A Technique for Drawing Directed Graphs

Emden R. Gansner
Eleftherios Koutsofios
Stephen C. North
Kiem-Phong Vo

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We describe a four-pass algorithm for drawing directed graphs. The first pass finds an optimal rank assignment using a network simplex algorithm. The second pass sets the vertex order within ranks by an iterative heuristic incorporating a novel weight function and local transpositions to reduce crossings. The third pass finds optimal coordinates for nodes by constructing and ranking an auxiliary graph. The fourth pass makes splines to draw edges. The algorithm makes good drawings and runs fast.

1. Introduction

Drawing abstract graphs is a topic of ongoing research having such applications as visualization of programs and data structures, and document preparation. This paper describes a technique for drawing directed graphs in the plane. The goal is to make high-quality drawings quickly enough for interactive use. These algorithms are the basis of a practical implementation [GNV1].

1.1 Aesthetic criteria

To make drawings, it helps to assume that a directed graph has an overall flow or direction, such as left to right or top to bottom. Such flows can be seen in hand-made drawings of finite automata (from initial to terminal states) and data flow graphs (from input to output). This observation has motivated a collection of methods for drawing digraphs based on the following aesthetic principles:

A1. Expose hierarchical structure in the graph. In particular, aim edges in the same general direction if possible. This aids finding directed paths and highlights source and sink nodes.

A2. Avoid visual anomalies that do not convey information about the underlying graph. For example, avoid edge crossings and sharp bends.

A3. Keep edges short. This makes it easier to find related nodes and contributes to A2.

A4. Favor symmetry and balance. This aesthetic has a secondary role in a few places in our algorithm.

There is no way to optimize all these aesthetics simultaneously. For instance, a placement of nodes and orientation of edges preferred according to A1 may force edge crossings that are undesirable according to A2. What is more, it is computationally intractable to minimize edge crossings or to find subgraphs having symmetry. We therefore make some simplifying assumptions and rely on heuristics that run quickly and make good layouts in common cases.

1.2 Description of problem

The input to the drawing algorithm is an attributed multigraph $G = (V,E)$ possibly containing loops and multi-edges. We assume that G is connected, as each connected component can be laid out separately. The attributes are:

$xsize(v), ysize(v)$	Size of bounding box of a node v.
$nodesep(G)$	Minimum horizontal separation between node boxes.
$ranksep(G)$	Minimum vertical separation between node boxes.
$\omega(e)$	Weight of an edge e, usually 1. The weight signifies the edge's importance, which translates to keeping the edge short and vertically aligned.

The algorithm assigns each node v to a rectangle in the plane with the center point $(x(v), y(v))$ and assigns each edge e to a sequence of B-spline control points $(x_0(e), y_0(e)),...,(x_n(e), y_n(e))$. Though the unit of these dimensions is not specified, it is convenient to use the traditional coordinate system of 72 units per inch in an implementation. The layout is generally guided by the aesthetic criteria A1-A4, and specifically by the graph attributes. The details of these constraints will be supplied in the following sections.

The input also contains sets $S_{\max}, S_{\min}, S_0, S_1, \ldots, S_k \subset V$. These constrain the algorithm in the placement of nodes. The initial pass of the algorithm described in the next section assigns nodes to discrete ranks $0...Max\_rank$. Nodes in the same rank receive the same Y coordinate value. $S_{\max}$, $S_{\min}$, and the sets S_i are (possibly empty) sets of nodes that must be placed together on the maximum, minimum, or same rank, respectively. The ability to give these sets as part of the input is useful for drawing graphs that have time-lines or for highlighting source and sink nodes.

1.3 Related work

Drawing digraphs using an iterative method to reduce edge crossing was first studied by Warfield [Wa], and similar methods were discovered by Carpano [Ca] and Sugiyama, Tagawa, and Toda [STT]. Di Battista and Tamassia describe an algorithm for embedding planar acyclic digraphs such that all edges flow in the same direction [DT]. We view our work as building on the approach of Warfield and Sugiyama et al.

1.4 Overview

The graph drawing algorithm has four passes, as shown in figure 1-1. The first pass places the nodes in discrete ranks. The second sets the order of nodes within ranks to avoid edge crossings. The third sets the actual layout coordinates of nodes. The final pass finds the spline control points for edges.

```
1.  draw_graph()
2.  {
3.      rank();
4.      ordering();
5.      position();
6.      make_splines();
7.  }
```

Figure 1-1. Main algorithm

Our contributions are: (1) an efficient way of ranking the nodes using a network simplex algorithm; (2) improved heuristics to reduce edge crossings; (3) a method for computing the node coordinates as a rank assignment problem; and (4) a method for setting spline control points. Techniques (1) and (2) were first implemented in the graph drawing program *dag*, described in [GNV1]. Further work, especially (3) and (4) have been incorporated in *dot*, a successor to *dag*. Figures 1-2 and 1-3 are samples of *dot*'s output.

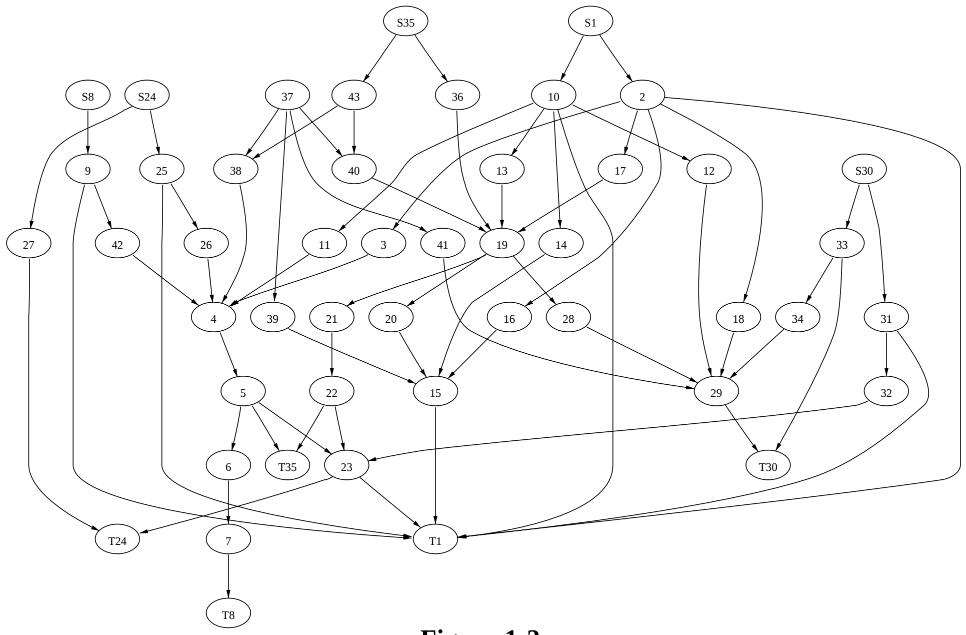

Figure 1-2.

(1.11 sec. user time on a Sun-4/280)

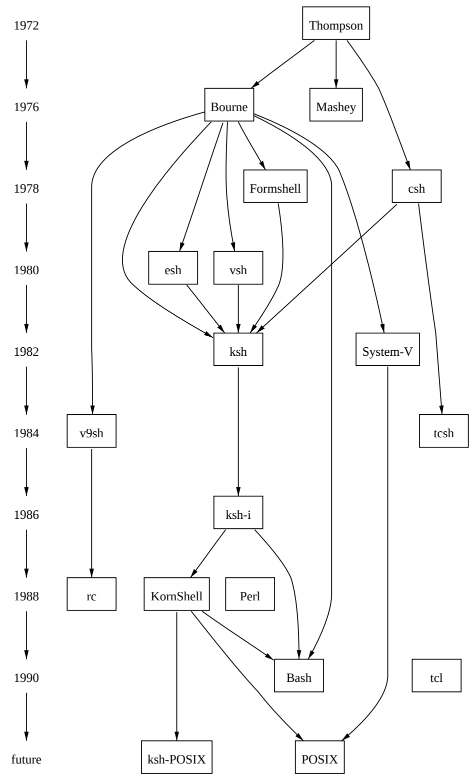

Figure 1-3.

(0.50 sec. user time on a Sun-4/280)

2. Optimal Rank Assignment

The first pass assigns each node $v \in G$ to an integer rank $\lambda(v)$ such that $l(e) \geq \delta(e)$ whenever $e = (v,w) \in E$. Here, node v is the tail of the edge, and w the head, the *length l(e)* of an edge $e = (v,w)$ is defined as $\lambda(w) - \lambda(v)$, and $\delta(e)$ represents the *minimum length*, usually 1. For this pass, each of the nonempty sets $S_{\max}, S_{\min}, S_0, \cdots, S_k$ is temporarily merged into one node. In addition, loops are ignored, and multiple edges are merged into one edge whose weight is the sum of the weights of the merged edges. For efficiency, leaf nodes that are not a member of one of the above sets may be ignored, since the rank of a leaf is trivially determined in an optimal ranking.

2.1 Making the graph acyclic

A graph must be acyclic to have a consistent rank assignment. Because the input graph may contain cycles, a preprocessing step detects cycles and breaks them by reversing certain edges. Of course these edges are only reversed internally; arrowheads in the drawing show the original direction. A useful heuristic for breaking cycles is based on depth-first search. Edges are searched in the ‘‘natural order’’ of the graph input, starting from some source or sink nodes if any exist. Depth-first search partitions edges into two sets: tree edges and non-tree edges. The tree defines a partial order on nodes. Given this partial order, the non-tree edges further partition into three sets: cross edges, forward edges, and back edges. Cross edges connect unrelated nodes in the partial order. Forward edges connect a node to some of its descendants. Back edges connect a descendant to some of its ancestors. It is clear that adding forward and cross edges to the partial order does not create cycles. Because reversing back edges makes them into forward edges, all cycles are broken by this heuristic.

It seems reasonable to try to reverse a smaller or even minimal set of edges. One difficulty is that finding a minimal set is NP-complete [EMW] [GJ]. More important, this would probably not improve the drawings. We implemented a heuristic to reverse edges that participate in many cycles. The heuristic takes one non-trivial strongly connected component at a time. Within each component, it counts the number of times each edge forms a cycle in a depth-first traversal. An edge with the maximal count is reversed. This is repeated until there are no more non-trivial strongly connected components.

Experiments with this heuristic show that most directed graphs arising from practical applications have a natural edge direction even when they contain cycles. Graph input usually reflects this natural direction. In fact, graphs are often created by a graph search performed by some other tool. Reversing an inappropriate edge disturbs the drawing. For instance, even when a procedure call graph has cycles, one still expects to see top-level functions near the top of the drawing, and not somewhere in the middle. From the standpoint of stability, the depth-first, cycle-breaking heuristic seems preferable. It also makes more informative drawings than would be obtained by collapsing all the nodes in a cycle into one node, or placing the nodes in a cycle on the same rank, or duplicating one of the nodes in the cycle, as various researchers have suggested.

One other detail is that $S_{\max}$ and $S_{\min}$ must always have the maximum and minimum rank assignments. This property is ensured by reversing out-edges of $S_{\max}$ and in-edges of $S_{\min}$. Also, for all nodes v with no in-edge, we make a temporary edge $(S_{\min}, v)$ with $\delta = 0$, and for all nodes v with no out-edge, we make temporary edge $(v, S_{\max})$ with $\delta = 0$. Thus, $\lambda(S_{\min}) \le \lambda(v) \le \lambda(S_{\max})$ for all v.

2.2 Problem Definition

Principle A3 prescribes making short edges. Besides making better layouts, short edges reduce the running time of later passes whose time depends on the total edge length. So it is desirable to find an optimal node ranking, *i.e.*, one for which the sum of all the weighted edge lengths is minimal.

Finding an optimal ranking can be reformulated as the following integer program:

$$\begin{aligned}&\mathit{min} \sum_{(v,w) \in E} \omega(v,w)(\lambda(w) - \lambda(v))\\ &\textit{subject to: } \lambda(w) - \lambda(v) \ge \delta(v,w) \ \forall \ (v,w) \in E\end{aligned}$$

The *weight* function ω and the *minimum length* function δ map the edge set E into the non-negative rational numbers and the non-negative integers, respectively.

There are various ways to solve this integer program in polynomial time. One method is to solve the equivalent linear program, then transform the solution to an integer one in polynomial time. Another involves converting the optimal rank assignment problem to an equivalent min-cost flow or circulation problem, for which there are polynomial-time algorithms (see [GT] and its references). As the constraint matrix is unimodular, the problem can also be solved, though not necessarily in polynomial time, by applying the simplex method. A more complete discussion of these and other techniques will be reported in [GNV2].

2.3 Network simplex

Here, we describe a simple approach to the problem based on a network simplex formulation. Although its time complexity is not provably polynomial, in practice it takes few iterations and runs quickly.

We begin with a few definitions and observations. A *feasible* ranking is one satisfying the length constraints $\lambda(e) \geq \delta(e)$ for all e. Given any ranking, not necessarily feasible, the *slack* of an edge is the difference of its length and its minimum length. Thus, a ranking is feasible if the slack of every edge is non-negative. An edge is *tight* if its slack is zero.

A spanning tree of a graph induces a ranking, or rather, a family of equivalent rankings. Note that the spanning tree is on the underlying unrooted undirected graph, and may not be a directed tree. This ranking is generated by picking an initial node and assigning it a rank. Then, for each node adjacent in the spanning tree to a ranked node, assign it the rank of the adjacent node, incremented or decremented by the minimum length of the connecting edge, depending on whether it is the head or tail of the connecting edge. This process is continued until all nodes are ranked. A spanning tree is *feasible* if it induces a feasible ranking. All edges in a feasible tree are tight.

Given a feasible spanning tree, we can associate an integer *cut value* with each tree edge as follows. If the tree edge is deleted, the tree breaks into two connected components, the tail component containing the tail node of the edge, and the head component containing the head node. The cut value is defined as the sum of the weights of all edges from the tail component to the head component, including the tree edge, minus the sum of the weights of all edges from the head component to the tail component.

Typically (but not always because of degeneracy) a negative cut value indicates that the weighted edge length sum could be reduced by lengthening the tree edge as much as possible, until one of the head component-to-tail component edges becomes tight. This corresponds to replacing the tree edge in the spanning tree with the newly tight edge, obtaining a new feasible spanning tree. It is also simple to see that an optimal ranking can be used to generate another optimal ranking induced by a feasible spanning tree. These observations are the key to solving the ranking problem in a graphical rather than algebraic context. Tree edges with negative cut values are replaced by appropriate non-tree edges, until all tree edges have non-negative cut values. The resulting spanning tree corresponds to an optimal ranking. For further discussion of the termination of the network simplex algorithm and optimality of the result, the interested reader is referred to [GNV2,Cu,Ch].

Figure 2-1 below describes our version of the network simplex algorithm.

```
 1.  rank ()
 2.  {
 3.      feasible_tree();
 4.      while (e = leave_edge())
 5.      {
 6.          f = enter_edge(e);
 7.          exchange(e,f);
 8.      }
 9.      normalize();
10.      balance();
11.  }
```

Figure 2-1. Network simplex

Remarks on Figure 2-1.

3: The function `feasible_tree` constructs an initial feasible spanning tree. This procedure is described more fully below.

4: `leave_edge` returns a tree edge with a negative cut value, or nil if there is none, implying the solution is optimal. Any edge with a negative cut value may be selected as the edge to remove.

6: `enter_edge` finds a non-tree edge to replace `e`. This is done by breaking the edge `e`, which divides the tree into a head and tail component. All edges going from the head component to the tail are considered, with an edge of minimum slack being chosen. This is necessary to maintain feasibility.

7: The edges are exchanged, updating the tree and its cut values.

9: The solution is normalized by setting the least rank to zero.

10: Nodes having equal in- and out-edge weights and multiple feasible ranks are moved to a feasible rank with the fewest nodes. The purpose is to reduce crowding and improve the aspect ratio of the drawing, following principle A4.

```
 1.  feasible_tree()
 2.  {
 3.      init_rank();
 4.      while (tight_tree() < |V|)
 5.      {
 6.          e = a non-tree edge incident on the tree
 7.              with a minimal amount of slack;
 8.         delta = slack(e);
 9.         if incident node is e.head {delta = –delta}
10.         for v in Tree
11.             v.rank = v.rank + delta;
12.     }
13.     init_cutvalues();
14. }
```

Figure 2-2. Finding an initial feasible tree

Remarks on Figure 2-2.

`3:` An initial feasible ranking is computed. For brevity, `init_rank` is not given here. Our version keeps nodes in a queue. Nodes are placed in the queue when they have no unscanned in-edges. As nodes are taken off the queue, they are assigned the least rank that satisfies their in-edges, and their out-edges are marked as scanned. In the simplest case, where $\delta = 1$ for all edges, this corresponds to viewing the graph as a poset and assigning the minimal elements to rank 0. These nodes are removed from the poset and the new set of minimal elements are assigned rank 1, etc.

`4:` The function `tight_tree` finds a maximal tree of tight edges containing some fixed node and returns the number of nodes in the tree. Note that such a maximal tree is just a spanning tree for the subgraph induced by all nodes reachable from the fixed node in the underlying undirected graph using only tight edges. In particular, all such trees have the same number of nodes.

`5–12:` This finds an edge to a non-tree node that is adjacent to the tree, and adjusts the ranks of the tree nodes to make this edge tight. As the edge was picked to have minimal slack, the resulting ranking is still feasible. Thus, on every iteration, the maximal tight tree gains at least one node, and the algorithm eventually terminates with a feasible spanning tree. This technique is essentially the one described by Sugiyama [STT].

`14:` The `init_cutvalues` function computes the cut values of the tree edges. For each tree edge, this is computed by marking the nodes as belonging to the head or tail component, and then performing the sum of the signed weights of all edges whose head and tail are in different components, the sign being negative for those edges going from the head to the tail component.

A small example of running the network simplex algorithm is shown in figure 2-3. Non-tree edges are dotted. In (a) the graph is shown after the initial ranking, with cut values as indicated. In (b) the edge (g,h) with a negative cut value has been replaced by the non-tree edge (a,e), with the new cut values shown. Because they are all non-negative, the solution is optimal and the algorithm terminates.

2.4 Implementation details

Versions of the network simplex algorithm are well understood and there are results in the literature to help tune an implementation [Ch]. We feel, however, it is worth pointing out several specific points to prospective implementors. These optimizations are useful here, but become crucial when we use the

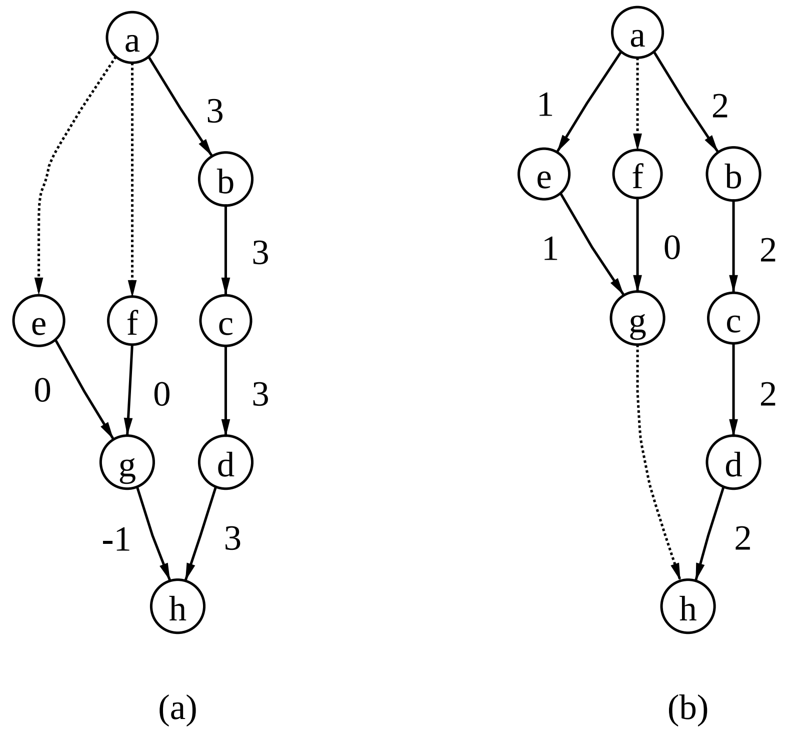

Figure 2-3. Finding an optimal feasible tree

network simplex again in section 4, applied to much larger graphs.

Computing the initial feasible tree and initial cut values is frequently a significant proportion of the cost in solving the network simplex algorithm. For many graphs in practice, the initial solution is close to optimal, requiring just a few iterations to reach the final solution. In a naive implementation, initial cut values can be found by taking every tree edge in turn, breaking it, labeling each node according to whether it belongs to the head or tail component, and performing the sum. This takes $O(VE)$ time.

To reduce this cost, we note that the cut values can be computed using information local to an edge if the search is ordered from the leaves of the feasible tree inward. It is trivial to compute the cut value of tree edge with one of its endpoints a leaf in the tree, since either the head or the tail component consists of a single node. Now, assuming the cut values are known for all the edges incident on a given node except one, the cut value of the remaining edge is the sum of the known cut values plus a term dependent only on the edges incident to the given node.

We illustrate this computation in figure 2-4 in the case where two tree edges, with known cut values, join a third, with the shown directionality. The other cases are handled similarly. We assume the cut values of (u,w) and (v,w) are known. The edges labeled with capital letters represent the set of all non-tree edges with the given direction and whose heads and tails belong to the components shown.

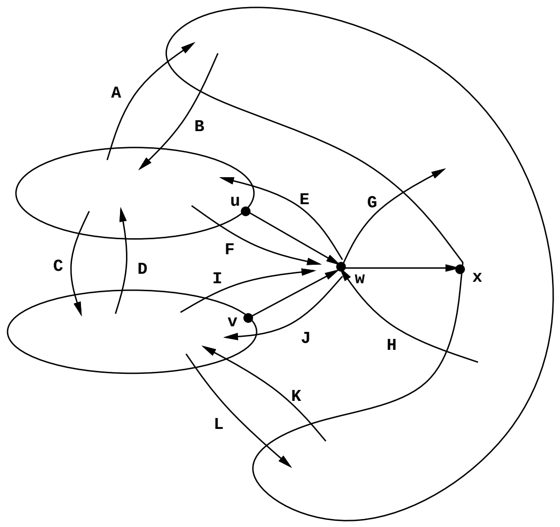

Figure 2-4. Incrementally computing cut values.

The cut values of (u,w) and (v,w) are given by

$$c_{(u,w)} = \omega(u,w)+A+C+F-B-E-D$$

and

$$c_{(v,w)} = \omega(v,w)+L+I+D-K-J-C$$

respectively. The cut value of (w,x) is then

$$\omega(w,x)+G+A+L-H-B-K = \omega(w,x)-\omega(u,w)-\omega(v,w)+c_{(u,w)}+c_{(v,w)}+G-H-F-I+E+J$$

an expression involving only local edge information and the known cut values. By thus computing cut values incrementally, we can ensure that every edge is examined only twice. This greatly reduces the time spent computing initial cut values.

Another valuable optimization, similar to a technique described in [Ch], is to perform a postorder traversal of the tree, starting from some fixed root node v_{root}, and labeling each node v with its postorder traversal number $lim(v)$, the least number $low(v)$ of any descendant in the search, and the edge $parent(v)$ by which the node was reached (see figure 2-5). This provides an inexpensive way to test whether a node lies in the head or tail component of a tree edge, and thus whether a non-tree edge crosses between the two components. For example, if $e=(u,v)$ is a tree edge and v_{root} is in the head component of the edge (*i.e.*, $lim(u) < lim(v)$), then a node w is in the tail component of e if and only if $low(u) \le lim(w) \le lim(u)$. These numbers can also be used to update the tree efficiently during the network simplex iterations. If $f=(w,x)$ is the entering edge, the only edges whose cut values must be adjusted are those in the path connecting w and x in the tree. This path is determined by following the

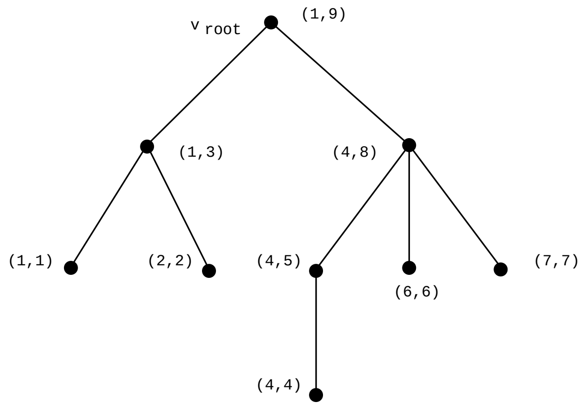

Figure 2-5. Postorder traversal with nodes labeled by (low,lim).

parent edges back from w and x until the least common ancestor is reached, *i.e.*, the first node l such that $low(l) \leq lim(w), lim(x) \leq lim(l)$. Of course, these postorder parameters must also be adjusted when exchanging tree edges, but only for nodes below l.

The network simplex is also very sensitive to the choice of the negative edge to replace. Searching cyclically through all the tree edges, instead of searching from the beginning of the list of tree edges every time, can save many iterations.

3. Vertex Ordering Within Ranks

After rank assignment, edges between nodes more than one rank apart are replaced by chains of unit length edges between temporary or ‘‘virtual’’ nodes. The virtual nodes are placed on the intermediate ranks, converting the original graph into one whose edges connect only nodes on adjacent ranks. Self-edges are ignored in this pass, and multi-edges are merged as in the previous pass.

The vertex order within ranks determines the edge crossings in the layout, so a good ordering is one with few crossings. Heuristics are appropriate since minimizing edge crossings in layouts of ranked graphs is NP-complete, even for only two ranks [EMW].

Several important heuristics for reducing edge crossings in ranked graphs are based on the following scheme first suggested by Warfield [Wa]. An initial ordering within each rank is computed. Then a sequence of iterations is performed to try to improve the orderings. Each iteration traverses from the first rank to the last one, or vice versa. When visiting a rank, each of its vertices is assigned a weight based on the relative positions of its incident vertices on the preceding rank. Then the vertices in the rank are re-ordered by sorting on these weights.

Two common vertex weighting methods are the barycenter function [STT] and the median function [EW]. Let *v* be a vertex and *P* the list of positions of its incident vertices on the appropriate adjacent rank. Note that the position of an adjacent node is only its ordinal number in the current ordering. The barycenter method defines the weight of *v* as the average of elements in *P*. The median method defines the weight of *v* as the median of elements in *P*. When the number of elements in *P* is even, there are two medians. This gives rise to two median methods: always using the left median, and always using the right median. The median method consistently performs better than the barycenter method and has a slight theoretical advantage since [EW] shows that the median layout of a two-level graph has no more than 3 times the minimum number of crossings. No such bound is known for the barycenter method.

Our node ordering heuristic is a refinement of the median method with two major innovations. First, when there are two median values, we use an interpolated value biased toward the side where vertices are more closely packed. The second improvement uses an additional heuristic to reduce obvious crossings after the vertices have been sorted. The additional heuristic transforms a given ordering to one that is locally optimal with respect to transposition of adjacent vertices. The improvement gained is typically 20-50%; we refer the reader to [GNV1] for detailed statistics.

Figure 3-1 shows the node ordering algorithm.

```
 1.  ordering()
 2.  {
 3.      order = init_order();
 4.      best = order;
 5.      for (i = 0 to Max_iterations)
 6.      {
 7.          wmedian(order,i);
 8.          transpose(order);
 9.          if (crossing(order) < crossing(best))
10.              best = order;
11.      }
12.      return best;
13.  }
```

Figure 3-1. Vertex ordering algorithm

Remarks on Figure 3-1.

3: `init_order` initially orders the nodes in each rank. This may be done by a depth-first or breadth-first search starting with vertices of minimum rank. Vertices are assigned positions in their ranks in left-to-right order as the search progresses. This strategy ensures that the initial ordering of a tree has no crossings. This is important because such crossings are obvious, easily-avoided ‘‘mistakes.’’

5–11: `Max_iterations` is the maximum number of iterations. At each iteration, if the number of crossings improves, the new ordering is saved. In an actual implementation, one might prefer an adaptive strategy that iterates as long as the solution has improved at least a few percent over the last several iterations. `wmedian` re-orders the nodes within each rank based on the weighted median heuristic. `transpose` repeatedly exchanges adjacent vertices on the same rank if this decreases the number of crossings. Both of these functions are described more completely below.

The weighted median heuristic is shown in figure 3-2. Depending on the parity of the current iteration number, the ranks are traversed from top to bottom or from bottom to top. To simplify the presentation, figure 3-2 only shows one direction in detail.

```
wmedian(order,iter)
{
    median[|V|];
    if (iter even) {
        for (r = 1 to Max_rank)
        {
            for (v in order[r])
                median[v] = median_value(v,r–1);
            sort(order[r],median);
        }
    }
    else {
        ...
    }
}

median_value(v, adj_rank)
{
    P[] = adj_position(v,adj_rank);
    m = |P|/2;
    if(|P| = 0)
        return –1.;
    else if(|P| odd)
        return P[m];
    else if(|P| = 2)
        return (P[0] + P[1])/2;
27.     else
28.     {
29.         left = P[m–1] – P[0];
30.         right = P[|P|–1] – P[m];
31.         return (P[m–1]*right + P[m]*left)/(left+right);
32.     }
33. }
```

Figure 3-2. The weighted median heuristic

Remarks on Figure 3-2.

`1–15:` In the forward traversal of the ranks, the main loop starts at rank `1` and ends at the maximum rank. At each rank a vertex is assigned a median based on the adjacent vertices on the previous rank. Then, the vertices in the rank are sorted by their medians. An important consideration is what to do with vertices that have no adjacent vertices on the previous rank. In our implementation such vertices are left fixed in their current positions with non-fixed vertices sorted into the remaining positions.

`17–33:` The median value of a vertex is defined as the median position of the adjacent vertices if that is uniquely defined. Otherwise, it is interpolated between the two median positions using a measure of tightness. Generally, the weighted median is biased toward the side where vertices are more closely packed.

`19:` The `adj_position` function returns an ordered array of the present positions of the nodes adjacent to `v` in the given adjacent rank.

`21–22:` Nodes with no adjacent vertices are given a median value of -1. This is used within the `sort` function to indicate that these nodes should be left in their current positions.

Figure 3-3 shows the transposition heuristic.

```
 1. transpose(rank)
 2. {
 3.     improved = True;
 4.     while (improved)
 5.     {
 6.         improved = False;
 7.         for (r = 0 to Max_rank)
 8.         {
 9.             for (i = 0 to |rank[r]|–2)
10.             {
```

```
11.                 v = rank[r][i];
12.                 w = rank[r][i+1];
13.                 if (crossing(v,w) > crossing(w,v))
14.                 {
15.                     improved = True;
16.                     exchange(rank[r][i],rank[r][i+1]);
17.                 }
18.             }
19.         }
20.     }
21. }
```

Figure 3-3. The transposition heuristic for reducing edge crossings

Remarks on Figure 3-3.

`4–20`: This is the main loop that iterates as long as the number of edge crossings can be reduced by transpositions. As in the loop in the `ordering` function, an adaptive strategy could be applied here to terminate the loop once the improvement is a sufficiently small fraction of the number of crossings.

`11–17`: Each adjacent pair of vertices is examined. Their order is switched if this reduces the number of crossings. The function `crossing(v,w)` simply counts the number of edge crossings if `v` appears to the left of `w` in their rank.

One small point is that the original graph may have edges between nodes on the same rank. We call these ‘‘flat edges.’’ Following criterion A1, we try to aim them all in the same direction across the rank. If ranks are ordered from top to bottom, flat edges generally point from left to right. This involves some minor modifications to the vertex ordering algorithms. If there are flat edges, their transitive closure is computed before finding the vertex order. The vertex order must always embed this partial order. In particular, the initial order must be consistent with it, and the `transpose` and the `sort` routines must not exchange nodes against the partial order.

When sorting nodes by medians and transposing adjacent nodes, equality can occur when comparing median values or number of edge crossings. We have found it helpful, and in keeping with the spirit of A4, to flip nodes with equal values during the sorting or transposing passes on every other forward and backward traversal.

One final point is that it is generally worth the extra cost to run the vertex ordering algorithm twice: once for an initial order determined by starting with vertices of minimal rank and searching out-edges, and the second time by starting with vertices of maximal rank and searching in-edges. This allows one to pick the better of two different solutions.

4. Node Coordinates

The third pass sets node coordinates. Previous work has treated this as a postprocessing step of the barycenter or median methods, making local adjustments to avoid bad layouts. Considering node placement as a separate, well-defined problem, however, yields better layouts and provides a foundation for further extensions, such as trying to set the vertex order by methods that are more topological than geometric.

X and *Y* coordinates are computed in two separate steps. The first step assigns *X* coordinates to all nodes (including virtual nodes), subject to the order within ranks already determined. The second step assigns *Y* coordinates, giving the same value to nodes in the same rank. The *Y* coordinate assignment maintains the minimum separation $ranksep(G)$ between node boxes. Optionally, the separation between adjacent ranks can be increased to improve the slope of nearly horizontal edges to make them more readable. Because the *Y* coordinate step is straightforward, the remainder of this section deals with *X* coordinates.

According to the aesthetic principles already mentioned, short, straight edges are preferable to long, crooked ones. This property of *X* coordinates is captured in the following integer optimization problem:

$$min \sum_{e=(v,w)} \Omega(e)\,\omega(e)\,|x_w - x_v|$$

$$subject\ to: x_b - x_a \geq \rho(a,b)$$

where a is the left neighbor of b on the same rank and $\rho(a,b) = \frac{xsize(a) + xsize(b)}{2} + nodesep(G)$

$\Omega(e)$, an internal value distinct from the input edge weight $\omega(e)$, is defined to favor straightening long edges. Since edges between real nodes in adjacent ranks can always be drawn as straight lines, it is more important to reduce the horizontal distance between virtual nodes, so chains may be aligned vertically and thus straightened. The failure to straighten long edges can result in a ‘‘spaghetti effect’’ of edges having many different slopes. Accordingly, edges are divided into three types depending on their end vertices: (1) both real nodes, (2) one real node and one virtual node, or (3) both virtual nodes. If e, f, and g are edges of types (1), (2), and (3), respectively, then $\Omega(e) \leq \Omega(f) \leq \Omega(g)$. Our implementation uses 1,2, and 8. ρ is a function on pairs of adjacent nodes in the same rank giving the minimum separation between their center points.

There are standard techniques for transforming this problem into a linear program by the addition of auxiliary variables and inequalities to remove the absolute values [Ch]. As the resulting constraints are unimodular, solving the linear program with the simplex method produces a solution to the integer program. This is easy to program, and the layouts it gives are aesthetically pleasing. Unfortunately, the transformation increases the size of the simplex matrix from VE to $2VE + E^2$ entries. Graphs of a few dozen nodes and edges can be drawn in a few seconds, but larger graphs take much longer, and even the amount of memory available becomes a limitation. So this is not a completely satisfactory way to make layouts, particularly on smaller computers.

4.1 Heuristic Approach

This approach replaces the linear program with a heuristic for finding *X* coordinates. The heuristic finds a ‘‘good’’ initial placement, then iteratively tries to improve it by sweeping up and down the ranks similar to the vertex ordering algorithm described in the previous section. The heuristic is sketched below.

```
 1.  xcoordinate()
 2.  {
 3.      xcoord = init_xcoord();
 4.      xbest = xcoord;
 5.      for (i = 0 to Max_iterations)
 6.      {
 7.          medianpos(i,xcoord);
 8.          minedge(i,xcoord);
 9.          minnode(i,xcoord);
10.          minpath(i,xcoord);
11.          packcut(i,xcoord);
12.          if (xlength(xcoord) < xlength(xbest))
13.              xbest = xcoord;
14.      }
15.      return xbest;
16. }
```

Figure 4-1. Assigning x-coordinates to vertices

Remarks on Figure 4-1.

3: An initial set of coordinates is computed as follows. For each rank, the left-most node is assigned coordinate 0. The coordinate of the next node is then assigned a value sufficient to satisfy the minimal separation from the previous one, and so on. Thus, on each rank, nodes are initially packed as far left as possible.

5–14: In each iteration, a collection of heuristics is applied to improve the coordinate assignment. If this results in an improvement over the previous best assignment, the coordinates are saved. The function `xlength` implements the objective function from the above optimization problem.

7: The median heuristic is based on the observation that the value $|x - x_0| + |x - x_1| + \cdots + |x - x_i|$ is minimized when x is the median of the x_i. The heuristic assigns each node both an upward and downward priority given by the weighted sum of its in- and out-edges, respectively. On downward iterations, nodes are processed in the downward priority order and placed at the median position of their downward neighbors subject to the placement of higher priority nodes and space requirements of nodes not yet placed. When there are two medians, taking their mean improves symmetry (A4). Upward placement is handled similarly.

8: `minedge` is similar to `medianpos` but considers only edges between two real nodes. It places the edge vertically as close as possible to the median of the nodes adjacent to either endpoint of the edge.

9: `minnode` performs local optimization one node at a time, using a queue. Initially all nodes are queued. When a node is removed from the queue, it is placed as close as possible to the median of all its neighbors (both up and down) subject to the separation function ρ. If the node's placement is changed, its neighbors are re-queued if not already in the queue. `minnode` terminates when it achieves a local minimum.

10: `minpath` straightens chains of virtual nodes by sequentially finding sub-chains that may be assigned the same *X* coordinate.

11: `packcut` sweeps the layout from left to right, searching for blocks that can be compacted. For each node, if all the nodes to the right of it can be shifted to the left by some increment without violating any positioning constraints, the shift is performed.

These heuristics make good layouts quickly, but they are complicated to program and the results are sometimes noticeably imperfect. Further fine tuning is difficult because the heuristics begin to interfere with each other.

4.2 Optimal Node Placement

We noticed that the *packcut* heuristic does not find subgraphs that could be compacted to improve the solution. We considered a heuristic to search for these subgraphs and shift them. We then observed that this is very similar to the way the network simplex algorithm moves entire subgraphs to find an optimal rank assignment (see section 2). This suggested that we apply the network simplex algorithm find optimal node coordinates, using the *X* coordinates as ‘‘ranks.’’

The method involves constructing an auxiliary graph as illustrated in figure 4-2. This transformation is the graphical analogue of the algebraic transformation mentioned above for removing the absolute values from the optimization problem. The nodes of the auxiliary graph G' are the nodes of the original graph G plus, for every edge e in G, there is a new node n_e. There are two kinds of edges in G'. One edge class encodes the cost of the original edges. Every edge $e = (u,v)$ in G is replaced by two edges (n_e, u) and $(n_{e,} v)$ with $\delta = 0$ and $\omega = \omega(e)\Omega(e)$. The other class of edges separates nodes in the same rank. If v is the left neighbor of w, then G' has an edge $f = e_{(v,w)}$ with $\delta(f) = \rho(v,w)$ and $\omega(f) = 0$. This edge forces the nodes to be sufficiently separated but does not affect the cost of the layout.

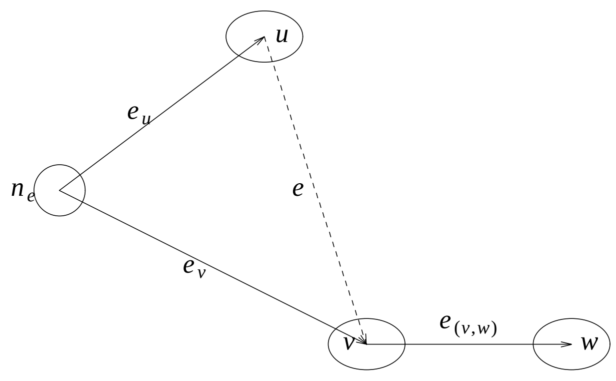

Figure 4-2.

We can now consider the level assignment problem on G', which can be solved using the network simplex method. Any solution of the positioning problem on G corresponds to a solution of the level assignment problem on G' with the same cost. This is achieved by assigning each n_e the value $min(x_u, x_v)$, using the notation of figure 4-2 and where x_u and x_v are the X coordinates assigned to u and v in G. Conversely, any level assignment in G' induces a valid positioning in G. In addition, in an optimal level assignment, one of e_u or e_v must have length 0, and the other has length $|x_u - x_v|$. This means the cost of an original edge (u,v) in G equals the sum of the cost of the two edges e_u, e_v in G' and, globally, the two solutions have the same cost, Thus, optimality of G' implies optimality for G and solving G' gives us a solution for G.

Using the auxiliary graph also permits the specification of ‘‘node ports,’’ or edge endpoints offset in the X direction from the center of the node. This makes it possible to draw pictures of flat records as shown in figure 4-3. When computing coordinates for nodes in these diagrams, the edge lengths must include the displacements of the node ports as well as the distance between the node center points. Let $e = (u,v)$ be an edge and let Δu and Δv be the X specified displacements of the endpoints from the centers of u and v, respectively. A $\Delta < 0$ indicates the port is to the left of the vertex’s center. Without loss of generality, assume $\Delta u \le \Delta v$ and let $d_e = \Delta v - \Delta u$. d_e is a constant since it depends only on the node ports and not the assignments of u and v. We want to solve the same optimization problem, but the cost of edge e is now given by $\Omega(e)\,\omega(e)\,|x_v - x_u + d_e|$. In the auxiliary graph, we now set $\delta(e_u) = d_e$ and $\delta(e_v) = 0$. We can then extend the argument above to show that any positioning for G corresponds to a level assignment for G'; that any optimal level assignment for G' induces a valid positioning for G; and, in both cases, we have

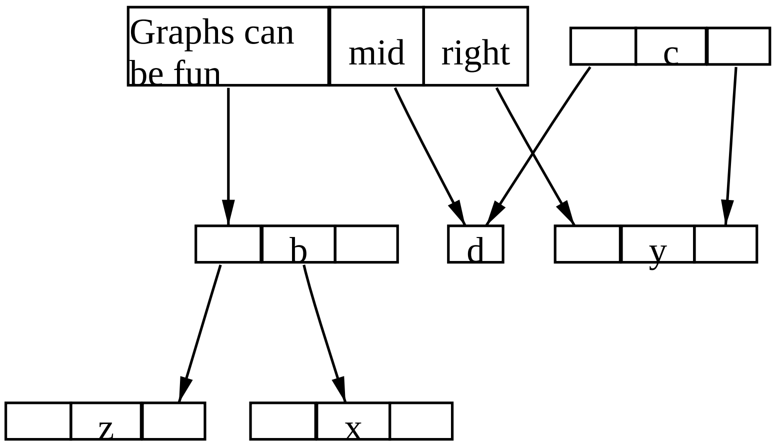

Figure 4-3. Node ports in a graph drawing

$$l(e_u) + l(e_v) = |x_v - x_u + d_e| + d_e$$

for all edges (u,v) in G, where l represents the length function in the level assignment on G'. This equation implies that the optimal costs of the problems on G and G' always differ by the constant $\sum_{e \in E} d_e$. Therefore, a minimal assignment for G' corresponds to a mimimal assignment for G.

4.3 Implementation details revisited

The auxiliary graph is considerably larger than the original one. If the original graph has V nodes, E edges, and R ranks, the graph with ‘‘virtual’’ nodes added has $V+D$ nodes and $E+D$ edges, where D is the number of ‘‘virtual nodes.’’ The auxiliary graph then has $V+E+2D$ nodes and $V+2E+3D-R$ edges. This graph requires disproportionately more time to use the network simplex approach. Consequently, the optimizations to the network simplex algorithm described at the end of section 2 are crucial for this pass.

Further improvement is possible by noting that it is easy to construct an initial feasible tree for the auxiliary graph by taking advantage of its structure. To construct a feasible tree, use all edges connecting nodes in the same rank. For each pair of adjacent ranks, pick an edge $f=(u,v)$ between the ranks and add both f_u and f_v in G' to the tree. This determines the relative placement of all the nodes in the two ranks. Finally, for every edge $e=(w,x) \neq f$ between the two ranks, add either e_w or e_x to the tree depending on whether w or x is placed leftmost.

Without these improvements, using network simplex to position the nodes took 5 to 10 times longer. With these improvements, our implementation runs as fast or faster than the heuristic implementation. We do not doubt that the heuristic in turn could also be tuned further, but the real advantage is that the network simplex is much simpler code and produces optimal solutions. Also, improvements that could be difficult to program into the heuristic can be handled in network simplex. As one example, local symmetry (A4) may be improved by scanning the graph after network simplex terminates. Tree edges whose cut value is exactly 0 identify subgraphs that may be adjusted to equalize the slack on their incident edges without changing the cost of the solution. This may be increase symmetry, such as centering a node with an even number of children.

5. Drawing Edges

In our method, edges are drawn as spline curves. Other graph drawing programs of which we are aware use line segments, and most make no attempt to avoid situations where line segments overlap unrelated nodes. Although splines are more difficult to program, they yield better drawings and help to satisfy aesthetic criterion A2.

In *dag*, edge splines are made by a collection of heuristics that replace the path of line segments between virtual nodes with various straight and curved segments, as described in [GNV88]. The drawback is that the splines sometimes bend sharply to turn inside virtual node boxes or to avoid nearby nodes. The virtual nodes end up being visible in the final layout. This method does not use the available space effectively.

It is better to try to find the smoothest curve between two points that avoids the ‘‘obstacles’’ of other nodes or splines. We can then divide the spline routing algorithm into a top half and a bottom half. The top half computes a polygonal region of the layout where the spline may be drawn. It calls the bottom half to compute the best spline within the region. As a final step, the top half clips the spline to the boundaries of the endpoint nodes. A region and its spline are illustrated in Figure 5-1.† The associated edge is from ‘‘Interdata’’ to ‘‘Unix/TS 3.0’’.

More formally, we draw splines by creating and solving instances of the following sub-problem. Given $B_0, \ldots, B_m, q, \theta_q, r, \theta_r$ where B_i are boxes parallel to the coordinate axes, such that B_i has edges in common with B_{i-1} and B_{i+1}; q and r are points on or inside the first and last box respectively, find $s_0, \ldots, s_n$ and $BB_0, \ldots, BB_m$, where s_i are the control points of a piecewise Bezier curve and BB_i are boxes parallel to the coordinate axes. The curve must have q and r as its endpoints. θ_q and θ_r are optional; if they are specified, then the curve must have the given slope at the corresponding endpoint. The BB_i correspond to the B_i and are the smallest boxes that contain the generated splines.

We next describe the two parts of the algorithm.

5.1 Finding the Region

There are three kinds of edges in the drawing: edges between nodes on different ranks, flat edges between different nodes on the same rank, and self-edges or loops.

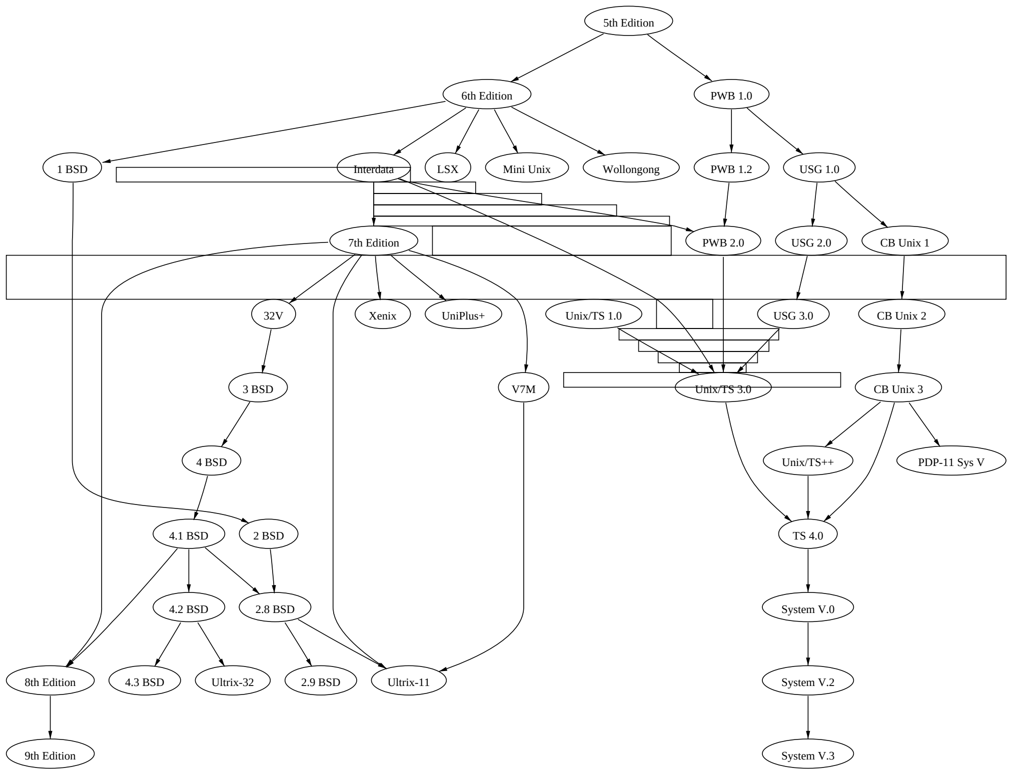

Figure 5-1. Region for a spline

(0.48 sec. user time Sun4-280)

5.1.1 Edges between ranks

In practice, most edges connect nodes on different ranks. The region for this kind of edge has a few boxes near its tail port, then an alternating sequence of inter-rank boxes and virtual node boxes, and finally a few boxes near the head port. The tail and head port boxes route the spline to the appropriate side of the node.

† Graph data courtesy of Ian F. Darwin, SoftQuad Inc.

To curve as smoothly as possible, a spline should be allowed all the space that is available. So the region should include not only virtual node boxes, but also any extra space next to them. After the spline has been computed, the virtual node boxes are updated according to the BB_i, so splines computed afterward will be able to use all the space remaining but not come too close to splines already drawn. Because splines are drawn by a ‘‘greedy’’ strategy, they depend on the order in which they are computed. It seems reasonable to route the shorter splines first because they can often be drawn as straight lines, but the order does not seem to affect the drawing quality much.

There are three details that can help to improve the appearance of the splines. First, when edges cross, they should not constrain each other too much. Otherwise, a spline may have an awkward, sharp turn. This is easily avoided by making an adjustment to the boxes. When setting the size of a box, we ignore virtual nodes to the left or right that correspond to edges that cross within two ranks. Crossings further away are not considered because unintended multiple crossings can occur when the boxes become too sloppy.

Second, when an edge has a section that is almost vertical, it looks better to just draw it as a vertical line. This is most obvious when edges run alongside each other, because parallel line segments look better than long segments with slightly different slopes. When the region finding procedure detects a long vertical section, it terminates the current region, draws its spline, draws the vertical line segment, and finally begins the region of the rest of the edge. This is one of the situations where θ_q and θ_r are used, since the splines must have a vertical tangent at the endpoint where they join the vertical line segment.

Third, when several splines approach a common termination point, it is important to avoid ‘‘accidental’’ intersections. To do this, we check if there are previous computed splines with the same endpoint. If so, we find the closest ones to the right and the left. We then subdivide the inter-rank space, and evaluate the left and right splines at the intervals. These points (or the boundaries of the layout, if one of the left or right splines does not exist) determine a set of boxes that separate the new spline from the existing ones as they approach the terminal node. The left and right splines and the boxes that result can be seen in Figure 5-1.

This subdivision of the inter-rank box could be viewed as approximating a polygonal region not necessarily aligned with the coordinate axes. In some layouts there are other places where non-aligned boxes or other polygons could prevent unintended tangencies. If we were writing this program again, we would try general polygons instead of boxes.

Thus far we have not mentioned multiple edges between the same pair of nodes. When these exist, a spline is computed for one of the edges, and the rest of the edges are drawn by adding an increasing X coordinate displacement to each one (multiples of $nodesep(G)$ work well). Space for multiple edges must be reserved in the previous pass, described in section 4, when setting the separation between nodes.

5.1.2 Flat Edges

Flat edges are handled much like inter-rank edges, but the region routes past intervening nodes and spaces between nodes. We omit most of the details since they are quite similar. For multiple flat edges, a spline is computed for the first one, and succeeding edges are drawn by adding Y coordinate displacements.

5.1.3 Self-edges

Self-edges are drawn as loops on the sides of nodes, as a special case. Space is allocated in the previous pass, described in section 4, when setting the separation between adjacent nodes. If there are multiple edges, their loops are nested.

The spline for a self-edge has two pieces, $p0,...,p3$ and $p3,...,p6$. $count(e)$ will refer to the index of a multiple edge. These control points are computed as follows:

$$\begin{aligned}
dx &= (count(e)+1)\,nodesep(G)\,; \\
dy &= ysize(v)+\frac{1}{2}count(e)\,nodesep(G) \\
p0 &= (x(v)+\frac{1}{2}xsize(v)\,,y(v)) \\
p1 &= p0+(\frac{1}{3}dx,\frac{1}{2}dy) \\
p2 &= p0+(dx,\frac{1}{2}dy) \\
p3 &= p0+(dx,0) \\
p4 &= p0+(dx,-\frac{1}{2}dy) \\
p5 &= p0+(\frac{1}{3}dx,-\frac{1}{2}dy) \\
p6 &= p0
\end{aligned}$$

5.2 Computing Splines

The computation of the splines has three stages. First, a piecewise linear curve or path lying entirely inside the region is computed. Then, the endpoints of this path are used as hints in the computation of a piecewise Bezier spline. Finally, the actual space used by the curve is computed in terms of the original boxes. The data structures computed by these three stages are shown in Figure 5-4. The region shown in this figure is the same one as in Figure 5-1. This example contains 13 boxes.

The three stages are outlined in Figure 5-2.

```
compute_splines (B_array, q, theta_q, use_theta_q, s, theta_s, use_theta_s)
{
 3.     compute_L_array (B_array);
 4.     compute_p_array (B_array, L_array, q, s);
 5.     if (use_theta_q) then vector_q = anglevector(theta_q)
 6.     else vector_q = zero_vector;
 7.     if (use_theta_s) vector_s = anglevector(theta_s)
 8.     else vector_s = zero_vector;
 9.     compute_s_array (B_array, L_array, p_array, vector_q, vector_s);
10.     compute_bboxes ();
11. }
```

Figure 5-2. Computing splines

Remarks on Figure 5-2.

`3:` `compute_L_array` computes the array $L_0, \ldots, L_{m+1}$ where L_i is the line segment that is the intersection of box B_{i-1} with box B_i. In Figure 5-4, these line segments as shown as thicker lines between boxes. There are 14 such segments.

`4:` `compute_p_array` computes an array of points $p_0, \ldots, p_k$ defining a feasible path that connects q and s. In Figure 5-4, there are 3 such points.

`5–8:` If `use_theta_q` or `use_theta_s` are true, the curve is constrained to approach the corresponding endpoint at the specified angles. `vector_q` and `vector_s` are normalized vectors.

`9:` `compute_s_array` computes an array of points $s_0, \ldots, s_k$ defining a piecewise Bezier spline that connects `q` and `s` and lies entirely inside the region. In the worst case, we can have one Bezier spline per box. In most cases, however, our approach generates significantly fewer splines. For example, in Figure 5-4, there are only 2 splines, one between p_0 and p_1 and one between p_1 and p_2. In more complex paths, there may even be fewer splines than line segments, since, unlike a line, a spline can curve around obstacles.

`10:` `compute_bboxes` computes the space actually taken up by the curve. It computes the array $BB_0, \ldots, BB_m$, where BB_i is the narrowest sub-box of B_i containing the curve.

`compute_p_array` and `compute_s_array` are both implemented as divide-and-conquer methods, as shown in Figure 5-3.

```
 1. compute_p_array (B_array, L_array, q, s)
 2. {
 3.     if (line_fits (B_array, L_array, q, s))
 4.         return;
 5.     p = compute_linesplit (B_array, L_array);
 6.     addto_p_array (p);
 7.     compute_p_array (B_array1, L_array1, q, p);
 8.     compute_p_array (B_array2, L_array2, p, s);
 9. }
10.
11. compute_s_array (B_array, L_array, p_array, vector_q, vector_s)
12. {
13.     spline = generate_spline (p_array, vector_q, vector_s);
14.     if (size (p_array) == 2) {
15.         while (spline_fits (spline, B_array, L_array) == False)
16.             straighten_spline (spline);
17.     } else if (spline_fits (spline, B_array, L_array) == False) {
18.         count = 0;
19.         ospline = spline;
20.         do {
21.             spline = refine_spline (p_array, ospline,
22.                                         mode (count, max_iterations));
23.             fits = spline_fits (spline, B_array, L_array);
24.             count = count + 1;
25.         } while ((fits == False) and (count <= max_iterations));
26.         if (fits == False) {
27.             p = compute_splinesplit (spline, p_array);
28.             compute_s_array (B_array1, L_array1, p_array1,
29.                                  vector_q, vector_p);
30.             compute_s_array (B_array2, L_array2, p_array2,
31.                                  reverse (vector_p), vector_s);
32.             return;
33.         }
34.     }
35.     addto_s_array (spline);
36. }
```

Figure 5-3. Spline drawing

Remarks on Figure 5-3.

`3–4:` `line_fits` checks if the line defined by `q` and `s` lies entirely inside the feasible region. The line is clipped to each box; if the line intersects a box, it must do so along the corresponding L segments.

`5:` If the (q, s) line does not fit, `compute_linesplit` finds the L segment that is the furthest from the (q, s) line and subdivides B_array and L_array along that segment. p is the one of the two endpoints of the subdivision segment that is closer to the (q, s) line. In Figure 5-4, for example, the path is subdivided along L_7

`6:` `addto_p_array` adds p to the array of endpoints for the path.

`7–8:` The two recursive calls to `compute_p_array` complete the computation of the path. `compute_p_array` is not guaranteed to be the shortest path, but it works very well so we have not developed it further. If it were important, the shortest path could be found in linear time using convex hulls, and Suri has an algorithm for solving the related problem of finding min-link paths [Su].

`13:` `generate_spline` computes a Bezier spline that approximates the path. This is done using a common technique [Gl].

`14–16:` The case where there is only one segment in the path is handled first. `spline_fits` checks if the spline lies entirely inside the region. The spline is sampled along its length and these samples are then clipped as a linear path against the box region. The process is similar to that of `line_fits`. As long as the spline does not fit, `straighten_spline` adjusts the control points of the spline to reduce the curvature. In the worst case, the spline becomes a line, and that is known to fit inside the path. This worst case can produce sharp turns. Most of the time, however, the spline fits inside the region after just a few iterations and this process does not produce any visual anomalies. It is only when the region itself makes sharp turns that the worst case may happen.

`17–34:` The second case is that the path has more than one segment. If the spline does not fit, `refine_spline` perturbs the control points of the spline in an attempt to make the spline fit. Our approach is similar to the straightening approach in lines 14-16. We try to decrease the curvature of the spline. If this does not seem to improve the fit, we try to increase the curvature. Since this process may never terminate, `max_iterations` controls how many times to try. `mode` returns a flag to indicate if the curvature is to be increased or decreased. If the spline still does not fit even after the refinement, we subdivide the problem. `compute_splinesplit` finds the endpoint of a segment on the path that is the furthest from the spline and subdivides the box and path arrays along that point. The two recursive calls to `compute_s_array` compute two piecewise Bezier splines, each fitting inside its corresponding part of the region. To force the two curves to join smoothly at the subdivision point, we also force the two splines to have the same unit tangent vector at that point. This guaranties C^1 continuity at the subdivision point. Forcing C^2 continuity does not seem to produce better results and is also much more expensive to compute. The straightening and refining heuristics, which save a lot of time, are based on the assumption that the tangent vectors at the endpoints of a spline can be scaled independently of the tangent vectors of the two adjacent splines. To maintain C^2 continuity, whenever a tangent vector is scaled, the tangent vector of the adjacent spline must also be scaled so that the two vectors will continue to have the same length. The scaling can propagate all the way to the end of the region. In addition, some of these splines may not fit even after scaling, and this would require more subdivisions, including subdivisions inside a single box. This is more trouble than it is worth.

35: Finally, `addto_s_array` adds the spline to the piecewise Bezier spline.

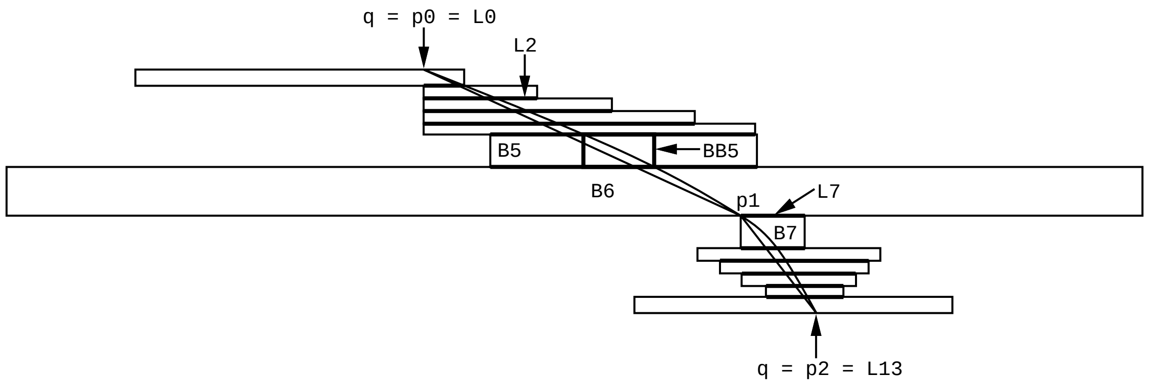

Figure 5-4. The three stages

5.3 Edge Labels

In *dag*, edge labels are placed next to the midpoint of the spline. This is an oversimplification since the placement does not avoid or even detect overlapping with other splines, labels, or nodes. Yet, graphs with edge labels are often small and sparse, so this technique is sometimes adequate.

In *dot*, edge labels on inter-rank edges are represented as off-center virtual nodes. This guarantees that labels never overlap other nodes, edges or labels. Certain adjustments are needed to make sure that adding labels does not affect the length of edges. Setting the minimum edge length to 2 (effectively doubling the ranks when virtual nodes are created) and halving the separation between ranks compensates for the label nodes. This makes it at least twice as expensive to draw a graph with labels, but the labels are readable. Figure 5-5 shows a drawing of a graph with edge labels.

Edge labels on self edges are easy to handle, but flat edges are more complicated. Here we must choose the left-to-right order for the virtual node of the label so that its *X* coordinate lies between the endpoint coordinates, not to the right or the left. At present we are still working on this problem.

More sophisticated placement of labels in diagrams (such as geographic maps) is a difficult research problem deserving further study. It is worth remarking that the label placing program as described by Freeman and Ahn [FA] is larger than our whole graph drawing program.

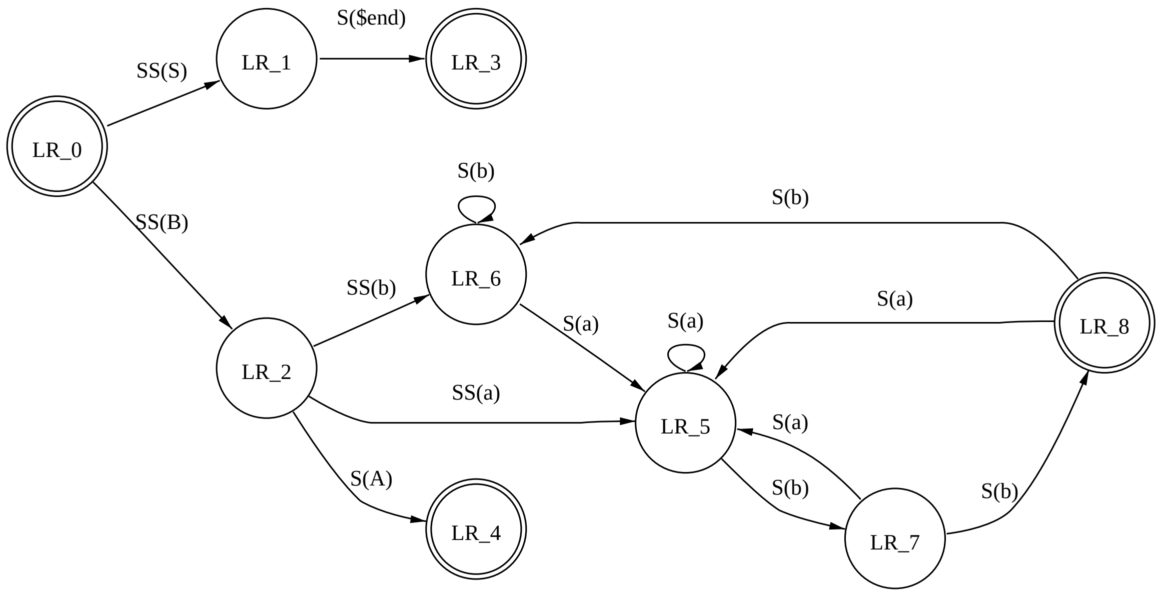

Figure 5-5. A finite state machine with labeled transitions

(0.15 sec. user time on a Sun-4/280)

6. Conclusions

We have described a method for drawing ranked digraphs. Our contributions are the application of network simplex for assigning ranks and final node coordinates, and an improved heuristic for reducing edge crossing. The method of finding node coordinates allows edges with *X* coordinate endpoint displacements. We also described a method for making edge splines. These techniques are straightforward to program, run fast enough for interactive use, and make drawings that compare well with previous work as to being readable and visually pleasing.

Further work might address the following:

• Understand how to modify the graph or its layout to enhance readability.

• Improve the edge crossing and spline drawing heuristics.

• Allow more interaction between the layout passes. Different solutions having the same cost in one phase may affect results a great deal in a following phase. For instance, two layouts can have the same number of crossings but much different final coordinates.

• Support incremental (on-line) graph drawing for animation. Stability from one drawing to the next is essential.

7. Acknowledgments

We wish to thank Guy Jacobson and Steve Lally for helpful comments on the content and style of this paper.

REFERENCES

[Ca] Carpano, M., ‘‘Automatic display of hierarchized graphs for computer aided decision analysis,’’ *IEEE Transactions on Software Engineering* SE-12(4), 1980, pp. 538-546.

[Ch] Chvatal, V., **Linear Programming**, W. H. Freeman, New York, 1983.

[Cu] Cunningham, W. H., ‘‘A network simplex method’’, *Mathematical Programming* 11, 1976, pp. 105-116.

[EMW] Eades, P., B. McKay and N. Wormald., ‘‘On an Edge Crossing Problem,’’ *Proc. 9th Australian Computer Science Conf.*, 1986, pp. 327-334.

[EW] Eades, P. and N. Wormald, ‘‘The Median Heuristic for Drawing 2-Layers Networks,’’ Technical Report 69, Dept. of Computer Science, Univ. of Queensland, 1986.

[FA] Freeman, Herbert and John Ahn, ‘‘On The Problem of Placing Names in a Geographic Map,’’ *International Journal of Pattern Recognition and Artificial Intelligence*, 1(1), 1987, pp. 121-140.

[GJ] Garey, Michael R. and David S. Johnson, **Computers and Intractability**, W. H. Freeman, San Francisco, 1979.

[Gl] Glassner, Andrew S., **Graphics Gems** (editor), Academic Press, San Diego, 1990.

[GNV1] Gansner, E. R., S. C. North and K.-P. Vo, ‘‘DAG - A Program that Draws Directed Graphs’’, *Software - Practice and Experience* 17(1), 1988, pp. 1047-1062.

[GNV2] Gansner, E. R., S. C. North and K.-P. Vo, ‘‘On the Rank Assignment Problem’’, to be submitted.

[GT] Goldberg, A. V. and R. E. Tarjan, ‘‘Finding minimum-cost circulations by successive approximation,’’ *Mathematics of Operations Research*, 15(3), 1990, pp. 430-466.

[Ka] Karmarkar, N., ‘‘A new polynomial-time algorithm for linear programming,’’ *Proc. 16th ACM STOC*, Washington, 1984, pp. 302-311.

[Kh] Khachiyan, L. G., ‘‘A polynomial algorithm in linear programming,’’ *Sov. Math. Doklady* 20, 1979, pp 191-194.

[STT] Sugiyama, K., S. Tagawa and M. Toda, ‘‘Methods for Visual Understanding of Hierarchical System Structures,’’ *IEEE Transactions on Systems, Man, and Cybernetics* SMC-11(2), February, 1981, pp. 109-125.

[Su] Suri, Subhash. ‘‘A linear time algorithm for minimum link paths inside a simply polygon,’’ *Computer Vision, Graphics, and Image Processing* 35, 1986, pp. 99-110.

[Wa] Warfield, John, ‘‘Crossing Theory and Hierarchy Mapping,’’ *IEEE Transactions on Systems, Man, and Cybernetics* SMC-7(7), July, 1977, pp. 505-523.