } f \text{ is the external face } f_0 \end{cases}$$

units of flow. Thus, the total supply is $4|V|$ and the total demand is

$$\sum_{f \neq f_0} (2\deg(f) - 4) + 2\deg(f_0) + 4 = 4|E| - 4|F| + 8$$

which is equal to the total supply, according to Euler's formula (Theorem 1). The arc set A of network N consists of two sets A_V and A_F where

$$A_V = \{(u_v, u_f) \mid u_v \in U_V,\ u_f \in U_F,\ v \text{ is adjacent to } f\} \quad \text{and}$$
$$A_F = \{(u_f, u_g) \mid u_f \neq u_g \in U_F,\ f \text{ is adjacent to } g\}$$
$$\cup\ \{(u_f, u_f) \mid f \text{ contains a bridge}\}.$$

Arcs in A_V have lower bound 1, capacity 4, and cost 0. Each unit of flow represents an angle of 90°, so a flow in an arc $(u_v, u_f) \in A_V$ corresponds to the angles formed at vertex v inside face f. Note that there can be more than one angle, see for example Figure 19 where the vertex common to edges e_6, e_5, e_4, and e_3 builds two angles in f_1. Precisely, the flow in (u_v, u_f) corresponds to the sum of the angles at v inside f. Following this interpretation, flow in arcs $(u_f, u_g) \in A_F$ find their analogy in bends occurring along edges separating f and g that form a 90° angle in f. Naturally their lower bound is 0, their capacity unbounded, and they have unit cost.

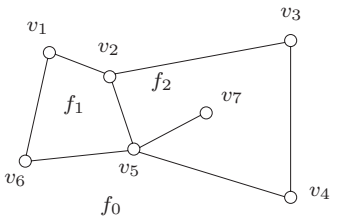
The conservation rule at vertices $u_v \in U_V$ expresses that the angle sum around the corresponding vertex v is equal to 360°. The vertices in U_F consume flow; here, the conservation rule states that every face has the shape of a rectilinear polygon. A planar graph and the transformation into a network is shown in Figure 20(a)–(d).

It is easy to see that there is always a feasible flow in network N: Flow produced by vertices in U_V can be transported to vertices in U_F where it satisfies their demand. In case it is not possible to satisfy every vertex in U_F by exclusively using arcs in A_V, units of flow can be shifted without restriction between vertices in U_F because of their mutual interconnection by arcs in A_F. Every feasible flow can be used to construct an orthogonal representation for the input graph G, in particular the minimum cost flow, leading to the orthogonal representation with the minimum number of bends. The following lemma states the analogy between flows in the network and orthogonal representations.

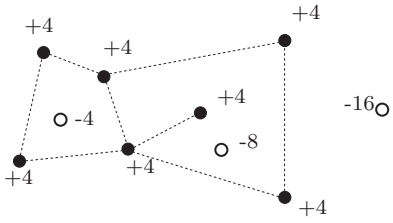
Lemma 1 (Tamassia [66]). *Let G be the input graph, Π its planar embedding, and N the constructed network. For each integer flow χ in network N, there is an orthogonal representation H that extends Π and whose number of bends is equal to the cost of χ. The flow χ can be used to construct the orthogonal representation.*

Figure 21 completes the example from Figure 20, showing the minimum cost flow in the constructed network and a realizing grid embedding for the derived orthogonal representation.

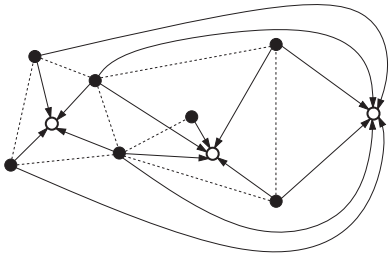
Vice versa, it can be shown that the number of bends in each orthogonal grid embedding of a graph with planar embedding Π is equal to the cost of some feasible flow in network N. This result and Lemma 1 lead to the following theorem, combining the basic results of [66] and [42]:



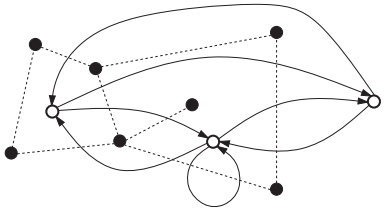
(a) Planar embedding P



(b) Nodes in network N. Supply/demand shown



(c) Arcs in A_V. Capacity 4, lower bound 1, cost 0



(d) Arcs in A_F. Capacity ∞, lower bound 0, cost 1

Fig. 20. Network construction.

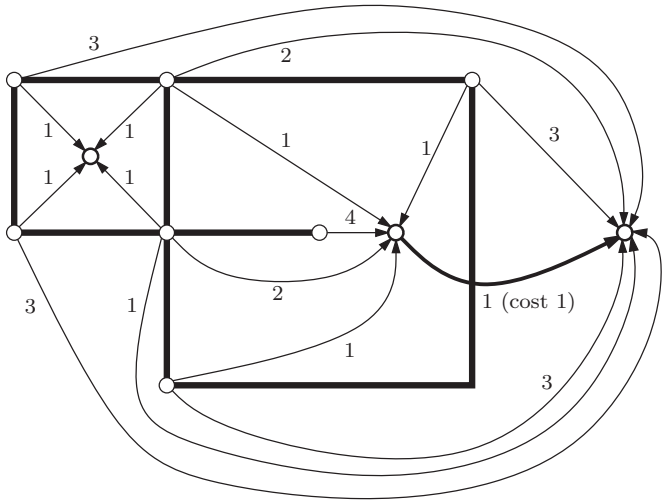


Fig. 21. Minimal cost flow in network N and a resulting grid embedding.

Theorem 6 (Tamassia [66]). *Let Π be a planar embedding of a connected 4-planar graph G and let N be the corresponding network. Each feasible flow χ in N corresponds to an orthogonal representation for G that extends Π and whose number of bends is equal to the cost of χ. In particular, the minimum cost flow can be used to construct the bend optimal orthogonal representation preserving Π.*

Garg and Tamassia [42] have shown that the minimum cost flow problem in this specific network can be solved in $O(|V|^{7/4}\sqrt{\log |V|})$ time.

Obviously, the number of bends is highly dependent on the chosen embedding. Unfortunately, the problem in the variable embedding setting is $\mathcal{NP}$-complete [41]. However, Bertolazzi *et al.* [4] have designed a branch-and-bound algorithm for solving the bend minimization problem over the set of all embeddings to optimality. An alternative approach for provably optimum solutions is based on integer linear programming and has been suggested by Mutzel and Weiskircher [58]. Both algorithms use the data structure of SPQR-trees in order to represent the set of all planar embeddings of the given planar graph.

Compaction. After the orthogonalization phase, the description is dimensionless, and coordinates need to be assigned to the vertices and bends. The problem of computing the edge lengths of an orthogonal representation minimizing the area or the total edge length of the drawing is called the *compaction problem*. This problem is $\mathcal{NP}$-hard [59].

There is a vast amount of literature concerning the compaction problem, since it has played an important role not only in graph drawing, but also in the context of circuit design. The heuristic methods can be categorized into constructive and improvement methods. Tamassia has suggested the first approach in the context of graph drawing. In [66], he provides a linear time algorithm based on rectilinear dissection of the orthogonal representation. Recently, Bridgeman *et al.* [12] have extended this technique by introducing the concept of turn-regularity, thus leading not only to better heuristics, but also to particular classes of orthogonal representations that can be solved to optimality. The compression-ridge method and other graph-based compaction methods originate in VLSI layout and constitute improvement heuristics for the compaction problem. They consider the one-dimensional compaction problem of reducing the horizontal or vertical edge lengths. Experimental studies by Klau *et al.* [53] have shown that the heuristics lead to tremendous improvements in area and edge length minimization.

Klau and Mutzel have presented a branch-and-cut algorithm which is able to solve the compaction problem with respect to edge length minimization to provable optimality [54]. The approach is based on a new combinatorial formulation of the problem. Besides the possibility of providing an integer linear programming formulation for the problem, the new approach also provides

new classes of orthogonal representations that can be solved in polynomial time [54].

High Degree Orthogonal Drawings. As already discussed above, the bend minimization algorithm by Tamassia only works for graphs with vertex degree bounded by four. In order to generate orthogonal drawings for graphs of arbitrary vertex degree, different drawing conventions have been introduced in the literature. Here we introduce the basic Kandinsky drawing convention, defined by Fößmeier and Kaufmann [36].

A *basic Kandinsky drawing* (see Figures 22(a) and (b)) is an orthogonal drawing such that:

1. Segments representing edges may not cross, with the exception that two segments that are incident on the same vertex may overlap. Observe that the angle between such segments has zero degree. Roughly speaking, a basic Kandinsky drawing is "almost" planar: it is planar everywhere but in the possible overlap of segments incident on the same vertex. Observe in Figure 22(b) the overlap of segments incident on vertices 1, 2, and 3.
2. All the polygons representing the faces have an area strictly greater than zero.

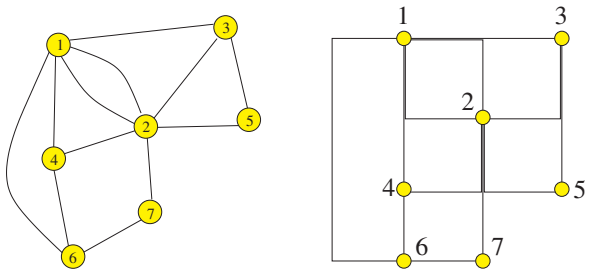
Basic Kandinsky drawings are usually visualized by representing vertices as boxes with equal size and representing two overlapping segments as two very near segments. See Figure 22(c).

In [36] an algorithm is presented that computes a basic Kandinsky drawing of an embedded planar graph with the minimum number of bends. Furthermore, the authors conjecture that the drawing problem becomes $\mathcal{NP}$-hard when condition 2 is omitted. Basic Kandinsky drawings generalize the concept of orthogonal representation, allowing angles between two edges incident to the same vertex to have zero degree. The consequence of the assumption that the polygons representing the faces have area strictly greater than zero is that the angles have specific constraints. Namely, because of conditions 1 and 2, each zero degree angle is in correspondence with exactly one bend [36]. An orthogonal representation corresponding to the above definition is a *basic Kandinsky orthogonal representation*.

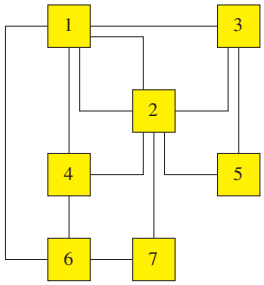
Figure 23 shows a typical planar orthogonal drawing in basic Kandinsky style.

The basic Kandinsky model has been extended in [26] to deal with drawings in which the size (width and height) of each single vertex is assigned by the user. We refer to this extended model as the Kandinsky model. A *Kandinsky drawing* has the following properties (see also Figure 22(d)):

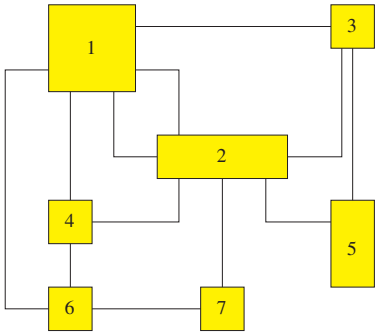
1. Each vertex is represented by a box with its specific width and height (width and height are assigned to each single vertex by the user).



(a) (b)



(c)



vertex	*width*	*height*
1	1	1
2	2	0
3	0	0
4	0	0
5	0	1
6	0	0
7	0	0

(d)

Fig. 22. (a) A planar graph and (b) one of its basic Kandinsky drawings; (c) A more effective visualization of the basic Kandinsky drawing in (b); (d) A Kandinsky drawing with the same shape as the drawing in (b); the sizes of the vertices are specified in the table.

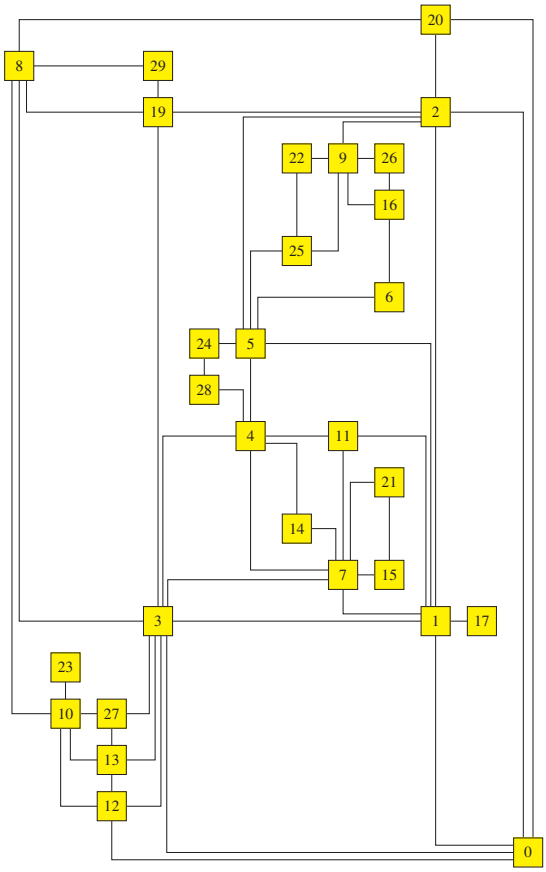


Fig. 23. A typical planar orthogonal drawing in basic Kandinsky style.

2. Consider any side of length $l \geq 0$ of a vertex v and consider the set I of arcs that are incident on such a side.
   (a) If $l + 1 \geq |I|$ then the edges of I may not overlap.
   (b) If $l + 1 < |I|$ then the edges of I are partitioned into $l + 1$ non-empty subsets such that all the edges of the same subset overlap.
3. The orthogonal representation constructed from a Kandinsky drawing by contracting each vertex into a single point is a basic Kandinsky orthogonal representation.

Di Battista *et al.* [26] suggest a polynomial time algorithm for computing Kandinsky drawings of an embedded planar graph that have the minimum number of bends over a wide class of Kandinsky drawings.

4.5 Force Directed Layout

The basic idea of force-directed methods is to associate the vertices of a graph with physical entities and the edges with interactions between their end-vertex entities. Imagine that the vertices are charged particles with mutual repulsion and that the edges are springs attached at their end-vertices. Let this physical system relax to a (locally) minimum energy state in three-dimensional space, assign to the vertices the Cartesian coordinates of their corresponding vertex particles in this state and draw the edges connecting the positions of their end-vertices as straight lines. This captures the general idea of force-directed methods for three-dimensional graph drawing. We apply this idea to simple undirected graph drawing in two dimensions and treat non-simple and/or directed graphs as well as three-dimensional layouts afterwards. See Figure 24 for a typical force-directed layout in two dimensions.

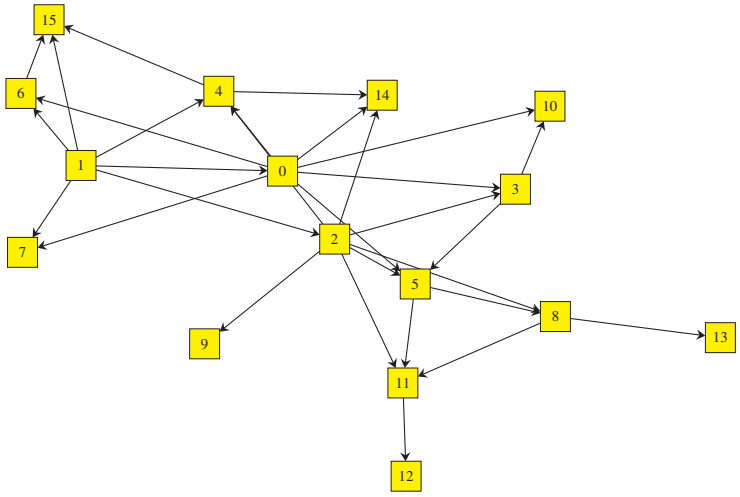


Fig. 24. A typical force-directed layout.

It should be stressed that the title of this section is a concession to the short tradition of automatic graph drawing and the title "Drawing on Physical *Analogies*" of a recommended survey of Brandes [9] would be much more appropriate, because what follows has little to do with what physicists do when they study ground states of physical systems and relaxation dynamics.

Two-Dimensional Force-Directed Layout of Undirected Graphs. For ease of notation, we identify vertices and edges with their physical counterparts. In approaching the charged particles and spring model, the force acting

on a vertex $v \in V$ depends on the locations $p_v = (x_v, y_v) \in \mathbb{R}^2$ of the vertices $v \in V$ and is composed of the repulsive forces by the other vertices and the forces by the edges in star(v), i.e.,

$$F(p_v) = \sum_{u \in V \setminus \{v\}} \text{repulsionforce}(p_u, p_v) + \sum_{\{u,v\} \in \text{star}(v)} \text{springforce}(\{p_u, p_v\}).$$

For $p = (x, y) \in \mathbb{R}^2$ let $\|p\| = \sqrt{x^2 + y^2}$ denote the Euclidean norm of p and let $\overrightarrow{p_u p_v} = \frac{p_v - p_u}{\|p_v - p_u\|}$ be the normalized difference vector of p_v and p_u. We may choose the repulsion force between vertices u and v to follow an inverse square law, i.e.,

$$\text{repulsionforce}(p_u, p_v) = \frac{R}{\|p_u - p_v\|^2} \cdot \overrightarrow{p_u p_v}$$

where R is a repulsion constant, and the spring force of edge $\{u, v\}$ according to Hooke's law, i.e.,

$$\text{springforce}(p_u, p_v) = S_{\{u,v\}} \cdot (\|p_u - p_v\| - l_{\{u,v\}}) \cdot \overrightarrow{p_u p_v}$$

where $S_{\{u,v\}}$ is the stiffness and $l_{\{u,v\}}$ is the natural length of the spring between u and v such that springforce(u, v) is proportional to the difference between the distance of u and v and the natural length of the spring. The natural spring lengths $l_{\{u,v\}}$ reflect the desirable lengths of the edges in the drawing and can be passed together with the stiffnesses $S_{\{u,v\}}$ as additional input to a force-directed method.

Force-directed methods try to compute vertex positions p_v for which the physical system attains an equilibrium state in which $F(p_v) = 0$ for all $v \in V$. Such a state is approximated in practice by iterative algorithms that, starting with some initial (possibly random) positions p_v for the vertices $v \in V$, compute $F(p_v)$ for all $v \in V$ and then update the positions $p_v \longleftarrow p_v + \mu \cdot F(p_v)$ where the step length μ is a small number, either given as a parameter or chosen dynamically depending on the number of iterations already performed. The (dynamic) choice of μ, a stopping criterion, and whether all moves are parallel or sequential are among the many choices an implementor of a force-directed method must make. For the dynamic choice of μ, all kinds of iterative improvement schemes, e.g., simulated annealing or genetic algorithms, can be used.

Inspired by previous usage of force-directed methods in VLSI layout algorithms, this method that is usually referred to as *spring embedder* was introduced into automatic graph drawing by Eades [28] with a modified force model that replaces the definition of springforce(p_u, p_v) with a logarithmic counterpart

$$\text{springforce}_{\text{Eades}}(p_u, p_v) = S_{\{u,v\}} \cdot \log\left(\frac{\|p_u - p_v\|}{l_{\{u,v\}}}\right) \cdot \overrightarrow{p_u p_v}\ .$$

A very early paper on automatic graph drawing by Tutte [68] can be interpreted as a spring embedder method for the special case $R = 0$, $S_e = 1$ and $l_e = 0$ for all $e \in E$ so that $F(p_v)$ is replaced by

$$F_{\text{Tutte}}(p_v) = \sum_{\{u,v\} \in \text{star}(v)} (p_v - p_u)$$

and the positions of at least three vertices are fixed in advance. (If no vertices are fixed, the solution $p_v = (0, 0)$ for all $v \in V$ is an undesired optimum.) Here, the problem of coordinate assignment is reduced to the solution of a (sparse) system of linear equations. The resulting coordinates have the nice property that each non-fixed vertex position is at the barycenter of its neighbor vertex positions. If Tutte's method is applied to a 3-connected planar graph and the coordinates of the vertices of a face of some planar embedding are fixed to their positions in a strictly convex planar drawing of this face, it produces a planar straight line drawing.

Many variants closer to the generic model have been proposed and experimentally evaluated in the literature, the modifications concern various redefinitions of the forces in order to facilitate their evaluation and/or obtain faster convergence of the iterative method, and speeding up the iterative process by evaluating only a subset of the repulsion force terms in $F(p_v)$. E.g., Fruchterman and Reingold [38] replace repulsionforce(p_u, p_v) by

$$\text{repulsionforce}_{\text{FG}}(p_u, p_v) = \frac{l_{\{u,v\}}^2}{\|p_u - p_v\|} \cdot \overrightarrow{p_u p_v}$$

and springforce(p_u, p_v) by

$$\text{springforce}_{\text{FG}}(p_u, p_v) = \frac{\|p_u - p_v\|^2}{l_{\{u,v\}}} \cdot \overrightarrow{p_v p_u}$$

whereas Frick *et al.* [37] use

$$\text{repulsionforce}_{\text{FLM}}(p_u, p_v) = \frac{l_{\{u,v\}}^2}{\|p_u - p_v\|^2} \cdot (p_u - p_v)$$

$$\text{springforce}_{\text{FLM}}(p_u, p_v) = \frac{\|p_u - p_v\|^2}{l_{\{u,v\}}^2 \cdot \Phi(v)} \cdot (p_v - p_u)$$

where $\Phi(v) = 1 + \frac{\deg(v)}{2}$. They also add an additional gravitational component

$$\text{gravitationforce}_{\text{FLM}}(p_v) = \Phi(v) \cdot \gamma \cdot \left(\frac{\sum_{w \in V} p_w}{|V|} - p_v \right)$$

to $F(p_v)$, where γ is a gravitational constant, and perform all calculations in (fast) integer arithmetic. Together with more refinements, a substantial

reduction in running time without visible compromises in layout quality can be achieved.

We have not yet discussed the choice of the stiffness and the natural length parameters that can be used to control the behavior of a force-directed method. An interesting suggestion has been made by Kamada and Kawai [51]. For a connected graph $G = (V, E)$ and $u, v \in V$ let $\delta(u, v)$ denote the length of a shortest path connecting u and v. The idea is to aim at a final layout in which the distance $\|p_u - p_v\|$ is approximately proportional to $\delta(u, v)$. To this end, Kamada and Kawai use springs between all $\binom{|V|}{2}$ vertex pairs so that the force between vertices u and v can be written as

$$\text{springforce}_{\text{KK}}(p_u, p_v) = S_{uv} \cdot (\|p_u - p_v\| - \delta(u, v)).$$

They choose the stiffness parameters so that they are strong for graph-theoretically near vertices and decay according to an inverse square law with increasing distance $\delta(u, v)$, namely

$$S_{uv} = \frac{S}{\delta(u, v)^2}$$

for some constant S.

The potential energy of the spring between u and v is

$$\begin{aligned} E(u, v) &= \int \text{springforce}_{\text{KK}}(p_u, p_v)\, d(\|p_u - p_v\| - \delta(u, v)) \\ &= \frac{1}{2} S_{uv} (\|p_u - p_v\| - \delta(u, v))^2 \\ &= \frac{S}{2} \left(\frac{\|p_u - p_v\|}{\delta(u, v)} - 1 \right)^2 \end{aligned}$$

and the potential energy of the whole drawing becomes

$$E = \sum_{u,v \in V,\, u \neq v} E(u, v) = \frac{S}{2} \sum_{u,v \in V,\, u \neq v} \left(\frac{\|p_u - p_v\|}{\delta(u, v)} - 1 \right)^2.$$

Necessary conditions for the optimality of vertex positions $p_v = (x_v, y_v)$ are

$$\frac{\partial E}{\partial x_v} = 0 \text{ and } \frac{\partial E}{\partial y_v} = 0 \text{ for all } v \in V.$$

For finding an approximate solution to this nonlinear system of equations, Kamada and Kawai use an iterative algorithm that in each iteration chooses a vertex $w \in V$ on which the largest force is acting, i.e.,

$$w = \text{argmax} \left\{ \sqrt{\left(\frac{\partial E}{\partial x_v} \right)^2 + \left(\frac{\partial E}{\partial y_v} \right)^2} \;\middle|\; v \in V \right\}$$

and a line search to move it to an energy minimizing position while the positions of the vertices $v \in V \setminus \{w\}$ are temporarily fixed.

Two-Dimensional Force-Directed Layout of Directed Graphs. For directed graphs, force-directed layout methods can accommodate a preferred direction within the drawing such that each directed edge is penalized proportionally to the angle φ its drawing deviates from the preferred direction. In a more general context, Sugiyama and Misue [64] propose the following amendment to basic force-directed methods: If $\overrightarrow{p_u p_v}^{\perp}$ is the unit length vector perpendicular to $\overrightarrow{p_u p_v}$ and pointing towards a decrease of φ, they add a rotation force

$$\text{rotationforce}(p_u, p_v) = M \cdot \|p_u - p_v\|^{\alpha} \cdot \varphi^{\beta} \cdot \overrightarrow{p_u p_v}^{\perp}$$

to springforce(p_u, p_v), where M is a constant for the strength of an exterior magnetic field and the parameters α and β control the relative influence of the exterior field on vertex distance and angle deviation, respectively.

Other Extensions. Eades *et al.* [32] propose a method for the layout of hierarchical (clustered) graphs with force-directed methods. For simplicity, let us assume that the vertex set of $G = (V, E)$ is partitioned into disjoint subsets V_i, $i \in \{1, 2, \ldots, k\}$ for some $k \in \mathbb{N}$, i.e., we have a hierarchy of depth one. For each V_i they introduce an additional artificial vertex v_i that is equipped with strong attractive forces with respect to the vertices $v \in V_i$ (realized by the appropriate artificial edges) and repulsive forces with respect to the artificial vertices v_j representing the other clusters $V_j \neq V_i$.

Davidson and Harel [22] propose general energy functions that try to capture various æsthetic requirements in automatic graph drawing. They try to

$$\text{minimize} \quad \eta = \lambda_1 \eta_1 + \lambda_2 \eta_2 + \lambda_3 \eta_3 + \lambda_4 \eta_4$$

where

$$\begin{aligned}
\eta_1 &= \sum_{u,v \in V} \frac{1}{\|p_u - p_v\|^2} \\
\eta_2 &= \sum_{v \in V} \left(\frac{1}{r_v^2} + \frac{1}{l_v^2} + \frac{1}{t_v^2} + \frac{1}{b_v^2} \right) \\
\eta_3 &= \sum_{\{u,v\} \in E} \|p_u - p_v\|^2 \\
\eta_4 &= \text{ number of edge crossings}
\end{aligned}$$

and r_v, l_v, t_v, and b_v are the distances of vertex v to the right, left, top, and bottom boundary of the drawing area, respectively. Thus, η_1 contributes the repulsion between vertices, η_2 the respect for the drawing area, η_3 the preference for short edges, and η_4 the number of crossings (that can be easily calculated for any given vertex positions p_v). The parameters λ_i ($i \in \{1, 2, 3, 4\}$) in the objective function control the relative emphasis on each of the four

criteria. Davidson and Harel use a simulated annealing procedure to find an approximation to an energy minimal state of the system.

Brandes and Wagner [11] show how the layout of curved edge representations with Bézier curves can be reduced to the straight line case by placing Bézier curve control points instead of vertices.

Final Remarks on Force-Directed Methods. In the discussion above, there is no hidden assumption on the dimension of the drawing space, so everything presented can be applied for three-dimensional force-directed layout as well with the obvious modifications. E.g., Bruß and Frick [13] present an extension of the method of Frick *et al.* [37] while Cruz and Twarog [20] present an extension of the method of Davidson and Harel [22] in which the cross counting component that is irrelevant in a three-dimensional drawing is replaced by an edge-edge repulsion term.

Due to their general applicability and the lack of special structural assumptions as well as for the ease of their implementation, force-directed methods play a central role in automatic graph drawing. Just like layered drawing methods they are included in many graph drawing software packages. An experimental comparison of various force-directed approaches is presented by Brandenburg *et al.* [8].

References

1. Barth, W., Jünger, M., Mutzel, P. (2002) Simple and efficient bilayer cross counting. In: M. Goodrich and S. Kobourov (eds.) Graph Drawing '02, Lecture Notes in Computer Science 2528, Springer-Verlag, 130–141
2. Batini, C., Nardelli, E., Tamassia, R. (1986) A layout algorithm for data flow diagrams. IEEE Transactions on Software Engineering **SE-12** (4), 538–546
3. Bertolazzi, P., Di Battista, G., Didimo, W. (1998) Quasi-upward planarity. In: S. H. Whitesides (ed.) Graph Drawing '98, Lecture Notes in Computer Science 1547, Springer-Verlag, 15–29
4. Bertolazzi, P., Di Battista, G., Didimo, W. (2000) Computing orthogonal drawings with the minimum number of bends. IEEE Transactions on Computers **49** (8), 826–840
5. Bertolazzi, P., Di Battista, G., Liotta, G., Mannino, C. (1994) Upward drawings of triconnected digraphs. Algorithmica **6** (12), 476–497
6. Bertolazzi, P., Di Battista, G., Mannino, C., Tamassia, R. (1998) Optimal upward planarity testing of single-source digraphs. SIAM Journal on Computing **27**, 132–169
7. Booth, K., Lueker, G. (1976) Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. Journal of Computer and System Sciences **13**, 335–379
8. Brandenburg, F. J., Himsolt, M., Rohrer, C. (1996) An experimental comparison of force-directed and randomized graph drawing algorithms. In: F.-J. Brandenburg (ed.) Graph Drawing '95, Lecture Notes in Computer Science 1027, Springer-Verlag, 76–87

9. Brandes, U. (2001) Drawing on Physical Analogies. In: M. Kaufmann and D. Wagner (eds.) Drawing Graphs, Lecture Notes in Computer Science 2025, Springer-Verlag, 71–86
10. Brandes, U., Köpf, B. (2002) Fast and simple horizontal coordinate assignment. In: P. Mutzel, M. Jünger, S. Leipert (eds.) Graph Drawing '01, Lecture Notes in Computer Science 2265, Springer-Verlag, 31–44
11. Brandes, U., Wagner, D. (2000) Using graph layout to visualize train interconnection data. J. Graph Algorithms and Applications **4** (3), 135–155
12. Bridgeman, S., Di Battista, G., Didimo, W., Liotta, G., Tamassia, R., Vismara, L. (2000) Turn-regularity and optimal area drawings of orthogonal representations. Computational Geometry: Theory and Applications, **16**, 53–93
13. Bruß, I., Frick, A. (1996) Fast interactive 3-D graph visualization. In: F.-J. Brandenburg (ed.) Graph Drawing '95, Lecture Notes in Computer Science 1027, Springer-Verlag, 99–110
14. Buchheim, C., Jünger, M., Leipert, S. (2001) A fast layout algorithm for k-level graphs. In: J. Marks (ed.) Graph Drawing '00, Lecture Notes in Computer Science 1984, Springer-Verlag, 229–240
15. Buchheim, C., Jünger, M., Leipert, S. (2002) Improving Walker's algorithm to run in linear time. In: M. Goodrich and S. Kobourov (eds.) Graph Drawing '02, Lecture Notes in Computer Science 2528, Springer-Verlag, 344–353
16. Cai, J., Han, X., Tarjan, R. E. (1993) An $O(m \log n)$-time algorithm for the maximal planar subgraph problem. SIAM Journal on Computing **22**, 1142–1164
17. Chiba, N., Nishizeki, T., Abe, S., Ozawa T. (1985) A linear algorithm for embedding planar graphs using PQ-trees. Journal of Computer System Science **30** (1), 54–76
18. Cook, W. J., Cunningham, W. H., Pulleyblank, W. R., Schrijver, A. (1998) Combinatorial Optimization. John Wiley & Sons
19. Coffman, E. G., Graham, R. L. (1972) Optimal scheduling for two processor systems. Acta Informatica **1**, 200–213
20. Cruz, I. F., and Twarog, J. P. (1996) 3D graph drawing with simulated annealing. In: F.-J. Brandenburg (ed.) Graph Drawing '95, Lecture Notes in Computer Science 1027, Springer-Verlag, 162–165
21. Dahlhaus, E. (1998) A linear time algorithm to recognize clustered graphs and its parallelization. In: C. L. Lucchesi and A. V. Moura (eds.) Latin '98, Lecture Notes in Computer Science 1380, Springer-Verlag, 239–248
22. Davidson, R., Harel, D. (1996) Drawing graphs nicely using simulated annealing. ACM Transactions on Graphics **15**, 301–331
23. Di Battista, G., Eades, P., Tamassia, R., Tollis, I. G. (1999) Graph Drawing: Algorithms for the visualization of graphs. Prentice Hall, New Jersey
24. Di Battista, G., Garg, A., Liotta, G., Parise, A., Tamassia, R., Tassinari, E., Vargiu, F., Vismara, L. (1997) Drawing Directed Graphs: an Experimental Study. In: S. North (ed.) Graph Drawing '96, Lecture Notes in Computer Science 1190, Springer-Verlag, 76–91
25. Di Battista, G., Garg, A., Liotta, G., Tamassia, R., Tassinari, E., Vargiu, F. (1997) Computational Geometry: Theory and Applications **7**, 303–316
26. Di Battista, G., Didimo, W., Patrignani, M., Pizzonia M. (1999) Orthogonal and quasi-upward drawings with vertices of arbitrary size. In: J. Kratochvíl (ed.) Graph Drawing '99, Lecture Notes in Computer Science 1731, Springer-Verlag, 297–310

27. Djidjev, H. N. (1995) A linear algorithm for the maximal planar subgraph problem. In: Proceedings of the 4th Workshop Algorithms Data Struct., Lecture Notes in Computer Science, Springer-Verlag
28. Eades, P. (1984) A heuristic for graph drawing. Congressus Numerantium **42**, 149–160
29. Eades, P. (1992) Drawing free trees. Bulletin of the Institute for Combinatorics and its Applications **5**, 10–36
30. Eades, P., Lin, X., Smyth, W. F. (1993) A fast and effective heuristic for the feedback arc set problem. Information Processing Letters **47**, 319–323
31. Eades, P., Wormald, N. (1994) Edge crossings in drawings of bipartite graphs. Algorithmica **11**, 379–403
32. Eades, P., Cohen, R. F., Huang, M. L. (1997) Online animated graph drawing for web animation. In: G. Di Battista (ed.) Graph Drawing '97, Lecture Notes in Computer Science 1353, Springer-Verlag, 330–335
33. Elf, M., Gutwenger, C., Jünger, M., Rinaldi, G. (2001) Branch-and-cut algorithms and their implementation in ABACUS. In: M. Jünger and D. Naddef (eds.) Computational Combinatorial Optimization, Lecture Notes in Computer Science 2241, Springer-Verlag, 157–222
34. Euler, L. (1750) Demonstratio nonnullarum insignium proprietatum quibus solida hedris planis inclusa sunt praedita. Novi Comm. Acad. Sci. Imp. Petropol. **4** (1752-3, published 1758), 140–160, also: Opera Omnia (1) **26**, 94–108
35. Feng, Q. W., Cohen, R. F., Eades, P. (1995) Planarity for clustered graphs. In: P. Spirakis (ed.) Algorithms – ESA '95, Lecture Notes in Computer Science 979, Springer-Verlag, 213–226
36. Fößmeier, U., Kaufmann, M. (1996) Drawing high degree graphs with low bend numbers. In: F. J. Brandenburg (ed.) Graph Drawing '95, Lecture Notes in Computer Science 1027, Springer-Verlag, 254–266
37. Frick, A., Ludwig, A., Mehldau, H. (1995) A fast adaptive layout algorithm for undirected graphs. In: R. Tamassia and I. G. Tollis (eds.) Graph Drawing '94, Lecture Notes in Computer Science 894, Springer-Verlag, 388–403
38. Fruchtermann, T. M. J., Reingold, E. M. (1991) Graph drawing by force-directed placement. Software – Practice and Experience **21**, 1129–1164
39. Garey, M. R., Johnson, D. S. (1983) Crossing number is NP-complete. SIAM J. Algebraic Discrete Methods **4**, 312–316
40. Gansner, E. R., Koutsofios, E., North, S. C., Vo, K. P. (1993) A technique for drawing directed graphs. IEEE Transactions on Software Engineering **19**, 214–230
41. Garg, A., Tamassia, R. (1995) On the computational complexity of upward and rectilinear planarity testing. In: R. Tamassia and I. G. Tollis (eds.) Graph Drawing '94, Lecture Notes in Computer Science 894, Springer-Verlag, 286–297
42. Garg, A., Tamassia, R. (1997) A New Minimum Cost Flow Algorithm with Applications to Graph Drawing. In: S. North (ed.) Graph Drawing '96, Lecture Notes in Computer Science 1190, Springer-Verlag, 201–216
43. Gutwenger, C., Mutzel, P., Weiskircher, R. (2001) Inserting an edge into a planar graph. In: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001), ACM Press, 246-255
44. Healy, P., Kuusik, A. (1999) The vertex-exchange graph: a new concept for multi-level crossing minimization. In: J. Kratochvíl (ed.) Graph Drawing '99, Lecture Notes in Computer Science 1731, Springer-Verlag, 205–216

45. Hopcroft, J., Tarjan, R. E. (1974) Efficient planarity testing. Journal of the ACM **21**, 549–568
46. Jayakumar, R., Thulasiraman, K., Swamy, M. N. S. (1989) $O(n^2)$ algorithms for graph planarization. IEEE Transactions on Computer Aided Design **8**, 257–267
47. Jünger, M., Leipert, S., Mutzel, P. (1998) A note on computing a maximal planar subgraph using PQ-trees. IEEE Transactions of Computer-Aided Design and Integrated Circuits and Systems **17**, 609–612
48. Jünger, M., Mutzel, P. (1996) Maximum planar subgraphs and nice embeddings: practical layout tools. Algorithmica **16**, 33–59
49. Jünger, M., Mutzel, P. (1997) 2-layer straight line crossing minimization: performance of exact and heuristic algorithms. Journal of Graph Algorithms and Applications **1**, 1–25
50. Jünger, M., Reinelt, G., Thienel, S. (1995) Practical Problem Solving with Cutting Plane Algorithms in Combinatorial Optimization. In: W. Cook, L. Lovász, P. Seymour (eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 111-152
51. Kamada, T., and Kawai, S. (1989) An algorithm for drawing general undirected graphs. Information Processing Letters **31**, 7–15
52. Kaufmann, M., Wagner, D. (eds.) (2001) Drawing Graphs: Methods and Models. Lecture Notes in Computer Science 2025, Springer-Verlag
53. Klau, G. W., Klein, K., Mutzel P. (2001) An Experimental Comparison of Orthogonal Compaction Algorithms. In: J. Marks (ed.) Graph Drawing '00, Lecture Notes in Computer Science 1984, Springer-Verlag, 37–51
54. Klau, G. W., Mutzel P. (1999) Optimal compaction of orthogonal grid drawings. In: G. Cornuejols, R. E. Burkard, and G. J. Woeginger (eds.), Integer Programming and Combinatorial Optimization (IPCO '99), Lecture Notes in Computer Science 1610, Springer-Verlag, 304–319
55. Lempel, A., Even, S., Cederbaum, I. (1967) An algorithm for planarity testing of graphs. Theory of Graphs: International Symposium: Rome, July 1966, Gordon and Breach, New York, 215–232
56. Liu, P.C., Geldmacher, R. C. (1977) On the deletion of nonplanar edges of a graph. Proceedings of the 10th S-E Conference on Comb., Graph Theory, and Comp., Boca Raton, FL, 727–738
57. Mehlhorn K., Mutzel, P. (1996) On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. Algorithmica **16**, 233–242
58. Mutzel, P., Weiskircher, R. (2002) Bend Minimization in Orthogonal Drawings Using Integer Programming. In: O. Ibarra and L. Zhang (eds.) Computing and Combinatorics, Eighth Annual International Conference (COCOON 2002), Lecture Notes in Computer Science 2387, Springer-Verlag, 484-493
59. Patrignani, M. (2001) On the complexity of orthogonal compaction. Computational Geometry: Theory and Applications **19** (1), 47–67
60. Reingold, E., Tilford, J. (1981) Tidier drawing of trees. IEEE Transactions on Software Engineering **7**, 223–228
61. Sander, G. (1996) Visualisierungstechniken für den Compilerbau. Pirrot Verlag & Druck, Saarbrücken
62. Sugiyama, K., Tagawa, S.,Toda, M. (1981) Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics **11**, 109–125

63. Sugiyama, K., Misue, K. (1991) Visualization of structural information: automatic drawing of compound digraphs. IEEE Transactions on Systems, Man, and Cybernetics **4** (21), 876–893
64. Sugiyama, K., Misue, K. (1995) A simple and unified method for drawing graphs: magnetic-spring algorithm. In: R. Tamassia and I. G. Tollis (eds.) Graph Drawing '94, Lecture Notes in Computer Science 894, Springer-Verlag, 364–375
65. Supowit, K. J., Reingold, E. M. (1983) The complexity of drawing trees nicely. Acta Inform. **18**, 377–392
66. Tamassia, R. (1987) On embedding a graph in the grid with the minimum number of bends. SIAM Journal on Computing **16** (3), 421–444
67. Tamassia, R., Di Battista, G., Batini, C. (1988) Automatic graph drawing and readability of diagrams. IEEE Transactions on Systems, Man, and Cybernetics **18**, 61–79
68. Tutte, W. T. (1963) How to draw a graph. Proceedings of the London Mathematical Society, Third Series **13**, 743–768
69. Waddle, V., Malhotra, A. (1999) An E log E line crossing algorithm for levelled graphs. In: J. Kratochvíl (ed.) Graph Drawing '99, Lecture Notes in Computer Science 1731, Springer-Verlag, 59–70
70. Walker II, J. Q. (1990) A node-positioning algorithm for general trees. Software – Practice and Experience **20**, 685–705
71. Ziegler, T. (2001) Crossing minimization in automatic graph drawing. Doctoral Thesis, Technische Fakultät der Universität des Saarlandes

http://www.springer.com/978-3-540-00881-1

Graph Drawing Software
Jünger, M.; Mutzel, P. (Eds.)
2004, XII, 378 p., Hardcover
ISBN: 978-3-540-00881-1