13

Hierarchical Drawing Algorithms

Patrick Healy
University of Limerick

Nikola S. Nikolov
University of Limerick

13.1 Introduction

In many cases a directed graph represents a hierarchy and we want to draw it in this way. We will define a hierarchy later, but for now it is sufficient to think of a hierarchy as a cycle-free digraph where it is useful for nodes of the graph to be stratified into discrete, parallel layers. Examples of hierarchies or near-hierarchies are, among others, PERT charts for project management, object-oriented class diagrams, and function call graphs from software engineering. As usual, nodes represent entities and edges represent relationships between the entities. Closely related to hierarchically layered drawings and discussed later are radial drawings where nodes are placed on concentric circles [DDLM04, Bac07] and cyclic level drawings, where nodes are placed on "spokes" emanating from a centre-point [BBBF12].

What is common to all of the examples described above is the need to represent all the relationships graphically so that the positioning of nodes are as consistent with the transitivity of the relationship as can be achieved. That is, the edges should "flow" in a uniform direction. Whether this direction should be top-to-bottom, or left-to-right, depends on the application domain, with different disciplines having different preferences.

Di Battista *et al.* [DGL$^+$00] conduct an experimental study of directed graph drawing algorithms. They look at two broad categories of algorithms, those that provide *layered* *drawings* and those that are *grid-based.* From the point of view of edge crossings, an important aspect in the readability of graph drawings [PCJ96], the hierarchical or layered approach performs better, they conclude.

For digraphs that are *almost* a hierarchy it still can be possible to take advantage of the methods we describe in this chapter. Since the methods work best for hierarchical digraphs, however, one may find the results disappointing when applied to a digraph that is fundamentally non-hierarchical. A method for computing the *hierarchical index* – that is, the amount of hierarchy in a directed graph – has been proposed [CHK02].

In addition to the applications we mentioned earlier, it is common and appropriate to represent computer file systems and social networks as a graph. Due to the possibility of symbolic links a graph representation of a computer file system may have directed cycles, and so may be more complicated than a tree. Likewise, a graph representation of relationships among a social network graph may have cycles.

In the following sections we describe the current approaches to drawing directed graphs hierarchically and consider in more depth the dominant player, the Sugiyama framework for drawing digraphs. It is perhaps a measure of its effectiveness and its success that the method has been used to draw undirected graphs, by imposing orientations on the edges. However, as we have mentioned earlier, results may be disappointing if care is not taken in how orientations of edges are fixed.

13.1.1 Current Approaches and Their Limitations

Force-directed methods, discussed elsewhere in this handbook, can be modified to take account of edge directions, and thus can be used to draw digraphs. Sugiyama and Misue [SM95b] propose modifications of Eades' spring embedder model [Ead84] to take account of the possible directedness of edges.

Far and away the most popular method of drawing directed graphs is the *Sugiyama method*, or *Sugiyama framework* [STT81], which separates the nodes into layers. The idea of layered drawings can be traced back earlier to work by Warfield [War77] and Carpano [Car80]. Systems such as *da Vinci* [FW95], *dot* [GKN02] (part of the `GraphViz` suite of tools [GN00]), *GraphLet* [Him00], the AGD graph drawing library [NPT90] and others implement this framework for drawing directed graphs. In testament to its popularity many modifications and enhancements have been proposed in the literature. However, the framework has its limitations. Figure 13.1 shows two drawings of C_4, firstly drawn using force-directed methods and secondly drawn using the Sugiyama framework. In spite of the directed edge entering node a, by imposing a leveling Figure 13.1b suggests that node d is inferior to the others.

In the following section we describe the Sugiyama framework in general terms. In subsequent sections we consider the framework in more detail, elaborating on issues specific to each step of the framework. It should be noted at the outset that this is only a framework and for many steps of the process alternative algorithms exist, each with their own merits. Equally important is the fact that the steps may interact with each other and a solution to one step can have a bearing on later steps.

13.1.2 Overview of Sugiyama's Framework

The Sugiyama framework is motivated by a number of aesthetically desirable properties that make for a more readable graph. Indeed the steps of the Sugiyama framework can be seen to address, in turn, each of the following aesthetics.

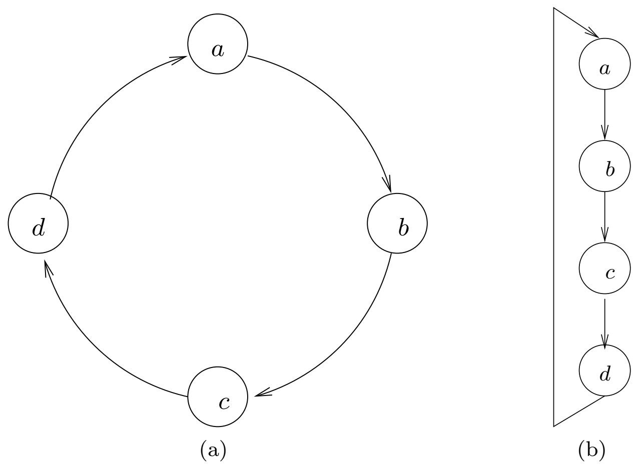


Figure 13.1 Two alternative drawings of C_4.

- Edges should point in a uniform direction
- Short edges are more readable
- Uniformly distributed nodes avoid clutter
- Edge crossings obstruct comprehension
- Straight edges are more readable

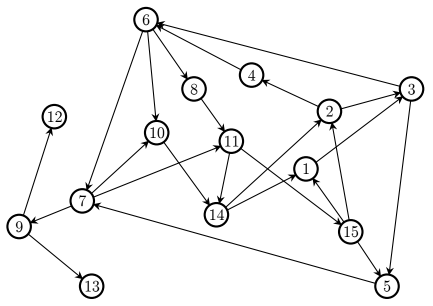
Figure 13.2 demonstrates how these aesthetics are achieved on a typical directed graph[1] G. So that all edges are directed uniformly, any directed cycles are broken by reversing a subset of edges (see Figure 13.2b). The resulting graph is then *leveled* (Figure 13.2c) where, through the introduction of *dummy vertices*, "long" edges are replaced by a series of shorter segments, after which vertices on each level are permuted in order to reduce edge crossings (Figure 13.2d). Finally, edges spanning more than one level (*long* edges) are straightened by adjusting the x-coordinates of their end vertices and by aligning the inserted dummy nodes on long edges.

As we have remarked before the Sugiyama framework draws directed graphs as layers of vertices. We have loosely described a hierarchy in terms of layers of nodes and this is part of its formal definition, also. Definition 13.1 formalizes a level graph, and the definition of a *hierarchy* follows from this.

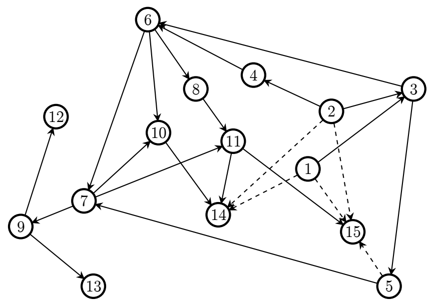
DEFINITION 13.1 A *level graph* $G = (V, E, \lambda)$ is a directed acyclic graph with a mapping $\lambda : V \to \{1, 2, \ldots, k\}$, $k \geq 1$, that partitions the vertex set V as $V = V_1 \cup V_2 \cup \cdots \cup V_k$, $V_j = \lambda^{-1}(j)$, $V_i \cap V_j = \emptyset$ for $i \neq j$, such that $\lambda(v) = \lambda(u) + 1$ for each edge $(u, v) \in E$.

DEFINITION 13.2 A *hierarchy* is a level graph $G(V, E, \lambda)$ where for every $v \in V_j,\ j > 1$, there exists at least one edge (w, v) such that $w \in V_{j-1}$.

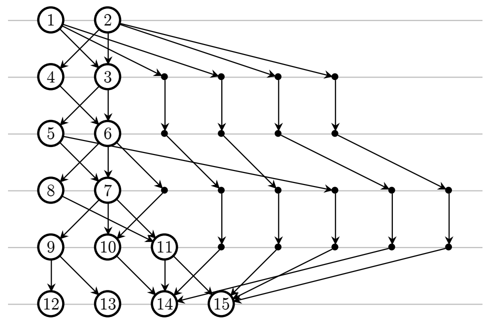
[1]The figure is due to Bachmaier *et al.* [BBBF12]; the authors' permission to reproduce the figure is gratefully acknowledged.



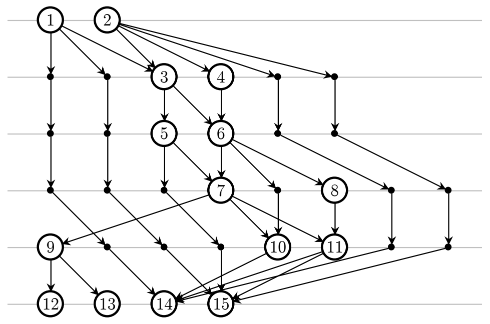
(a) Input digraph, G.



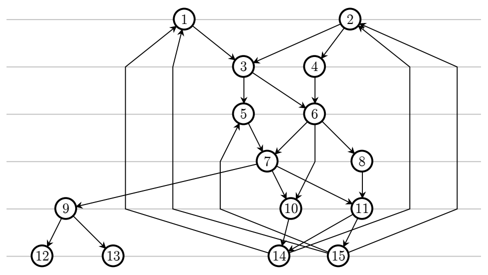
(b) Cycles removed.



(c) After leveling.



(d) Edge crossings minimized.



(e) Edges straightened.

Figure 13.2 A digraph drawn according to the Sugiyama framework.

Note that Definition 13.2 restricts all sources of the graph to appear on the first level but this may be relaxed if desired. Further, the definition implies that all edges are of unit length, a property that is necessary for the *crossing minimization* step. While an input digraph may not be a hierarchy initially, the steps described in the following subsections will transform it into an equivalent hierarchy.

13.2 Cycle Removal

The first step of the Sugiyama method is a preprocessing step that aims the reversal of the direction of some edges in order to make the input digraph acyclic. A digraph is acyclic if it does not contain any directed cycles. Note that the digraph may have undirected cycles and be acyclic. It is usually assumed that the input digraph has no *two-cycles*. A two-cycle is a cycle consisting of a pair of edges (u, v) and (v, u). If any are present then one edge of each pair can be removed before applying the Sugiyama method and reintroduced back into the final drawing.

The cycle-removal preprocessing step is necessary because the input to the the layer-assignment step must be an acyclic digraph, also called a DAG (directed acyclic graph). Once vertices are assigned to layers, the original direction of the reversed edges can be restored. These are edges which point against the flow in the final drawing. It is also possible to remove edges instead of reversing them, and introduce them back after the layer-assignment step. However, if edges are removed then the layer-assignment step will work with a subgraph of the input digraph and may have undesirable results.

A set of edges whose removal makes the digraph acyclic is commonly known as a *feedback arc set* (FAS). Following the terminology used by Di Battista *et al.* [DETT99] we call a set of edges whose reversal makes the digraph acyclic a *feedback set* (FS). Each FS is also a FAS. However, not each FAS is a FS. For example, if a digraph has only one cycle, then the set of all edges in the cycle is a FAS but not a FS.

It is always possible and easy to find a FS for a digraph. Any linear ordering of the vertices partitions the edge set into two subsets, a subset of edges whose source is before their target in the ordering, and a subset of edges edges whose source is after their target in the ordering. Each of the two subsets is a FS. However, it might be much harder to find a FS with some specific properties.

A typical requirement for a FS is to contain as few edges as possible because they are the edges against the flow in the final drawing. The problem of finding a minimum-cardinality FS is known as the *minimum FS problem*. As we mentioned above not every FAS is a FS. However, it is easy to see that every minimal cardinality FAS is also a FS. Thus, the minimum FS problem is as hard as the widely studied *minimum FAS problem* which is known to be NP-hard [Kar72, GJ79].

Any heuristic for solving the minimum FAS problem can be applied for solving the minimum FS problem as well. Consider a digraph $G = (V, E)$ and let $F \subseteq E$ be a FAS. F is a minimal FAS if for each $e \in F$ there is a cycle in $(E \setminus F) \cup \{e\}$. If F is not minimal then one by one we can remove edges from it until it becomes minimal. However, such a procedure will add additional running time.

The remainder of this section summarizes the known heuristics for solving either the minimum FS problem or the minimum FAS problem. Some of the FAS heuristics have been originally proposed as heuristic for solving its complimentary problem, i.e., the maximum acyclic subgraph problem.

13.2.1 Heuristics Based on Vertex Orderings

As we mentioned above any linear ordering of the vertices provides two FSs. Imagine the vertices placed on a horizontal line in accordance with the provided linear ordering. The first FS consists of the edges with direction from left to right, and the second FS consists of the edges with direction from right to left. It can be shown that there are digraphs and linear orderings of the vertices for which both FSs have the same size [BS90]. Thus, the simple approach is a 2-approximation heuristic for solving both the minimum FS and FAS problems.

It is clear that the outcome of this simple approach depends on the linear ordering of the vertices. Some researchers have suggested to use a linear ordering provided by a depth-first traversal [RDM$^+$87, GKNV93]. This is based on the intuition that such an ordering may be "natural" for the digraph *i.e.*, it may look "natural" to the viewer that the edges against the flow are drawn this way. A depth-first-traversal ordering can be computed in linear time. Here it is assumed that always the set of edges with direction to the left, i.e., against the ordering, are chosen as FSs. These are the back edges in the depth-first-traversal tree. Their number can be at most $|E| - |V| - 1$ which could be high for dense digraphs.

Eades *et al.* proposed two alternative linear orderings [ELS89]. The first one is based on the intuition that vertices with large outdegree should appear at the top of the final drawing. First in the ordering comes vertex v with the maximum $d^+_G(v)$, next comes vertex v' with the maximum $d^+_{G-v}(v')$, etc. This linear ordering can be computed in linear time and reportedly gives smaller FSs than the ordering provided by the depth-first search.

The second linear ordering proposed by Eades at al. is the result of a divide-and-conquer approach. Assuming an input digraph $G = (V, E)$ the vertices of which have to be assigned the labels $i, i+1, \ldots, i+|V|-1$, the recursive procedure for assigning these labels works as follows. If $|V| = 1$ then the single vertex gets the label i. Otherwise V is partitioned into two subsets V_1 and V_2 and the procedure is applied recursively to $G[V_1]$ and $G[V_2]$ with sets of labels $i, i+1, \ldots, i+|V_1|-1$ and $i+|V_1|, i+|V_1|+1, \ldots, i+|V|-1$, respectively. V_1 and V_2 are such that for each pair of vertices $(v_1, v_2) \in V_1 \times V_2$ $d^+_G(v_1) \geq d^+_G(v_2)$. If $|V|$ is even then V_1 and V_2 have the same cardinality. Otherwise, V_1 contains a singe vertex, and V_2 contains the rest of the vertices. It takes $O(min((|V| + |A|)log|V|, |V|^2))$ time to compute this linear ordering. However, the the authors have observed that it regularly obtains results 20% better than the results obtained by their other ordering. They also prove performance guarantees for dense digraphs.

13.2.2 Berger-Shor Algorithm

The first polynomial-time algorithm for solving the minimum FAS problem with an approximation ratio less than 2 in the worst case is the algorithm proposed by Berger and Shor in 1990 [BS90].

Consider a digraph $G = (V, E)$ and let $E_a \subset E$ denote a set of edges such that $G[E_a]$ is acyclic. Let also $\delta(v)$ denote all edges adjacent to vertex v, and $\delta^-(v)$ and $\delta^+(v)$ denote the sets of incoming and outgoing edges of v, respectively. The algorithm starts with an empty set E_a and one by one scans all vertices of G, in an arbitrary order. For each vertex $v \in V$ if $d^+(v) \geq d^-(v)$ then $E_a \leftarrow E_a \cup \delta^+(v)$. Otherwise, $E_a \leftarrow E_a \cup \delta^-(v)$. After processing vertex v it is deleted from G (together with its adjacent edges).

The time complexity of this algorithm is $O(|V| + |E|)$. Berger and Shor prove that $G' = (V, E_a)$ is a DAG and thus $F = E \backslash E_a$ is a FAS. They also propose a modification to the algorithm for making sure the FAS is minimal. It consists of running a strongly connected components algorithm before processing each vertex, adding the edges between strongly

Figure 13.3 Greedy Cycle Removal

Require: digraph $G = (V, E)$

```
  $S_l \leftarrow \phi$
  $S_r \leftarrow \phi$
  while $G$ is not empty do
    while $G$ contains a sink do
      Choose a sink $v$
      Remove $v$ from $G$
      Prepend $v$ to $S_r$
    end while
    while $G$ contains a source do
      Choose a source $u$
      Remove $u$ from $G$
      Append $u$ to $S_l$
    end while
    if $G$ is not empty then
      Choose a vertex $w$ such that $d^+(w) - d^-(w)$ is maximum
      Remove $w$ from $G$
      Append $w$ it to $S_l$
    end if
  end while
```

connected components to E_a, and removing them from E. However, this modification adds $O(|V||E|)$ to the running time.

It is easy to see that at the end F will contain at most a half of all the edges, i.e., $|F| \leq \frac{|E|}{2}$. Better approximation ratio can be achieved by processing the vertices in a special order. Berger and Shor show that if at each step the vertex to be processed is the vertex that minimizes the number of edges in the FAS over all possible orderings of the unprocessed yet vertices then at the end $|F| \leq |E|(\frac{1}{2} - \Omega(\frac{1}{\sqrt{\Delta(G)}}))$ where $\Delta(G)$ is the maximum degree of a vertex in G. It is possible to process the vertices in such an order efficiently by increasing the running time to $O(|V||E|)$ [BS90].

13.2.3 Greedy Cycle Removal

It can be observed that edges incident to either a sink or a source of the digraph cannot be a part of a cycle. By making this observation Eades *et al.* were able to improve the result of Berger and Shor [ELS93]. They proposed a linear-time algorithm with a performance guarantee at least as good as the performance guarantee of the Berger-Shor algorithm for $|E| \in O(|V|)$ and even better if $\Delta(G) \notin O(1)$ [ELS93]. The algorithm of Eades *et al.*, known as Greedy Cycle Removal, always finds a FS for the input digraph.

Similar to the approach of Berger and Shor, Algorithm 13.3 processes the vertices one by one and removes the processed vertices from the digraph. However, it builds the FAS in a different way. It computes a linear ordering of the vertices and takes the edges with direction against the ordering as an FS. Eades *et al.* show that the cardinality of the FS found by Algorithm 13.3 is at most $\frac{|E|}{2} - \frac{|V|}{6}$. For digraphs with $\Delta(G) \leq 3$ the cardinality is at most $\frac{2}{3}|E|$.

Sander has proposed a modification to Algorithm 13.3 similar to the modification which involves the computation of strongly connected components in the Berger-Shor algorithm [San96b]. It decreases the running time to $O(|V||E|)$ but reportedly leads to better results in practice. Generalizing the ideas of Sander, Eades and Lin showed how Algorithm 13.3 can be modified to guarantee that no more that one quarter of the edges will be reversed for cubic digraphs [EL95].

13.2.4 Heuristics Based on Cycle Breaking

Most of the heuristics discussed above focus on computing linear orderings of the vertices which provide FSs. Alternative point of view the minimum FAS problem (and also the minimum FS problem) is to build the FAS edge by edge choosing edges which belong to cycles.

A very simple algorithm based on this approach is the following one. Start with two empty sets S and T and scan all edges one by one. For each edge e, if $S \cup \{e\}$ is acyclic then add e to S. Otherwise add e to T. It is easy to show that at the end of this process both S and T are acyclic and the smaller of the two sets provides a FAS with at most a half of all the edges. Note that T is a minimal FAS, while S might not be.

Related to this approach is the heuristic implemented by Gansner *et al.* in their system `dot` [GKNV93]. It takes one non-trivial strongly connected component of the digraph at a time, in an arbitrary order. Within each component it performs a depth-first traversal and adds to the FS an edge which participates in a maximum number of cycles. This is repeated until there are no more non-trivial strongly connected components. Gansner *et al.* report that this heuristic performs well in practice. They also observed that it reverses edges whose direction against the flow is "natural" for the input digraph [GKNV93].

13.2.5 Minimum FAS in a Weighted Digraph

In some applications edges are assigned nonnegative weight. Then it might be required to reverse not the minimum number of edges but a set of edges with the minimum total weight. Demetrescu and Finocchi proposed an algorithm for the weighted minimum FAS problem that runs in $O(|V||E|)$ time. Their approach compromises between two possible approaches, i.e., greedily adding light edges to the FAS, and adding edges that belong to a large number of cycles. The latter is the approach of Gansner *et al.*, which we discussed in Section 13.2.4.

The algorithm of Demetrescu and Finocchi is presented as Algorithm 13.4. Within the `while`-loop heavy edges which belong to a large number of cycles become progressively more likely to be added to the FAS F. Note also that at the end edges which do not form a cycle with $E \setminus F$ are excluded from F, thus making sure F is minimal. Demetrescu and Finocchi also prove that their algorithm approximates a minimum FAS of the input digraph G within a ratio bounded by the length of a longest simple cycle of G [DF03].

13.2.6 Other Approaches

There are other approaches to the minimum FAS problem which we would like to refer the reader to. These include the heuristic of Flood [Flo90], and the best-known approximation algorithm which achieves a performance ratio $O(\log |V| \log\log |V|)$, and requires to solve a linear program [ENRS95, Sey95]. An interesting result is that all minimal solutions can be enumerated with polynomial delay [SS97].

Figure 13.4 Algorithm of Demetrescu and Finocchi

Require: digraph $G = (V, E)$, $w : E \to \mathcal{R}$

```
$F \leftarrow \phi$
while $V, E \setminus F)$ is not acyclic do
  Let $C$ be a simple cycle in $(V, E \setminus F)$
  Let $(x, y)$ be a minimum weight edge in $C$
  Let $\epsilon = w(x, y)$
  for all $(u, v) \in C$ do
    $w(u, v) \leftarrow w(u, v) - \epsilon$
    if $w(u, v) = 0$ then
      $F \leftarrow F \cup \{(u, v)\}$
    end if
  end for
end while
for all $(u, v) \in F$ do
  if $(V, E \setminus F \cup \{(u, v)\})$ is acyclic then
    $F \leftarrow F \setminus \{(u, v)\}$
  end if
end for
```

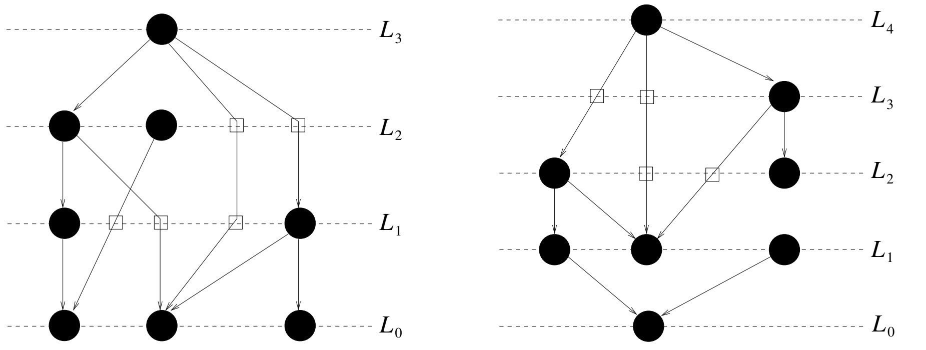
For an exact ILP approach to the minimum FAS problem we refer the reader to the work of Grötschel *et al.* and Rienelt *et al.* [GJR85, Rei85]. Their study of the facial structure of the acyclic subgraph polytope can be used for finding the minimum FS by a branch-and-cut algorithm.

13.3 Layer Assignment

Consider a DAG $G = (V, E)$ with a set of vertices V and a set of edges E. Let $\mathcal{L} = \{L_0, L_1, \ldots, L_h\}$ be a partition of the vertex set of G into $h \geq 1$ subsets such that if $(u, v) \in E$ with $u \in L_j$ and $v \in L_i$ then $i < j$. $\mathcal{L}$ is called a *layering* of G and the sets L_0, L_1, ..., L_h are called *layers.* A DAG with a layering is called a *layered* DAG. The problem of partitioning the vertex set of a graph into layers is known as *the layering problem* or *the layer assignment problem.*

Sometimes the term *levels* is used instead of layers. It emphasizes the usual visual representation of layers as mapped to either parallel horizontal lines or concentric circles. In this section we consider the layer assignment problem without relating it to a specific visual representation. The example drawings we give have only illustrative character. They employ the parallel horizontal levels convention, i.e., all vertices in layer L_i are placed on the horizontal level with an y-coordinate equal to i.

Let $l(u, \mathcal{L})$ be the number of the layer that contains vertex $u \in V$, i.e., $l(u, \mathcal{L}) = i$ if and only if $u \in L_i$. Sometimes $l(u, \mathcal{L})$ is called *rank* of vertex u. The *span* of edge $e = (u, v)$ in layering $\mathcal{L}$ is defined as $s(e, \mathcal{L}) = l(u, \mathcal{L}) - l(v, \mathcal{L})$. Clearly, $s(e, \mathcal{L}) \geq 1$ for each $e \in E$; edges with a span 1 are *tight* edges; edges with a span greater than 1 are *long edges.* A layering of G is *proper* if all edges are tight. The layering found by a layering algorithm might not be proper because only a small fraction of DAGs can be layered properly and also because a proper layering may not satisfy other layering requirements.



(a) A layered DAG with 4 layers and 5 dummy vertices.

(b) A layered DAG with 5 layers and 4 dummy vertices.

Figure 13.5 Two alternative layered drawings of the same DAG with introduced dummy vertices which subdivide long edges. Dummy vertices are represented by transparent squares.

Within the Sugiyama method the vertex ordering algorithms applied after the layer assignment phase assume that their input is a DAG with a proper layering. Thus, if the layering found at the layering phase is not proper then it must be transformed into a proper one. Normally, this is done by introducing so-called *dummy vertices* which subdivide long edges (see the illustration in Figure 13.5). Formally, let $e = (u, v)$ be an edge with $l(u, \mathcal{L}) = j$ and $l(v, \mathcal{L}) = i$ and $s(e, \mathcal{L}) = j - i > 1$. Then we add dummy vertices d_e^{i+1}, d_e^{i+2}, $\ldots$, d_e^{j-1} to layers L_{i+1}, L_{i+2}, $\ldots$, L_{j-1} respectively and we replace edge e by the path $(u, d_e^{j-1}, \ldots, d_e^{i+1}, v)$. We refer to vertices which are not dummy as *original* vertices. We also denote the set of all dummy vertices introduced to a layered DAG G with a layering $\mathcal{L}$ by $\mathcal{D}(G, \mathcal{L})$. Clearly,

$$|\mathcal{D}(G, \mathcal{L})| = \sum_{e \in E} s(e, \mathcal{L}) - |E|.$$

13.3.1 Additional Criteria and Variations of the Problem

If there are no additional requirements it is not hard to find a layering of a DAG. Classical graph algorithms such as breadth-first search, depth-first search and algorithms for finding a minimum spanning tree can be easily modified to partition the vertex set of a DAG into layers. However, normally it is desirable to take into account a number of additional criteria when computing a layering [ES90].

It is desirable that $|\mathcal{D}(G, \mathcal{L})|$ is as small as possible because a large number of dummy vertices significantly slows down the vertex ordering phase of the Sugiyama method. Thus, one of the goals of a layering assignment algorithm should be to find a layering with as few as possible dummy vertices. There are also aesthetic reasons for keeping the number of dummy vertices small. A layered DAG with a small number of dummy vertices would also have a small number of undesirable long edges and edge bends. The problem of finding a layering with the minimum number of dummy vertices is in P. It can be modeled as an integer linear programming problem and safely relaxed to a linear programming problem for which there are available polynomial-time algorithms. Alternatively, it can be converted to a min-cost flow or circulation problem, for which there are polynomial-time algorithms [GKNV93]. Gansner *et al.* have introduced an integer linear programming (ILP) model of the problem and a specific network simplex algorithm for solving it which although not proven polynomial-time finds a layering with the minimum number of dummy vertices reportedly fast [GKNV93].

Other parameters of a layering that reflect on the quality of the drawing are the width and the height of a layering and the edge density between adjacent layers. The *height* of a layering is the number of layers, and the *width* is the maximum number of vertices in a layer. Usually these two parameters are used to approximate the dimensions of the final drawing. When measuring the width of a layering the contribution of the dummy vertices may or may not be taken into account. A more precise definition of the layering width takes into account both variable vertex widths and the contribution of the dummy vertices [BLME02, HN02a]. The *area* of a layering, used to approximate the area of the final drawing, is defined as the product of the layering width and the layering height.

A layering with the minimum height can be found in linear time by the longest-path algorithm [ES90]. It is also easy to find a layering with the minimum number of dummy vertices subject to an upper bound on the number of layers. The ILP model of Gansner *et al.* can be easily modified to take such an upper bound into account without making it more difficult to solve.

It is trivial to find a layering with the minimum number of original vertices per layer and no upper bound on the height. Any layering with a single vertex per layer is an optimal solution. However, it is NP-hard to find a layering with a given upper bound on the width if the width of the dummy vertices is considered greater than zero [BLME02]. A few heuristics have emerged since for layering with the minimum width and consideration of dummy vertices [BELM01, TNB04].

It is also NP-hard to find a layering with given upper bounds both on the height and on the width even without taking into account the contribution of the dummy vertices to the width and if all original vertices have the same unit width [ES90]. This variation of the layer assignment problem is equivalent to the precedence-constrained multiprocessor scheduling (PCMS) problem. The Coffman-Graham algorithm, which is an early and highly influential polynomial-time algorithm for solving PCMS approximately, has been also largely employed as a layering algorithm [CG72]. Healy and Nikolov have designed a branch-and-cut algorithm which finds layerings with the minimum number of dummy vertices and with height and width within pre-specified upper bounds [HN02a]. It is not a polynomial-time algorithm but it is an efficient way to find an optimal solution if the problem is not infeasible. The branch-and-cut algorithm of Healy and Nikolov takes into account variable vertex widths as well as the contribution of the dummy vertices to the width of the layering.

The *edge density* between layers L_i and L_j with $i < j$ is defined as the number of edges (u, v) with $u \in L_j \cup L_{j+1} \cup \ldots \cup L_h$ and $v \in L_0 \cup L_1 \cup \ldots \cup L_i$. The edge density of a layered DAG is the maximum edge density between adjacent layers. Naturally, drawings with low maximum and average edge density are clearer and easier to comprehend provided they are also compact and having not too many long edges. There has not been proposed any algorithm which find layerings specifically with consideration of edge density. A few studies have compared the edge density in layerings found by some of the algorithms mentioned above [HN02b, NT06, TNB04].

In the next section we take a closer look at the algorithms used for layer assignment.

13.3.2 Layer Assignment Algorithms

The easiest way to partition the vertex set of a DAG into layers is to take any spanning tree of the underlying undirected graph. It can be the depth first search tree or the breadth first search tree, for example. For each spanning tree a layering can be generated by picking a vertex and assigning it to an arbitrary layer, L_i. Then assign all its neighbours in the tree to either layer L_{i-1} or L_{i+1} depending on the direction of the connecting tree-edge. Then repeat the same with all neighbours of the already assigned vertices and so on until all vertices are assigned to a layer. It cannot be guaranteed that the set of layers will start with L_1 but this can be easily fixed by shifting the whole set of layers.

In general, this method does not guarantee any properties of the layering. The layer assignment algorithms, which find layerings subject to some of the criteria discussed in the previous section, broadly fall into two groups. The first group of algorithms are adopted from the area of static precedence-constrained multiprocessor scheduling. They produce layerings with either the minimum height or a specified maximum number of vertices per layer. The second group of algorithms employ network simplex and branch-and-cut techniques, respectively, for minimizing the number of dummy vertices.

List Scheduling Algorithms

The precedence-constrained multiprocessor scheduling problem is the problem of scheduling n causally related tasks (which represent a parallel program) on m processors with the goal of minimizing the completion time of the parallel program. This problem is NP-hard when $m < \infty$ [Ulm75]. It is also known as *static scheduling* because all the tasks with their causal relationship are given in advance and the schedule must be constructed prior executing any of them [KA99]. A simplified version of this problem, when all the tasks have the same computational cost and the communication time between tasks is neglected, is equivalent to the problem of finding a layering of a DAG with at most m vertices per layer and the minimum number of layers. Thus, the earliest static scheduling algorithms which deal with simplified models have also found an application as DAG layering algorithms.

Most of the static scheduling algorithms are variations of a generic list scheduling technique which consists of two main steps:

1. Build a scheduling list that contains all the tasks.
2. While the scheduling list is not empty remove the first task from it and schedule it for execution on a processor which allows earliest start time.

There are two list scheduling algorithms that have been widely employed as layering algorithms: the *longest-path algorithm* and the Coffman-Graham algorithm.

The Longest-Path Algorithm

The longest-path algorithm solves the static scheduling problem for $m = \infty$. Let π be the number of vertices in the longest directed path in a DAG. The longest path algorithm builds the scheduling list by assigning priority π to the vertices without outgoing edges. If all immediate successors of a vertex have been assigned a priority then that vertex is assigned the lowest of the priorities of its immediate successors minus one. This is repeated until all vertices are assigned a priority. The vertices with the same priority k form layer $L_{\pi-k+1}$. It has been shown that the longest-path algorithm has linear time complexity. [Meh84].

Algorithm 13.6 is a version of the longest-path algorithm where vertices are assigned to layers as soon as they are assigned priority. It employs two vertex sets U and Z which are empty in the beginning. The value of the variable *current_layer* is the label of the layer currently being built. As soon as a vertex gets assigned to a layer it is also added to the set U. Thus, U is the set of all vertices already assigned to a layer. Z is the set of all vertices assigned to a layer below the current layer. A new vertex v to be assigned to the current layer is picked among the vertices which have not been already assigned to a layer,

Figure 13.6 The Longest-Path Algorithm(G)

```
Requires: DAG $G = (V, E)$

$U \leftarrow \phi$
$Z \leftarrow \phi$
$currentLayer \leftarrow 1$
while $U \neq V$ do
   Select vertex $v \in V \setminus U$ with $N_G^+(v) \subseteq Z$
   if $v$ has been selected then
      Assign $v$ to the layer with a number $currentLayer$
      $U \leftarrow U \cup \{v\}$
   end if
   if no vertex has been selected then
      $currentLayer \leftarrow currentLayer + 1$
      $Z \leftarrow Z \cup U$
   end if
end while
```

i.e., $v \in V \setminus U$, and which have all their immediate successors already assigned to the layers below the current one, i.e., $N_G^+(v) \subseteq Z$.

The advantages of the longest path algorithm are its simplicity and its linear time complexity. The layerings it finds have the minimum height. However, it performs very poorly in terms of drawing area, number of dummy vertices and edge density [HN02b]. The longest-path layerings tend to be very wide at the bottom layers.

The Coffman-Graham Algorithm

The second list scheduling algorithm used for DAG layering is the Coffman-Graham algorithm [CG72] which is based on an earlier algorithm by Hu [Hu61]. It approximately solves the NP-hard static scheduling problem for $m < \infty$. The technique used for building the scheduling list is more complex than the one used by the longest path algorithm. The worst-case time complexity of the Coffman-Graham algorithm is $O(|V|^2)$. It guarantees a layering with at most m original vertices per layer and in the worst case the height of the layering may become close to twice the optimal height [CG72].

The Coffman-Graham algorithm requires that the input graph $G = (V, E)$ has no transitive edges. An edge $e = (u, v) \in E$ is *transitive* if there is a directed path with a start vertex u and an end vertex v in G with length greater than 1, i.e., e is not the only directed path from u to v. Let $G_r = (V, E_r)$ and $G_c = (V, E_c)$ be also DAGs. G_c is a *transitive closure* of G if for each pair of vertices $u, v \in V$ there is a directed path from u to v in G if and only if $(u, v) \in E_c$. G_r is a *transitive reduction* of G if G_r is a DAG with a minimum number of edges among all the DAGs which have the same transitive closure as G. If a DAG contains transitive edges then the Coffman-Graham algorithm can be applied to its transitive reduction. A DAG's transitive reduction can be computed in $O(M(|V|))$ time, where $M(n)$ is the time for computing the product of two $n \times n$ matrices [AGU72].

Let $G = (V, E)$ be a DAG without transitive edges. The Coffman-Graham algorithm is the two-step Algorithm 13.7. The first step consists of computing unique labels $\lambda : V \to \mathbb{N}$ of all the vertices of G which then are used at the second step as priority for placing the vertices in layers. The computation of the labels at the first step involves the comparison

Figure 13.7 The Coffman-Graham Layering Algorithm

Require: A DAG $G = (V, E)$ without transitive edges and an integer $W > 0$

```
for all $v \in V$ do
  $\lambda(u) \leftarrow \infty$
end for
for $i \leftarrow 1$ to $|V|$ do
  Choose $v \in V$ with $\lambda(v) = \infty$ such that $N_G^-(v)$ is minimized;
  $\lambda(u) \leftarrow i$;
end for
$k \leftarrow 1$, $L_1 \leftarrow \emptyset$, $U \leftarrow \emptyset$
while $U \neq V$ do
  Choose $v \in V \setminus U$ such that $N_G^+(v) \subseteq U$ and $\lambda(v)$ is maximized;
  if $|L_k| \leq W$ and $N_G^+(v) \subseteq L_1 \cup L_2 \cup \ldots \cup L_{k-1}$ then
    $L_k \leftarrow L_k \cup \{v\}$
  else
    $k \leftarrow k + 1$; $L_k \leftarrow \{v\}$
  end if
  $U \leftarrow U \cup \{v\}$
end while
```

between vertex sets defined as follows. If U_1 and U_2 are two sets of vertices then $U_1 < U_2$ if either

- $U_1 = \emptyset$ and $U_2 \neq \emptyset$; or
- $U_1 \neq \emptyset$, $U_2 \neq \emptyset$, and $\max\{\lambda(v) : v \in U_1\} < \max\{\lambda(v) : v \in U_2\}$; or
- $U_1 \neq \emptyset$, $U_2 \neq \emptyset$, $\max\{\lambda(v) : v \in U_1\} = \max\{\lambda(v) : v \in U_2\}$, and $U_1 \setminus \{v : \lambda(v) = \max\{\lambda(u) : u \in U_1\}\} < U_2 \setminus \{v : \lambda(v) = \max\{\lambda(u) : u \in U_2\}\}$

In the second step each vertex is assigned to a layer starting from the bottom layer and going upward keeping the maximum number of vertices in a layer less than or equal to an upper bound W.

It has been observed that Coffman-Graham layerings have a large amount of dummy vertices and when they are taken into account the area of the layerings can be even worse than the area of the longest path layerings [HN02b].

Nikolov and Tarassov have proposed a vertex-promotion improvement heuristic which can be applied after either the longest-path algorithm or the Coffman-Graham algorithm for reducing the number of dummy vertices [NT06]. It is a cubic algorithm which is compensated by its simplicity. We describe the vertex-promotion heuristic in the following section.

Layering with the Minimum Width

It is NP-hard to find a layering with the minimum width if the dummy vertices are assigned non-negative width. This problem can be solved exactly by the Healy and Nikolov's branch-and-cut layering algorithm described in Section 13.3.2. In this section we describe the fast heuristic approach proposed by Tarassov *et al.* [TNB04]. Their min-width layering algorithm is the first successful attempt to design a heuristic for layering with the minimum width and consideration of dummy vertices. An earlier attempt is the heuristic developed by Branke *et al.* [BELM01].

Figure 13.8 `Min-width`(G, W, c)

```
Requires: DAG $G = (V, E)$, integers $W$ and $c$

$U \leftarrow \phi$; $Z \leftarrow \phi$
$currentLayer \leftarrow 1$; $widthCurrent \leftarrow 0$; $widthUp \leftarrow 0$
while $U \neq V$ do
  Select vertex $v \in V \setminus U$ with $N_G^+(v) \subseteq Z$ and ConditionSelect
  if $v$ has been selected then
    Assign $v$ to the layer with a number $currentLayer$
    $U \leftarrow U \cup \{v\}$
    $widthCurrent \leftarrow widthCurrent - d^+(v) + 1$
    $widthUp \leftarrow widthup + d^-(v)$
  end if
  if no vertex has been selected OR ConditionGoUp then
    $currentLayer \leftarrow currentLayer + 1$
    $Z \leftarrow Z \cup U$
    $widthCurrent \leftarrow widthUp$
    $widthUp \leftarrow 0$
  end if
end while
```

The min-width algorithm, presented as Algorithm 13.8, is roughly based on the longest-path algorithm which is shown in detail in Algorithm 13.6. Besides the DAG G the min-width algorithm has two input parameters W and c which are explained below.

Similar to the longest-path algorithm, the min-width algorithm builds the layering layer by layer starting from layer 1. The two variables `widthCurrent` and `widthUp` are used to store the width of the current layer and the width of the layers above it, respectively. The width of the current layer, `widthCurrent`, is calculated as the number of original vertices already placed in that layer plus the number of potential dummy vertices along edges with a source in $V \setminus U$ and a target in Z (one dummy vertex per edge). The variable `widthUp` provides an estimation of the width of *any* layer above the current one. It is the number of potential dummy vertices along edges with a source in $V \setminus U$ and a target in the current layer (one dummy vertex per edge).

Vertex v is selected to be placed in a layer subject to an additional condition `ConditionSelect` which is true if v is the vertex with the maximum out-degree among the candidates to be placed in the current layer. Such a choice of v results in maximum reduction of `widthCurrent`. If either no vertex has been selected or `ConditionGoUp` is true then the current layer is completed and the algorithm moves to the next layer. `ConditionGoUp` is true if either:

- `widthCurrent` $\geq W$ and $d^+(v) < 1$, or
- `widthUp` $\geq c \times W$.

It is required that $d^+(v) < 1$ when `widthCurrent` $\geq W$ because the initial value of `widthCurrent` is determined by the dummy vertices in the current layer and it gets smaller (or at least it does not change) when a vertex with a positive out-degree gets placed in the current layer. In that case, the dummy vertices along edges with a source v are removed from the current layer and get replaced by v. If $d^+(v) \geq 1$, the condition `widthCurrent` $\geq W$ on its own is not a reason for moving to the upper layer because there is still a chance to add vertices to the current layer which will reduce `widthCurrent`. If $d^+(v) < 1$ then the assignment of v to the current layer increases `widthCurrent` because it does not replace any dummy vertices. This is an indication that `widthCurrent` can not be reduced further.

The min-width layering algorithm has the same time complexity as the longest-path algorithm which has been shown to run in linear time [Ulm75]. Through an extensive computational study Tarassov *et al.* have determined that the narrowest layerings are found for $1 \leq UBW \leq 4$ and $1 \leq c \leq 2$. Thus, they propose to run the algorithm for $UBW \in \{1, 2, 3, 4\}$ and $c \in \{1, 2\}$, choose the narrowest of the eight layerings and apply to it vertex-promotion heuristic described in the following section.

Improvement by Promotion of Vertices

The list-scheduling based layering algorithms described above have been shown to find layerings with a relatively large number of dummy vertices. Nikolov and Tarassov have proposed a simple vertex-promotion heuristic that can be applied to any layering for reducing its dummy vertex count [NT06].

The vertex-promotion heuristic modifies a given layering $\mathcal{L} = \{L_0, L_1, \ldots, L_h\}$ of a DAG G by promoting vertices from the layer where they are placed to the layer above. It is applied only to the original DAG vertices, not to the dummy vertices. To *promote* vertex v with $l(v, \mathcal{L}) = k$ is to move v from L_k to L_{k+1} which results in a new partition $\mathcal{L}^* = \{L_0, \ldots, L_k \setminus \{v\}, L_{k+1} \cup \{v\}, \ldots, L_h\}$. If $v \in L_h$ has to be promoted then a new empty layer L_{h+1} is added to the layering and v is promoted to it. If v has an immediate predecessor placed in layer L_{k+1} then $\mathcal{L}^*$ is *not* a layering of G. To ensure that the result of the promotion of vertex v to layer L_{k+1} is a layering all immediate predecessors of v in layer L_{k+1} (if there is any) have to be promoted to layer L_{k+2}; the same applies to their immediate predecessors and so on.

The recursive function which performs the described vertex promotion is shown in Algorithm 13.9. It takes vertex v as an input parameter and returns *dummydiff* which is the difference between the number of dummy vertices before and after the promotion v. In the **for** loop, each immediate predecessor u of v which lies in the layer above v gets promoted. The return value of its promotion is added to *dummydiff*. Then we promote v, subtract from *dummydiff* the number of immediate predecessors of v, and add to it the number of immediate successors of v. That is, we promote v one layer up, recursively promoting in advance all its immediate predecessors which need to be promoted. The time complexity of `PromoteVertex` is $O(|E|)$ because in the worst case all DAG edges might be traversed while promoting vertices recursively.

Then the vertex-promotion heuristic consists of two nested loops shown in Algorithm 13.10, an external **repeat-until** loop and an internal **for** loop. In the internal loop all vertices in a layered DAG are scanned in no particular order and each vertex with a positive in-degree gets promoted by `PromoteVertex` (see Algorithm 13.9) if its layering-preserving promotion reduces the total number of dummy vertices. The external loop goes on until the internal loop makes no promotion.

When performed after the min-width layering algorithm, described above, the vertex-promotion heuristic performs only promotions which do not increase the maximum number of vertices (original plus dummy) in a layer.

There is empirical evidence that 80 iterations of the **repeat-until** loop are enough for achieving a significant reduction of the number of dummy vertices for graphs with up to 100 vertices. If the number of iterations of the **repeat-until** loop is $O(|V|)$ then the vertex-promotion heuristic is cubic in the worst case.

Figure 13.9 `PromoteVertex`(v)

Require: A layered DAG $G = (V, E)$ with the layering information stored in a global vertex array of integers called *layering*; a vertex $v \in V$.

```
$dummydiff \leftarrow 0$
for all $u \in N_G^-(v)$ do
    if $layering[u] = layering[v] + 1$ then
        $dummydiff \leftarrow dummydiff +$ PromoteVertex$(u)$
    end if
end for
$layering[v] \leftarrow layering[v] + 1$
$dummydiff \leftarrow dummydiff - N_G^-(v) + N_G^+(v)$
return $dummydiff$
```

Figure 13.10 `Vertex-Promotion Heuristic`

Require: $G = (V, E)$ is a layered DAG; a valid layering of G is stored in a global vertex array called *layering*.

```
$layeringBackUp \leftarrow layering$
repeat
    $promotions \leftarrow 0$
    for all $v \in V$ do
        if $d^-(v) > 0$ then
            if PromoteVertex$(v) < 0$ then
                $promotions \leftarrow promotions + 1$
                $layeringBackUp \leftarrow layering$
            else
                $layering \leftarrow layeringBackUp$
            end if
        end if
    end for
until $promotions = 0$
```

Network-Simplex Layering Algorithm

The integer linear programming (ILP) approaches to the layering algorithm have been introduced for layering with the minimum number of dummy vertices. The first such approach is the layering technique designed by Gansner, Koutsofios, North and Vo for their system **dot**,[2] which is probably the most popular system for layered graph drawing [GKNV93].

[2]http://www.graphviz.org/

Figure 13.11 `Network Simplex Layer Assignment`

Require: $G = (V, E)$ is a DAG.

```
feasible_tree
while ($e$ =leave_edge()) $\neq$ nil do
  $f$ =enter_edge($e$)
  exchange($e, f$)
end while
normalize()
balance()
```

They model the layering problem by the following integer linear program:

$$\begin{aligned} \min \sum_{(u,v)\in E} & \; l(u,\mathcal{L}) - l(v,\mathcal{L}) \\ \text{subject to:} \quad & l(u,\mathcal{L}) - l(v,\mathcal{L}) \geq 1, \quad \forall (u,v) \in E \\ & l(u,\mathcal{L}) \geq 0, \quad \forall u \in V \\ & \text{all } l(u,\mathcal{L}) \text{ are integer} \end{aligned}$$

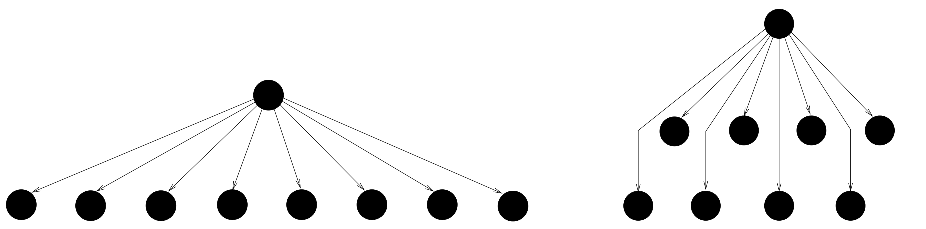
The linear programming relaxation of this integer program always has an integer solution because its constraint matrix is totally unimodular [NW88]. Thus, the integer program can be solved by the simplex method or any of the polynomial-time algorithms for solving linear programs. If we add an additional set of constraints of the type $l(u, \mathcal{L}) \leq H$, where H is an upper bound on the number of layers, the constraint matrix remains totally unimodular. Thus, the problem of finding a layering with the minimum number of dummy vertices subject to an upper bound on the number of layers is also in P.

Gansner *et al.* go further by introducing a network simplex algorithm for solving their ILP formulation [GKNV93]. It has not been proved to run in polynomial-time but reportedly requires a few iterations and runs fast. Its main part is presented in Algorithm 13.11.

The network simplex algorithm is based on the idea that each spanning tree of the underlying undirected graph of a DAG induces a family of equivalent (in terms of dummy vertex count) layerings. The algorithm starts with an initial spanning tree which is modified by replacing edges in order to get a spanning tree that induces a layering with the minimum number of dummy vertices. The procedure `normalize()` at the end of the algorithm is the one that makes sure the set of layers in the induced layering start from L_1.

The network simplex starts with an initial spanning tree built by the procedure `feasible_tree`. Gansner *et al.* suggest to compute a longest-path layering and then take a spanning tree of short subject to the layering edges. Then each iteration of the **while**-loop removes an edge from the spanning tree which breaks the tree into two connected components. Then a new edge is added to the tree that connects the two components into a new spanning tree. The two edges to leave and enter the tree respectively are chosen so that the new tree induces a layering with a lower dummy vertex count.

The edge to leave the tree at each iteration of the **while**-loop is chosen by the function `leave_edge()` which picks an edge with a negative *cut value*, or nil if all edges have non-negative cut value. The cut value is defined as follows. If an edge e is removed from the spanning tree, it breaks into two connected components, a *tail* and a *head*. The tail is the component that contains the source of e and head is the component that contains its target. The cut value of e is the number of all directed edges from the tail to the head, including e,


(a) Two layers and no dummy vertices.

(b) Three layers and four dummy vertices.

Figure 13.12 Two alternative layerings of the same DAG: a layering with the minimum number of dummy vertices may become too wide.

minus the number of all directed edges from the head to the tail. Typically a negative cut value of an edge means that the dummy vertex count can be reduced by lengthening that edge as much as possible, until one of the head-to-tail edges becomes tight. That tight edge is the one chosen by the function `enter_edge()`. It is the edge with the minimum span in the layering induced by the spanning tree before removing e.

After the end of the **while**-loop the spanning tree induces a layering with the minimum number of dummy vertices. The procedure `balance()`, applied at the end, moves vertices with equal in- and out-degree to a feasible layer with the fewest vertices. This is done in order to have more even distribution of vertices between layers. Gansner *et al.* also show how the network-simplex algorithm can work for graphs with weighted edges and with edges which are required to have span greater than 1 [GKNV93].

In general, layerings with the minimum number of dummy vertices lead to compact drawings. However some patterns in a DAG can result in too wide layerings with the minimum number of dummy vertices as shown in Figure 13.12.

Healy-Nikolov's Branch-and-Cut Algorithm

Another ILP approach is the branch-and-cut layering algorithm introduced by Healy and Nikolov [HN02a]. It finds layerings with the minimum number of dummy vertices subject to upper bounds on both the height and the width of the layering if there is any feasible solution. Variable vertex width and the contribution of the dummy vertices to the width of the layering can be taken into account. Since it solves exactly an NP-hard problem, this algorithm has exponential running time. It is especially designed for producing high quality layerings which satisfy exactly the pre-specified upper bounds on the width and the height.

Consider a DAG $G = (V, E)$ and let x be the incidence vector of a subset of $V \times \{1, \ldots, H\}$. Healy and Nikolov model the layering problem by the following ILP formulation.

$$\min \sum_{(u,v)\in E} \left(\sum_{k=\varphi(u)}^{\rho(u)} kx_{uk} - \sum_{k=\varphi(v)}^{\rho(v)} kx_{vk} \right) \tag{13.1}$$

Subject to
$$\sum_{k=\varphi(v)}^{\rho(v)} x_{vk} = 1 \qquad \forall v \in V \tag{13.2}$$
$$\sum_{i=\varphi(u)}^{k} x_{ui} + \sum_{i=k}^{\rho(v)} x_{vi} \leq 1 \qquad \forall k \in LS(u) \cap LS(v),\ \forall (u,v) \in E \tag{13.3}$$
$$\sum_{v \in V_k^*} w_v x_{vk} + \mathcal{D}_k \leq W \qquad \forall k = 1,\dots,H \tag{13.4}$$
$$\sum_{v \in V_k^*} x_{vk} \geq 1 \qquad \forall k = 1,\dots,\pi(G) \tag{13.5}$$
all x_{vk} are binaries

W and H are upper bounds on the width and height of the layering respectively; $\pi(G)$ is the number of vertices in the longest directed path in G; $\varphi(v)$ and $\rho(v)$ are respectively the lowest and the highest layer where vertex v can be placed in; $LS(v) = \{\varphi(v),\dots,\rho(v)\}$ and $V_k^* = \{v \in V : \varphi(v) \leq k \leq \rho(v)\}$. The objective minimizes the sum of edge spans, i.e., the number of dummy vertices. Equalities (13.2) force each vertex to be placed in exactly one layer; inequalities (13.3) force each edge to point downward; and inequalities (13.5) introduce the additional requirement of having at least one vertex in the first $\pi(G)$ layers. This reduces the number of identical layerings (but shifted vertically) if the height of the solution is less than the upper bound H.

Inequalities (13.4) restrict the width of each layer (including the dummy vertices) to be less than or equal to W: the first term on the left hand side represents the contribution of the real vertices to the width of layer V_k while $\mathcal{D}_k$ represents the contribution of the dummy vertices. We set
$$\mathcal{D}_k = \sum_{e=(u,v)\in E} w_e^d \left(\sum_{l>k}^{\rho(u)} x_{ul} - \sum_{l \geq k}^{\rho(v)} x_{vl} \right)$$
where w_e^d is the width of the dummy vertices along edge e. The difference of the two sums in the parentheses is 1 if edge $e = (u,v)$ spans layer V_k and 0 otherwise.

Healy and Nikolov propose solving their ILP formulation in a branch-and-bound framework with the employment of a cutting-plane algorithm at each vertex of the branch-and-bound tree. The cutting-plane algorithm generates valid inequalities for the constraint polytope of their formulation, some of which are facet-defining. Healy and Nikolov report that the running-time of this branch-and-cut algorithm is close to the time necessary for the ILP solver of CPLEX[3] to solve the formulation and they show some examples where the branch-and-cut algorithm is significantly faster than CPLEX.

13.3.3 The Layering Algorithms Compared

If any layering is acceptable then probably the easiest way to construct one is either to use the longest-path algorithm (Section 13.3.2) or to find any spanning tree of the underlying undirected graph and take the layering induced by it as described in the beginning of Section 13.3.2. If there is an upper bound on the the number of original vertices per layer then the Coffman-Graham algorithm, described in Section 13.3.2, is the best solution.

[3]http://www.ilog.com/products/cplex/

Although the longest-path algorithm finds layerings with the minimum number of layers and the Coffman-Graham algorithm finds layerings with a pre-specified maximum number of vertices in a layer, their layerings typically lead to drawings which occupy large drawing area and have too many long edges.

For a compact layering, the network simplex algorithm of Gansner *et al.*, described in Section 13.3.2, is probably the best fast solution. It finds layerings with the minimum number of dummy vertices which are also very compact in general. However, there are particular patterns in graph that may make the network simplex layering either too wide or too long. The branch-and-cut algorithm of Healy and Nikolov, outlined in Section 13.3.2, is much slower but in addition to the network simplex algorithm it guarantees that the layering's width and height will be within pre-specified bounds. It also considers variable vertex width and the contribution of the dummy vertices to the width of the layering.

The first fast heuristic for layer assignment with the minimum number of vertices per layer when both the original and the dummy vertices are considered is the min-width algorithm described in Section 13.3.2. It is not optimal but when followed by the vertex promotion heuristic, described in Section 13.3.2, it finds layerings which on average are narrower than the layerings found by any other known layering algorithm. The same vertex-promotion heuristic significantly improves the layerings found by the longest-path and the Coffman-Graham algorithms and makes them comparable to the layerings found by the network simplex algorithm. The longest-path algorithm followed by the vertex-promotion heuristic is probably the easiest to implement layering algorithm which results in good-quality layerings. Its only disadvantage is the relatively slow running time, which is cubic in the worst case.

13.3.4 Layer-Assignment with Long Vertices

The layer-assignment algorithms described above assume that all the vertices have similar height and can be aesthetically arranged on parallel horizontal levels without too much blank space between the levels. However, in some applications a few vertices in the input digraph may have large labels which can make them occupy significantly larger space in the vertical direction than the rest of the vertices. Such *long* vertices can be allowed to occupy more than one horizontal level in order to achieve aesthetically pleasant drawing.

Misue *et al.* propose to assign vertices to layers with one of the describe algorithms assuming all vertices have the same size. Then in a postprocessing step vertices can be enlarged to their original size and the eventual intersection between vertices are removed [MELS95]. However, this approach may lead to drawings which are not aesthetically acceptable.

Recently two studies have proposed layering algorithms that consider the actual size of the vertices while assigning them to layers. North and Woodhull have introduced an algorithm that assigns each vertex to two layers which correspond to the lower and the upper bounds of its height, respectively [NW01]. In a subsequent step vertices and edges which cross one or more layers are split into chains of vertices to obtain simpler layering. The second algorithm has been introduced by Friedrich and Schreiber [FS04]. It breaks the long vertices into chains of vertices while assigning them to layers.

While the special treatment of long vertices makes the final drawing compact, it also complicates the subsequent steps of the Sugiyama method. The vertex-ordering and the coordinate-assignment steps need to take the long vertices into account. It also becomes more difficult to rout edges so that they do not go through the long vertices. This may result in a large number of edge bends, which is compensated by their short length in the compact drawing.

13.4 Edge Concentration

Edge concentration is an optional step in the Sugiyama algorithm. It reduces the edge density between adjacent layers and the number of edge crossings. It can also reduce the dummy vertex count. The eventual drawbacks are that edge concentration modifies the graph and may increase the number of layers.

Consider a layered DAG $G = (V, E)$ with a layering $L = \{L_1, \ldots, L_h\}$. An *intersection* in G is a complete bipartite (biclique) subgraph I with a set of source vertices S_I and a set of target vertices T_I, such that $S_I \subseteq L_j$, and $T_I \subseteq L_i$ for some $i < j$. We use the notation $I = (S_I, T_I)$. If $|S_I| = |T_I| = 1$ then the intersection is *trivial*. The *vertex size* of I is $|S| + |T|$, and the *edge size* of I is $|S| * |T|$.

To perform *edge concentration* on a given nontrivial intersection I is to

- remove all edges between S_I and T_I from G
- add a new *edge-concentration vertex ec* to G, i.e., $V \leftarrow V \cup \{ec\}$
- add edges $\{e = (ec, u) : u \in T_I\}$ and $\{e = (u, ec) : u \in S_I\}$ to E.

That is, all edges of the intersection C are removed from the graph, a new edge-concentration vertex is added and all vertices in S_I and T_I are connected to it by an edge to form a star-like subgraph. An example is shown in Figure 13.13. If S_I and T_I occupy adjacent layers, i.e., $j - i = 1$, then a new layer is introduced between L_i and L_j and the edge-concentration vertex ec is placed in it. Otherwise ec is placed in some layer L_k with $i < k < j$.

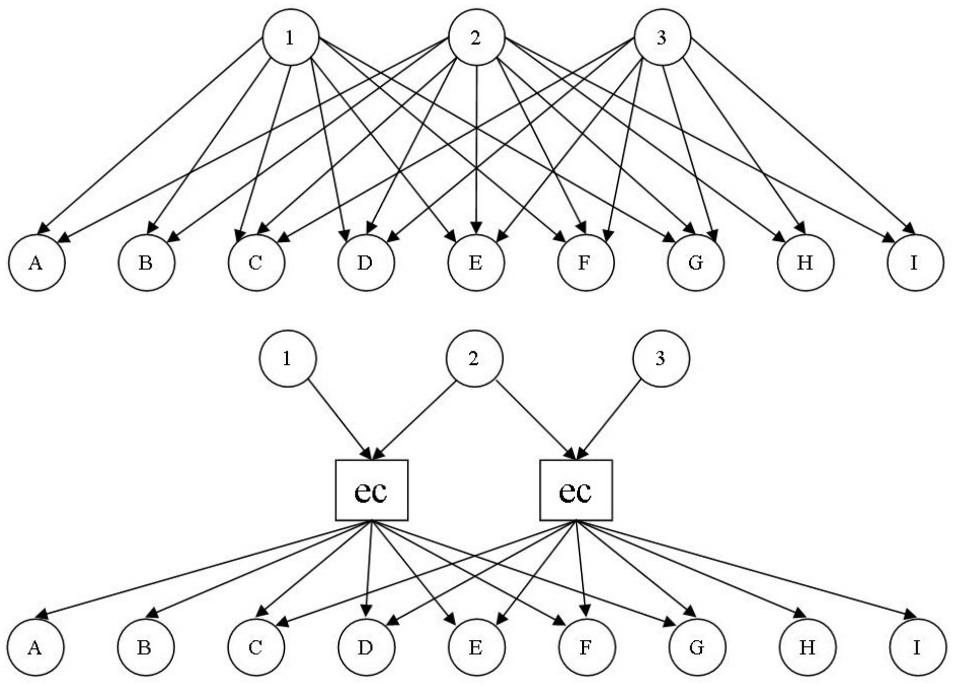


Figure 13.13 An example of a bipartite graph before and after the introduction of two edge-concentration vertices labeled "ec" [New89].

A set of intersections $\mathcal{I} = \{I_1, \ldots, I_k : 1 \leq k \leq |E|\}$ is an *intersection cover* of G if each edge of G is contained in at least one intersection in the set. The edge concentration step within the Sugiyama method consists of finding an intersection cover $\mathcal{I}$ of G and performing edge concentration on each non-trivial intersection in $\mathcal{I}$. If intersections between non-adjacent layers are considered then the dummy vertices are ignored and edge concentration is applied to intersections of original vertices and edges. After the edge concentration step dummy vertices are introduced again to subdivide long edges.

In the next section we discuss the different approaches to choosing an intersection cover for edge concentration.

13.4.1 Intersection Cover

The most important part of the edge concentration step is the choice of an intersection cover. There are two alternative approaches to this problem proposed in the graph drawing literature: choose either only intersections between adjacent layers, or only intersections between non-adjacent layers.

The only edge concentration approach with intersections between non-adjacent layers is the one employed by AT&T's **dot** [GKN02]. Only non-trivial intersections with a single target vertex are considered. This is a simple but fast solution for the edge concentration step.

Newbery as well as Eppstein *et al.* suggest a different approach to building the intersection cover [New89, EGM04]. The non-trivial intersections are only intersections between adjacent layers and the choice of intersections between two adjacent layers does not depend on the intersections between other layers. Thus, the problem of choosing an intersection cover of the whole graph is reduced to the problem of choosing a biclique (i.e., complete bipartite graph) cover of a bipartite graph.

The best biclique cover from the point of view of edge concentration is the one that will result in the fewest number of edges after applying edge concentration. This is what Newbery calls the Edge Concentration problem and it is defined as a decision problem as follows.

Edge Concentration
Instance: A bipartite graph $G = (V, E)$ and a positive number K.
Question: Is there an biclique cover $\mathcal{I} = \{I_1, \ldots, I_k : 1 \leq k \leq |E|\}$ of G with $\sum_{i=1}^{k} vs(I_i) \leq K$?

Edge Concentration is NP-complete [Lin00]. Newbery proposes a polynomial-time heuristic algorithm for solving its optimization version, i.e., to find a biclique cover $\mathcal{I} = \{I_1, \ldots, I_k : 1 \leq k \leq |E|\}$ with the minimum $\sum_{j=1}^{k} vs(I_j)$ [New89]. We describe the heuristic in detail in the following section.

Related to Edge Concentration is the problem of finding a biclique cover of a bipartite graph with no more than $K > 0$ bicliques. This problem is known as Complete Bipartite Subgraph Cover, problem GT18 of Garey and Johnson's NP-complete problems [GJ79]. In their paper on confluent layered drawings, Eppstein *et al.* propose two-layer edge concentration with a biclique cover which is an approximate solution to the optimization version of Complete Bipartite Subgraph Cover [EGM04]. Fishburn and Hammer have shown that Complete Bipartite Subgraph Cover is equivalent to a simply-restricted edge coloring problem which in turn can be transformed to a vertex coloring problem for bipartite graphs [FH96]. Thus, Eppstein *et al.* propose the biclique cover for edge concentration to be computed by one of the vertex coloring algorithms and specifically by either the

Recursive Largest First (RLF) algorithm of Leighton[Lei79] or the DSATUR algorithm of Brélaz [Bré79]. Both vertex coloring algorithms have $O(|E|^3)$ worst-case time complexity and their solutions can be transformed to a biclique cover in $O(|E|^2)$ time.

In the next section we describe in detail Newbery's heuristic for solving Edge Concentration approximately.

13.4.2 Newbery's Algorithm

Newbery's algorithm (Algorithm 13.14) finds an approximate solution to the optimization version of Edge Concentration [New89]. Consider a bipartite graph $G = (V, E)$. Two lists of bicliques, B_1 and B_2, are maintained throughout. B_1 is the list of all possible bicliques with two source vertices at all times. The bicliques in B_1 are sorted in increasing order by number of target vertices. Initially B_2 contains only the empty biclique, i.e., a biclique with an empty source and target sets. At the end B_2 is a list of the non-trivial bicliques for edge concentration.

The algorithm takes an input parameter M which is a lower bound on the edge size of a biclique. Newbery defines the edge size of a biclique x as $|S_x| * (|T_x| - 1)$ which is slightly different from the actual number of edges in the biclique. This is chosen in order to avoid bicliques with a single target vertex.

The main part of the heuristic is the **for** loop that goes through all bicliques in B_1 with two nested **for** loops that go through all bicliques in B_2. Consider a biclique $x \in B_1$ with a source set S_x and a target set T_x. If the edge size of x is less than the lower bound M then x is discarder from B_1. Otherwise x is compared to each biclique y in B_2. If x and y have the same target set then the source vertices of x are added to the source vertices of y. If there is no biclique y in B_2 with the same target set as x then x is compared once again to all bicliques in B_2 in the second nested **for** loop. Let S_y and T_y be the source and target sets of y respectively. Two cases are considered:

- **Case 1.** If $T_x \subseteq T_y$ then add $(S_y \cup S_x, T_x)$ to the front of B_2 and remove T_x from the target set of y.
- **Case 2.** If $T_y \subseteq T_x$ then add $(S_x, T_x \setminus T_y)$ to the front of B_2 and add S_x to the source set of y, i.e., $S_y \leftarrow S_y \cup S_x$.

Note that if no other condition becomes true then at the end of the second nested **for** loop x will be compared to the biclique with the empty target set in B_2 which will result in adding x to B_2. At the end all bicliques in B_2 with the same target set are merged and bicliques with size less than M are discarded. The biclique with the empty target set will also be discarded from B_2.

The heuristic has $O(n^4)$ worst-case time complexity assumed $m < n^3$. The worst case is much worse than the typical case encountered in practice.

13.5 Vertex Ordering

For readability edge crossings are one of the crucial parameters of a graph drawing [Pur97]. While edge reversal and layer assignment can have an impact on this, it is through the ordering of vertices that minimizing edge crossings between adjacent layers is mainly achieved (crossings are dependent on the relative order of vertices and not their positions). Thus, this step is often known as the *crossing minimization* or *crossing reduction* step.

Crossing minimization is of interest to VLSI-layout researchers and so has a history that predates much of the graph drawing literature. Garey and Johnson showed that the

Figure 13.14 Newbery's Biclique Cover Heuristic

Require: A bipartite graph $G = (V, E)$ and an integer $M > 0$

```
$B_1 \leftarrow \phi$
$B_2 \leftarrow \phi$
for all pair of source vertices $(u, v)$ do
  add the largest biclique with a source set $\{u, v\}$ to $B_1$.
end for
Sort the bicliques in $B_1$ in increasing order by number of target vertices
Add a biclique with empty source and target sets to $B_2$
for all $x \in B_1$ do
  if the $S_x * (T_x - 1) < M$ then
    Discard $x$
  else
    for all $y \in B_2$ do
      if $T_x = T_y$ then
        $S_y \leftarrow S_y \cup S_x$
        Continue the external for loop
      end if
    end for
    for all $y \in B_2$ do
      if $T_x \subseteq T_y$ then
        Add $(S_y \cup S_x, T_x)$ to the front of $B_2$
        $T_y \leftarrow T_y \setminus T_x$
        Continue the external for loop
      end if
      if $T_y \subseteq T_x$ then
        Add $(S_x, T_x \setminus T_y)$ to the front of $B_2$
        $S_y \leftarrow S_y \cup S_x$
        Continue the external for loop
      end if
    end for
  end if
end for
```

general crossing minimization problem, where permutations for both layers that minimize the number of crossings are required, is NP-hard [GJ83]; even if one layer is fixed, the problem still remains NP-hard [EW94]. To counter these apparently negative results, several heuristic and exact methods have been proposed.

While we will mainly be concerned with algorithms that *find* a drawing with few crossings, the decision version of the problem [GJ83, EW94] also is of interest. In particular, *fixed-parameter tractable* algorithms have been developed to answer the question of whether an ordering of the vertices on one "shore" of a bipartite graph admits a k-crossing (or less) solution when the vertices of the other shore remain fixed [DFH$^+$01a, DW02, DFK03]. In the latter an $O(1.4656^k + kn^2)$-time decision algorithm is presented.

We may assume that the graph $G = (V, E)$ is properly k-layered. That is, like the level graph in Definition 13.1 $V = V_1 \cup \cdots \cup V_k$, $V_i \cap V_j = \emptyset, 1 \leq i \neq j \leq k$, although we do not insist that sources appear on the first layer. The set of edges, $E = \{(u, v) | u \in V_i, v \in V_{i+1}, 1 \leq i \leq k-1\}$. We let $n_i = |V_i|, m = |E|$ and we call $N(u) = \{v \in V | (u, v) \in E\}$ the neighborhood of vertex u. An ordering or *permutation*, π_i, of each V_i provides a solution for the crossing minimization problem since it is the relative ordering along the line $y = l_i$ that causes edges incident on that layer to cross each other. What we seek, then, is the set of permutations, $\Pi = \{\pi_i | 1 \leq i \leq k\}$ that minimizes the edge crossings $C(G, \pi_1, \pi_2)$.

In the following sections we will describe the one-layer and two-layer crossing minimization problems and solutions. We then go on to discuss techniques for handling multi-layer graphs. Finally, we discuss an alternative to crossing minimization, where the goal is to initially extract (and draw) a large planar subgraph, and then draw the remaining edges.

13.5.1 One-Sided Crossing Minimization

Few systems attempt to globally minimize edge crossings. Instead, a heuristic approach based on one-layer crossing minimization is adopted. This problem, then, is key to many of the algorithms that have been proposed for the bi- and multi-partite crossing minimization problems. The goal of the one-layered crossing minimization (OLCM) problem is to find, for a given G and π_1, the permutation π_2 that minimizes $C(G, \pi_1, \pi_2)$.

Counting Crossings

Many of the algorithms we describe below require knowledge of the exact number of crossings between two layers. Algorithms that compute crossings fall in two categories: those that simply count the crossings and those that can *report* those edge crossings, also. Since it is possible to have $\Omega(|E|^2)$ crossings, the latter class of algorithms have running time $\Omega(|E|^2)$ in the worst case. The naive algorithm that considers every pair of edges and runs in $O(|E|^2)$ time is, in this sense, optimal.

A further issue is whether the algorithm can be used to compute a *crossing number matrix* which is used by many heuristics. The crossing number matrix, with entries c_{uv}, counts the number of edge crossings between edges incident to $u, v \in V_2$, when u is to the left of v ($\pi_2(u) < \pi_2(v)$). Since it assumes that the vertices in V_1 are fixed, the matrix is only relevant for OLCM. Note that since u is to the left of v or v is to the left of u in any solution then the sum $\sum_{u,v} \min(c_{uv}, c_{vu})$ yields a simple lower bound on the optimal number of crossings. Jünger and Mutzel report that this figure is surprisingly tight on a variety of graphs [JM97]. For dense graphs, Nagamochi [Nag05] presents an upper bound on OLCM of $1.2964 + 12/(\delta - 4)$, where $\delta > 4$ is the minimum degree of a vertex.

Computing the crossing number matrix can be done naively in $O(|E|^2)$-time although Sander [San94] proposes a sweep algorithm that can compute all entries in $O(|V_1| + |V_2| +$

$|E| + C)$, where C is the number of crossings. Barth *et al.* [BJM02] investigate a number of different existing algorithms for computing the bilayer crossing number, as well as proposing a new $O(|E| \log |V_s|)$-time algorithm, where V_s is the smaller of the two sets V_1 and V_2. The algorithm is based on Waddle and Malhotra's *accumulation tree* used in their earlier algorithm [WM99]. Nagamochi and Yamada [NY04] have proposed two algorithms based on dynamic programming and divide-and-conquer that run in time $O(|V_1||V_2|)$ and $O(\min\{|V_1||V_2|, |E| \log |V_s|\})$, respectively; both algorithms use $O(|E|)$-space. For dense graphs these algorithms will outperform the $O(|E| \log |V_s|)$-time algorithm by Barth *et al.* While Sander's algorithm turns out to be uncompetitive for large graphs, it does have the advantage of being able to compute the crossing matrix, which is not possible with either of the other faster-running algorithms.

Heuristic Approaches

Eades and Kelly [EK86] propose three heuristics: 1) greedy insertion, 2) greedy switching, and 3) split for the problem. All three methods require the precomputation of the crossing number matrix.

The *greedy insertion* heuristic orders vertices left to right on the "free" layer according to the one which minimizes the number of crossings that the edges incident to u make with the edges incident to vertices to u's right. That is, u is chosen to minimize $\sum_{v \in R}$, where R is the set of unchosen vertices. The algorithm runs in time $O(|V_2|^2)$. The *greedy switching* heuristic compares adjacent vertices and switches them if the change in crossing count, $c_{uv} - c_{vu} > 0$, where vertex u immediately precedes v. With a precomputed crossing number matrix the algorithm's running time is obviously $O(|V_2|^2)$. Like greedy switching, the *split* heuristic is reminiscent of sorting. Here, however, the analogy is with *quicksort* where a vertex, p, is chosen as a "pivot" and the vertices are rearranged into two consecutive sets, V_l and V_r so that $c_{up} < c_{pu}$ for all vertices in V_l and $c_{pu} \leq c_{up}$ for all vertices in V_r; the left and right subsets of vertices are then ordered recursively. As with quicksort its worst-case running time is $O(|V_2|^2)$, but its expected running time is $O(|V_2| \log |V_2|)$.

Eades and Wormald [EW94] propose the *median* heuristic that assigns to each vertex $v \in V_2$ the median position of the x-coordinates of its neighbours, $N(v) \subseteq V_1$. This heuristic has the important property that it will find an ordering with 0 crossings if such an ordering exists; in general, it guarantees an ordering, π_2, such that $C(G, \pi_1, \pi_2) \leq 3C_{\text{opt}}(G, \pi_1, \pi_2)$. Computing the median position for a vertex v has time complexity $O(|N(v)|)$ and, thus, all medians can be found in $O(|E|)$-time. Sorting the set V_2 according to the computed medians provides the ordering and requires $O(|V_2| \log |V_2|)$-time.

An alternative to placing a vertex at the median of its neighbours' x-coordinates is to place it at the *average* of them. This gives rise to the *barycenter* or averaging heuristic [STT81]. It has the same running time bounds as the median heuristic and it, too, will find a crossing-free ordering if one exists, but there does not exist a general performance guarantee as exists for the median heuristic.

Some other heuristics of note are Catarci's [Cat95] *assignment heuristic* which is based on an approximation of the linear assignment problem. Catarci claims that the heuristic is more accurate than the median heuristic, especially when applied to dense graphs. Dresbach [Dre94] has proposed a *stochastic heuristic* which, having calculated *assessment* numbers for each vertex and position (the assessment number is a term borrowed from statistics), begins placing vertices in the position with the smallest assessment position, updating the remaining numbers after each placement.

Genetic algorithms have also been employed as heuristic solutions to the one-layer crossing minimization problem. Mäkinen and Sieranta [MS94] encode a permutation of V_2 as a tuple. The mutation operator is defined as removing and re-inserting a random member x_i of the tuple to a new random position, with intervening members shifted accordingly. The crossover operation on tuples generates two points $i < j$, fixes the elements in the two tuples between i and j, and reorders the remainders of the two tuples so that they are both valid permutations. Branke *et al.* [UBSE98] also use a genetic algorithm for crossing minimization. Due to its more general setting, however, we will postpone its discussion until Section 13.7.

Exact Methods

Jünger and Mutzel [JM97] present a branch-and-cut algorithm for OLCM that draws on solving the *linear ordering* problem. Letting δ_{ij}^k be a 0-1 variable that represents the ordering of vertices i and j on level $k = 1, 2$, they develop an expression for the number of crossings of a pair of permutations, π_i, which will be the objective function to minimize. That is, let $\delta_{ij}^k = 1$ if $\pi_k(i) < \pi_k(j)$ and 0 otherwise; it is clear that $\delta_{ij} = 1 - \delta_{ji}$. Then

$$C(\pi_1, \pi_2) = \sum_{i=1}^{n_2-1} \sum_{j=i+1}^{n_2} \sum_{k \in N(i)} \sum_{l \in N(j)} \delta_{kl}^1 \delta_{ji}^2 + \delta_{lk}^1 \delta_{ij}^2 \tag{13.6}$$

For OLCM we assume a fixed π_1 and seek the ordering of V_2, π_2, that minimizes the crossings. That is, we wish to minimize

$$C(\pi_2) = \sum_{i=1}^{n_2-1} \sum_{j=i+1}^{n_2} \sum_{k \in N(i)} \sum_{l \in N(j)} \delta_{kl}^1 \delta_{ji}^2 + \delta_{lk}^1 \delta_{ij}^2 \tag{13.7}$$

As currently posed $C(\pi_2)$ is problematical since it involves quadratic terms. However, using the crossing number for a pair of vertices i and j in V_2 from before

$$c_{ij} = \sum_{k \in N(i)} \sum_{k \in N(i)} \delta_{lk}^1$$

we can rewrite equation (13.7) as

$$C(\pi_2) = \sum_{i=1}^{n_2-1} \sum_{j=i+1}^{n_2} c_{ji}(1 - \delta_{ji}^2) + c_{ij}\delta_{ij}^2 \tag{13.8}$$

$$\sum_{i=1}^{n_2-1} \sum_{j=i+1}^{n_2} (c_{ij} - c_{ji})\delta_{ij}^2 + \sum_{i=1}^{n_2-1} \sum_{j=i+1}^{n_2} c_{ji} \tag{13.9}$$

The problem then reduces to finding the vector $\delta^2 \in \{0,1\}^{\binom{n_2}{2}}$ that orders the vertices. In order for the vertices to be given a consistent ordering, we need to impose a "3-cycle" constraint that says that if vertex i precedes vertex j and vertex j precedes vertex k then vertex i must precede vertex k.

Since c_{ij} and c_{ji} can be determined beforehand and since the second double sum of equation (13.9) is a constant, the problem can be written as the following linear program, where δ_{ij}^2 is replaced by the more usual x_{ij}

$$\text{minimize} \quad \sum_{i=1}^{n_2} \sum_{j=i+1}^{n_2} a_{ij} x_{ij} \tag{13.10}$$

$$0 \le x_{ij} + x_{jk} - x_{ik} \le 1 \qquad 1 \le i < j < k \le n_2 \tag{13.11}$$

$$0 \le x_{ij} \le 1 \qquad 1 \le i < j \le n_2 \tag{13.12}$$

$$x_{ij} \text{ an integer} \tag{13.13}$$

This is a well-known formulation of the linear ordering problem and from it the number of crossings can easily be computed. As the integrality constraint (equation (13.13)) causes significant difficulty from a computational complexity consideration, a standard approach is to solve a relaxation of the formulation. This approach, called *branch-and-cut*, is achieved by dropping initially the integrality constraint and as many of the constraints as is necessary, and introducing them subsequently on an as-needed basis. In this case since there are $2\binom{n_2}{3}$ 3-cycle constraints (equation (13.11)) and $2\binom{n_2}{2}$ hypercube constraints (equation (13.12)) it has been found to be reasonable to begin simply with the hypercube constraints and to introduce constraints for violated 3-cycles until a solution is found. If the solution is non-integral a branch step takes place where a non-integral variable is selected and two subproblems are solved, one using 0 as the value of the selected variable and the second using 1 as its value.

Algorithm Performance

The most comprehensive experimental analysis of the performance of the various approaches to OLCM that we have discussed is Jünger and Mutzel's [JM97]. Their focus is in bipartite graphs and they consider the OLCM and TLCM versions of the problem using most of the heuristics we have discussed above, and others. In this section we will focus on their findings for OLCM.

Two classes of experiments are performed. First, the algorithms are applied to random instances of graphs, of varying edge density, each with 20 vertices on each side. Secondly, motivated by the sparseness that is often observable in graphs of interest, a set of sparse graphs of varying sizes were considered. It is worth noting that as graphs get denser and the exact crossing count increases all algorithms perform relatively better. Thus, the second class of experiments may tell more about the algorithms' behaviors.

For the first set of experiments it was observed that the greedy switch, greedy insertion and split heuristics had significantly higher running times as the edge densities increased. This is most certainly due to the requirement to compute the crossing number matrix. The running time of the exact method was surprisingly competitive here, although its exponential behavior became more obvious on the second class of graphs. The split algorithm performed best in terms of solution quality and also fared very well in the second class of sparse graphs.

Over both classes of graphs it would appear that barycenter is the best performer. Its running time is always among the best and its solution quality on the sparse graphs is always with 3% of optimality. In spite of their similarity, the median algorithm performs poorer than barycenter on both classes of graphs. (The barycenter algorithm is also an easier algorithm to implement.)

Finally, Mäkinen and Sieranta [MS94] claim that their genetic algorithm outperforms the barycenter method in terms of accuracy, although running time is significantly higher.

13.5.2 Multi-Layer Crossing Minimization

In the multi-layer crossing minimization problem (MLCM) we are given a k-layered graph, $G = (V_1, \ldots, V_k, E)$, with the goal of finding permutations $\pi_1, \ldots, \pi_k$ such that the edge crossings are minimized. Again, both exact and heuristic methods have been proposed.

Before we deal with the full k-layer problem, it is worth considering a number of approaches to the two-layer crossing minimization problem (TLCM). Recall that it was on this problem that first complexity results were found [GJ83].

Two-Layer Crossing Minimization

For TLCM Valls *et al.* [VML96] propose a branch-and-bound algorithm that, they claim, considers a small fraction of the total search space. They report that the average time taken to compute C_{opt} for a bipartite graph $G(U, V, E)$, $|U| = |V| = 13$ and edge-density 0.3 was 40 minutes. As the following method appears to give better results we focus on this method.

In the course of considering OLCM Jünger and Mutzel [JM97] observed that a simple lower bound on the number of crossings proved to be tight. Based on this they developed a branch-and-bound algorithm for TLCM. Given an hour of computation time on a Sun Sparcstation 10 it was possible to solve to optimality problems varying in size from vertex size $V_1 = V_2 = 11$ with 80% edge density, to vertex size $V_1 = V_2 = 16$ with 10% edge density. Of heuristic methods for TLCM iterated barycenter was the clear champion, even surpassing, surprisingly, an iterated scheme based around an exact OLCM algorithm.

Newton *et al.* [NSV02] consider two new heuristics that, they claim, outperforms the iterated barycenter method. The first method is based on a connection between the bipartite crossing number and the linear arrangement problem which seeks to order the vertices of a graph so that the absolute distance between edges is minimized [SSSV01]. Any heuristic solution to the linear arrangement problem can then be immediately used as a solution to the bipartite crossing number problem. Two methods based on computing the Fiedler eigenvector of the Laplacian of G are proposed. Another method proposed by the authors [NSV02] that also yields very good results is to repeatedly randomly choose a pair of vertices on the same side, swap their positions and keep this new solution (only) if an improvement resulted.

Heuristic Approaches

The most usual heuristic approach is to repeatedly apply a *layer-by-layer sweep* of a one-layer crossing minimization algorithm until no further improvement is possible [STT81]. That is, for a layer i, the vertices of layers $i-1$ and $i+1$ are held fixed if the layers exist and V_i is permuted. This procedure is then applied for successively increasing and decreasing i until there is no further improvement. Obviously other stopping criteria may be used instead and more sophisticated control mechanisms can replace the simple up-down strategy described here [GKNV93, San95].

An alternative to the layer-by-layer sweep approach was proposed by Matuszewski *et al.* [MSM99]. The idea, which can also be applied to OLCM, orders all vertices in $V = V_1 \cup \cdots \cup V_k$ according to their degree and finds the best position for it on its layer. It does this by *sifting* the vertex through all possible positions (all others remaining fixed), using the crossing number matrix to count the crossings that would result at each position.

A different heuristic alternative solves an exact crossing minimization problem on a subset or *window* of the multi-layer graph that matches certain criteria [EGDB02]. Some improvements over the champion iterated barycenter are presented, although running time remains a problem with the procedure.

Exact Methods

Following from their work on the linear ordering and OLCM problems Jünger *et al.* derived an integer linear programming (ILP) formulation for MLCM [JLMO97]. As in the one-layer case, quadratic terms analogous to those appearing in equation (13.7) prevent formulation as a *linear* program. In this case, however, neither layer is fixed so a different strategy is called for. The proposed solution is to introduce crossing variables, c_{ijkl}, denoting whether the edges (i, j) and (k, l) cross when $i < j$, $k < l$, $i < k$ and $j \neq l$. Using this notation along with boolean variables x_{ij} to denote $\pi_1(i) < \pi_1(j)$ and y_{ij} to denote $\pi_2(i) < \pi_2(j)$ TLCM can be formulated as

$$\text{minimize} \quad \sum_{(i,j),(k,l)\in E} c_{ijkl} \tag{13.14}$$

$$-c_{ijkl} \leq y_{jl} - x_{ik} \leq c_{ijkl} \qquad (i,j),(k,l) \in E, j < l \tag{13.15}$$

$$1 - c_{ijkl} \leq y_{lj} + x_{ik} \leq 1 + c_{ijkl} \qquad (i,j),(k,l) \in E, l > j \tag{13.16}$$

$$0 \leq x_{ij} + x_{jk} - x_{ik} \leq 1 \qquad 1 \leq i < j < k \leq n_1 \tag{13.17}$$

$$0 \leq y_{ij} + y_{jk} - y_{ik} \leq 1 \qquad 1 \leq i < j < k \leq n_2 \tag{13.18}$$

$$x_{ij}, y_{ij}, c_{ijkl} \in \{0, 1\} \tag{13.19}$$

If $c_{ijkl} = 0$ then equation (13.15) forces the ordering of vertices i and k to be the same as that of j and l; similarly, if edges (i, j) and (k, l) cross then $y_{jl} - x_{ik} = \{-1, 1\}$, forcing $c_{ijkl} = 0$. Since the variable $y_{jl}, l < j$ does not exist equation (13.16) uses the identity $y_{jl} = 1 - y_{jl}$ analogously.

The k-layer problem can then be formulated as

$$\text{minimize} \quad \sum_{r=1}^{p-1} \sum_{(i,j),(k,l)\in E_r} c^r_{ijkl} \tag{13.20}$$

$$-c^r_{ijkl} \leq x^{r+1}_{jl} - x^r_{ik} \leq c^r_{ijkl} \qquad (i,j),(k,l) \in E_r, j < l \tag{13.21}$$

$$1 - c^r_{ijkl} \leq x^{r+1}_{lj} + x^r_{ik} \leq 1 + c^r_{ijkl} \qquad (i,j),(k,l) \in E_r, l > j \tag{13.22}$$

$$0 \leq x^r_{ij} + x^r_{jk} - x^r_{ik} \leq 1 \qquad 1 \leq i < j < k \leq n_r \tag{13.23}$$

$$x^r_{ij}, c^r_{ijkl} \in \{0, 1\} \tag{13.24}$$

where

$$E_r = \{(u, v) \in E | u \in V_r, v \in V_{r+1}\}. \tag{13.25}$$

Jünger *et al.* [JLMO97] analyze the polytope associated with this ILP formulation and present some results that describe facets of the polytope. Healy and Kuusik analyze the *cycle space* of an MLCM instance [HK04] and derive additional constraints for the ILP formulation (13.20)–(13.24). In his PhD thesis Kuusik [Kuu00] uses these new inequalities. Also described is how additional constraints such as different paths that may not cross, or a group of vertices must appear consecutively can be included.

Algorithm Performance

Matuszewski *et al.* [MSM99] claim that their global sifting method leads to a 20% improvement over iterated barycenter, although the time taken is considerable. An improved version of the algorithm [GSBM01] improves the running time of the original by a factor of 10, the authors claim.

Jünger *et al.* [JLMO97] present some preliminary results of their ILP formulation of MLCM on 3-layer problems. The largest problem has $V_1 = 26, V_2 = 18, V_3 = 17, E_1 = 29, E_2 = 29$ and takes 330 seconds to find an optimal solution. Kuusik [Kuu00] presents solutions to larger problems, one measuring over 90 vertices and 112 edges spread over 9 levels (Forrester's "World Dynamics" [GKNV93]). What emerges is that the model is sensitive to the number of linear ordering variables, which is driven by the average node count per layer.

More recently, semidefinite programming (SDP) methods have begun to offer the prospect of solving larger problems to optimality. Following on from the 2-level case formulated by Buchheim *et al.* [BWZ10] general k-level solutions have been obtained. Chimani *et al.* [CHJM11] report on experiments where the method outperformed an ILP implementation. Test cases with up to 20 levels and up to 25 vertices per layer were randomly generated and while an ILP implementation was capable of solving sparse instances (inter-layer edge density, d, of up to 0.1) it was unable to solve a single case where $d \geq 0.2$, instances which their SDP solver was routinely able to solve.

13.5.3 Planarization – An Alternative

Mutzel [Mut01] observes that the understandability of a graph may be subtler than simply counting the number of crossings in the graph. It may be that if a *few* edges account for the crossings then the eye may more easily filter these crossings even if more than the minimum. Thus, an alternative to minimizing edge crossings in a k-level graph presents itself: determine the maximum level planar subgraph; draw this k-level subgraph without crossings; and, finally, reinsert the "nonplanar" edges, in the expectation that not many crossings will result. This problem has become of interest in the graph drawing community and a number of complexity results are known [DFH$^+$01a, DFH$^+$01b].

Mutzel and Weiskircher [MW98] adopt this approach by finding the maximum planar subgraph, which they call the two-layer planarization problem. They formulate the planarization problem as an ILP and they identify a set of facets of the associated polytope in the cases where one or both layers are fixed. In order to solve k-layer problems they solve a succession of two-layer problems, although this is clearly a sub-optimal strategy.

Kuusik uses ILP methods also for the general k-level planarization problem [Kuu00]. We state the problem as, given a k-level graph, $G = (V, E)$, find a k-level planar subgraph $G_p = (V, E_p)$ so that $|E_p|$ is maximum. An important aspect of finding the maximum k-level planar subgraph is identifying k-level subgraphs that are minimally non-planar and based on this characterization [HKL04] and the cycle space of the graph [HK04] facets of the associated polytope are identified. Some of these facets are put to use in a branch-and-cut algorithm.

An ILP formulation follows easily from that described for the minimum-crossing ILP (13.20)–(13.24). By introducing a binary variable p_{ij} for each edge where $p_{ij} = 1$ if and only if $(i,j) \in E_p$ and observing that if $(i,j) \in E_p$ and $(k,l) \in E_p$ then $c_{ijkl} = 0$, the following constraint expresses the relationship between the two

$$p_{ij} + p_{kl} + c_{ijkl} \leq 2 \tag{13.26}$$

A complete ILP may then be expressed as

$$\text{maximize} \sum_{(i,j) \in E} p_{ij} \tag{13.27}$$

subject to

$$p_{ij} + p_{kl} + c_{ijkl} \leq 2 \tag{13.28}$$

$$\text{constraints}(13.21) - (13.24) \tag{13.29}$$

$$p_{ij} \in \{0, 1\} \tag{13.30}$$

Among several equal in size maximum level planar subgraph solutions it may be desirable to favor ones with that will result in fewer crossings. Objective function (13.27) can be modified to take account of crossing variables and favor solutions with fewer crossings

$$\text{maximize} \sum_{(i,j)\in E} p_{ij} + \frac{1}{k_c + 1} \sum_{(i,j),(k,l)\in E} c_{ijkl} \tag{13.31}$$

where k_c is the number of crossing variables.

Kuusik reports [Kuu00] being able to solve a set of randomly generated problems of size $k = 8$, $V_1 = \cdots = V_8 = 12$, $E = 110$ in times ranging from 62(s) to, in an extreme case, 343(s).

13.6 x-Coordinate Assignment

Having determined a relative ordering of the vertices on each level the final step requires positioning the vertices so that, insofar as is possible, the edges are straight and vertices are centered among their neighbors. Aesthetically straight edges are desirable and this can be achieved by adjusting the positioning – the x-coordinate – of each vertex; there is also some evidence that, perceptually, straight edges are preferable [HEH09]. However, it is likely that the width of the graph will increase during this step.

It will be remembered that long edges of the input graph will have been subdivided by the introduction of dummy nodes and a vertical "flow through" for these vertices is particularly desirable. Given an edge from level i to level j in the input graph it becomes the path $(v_i, v_{i+1}, \ldots, v_j)$ after the insertion of dummy nodes. It is reasonable to expect that the sub-path $(v_{i+1}, \ldots, v_{j-1})$ be drawn strictly vertical with the (at most two) bends occurring at vertices v_{i+1} and v_{j-1}, if necessary. Most algorithms in the literature make this a priority.

In their original paper Sugiyama *et al.* [STT81] propose a mathematical programming solution with a quadratic objective function that is a convex combination of two separate goals. Closeness to connected vertices and a balance, or centering, between a vertex's predecessor neighbors and its successor neighbors are the two criteria identified. Since these goals will counteract each other in general they are each weighted by parameters c and $1-c$, $0 < c \leq 1$ respectively. All dummy nodes associated with a long edge are restricted to be vertically aligned; this presupposes that if two long edges cross, the crossing occurs on either the first or last segment of both edges. Because quadratic programming was computationally expensive at the time the authors also proposed a "Priority Layout Method" that operates in a similar spirit to their crossing-minimization level-by-level sweep. Dummy nodes are assigned a high priority so that they will be aligned, though care is required to ensure that the level's vertex ordering is maintained.

Gansner *et al.*, too, propose exact and heuristic algorithms [GKNV93]. Their exact algorithm is linear programming-based again. Whereas the quadratic objective function of Sugiyama [STT81] minimizes terms (among others) such as $\sum_{(u,v\in E}(x(u) - x(v)^2$, Gansner *et al.* replace the quadratic term by $|x(u) - x(v)|$. This can now be linearized but at the expense of introducing an additional variable and two additional constraints for each edge.

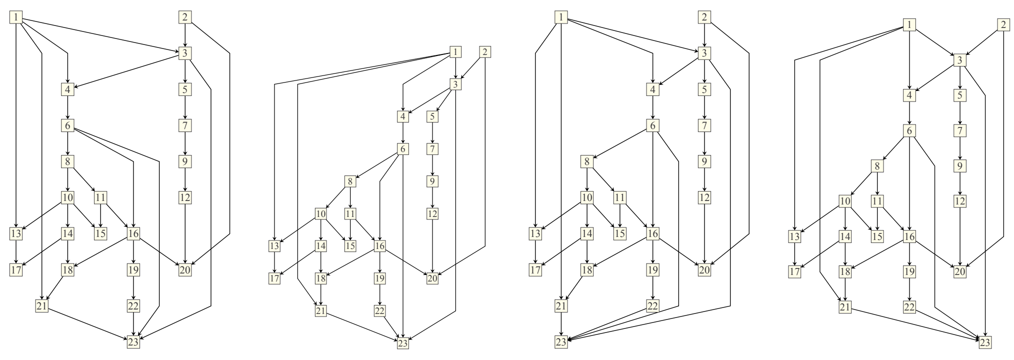

(a) leftmost upper (b) rightmost upper (c) leftmost lower (d) rightmost lower

Figure 13.15 Four "extreme" alignments computed by the Brandes-Köpf algorithm.

Thus, on the grounds that the constraint matrix grows from size VE to size $(V+E)E$, they dismiss this and propose a heuristic that operates, again, by sweeping up and down over the levels. Other heuristics have been proposed by Eades *et al.* who place nodes belonging to any layer other than the top or bottom according to a degree-weighted barycenter and sweep up and down the levels "until x converges" [ELT96], and Sander with his Pendulum heuristic, which lays out edges Manhattan style and thus may require four bends per edge [San96a].

We close this section with a more detailed look at Brandes and Köpf's algorithm [BK02], which is considered to be the algorithm of choice for assignment of x-coordinate; the algorithm's running time is linear in the number of vertices and edge segments. As we have remarked, the algorithm guarantees that, for long edges, at most two bends will occur, and if they are necessary they will occur on the external segments of an edge. Thus, provided the crossing reduction step has ensured that no two long edges cross internally, all dummy (internal) nodes of a long edge will be aligned.

Similar to Gansner *et al.* above, the algorithm uses $|x(u) - x(v)|$ as a surrogate for edge-length. Since, for a set $X = \{x_i\}$ of real numbers, $\sum_{i=1}^{k} |x - x_i|$ is minimized when x is the median of the set, the algorithm focuses on placing a vertex at the median of its neighbors.

The algorithm firstly computes four "extreme" layouts where each vertex is aligned either with its upper or lower median neighbor and, for each case, vertices on a level are considered in a left-to-right (left alignment) or right-to-left (right alignment) order. Figure 13.15 (reproduced with the authors' kind permission from their paper [BK02]) illustrates the four layouts for a given graph. Conflicts – called *type 0 conflicts* – can arise due to a pair of external segments that either cross or are incident upon the same vertex and these are resolved by the order of consideration. Because the algorithm favors vertical inner segments a crossing of an inner segment with an outer segment – a *type 1 conflict* – should give precedence to aligning the vertices of the inner segment. A preprocessing step marks such external segments as being excluded from consideration for alignment.

After each of the four extreme layouts are computed, the layout's blocks are computed. A *block* is a maximal set of vertically aligned vertices. These blocks may then be *compacted*, subject to minimum separation requirements, giving an x-coordinate for each vertex.

Having performed this procedure four times, each with a directional bias, what is surprising, perhaps, is that the four candidate positions can be combined so straightforwardly to get the final position with such satisfactory results. Of the four possibilities the algorithm chooses what is called the *average median*, which, for $x_1 \leq x_2 \leq \ldots \leq x_k$, is

$(x_{\lfloor (k+1)/2 \rfloor} + x_{\lceil (k+1)/2 \rceil})/2$. One of the arguments for this choice as opposed to the mean, say, is if a vertex is aligned twice with its upper median neighbor and unevenly positioned in the other two runs the average median will maintain verticality of the segment.

The main body of the algorithm is the four linear sweeps over the segments. The authors demonstrate how the type 1 conflicts can be detected in linear time, also. Finally, for each vertex, the average median of four can be determined in linear time.

13.7 Extensions and Alternatives to Sugiyama's Framework

There are a few attempts to extend the Sugiyama method beyond the originally proposed framework. Some extensions such as the cycle-removal step and the edge-concentration step have become standard features of the Sugiyama framework nowadays.

Do Nascimento and Eades have proposed and extensively studied methods for applying user constraints to hierarchical drawings interactively [dNE01a, dNE01b]. In particular, they allow the user to interact with an already-computed hierarchical drawing and introduce additional constraints on the relative position of some vertices. Taking the user constraints into account they show how to reapply the Sugiyama method locally to a subgraph of the digraph and efficiently recompute the coordinates of the affected nodes only. They also propose an interactive genetic algorithm with similar features [dNE02].

Although the Sugiyama method has been the most popular method for hierarchical drawing of digraphs, several researchers have proposed successful alternative methods. Sugiyama and Misue proposed a magnetic-field method for hierarchical drawing of acyclic digraphs [SM95a]. A global magnetic field is applied to the edges and they are aligned to it. As a result all edges point unidirectionally without specifically placing the vertices on parallel horizontal lines. Note that in any drawing with edges pointing into the same direction and continuous y-coordinates of the vertices levels can be easily induced from by a simple quantization procedure.

Utech *et al.* [UBSE98] proposed a genetic algorithm as an alternative to the Sugiyama method. The chromosome contains values which represent the y-coordinate (i.e., layer) and x-coordinate (i.e., position within the layer) of each original and dummy vertex. Thus, the x- and y-coordinates of the vertices evolved simultaneously. The fitness of each individual is provided by the number of edge crossings.

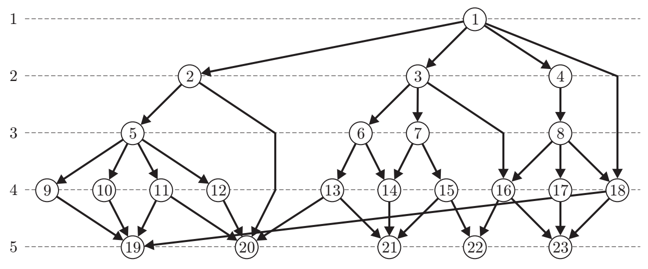
Carmel *et al.* proposed an energy-based alternative to the Sugiyama method [CHK04]. Based on rapidly solving a unique one-dimensional optimization problem for each of the axes their algorithm produces a clear representation of the hierarchical structure of the digraph. Vertices are not restricted to lie on horizontal levels which allows their layouts to convey the symmetries of the digraph. An interesting detail is that the algorithm can be applied without change to both cyclic or acyclic digraphs, and even to graphs containing both directed and undirected edges.

A few researchers have attempted 3D hierarchical drawing of digraphs. Ostry has proposed wrapping a 2D hierarchy around either a cylinder or a cone [Ost96]. A 3D approach intrinsically different from the Sugiyama method has been proposed within the graph drawing system GIOTTO3D [GT97]. In its first phase it applies a planarization method to draw the digraph in 2D; in the second phase vertices and edges are assigned z-coordinates so that all edges point into the same vertical direction and the total edge span is minimized; and at the third phase the shape of the vertices and the edges is determined.

Hong and Nikolov have proposed an extension to the Sugiyama framework for hierarchical drawing of digraphs in 3D [HN05b, HN05a]. They introduced the convention of drawing the digraph in a set of parallel planes, called *walls*, each containing a 2D hierarchical drawing of a subgraph of the input digraph. The partition of the vertex set into walls is done at a separate wall-assigning step applied after the layer-assignment step and before the vertex-ordering step. Hong and Nikolov proposed and evaluated various wall-assignment algorithms which partition the vertex set into walls according to different criteria.

One of the limitations of the Sugiyama framework is that decisions made at previous steps influence later steps and yet cannot be undone. We have seen that the edge of a cycle one chooses to break can have a bearing on how the graph is interpreted. Likewise, how one levels the graph impacts on the number of dummy nodes which has a bearing on the running time of the algorithm. Chimani *et al.* attempt to decouple some of these dependencies by sidestepping the three-step decomposition and instead search for a global solution with the principal goal of minimizing crossings [CGMW11, CGMW10]. The algorithm necessarily sacrifices drawing height for a lower crossing count and thus edge-length is increased. In spite of this the authors assert that their UPL system improves aspect-ratio; running time of their system suffers and is greater than their Sugiyama implementation [CGMW11].

An alternative fashion of drawing 2D hierarchies is by arranging the layers as concentric circles [RFEM88, Ead92]. Such drawings are known as *radial level drawings*. They can be very useful if the input digraph represents the structure of a website or a file system, for example. They have been used for visualizing social networks as well [BKW03]. The Sugiyama method can be applied with all its steps for making radial hierarchical drawings. However, each step may require a specific algorithm for producing aesthetically pleasing result. There are a few recent results in this area of research. Bachmaier *et al.* have proposed an algorithm for computing a radial level planar embedding if one exists along with a linear-time algorithm for the coordinate-assignment step [BFF05]. The figure below (reproduced with permission of the authors [BBF05]) illustrates one of the advantages of the drawing style: edge (18,19) of Figure 13.16a effectively can be routed "around the back of the drawing" thus avoiding the crossings, as demonstrated in Figure 13.16b.


(a) Horizontal leveling.

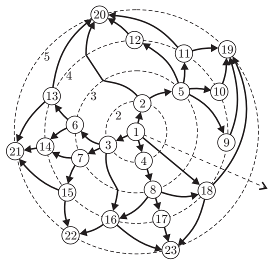
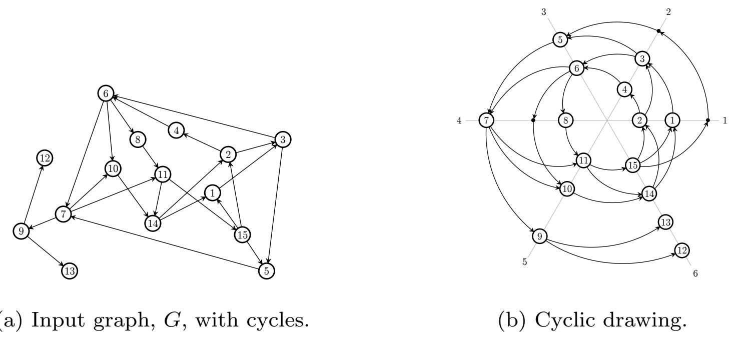

(b) Radial leveling.

Figure 13.16 A Horizontally Leveled Drawing and its Radial Counterpart.

Much of what has gone before relies on the input (directed) graph being acyclic, or being preprocessed so that it becomes one. Yet, Sugiyama and his colleagues in their seminal paper [STT81] also addressed the drawing of a particular class of cyclic digraph they called *recurrent hierarchies*. In this situation an additional set of edges $E_n \subseteq V_n \times V_1$ connect vertices on the bottom layer to vertices on the top layer. Brunner *et al.* consider the *cyclic* style of drawing [Bru10, BBBL08] where the inevitable "implied ranking" associated with a top-to-bottom leveling (see Figure 13.1) is avoided. Vertices assigned to a level are distributed along a spoke of an imaginary wheel with spokes evenly distributed. Figure 13.17 below illustrates the technique on the example graph of Figure 13.2.



(a) Input graph, G, with cycles.

(b) Cyclic drawing.

Figure 13.17 A Cyclic Drawing of the graph G of Figure 13.2.

References

[AGU72] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, June 1972.

[Bac07] C. Bachmaier. A radial adaptation of the Sugiyama framework for visualizing hierarchical information. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):583 –594, 2007.

[BBBF12] Christian Bachmaier, Franz J. Brandenburg, Wolfgang Brunner, and Raymund Fülöp. Drawing recurrent hierarchies. *Journal of Graph Algorithms and Applications*, 16(2):151–198, 2012.

[BBBL08] Christian Bachmaier, Franz-Josef Brandenburg, Wolfgang Brunner, and Gergö Lovász. Cyclic leveling of directed graphs. In Ioannis G. Tollis and Maurizio Patrignani, editors, *Graph Drawing*, volume 5417 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2008.

[BBF05] Christian Bachmaier, Franz J. Brandenburg, and Michael Forster. Radial level planarity testing and embedding in linear time. *Journal of Graph Algorithms and Applications*, 9(1):53–97, 2005.

[BELM01] J. Branke, P. Eades, S. Leppert, and M. Middendorf. Width restricted layering of acyclic digraphs with consideration of dummy nodes. Technical Report No. 403, Intitute AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany, 2001.

[BFF05] Christian Bachmaier, Florian Fischer, and Michael Forster. Radial coordinate assignment for level graphs. In L. Wang, editor, *Proc. Computing and Combinatorics Conference, COCOON 2005*, volume 3595 of *Lecture Notes in Computer Science*, pages 401–410. Springer, 2005.

[BJM02] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and efficient bilayer cross counting. In Kobourov and Goodrich [KG02], pages 130–141.

[BK02] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In P. Mutzel, M. Jünger, and Leipert S., editors, *Graph Drawing: Proceedings of 9th International Symposium, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 31–44. Springer-Verlag, 2002.

[BKW03] U. Brandes, P. Kenis, and D. Wagner. Communicating centrality in policy network drawings. *IEEE Transact. Vis. Comput. Graph.*, 9(2):241–253, 2003.

[BLME02] J. Branke, S. Leppert, M. Middendorf, and P. Eades. Width-restriced layering of acyclic digraphs with consideration of dummy nodes. *Information Processing Letters*, 81(2):59–63, January 2002.

[Bré79] D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[Bru10] Wolfgang Brunner. *Cyclic Level Drawings of Directed Graphs*. PhD thesis, Universitt Passau, Innstrasse 29, 94032 Passau, 2010.

[BS90] B. Berger and P. W. Shor. Approximation algorithms for the maximum acyclic subgraph problem. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 236–243, 1990.

[BWZ10] Christoph Buchheim, Angelika Wiegele, and Lanbo Zheng. Exact algorithms for the quadratic linear ordering problem. *INFORMS Journal on Computing*, 22(1):168–177, 2010.

[Car80] M. J. Carpano. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Transactions on Systems, Man and Cybernetics*, 10(11):705–715, 1980.

[Cat95] T. Catarci. The assignment heuristics for crossing reduction. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(3):515–521, 1995.

[CG72] E. G. Coffman and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[CGMW10] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Layer-free upward crossing minimization. *J. Exp. Algorithmics*, 15:2.2:2.1–2.2:2.27, March 2010.

[CGMW11] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Upward planarization layout. *J. Graph Algorithms Appl.*, 15(1):127–155, 2011.

[CHJM11] Markus Chimani, Philipp Hungerländer, Michael Jünger, and Petra Mutzel. An SDP approach to multi-level crossing minimization. In Matthias Müller-Hannemann and Renato F. Werneck, editors, *ALENEX*, pages 116–126. SIAM, 2011.

[CHK02] Liran Carmel, David Harel, and Yehuda Koren. Drawing directed graphs using one-dimensional optimization. In Michael Goodrich and Stephen Koubourov, editors, *Proceedings of 10th International Symposium, GD 2002*, number 2528 in Lecture Notes in Computer Science, pages 193–206. Springer, 2002.

[CHK04] L. Carmel, D. Harel, and Y. Koren. Combining hierarchy and energy for drawing directed graphs. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):46–57, 2004.

[DDLM04] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, and Henk Meijer. Computing radial drawings on the minimum number of circles. In János Pach, editor, *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2004.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing.* Prentice Hall, 1999.

[DF03] C. Demetrescu and I. Finocchi. Combinatorial algorithms for feedback problems with directed graphs. *Information Processing Letters*, 86:129–136, 2003.

[DFH$^+$01a] Vida Dujmovic, Michael R. Fellows, Michael T. Hallett, Matthew Kitching, Giuseppe Liotta, Catherine McCartin, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Matthew Suderman, Sue Whitesides, and David R. Wood. On the parameterized complexity of layered graph drawing. In Friedhelm Meyer auf der Heide, editor, *ESA*, volume 2161 of *Lecture Notes in Computer Science*, pages 488–499. Springer, 2001.

[DFH$^+$01b] Vida Dujmovic, Michael R. Fellows, Michael T. Hallett, Matthew Kitching, Giuseppe Liotta, Catherine McCartin, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Matthew Suderman, Sue Whitesides, and David R. Wood. A fixed-parameter approach to two-layer planarization. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2001.

[DFK03] Vida Dujmovic, Henning Fernau, and Michael Kaufmann. Fixed parameter algorithms for one-sided crossing minimization revisited. In Giuseppe Liotta, editor, *Graph Drawing*, volume 2912 of *Lecture Notes in Computer Science*, pages 332–344. Springer, 2003.

[DGL$^+$00] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Armando Parise, Roberto Tamassia, Emanuele Tassinari, Francesco Vargiu, and Luca Vismara. Drawing directed acyclic graphs: An experimental study. *Int. J. Comput. Geometry Appl.*, 10(6):623–648, 2000.

[dNE01a] H. A. D. do Nascimento and P. Eades. A framework for human-computer interaction in directed graph drawing. In *InVis.au*, pages 63–69. CRPIT, 2001.

[dNE01b] H. A. D. do Nascimento and P. Eades. User hints for directed graph drawing. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing: Proceedings of 9th International Symposium, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 205–219. Springer-Verlag, 2001.

[dNE02] H. A. D. do Nascimento and P. Eades. A focus and constraint-based genetic algorithm for interactive directed drawing. *HIS*, pages 634–643, 2002.

[Dre94] S. Dresbach. A new heuristic layout algorithm for directed acyclic graphs. In U. Derigs, A. Bachem, and A. Drexl, editors, *Operations Research Proceedings 1994*, pages 121–126, Berlin–Heidelberg, 1994.

[DW02] Vida Dujmovic and Sue Whitesides. An efficient fixed parameter tractable algorithm for 1-sided crossing minimization. In Kobourov and Goodrich [KG02], pages 118–129.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[Ead92] P. Eades. Drawing free trees. *Bulletin of the Institute of Combinatorics and its Applications*, 5:10–36, 1992.

[EGDB02] Thomas Eschbach, Wolfgang Günther, Rolf Drechsler, and Bernd Becker. Crossing reduction by windows optimization. In Kobourov and Goodrich [KG02], pages 285–294.

[EGM04] D. Eppstein, M. T. Goodrich, and J. Y. Meng. Confluent layered drawings. In J. Pach, editor, *Graph Drawing: Proceedings of 12th International Symposium, GD 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 184–194. Springer-Verlag, 2004.

[EK86] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria*, 21-A:89–98, 1986.

[EL95] P. Eades and X. Lin. A new heuristic for the feedback arc set problem. *Australian Journal of Combinatorics*, 12:15–26, 1995.

[ELS89] P. Eades, X. Lin, and W. F. Smyth. Heuristics for the feedback arc set problem. Technical Report 1, Curtin University of Technology, School of Computing Science, Perth, Australia, 1989.

[ELS93] P. Eades, X. Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[ELT96] Peter Eades, Xue-Min Lin, and Roberto Tamassia. An algorithm for drawing hierarchical graphs. *Int. J. Comput. Geom. Appl.*, 6:145–156, 1996.

[ENRS95] G. Even, J. Naor, S. Rao, and B. Schieber. Divide-and-conquer approximation algorithms via spreading metrics. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 62–71, 1995.

[ES90] P. Eades and K. Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4):424–437, 1990.

[EW94] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11:379–403, 1994.

[FH96] P. C. Fishburn and P. L. Hammer. Bipartite dimensions and bipartite degree of graphs. *Discrete Mathematics*, 160:127–148, 1996.

[Flo90] M. M. Flood. Exact and heuristic algorithms for the weighted feedback arc set problem: A special case of the skew-symetric quadratic assignment problem. *Networks*, 20:1–23, 1990.

[FS04] C. Friedrich and F. Schreiber. Flexible layering in hierarchical drawings with nodes of arbitrary size. In *Proceedings of the 27th Conference on Australasian Computer Science*, pages 369–376, Dunedin, New Zealand, 2004.

[FW95] M. Fröhlich and M. Werner. Demonstration of the interactive graph visualization system daVinci. In R. Tamassia and I. Tollis, editors, *Proceedings of DIMACS Workshop on Graph Drawing '94, Princeton (USA) 1994*, volume 894 of *LNCS*. Springer-Verlag, 1995.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[GJ83] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM J. Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[GJR85] M. Grötschel, M. Jünger, and G. Reinelt. On the acyclic subgraph polytope. *Mathematical Programming*, 33(1):28–42, 1985.

[GKN02] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with *dot*. Technical report, AT&T Labs—Research, 2002. `http://www.research.att.com/sw/tools/graphviz/dotguide.pdf`.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw., Pract. Exper.*, 30(11):1203–1233, 2000.

[GSBM01] W. Günther, R. Schönfeld, B. Becker, and P. Molitor. k-layer straightline crossing minimization by speeding up sifting. In J. Marks, editor, *Graph Drawing: Proceedings of 8th International Symposium, GD 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 253–258. Springer-Verlag, 2001.

[GT97] A. Garg and R. Tamassia. GIOTTO: A system for visualizing hierarchical structures in 3D. In S. North, editor, *Graph Drawing: Symposium on Graph Drawing, GD '96*, volume 1190 of *Lecture Notes in Computer Science*, pages 193–200. Springer-Verlag, 1997.

[HEH09] Weidong Huang, Peter Eades, and Seok-Hee Hong. A graph reading behavior: Geodesic-path tendency. In Peter Eades, Thomas Ertl, and Han-Wei Shen, editors, *PacificVis*, pages 137–144. IEEE Computer Society, 2009.

[Him00] Michael Himsolt. Graphlet: design and implementation of a graph editor. *Softw., Pract. Exper.*, 30(11):1303–1324, 2000.

[HK04] Patrick Healy and Ago Kuusik. Algorithms for multi-level graph planarity testing and layout. *Theoretical Computer Science*, 320(2–3):331–344, 2004.

[HKL04] Patrick Healy, Ago Kuusik, and Sebastian Leipert. A characterisation of level planar graphs. *Discrete Mathematics*, 280(1–3):51–63, 2004.

[HN02a] P. Healy and N. S. Nikolov. A branch-and-cut approach to the directed acyclic graph layering problem. In M. Goodrich and S. Koburov, editors, *Graph Drawing: Proceedings of 10th International Symposium, GD 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 98–109. Springer-Verlag, 2002.

[HN02b] P. Healy and N. S. Nikolov. How to layer a directed acyclic graph. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing: Proceedings of 9th International Symposium, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 2002.

[HN05a] S.-H. Hong and N. S. Nikolov. Hierarchical layouts of directed graphs in three dimensions. In P. Healy and N. S. Nikolov, editors, *Graph Drawing: Proceedings of 13th International Symposium, GD 2005*, volume 3843 of *LNCS*. Springer-Verlag, 2005.

[HN05b] S.-H. Hong and N. S. Nikolov. Layered drawings of directed graphs in three dimensions. In S.-H. Hong, editor, *Information Visualisation 2005: Asia-Pacific Symposium on Information Visualisation (APVIS2005)*, volume 45, pages 69–74. CRPIT, 2005.

[Hu61] T. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, November 1961.

[JLMO97] M. Jünger, E. K. Lee, P. Mutzel, and T. Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In G. Di Battista, editor, *Graph Drawing: 5th International Symposium, GD '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 13–24, Rome, Italy, September 1997. Springer-Verlag.

[JM97] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *J. Graph Algorithms Appl.*, 1, 1997.

[KA99] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999. ACM Computing Surveys 31(4): 406-471 (1999).

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KG02] Stephen G. Kobourov and Michael T. Goodrich, editors. *Graph Drawing, 10th International Symposium, GD 2002*, volume 2528 of *Lecture Notes in Computer Science*. Springer, 2002.

[Kra99] Jan Kratochvíl, editor. *Graph Drawing, 7th International Symposium, GD'99, Stirín Castle, Czech Republic, September 1999, Proceedings*, volume 1731 of *Lecture Notes in Computer Science*. Springer, 1999.

[Kuu00] Ago Kuusik. *Integer Linear Programming Approaches to Hierarchical Graph Drawing*. PhD thesis, University of Limerick, 2000.

[Lei79] F. T. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of National Bureau of Standard*, 84:489–506, 1979.

[Lin00] X. Lin. On the computational complexity of edge concentration. *Discrete Applied Mathematics*, 101:197–205, 2000.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms, Volume 2: Graph Algorithms and NP-Completeness*. Springer-Verlag, Heidelberg, Germany, 1984.

[MELS95] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6:183–210, 1995.

[MS94] E. Mäkinen and M. Sieranta. Genetic algorithms for drawing bipartite graphs. Technical Report A-1994-1, Department of Computer Science, University of Tampere, 1994.

[MSM99] Christian Matuszewski, Robby Schönfeld, and Paul Molitor. Using sifting for k-layer straightline crossing minimization. In Kratochvíl [Kra99], pages 217–224.

[Mut01] P. Mutzel. An alternative method to crossing minimization on hierarchical graphs. *SIAM Journal on Optimization*, 11(4):1065–1080, 2001.

[MW98] Petra Mutzel and René Weiskircher. Two-layer planarization in graph drawing. In Kyung-Yong Chwa and Oscar H. Ibarra, editors, *ISAAC*, volume 1533 of *Lecture Notes in Computer Science*, pages 69–78. Springer, 1998.

[Nag05] Hiroshi Nagamochi. On the one-sided crossing minimization in a bipartite graph with large degrees. *Theor. Comput. Sci.*, 332(1-3):417–446, 2005.

[New89] F. J. Newbery. Edge concentration: A method for clustering directed graphs. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 76–85. ACM Press, 1989.

[NPT90] Frances Newbery-Paulisch and Walter F. Tichy. Edge: An extendible graph editor. *Softw., Pract. Exper.*, 20(S1):S1/63–S1/88, 1990.

[NSV02] Matthew Newton, Ondrej Sýkora, and Imrich Vrto. Two new heuristics for two-sided bipartite graph drawing. In Kobourov and Goodrich [KG02], pages 312–319.

[NT06] N. S. Nikolov and A. Tarassov. Graph layering by promotion of nodes. *Discrete Applied Mathematics*, 154(5):848–860, 2006.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience series in discrete mathematics and optimization. John Wiley & Sons, Inc., 1988.

[NW01] S. North and G. Woodhull. Online hierarchical graph drawing. In P. Mutzel, M. Jünger, and Leipert S., editors, *Graph Drawing: Proceedings of 9th International Symposium, GD 2001*, volume 2265 of *LNCS*, pages 232–246. Springer-Verlag, 2001.

[NY04] Hiroshi Nagamochi and Nobuyasu Yamada. Counting edge crossings in a 2-layered drawing. *Inf. Process. Lett.*, 91(5):221–225, 2004.

[Ost96] D. Ostry. Some three-dimensional graph drawing algorithms. Master's thesis, University of Newcastle, 1996.

[PCJ96] H. C. Purchase, R. F. Cohen, and M. James. Validating graph drawing aesthetics. In F. J. Brandenburg, editor, *Graph Drawing: Symposium on Graph Drawing, GD '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 435–446. Springer-Verlag, 1996.

[Pur97] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In Giuseppe Di Battista, editor, *Graph Drawing. 5th International Symposium, GD '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 1997.

[RDM$^+$87] L. A. Rowe, M. Davis, E. Messinger, C. Mayer, C.and Spirakis, and A. Tuan. A browser for directed graphs. *Software Practice and Experience*, 17(1):61–76, 1987.

[Rei85] G. Reinelt. The linear ordering problem: Algorithms and applications. In *Research and Exposition in Mathematics*, volume 8. Heldermann, 1985.

[RFEM88] M. G. Reggiani and F. E. F. E. Marchetti. A proposed method for representing hierarchies. *IEEE Transact. Systems, Man, and Cyb.*, 18(1):2–8, 1988.

[San94] G. Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, 1994.

[San95] G. Sander. Graph layout through the VCG tool. In *Graph Drawing. DIMACS International Workshop GD'94*, volume 894 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[San96a] G. Sander. A fast heuristic for hierarchical Manhattan layout. In Franz Brandenburg, editor, *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 447–458. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0021828.

[San96b] G. Sander. Graph layout for applications in compiler construction. Technical Report A01-96, Universität des Saarlandes, FB 14 Informatik, 1996.

[Sey95] P. D. Seymour. Packing directed circuits fractionally. *Combinatorica*, 15:281–288, 1995.

[SM95a] K. Sugiyama and K. Misue. Graph drawing by the magneting spring model. *Journal of Visual Languages and Computing*, 6(3):217–231, 1995.

[SM95b] K. Sugiyama and K. Misue. A simple and unified method for drawing graphs: Magnetic-spring algorithm. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing: DIMACS International Workshop, GD '94*, volume 894 of *Lecture Notes in Computer Science*, pages 364–375. Springer-Verlag, 1995.

[SS97] B. Schwikowski and E. Speckenmeyer. On computing all minimal solutions for feedback problems. Technical report, Universität zu Köln, 1997.

[SSSV01] Farhad Shahrokhi, Ondrej Sýkora, László A. Székely, and Imrich Vrto. On bipartite drawings and the linear arrangement problem. *SIAM J. Computing*, 30:1773–1789, 2001.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transaction on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[TNB04] A. Tarassov, N. S. Nikolov, and J. Branke. A heuristic for minimum-width of graph layering with consideration of dummy nodes. In C. C. Ribeiro and S. L. Martins, editors, *Experimental and Efficient Algorithms, Third International Workshop, WEA 2004*, volume 3059 of *Lecture Notes in Computer Science*, pages 570–583. Springer-Verlag, 2004.

[UBSE98] J. Utech, J. Branke, H. Schmeck, and P. Eades. An evolutionary algorithm for drawing directed graphs. In *Proceedings of the 1998 International Conference on Imaging Science, Systems, and Technology (CISST'98)*, pages 154–160, 1998.

[Ulm75] J. Ulman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[VML96] V. Valls, R. Marti, and P Lino. A branch and bound algorithm for minimizing the number of crossing arcs in bipartite graphs. *Eur. J. Op. Res.*, 90:303–319, 1996.

[War77] J. N. Warfield. Crossing theory and hierarchical mapping. *IEEE Transactions on Systems, Man and Cybernetics*, 7(7):502–523, 1977.

[WM99] Vance E. Waddle and Ashok Malhotra. An $E \log E$ line crossing algorithm for levelled graphs. In Kratochvíl [Kra99], pages 59–71.