

Graph Visualization in Software Analysis

E. R. Gansner, E. Koutsofios, S. C. North and K. P. Vo
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

Directed graphs are ubiquitous in most aspects of software analysis. Presented abstractly, as a list of edges, a graph does not manifest much of the important structural information that becomes obvious if the graph displayed pictorially. This paper presents a technique for drawing directed graphs quickly and attractively. It also describes how a tool implementing this technique has been used, in conjunction with other programming and analysis tools, in various aspects of software engineering.

1 Introduction

Directed graphs serve as a fundamental data structure in software engineering. It takes little imagination to create a list of where directed graphs arise during the design, maintenance, analysis or reverse engineering of a piece of software. At the algorithmic level, graphs can describe data structures and finite state machines. In static and dynamic program analysis, graphs describe data flow and procedure call dependencies, and underlie profiling information. The inheritance structure in object-oriented systems forms a directed graph. Software configuration and version control dependencies are best understood as directed graphs. At the project management level, PERT charts and organization charts provide additional examples of directed graphs.

Directed graphs, whether explicit or derived by some analysis tool, are usually maintained in abstract form, frequently in a simple textual description. This abstract form is useful, in that it makes the graph amenable to further analysis by other computer-based tools. However, certain aspects of a graph can be very hard to understand from a list of edges. With a picture, the brain's pattern recognition ability can discern features and substructures totally disguised in an abstract presentation. Special nodes, groups of related nodes, distinguished paths quickly become ev-

ident. Sometimes the overall structure of the graph only becomes understandable when displayed in concrete form. At the least, a drawing of a graph provides a complementary way of understanding the graph. A problem with this technique is that, except for very small graphs, it can be difficult and tedious to draw graphs by hand. An automated drawing tool is necessary.

This paper describes an efficient tool, called *dag*, for producing high-quality drawings of directed graphs, and shows how this tool can be used, in conjunction with various software analysis tools, to give the software engineer a useful new view of graph data. In the following section, we provide an overview of *dag*'s technique for drawing directed graphs along with the aesthetic principles underlying its design. Section 3 shows how *dag* has been used in the design and analysis of software. We present a description of the implementation of *dag* in section 4, followed by some concluding remarks.

2 Drawing graphs

The *dag* program accepts a description of an abstract directed graph and produces a concrete embedding of the graph in the plane. It was designed to work particularly well on acyclic graphs such as trees and general hierarchies. It assumes that a directed graph has an overall flow or direction. Although, strictly speaking, this direction is ambiguous in graphs containing cycles, most directed graphs arising in software engineering have an underlying direction. 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).

There are certain principles that we follow in drawing directed graphs. The drawing should expose the hierarchical structure in the graph, aiming edges in the same general direction if possible. This aids finding directed paths and highlights source and sink nodes. The drawing should avoid visual anomalies, such as

edge crossings and sharp bends, that do not convey information about the underlying graph. Edges should be kept short. Lastly, the drawing should favor symmetry and balance. Clearly, these characteristics can conflict. For instance, a placement of nodes better emphasizing the hierarchy may force more edge crossings than necessary.

dag calculates the layout in four passes, generally guided by the aesthetic criteria above. The first pass places the nodes in discrete ranks. It does this optimally, in a sense to be made precise in section 4, while respecting the flow of the graph, i.e., the head of an edge tends to be assigned a higher rank than the tail. The second pass orders the nodes within ranks to avoid edge crossings. The third pass sets the actual coordinates of nodes by solving an optimization problem that prefers short straight edges. For aesthetics and clarity, *dag* draws edges as splines. The final pass thus involves finding the spline control points for edges. The entire procedure can be implemented efficiently. To show this, the figures below give the real time for each layout on a Sun-4/280.

The procedure *dag* uses to lay out a graph builds on the approach of Warfield [20] and Sugiyama, Tagawa, and Toda [16]. Our contributions include the formalization of passes 1 and 3 as optimization problems; the application of network simplex techniques to solve these problems; an improved heuristic for reducing edge crossings; and a method for representing edges as splines.

3 Graphs and software analysis

Because of its straightforward filter model, *dag* is compatible with a range of software engineering tools that create graphs. Output of pre-existing programs can be translated into *dag* format by scripts, while some newer programs are directly compatible. For instance, a short *awk* script translates the output of the standard *cflow* utility [17] into *dag* format. Similarly, the software configuration manager *nmake* [8] has a set of rules for translating makefile dependencies into abstract graphs and a script to convert graphs to *dag* specifications. Other applications include drawing graphs of concurrency conflicts in transactional databases, visual rendering of partial ordering of events generated by concurrent Ada programs, programs written in an experimental data-parallel language for a shared-memory multiprocessor, visual object recognition in artificial intelligence, and algorithms to solve the scheduling problem in computer-aided high-level microcircuit synthesis.

Most graph drawing programs are window-based. Two well-known examples are *Edge* [18] and *Grab* [14]. Progress in effective abstraction and interaction techniques, areas of current research [13] may help in understanding and exploring large systems represented as graphs. However, in the setting of a diverse collection of existing software tools, an explicit package that translates graphs to pictures works better for several reasons. It does not have to be run “by hand,” but can be embedded in scripts and combined with pre- and post-processors. It is more likely to be portable since it does not depend on any particular window system or graphics toolkit. It does not make many assumptions about the programs that create the graphs. Finally, in developing such a translator, one can concentrate on making the best pictures possible, whereas in interactive graph programs, the drawing quality sometimes seems secondary to the user interface.

Of course, *dag* can be integrated with an interactive graph viewer (e.g., section 3.4). There is a good partition of functionality, because both graph drawing and user interaction involve difficult programming, but the information they need to share is limited. The viewer simply provides an abstract graph to *dag*, and receives the layout coordinates in return.

The remainder of this section describes, in more detail, specific experiences of software visualization using *dag*.

3.1 Profiling

The profiler *gprof* [9] prints a summary of procedure calls made in a program execution and the CPU time taken by each procedure. The summary is given in tabular form. A profile may be easier to understand when drawn as a graph, using a scale of colors to show the costs of various procedures and procedure calls. An example graph is shown in figure 1.

We learned that using a rainbow of bright colors is confusing, because the eye does not automatically evaluate bright blue as less important than bright red. It is more helpful to vary both the color hue and brightness (or possibly saturation) to draw attention to expensive nodes and edges. The colored profile graph helps one to identify both expensive procedures and call paths, subject to *gprof*'s limitations.

3.2 CIA and CIA++

CIA is a C program database system [2]. It creates and accesses relational databases of the types, variables, procedures, and files that comprise a C program. CIA++ is a similar system for C++ programs

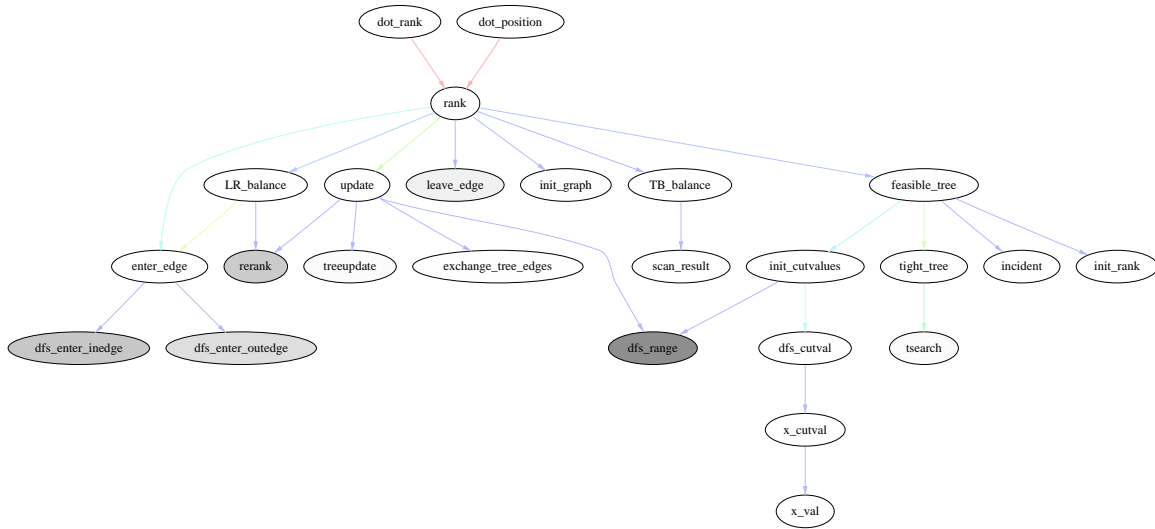


Figure 1: Call graph showing CPU costs (0.31 sec. real time).

[1]. Some CIA and CIA++ commands use a text-oriented relational view that is appropriate for tasks such as listing the sites where certain symbols are defined or used. However, to explore a high-level view, it is often helpful to see drawings of graphs created from the program database. The CIA *dagen* utility creates *dag* specifications of procedure call graphs, type dependencies, and file inclusion hierarchies. In general, *dagen* creates a graph of any two kinds of entities in its conceptual model. *dagen* may be run on the whole database or in a pipeline in which it receives the result of a database query. A type inheritance graph made by *dagen* is shown in figure 2.

Other tools in the CIA suite also create graphs. *incl* [19] analyzes relationships between include files to detect superfluous inclusion, and to prepare graphs for *dag* (figure 3).

ciatso performs topological sorting of call graphs and collapses strong components into nodes. *subsys* computes the reachable set of a CIA object under selected reference relationships and may be run as a pre-processor to *dagen* to extract or mark their nodes and edges, for instance, to highlight dead code.

The source release of AT&T C++ includes certain tools that process source code directly (without an intermediate database), including the *hier* program that creates *dag* files of inheritance hierarchies, and an *incl* utility. That these programs were created independently from the CIA tools suggests the general applicability of graph drawing techniques to source code analysis.

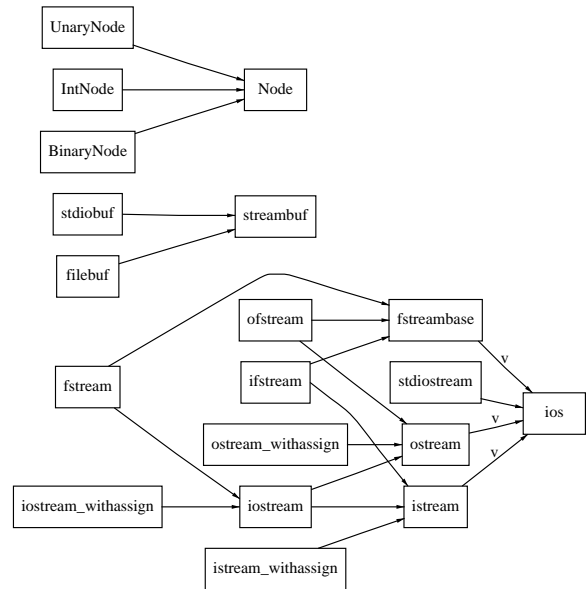


Figure 2: Type inheritance graph (0.16 sec. real time).

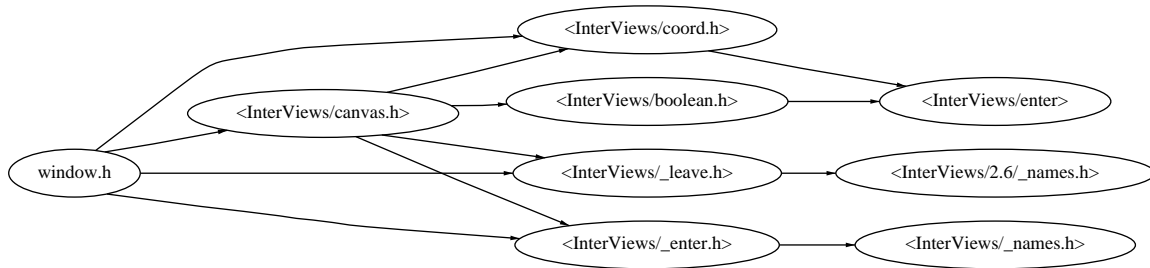


Figure 3: Include file graph (0.13 sec. real time).

3.3 Finite state machines and parse trees

Diagrams are a natural way to represent state machines. Multiple edges are common, and good edge label placement is vital for showing transitions. Figure 4 shows a graph of a large state machine (courtesy Scott Robinson at CMU).

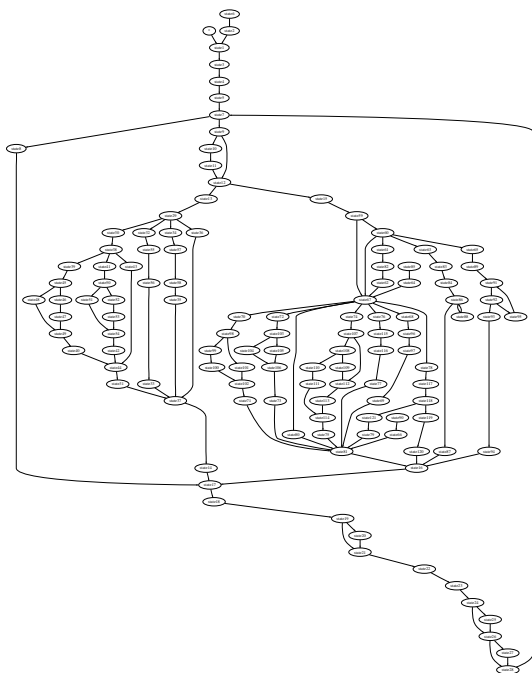


Figure 4: State machine graph (2.35 sec. real time).

Figure 5 shows a smaller state machine with labeled edge transitions.

Others have used dag to draw FSMs from a parser generator based on right regular grammars. Similarly, *dag*'s “ordered edges” are useful for drawing parse

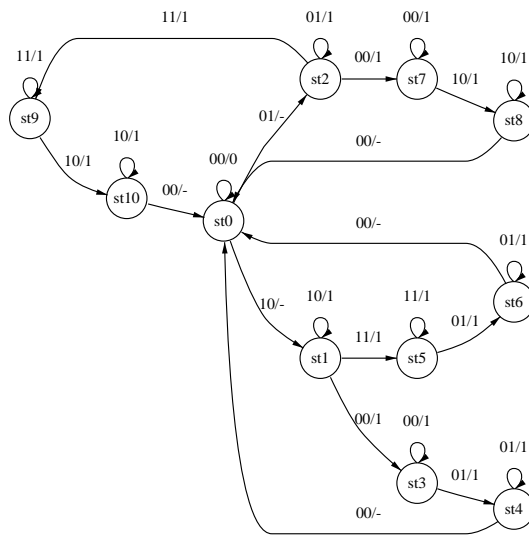


Figure 5: State machine graph with edge labels (0.28 sec. real time).

trees. These edges constrain the left-to-right order of their endpoint nodes.

3.4 Xray

Xray is an experimental program animation system. It uses graphics to visualize several aspects of the execution of programs. For some purposes, these graphical representations are easier to understand than conventional textual representations. *Xray* uses *lefty* [5] to implement its graphical views. *lefty* is a graphics editor in which all aspects of picture editing, including drawing, computing the layout, and binding user actions to editing operations, are described by a program in *lefty*'s language. This allows *lefty* to be customized for a variety of purposes. An important feature of *lefty* is its ability to communicate with other processes. In *Xray*, *lefty* communicates with CIA, *dbx* and *dag*. Currently, *Xray* animates sequences of function calls and also displays data structures graphically.

3.4.1 Function call animation

To use this feature, the user runs a special preprocessor that instruments the program [15]. The instrumented program, when executed, generates a trace log of function calls, their arguments and their return values. Using this log, as well as information extracted from the CIA database, *Xray* constructs and displays a dynamic call dependency graph. Figure 6 shows such a graph.

Nodes and edges in this graph are color-coded. Blue indicates code that was not exercised in this particular run. Green indicates objects that were exercised at least once. Red indicates heavily used code. The sequence of function calls can be animated. When a function is called, its node changes to yellow. When the function returns, its color reverts to either green or red, depending on its frequency of usage. The user can perform certain filtering operations on the graph, such as deleting individual nodes and collapsing several nodes into one. As the graph changes, the layout of the graph changes also. *Xray* uses *dag* to compute these layouts. *lefty* sets up a two-way communication channel with *dag* and whenever the graph changes, *lefty* sends the changed graph to *dag* and waits for *dag* to return the node and edge coordinates.

3.4.2 Display of data structures

Xray can display data structures graphically. This feature does not require any modifications to the source

program. Figure 7 shows a binary search tree that was generated from a user program using *Xray*.

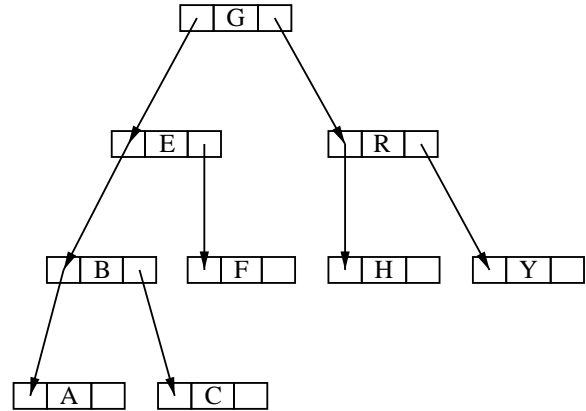


Figure 7: Binary search tree (0.15 sec. real time).

Such graphical views are easier to understand than textual views. Just by looking at the picture in figure 7, we can tell it is a tree, while this takes much longer to realize from studying a listing.

Drawing data structures with *Xray* is implemented using existing tools as separate processes. *lefty* opens a terminal window that runs a multiplexing/demultiplexing process. This process accepts input from the user, sends it to *dbx*, and copies *dbx*'s response to the terminal window. It also accepts data structure queries from *lefty*, sends them to *dbx*, and then parses *dbx*'s responses and sends them to *lefty*. *lefty* then sends these data structure graphs to *dag* to obtain their layout.

Using *dbx* as a subprocess has the advantage that we do not have to modify *dbx* or write our own debugger. The interface to *dbx* consists of a filter that parses *dbx*'s responses and converts them to statements that *lefty* can process. Using a different debugger is trivial; we just need a different filter.

4 Implementation

As discussed in Section 2, *dag* uses four passes to lay out a graph. In this section, we describe how each of these passes is implemented. The interested reader is referred to [11, 10] for complete details.

4.1 Rank assignment

The first pass involves assigning nodes to discrete levels in a way that respects the overall flow of the

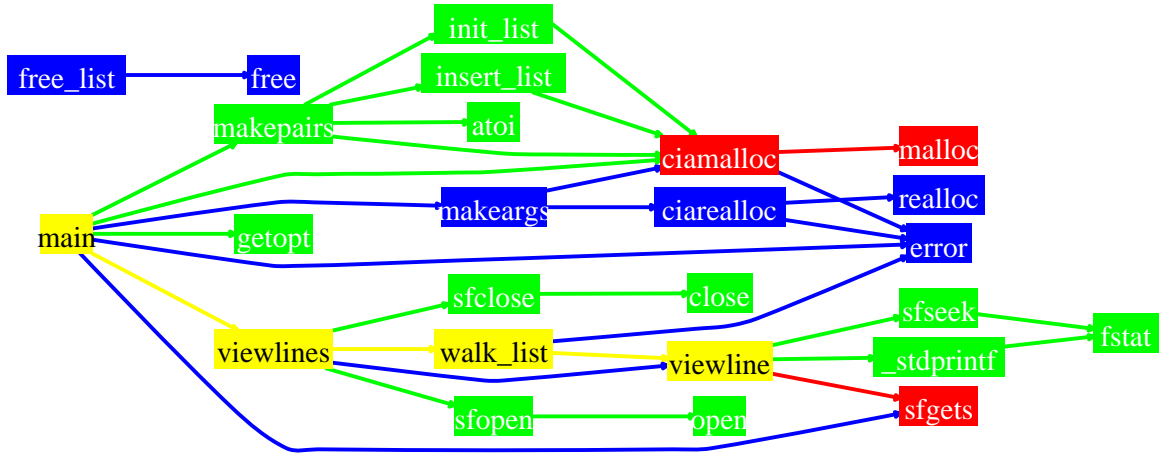


Figure 6: Dynamic call dependency graph (0.26 sec. real time).

graph. Formally, we assign 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 to be possible, the graph must be acyclic. Because the input graph may contain cycles, a preprocessing step detects cycles and breaks them by reversing certain edges. 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. Reversing all back edges (i.e., edges from a node to an ancestor in the search tree) makes the graph acyclic.

It is desirable to find an optimal node ranking, i.e., one for which the sum of all the edge lengths is minimal. Besides making better layouts, short edges reduce the running time of later passes whose times depend on the total edge length. Finding an optimal ranking can be generalized and reformulated as the following integer program:

$$\begin{aligned} \min \quad & \sum_{(v,w) \in E} \omega(v,w)(\lambda(w) - \lambda(v)) \\ \text{subject to} \quad & \lambda(w) - \lambda(v) \geq \delta(v,w) \forall (v,w) \in E \end{aligned}$$

The *weight* function ω maps the edge set E into the non-negative rational numbers.

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 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 by requiring all tree edges to be tight. A spanning tree is *feasible* if it induces a feasible ranking.

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.

It is now easy to state the algorithm. We compute an initial feasible tree, using essentially the technique described in [16]. Given a feasible tree, we pick a tree edge e with negative cut value. Then, from all non-tree edges going from the head component to the tail component of e , we pick one of minimum slack and add it to the tree, removing e . This gives us a new spanning tree, and the minimum slack condition guarantees that the tree is feasible. This process continues until the spanning tree has no edges with negative cut values, at which point it corresponds to an optimal ranking. For further discussion of the termination and optimality of this algorithm, the interested reader is referred to [12, 4, 3].

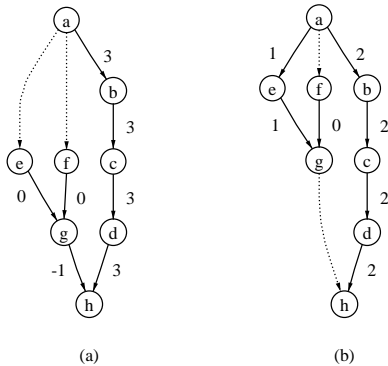


Figure 8: Finding an optimal feasible tree.

A small example of running the network simplex algorithm is shown in figure 8. 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.

4.2 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.

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 [6]. We use a variation of the median heuristic [7], which follows a scheme first suggested by Warfield [20]. An initial ordering within each rank is computed. This is done by a 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. A sequence of iterations is then performed to try to reduce the number of crossings. We use an adaptive strategy that iterates as long as the solution has improved at least a few percent over the last several iterations.

Each iteration alternately 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. The weight of a node is the median of the 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. When there are two median values, we use an interpolated value biased toward the side where vertices are more closely packed. After all the vertices in the rank have been assigned a weight, they are re-ordered by sorting on these weights. Vertices with no adjacent vertices on the previous rank are left fixed in their current positions with non-fixed vertices sorted into the remaining positions. At the end of each iteration, we apply an additional heuristic that transforms a given ordering to one that is locally optimal with respect to transposition of adjacent vertices, thereby reducing obvious crossings.

One small point is that the original graph may have edges between nodes on the same rank. We call these “flat edges.” 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 transposition and the sorting processes must not exchange nodes against the partial order.

4.3 Vertex coordinates

The third pass sets node coordinates. Previous work has treated this as a postprocessing step of the preceding pass, 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. 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:

$$\begin{aligned} \min \quad & \sum_{e=(v,w)} \Omega(e)\omega(e)|x_w - x_v| \\ \text{subject to} \quad & x_b - x_a \geq \rho(a,b) \end{aligned}$$

where a is the left neighbor of b on the same rank and $\rho(a,b)$ is a function on pairs of adjacent nodes in the same rank giving the minimum separation between their center points. $\Omega(e)$, 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. 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.

It turns out that this optimization problem is amenable to the same efficient network simplex solution as that used for rank assignment. We do this by constructing an auxiliary graph.

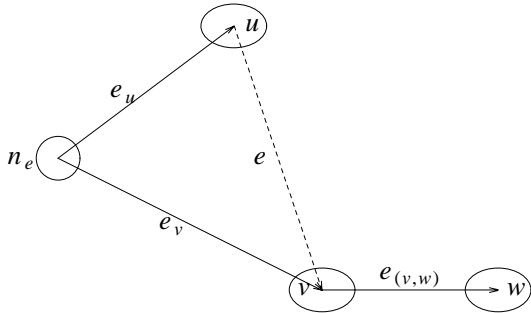


Figure 9:

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. Standard arguments (e.g., see [10]) show that solving the optimal rank assignment problem on G' gives us the desired optimal solution to the coordinate assignment problem on G .

4.4 Drawing edges

In *dag*, edges are drawn using splines. The splines are computed by iterating over the edges, routing the shorter edges first because they can often be drawn as straight lines. For a given edge, we try to find the “smoothest” curve between two points that avoids the obstacles of other nodes or splines. This is done by computing a polygonal region of the layout in which the spline may be drawn, computing the best spline that lies within the region, and then clipping the spline to the boundaries of the endpoint nodes. A region and its spline are illustrated in figure 10. (Graph data courtesy of Ian F. Darwin, SoftQuad Inc.)

The associated edge is from “Interdata” to “Unix/TS 3.0”.

A region is composed of collection of boxes B_0, \dots, B_m , parallel to the coordinate axes, such that B_i has edges in common with B_{i-1} and B_{i+1} . To curve as smoothly as possible, the boxes should be as large as possible. In general, the sequence alternates between inter-rank boxes and virtual node boxes. The former are made as wide as the graph. The latter contain the virtual node corresponding to the intersection of the edge with a rank, plus all extra space up to the adjacent nodes on the same rank. Occasionally, there are a few additional boxes near the head or tail node in order to route the spline to the appropriate side of the node to avoid intersection with previously drawn splines.

Certain cases must be handled specially. For multiple edges between the same pair of nodes, 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. Flat edges are handled much like inter-rank edges, but the region routes past intervening nodes and spaces between nodes. For multiple flat edges, a spline is computed for the first one, and succeeding edges are drawn by adding Y coordinate displacements. Self-edges are drawn as loops on the sides of nodes, as a special case. If there are multiple edges, their loops are nested.

Given the boxes composing the polygon region, and initial and final points q and r , we then must construct control points of a piecewise Bezier curve connecting q and r and lying within the region. The computation

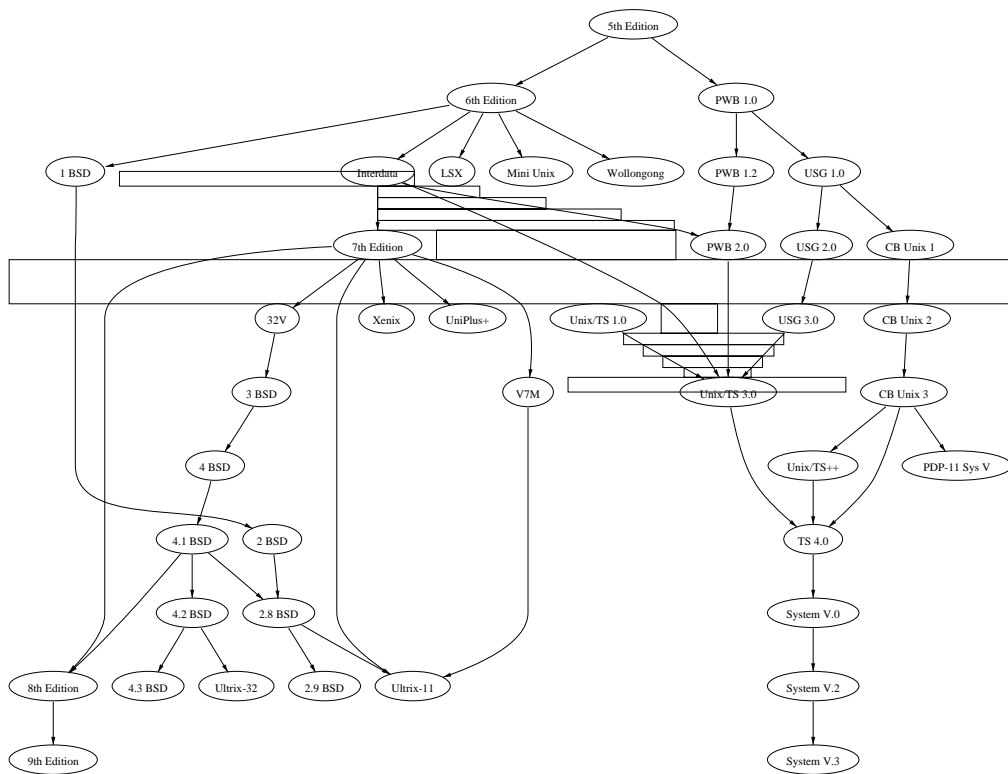


Figure 10: Region for a spline.

has three stages. First, a piecewise linear curve or path lying entirely inside the region and connecting q and s is computed. Then, the vertices of this path are used as hints in the computation of a piecewise Bezier spline. In the worst case, we can have one Bezier spline per box. In most cases, however, our approach generates significantly fewer splines since, unlike a line, a spline can curve around obstacles. Finally, after the spline has been computed, the width and position of its virtual nodes are adjusted. Specifically, the bounding box of the virtual node corresponding to B_i is replaced by the smallest box fitting in B_i that contains the generated spline.

5 Conclusions

We have described a program for drawing directed graphs and shown how it has been used in various areas of software engineering. The layout technique runs fast enough for interactive use and makes drawings that compare well with previous work as to being readable and visually pleasing. By using the layout algorithm with graph data produced by various software analysis tools, the programmer is better enabled to visualize the underlying program. Certain substructures and relationships, which would be buried in a textual or abstract graph representation, leap to the eye when the graph is concretely displayed. Such representations could be produced manually, but the difficulty and tedium involved usually prohibit this. By automating the layout, software visualization becomes a viable tool in the programmer's repertoire.

References

- [1] J. Grass and Y.-F. Chen, "The C++ Information Abstractor," *The Second USENIX C++ Conference*, pp. 265-277, 1990.
- [2] Y.-F. Chen, "The C Program Database and Its Applications," *USENIX Baltimore Summer Conference Proceedings*, pp. 157-171, 1989.
- [3] V. Chvatal, **Linear Programming**, W. H. Freeman, New York, 1983.
- [4] W. H. Cunningham, "A network simplex method", *Mathematical Programming* 11, 1976, pp. 105-116.
- [5] D. Dobkin and E. Koutsofios, "LEFTY: A Two-view Editor for Technical Pictures," *Proceedings of Graphics Interface '91*, pp. 68-76.
- [6] P. Eades and B. McKay and N. Wormald, "On an Edge Crossing Problem," *Proceedings of 9th Australian Computer Science Conference*, 1986, pp. 327-334.
- [7] Eades, P. and N. Wormald, "The Median Heuristic for Drawing 2-Layers Networks," Technical Report 69, Dept. of Computer Science, Univ. of Queensland, 1986.
- [8] Glenn S. Fowler, "A Case for make," *Software - Practice and Experience* 20(S1), June 1990, pp. 30-46.
- [9] S. L. Graham and P. B. Kessler and M. K. McKusick, "gprof: A Call Graph Execution Profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices* 17(6), June 1982, pp. 120-126.
- [10] E. R. Gansner and E. Koutsofios and S. C. North and K.-P. Vo, "A Technique for Drawing Directed Graphs", submitted.
- [11] E. R. Gansner and S. C. North and K.-P. Vo, "DAG - A Program that Draws Directed Graphs", *Software - Practice and Experience* 17(1), 1988, pp. 1047-1062.
- [12] E. R. Gansner and S. C. North and K.-P. Vo, "On the Rank Assignment Problem", in preparation.
- [13] Tyson R. Henry and Scott E. Hudson, "Interactive Graph Layout," *Proceedings ACM SIGGRAPH Symposium on User Interface Software and Technology*, November, 1991, to appear.
- [14] L. A. Rowe and M. Davis and E. Messinger and C. Meyer and C. Spirakis and Allen Tuan, "A Browser for Directed Graphs," *Software - Practice and Experience* 17(1), January 1987, pp. 61-76.
- [15] D. S. Rosenblum, "Towards a Method of Programming with Assertions," *Proceedings of 14th International Conference on Software Engineering*, to appear.
- [16] K. Sugiyama and 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.
- [17] Unix Software Operation, **Unix System V Release 4 Programmer's Reference Manual**, Prentice Hall, 1990.

- [18] W. F. Tichy and F. J. Newbery, "Knowledge-based editors for directed graphs," *Proceedings of 1st European Software Engineering Conference*, E H. Nichols and E D. Simpson, editors, Springer-Verlag, 1987, pp. 101-109.
- [19] Vo, Kiem-Phong and Yih-Farn Chen, "Incl: A Tool to Analyze Include Hierarchies", AT&T Bell Laboratories, Murray Hill, N.J., to appear, 1991.
- [20] John Warfield, "Crossing Theory and Hierarchy Mapping," *IEEE Transactions on Systems, Man, and Cybernetics* SMC-7(7), July, 1977, pp. 505-523.