Chapter 9

Layered Drawing of graphs

This chapter presents a widely used approach for drawing digraphs, the hierarchical approach, that aims at creating polyline layered drawings.

Layered graph drawing (see Figure 9.1) is a popular paradigm for drawing graphs and has applications in visualization, in DNA mapping and in VLSI layout.

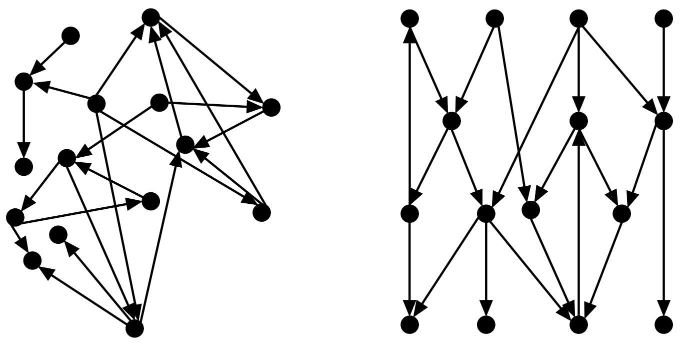

Figure 9.1: An example graph - A possible layered drawing.

The development of algorithms for computing layered drawings has started in the seventies. Graph layering is an important step in many algorithms for drawing directed acyclic graphs (DAGs): a graph layering algorithm partitions the graph into layers of independent sets of nodes and with this layering, the inter-layer connections and the ordering of nodes within a layer are determined in an algorithm-specific way.

There are several *Layering-Based Algorithms* that construct layered drawings; they accept, as input, directed graphs without any particular restriction (the input directed graph can be planar or not, acyclic or cyclic). In layered drawings, even though vertices and edge-bends are placed at integer coordinates, the edge-crossings can be arbitrarily close to each other or to the vertices and edgebends. Layering-based algorithms generally follow the methodology of Sugiyama, which consists of the following steps (see Figure 9.2):

1. Remove existing cycles in the input graph by reversing the direction of some edges.

2. Assign vertices to layers heuristically, optimizing some criteria, such as the total edge length.

3. Introduce fictitious vertices along edges whose end-vertices are not on consecutive layers. The result is a proper k-level graph.

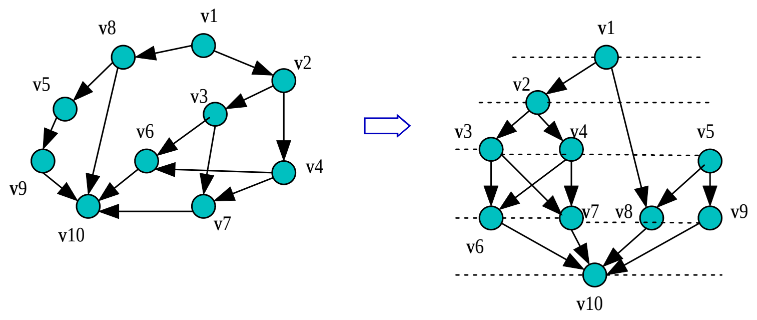

Figure 9.2: An example of Sugiyama hierarchical drawing style for directed graphs.

4. Reduce the crossings among edges by permuting the order of vertices on each layer.
5. Remove the fictitious vertices introduced in Step 3, replacing them with edge-bends.
6. Reduce the number of bends by readjusting the position of vertices on each layer.

The hierarchical approach consists of three main steps:

1. Layer Assignment: Assign each vertex to a horizontal layer, determining the y-coordinate of each vertex.
2. Crossing Reduction: Vertices within a layer are ordered to reduce edge crossings.
3. Horizontal Coordinate Assignment: Assign a x-coordinate to each vertex, with the goal of distributing the vertices uniformly and minimizing the number of bends.

   If the graph contains cycles, then an additional step is required:

4. Cycle Removal: As few edges as possible are *reversed* to make the graph acyclic. This allows drawing all edges in one direction which is important for the next step. At the end of the algorithm, the reversed edges are reversed again to obtain their initial orientation.

Most of the steps are NP-complete problems and thus a variety of heuristics are used. Each step is also fairly independent of the others and various techniques can be examined at each stage without needing to know those of the previous stage.

9.1 Layer Assignment

Formally, a layering of an acyclic digraph, $G = (V, E)$ is a partition of V into subsets $L_1, L_2, \ldots, L_h$, such that if $(u, v) \in E$, where $u \in L_i$ and $v \in L_j$, then $i > j$. The height is h and the width is the number of vertices in the largest layer. The span of an edge is the difference between the levels of the vertices to which it is incident. A digraph is proper if no edge has a span greater than one 1.

In the layering step, we want to find a layering of an acyclic digraph, such that the layered digraph has small width and height. Also we want the layering to be proper for the crossing reduction step that follows. The latter requirement is satisfied by inserting dummy vertices at each level along the path of every edge with span greater than one. Later on they are removed, leaving a polyline edge. Edges that originally had span one, are drawn as straight lines.

A layering algorithm should find a layering of a DAG subject to certain aesthetic criteria important to the final drawing:

1. The drawing should be compact; large edge spans should be avoided; and, the edges should be as straight as possible. Compactness can be achieved by specifying bounds W and H, on the width and the height of the layering, respectively. Short edge spans are desirable aesthetically because they increase the readability of the drawing but also because the forced introduction of dummy nodes complicates further stages of drawing algorithms. Further, these dummy nodes may also permit additional bends on edges since edge bends mainly occur at dummy nodes.

2. A *proper* layering is needed. A layering is called *proper* if edges occur between adjacent layers only. To achieve the latter, *dummy* vertices are introduced along the edges. Each edge (u, v) of span $k > 1$, is replaced with path $(u = v_1, v_2, \ldots, v_k = v)$, adding *dummy* vertices $v_2, v_3, \ldots, v_{k-1}$. Dummy vertices are needed in crossing reduction stage.

3. The number of dummy vertices should be small. Techniques exist to minimize the number of dummy vertices used. This is desirable since later steps depend on the number of vertices, dummy ones included, and because bends occur where dummy vertices lie and we want to minimize the number of bends. One technique defines a cost function which is equal to the total vertical span of all the edges. Then the layer assignment problem is reduced to choosing the y-coordinates as before, but now subject to miminizing this cost function. It can be solved as an integer linear programming problem in a standard way. Moreover, the total number of dummy and real vertices increases the time of steps required in layering approach.

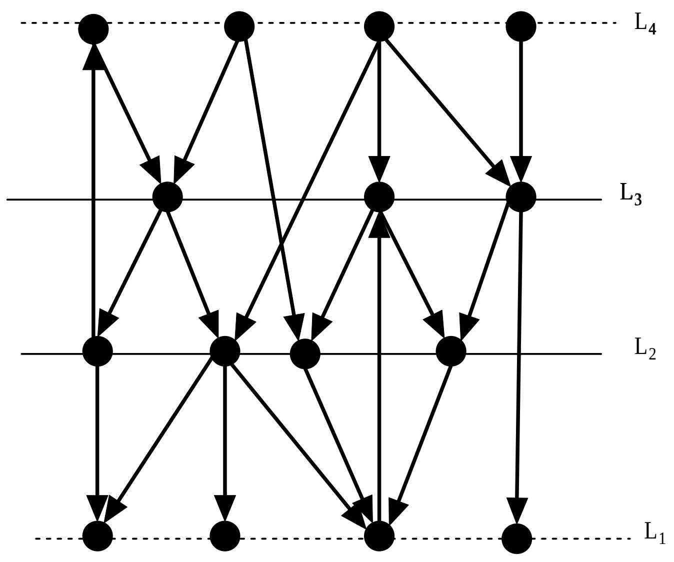

Figure 9.3: A layering of the graph.

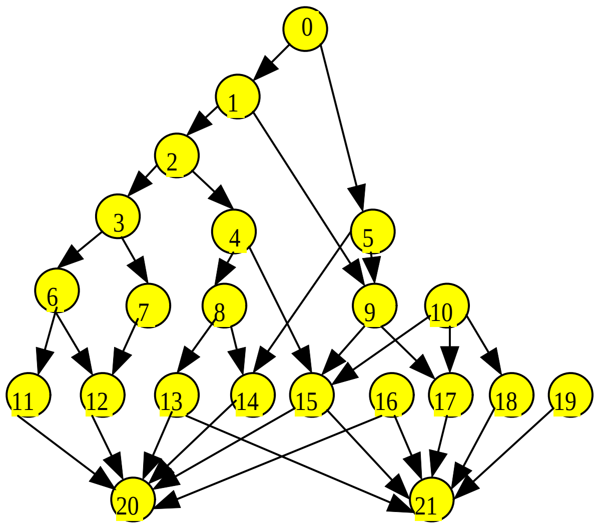

Figure 9.4: A layered drawing of a directed graph.

9.1.1 The Longest Path Layering

The so-called *Longest Path Layering* guarantees a layering with a minimum number of layers. All sinks are placed in L_1, then each remaining vertex, u is placed in layer L_{p+1}, where p is the longest path from u to a sink. Since the digraph is acyclic, this is done in linear time. The nymber of layers is addressed to be minimum, so that the height of the layering should be minimum. The shortcoming of this layering technique is that lower levels may be quite wide.

The following algorithm which is similar to the above algorithm computes layerings of minimum height. We assume that the graphs are acyclic. Each sink is placed in layer L_1. For the remaining vertices, the layer will be recursively defined by:

$$y(u) := \max\{i \,|\, v \in N^+(u) \qquad y(v) = i + 1\} \tag{9.1}$$

and

$$N^+(u) := \{v \in V \,|\, \exists\, (u, v) \in E\}. \tag{9.2}$$

This produces a layering where many vertices will stay close to the bottom. The algorithm can be implemented in linear time, using a topological ordering of the vertices.

9.1.2 Layering to minimize width

Given a fixed width greater or equal to three, the problem of finding a layering with minimum height is NP-complete. Suppose each vertex is a unit-time task to be executed in one of the processors of a multiprocessor, and each edge (u, v) of digraph G is the precedence constraint that u must precede v. In the precedence constrained multiprocessor scheduling problem all tasks that are assigned to the W processors must be executed in time H. This problem is also NP - complete.

In the following we will present the *Coffman-Graham-Algorithm*. The Coffman-Graham-Layering algorithm from the theory of multiprocessor scheduling is a layering method which tries to minimize the width (ignoring dummy vertices, since their width is unimportant in the final drawing) as well as the height. However, it does not guarantee a minimum width and height since this problem is NP-complete. It computes a layering with width at most w. The Coffman-Graham-Algorithm takes as input a reduced digraph, i.e., no transitive edges are present in the graph, and a width w. An edge (u, v) is called transitive if a path $(u = v_1, v_2, \dots , v_k = v)$ exists in the graph.

Observe that the absence of transitive edges does not affect the width of the layering significantly and that transitive edges can be found in linear time. The weakness of a simple greedy heuristic is illustrated in Figure 9.6. w is assumed to be 2. The graph in Figure 9.6 consists of $n/2$, $n \bmod 4 = 0$, isolated vertices and a directed path of $n/2$ vertices (Figure 9.5.8(a)). The greedy heuristic would probably assign the isolated vertices to the $n/4$ first layers. This would result in a layering of height $3n/4$ (Figure 9.6(b)). An optimal solution is depicted in Figure 9.6c).

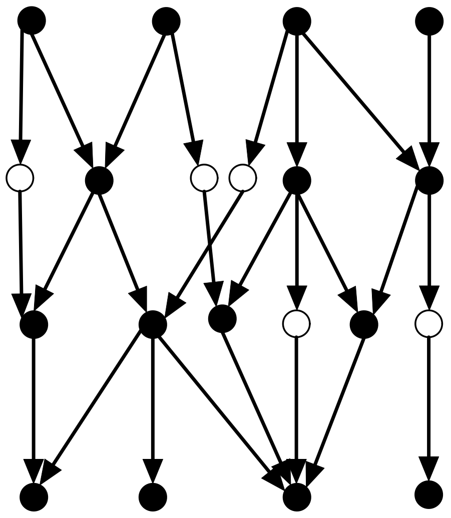

Figure 9.5: Introducing dummy vertices in the graph.

The greedy solution is far from optimal since it does not consider the long path in the graph. This is exactly what the Coffman-Graham-Algorithm tries to avoid. It proceeds in two phases. The first orders the vertices mainly by their distance from the source vertices of the graph, the second assigns the vertices to the layers. Vertices with large distances from the sources will be assigned to layers as close to the bottom as possible. We need a special lexicographical ordering on finite integer sets to describe the algorithm in more detail: Then $S < T$ if either

1. $S = 0$ and $T \neq 0$, or
2. $S \neq 0$, $T \neq 0$, and $\max(S) < \max(T)$, or
3. $S \neq 0$, $T \neq 0$, $\max(S) = \max(T)$ and $S - \{\max(S)\} < T - \{\max(T)\}$.

.

Given a fixed width greater or equal to three, the problem of finding a layering with minimum height is NP-complete. Suppose each vertex is a unit-time task to be executed in one of the processors of a multiprocessor and each edge (u, v) of digraph G is the precedence

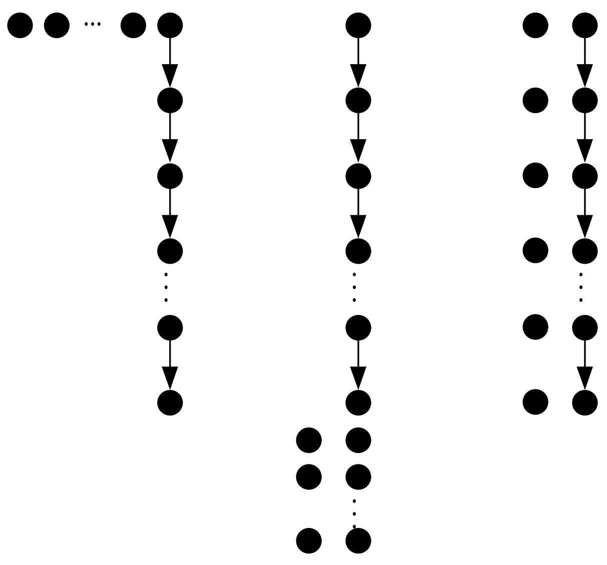

Figure 9.6: (a) A graph. (b) Greedy solution. (c) Optimal solution. The worst and the best layering for the given graph.

constraint that u must precede v. In the precedence constrained multiprocessor scheduling problem all tasks that are assigned to the W processors must be executed in time H. This problem is also NP-complete. The Coffman-Graham-Layering algorithm from the theory of mulitprocessor scheduling is a layering method which tries to minimize the width (ignoring dummy vertices, since their width is unimportant in the final drawing) as well as the height. However, it does not guarantee a minimum width and height since this problem is NP-complete.

Input: a reduced digraph $G = (V, E)$ and the desired maximum width, W.

Output: a proper layered digraph G.

```
COFFMAN-GRAHAM-LAYERING($G, w$)
 Order the vertices by assigning an integer label, $\pi(u)$, to each vertex:
 $i \leftarrow 1$
 repeat
         (a) Choose an unlabeled vertex, $v$, such that $\{\pi(v) : (u, v) \in E\}$
         is minimized. Use a lexicographic order on the sets.
         (b) Set $\pi(v) = i$.
   until $i = V$
 Assign vertices to layers, ensuring that no layer receives more than w vertices:
 $k \leftarrow 1$
 $L_1 \leftarrow \emptyset$
 $U \leftarrow \emptyset$
 while $U \neq V$
    do (a) Choose $u \in V - U$, such that every vertex in $\{v : (u, v) \in E\}$ is in $U$, and $\pi(u)$ is maximized
         (b) if $|L_k| < W$ and for every edge $(u, w)$, $w \in L_1 \cup L_2 \cup \ldots L_{k-1}$
```

```
15              then
16                  add $u$ to $L_k$
17              else  $k \leftarrow k+1,\ L_k \leftarrow \{u\}$
18          (c) Add $u$ to $U$.
```

Suppose that h_{min} is the minimum height of a layering of width W, then the algorithm above guarantees that the height of the layering is not too large, i.e that $h \leq (2 - \frac{2}{W})h_{\min}$. Thus the Coffman-Graham-Algorithm is an exact algorithm for $w \leq 2$.

One technique defines a cost function which is equal to the total vertical span of all the edges. Then the layer assignment problem is reduced to choosing the y-coordinates as before, but now subject to miminizing this cost function. It can be solved as an integer linear programming problem in a standard way.

9.1.3 Minimizing the Number of Dummy Vertices

Minimizing the number of dummy vertices implies minimum height. The objective to minimize the number of dummy vertices is equivalent to minimize the total edge span. To solve this problem we formulate it as an integer program. Given an acyclic digraph, each vertex u has a y-coordinate that satisfies the following layering properties:

1. $y(u)$ is an integer for each vertex u.
2. $y(u) \geq 1$ for each vertex u.
3. $y(u) - y(v) \geq 1$ for each $(u, v) \in E$.

Thus minimizing over $f = \sum_{(u,v)\in E}(y(u) - y(v) - 1)$ minimizes the total span of the edges and thus the number of vertices. Note that f measures the total span of the edges and consequently can be assumed as the number of dummy vertices. With the goal of minimizing the sum of edge spans ,an Integer Linear Programming formulation has been developed, that layers the graph accurately according to the specied dimensions. The contribution to the width of the layering of the dummy nodes that are introduced by the layering process is taken into account.

An example, where this formulation has been applied, is the graph *Grafo1002* from the graph database of the University of Rome introduced by Di Battista et al. in 1997. It is a DAG with 20 nodes and 21 edges. Several techniques exist to minimize the number of dummy vertices used. This is desirable since later steps depend on the number of vertices and because bends occur where dummy vertices lie and we want to minimize the number of bends. One technique defines a cost function which is equal to the total vertical span of all the edges. Then the layer assignment problem is reduced to choosing the y-coordinates as before, but now subject to miminizing this cost function. It can be solved as an integer linear programming problem in a standard way.

9.2 Crossing Reduction

The output of the layering step is a proper layered digraph with a (possibly large) number of crossings. Crossings do not convey any useful information about the structures represented in the graph. Thus, their minimization is among the most important aesthetic criteria.

What affects the number of crossings in a drawing of a layered digraph, is obviously the relative - not the absolute - position of vertices within each layer, i.e., their ordering. The crossing reduction problem is a combinatorial one, to choose for each layer the permutation of vertices that "produces" less crossings. Still, it is NP-complete, even if we consider a 2-layer digraph.

It seems reasonable therefore, to apply heuristic algorithms to solve this problem ([2], [4], [1]). All these methods use a layer-by-layer sweep along with a technique for reducing crossings between two layers.

9.2.1 The Layer-by-Layer Sweep

Let $G = (V, E) = (V_1, V_2, \dots, V_k, E)$ a k-layered graph, with disjoint sets $V_1, \dots, V_k$ of vertices that are assigned to layers $L_1, \dots, L_k$, using longest path layering or Coffman-Graham layering, for example. The edge set E consists of edges with endpoints in layers $L_{i-1},\ L_i,\ i = 2, \dots, k$ (their span equals 1 since the graph is proper).

The layer-by-layer sweep in a k-layered graph, proceeds in the following way: The relative position of vertices in the 1^{st} layer is held fixed, chosen at random at first, or "optimized" somehow. Then, we repeatedly change the order of vertices in the 2^{nd} layer so that the number of crossings between edges with endpoints in layers L_1 and L_2, is minimized. The sweep proceeds, keeping fixed the permutation in the 2^{nd} layer, and repermuting the vertices in the 3^{rd} layer, and so on. When all the layers have been processed, we go backwards, holding the vertex order in layer L_i fixed, and repermuting the vertices in layer L_{i-1}, for $i = k,\ k-1,\ \dots,\ 2$. The sweeps are repeated, from top-to-bottom and bottom-to-top, until no more improvement is observed. Another variation of the above technique considers a fixed vertex ordering in layers L_{i-1} and L_{i+1}, while repermuting the vertices in L_i layer in order to minimize crossings between edges in layers $L_{i-1},\ L_i,\ L_{i+1}$.

We conclude, then, that we have to solve a series of *two-layer crossing problems*: minimize crossings between two adjacent layers L_i and L_{i+1} with L_i fixed, reordering vertices in L_{i+1}.

Let $G = (V_1, V_2, E)$ a 2-layered, or bipartite graph, whose vertices are assigned to layers L_1 and L_2, and $E \subset V_1 \times V_2$, the set of edges.

For convenience, we describe the ordering of vertices in each layer with *x-coordinates*: $x_i = \{x_i(u)\ \forall u \in L_i,\ i = 1, 2\}$, that is, $x_i(u)$ is the ordinal position of u in the current ordering . Then, $cross(G, x_1, x_2)$ is the number of crossings between edges in E when the vertices in L_1, L_2 are ordered by x_1 and x_2, respectively, and

$$opt(G, x_1) = \min_{x_2} cross(G, x_1, x_2) \tag{9.3}$$

is the minimum number of crossings given that vertices in L_1 have the (fixed) relative positioning x_1. Then, the *two-layer crossing problem* can be formulated as an optimization problem:

Input: a two-layered digraph $G = (V_1, V_2, E)$ and an ordering x_1 of L_1.

Output: an ordering x_2 of L_2 that minimizes $cross(G, x_1, x_2)$

Since even the *two-layer* crossing problem is NP-complete, heuristic methods are appropriate; three popular heuristic methods are presented in subsequent sections. Also, in [4] it is shown that if the ordering in one layer is fixed, the problem can be transformed in a linear ordering problem that can be solved exactly in very short computational time, via the branch and cut method.

Let $n_1 = |V_1|$, $n_2 = |V_2|$, $m = |E|$, and $N_u = \{v \in V \mid e = (u,v) \in E\}$ denotes the set of neighbors of $u \in V = V_1 \cup V_2$. Let δ^i_{uv}, $i = 1,2$ a binary vector, $\delta^i_{uv} = 1$ if u is to the left of v ($x_i(u) < x_i(v)$), and zero otherwise. Thus, the vector $\delta^i \in \{0,1\}^{\binom{n_i}{2}}$ describes adequately the orderings x_i $(i = 1,2)$. With the permutation x_1 of V_1 fixed, the number of crossings among the edges adjacent to $u \neq v \in V_2$ if $x_2(u) < x_2(v)$, is then:

$$c_{uv} = \sum_{k \in N(u)} \sum_{l \in N(v)} \delta^1_{lk}. \tag{9.4}$$

The *crossing number* c_{uv} depends therefore only on the relative position of u, v; it equals the number of pairs (u,k), (v,l) of edges with $x_1(l) < x_1(k)$. Obviously, $c_{uu} = 0$ for all $u \in V_2$. Table 9.1 below, shows the crossing numbers for all pairs of vertices of V_2 in the 2-layer digraph in Figure (9.7).

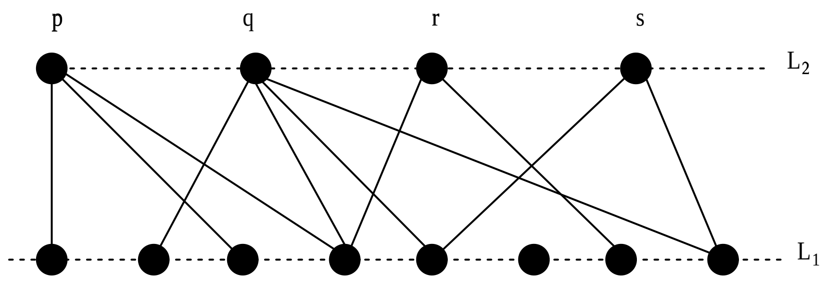

Figure 9.7: Drawing of a 2-layer digraph

	p	q	r	s
p	0	2	0	0
q	3	0	3	1
r	5	4	0	1
s	6	5	3	0

Table 9.1: Crossing numbers for all pairs of vertices in layer 2.

Lemma 9.1 gives $cross(G, x_1, x_2)$ as a function of crossing numbers, and a lower bound for $opt(G, x_1)$ that seems trivial but, according to [4], it is very good

Lemma 9.1 *Let $G = (V_1, V_2, E)$ be a two-layer digraph and x_1, x_2 orderings of V_1, V_2 respectively. Then*

$$cross(G, x_1, x_2) = \sum_{x_2(u) < x_2(v)} c_{uv}, \tag{9.5}$$

and

$$opt(G, x_1) \geq \sum_{u,v} \min(c_{uv}, c_{vu}) = \sum_{u=1}^{n_2 - 1} \sum_{v=u+1}^{n_2} \min(c_{uv}, c_{vu}), \tag{9.6}$$

where we sum over all pairs of vertices of the second layer.

Proof. The relation 9.5 is obvious. Inequality 9.5 holds because every ordering x_2 of V_2 has either $x_2(u) < x_2(v)$ or $x_2(v) < x_2(u)$.

□

The *general two-layer crossing problem* permits both layers to be permuted; as shown in [4], the number of crossings is considerably less than with one layer fixed. For large number of vertices, however, two-sided crossing minimization is possible only through a modification: the first layer is kept fixed while the second is "optimized", then the second layer is fixed and the first is "optimized", and so on; the iterations continue until the crossing number does not decrease any more. The following heuristic methods, as well as the exact method mentioned in [4], can be used either for one-sided, or for two-sided crossing minimization.

9.2.2 Sorting Methods

Several simple heuristic methods require precomputation of the crossing numbers, a process that seems to require $O(|E|^2)$ time but - if implemented more carefully - can take just $O\left(\sum_{u,v} c_{uv}\right)$.

The greedy-switch heuristic proceeds like bubble-sort, switching adjacent pairs of vertices if the number of crossings is reduced. Algorithm *Adjacent-Exchange* uses precomputed crossing numbers c_{uv} (there is no need to compute them again since they depend only on the relative position of u and v) and needs $O(n_2^2)$ time:

Input: a two-layer digraph $G = (V_1, V_2, E)$ and an ordering x_1 of vertices $\in V_1$.

Output: an ordering x_2 of vertices $\in V_2$.

```
ADJACENT-EXCHANGE($G$)
1   Choose an initial order for $V_2$
2   repeat
3         for each adjacent pair of vertices $i, j \leftarrow i+1 \in V_2$
4             do if $c_{ij} > c_{ji}$
5                 then exchange vertices $i, j$
6     until the number of crossings is not reduced any more.
```

An example of worst case performance for greedy-switch heuristic is depicted in Figure (9.8).

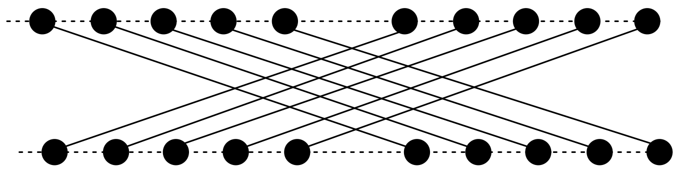

Figure 9.8: Pathological case for adjacent-exchange heuristic.

Split heuristic chooses a *pivot vertex* $p \in V_2$ and places each vertex $q \neq p \in V_2$ to the left or right of p, depending on whether it would result in fewer crossings. As in quicksort, this step is applied recursively to order the sets of vertices to the left and right of p. Worst-case running time of split is $O(n_2^2)$, but, on the average, it runs in $O(n_2 \log n_2)$ time (as quicksort):

Input: a two-layer digraph $G = (V_1, V_2, E)$ and an ordering x_1 of vertices of V_1.

Output: an ordering x_2 of vertices of V_2.

```
SPLIT($V_2$)
1  if $V_2 \neq \emptyset$
2     then (a) $p \in V_2$
3          (b) $V_{left} \leftarrow \emptyset$; $V_{right} \leftarrow \emptyset$
4          (c) for each vertex $q \in V_2 - \{p\}$
5               do if $c_{qp} \leq c_{pq}$
6                    then $V_{left} \leftarrow V_{left} \cup q$
7                    else $V_{right} \leftarrow V_{right} \cup q$
8          (d) SPLIT($V_{left}$)   SPLIT($V_{right}$)
9           The output is the concatenation of their outputs.
```

Another sorting algorithm (mentioned in [4]) is *greedy-insert*: like insertion sort, the next vertex p is chosen so that the number of crossings that edges adjacent to p make with edges adjacent to vertices on the left of p, is minimized. All three of the above methods use precomputed crossing numbers, so they all have non-linear time complexity.

9.2.3 The Barycenter and Median Methods

The *averaging* heuristics, i.e., the *barycenter* and the *median* methods, are very common methods to order vertices in V_2. They simply compute the average position of their neighbors in V_1, the barycenter or median, respectively, and sort vertices according to these numbers.

More precisely, the *barycenter method* estimates the new ordinal number $x_2(u)$ of each vertex $u \in V_2$, as the average position of its neighbors in V_1:

$$x_2(u) = avg(u) = \frac{1}{deg(u)} \sum_{v \in N(u)} x_1(v)., \tag{9.7}$$

where $N(u)$ denotes the set of neighbors of u and $deg(u)$ denotes their number, i.e. the degree of u. If the barycenter of two vertices coincides, their order is choosen arbitrarily. The running time of the method is linear. We denote the number of crossings with this method as $avg(G, x_1)$.

The *median* method estimates the new ordinal number $x_2(u)$ of each vertex $u \in V_2$, as the median of the positions of its neighbors in V_1: $x_2(u) = med(u)$. The running time of this method is linear, too, that is, $O(deg(u))$. The number of crossings is denoted as $med(G, x_1)$.

If the set $N(u) = \{v_1, v_2, \ldots, v_j\}$ of neighbors of u is already sorted, i.e., $x_1(v_1) < x_1(v_2) < \ldots < x_1(v_j)$, and the number of neighbors of $u \in V_2$ is odd ($j \bmod 2 = 1$), then the median is uniquely defined as

$$med_{odd}(u) = x_1(v_{\lfloor j/2 \rfloor}). \tag{9.8}$$

When $|N(u)|$ is even ($j \bmod 2 = 0 \Rightarrow \lfloor j/2 \rfloor = j/2$), there are two median values, the *left*: $med_{el}(u)$ (the same as $med_{odd}(u)$ defined in 9.8), and the *right*, $med_{er}(u)$:

$$med_{er}(u) = x_1(v_{j/2+1}). \tag{9.9}$$

In this case, in [1] an interpolated value between the two medians is used, biased towards the side where vertices are more closely packed:

$$med_{even}(u) = \frac{med_{el}(u) * (x_1(v_j) - x_1(v_{j/2+1})) + med_{er}(u) * (x_1(v_{j/2}) - x_1(v_1))}{x_1(v_j) - x_1(v_{j/2+1}) + x_1(v_{j/2}) - x_1(v_1))} \tag{9.10}$$

If $N(u) = \emptyset$, then $med(u) = 0$ - in this case, [1] do not change at all $x_2(u)$, and sort the other vertices $w \in V_2$ with $N(w) \neq \emptyset$, into the remaining positions.

If the median of two vertices coincides, and their degrees do not have the same parity, then we place the odd degree vertex on the left of the even degree vertex. If their parity is the same, we can choose their order arbitrarily. [1] flip vertices with equal medians during the sorting phase on every other forward and backward traversal.

The following theorem proves that both methods give a zero-crossing drawing if this actually exists.

Theorem 9.1 *Let $G = (V_1, V_2, E)$ be a two-layer digraph and x_1 an ordering of V_1. If $opt(G, x_1) = 0$, then $avg(G, x_1) = med(G, x_1) = 0$.*

On the other hand, none of these methods is guaranteed to give an optimum solution. Figures (9.9) and (9.10) present worst cases for the averaging methods, that imply a lower bound to the crossings they achieve. This lower bound is given in the following lemma:

Lemma 9.2 *1. For each n there is a two-layer digraph $G = (V_1, V_2, E)$ with $|V_1| = n$, $|V_2| = 2$, and an ordering x_1 of $|V_1|$, such that*

$$\frac{avg(G, x_1)}{opt(G, x_1)} \propto \Omega(\sqrt{n}). \tag{9.11}$$

2. For each n there is a two-layer digraph $G' = (V_1', V_2', E')$ with $|V_1'| = n$, $|V_2'| = 2$, and an ordering x_1 of $|V_1'|$, such that

$$\frac{med(G', x_1)}{opt(G', x_1)} \geq 3 - O(\frac{1}{n}). \tag{9.12}$$

Proof. For the relation concerning barycenter method, we will use the graph in Figure (9.9). Let $v_1, v_2, \ldots, v_n$ be the vertices of V_1, positioned with this order, where $n = k^2+k-1$, and u, w the 2 vertices of V_2. If $N(w) = \{v_{k^2}\}$ and $N(u) = \{v_1, v_{k^2+1}, v_{k^2+2}, \ldots, v_{k^2+k-1}\}$, then $avg(w) = k^2$ and: $avg(u) = \frac{1+k^2+1+\cdots+k^2+k-1}{k} = k^2 - k/2 - 1/2 < k^2$. Since $avg(u) < avg(w)$, the barycenter method places u to the left of w and $avg(G, x_1) = k - 1$. But $opt(G, x_1) = 1$, so $\frac{avg(G,x_1)}{opt(G,x_1)} = k - 1$, where k is $O(\sqrt{n}$, and the bound is proven.

In Figure (9.10) the digraph is a worst case example for the median method. V_1 has $n = 4k + 2$ vertices, V_2 has the vertices u, w and $N(u) = \{v_{k+1}, \ldots, v_{2k+1}, v_{3k+3}, \ldots, v_{4k+2}\}$, $med(u) = 2k + 1$, $N(w) = \{v_1, \ldots, v_k, v_{2k+2}, \ldots, v_{3k+2}\}$, and $med(w) =$

$2k + 2 > med(u)$. Then u is placed to the left of w and $med(G', x_1) = 2k(k+1) + k^2$ while $opt(G', x_1) = (k+1)^2$. So, we have that:

$$\begin{aligned} \frac{med(G', x_1)}{opt(G', x_1)} &= \frac{2k(k+1) + k^2}{(k+1)^2} \\ &= \frac{3(k+1)^2 - 4k - 4 + 1}{(k+1)^2} \\ &= 3 - \frac{4}{k+1} + \frac{1}{(k+1)^2} \\ &= 3 - \frac{16}{n+2} + \frac{16}{(n+2)^2} \\ &\geq 3 - O(\frac{1}{n}). \end{aligned} \tag{9.13}$$

□

The lower bound to the number of crossings through barycenter method is proportional to $\sqrt{n} opt(G, x_1)$; that is, its ratio to $opt(G, x_1)$ grows with increasing n. On the contrary, the same lower bound for median heuristic is proportional to a *constant* times $opt(G, x_1)$, or $med(G, x_1, x_2) \propto 3 opt(G, x_1)$. Therefore, median method is better than barycenter in terms of their theoretical lower bounds in performance.

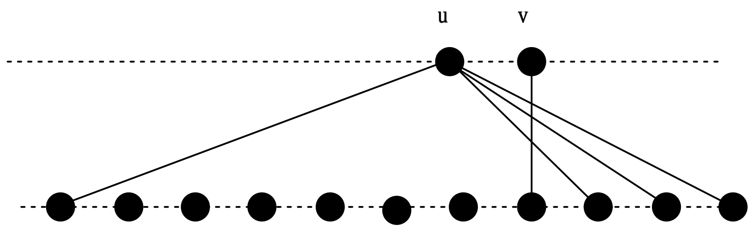

Figure 9.9: Pathological case for barycenter method.

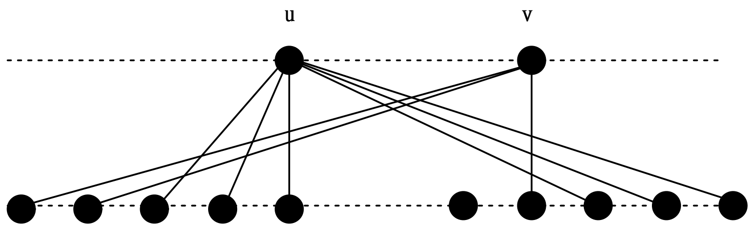

Figure 9.10: Pathological case for median heuristic.

The median method has one more theoretical advantage over barycenter method, since there is an upper bound on $med(G, x_1)$ according to the following theorem:

Theorem 9.2 *For all two-layer digraphs $G = (V_1, V_2, E)$ and all vertex orderings x_1 of V_1, $med(G, x_1) \leq 3 opt(G, x_1)$.*

Proof. Let u and v vertices of V_2, and u be positioned to the left of v with median heuristic. As depicted in Figure (9.11), we can partition edges incident with u and v, in four groups, according to their relative positions:

$$\begin{aligned}
\alpha &= \{(u, w_\alpha) \in E : x_1(w_\alpha) < med(u)\} \\
\beta &= \{(v, w_\beta) \in E : x_1(w_\beta) > med(v)\} \\
\gamma &= \{(v, w_\gamma) \in E : x_1(w_\gamma) < med(v)\} \\
\delta &= \{(u, w_\delta) \in E : x_1(w_\delta) > med(u)\}
\end{aligned}$$

where w_i a vertex of V_1 that is an end-vertex of an edge belonging in group i. These groups, of course, do not include the edges that join u and v to their respective medians, denoted as e_u and e_v respectively.

If we refer to the number of edges in each group as $a = |\alpha|$, $b = |\beta|$, $c = |\gamma|$, and $d = |\delta|$, we can prove that

$$c_{vu} \geq ab + a + b + \epsilon, \tag{9.14}$$

where $\epsilon = 0$ if $med(u) = med(v)$, and $\epsilon = 1$ in the opposite case. Indeed, if v is positioned to the left of u, as shown in Figure (9.12), all edges in α group will cross e_v (since $x_1(w_\alpha) < med(u) \leq med(v)$), as well as all edges in β group (since $x_1(w_\alpha) < med(u) < x_1(w_\beta)$, yielding $a + ab$ crossings. Also, all edges in β group will cross e_u (since $x_1(w_\beta) > med(v) \geq med(u)$), resulting in b crossings. Concerning ϵ, if $med(u) = med(v)$, edges e_u and e_v do not cross, and $\epsilon = 0$. On the otherhand, if $med(u) \neq med(v)$, edge e_u will cross e_v, and $\epsilon = 1$. Inequality (9.14) follows.

Now, if u is positioned to the left of v, edges in α group will not cross any edge of β group. Edges e_u and e_v cannot cross, too; see Figure (9.11). Then:

$$c_{uv} \leq ac + cd + bd + c + d. \tag{9.15}$$

If the degree of u is odd, then the number of edges in group α equals that of δ (by definition of groups); if the degree of u is even, then $a + 1 = d$. If $deg(v)$ is odd, $c = b$, while if it is even, $c + 1 = b$. In any case, we have that $d \leq a + 1$ and $c \leq b$. Substituting in 9.15 we get

$$c_{uv} \leq 3ab + a + 3b + 1. \tag{9.16}$$

Next we have to show that $c_{uv} \leq 3c_{vu}$ in order to prove the theorem. Assume that $c_{uv} > 3c_{vu}$, instead, or

$$c_{uv} - 3c_{vu} > 0. \tag{9.17}$$

Multiplying 9.14 with -3, its direction is reversed; adding it to 9.16 and using 9.17, implies that $-2a - 3\epsilon + 1 > 0$, or $2a + 3\epsilon - 1 < 0$. Since $a \geq 0$, $\epsilon \geq 0$ and both are integers,

$$a = \epsilon = 0. \tag{9.18}$$

Then $\alpha = \emptyset$ and

$$deg(u) = |N(u)| = |\alpha \cup \{e_u\} \cup \delta| = 0 + 1 + d \leq 2. \tag{9.19}$$

Using 9.14, 9.15 and the inequality $c \leq b$, we can show that $d \neq 0$. Then $d = 1$, and $deg(u) = 2$ according to 9.19.

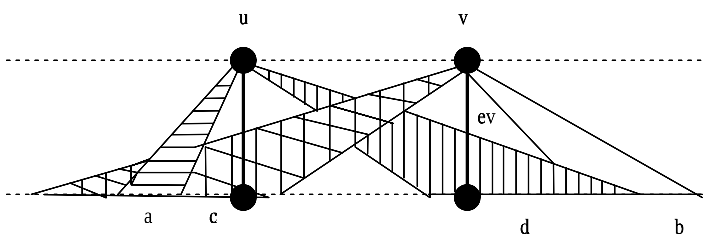

Figure 9.11: Edges belonging to groups α, β, γ, and δ, when u is positioned to the left of v.

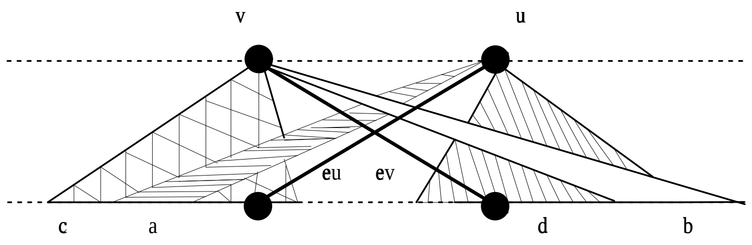

Figure 9.12: Edges belonging to the same groups as above, when v is positioned to the left of u.

By definition of ϵ, and 9.18, $med(u) = med(v)$; since u is positioned to the left of v, and the degree of u is even, the degree of v must be even, too, otherwise the median heuristic would place it to the left of u. Then, $c + 1 = b$ as mentioned above, or $c = b - 1$. Using this equality and 9.18 in 9.14 and 9.15, it follows that $c_{vu} \geq b$ or $-3c_{vu} \leq -3b$, and $c_{uv} \leq 3b - 1$. Summing the last inequalities and using 9.17, implies that $-1 > 0$, that is a contradiction.

So, for any vertices $u, v \in V_2$, it holds that $c_{uv} \leq 3c_{vu}$. This is equivalent to:

$$c_{uv} \leq 3 \min(c_{vu}, c_{uv}).$$

We sum the last inequality over all pairs $u, v \in V_2$, with $x_2(u) < x_2(v)$, and use both relations of Lemma 9.1, to conclude that

$$c_{uv} \leq 3 \min(c_{vu}, c_{uv}).$$

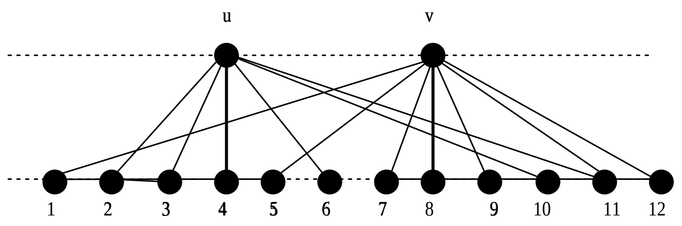

Figure 9.13: A two-layer digraph with groups of edges $\alpha = \{(u,2),(u,3)\}$, $\beta = \{(v,9),(v,11),(v,12)\}$, $\gamma = \{(v,1),(v,5),(v,7)\}$, and $\delta = \{(u,6),(u,10),(u,11)\}$

.

□

Considering only low-degree digraphs and applying a similar - but case-by-case - argument, gives the following result:

Theorem 9.3 *For all two-layer digraphs $G = (V_1, V_2, E)$ with $deg(u) \leq 3\,\forall u \in V_2$ and all vertex orderings x_1 of V_1, $med(G, x_1) \leq 2opt(G, x_1)$.*

9.2.4 Integer Programming Methods

The two layer crossing minimization problem can be formulated as an integer program (a linear ordering problem).

Equation 9.4 gives the number of crossings c_{uv} among the edges adjacent to $u \neq v \in V_2$ if $x_2(u) < x_2(v)$, when the permutation x_1 of V_1 is fixed. If $x_2(v) < x_2(u)$, the number of crossings is denoted as c_{vu} and is given by:

$$c_{vu} = \sum_{k \in N(u)} \sum_{l \in N(v)} \delta^1_{kl}. \tag{9.20}$$

where δ^1_{kl} a binary vector (already defined in subsection 9.2.1), $\delta^1_{kl} = 1$ if k is to the left of l, or $x_1(k) < x_1(l)$, and $\delta^1_{kl} = 0$ otherwise. Below, we use δ^2_{uv} to denote the relative position of vertices u, v in second layer. Obviously,

$$\delta^2_{vu} = 1 - \delta^2_{uv}. \tag{9.21}$$

Combining equations 9.4, 9.20, 9.21 and 9.5 from Lemma 9.1, gives:

$$\begin{aligned} cross(G, x_1, x_2) &= \sum_{x_2(u) < x_2(v)} (c_{uv}\delta^2_{uv} + c_{vu}(1 - \delta^2_{uv})) \\ &= \sum_{x_2(u) < x_2(v)} (c_{uv} - c_{vu})\delta^2_{uv} + \sum_{x_2(u) < x_2(v)} c_{vu}. \end{aligned} \tag{9.22}$$

Since the problem is to minimize $cross(G, x_1, x_2)$ and the term $\sum_{x_2(u)<x_2(v)} c_{vu}$ in 9.22 is a constant, the equivalent linear ordering problem is:

Minimize $z = \sum_{x_2(u)<x_2(v)} (c_{uv} - c_{vu})\delta^2_{uv}$, subject to:

1. $0 \leq \delta^2_{uv} + \delta^2_{vw} - \delta^2_{uw} \leq 1$ for $1 \leq x_2(u) < x_2(v) < x_2(w) \leq n_2$
2. $0 \leq \delta^2_{uv} \leq 1$ for $1 \leq x_2(u) < x_2(v) \leq n_2$
3. $\delta^2_{uv} \in \{0, 1\}$ for $1 \leq x_2(u) < x_2(v) \leq n_2$

If z^* is the optimum value of z, the minimum number of crossings is

$$z^* + \sum_{x_2(u)<x_2(v)} c_{vu}.$$

The constraints ensure that the solutions actually correspond to all permutations x_2 of V_2.

The problem is NP-hard and a complete linear description of feasible solutions is unlikely to be found and exploited algorithmically.

In [4] it is shown that, if the third set of constraints, the integrality conditions, is dropped, the solution can be given with standard linear programming techniques. In that case, the first set of constraints, the $2\binom{n_2}{3}$ *3-cycle* inequalities, and the second set, the $2\binom{n_2}{2}$ *hypercube* inequalities, define a relaxation of the integer program that is solved in [4] with a *branch and cut* algorithm. Since the set of 3-cycle inequalities is large $(O(n_2^3))$, the algorithm uses a cutting plane approach. It starts with the hypercube inequalities which are solved implicitly by the LP-solver; 3-cycle constraints are used iteratively to cut the solution space, after an LP has been solved: the algorithm continues to add violated 3-cycle constraints and to delete nonbinding 3-cycle constraints, until the relaxation is solved. The algorithm stops if the solution is integral; if it is not, the algorithm is applied recursively to two subproblems in one of which a fractional x_{uv} is set to 1 and in the other set to 0. The solution is optimal for digraphs of limited size (n_2 can be up to 60).

This method is exact, i.e., it is guaranteed to find the optimum solution, in contrast to the methods described in previous subsection. However it does not always terminate in polynomial time (see subsection 9.2.6).

9.2.5 The Two-Layer Crossing Problem on Dense Digraphs

Let $G = (V_1, V_2, E)$ be a dense, two-layer digraph; then, since for any vertices u and v of V_2, the number of their common neighbors, χ_{uv}, is large, both crossing numbers, c_{uv} and c_{vu}, will be large. This explains intuitively why, for dense digraphs, $cross(G, x_1, x_2)$ is close to $opt(G, x_1)$, for any ordering x_2 of V_2.

The following lemma sets bounds on c_{uv} in terms of χ_{uv}:

Lemma 9.3 *If u and v vertices of V_2, then*

1. $c_{uv} + c_{vu} + \chi_{uv} = deg(u)\, deg(v)$
2. $c_{uv} \geq \binom{\chi_{uv}}{2}$
3. $c_{uv} \leq deg(u)\, deg(v) - \binom{\chi_{uv}+1}{2}$

Theorem 9.4 *Let $G = (V_1, V_2, E)$ be a two-layer digraph, $|V_1| = |V_2| = n$, and $E = \epsilon n^2$. Then*

$$\lim_{\epsilon \to 1} \frac{\max_{x_2} cross(G, x_1, x_2)}{opt(G, x_1, x_2)} = 1.$$

Proof: Inequality 9.3.2 implies that since $\chi_{uv} = \chi_{vu}$ is large, both c_{uv} and c_{vu} will be large.

Multiplying inequality 9.1.2 by -1 (its direction is reversed) and adding it to 9.1.1, we have:

$$\begin{aligned} cross(G, x_1, x_2) - opt(G, x_1) &\leq \sum_{x_2(u) < x_2(v)} (c_{uv} - \min(c_{uv}, c_{vu})) \\ &\leq \sum_{x_2(u) < x_2(v)} |c_{uv} - c_{vu}|. \end{aligned} \tag{9.23}$$

Also, multiplying inequality 9.3.2 by -1 and adding it to 9.3.3 (either relation could refer to c_{vu} or c_{uv}, so we will refer to the absolute value of this difference), we have:

$$\begin{aligned} |c_{uv} - c_{vu}| &\leq deg(u)deg(v) - \binom{\chi_{uv}+1}{2} - \binom{\chi_{uv}}{2} \\ &\leq deg(u)deg(v) - \chi_{uv}^2. \end{aligned} \tag{9.24}$$

Summing last inequality 9.24 over all pairs u, v of V_2 and using inequality 9.23, we get:

$$\sum_{x_2(u)<x_2(v)} \left(deg(u)deg(v) - \chi_{uv}^2\right) \geq cross(G, x_1, x_2) - opt(G, x_1) \tag{9.25}$$

It is simple to estimate the following upper bound:

$$\sum_{x_2(u)<x_2(v)} deg(u)deg(v) \leq \frac{|E|^2}{2} = \frac{\epsilon^2 n^4}{2}. \tag{9.26}$$

Next, a lower bound on the sum of the second term of the same, left-hand side of 9.25 can be obtained through the following steps:

$$\sum_{x_2(u)<x_2(v)} \chi_{uv} = \sum_{w \in V_1} \binom{deg(w)}{2} \tag{9.27}$$

and

$$\sum_{w \in V_1} deg(w) = |E| = \epsilon n^2. \tag{9.28}$$

Then

$$\sum_{w \in V_1} \binom{deg(w)}{2} \geq n \binom{\epsilon n}{2} \tag{9.29}$$

and, from 9.27 and 9.29, the lower bound follows:

$$\sum_{x_2(u)<x_2(v)} \chi_{uv}^2 \geq \binom{n}{2} \left(\frac{n\binom{\epsilon n}{2}}{\binom{n}{2}} \right)^2 = \frac{\epsilon^2 n^3 (\epsilon n - 1)^2}{2(n-1)}. \tag{9.30}$$

From 9.25 9.26 and 9.30 we get:

$$\begin{aligned} cross(G, x_1, x_2) - opt(G, x_1) &\leq \frac{\epsilon^2 n^4}{2} - \frac{\epsilon^2 n^3 (\epsilon n - 1)^2}{2(n-1)} \\ &\leq \epsilon^2 n^2 \left(\epsilon n - \frac{1}{2} \right). \end{aligned} \tag{9.31}$$

Through a similar argument, it follows that:

$$opt(G, x_1) \geq \frac{\epsilon^2 n^4}{2} - o(n^3). \tag{9.32}$$

Division by parts of 9.31 and 9.32 gives that $\frac{cross(G,x_1,x_2) - opt(G,x_1)}{opt(G,x_1)}$ is $O(\frac{\epsilon}{n})$ and the theorem is proved.

$\square$

9.2.6 Remarks on the Two-Layer Crossing Problem

The median method is the only heuristic with a theoretical upper bound to the number of crossings it achieves; moreover, it runs in polynomial time. However, comparative tests on pseudo-random digraphs have shown that the barycenter method may outperform the median in terms of optimality of solution, having the same (linear) running time.

In [4] various heuristics are compared with their exact method 9.2.4), both for one-sided and for two-sided crossing minimization. For sparse graphs of medium size - in automatic graph drawing the graphs are usually sparse - the exact computation by branch and cut algorithm is faster than many of the heuristics, except barycenter and median method. In summary, barycenter method turned out the fastest method with a solution very close to optimum.

For all heuristics, a considerable performance gain was achieved if the same experiment was repeated 10 times , starting with random orderings of nodes in V_1, and taking the best solution found.

Also, an hybrid approach, adopted in [1], yields good results on "real world" digraphs:

1. Determine an initial ordering of nodes in each rank that avoids obvious crossings (through depth-first or breadth-first search starting with vertices of the first layer).

2. Iteratively permute vertices in each layer, based on the weighted median heuristic, as long as the number of crossings improves.

3. Optimize the output through greedy switch (Algorithm *Adjacent Exchange*).

9.3 Horizontal Coordinate Assignment

The computation of the horizontal coordinates has mainly two different objectives. The layout should have as few bends as possible. As mentioned before, bends only occur at dummy vertices, unless the expansion of the vertices is very large. In some applications both straight edges and vertical edges are preferred. Approaches to this problem are presented below. Given the layers of the digraph and the order of the vertices within each layer, we want to find the best x-coordinate of each vertex. By 'best' we mean the position that minimizes the amount of bending. Since the number of dummy vertices is set in the Layer Assignment step, we can only shift horizontally the vertices, maintaining the order we selected in the Crossing Reduction step. One way is to attempt to draw each edge as straight as possible, with no preferred direction. Suppose we want some edge, $w_1 w_n$, to be as straight as possible. Then we could try to fit dummy vertices in between w_1 and w_n, call them $w_2, w_3, \ldots, w_{n-1}$, to the line:

$$\frac{i-1}{n-1} = \frac{x(w_i) - x(w_1)}{x(w_n) - x(w_1)} \tag{9.33}$$

(Note that $y(w_i) = i$ because the layers are evenly spaced). Then we turn the line equation into a cost function by placing all terms on the same side and squaring:

$$Cost(w_1 w_n) = \sum_{i=2}^{n-1} (x(w_i) - \alpha_i)^2, \tag{9.34}$$

where

$$\alpha_i = \frac{i-1}{n-1} x(w_n) - x(w_1)) + x(w_1). \tag{9.35}$$

Because we need such a term for each edge, we sum all these terms to obtain our cost function which we can then minimize using some standard method. We do not want to lose our crossing reduction step, so we must also subject our solution to the constraint: $x(u) - x(v) \geq \delta$ for all $u,\ v$ in the same layer, where u lies to the left of v. In an analogous manner, we could try to keep the edges as close to the vertical as possible or some other orientation that is desirable for a particular application. The problem of finding a layout with as straight edges as possible can be formulated as follows. Consider a directed path $p = (v_1, v_2, \dots, v_k)$ where $v_2, v_3, \dots, v_{k-1}$ are dummy vertices. We call this an *edge-path.* If the edge-path would be drawn straight, the dummy vertices would satisfy:

$$x(v_i) - x(u_1) = \frac{i-1}{k-1}(x(u_k) - x(u_1)) \tag{9.36}$$

for all $1 < i < k$. Observe that this formula is only valid for equidistant layers, but it is straightforward to adjust this formula for unequal layer distances. To be able to state the objective function more compact, we introduce the term

$$\overline{x}(v_i) := \frac{i-1}{k-1}(x(u_k) - x(u_1)) + x(u_1) \tag{9.37}$$

which would be the the x-coordinate of v_i if it would lie on the straight line between $x(v_1)$ and $x(v_k)$. We can now formulate a measure for the deviation of the path from a straight line

$$dev(p) = \sum_{i=2}^{k-1} (x(u_i) - \overline{x}(u_i))^2 \tag{9.38}$$

To make the edges as straight as possible, we minimize the sum:

$$\sum_{p\ is\ edge-path} dev(p) \tag{9.39}$$

subject to the constraints

$$x(w) - x(u) \geq p(w, v) \tag{9.40}$$

for all pairs w, v of vertices in the same layer with w to the right of v. The constraints ensure that the ordering within each layer computed by the crossing reduction step is preserved and that the horizontal distance $p(w, v)$ between the vertices is observed. The value $p(w, v)$ usually is calculated from the size of the vertices and the requested minimum horizontal distance between two succeeding vertices. An optimal solution to this optimization problem may result in exponential width of the drawing and thus, if the width should be kept small, further inequalities would have to be added. The main disadvantage is that since this problem has a ,ratic objective function, it can only be solved to optimality for small instances.

Another objective is to draw the lines as close to vertical lines as possible. In this case the objective function can be stated as (Gansner et al., 1993)

$$\sum_{(u,v)\in E} \Omega(u,v)\omega(u,v)|x(u)-x(v)| \tag{9.41}$$

where ω is a measure for the importance of an edge and Ω denotes an internal weight for straightening long edges. Therefore, the authors suggest higher priorities for edges between dummy vertices than between the other vertices ($\Omega(e) = 8$ if both end vertices are dummy vertices, $\Omega(e) = 2$ if exactly one end vertex is a dummy vertex and $\Omega(e) = 1$ otherwise). The introduction of a weight function may improve the layouts computed by the preceding model as well.

Another idea to solve this problem efficiently was introduced in (Gansner et al., 1993). Gansner et al. construct an auxiliary graph on which this problem transforms to a layering problem introduced which can be solved easily. The x-coordinates correspond to the layers and vice versa.

The auxiliary graph $G_a = (V_a, E_a)$ contains as vertices all vertices of G plus a vertex for each edge in G. Hence, $V_a = V \cup [uv]|(u,v) \in E$. We introduce two kinds of edges in G_a. The first class of edges encodes the original edges and is needed to eliminate the absolute values in the objective function. For every edge $(u,v) \in E$, we introduce two edges $([uv],u)$ and $([uv],v)$ in G_a. We define

$$\omega_a([uv],u) = \omega_a([uv],v) = \Omega(u,v)\omega(u,v) \tag{9.42}$$

and

$$\lambda_a([uv],v) = \lambda_a([uv],v) = 0. \tag{9.43}$$

The second class of edges separates the vertices with the same rank. If v is a left neighbor of w in G, we insert an edge (v,w) in E_a and define $\omega_a(v,w) = 0$ and $\lambda_a(v,w) = p(v,w)$.

A solution of the layering on G_a corresponds to a solution of the positioning problem on G and that both have the same cost. Let a solution of the positioning problem on G be given. Assign $[uv]$ to the layer $\min\{x(u), x(v)\}$. Conversely, in an optimal layer assignment in G_a, the vertex $[uv]$ lies in either the layer of u or the layer of v. Thus, one of the edges $([uv],u)$, $([uv],v)$ has length 0 and the other has length $|x(u)-x(v)|$. Hence, optimality in G_a implies optimality in G and a layering for G_a gives a solution for G.

Another possibility is to obtain the x-coordinates by an improvement heuristic which can roughly be stated like the following:

```
A Heuristic()
1   Choose initial coordinates
2  while some conditions hold
3      do
4           positioning
5           straightening
6           packing
```

One possibility for computing an initial solution is to position the vertices with minimal distance from left to right in the order given by the crossing minimization.

In the positioning phase essentially the ideas used in the previous section for crossing reduction between two layers might be applied, like the median or barycenter heuristic. Another idea is to think of the vertices as balls and the edges as strings of a pendulum (Sander, 1996b).

Since these strategies compute layouts with many bends, in the straightening phase one tries to assign paths of dummy vertices to the same x-coordinate. The edges can be seen as rubber bends with vertices at both ends and the dummy vertices in between. This enlarges the drawing in x-direction of course.Hence, the drawing is compressed by moving the vertices closer together again without introducing new bends. These steps might be iterated to obtain a satisfying solution.

9.4 Cycle Removal

In this section, we will address solution methods for the *maximum acyclic subgraph problem*: find a maximum set $E_a \subset E$, such that the graph (V, E_a) contains no cycles (see Figure 9.14). The problem is often stated as the *feedback arc set problem*: Find a minimum set $E_f \subset E$ such that the graph $(V, E_f \subset E)$ contains no cycles. Since we do not want to lose the information concerning the fact that two vertices are adjacent or not, the edges in $E_a \subset E$ will be reversed. It is an easy exercise to show that the resulting graph is acyclic. Only a small subset of edges should be reversed so that the 'flux' of the drawing is fairly smooth when the edges are restored to their proper direction and viewed. We can rephrase the problem in terms of vertex sequences. Suppose we draw the vertices of a digraph, G, along a horizontal line. Then the vertex sequence of that drawing of G is simply the list of the vertices from left to right as they appear on the line. To find a minimum set of feedback edges, we need to find the vertex sequence of the graph that has the least number of edge, pointing 'backwards', in this case from right to left. This problem is NP-complete and so we use heuristics to solve it. A simple depth-first search on the digraph can work well but may reverse up to $|E| - |V| - 1$ edges. An arbitrary sequence of G might also work well, and if it doesn't, then we can just reverse the order. This guarantees that no more than half the edges are reversed.

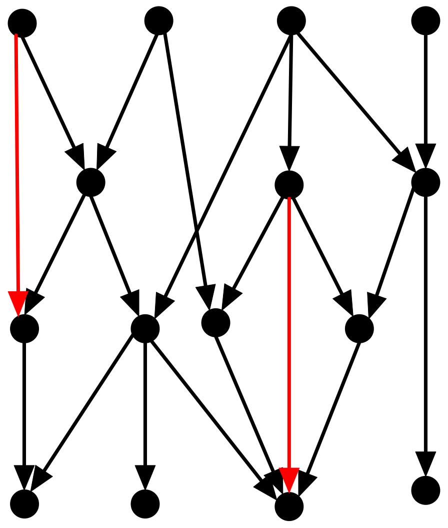

Figure 9.14: The graph made acyclic by reversing two edges.

Unfortunately, the maximum acyclic subgraph problem is NP-hard. To simplify the analysis of the forthcoming heuristics, we assume that the graph does not contain two-cycles. A two-cycle consists of two antipodal edges (u, v) and (v, u). Otherwise, we delete both edges of the two-cycle, apply an algorithm or heuristic for finding a maximum acyclic subgraph and insert two edges pointing in the same direction into the graph. The direction should be chosen in such a way that no cycles are generated by the insertion.

A less ad hoc method that runs in linear time is a greedy algorithm: If the graph is not connected, then it is run on each connected component. One at a time, the algorithm adds each vertex to one of two lists, to the right of the 'left' list or to the left of the 'right' list. The final sequence of the vertices is the concatenation of the two lists in the obvious way ('left' on the left,'right' on the right). First, all isolated and all sink vertices are added to the 'right' list. Then all source vertices are added to the 'left' list. In this way, edges incident to them will not go 'backwards'. Then, until all vertices have been dealth with, we keep choosing the vertex with the largest net out degree, and append it to the 'left' list - the idea being to find a good local tradeoff each time.

9.4.1 The Greedy cycle-removal algorithm

Digraphs with many 2-cycles do not have small feedback edge sets since one edge from every 2-cycle must be reversed. Use of the greedy algorithm on digraphs without 2-cycles produces a vertex sequence with at most $|E|/2 - |E|/6$ 'backward' edges. The proof of this works by partitioning the vertices of G based on their in and out degrees at the time the algorithm processed them. The algorithm actually removes vertices from G as they are processed, so these degrees change over time. Then, with a few insightful observations about when certain vertices are processed, and what types of vertices are left over, the desired result is obtained. It is based on the intuition that nodes of large out-degree should appear near the top of the drawing. The nodes are sorted with respect to their out-degree. The node with the most outgoing edges gets the first position. Then a feedback arc set can be determined by finding all the edges going back in the nodes' order. This algorithm can be implemented consuming $O(|V| + |E|)$ run time, where $|V|$ denotes the number of vertices and $|E|$ the number of edges (or arcs). In practice its run time is competitive with the trivial search algorithm but provides better results.

Observe that the maximum acyclic subgraph problem is equivalent to the unweighted linear ordering problem: Find an ordering of the vertices of G,i.e., find a mapping $o : V \mapsto \{1, 2, \ldots, |V|\}$ such that the number of edges $(u, v) \in E : o(u) > o(v)$ is minimized. Thus, the easiest heuristic for the maximum acyclic subgraph problem is to take an arbitrary ordering of the graph and delete the edges (u, v) with $o(u) > o(v)$. We might use a given ordering or, e.g., use an ordering computed by applying breadth first search or depth first search to the graph. These heuristics are fast but do not allow to give any quality guarantees. Next, we present a heuristic which guarantees an acyclic set of size at least $\frac{|E|}{2}$. The idea is to delete for every vertex either the incoming or outgoing edges. We define $\delta^+(v) = \{(v, u) | (v, u) \in E\}$, the set of the outgoing edges of v, $\delta^-(v) = \{(u, v) | (u, v) \in E\}$, the set of the ingoing edges into v, and $\delta(v) = \delta^+(v) \cup \delta^-(v)$, the set of edges incident to v, $v \in V$. $|\delta^+(v)|$ $(|\delta^-(v)|)$ is called the *outdegree (indegree)* of v.

Input: A digraph G.

Output: Acyclic Set E_a.

```
A GREEDY - CYCLE - REMOVAL ALGORITHM($G$)
1  $E_a \leftarrow \emptyset$
2  for each vertex $v$ in $V$
3      do if $|\delta_+(v)| \geq |\delta^-(v)|$
4            then append $\delta_+(v)$ to $E_a$
5            else append $\delta^-(v)$ to $E_a$
6         delete $\delta(v)$ from $G$
```

The algorithm computes an acyclic set E_a with size $|E_a| \geq \frac{|E||}{2}$ and runs in linear time (Berger and Shor, 1990).

9.4.2 An Enhanced Greedy Heuristic

A closer look at the problem shows that sources and sinks (which may arise during the algorithm) play a special role: edges incident to sources or sinks cannot be part of a cycle. This observation is used in the following algorithm (Eades et al., 1993):

Input: digraph G.

Output: Acyclic Set E_a.

```
AN ENHANCED GREEDY HEURISTIC($G$)
1  $E_a \leftarrow \emptyset$
2  while $G \neq \emptyset$
3      do while $G$ contains a sink $v$
4             do add $\delta^-(v)$ to $E_a$ and delete $v$ and $\delta^-(v)$ from $G$
5                delete all isolated vertices from $G$
6         while $G$ contains a source $v$
7             do add $\delta_+(v)$ to $E_a$ and delete $v$ and $\delta_+(v)$ from $G$
8                if $G \neq \emptyset$
9                   then let $v$ be a vertex in $G$ with maximum value $|\delta_+(v)| - |\delta^-(v)|$
10                       add $\delta_+(v)$ to $E_a$ and delete $v$ and $\delta(v)$ from $G$
```

The only difference between Algorithm 9.4.1 and Algorithm 9.4.2 is that the latter one processes the vertices in a special order. Hence, the output of Algorithm 9.4.2 is acyclic as well.

9.4.3 The Divide-and-Conquer Cycle Removal Algorithm

Similar to the greedy approach, this recursive algorithm also deals with the out-degree. Again, the nodes are sorted with respect to their out-degree or, in other words, labelled with integer values from 1 to $|V|$. The recursion is defined as follows:

1. If the given graph has no edges, then assign labels arbitrarily.

2. If the number of vertices is odd, then the node with the highest out-degree is called graph G_1. All remaining nodes and all edges connecting them are called graph G_2. Recursion on G_1 and G_2 provides the appropriate range of labels.

3. If the number of vertices is even, then partition the nodes into two sets of equal cardinality, G_1 and G_2. All out-degrees of the first set must be greater or equal to all out-degrees of the second set. Recursion on G_1 and G_2 provides the appropriate range of labels.

4. Finally, those edges pointing from a lower to a higher level belong to the resulting feedback arc set.

This algorithm can be implemented with time complexity of:

$$O(\min |V|^2,\ (|V| + |E|) \log |V|) \tag{9.44}$$

Thus for sparse graphs (where $|E|$ is $O(|V|)$) it is slower than the greedy algorithm. In practice it takes up to four times as long on graphs with up to 400 nodes.

Theorem 9.5 *Let $G = (V, E)$ be a connected digraph with no two-cycles. Then Algorithm 9.4.2 computes an acyclic edge set E_a with $|E_a| \geq \frac{|E|}{2} + \frac{|V|}{6}$.*

Proof. The vertex set V can be partitioned into five sets V_{sink}, V_{iso}, V_{source}, $V_=$ and $V <$. V_{sink} consists of the non-isolated sink vertices removed from G in Step 1, V_{iso} consists of the isolated vertices removed from G in Step 2, V_{source} consists of the non-isolated source vertices removed from G in Step 3, $V_=$ consists of the vertices whose indegree equals its outdegree, removed from G in Step 4 and $V_<$ consists of the vertices whose indegree is less than its outdegree, removed from G in Step 4. Note that these sets form a partition of V . Denote by m_i the number of edges removed from G as the result of the removal of the vertices in V_i, $i \in sink, iso, source, =, < =: I$, and by n_i the cardinality of V_i. Clearly, $|V| = \sum_{i \in I} n_i$, $|E| = \sum_{i \in I} m_i$, and $m_{iso} = 0$ holds.

Since the input graph is connected, isolated vertices can only be created in Step 1 and hence, $n_{iso} \leq m_{sink}$. It is not hard to see that after the removal of a vertex from $V_=$, at least one vertex whose indegree is not equal to its outdegree exists. Since the resulting graph contains no isolated vertices, the next deleted vertex will be in $V_{sink} \cup V_{source} \cup V_<$. Hence, we get $n_= \leq n_{sink} + n_{source} + n_<$. This can be used to find an estimation of n by substituting $n_= : n \leq 2n_{sink} + n_{iso} + 2n_{source} + 2n_<$. This can be relaxed to $n \leq 2n_{sink} + n_{iso} + 3n_{source} + 3n_<$. Using the facts $n_{iso} \leq m_{sink}$ and $n_{sink} \leq m_{sink}$, we get

$$n \leq 3(m_{sink} + n_{source} + n_<). \tag{9.45}$$

Observe that the only step where edges from E are thrown away and not inserted in E_a is Step 4. Thus, the number of thrown away edges is at most

$$|E| - |E_a| \leq \frac{m_=}{2} + \frac{m_< - n_<}{2} = \frac{m}{2} - \frac{m_{sink} + m_{source} + n_<}{2} \leq \frac{m}{2} - \frac{m_{sink} + n_{source} + n_<}{2}, \tag{9.46}$$

where the last inequality is true since $n_{source} \leq m_{source}$. By applying relation (9.45) we obtain

$$|E| - |E_a| \leq \frac{m}{2} - \frac{n}{6}. \tag{9.47}$$

The proof is now completed.

Bibliography

[1] Stephen C. North Emden R. Gansner, Eleftherios Koutsofios and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19, 214–230, 1993.

[2] Roberto Tamassia Giuseppe Di Battista, Peter Eades and Ioannis G. Tollis. *Graph Drawing*. Prentice-Hall, U.S.A., 1999.

[3] Patrick Healy and Nikola S. Nikolov. An optimal solution to dimension-constrained graph layering. 1999.

[4] M. Junger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1, no. 1, 1–25, 1997.

[5] Jennifer Listgarten. Csc 2410 project: Two (and a quarter) graph drawing paradigms. 1999.