DAG – A Program to Draw Directed Graphs

DAG——A Program that Draws Directed Graphs

E. R. GANSNER, S. C. NORTH AND K. P. VO
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974, U.S.A.

SUMMARY

DAG is a program that draws directed graphs by reading a list of nodes and edges, computing a layout and then writing a PIC or PostScript description of a picture. Optional drawing instructions specify the way nodes are drawn, attach labels and control spacing. DAG works best on directed acyclic graphs, which are often used to represent hierarchical relationships. For example, here is a drawing of a graph from J. W. Forrester's book, *World Dynamics* (Wright-Allen, Cambridge, MA, 1971), which took 1·63 CPU seconds to draw using a VAX-8650.

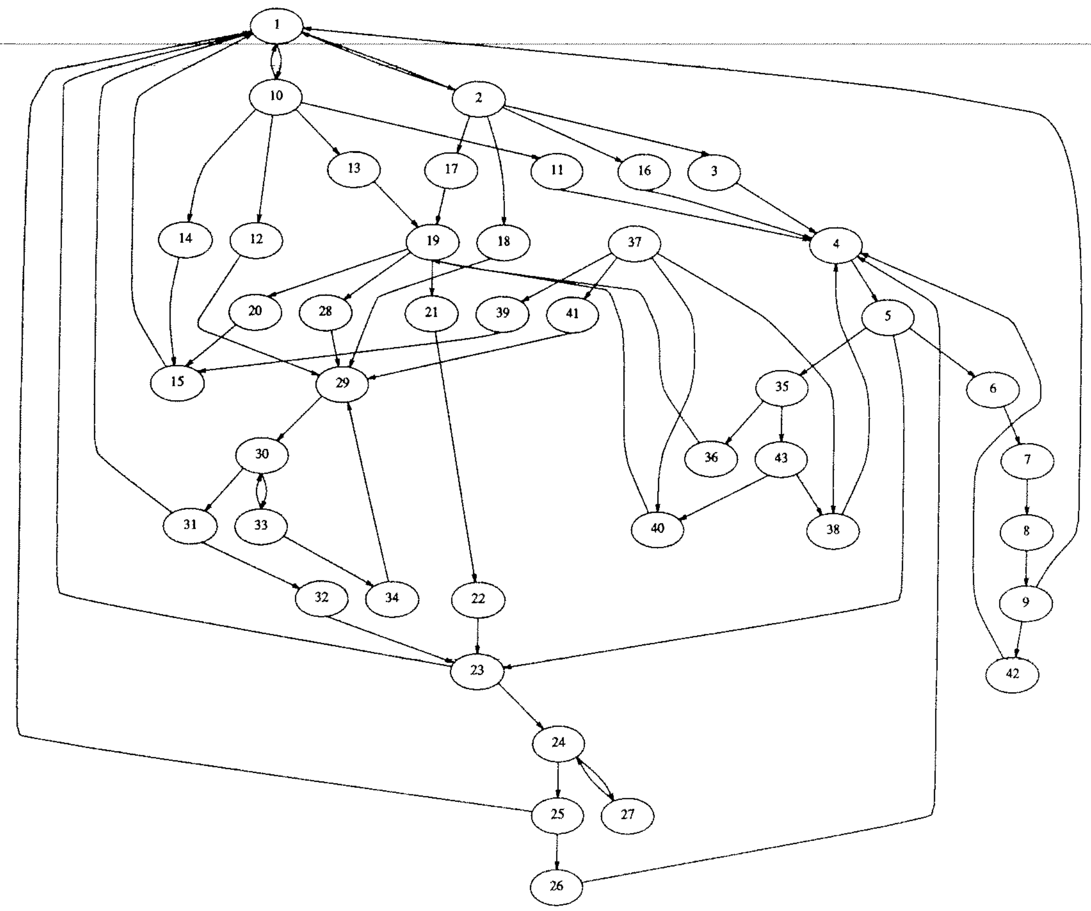

Figure 1

0038–0644/88/111047–16$08.00
© 1988 by John Wiley & Sons, Ltd.

Received 10 December 1988
Revised 11 May 1988

KEY WORDS Directed graph layout algorithms PostScript PIC

INTRODUCTION

Directed graphs are useful for describing relationships. They have many applications in computer programming, including the representation of data structures, finite automata, data flow, procedure calls and software configuration dependencies. They are frequently used in other disciplines as well.

A picture is a good way of representing many kinds of data, including directed graphs. Although it is seldom easy to understand a graph from a list of edges, with a picture, one can quickly find individual nodes and groups of related nodes, and trace paths in the graph. The main obstacle is that it can be difficult and tedious to make good drawings of graphs by hand.

DAG is a program that automatically draws directed graphs from edge lists. It is particularly effective for drawing acyclic graphs such as trees and partially ordered sets. DAG reads descriptions in a concise language, and makes picture in PIC[1] or PostScript.[2] As a PIC pre-processor, DAG fits in the TROFF pipeline:

```
dag file | pic | troff
```

BASICS

Figure 2 shows a DAG description and a reduced copy of the resulting picture. The graph description of Figure 2 is a sequence of edge statements. An edge statement

```
.GS
draw nodes as Box;
"5th Edition"   "6th Edition" "PWB 1.0";
"6th Edition"   "Interdata" "Wollongong"
                "Mini Unix" "1 BSD" "LSX";
"Interdata"     "7th Edition" "Unix/TS 3.0"
                "PWB 2.0";
"7th Edition"   "8th Edition" "32V" "V7M"
                "Ultrix-11" "Xenix" "UniPlus+";
"V7M"           "Ultrix-11";
"8th Edition"   "9th Edition";
"1 BSD"         "2 BSD";
"2 BSD"         "2.8 BSD";
"2.8 BSD"       "2.9 BSD" "Ultrix-11";
"32V"           "3 BSD";
"3 BSD"         "4 BSD";
"4 BSD"         "4.1 BSD";
"4.1 BSD"       "4.2 BSD" "2.8 BSD" "8th Edition";
"4.2 BSD"       "4.3 BSD" "Ultrix-32";
"PWB 1.0"       "PWB 1.2" "USG 1.0";
"PWB 1.2"       "PWB 2.0";
"USG 1.0"       "CB Unix 1" "USG 2.0";
"CB Unix 1"     "CB Unix 2";
"CB Unix 2"     "CB Unix 3";
"CB Unix 3"     "Unix/TS++" "PDP-11 Sys V";
"USG 2.0"       "USG 3.0";
"USG 3.0"       "Unix/TS 3.0";
"PWB 2.0"       "Unix/TS 3.0";
"Unix/TS 1.0"   "Unix/TS 3.0";
"Unix/TS 3.0"   "TS 4.0";
"Unix/TS++"     "TS 4.0";
"CB Unix 3"     "TS 4.0";
"TS 4.0"               "System V.0";
"System V.0"    "System V.2";
"System V.2"    "System V.3";
.GE
```

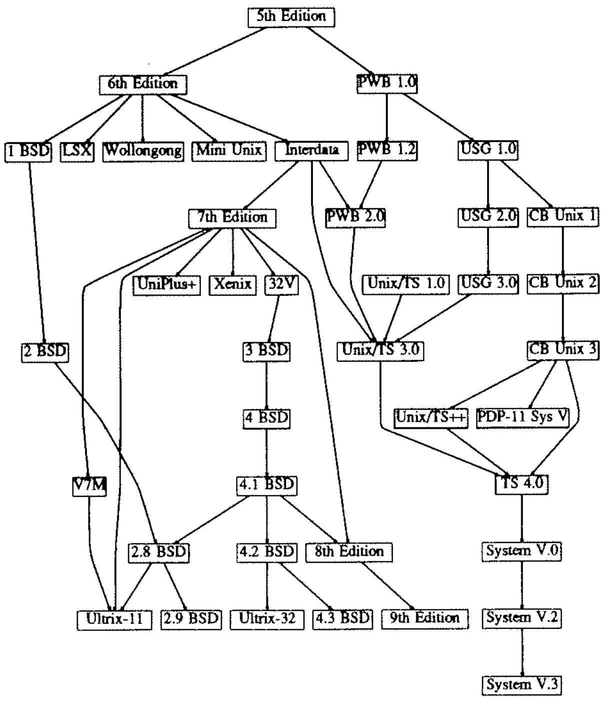

Figure 2

names a tail node and a list of head nodes. In the drawing, arrows point from tail nodes to head nodes. The statement

```
"5th Edition" "6th Edition" "PWB 1.0";
```

declares an edge from 5th Edition to 6th Edition, and another edge from 5th Edition to PWB 1.0. Because the syntax of edge statements is simple, graph descriptions can be generated from the output of other tools, such as *cflow*, *gprof* or *make* utilities, without much effort.

Each description begins with .GS and ends with .GE. When a file contains more than one graph, each is drawn separately. Optional arguments on the .GS line set the bounding box of the drawing, and the drawing is scaled if necessary.

Notice how DAG positions the nodes in the drawing of Figure 2. Nodes are placed in ranks, so that edges point downward. Within a rank, nodes are placed from left to right to reduce crossings and long edges. In some graphs it is not possible for all edges to point downward because there are cycles. To draw such cyclic graphs some edges are necessarily inverted. For instance, the graph in Figure 3 contains a cycle $a \rightarrow b \rightarrow c \rightarrow a$, so the edge $c \rightarrow a$ points upward.

```
.GS
edge from a to b;
edge from b to c;
edge from c to a;
.GE
```

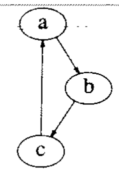

Figure 3

This example uses the alternative, verbose style for edge statements. It is safe to drop the quotes from node names here, because they do not contain white space, punctuation or DAG keywords. There is also a way to make edges point backward (see Figure 4).

```
.GS
backedge from a to b;
edge from b to c;
edge from c to a;
.GE
```

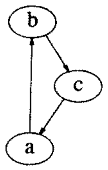

Figure 4

Sometimes it is more appropriate for the ranks to go from left to right, instead of from top to bottom. Replacing .GS with .GR (for 'graph rotated') requests a left to right drawing, as in Figure 5.

NODE DRAWING COMMANDS

The attributes of a node are its shape, size, label, the point size of the label and its colour. By default, DAG draws nodes as ellipses, labelled by the node name in 14-point type. The default size is $\frac{1}{2}$ in. by $\frac{3}{4}$ in., but the width may be increased

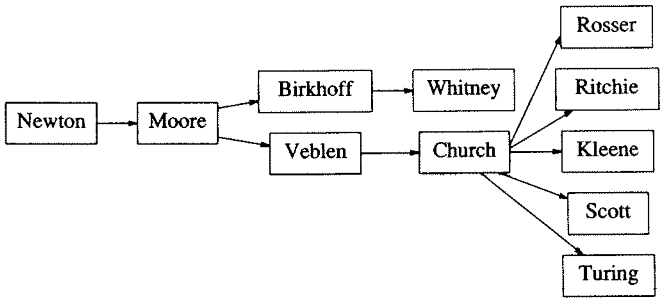

Figure 5

automatically to fit the text of the label. The draw statement changes node attributes. The keyword nodes indicates a change to the default attributes of all nodes introduced afterward. Here are some examples:

```
draw a,b,c as Box;
draw nodes as Doublecircle;
draw nodes width 1.5 height 1;
draw Newton, Moore pointsize 16;
draw "/usr/src/cmd" label "cmd";
draw tinyvertex as {circle rad .25};
draw "main" color red;
```

There are seven predefined shapes: Box, Square, Circle, Doublecircle, Ellipse, Diamond and Plaintext. These correspond to shape macro definitions in a library file. The user may define new shape macros or supply in-line graphics code in the target language, enclosed in braces, as shown in the tinyvertex statement. This example assumes that the drawing is in PIC; PostScript drawing commands are handled similarly. The macros receive the node height, width and label as arguments; in-line code may contain the variables $NAME, $HEIGHT and $WIDTH.

A colour value is a string whose interpretation depends on the target graphics language. Currently only PostScript drawings use colour. The colour of a node or edge is set in the PostScript drawing by emitting the colour value as executable code, followed by a call to the procedure dagsetcolor. By default, colours are assumed to be hue–saturation–brightness (HSB) triples, so DAG's standard PostScript preamble defines dagsetcolor as a call to sethsbvalue. The user may give another interpretation to colour by redefining this procedure and possibly changing the colour values to agree with the dagsetcolor interface.

EDGE DRAWING COMMANDS

The attributes of an edge are its label, the point size of the label, weight, ink style and colour. Edge attributes may be set when the edge is created. The draw edges statement changes the default attributes for all edges created afterward. Here are some examples:

```
edge from "module.a" to "module.b" label "x" pointsize 8;
edge from "main" to "innerloop" weight 100;
```

```
draw edges dotted;
draw edges color blue;
```

Often it is useful to attach labels to edges. As with nodes, an edge label can be given either as a piece of text or as drawing code. The label is drawn at the midpoint of the edge. Figure 6 is the description of a finite automaton using labelled edges and its picture.

```
.GR 6
draw nodes as Circle width .5 height .5;
draw LR_0 LR_3 LR_4 LR_8 as Doublecircle width .5 height .5;
draw edges pointsize 8;
LR_0 LR_2 label "SS(B)";
LR_0 LR_1 label "SS(S)";
LR_1 LR_3 label "S($end)";
LR_2 LR_6 label "SS(b)";
LR_2 LR_5 label "SS(a)";
LR_2 LR_4 label "S(A)";
LR_5 LR_7 label "S(b)";
LR_5 LR_5 label "S(a)";
LR_6 LR_6 label "S(b)";
LR_6 LR_5 label "S(a)";
LR_7 LR_8 label "S(b)";
LR_7 LR_5 label "S(a)";
LR_8 LR_6 label "S(b)";
LR_8 LR_5 label "S(a)";
.GE
```

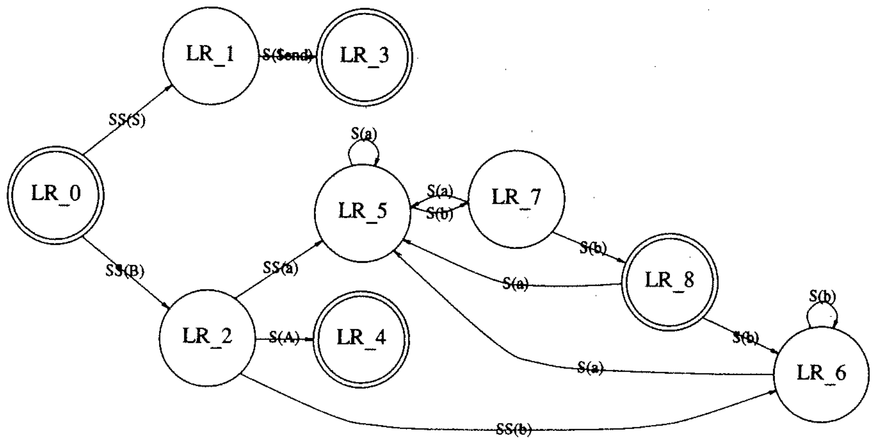

Figure 6

An edge statement can specify integer weights on edges. The default edge weight is 1. Increasing the weight of an edge makes DAG try to shorten it. For example, in the Unix History graph of Figure 2, one might decide that the edge between 2 BSD and 2.8 BSD should be favoured over the edge between 4.1 BSD and 2.8 BSD. This suggests

```
edge from "2 BSD" to "2.8 BSD" weight 64;
```

The user may also name the 'ink' for drawing edges: solid, dotted, dashed or invis:

```
edge from "2 BSD" to "2.8 BSD" dashed;
```

SUPPLEMENTARY COMMANDS

DAG has other commands for setting global attributes of the picture and manipulating sets of nodes. The global attributes affect the spacing in the picture. The separate command sets the minimum separation between nodes in the same rank, or the separation between ranks, in inches. Here are a few examples of its use:

```
separate nodes .75;
separate ranks 2;
separate ranks 2 exactly;
separate ranks 2 equally;
```

The value given to separate ranks specifies only the minimum separation. Often, when there are edges between nodes that are far apart horizontally, the connecting edges may intersect intermediate nodes. In such cases, DAG may increase the separation between certain adjacent levels to reduce such intersections, with the result that the ranks are not equally spaced. The last two separate commands above show how to control the rank separation. The adverb exactly forces ranks to be separated by exactly 2 in. On the other hand, equally tells DAG to do its best to reduce node–edge intersection while maintaining evenly separated ranks, which may be more aesthetically pleasing and readable. The optional value tells DAG to make the ranks at least that many inches apart.

There are also commands to control the rank assignments of individual nodes. A graph description can specify that certain nodes should be placed on the minimum or maximum rank, or that a group of nodes should be kept together on the same rank.

```
minimum rank root1 root2 root3;
maximum rank leaf38 leaf39;
same rank add sub mul div shift;
```

Figure 7 is another drawing of Forrester's *World Dynamics* graph. In this drawing, nodes have been constrained to appear at the same ranks as they appear in Figure 8 of Reference 3.

Forcing a group of nodes to be on the same rank can cause *flat* edges, i.e. edges that point sideways instead of downward. If at all possible, DAG will attempt to draw flat edges from left to right. Therefore, left-to-right order in a rank can be controlled by creating invisible flat edges. Figure 8 shows a graph description in which the relative placements of nodes are completely specified. Note that the edge crossing is unavoidable.

There are many other ways one might want to control a drawing, but it is difficult to build a satisfactory tool that achieves many, possibly conflicting, aesthetic goals. Users can exercise some control over node placement by setting edge weights and creating invisible edges. For finer control over drawings the user can change the drawing with a graphical editor or the generated code itself. We also refer the interested reader to Reference 4 for a different approach.

There are no absolute restrictions on the graphs that DAG can draw. Self-edges and multi-edges are allowed. For picture clarity, it is best to avoid nodes with many incident edges. For instance, in procedure call graphs, it is a good idea to eliminate common

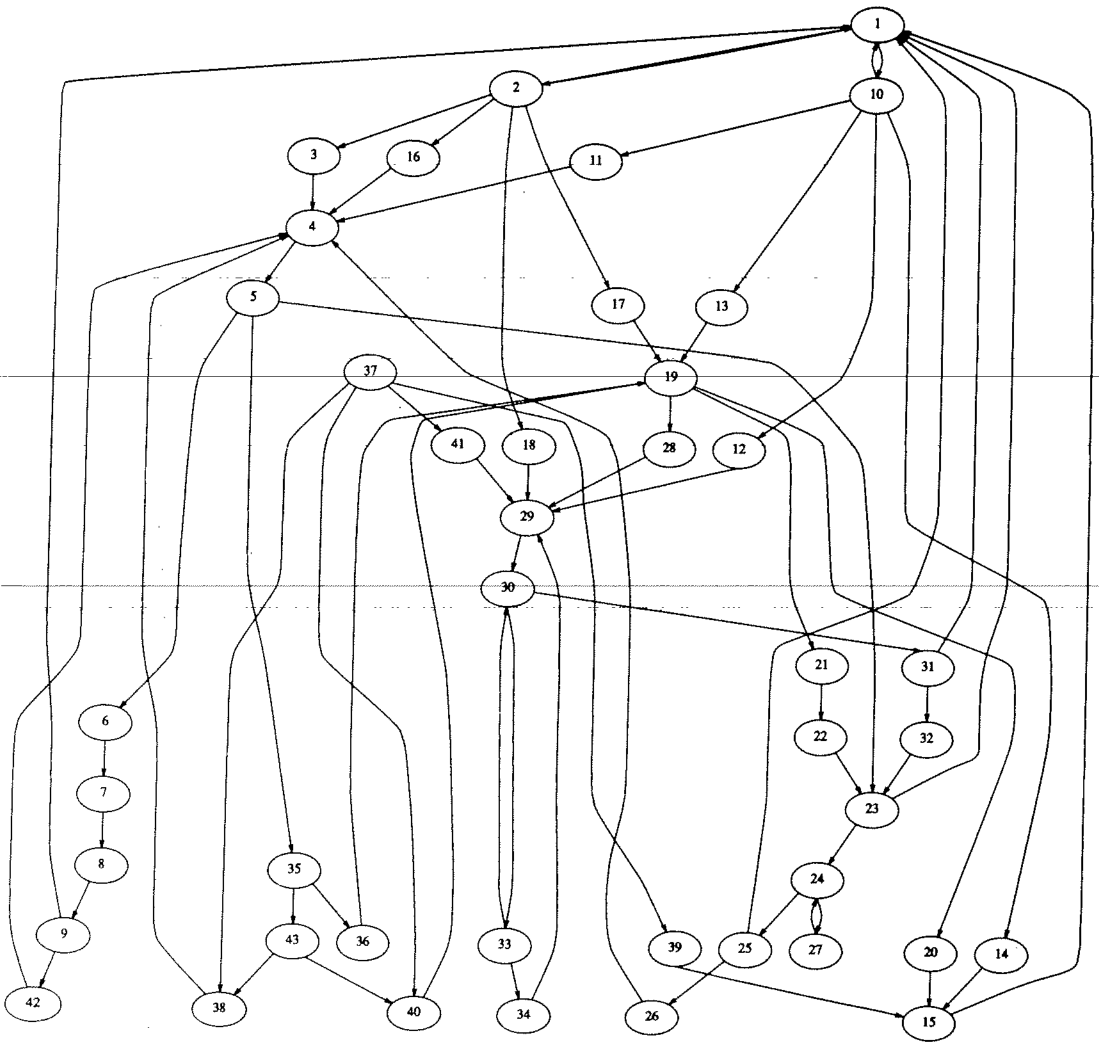

Figure 7

```
.GS
same rank 1 2;
same rank 3 4;
1 4;
2 3;
1 2 invis;
3 4 invis;
.GE
```

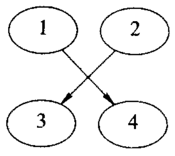

Figure 8

library calls before making the drawing. Removing these nodes both yields a more informative drawing and reduces run-time. Finally, if run-time is not critical, DAG has an 'optimize' option to trade run-time for drawing quality in the horizontal placement of nodes. Figure 1 was drawin in 1·63 s on a VAX-8650. Figure 9 shows the same graph drawn in optimal mode in 17·40 s.

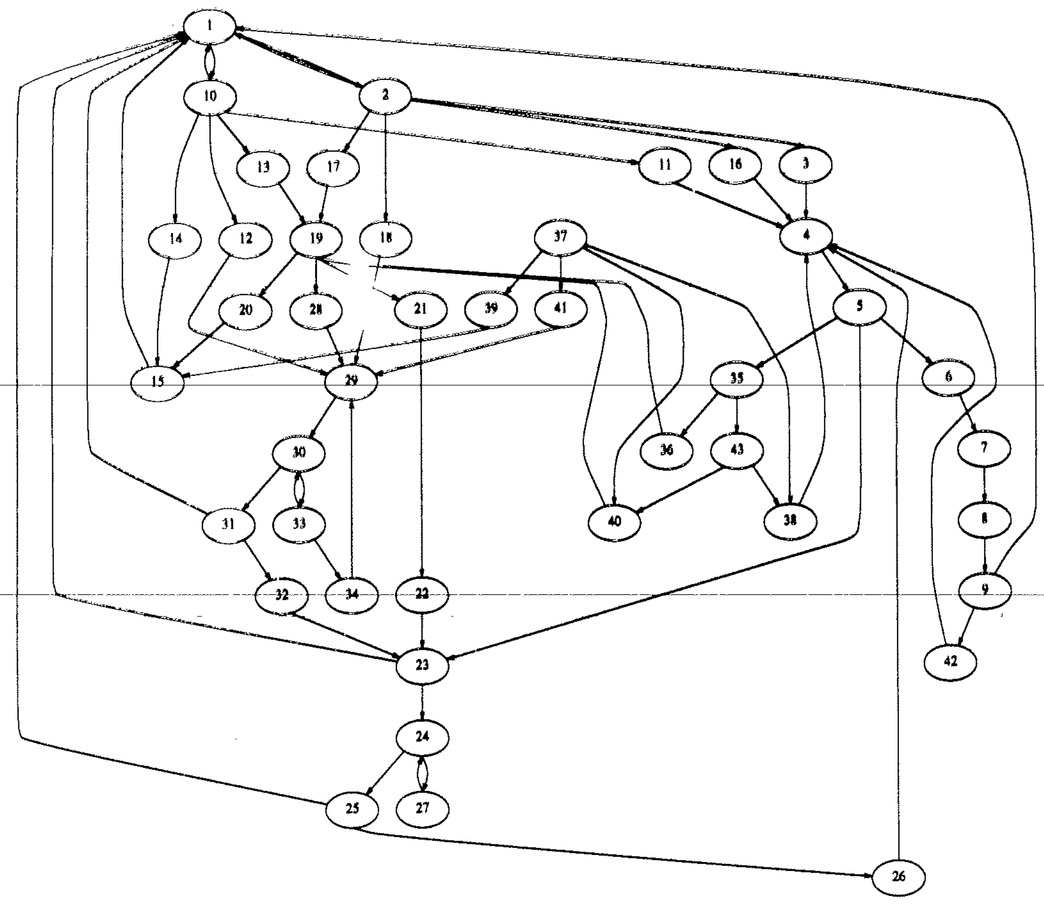

Figure 9

DRAWING ALGORITHMS

DAG has three main components: the parser, draw_dag and the code generator. The parser is written in YACC. It builds an input graph of attributed nodes and edges. The graph is passed to the drawing procedure draw_dag, which creates the layout by setting X and Y co-ordinates of nodes and finding spline control points for drawing edges. The code generator traverses the attributed graph to emit target code.

To design algorithms for drawing graphs, one first needs to define what makes a 'good' drawing. We have chosen three desirable properties for drawings of directed graphs:

P1. The drawing should emphasize the partial order implied by the edges of the graph, placing nodes in ranks so that a tail node is above its corresponding head node.

P2. The drawing should minimize artefacts such as edge/edge crossings and sharp corners on edges.

P3. The drawing should keep edges short.

These are essentially the properties used by Sugiyama *et al.*[5] They correspond to the desire to reveal relationships (P1 and P3), while avoiding misleading information (P2). Note that property P3 implies that related nodes should be placed close together and edges should be reasonably straight.

Unfortunately, these goals conflict. For instance, the placement of nodes in ranks may make it impossible to avoid edge crossings. Also, the minimization of edge crossings in a layout is computationally intractable[8] for relevant definitions of this problem. So DAG employs various heuristics that run quickly and make good layouts in common cases. These techniques are described in depth in Reference 7; here we sketch them briefly as an aid to understanding the layouts made by DAG.

The draw_dag component has four passes. Led by rules P1 and P3, the first pass breaks any cycles by reversing some edges in a depth-first search, and finds an optimal rank assignment for the nodes. The optimal rank assignment problem is to compute integer ranks for the nodes so that the sum of the costs of the edges is minimized. The cost of an edge is the product of its weight and its length, where the length is the difference in the rank assignments of its head and tail. Letting ρ be a rank assignment and ω the edge weight function, the optimal rank assignment problem can be formulated as the following integer programme:

$$\min \sum_{v \to w} (\rho(w) - \rho(v))\omega(v \to w)$$

$$\text{subject to: } \rho(w) \geq \rho(v) + 1 \text{ for all } v \to w$$

Because of the special nature of the problem, the simplex method for linear programming can be used to solve the integer programme. This requires $O(VE)$ space and runs in $O(IVE)$ time. Here, V is the number of nodes and E the number of edges in the graph. The value I represents some number of simplex iterations, possibly exponential in the size of the graph. However, as is usual with the simplex method, I is usually $O(E)$ in practice. For DAG, we have developed a combinatorial variation of the simplex algorithm that finds the solution in a factor of $O(E)$ less time and uses only $O(E)$ space.[6]

Following rank assignment, draw_dag creates dummy nodes where long edges cross ranks. Figure 10(b) illustrates where dummy nodes would be created for the graph in Figure 10(a).

Dummy nodes are useful as guiding points for generating splines of long edges. More importantly, they allow the edge crossing problem to be dealt with uniformly as the placement of nodes in two adjacent ranks. Since the number of edge crossings depends on the placement of both the real and dummy nodes, the optional rank assignment is important not only for its contribution to drawing quality, but also because it reduces the number of dummy nodes. This decreases the cost of subsequent passes, which have a factor of $O(V)$ in their running time. It also restricts the possible number of edge crossings. This is clearly illustrated in drawing Figures 1 and 7. Figure 1 is optimally ranked, has 20 edge crossings and takes 1·63 s to generate. By contrast, Figure 7, using the same ranking as that produced by techniques described in Reference 3, has 44 edge crossings and takes 3·68 s to generate.

The second pass orders nodes from left to right within ranks. The order respects any flat edges (P1), which might arise from same rank statements. This pass reduces

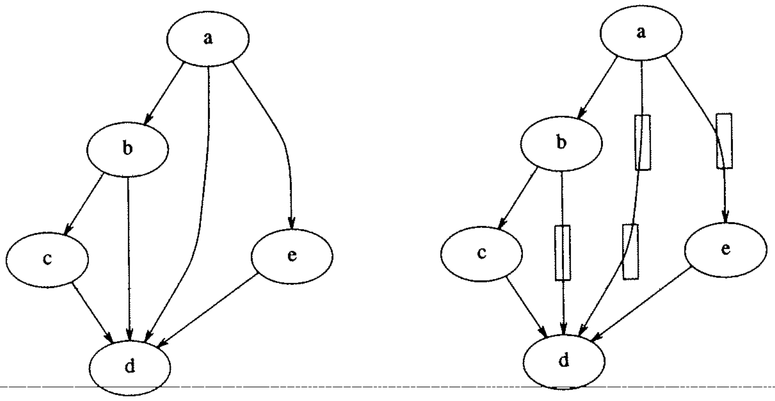

Figure 10

the number of edge crossings (P2). It is based on an iterative technique like that of References 3 and 5. The key improvements are a new weight function and local optimizations (cf. Tables I–III). We call the weight function the 'weighted median'; it produces fewer crossings by reducing the effects of widely spread nodes, and is inexpensive to compute. Informally, the weighted median of a node with respect to an adjacent rank is defined as follows. When the number of neighbours of a node in the adjacent rank is odd, the weighted median is simply the median position of its neighbours. When the number of neighbours is even, the weighted median is in between the left and the right median positions. The exact value is biased toward the side where neighbour nodes are more tightly packed.

The second pass begins by constructing an initial order of nodes within ranks in a depth-first traversal of the graph starting from nodes in the minimum rank. This ordering ensures zero edge crossings for series–parallel graphs and graphs whose underlying undirected graphs are trees. This means that node ordering for these graphs can be done optimally in linear time. After the initial order is computed, DAG iterates over the graph a number of times to reduce edge crossings. Each iteration consists of two steps. In the first step, each rank, from highest to lowest or from lowest to highest, is ordered using the weighted median values of nodes. After all ranks have been reordered, DAG considers pairs of adjacent nodes and transposes them if doing so reduces the number of crossings. The transposition heuristic can be viewed as a descent method to find local optima. It enhances the final ordering and helps the weighted median heuristic to converge quickly during intermediate steps. This iterative process is carried out a fixed number of times, or until no discernible progress is being made towards reducing the number of edge crossings.

The third pass assigns absolute co-ordinates to nodes, respecting the ordering determined by the previous pass and minimizing horizontal displacements between endpoints of edges (P3). Let χ be an optimal co-ordinate assignment. Solving for χ is equivalent to optimizing an objective function involving absolute values with linear constraints:

$$\min \sum_{v \to w} |\chi(v) - \chi(w)| \Omega(v \to w)$$

subject to: $\chi(x)+\delta < \chi(y)$ if $\rho(x) = \rho(y)$ and x is left of y

The edge weight function Ω is chosen to be heavier on long edges and edges joining nodes of degree two. This straightens long edges and simple paths and improves the picture clarity. The value δ represents the desired separation between consecutive nodes on the same rank. There is a standard technique for transforming this problem into a linear programme by introducing extra variables.

Since solving this optimization problem can be a bottleneck in draw_dag, DAG uses, by default, a fast iterative heuristic based on principles similar to those used in the second pass. In each iteration, the heuristic traverses ranks either from top to bottom or from bottom to top. Within each rank, nodes are assigned priorities. Nodes with higher priorities are placed before nodes with lower priorities. The priority of a node is based on the Ω values and the weights of its incident edges. When a node is placed, it is assigned a position as close as possible to the position of the median of its adjacent nodes. Since node placement respects the node ordering found in the second pass, the median of such a set of nodes is determined solely by their relative order. Therefore, the medians can be precomputed to enhance efficiency.

As with the median heuristic of the second pass, the node placement median heuristic performs better if the starting configuration is good. In each iteration, DAG employs heuristics that locally improve node placements. The first heuristic of this kind simply places a node in the position that minimizes the distance from all of its neighbours. The second heuristic straightens simple paths. Finally, since the median heuristic tends to separate portions of the given graph horizontally, a last heuristic is employed to pack such separations.

The node placement algorithm performs well in practice and produces good quality pictures. However, as the linear programme can produce better layouts (cf. Figures 1 and 9), it is available for 'optimized' drawings.

The final pass finds the spline control points for edge placement. There are several cases to consider, including self-edges, edges between nodes on the same rank, adjacent ranks or non-adjacent ranks, possibly in the presence of multi-edges. Short edges are easy to draw; DAG uses a single B-spline[9] that is shaped according to the number of parallel multi-edges being drawn. Long edges, which have endpoints on non-adjacent ranks, are more difficult since they may pass near other nodes and change direction. The B-spline control points are chosen for smoothness, and avoidance of node intersections and edge crossings.

The dummy nodes are created with a height a little larger than the highest real node in the same rank, and with a width usually twice the default node separation. This is big enough to make a spline inside the boundaries of the dummy node, if necessary. On the other hand, if there is space next to the dummy node, it is often preferable to encroach on it to draw a smoother spline.

The edge drawing procedure in draw_dag visits each dummy node to choose its spline control points. If a straight line between the neighbours of the dummy node does not intersect any other node, nor change the order of edge crossings, then the dummy node is deleted (Figure 11(a)). This removes small bumps in the edges. Next, draw_dag aims the incident edges as close to the centre of the dummy node box as possible, without intersecting adjacent nodes. draw_dag uses the points where the edges intersect the dummy node as spline control points. It also finds a third point to determine how the spline bends as it passes through the dummy node. If the intersection of the edges lies inside the dummy node (Figure 11(b)), draw_dag chooses it as the third control point. Otherwise draw_dag tries to move the edges toward each other so they do intersect (Figure 11(c)) and again takes the intersection as the third control point. If the edges cannot be made to intersect, but the angle between them is acute, draw_dag can still make a smooth spline by choosing the midpoint of the opposite side. When the angle is obtuse, that is the edges are close to being parallel, then it chooses the midpoint of the dummy node box. In the latter case the spline has two points of inflection as it passes through the dummy node, which is not as smooth as the other cases which have only one, but the extra turn is necessary when an edge makes a 'jog' as it passes through a rank near other nodes (Figure 11(d)).

Moving an edge may create an edge intersection near an incident real node, as in Figure 11(e). Such intersections are eliminated by sorting the incident edges according to the X co-ordinates of the other endpoints, and moving their nearby endpoints slightly (Figure 11(f)).

PERFORMANCE STATISTICS

Tables I–III show the performance of draw_dag using various node placement methods. The tables list the time spent in the three node placement passes: time spent on level (or rank assignment) t_{level}, time spent ordering nodes within ranks to reduce edge crossings t_{cross}, and time spent finding final co-ordinates $t_{\text{co-ord}}$. The total time t_{total} is the sum of these plus time spent in initialization and creating splines. The times were measured in seconds on a VAX-8650 under the 4.3BSD UNIX system and averaged over five runs to reduce statistical fluctuation. Times were measured on three graphs, the *World Dynamics* graph (Figure 1) and two procedure call graphs from C programs. *World Dynamics* has 43 nodes and 64 edges. The initial node ordering has 132 crossings. *CallGraph* is the graph of Figure 12, a good test case. It has 54 nodes and 90 edges, and the initial node ordering has 442 crossings. *SelfPortrait* is shown in

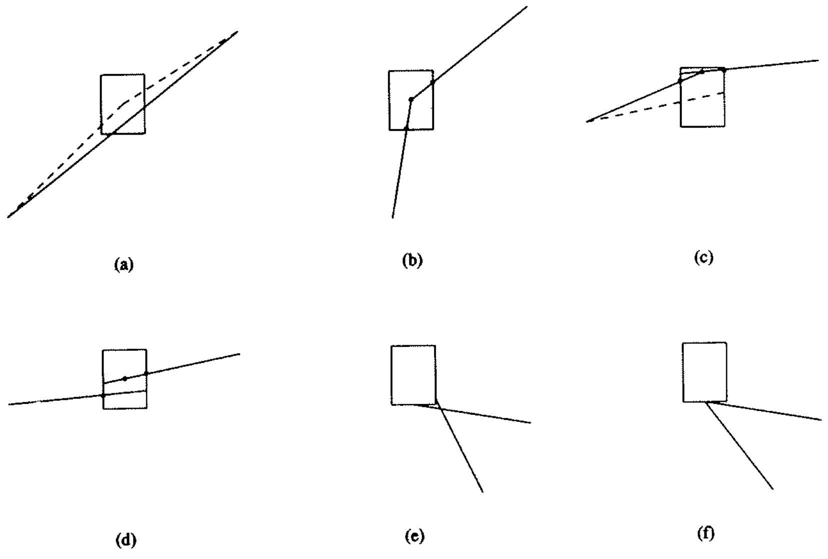

Figure 11

Table I. *World Dynamics*

Method	n_{cross}	t_{level}	t_{crossing}	$t_{\text{co-ord}}$	t_{total}
wmedian	29	0·04	0·34	0·58	1·02
bcenter	25	0·05	0·34	0·57	1·03
rmedian	30	0·05	0·35	0·56	1·03
wmedian+	20	0·06	0·85	0·52	1·47
bcenter+	21	0·06	0·85	0·43	1·40
rmedian+	20	0·05	0·90	0·41	1·43

Table II. *CallGraph*

Method	n_{cross}	t_{level}	t_{crossing}	$t_{\text{co-ord}}$	t_{total}
wmedian	59	0·06	0·34	0·35	0·81
bcenter	65	0·06	0·29	0·26	0·69
rmedian	62	0·05	0·37	0·26	0·75
wmedian+	34	0·06	0·87	0·27	1·27
bcenter+	41	0·05	0·90	0·25	1·29
rmedian+	34	0·05	0·85	0·36	1·34

Table III. *SelfPortrait*

Method	n_{cross}	t_{level}	t_{crossing}	$t_{\text{co-ord}}$	t_{total}
wmedian	22	0·11	0·38	0·58	1·15
bcenter	23	0·12	0·35	0·36	0·91
rmedian	34	0·10	0·39	0·67	1·27
wmedian+	17	0·11	0·83	0·40	1·45
bcenter+	19	0·10	0·81	0·61	1·63
rmedian+	18	0·10	0·91	0·34	1·45

Figure 13. It is the function call graph of the draw_dag part of DAG. It has 81 nodes, 107 edges and the initial node ordering has 170 crossings.

We tried three algorithms for ordering of nodes within ranks: (1) *wmedian* uses DAG's weighted median value; (2) *bcenter* uses the barycentre statistic described in Reference 5; and (3) *rmedian* uses the 'right median' statistic,[10] which is defined as the position of the median neighbour when the number of neighbours is odd, and the position of the right median neighbour otherwise. We also tried all three methods in combination with transposition of adjacent nodes. To simplify the comparison of relative performance, the iterative loop for minimizing edge crossing is set to terminate after exactly 20 iterations. In practice, the loop termination is determined by an adaptive parameter that depends on the convergent rate of a solution.[6] Thus, for example, the total time for the *World Dynamics* graph using *wmedian+* is slightly higher than it would be in an actual run. The experiments show that the algorithm

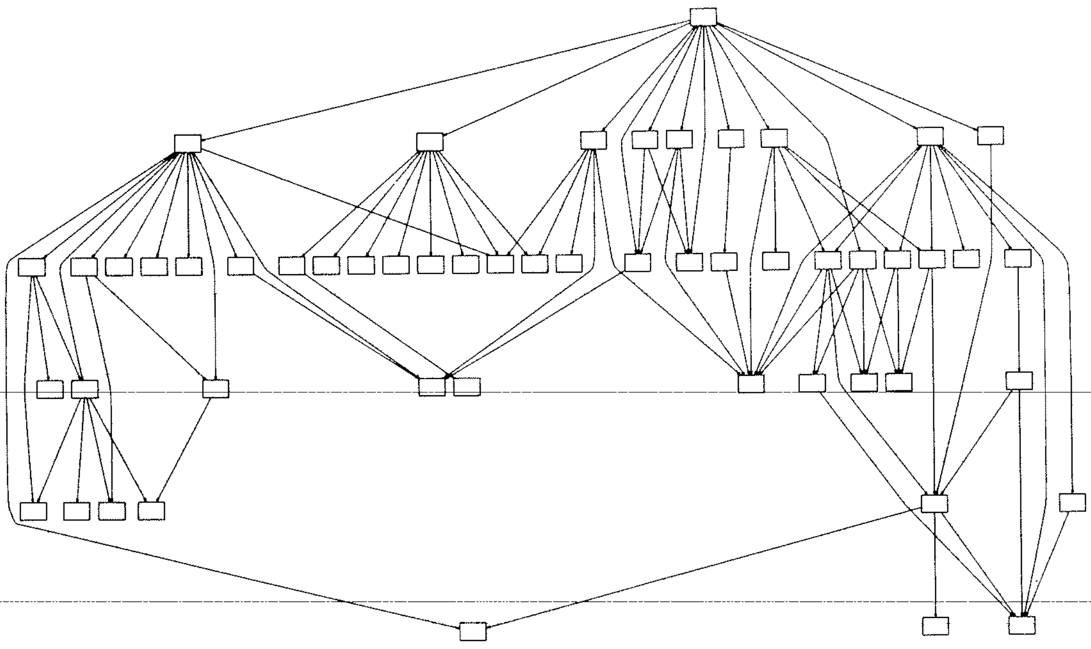

Figure 12

based on weighted median typically finds the drawings with the fewest number of crossings, and in about the same run-time as the other methods. All three methods improved by adding transposition, which seems well worth the extra computational expense. For instance, in *CallGraph* drawn by weighted median, transposition reduced the number of crossings by nearly 50 per cent, from 59 to 34.

CONCLUSIONS

We have presented an overview of DAG, a program for drawing directed graphs, and the language, aesthetic principles and algorithms it employs to produce pictures. Though the aesthetic principles used in DAG are similar to those described in References 3 and 5, we improved on the techniques developed in these works. First, an efficient and optimal algorithm for rank assignment was developed. This algorithm improves picture quality and enhances efficiency in later passes of the drawing algorithm. Our major divergence from earlier techniques is in the use of the median statistic instead of the mean statistic as the weight function that drives the heuristics for node ordering and placement. For node ordering, we showed experimentally that a weighted median heuristic coupled with local transpositions for descent performs best in reducing edge crossings. For node placement, DAG uses an l_1-norm objective function instead of a quadratic objective function.[5] This reduces the optimization problem to solving a linear programme. Even for a moderate size graph, solving the associated linear programme can take too long. We developed a fast heuristic method based on the median statistic and local improvements for node placement. Finally, to improve picture quality, we developed techniques for drawing edges using splines that reduce sharp corners and node intersections.

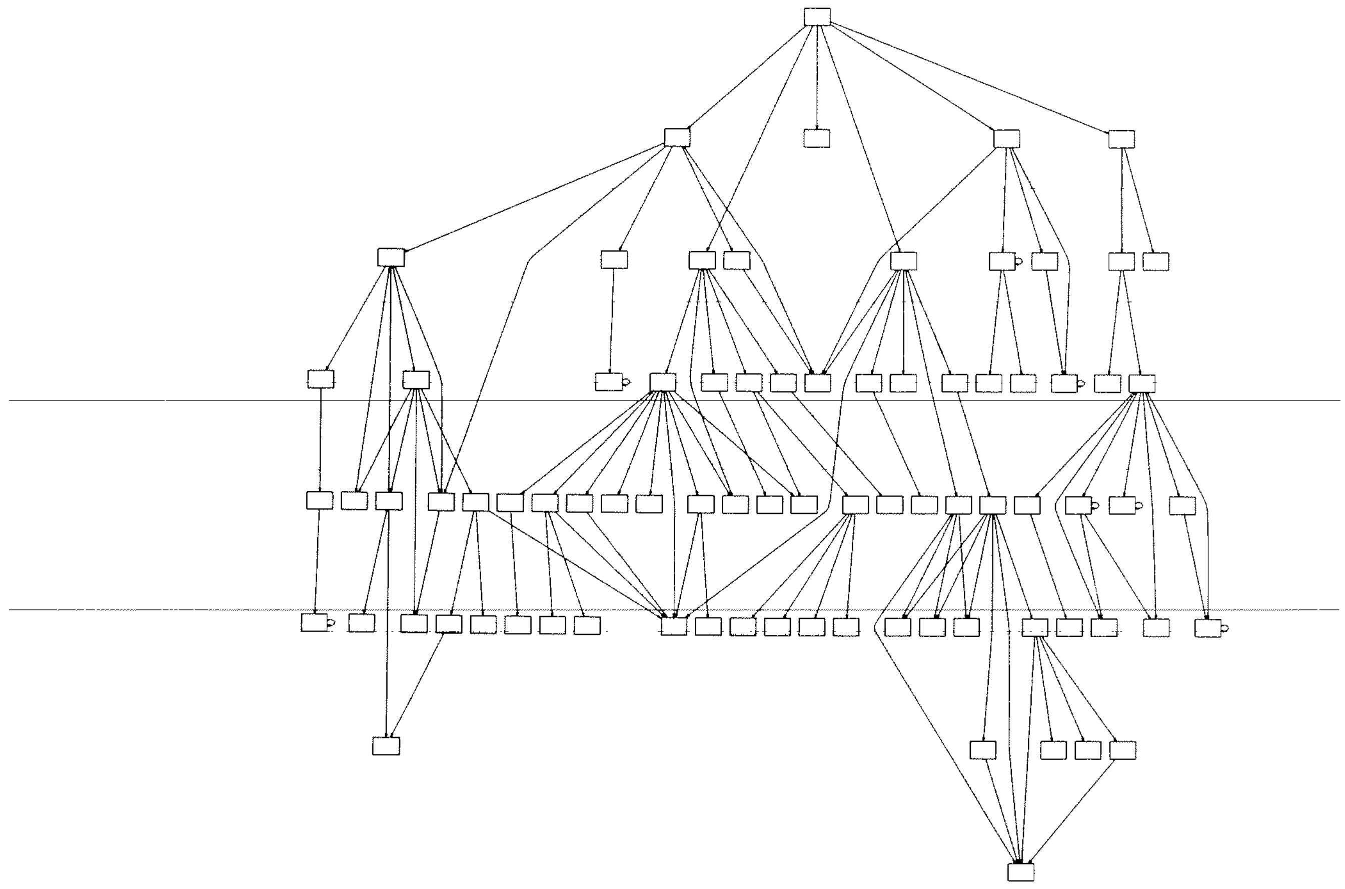

Figure 13

In the final analysis, the quality of a picture depends on the taste of the viewer. To be practical, a drawing tool must balance aesthetics and resource consumption. DAG has proved capable of producing good quality pictures in reasonable time. In our experience, graphs with a few hundred nodes and edges can be drawn in nearly interactive speed. In fact, to produce hard copies, typesetting tools such as PIC and TROFF typically consume more time than DAG.

REFERENCES

1. Brian W. Kernighan, 'PIC: a language for typesetting graphics', *Software—Practice and Experience,* **12**, (1), 1–21 (1982).
2. Adobe Systems, *PostScript Language Reference Manual*, Addison-Wesley, 1985.
3. Larry A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirahis and A. Tuan, 'A browser for directed graphs', *Software—Practice and Experience,* **17**, (1), 61–76 (1987).
4. Howard Trickey, 'Drag: a graph drawing system', *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography*, Cambridge University Press, April 1988.
5. K. Sugiyama, S. Tagawa and M. Toda, 'Methods for visual understanding of hierarchical system structures', *IEEE Trans. on Systems, Man, and Cybernetics,* **SMC-11**, (2), 109–125 (1981).

6. Emden R. Gansner, Stephen C. North and Kiem Phong Vo, 'Methods for drawing directed graphs', in preparation.
7. Emden R. Gansner, Stephen C. North and Kiem Phong Vo, 'On the level assignment problem', in preparation.
8. Peter Eades, Brendan McKay and Nicholas Wormald, 'An NP-complete crossing number problem for bipartite graphs', *Technical Report No. 60*, University of Queensland, Dept. of Computer Science, St. Lucia, Queensland, 4067, Australia, 1985.
9. Michael E. Mortenson, *Geometric Modeling*, Wiley, 1985.
10. Peter Eades and David Kelly, 'Heuristics for drawing 2-layered networks', *ARS Combinatoria,* **21**, (A), 89–98 (1986).