1994

A layout algorithm for hierarchical graphs with constraints

Michael Slade

Rochester Institute of Technology

Computer Science Department

A Layout Algorithm for Hierarchical Graphs with Constraints

by

Michael L. Slade

A thesis submitted to

The Faculty of the Computer Science Department

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science.

Approved by:

Dr. Gilbert B. Porter III

Professor Stanislaw Radziszowski

Professor Peter G. Anderson

September 2, 1994

Title of Thesis: **A Layout Algorithm for Hierarchical Graphs with Constraints**

I, Michael L. Slade hereby grant permission to the Wallace Memorial Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

September 2, 1994

Acknowledgement

I gratefully acknowledge the sponsorship of the Xerox Webster Research Center and the generous assistance of my colleagues, both old and new as I left one world and began learning about a new one. Each member of my thesis committee, Gil Porter, Stanislaw Radziszowski and Peter Anderson has had special expertise which they have generously shared with me as the thesis grew from ill-formed ideas into reality. Throughout this process my wife, Judy has been most understanding as I became a student once again.

Abstract

A new method is developed for reducing edge crossings in the layout of directed graphs for display. The method will reduce edge crossings in graphs which have constraints on the location or movement of some of the nodes. This has not been available in previously published methods. An analysis of the strategies used to choose rank pairs for edge crossing reduction shows that this choice will dramatically affect the amount of crossings eliminated. This method is directly applicable to the reduction of edge crossings in the general graph.

Key Words and Phrases

directed graphs

hierarchical graphs

graph layout alogrithms

edge crossings

Computing Review Subject Codes

Primary: G.2.2 Graph Theory

Secondary: F.2.2 Nonnumerical Algorithms and Problems

Contents

1 Introduction and Background

1.1 Problem Statement

This thesis develops a method of reducing the edge crossing count of a hierarchical graph. The method includes techniques for situations which have constraints on node locations. Since the algorithm ignores the edge directivity, it is also applicable to reducing edge crossings in the general graph.

1.2 Introduction

A hierarchical or directed graph consists of a set of nodes and a set of straight lines that run between pairs of nodes. The lines, which in a directed graph have an arrowhead at one end, are called edges. A formal definition will be given in section 1.4.1. Without changing the topology, a graph can be displayed with the nodes confined to rectangular grid coordinates. Then the horizontal rows of nodes are called ranks or levels.

The directed graph forms a natural representation in such diverse applications as PERT charts, hypertext document relations and visualization of the structure of complex organizations. Depending on the graph purpose, the nodes are given

different meanings such as tasks in a PERT chart or paragraphs in a hypertext document. There can be different types of nodes within the same graph, such as nodes for authors and other nodes for paragraphs in the hypertext document. The different node types can be represented by different shapes. Each edge also has a meaning, such as in a PERT chart where they show the ordering of tasks. In the PERT chart example the edges usually have a direction indicated by an arrowhead indicating the time sequence.

In addition to showing relationships with nodes and edges, it is often desirable to have fixed relationships among node types. For example, square shaped nodes might be shown to the right of any round nodes on the same level. In the hypertext example with nodes for authors and paragraphs, the author nodes might be in different ranks than the paragraph nodes in order to more easily distinguish between node types.

The visualization of systems modeled as a hierarchical graph is often accomplished via graphical display on either a CRT or hardcopy device. The location of the nodes in the display fixes the end points of the edges. However, an arbitrary node layout will most likely result in a visualization that confuses the viewer rather than enhances his/her understanding of the system being represented. The requirement is that the graph have good readability; ie. that it be easy for the

viewer to gain understanding of the system being modeled by the graph.

Notions of graph readability [TDBB88, STT81] are difficult to quantify. Placing the nodes on a rectilinear grid is most often more aesthetically pleasing then a random placement. A fixed spacing between parallel edges also looks pleasing, but more complex standards for readability are often hard to express, much less quantify.

Minimization of edge crossings and the resultant groupings of nodes is intuitively an important aspect of readability. It is not surprising, therefore, that edge crossing minimization is part of hierarchical graph layout algorithms. Once the number of crossings has been reduced to an acceptable level (reduction to the theoretical minimum is not required for readability improvement) other aesthetic considerations may be applied. Recent hierarchical graph programs have included maximizing edge straightness [STT81] and minimizing edge length [GNV88] as well as more subtle features such as the shape of the bends in edges [GNV88].

Application specific constraints may require nodes of different types to be displayed on different ranks or in some order within a rank. Alternatively, different sets of edges may be shown in distinct views of the graph while keeping the location of the nodes constant in all views. This thesis proposes a constraint graph layout theory for an interesting class of constraints and a modified layout

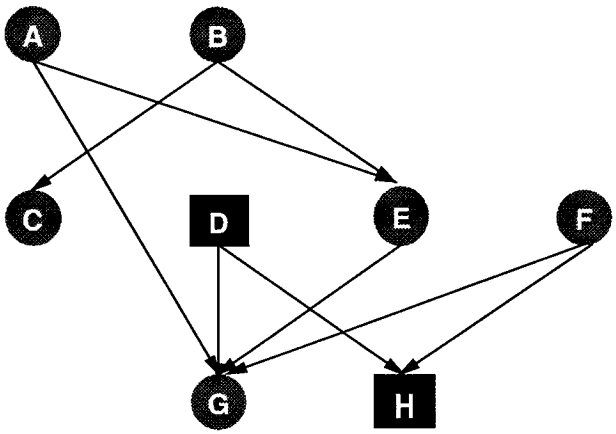


Figure 1: A directed graph with edge crossings.

algorithm which implements it. Specifically, layout algorithms which attempt to maximize readability have been developed for the general directed graph. This thesis proposes modifications to the hierarchical graph layout algorithm of Sugiyama, Tagawa, and Toda [STT81] to allow the case of constraints on the node and edge layout.

An illustration of the problem is shown in figures 1 and 2. Figure 1 shows a relatively simple directed graph which has been layed out arbitrarily. The graph, G, has a set of eight nodes, V, and a set of nine edges, E. There are two types of nodes: squares and circles. The vertices are shown in three ranks or levels and simple count of the number of edge crossings shows that there are 4. It

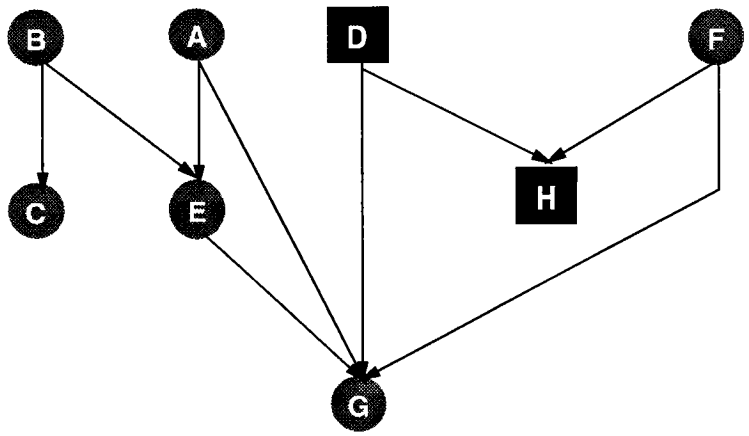


Figure 2: The directed graph of figure 1 with nodes rearranged to minimize edge crossings.

is not obvious that rearranging the node placement while maintaining the same connectivity can reduce the number of edge crossings to zero as shown in figure 2. (The bend in the edge (F, G) in figures 2 and 3 is a consequence of an implicit dummy node which will be explained later.) If we require that all circles must appear to the left of any square within a level, then figure 3 shows one possible solution. This thesis proposes an algorithm that given a graph such as shown in figure 1, will produce a layout such as shown in figures 2 and 3, i.e. one with a reduced number of edge crossings.

1.3 Previous Work

The previous work on directed graphs can be benchmarked by two recent literature surveys of graph drawing. Tamassia, Di Battista, and Batini [TDBB88] surveyed the literature as background to giving their approach to the general graph problem. Subsequently, Eades and Tamassia [ET88] issued an annotated bibliography on algorithms for graph drawing.

Before looking at the automated computer tools for directed graph drawing, it is instructive to examine the state of the art in manual computer graphics packages that could be used to draw a directed graph. Here, 'manual' means that the tool does not embody the notion of nodes and edges and their placement. 'More II' [Sym88] on the MacIntosh or the Viewpoint graphical editor [LENW82], for example, are of this class. While both of these are powerful drawing programs, the placement of nodes and links is left to the user with, at best, background grid insertion points to make the figure look more orderly. There is no method for minimizing the number of link crossings.

True automated directed graph layout tools are now commonplace but their layout algorithms are often not very sophisticated. One example is the popular 'grapher' package of InterLispD [Xer85] used in such programs as Notecards

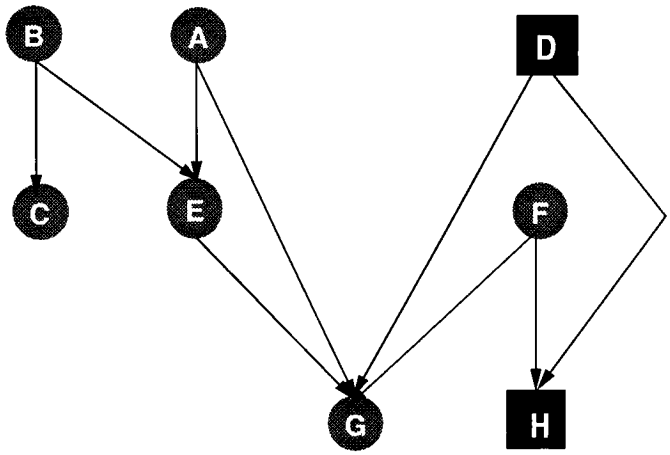


Figure 3: The directed graph of figure 1 rearranged to minimize edge crossings with the constraint that on any level the squares must be to the right of the circles on the same rank.

[HMT87]. A depth-first search is done for nodes to be plotted but when a cross link is required, the link is not drawn. Instead a duplicate 'marked' node is inserted. In order to simplify the visualization when the graph gets cluttered with nodes and edges, a 'virtual' node is used to compact a collection of nodes and links into a single node. The virtual node can be expanded to show additional detail.

Graph layout algorithms can be categorized by the types of graphs handled starting with trees and becoming more general. Wetherell and Shannon [WS79], Vaucher [Vau80], and Reinhold and Tilford [RT81] give solutions for the tree layout problem. Woods [Woo81] gives an algorithm for planar graphs. With his

algorithm, certain additional constraints are tractable. For instance, if the planar graph must be no larger than a maximum size, his algorithm can be used. Although a directed hierarchical graph could be planar, in general it is not. Even if a directed graph were planar, the Woods approach is very unlikely to yield a hierarchical layout.

The problem of finding the minimum number of edge crossings of a directed graph, even in the simple case of only two levels where the positions of the vertices in one of the levels is fixed, has been shown to be NP-complete by Eades, et al. [EMW85]. This does not mean that it is impossible to reduce the number of crossings. The minimum number of crossings may be calculated by permuting vertices and then counting the number of crossings for each combination. Any of the orientations that give the fewest crossings is a valid solution. This becomes impractical for any problem with more than a few nodes as the calculation time increase exponentially with the node count.

There have been two heuristic approaches to the directed graph problem. The first finds some form of a maximal planar subgraph of the given graph and then adds the rest of the nodes in a fashion that tends to minimize the number of crossings. This method has been suggested by Woods [Woo81]. However, this method is not applicable when a hierarchical layout is required.

The other approach stems from the work of Warfield [War77] in developing a crossing theory for hierarchical mappings. The problem is subdivided into two phases. The first arranges the vertices into a proper hierarchy, but in principle has no effect on the number of crossings for the graph. The second phase is divided into two parts. The first part of this phase minimizes the number of crossings between an adjacent pair of levels of the hierarchy by an algorithm or heuristic that rearranges the ordering of vertices within a level. This procedure is then repeated with other pairs of levels. The second part of this phase requires a second heuristic to decide the ordering of pairs.

While a more detailed analysis of this approach is given in the next subsection, some comments are appropriate here. The scheme is not guaranteed to give a solution with an absolute minimum number of crossings; it only yields a local minimum. Several different algorithms have been used to decide the ordering of pairs of levels to subject to the crossing reduction process, i.e. go down pairwise for some arbitrary number of passes or alternate between a pass going down and then one going up. None of them avoids the problem that the movement of a node along a rank to reduce the number of crossings between its rank and the rank below may increase the number of crossings between its rank and the rank above. It is possible that an adjacent level reduction will actually increase the total crossing

count.

Almost no attention has been paid to the problem of laying out a hierarchical graph with constraints. Gansner [GNV88] considers the case of edges of different weight in which the algorithm places greater emphasis on trying to shorten the links of higher weight. But the more general problem of applying constraints to the layout of a directed graph has not been addressed. Example applications that require constraints would be keeping the critical path in a PERT chart vertically aligned or having direct staff reports on a different rank than line reports in an organization chart.

1.4 Theoretical and Conceptual Development

1.4.1 Directed Graph Description

The notation and definitions used herein are that of Sugiyama et al [STT81] which follow that of Warfield [War77] with one exception: hierarchies have their 'roots' at the top which is level 1. As already noted, a directed graph $G(V, E)$ is composed of a set of vertices, V and a set of edges E. So far, there has been no restriction on where the set of vertices are placed in the plane.

Restricting the placement of all $v, v \in V$, so that they are situated on the lattice

points of a rectangular lattice can be accomplished so that it does not change the topology of G. This allows the description of subsets of V which are somewhere on a horizontal level (row or rank) n_i. Expressing the partitioning of V into n subsets as

$$V = V_1 \cup V_2 \cup \cdots \cup V_n \quad (V_i \cap V_j = \emptyset, i \neq j) \tag{1}$$

gives an n level hierarchy with the subset V_i at the ith level. The notation for a directed graph of n levels is $G = (V, E, n)$ where the order of the vertices within a level has not yet been described.

For each edge

$$e = (v_i, v_j) \in E, \text{ with } v_i \in V_i, v_j \in V_j \text{ and } i < j. \tag{2}$$

There are several implications of the requirement that $i < j$. First there can be no edges between nodes on the same level even if they are adjacent, i.e. in figure 1 there may not be edges (A, B) or (D, E) or (D, F). The restriction on edges between adjoining vertices can be relaxed without much effort, and in this implementation it is another form of constraint. Second, all edges must point downward and thus cycles are not allowed. Once again this is overly restrictive. Some workers have used appropriate temporary arc direction reversals to overcome this limitation. In our implementation cycles are allowed.

The edge (A, G) of figure 1 spans more than level, ie $j - i > 1$. If all edges span only one level then the directed graph is, by definition, a proper $n-$level hierarchy.

Continuing with the formulation of Sugiyama, this is equivalent to stating that

$$E = E_1 \cup E_2 \cdots \cup E_{n-1} \quad (E_i \cap E_j = \emptyset, i \neq j) \tag{3}$$

where $E_i \subset V_i \times V_{i+1}, i = 1, \ldots, n-1$. Note that requiring that the ends of the edges in each subset E_i come from adjacent levels forces the hierarchy to be a proper hierarchy.

Any hierarchy may be made into a proper hierarchy by inserting 'dummy' vertices of degree two at each of the intermediate levels of any edge where $j-i > 1$. If these additional vertices are inserted along the edge they do not change the number of edge crossings; the original and new graph are homeomorphic. For example the hierarchy of figure 1 can be made proper by the insertion of vertex K on level 2 between vertices C and D with edges (A, K) and (K, G) replacing (A, G). No generality is lost by the requirement that the hierarchy is proper.

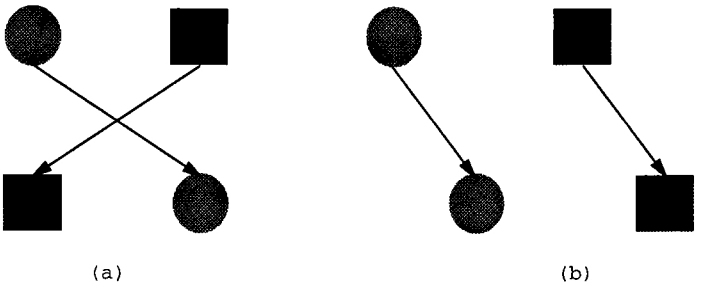


Figure 4: Topological possibilities for two edges. (a) Edges that cross. (b) Edges that do not cross.

1.4.2 Edge Crossing Counting

For a proper hierarchy, the number of crossings is only affected by the ordering of the vertices and not their separation. Each level consists of a subset of vertices, V_i, whose order is $\sigma_i = v_1 v_2 \ldots v_{|V_i|}$, where $\mid V_i \mid$ is used to express the number of vertices in the subset V_i. The set $\sigma = (\sigma_1, \ldots, \sigma_n)$ gives the orderings for all levels within the $n-$level hierarchy, and the directed graph is completely defined by $G = (V, E, n, \sigma)$.

The crossing reduction problem now can be expressed for any $G = (V, E, n, \sigma)$ as finding a set σ which minimizes the number of crossings. In order to decide which set σ yields the minimal crossing count, a method is needed to count the number of crossings for a given σ.

Before counting the number of edge crossings, it must be determined if two

edges cross. Figure 4 shows the topological possibilities.

Even in the simplest case of two edges $(u, r) and (v, s)$, determining whether the edges cross is easy to do by eye but must be expressed mathematically. What is needed is to express the difference between the case of figure 4(a) and (b). Let $X(i)$ be the x coordinate of each vertex. Remember that the actual value of the x coordinates is not important; only the relative ordering of vertices within each level must be preserved if the edge crossings are to be uneffected. Then, if the sign of

$$T = (X(u) - X(v))(X(r) - X(s)) \tag{4}$$

is negative, the edges cross[EW89]. However, the need for a multiplication step is computationally expensive but can easily be avoided.

One approach to calculating the number of crossings for a given σ is to apply the preceding procedure to all pairs of edges emanating from all pairs of vertices within a level and then summing over all levels. A more elegant formulation of the solution was given by Warfield [War77, equations 2-4].

Figure 5(a) shows a 2-level hierarchy and (b) its adjacency matrix. In general two levels will produce an $m \times p$ matrix where $m = \mid V_t \mid$ and $p = \mid V_{t+1} \mid$. Each

row in the matrix

$$R^i = r_1^i, r_2^i, \ldots, r_j^i, \ldots, r_{|V_{t+1}|}^i \tag{5}$$

is a sequence of 1's and 0's indicating whether the ith vertex in sequence σ_t has an edge that goes to the jth vertex in sequence σ_{t+1}. The number of crossings caused by the edges which emanate from two vertices on the same level $K(R^i, R^j)$ is given by

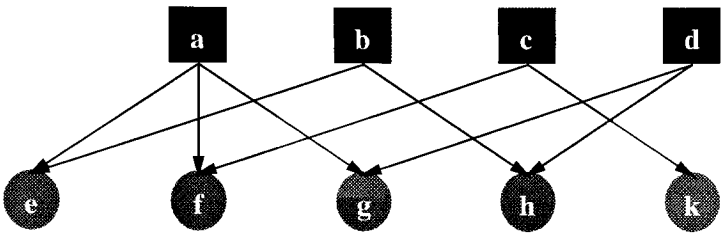
$$K(R^i, R^j) = \sum_{\tau=1}^{p-1} \sum_{\rho=\tau+1}^{p} r_\tau^j r_\rho^i, \text{ where } p = \mid V_{t+1} \mid . \tag{6}$$

In order to see this more clearly, figure 5 (a) has been redrawn in figure 5 (c) to show only the edges for the case $i = b, j = c$. If $\tau = f$ and $\rho = h$, the product

$$r_\tau^j r_\rho^i = r_f^c r_h^b = 1 \cdot 1 = 1 \tag{7}$$

shows that these two edges cross and must be included in the summation. Since $r_\tau^j r_\rho^i$ is the product of only 1's and 0's, it is equivalent to a logical and, $r_\tau^j \wedge r_\rho^i$, which is computationally more efficient on typical computers. The limits on both summations in equation 6 take into account the fact that edges drawn from all vertices in V_t that connect to the same vertex in V_{t+1} cannot cross.

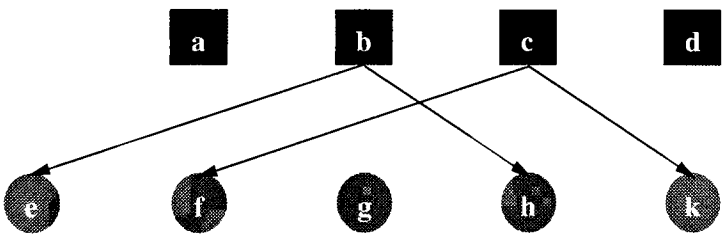
The total number of crossings $C(t)$ between the two levels t and $t + 1$ is the



(a)

	e	f	g	h	k
a	1	1	1	0	0
b	1	0	0	1	0
c	0	1	0	0	1
d	0	0	1	1	0

(b)



(c)

Figure 5: (a) A 2-level hierarchy and (b) its adjacency matrix. The hierarchy of (a) with only the edges emanating from vertices b and c shown.

sum over all pairs of vertices in t and is given by

$$C(t) = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} K(R^{(i)}, R^{(j)}) \tag{8}$$

and the total of all crossings C_{total} in the n level hierarchy is given by

$$C_{total} = \sum_{t=1}^{n-1} C(t). \tag{9}$$

1.4.3 Directed Graph Drawing

As previously indicated, the algorithm for drawing directed graphs has two segments. The purpose of the first part is to make the graph a proper hierarchy.

Previous directed graph layout algorithms have not included layout constraints. When the directed graph layout includes a constraint that requires nodes of different types to be on separate ranks, the separation is best done prior to making the graph proper. If the implementation of this form of constraint is delayed until after the hierarchy is proper, the graph is very likely not to be proper after the constraint is imposed and will have to be made proper again.

An example of this type of allowed constraint is that circle and square nodes must be on alternate levels. Figure 6 is the directed hierarchy of figure 1 with this modification. In the redrawing new edge crossings may occur.

An examination of edge crossings $(B, C), (A, G)$ or $(F, G), (D, H)$ in either

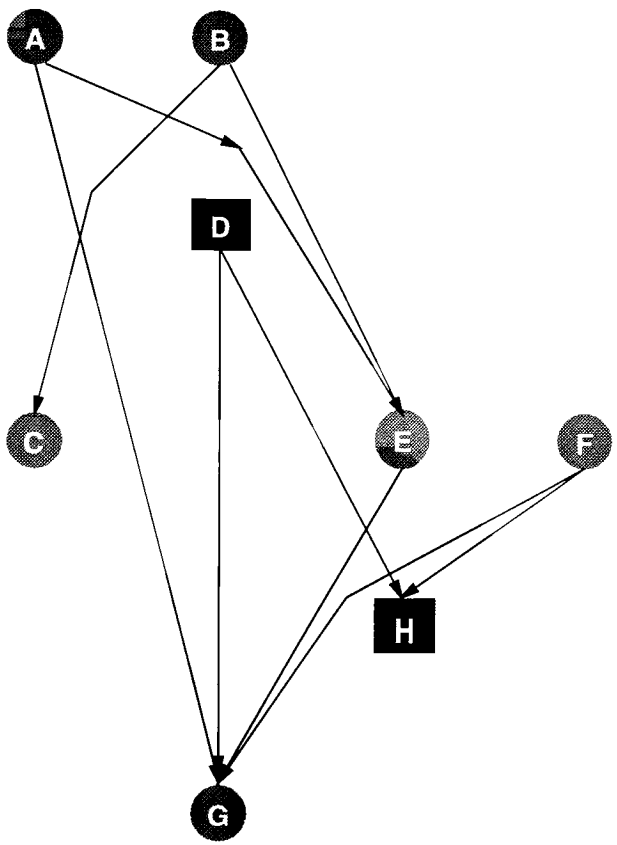


Figure 6: The hierarchy of figure 1 with the added constraint that circles and squares must be on alternate levels.

figure 1 or figure 6 suggests that the absolute x position of a node *does* make a difference in the number of edge crossings. If node C were shifted to the right closer to node E (i.e. putting it on the right of edge (A, G)), one crossing could be eliminated without changing the ordering of the nodes within a level. A similar situation involves edge (F, G).

This potential ambiguity is resolved once the hierarchy is made proper as the hierarchy of figure 6 has been in figure 7. Crossing among edges that originate among nodes on adjacent ranks cannot be affected by the absolute x position of the nodes. Some attention is required to the dummy node insertion process. The open numbered square nodes are the added dummy nodes. Separating adjacent nodes within a level may be required to put in the dummies on the same grid as the set of real nodes. Here dummies $5, 6$ and 7 needed to be squeezed in between C and E. If this is done correctly, both the number of crossings and the implicit real node to real node edges involved in each crossing are the same as before the dummy nodes were added. Now, eliminating any edge crossing requires a reordering of the nodes within a level.

An important characteristic of dummy nodes is that they act just like regular nodes in the reordering algorithm. In this implementation a dummy node may have only one edge pointing to it and may be the source of only one edge. However,

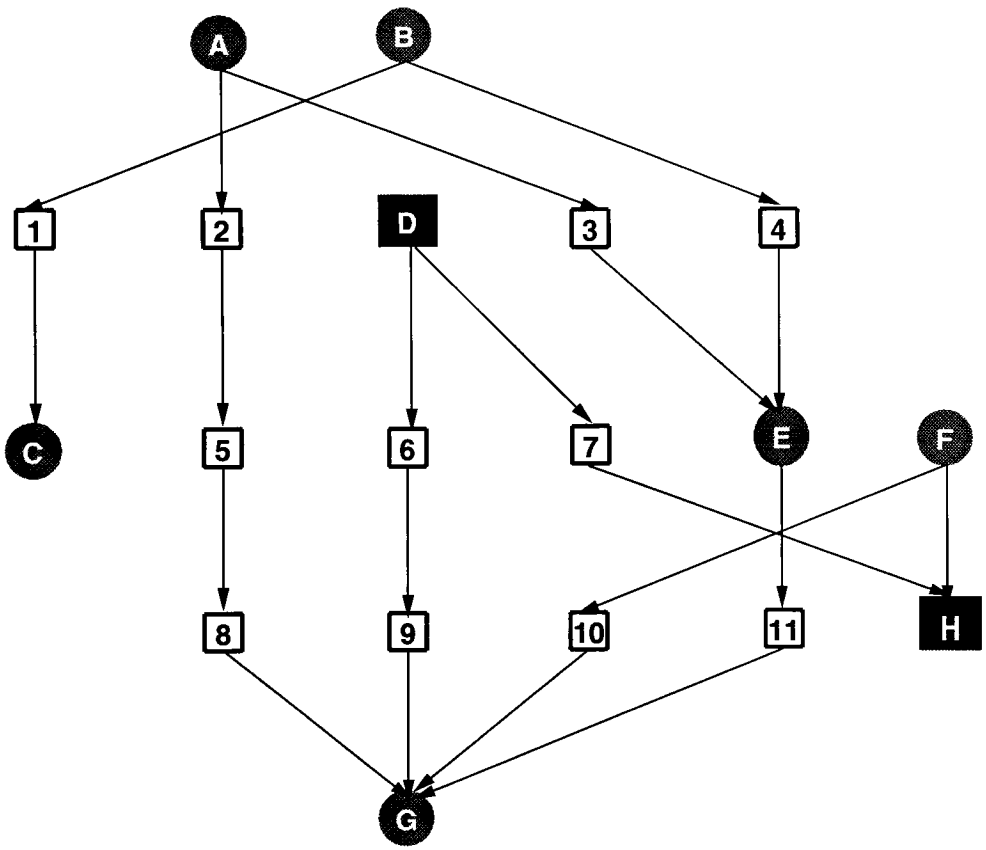


Figure 7: The hierarchy of figure 6 after dummy nodes have been added to make it a proper hierarchy.

dummy nodes could also be used to reduce graph complexity by collapsing several nodes and their connecting edges into one node; then there might be more than two edges connected to the dummy node. This collapsing into one node technique has been used as a method of deleting cycles from the graph; in this implementation cycles are allowed. Bends in the implicit real edge take place at dummy nodes; this can be used in routines that improve readability by modifying the bend shape or location after the edge crossing reduction process.

Vertex F of figure 7 illustrates an implementation dependent decision which might be considered a different type of level-based constraint which will not be considered here. F could be in the first level (as was done in the redrawing in figure 2) or alternatively in level two or even in level three. Level one is appropriate if there is a constraint which requires all nodes should be in the highest (closest to the top) rank possible. Other constraint definitions (shortest edges, lowest rank possible) result in level three being the appropriate location.

The crux of the algorithm is part two which creates a new set σ of orderings for each level. Part two naturally separates into two subparts, the rearranging of the individual level order σ_i and the iteration over all levels. Two approaches have been taken to the individual level aspect. (Individual level is really a misnomer. Since the edge crossings involve vertices from adjacent levels, *two* levels are actually

involved, but the usual practice is only to rearrange the vertices on one of them.) One approach can be considered a direct method and the other indirect.

The direct method actually monitors the crossing count. Warfield [War77] pioneered this technique using interconnection matrix manipulations but it was limited to a small number of vertices because of the computational complexity. Sugiyama, et al [STT81] expanded the method to larger hierarchies but it still had the computational expense problem. These direct methods are only used to check the results of the indirect method in small examples.

Eades and Kelly [EK86] used a different but still direct method with several variations. All are sorting schemes, (greedy insertion, greedy switching and splitting), based on minimizing the total crossing count between the two levels being adjusted. First one level is sorted and then the other is done until no further reduction is gained. Their performance tests show results similar to Sugiyama's averaging indirect method. Although they only tested two level hierarchies, their methods can be used in larger hierarchies as will be explained later.

The indirect method was developed by Sugiyama [STT81] and has since given birth to several variants[RDM+87] [Mey83] [Dav85] [GNV88] [TDBB88] [Sug87] [EK86] [ET88]. The basic concept is to keep a vertex in level i 'near' the vertices in level $i+1$ that its edges connect to. The 'nearer' connecting ver-

tices are, the less likely their edge will cross another edge. That is, in figure 7 if vertex F were nearer vertex10, the less likely edge $(F, 10)$ would cross any other edge. Using the index v_x, i.e. the x coordinate of the vertex as the measure of its position within the row, the method translates to having the x coordinate of the vertex in level i be as close as practical to the average of the x coordinates of the connected vertices in level $i + 1$. Sugiyama called this the barycentric method. Rather than using the mean, the median could also be chosen as the x coordinate. Eades [EW89] proved that using the median gives an upper limit of three times the optimal number of crossings. An upper bound was also given for using the mean, i.e. barycenter, method.

Some consideration must be given to the sorting method used in the indirect method in order to be computationally efficient. All the standard criteria for choosing a method could apply. Here it is likely that the number of nodes within a rank is going to be relatively small, i.e. on the order of tens of nodes rather than thousands per level. Except for the initial pairings of levels, each rank will be close to being ordered each time it needs to be sorted.

As part of the within-level ordering process, this thesis adds the possibility of including level order constraints. Either the x position calculation or the sorting process must maintain the constraint requirements. In the example of figure 3

where squares must be to the right of circles on the same rank, the more likely control point is the x coordinate calculation.

Whatever technique is used for the pairwise level edge crossing reduction process, it will produce a local minimum in one pass. The same is not true for the iteration over all adjacent pairs of levels. Sugiyama originally performed several iterations of alternatively going pairwise down the hierarchy and then going up the hierarchy for a fixed number of passes or until there was no crossing count reduction, whichever came first. Only a few passes, usually less than five, were the maximum attempted. On the downward phase, level i was sorted and level $i + 1$ was fixed ('down barycenters') and on the upward phase level $i + 1$ was sorted and level i was held fixed ('up barycenters'). Various alternatives, such as averaging the up and down barycenters have been used, but tests [GNV88] have not suggested that one method is far better than another. This is the most empirical part of the algorithm.

After the crossing reduction process, most directed graph algorithms add other readability improvement operations. Typically a straightening algorithm is used to take out excess kinks in each edge. Sometimes this is followed by an algorithm to shape the bend itself [GNV88].

2 Implementation

2.1 Additional Requirements

This thesis presents improvements and extensions to heuristics which minimize the number of crossings in the hierarchical graph drawing problem. As such the implementation does not need to concern itself with some of the preprocessing and postprocessing steps required of a complete directed graph drawing package.

The implementation must, however, facilitate the incorporation of extensions which will handle hierarchical graphs that have some added constraints on node placement. The general technique will be demonstrated by several examples.

One constraint type that will be demonstrated is an example of nodes that have been required to remain in their initial location as other nodes on the same rank are relocated to produce a lower crossing count.

Another constraint example is the case where one set of nodes must be before or after another set on the same rank, such as: squares must be after circles. In this example the ordering among the squares or among the circles can still be changed to reduce the total crossing count.

A third constraint type will require a method of rebuilding the hierarchy according to the constraint prescription prior to starting the crossing count reduction

process. An example of this type is a constraint that requires nodes which are circles not be on the same rank as nodes that are squares. The previous examples require a method of checking for the constraint satisfaction during the level reordering process. Here, and for other constraints of this form, a method of rebuilding the hierarchy according to the constraint prescription prior to crossing count reduction is required.

The implementation should also be robust. That is, it should be able to handle graphs of indeterminate size and should be flexible enough that it can be tailored to be useful in different applications. The initial application is expected to be in a hypertext environment which would require tracking of node and link attributes. From a programming perspective, the elegance of the implementation would be enhanced by a clean separation betweeen actions that reorganize the graph and those that display the graph representation.

2.2 Tools

2.2.1 Hardware

The software associated with this thesis has been developed on Sun workstations running SunOS and the X windowing system. Laser printers were used for

hardcopy output, either listings or figures. The equipment was interconnected over an extensive ethernet which runs a large variety of protocols, including XNS and TCP/IP.

2.2.2 Software

Since the main thrust of this work was the calculation of the node positions of a directed graph, any one of a variety of programming languages would be suitable. The choice of C++ (Sun's version of ATT Cfront 2.0, but initially 1.2) enabled the use of an object-oriented style.

A required adjunct was some method for displaying the graphs both before and after crossing reduction. While manual plotting the graphs would have given the required information, it would have rapidly become a tedious process. Publication quality printed graphs hardcopy printed also were needed for this thesis. To meet both goals, functions were written that generate PostScript descriptions of the graphs. Either the workstation's windowing system and a Postscript previewer or a printer that understands PostScript was used for displaying these graphs. Other graphs were manually drawn using the Interview's Idraw package.

2.3 Data Structures

At the very minimum, data structures are required for nodes, edges, ranks, and the adjacency matrix. If the structure for nodes and edges is to be useful in a more general system, more extensive structures are required. Provisions for attributes of both nodes and edges as well as node contents must be included.

The more generalized linked list structure for nodes and edges which was created is shown in figure 8. The added structure for attributes was used to store the value of the node shape (ie. circle or square). $Head$ in figure 8 represents the root of the linked list structure. The main linked list consists of one or more $Nodes$. Each of the $Node$ structures on the list contains a set of data for itself (x,y location, etc.) and a linked list for its $Edges$ to other $Nodes$. The $Node$ structure also has a linked list attached to itself for its attributes.

Note that each node stores only a linked list of pointers to the nodes that make up the opposite end of attached edges. The x,y location of the nodes at the other end of each edge must be evaluated via the structure of the node at the other end of the edge. This format also requires that each edge will be doubly counted in the overall structure since it will be on the edge linked list for the node at each end of the edge (with a duplicated attribute list for the edge).

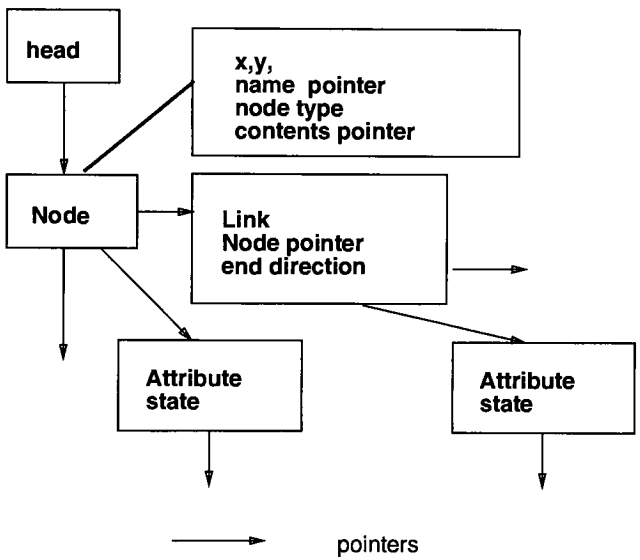


Figure 8: The node linked list structure. Arrows may either be null pointers or pointers to additional elements in their linked lists.

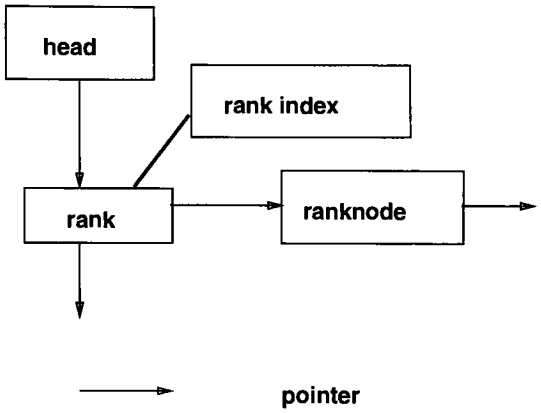


Figure 9: The rank linked list structure. Arrows may either be null pointers or pointers to additional elements in their linked lists.

The data structure for ranks is shown in figure 9. This only requires the storage of the node pointers for each node in a rank. Typically ranks are stored in ascending index order and node pointers for a rank are stored in increasing order of their ordinate.

Unlike the node and rank structures, a simple matrix is used as the adjacency matrix data structure. Although more efficient storage usage would suggest using a bitwise storage scheme, an integer matrix was used. In this situation, the required matrix is relatively small (the product of the number of nodes on two adjacent ranks), and only one at a time is needed.

2.4 Cross Count Reduction Method

The first choice in deciding on the crossing count reduction method is in choosing between a direct or indirect approach. Eades [EK86, figure 3] has shown that for low edge densities, an averaging (indirect) technique produces fewer crossings. However, at higher densities the results are essentially the same as the greedy switching (direct) method. However, if speed is important, the analysis shows that the greedy switching method comes out a little faster.

Another consideration is the ease of implementation of constraints. If a greedy switching technique is used, before every switch the constraint condition may be easily tested by checking if the swap is allowed. In the averaging (indirect) methods, constraint satisfaction is a little more complicated.

Unlike the Eades studies on simple n= 2 layered networks, a direct method for a higher order $n-$level hierarchy requires more than a single adjacency matrix calculation. For n>2 the adjacency matrix must be recomputed each time a pair of levels is chosen for crossing count reduction. Additionally, for each proposed swap of nodes in the greedy switching process, the crossing count must be calculated in order to check if the swap would result in a reduced crossing count.

But the simple logical *and* test which reduces the computational burden in

the adjacency calculation and the need for including constraints indicates greedy switching as the method of choice. Greedy switching can be considered analogous to a bubble sort. If the swapping of two adjacent nodes reduces the crossing count (analogous to seeing if two items should be swapped to improve the sorting order), they are swapped. The rank data structure implementation requires a simple node pointer and node ordinate value interchange to accomplish this.

However, the Eades form of simple greedy switching fails on a pathological case, which can be seen using the configuration in figure 7. Assume the particular ordering of nodes had been formed during a crossing reduction process that had started with levels paired starting at the bottom (here level 5) with level above (4) held fixed. This would be considered pairing by going *back* facing *up*. With the appropriate choice of starting level, there are four combinations, *forward/down*, *forward/up*, *back/up* and *back/down*.

At the instance during this sequence illustrated by figure 7, nodes within level 2 are going to be swapped in order to minimize the crossings between level 1 (nodes A and B) and level 2 (nodes $1, 2, D, 3$ and 4). All the nodes in level 1 (and all nodes in levels 3 or higher) are held fixed. Changes in the crossing count between nodes in level 2 and level 3 as a result of the node exchanges within level 2 are not considered.

The problem centers around node D in level 2. Ideally node 1 (ignoring where node C should go) should be between nodes 3 and 4 or after 4. Swapping nodes 1 and 2 reduces the level 1-2 crossing count by one so the swap will be carried out. But when the time comes to test swapping adjacent nodes 1 and D, there is no crossing count reduction and 1 becomes pinned to the left of D. This cannot happen in the Eades 2-level examples because each node has at least one edge, otherwise there is no use for it in the example!

An increase in the complexity of the greedy switch resolves the problem. All possible pairs of nodes on a level are tested. The cost is, of course, an increase in computing time. Note that in both the simple greedy switch and the modified greedy switch implemented here, more than one pass is required to percolate the nodes to the positions which reflect the minimum crossing count. However, the modified greedy switch represents a significant improvement over the worst cases of the simple greedy switch. In the worse case for the simple greedy switch, like the bubble sort, a node at the opposite end from its desired position is moved by swapping on place at a time by adjacent node swapping. In the modified greedy switch, a node can move multiple positions in one swap, possibly taking only one swap to get to where it 'belongs'.

Note, however, that this is a swapping, not an insertion algorithm. The position

of two nodes at a time are affected. Node 1 cannot get to its 'desired' place by a simple swap. Checking a complete set of all combinations of nodes on one level is considered one pass. Since it is not known how many passes are required until a pair of levels produce a crossing count minimum, passes are repeated until a pass fails to produce a reduction in the crossing count. Specifically not considered are the effects on adjacent crossing counts and the possibility that swapping nodes which produce no change in crossing count might lead to an eventual lower count for the pair of levels.

2.5 Hierarchy Pass Strategy

As noted in the previous discussion, there are four basic methods of doing a set of level crossing reductions for an entire hierarchy, i.e. a hierarchy pass (HP). Various combinations have been suggested [STT81, RDM+87, Car80] and an estimate of the number of HP cycles to use given, but no analysis has been proposed. Among the suggested strategies are repeated HP down and alternating down and up. An indication of how many HPs are required using these strategies can be seen by looking at figures 10-12.

Figure 10(a) is the example hierarchy before any attempts at crossing count

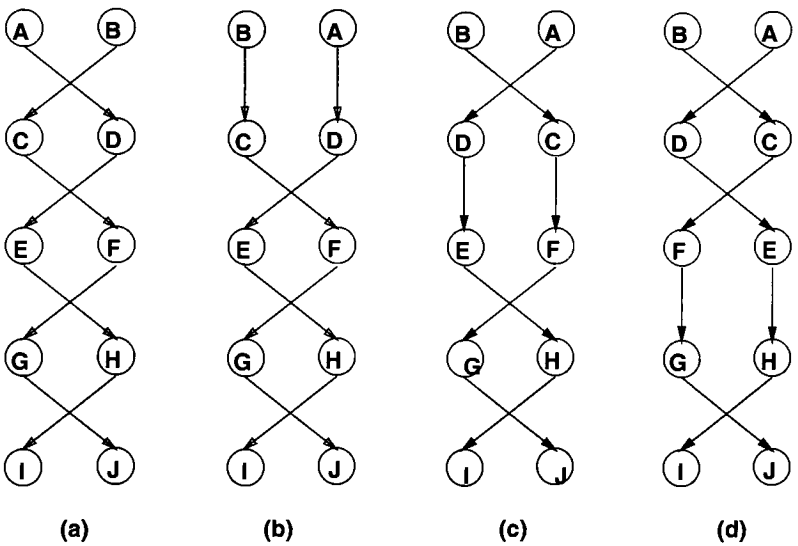


Figure 10: (a) Before any hierarchy passes and level reductions. (b) After first $forward/down$ on level 1. (c) Another $forward/down$ on level 2. (d) Another $forward/down$ on level 3.

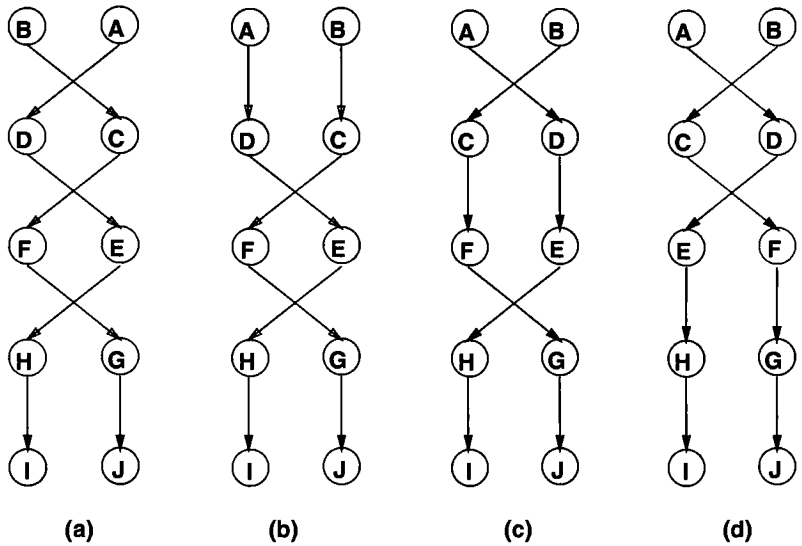


Figure 11: (a) After the $forward/down$ pass on level 4. (b) After the start of the second HP with a $forward/down$ pass on level 1. (c) After the $forward/down$ pass on level 2 of the second HP. (d)After the $forward/down$ pass on level 3 of the second HP. Another $forward/down$ on level 3.

reduction. At every point we will assume that the level crossing algorithm is perfect and reduces the level count from 1 to 0. Successive $forward/down$ level passes on each level are shown in figures 10(b) through 11(a). Note that after a complete HP only a single crossing has been eliminated; the effect of every level pass after the first has just been to move the uncrossed point down the hierarchy.

The results of the individual level passes for the second set of $forward/down$ steps that complete the second HP are shown in figures 11(b)-(d). Note that a

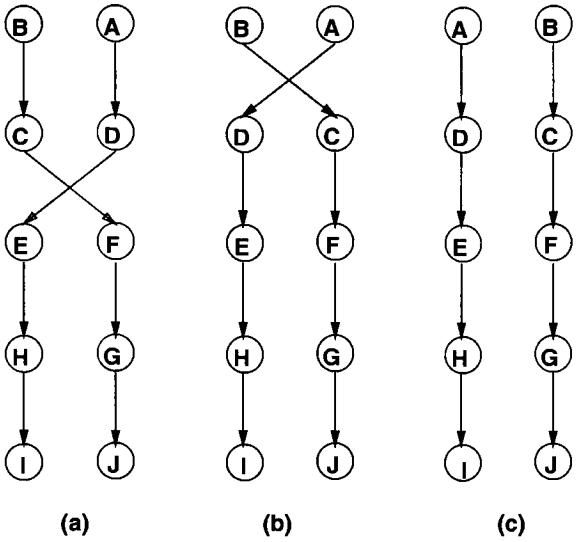


Figure 12: (a) The first $forward/down$ level pass of the third HP. Finally, after the completion of the fourth HP, a zero crossing count.

level pass on level 4 is not needed on this HP. It is tempting to assume that a decreasing number of level passes are required on successive HPs but there are counter examples which show this assumption to be invalid.

Only one level pass of the third HP changes the hierarchy at all. The status after the third HP is complete is shown in figure 12(a). Only the second level pass of the fourth HP changes the hierarchy as the remaining crossing is moved up as seen in figure 12(b). A fifth HP, the same number as the number of ranks, is required to reduce the overall crossing count to zero. Finally figure 12(c) shows

the hierarchy without any crossings.

Surprisingly, for this example, alternating $forward/down$ and $back/up$ HPs can require more HPs than just successive $forward/down$ HPs. As indicated earlier, the completion of a $forward/down$ HP is shown in figure 11(a). Now a $back/up$ HP at this point is topologically the same as being at figure 10(b), except for the node labels. Completing the $back/up$ HP would result in figure 11(a) which is the same as the start of this HP except upside down. Then the third HP, a $forward/down$ set would begin at the stage of figure 10(b) again! Successive HP under the alternating scheme on this example fail to give the known minimum solution yielding only a slight improvement over the original crossing count in reducing it only by 1!

An analysis of the problem indicates that $forward/up$ and $back/down$ HPs are surprisingly more efficient in reaching a relative minimum total crossing count. A single $forward/up$ or $back/down$ HP *must* yield a local minimum. Although using either of these two methods is likely to increase the crossing count in the level pair that follows each step in the HP, a second HP using the same method produces no changes! A way to look at this is to think of this as a tube of toothpaste being rolled up from the bottom, pushing the knot of crossings toward the edge at the open end of the tube. Visualize this by looking at figure 10(a). A $forward/up$

HP begins at level 2, adjusting the position of C and D in order to minimize the crossings with level 1 nodes A and B. (A $forward/down$ pass at level 1 could have been done to take advantage of adjusting level 1 nodes also).

Rearranging the level 2 nodes in a hierarchy with more nodes at each level is likely to have made the crossing count between level 2 and 3 worse, although in this case it makes things better. But once the $forward/up$ pass is done at level two, the next step in the HP will not affect the relationship between level 1 and level 2. Note that this is different from the $forward/down$ HP technique described earlier. Another $forward/up$ complete HP will find level 1 and level 2 unchanged from the first HP and any improvement that could be done has been done already! This is not to state that no further improvements are possible; a $down$ facing HP might possibly find an even lower relative minimum, but since it will affect changes made during the previous HP things could get worse. Given that there are two types of $down$ facing HPs, once again, the one that squeezes complexity along the hierarchy ahead of it, i.e. $back/down$ might be desirable.

In order to more fully explore the possible HP sequences, the software has been implemented that does any of the possible combinations discussed above plus any individual pair of adjacent levels facing in either direction. One interesting possible HP sequence is the equivalent of squeezing the toothpaste tube in the middle; the

software also has this case.

2.6 C++ Classes

The desire to separate the graph manipulations from the node storage and manipulations resulted in a design with two classes, `NodeList` and `Brows`. `NodeList` contains the node and link creation functions and anything that operates on nodes or links. `Brows` (named earlier as a class that started life as a browser construction) does the higher level graph oriented functions as loading (graph input), adding dummy nodes, counting crossings, etc. `Brows` is not derived from `NodeList` although this clearly is an alternative formulation. At the time these classes were being implemented, multiple inheritance was not available in the 1.2 version of Cfront, the AT&T C++ translator. Since both node functions and displaying functions were desired in `Brows`, it was unclear which type of base class should be developed. Instead, the technique of using a private pointer to an instance of the 'base' class is used. If necessary, this allows a class to have multiple instances of a base class associated with it.

2.7 Data Generation

Test hierarchical graph data has been generated by two methods. The first is manually writing an input file. Only small hierarchies (typically less than a dozen total nodes on all levels) are practical to implement by hand. However, this was the technique of choice to debug the software as each function was developed. Test cases were constructed that check rank ordering, dummy node generation and insertion, etc.

Performance and analysis of the system required larger hierarchies and random placements. In these instances a separate data generation program, `dater`, produced the required input files. Parameters included the number of ranks desired, nodes per rank and density of links and the number of test cases desired. Shape functions were built in so that hierarchies with the same number of nodes on each rank, the same number of nodes as the rank index (i.e. linear increasing), etc. could be generated.

2.8 System Organization

An overall perspective of the system organization is shown in figure 13. Test cases are output to a file which then serves as the input for the crossing reduction and

display file generation program. If there are constraints they are defined as separate modules which are referenced by the main program. The graphs associated with a particular test can be viewed via the PostScript display previewer or via hardcopy output. The main program will also output information about the number of crossings before and after crossing reduction which can be used in the analysis of performance plotting.

2.9 Algorithm Description

A high level view of the algorithm developed for edge crossing reduction is elegant but deceptively simple:

```
load the graph
if necessary
   separate by vertical constraint
make rank order list
if necessary
   reorder rank by constraint condition
add any required dummy nodes
as per desired hierarchical pass strategy
   reduce the crossing count each required level pair
```

At any time during the process a PostScript display of the graph may be generated. After the graph is proper, ie, any needed dummy nodes have been added, a count of the edges and crossing may be made.

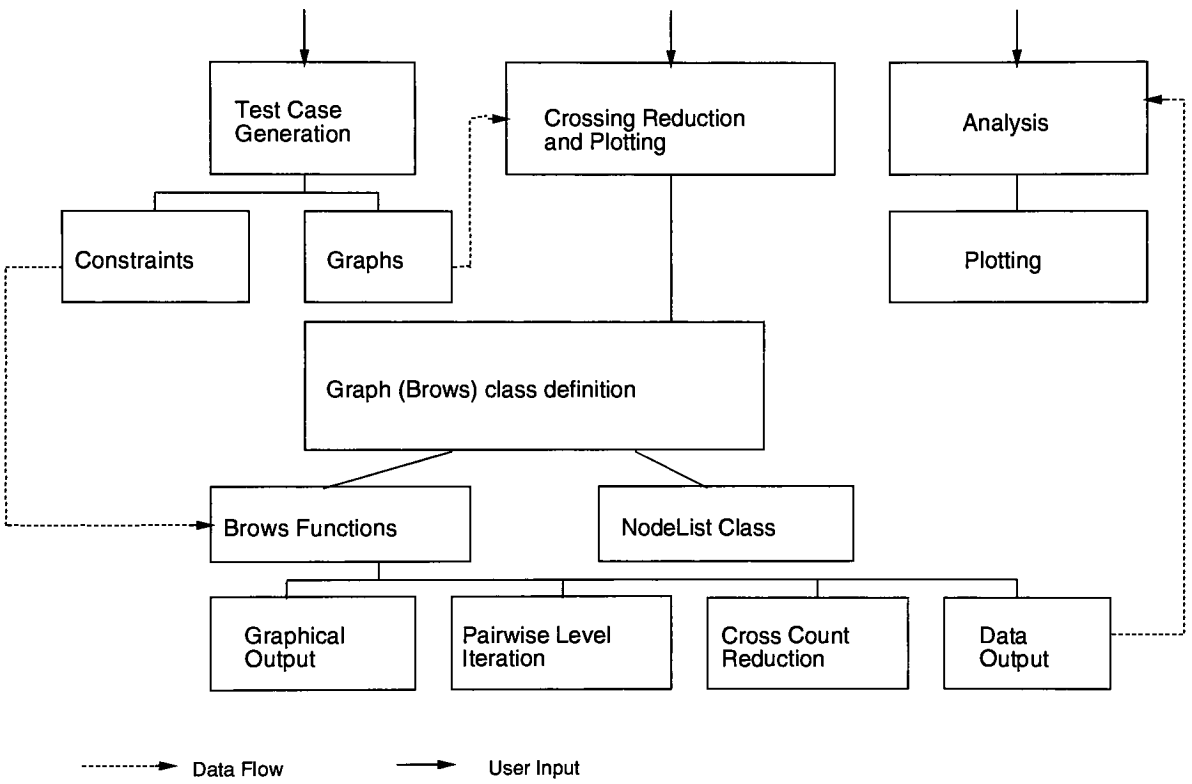


Figure 13: An overview of the programs, their major components and the data flow.

The most interesting parts of the algorithm are included in the level pair reduction process and to a much lesser extent, the dummy insertion process.

The dummy insertion process proceeds as follows:

```
for each rank starting at the top
   for each node in the rank
      for each attached edge in a greater rank
         if the rank difference is greater than one
            insert a dummy
```

The dummy rank is known but the x location needs to be decided. The procedure for doing this is:

```
if next rank is empty
   put it directly below this node
else
   find closest place to the left below here
   find closest place to the right below here
   put it nearest to below here
```

Actual edge crossing reduction takes place in the reduce layer function. Given the level and the level either above or below as appropriate to the hierarchical pass (HP) strategy:

```
create an adjacency matrix
count crossings
until new crossing count is the same as old
   for all pairs of nodes in this level
```

```
if ok to swap based on constraints
   make a temporary adjacency matrix swap
   count crossings
   if edge count is less
      make swap permanent
      reset running crossing count
```

Since an edge count for an entire level pair is required for each swap, the computational costs could be high. The replacement of the multiplication of the sign computation in equation 4 by a logical *and* as indicated following equation 7 probably is a significant increase in computational efficiency.

Although the analysis has broken down the algorithm in what looks like small simple chunks, implementation requires several thousand lines of code and comments.

3 Functional Specification

3.1 Functions Performed

Given an ASCII text list of nodes and edges, the program written for this thesis inputs the data and then after processing can produce output PostScript files representing plots of the graph as well as statistics such as initial and final crossing count, number of edges, etc. The files can either be printed on a PostScript printer

or used as input to a PostScript display previewer. The program can also be set to produce an output listing of nodes, edges and coordinates. Several different types of constraints have been built into the software and additional ones may be easily added via a recompilation of a single module. Thereafter the choice of constraints can be set from a main program parameter which could be input from `cin` (C++ `stdin`) or a file as appropriate.

Test files can be generated using the `dater` directed graph generating program. Input required for `dater` includes number of ranks, maximum nodes per rank, nodes per rank (fixed, linear increasing ,etc.), edge density factor and number of directed graphs desired.

3.2 Limitations and Restrictions

The input is assumed to be a valid hierarchy. Unlike other implementations, cycles are permitted. The software has no inherent limitation on the number of nodes and edges aside from the memory and cpu speed limitations. Typical processing times on a Sparcstation 1 (approximately 12 MIPS) are only a few seconds for graphs of less than 100 nodes and several hundred edges even if the HP strategy requires several HPs.

3.3 User Inputs

The graph input is a file of node names, initial positions and (typically) a node type and a shape parameter which can be used in defining constraints. Constraints, HP strategies and other parameters and/or operations are controlled by class methods and variables specified in the main program. The variables may be set by using `cin` and `cout` C++ functions in console or file command operation if desired. Operation of the program begins by executing the program and then responding with the file name, etc. to queries posed by the program.

3.4 User Outputs

The program displays are a function of the class calls that form the main function of the program. In addition to the PostScript output file data, test data and crossing count numbers may be output as appropriate. The PostScript output file can then be sent to a PostScript printer for hardcopies of the graphical output. Alternatively, the same file can be the input for a PostScript display viewer such as Pageview.

4 Proof of Performance

4.1 Basic Directed Hierarchy

A proper directed hierarchy which has not been subject to crossing count reduction is shown in figure 21. The same hierarchy after crossing count reduction is shown in figure 22. A more detailed discussion of crossing reduction in this directed graph follows in section 5.

In order to demonstrate the insertion of dummy nodes and the application of constraints to the crossing reduction problem, a more complex directed graph than figure 21 is required. The directed graph of figure 14 has a mix of circle and square nodes where the square nodes were randomly chosen to be about 25 percent of the total number of nodes. The graph has a total of 35 nodes (eight of which are squares) arranged in five ranks of seven nodes each. A .2 edge density factor resulted in 52 edges and 133 crossings initially.

4.2 Dummy Node Generation

The crossing reduction method requires that the graph be proper, ie. that any link be between nodes on adjacent levels. If the links are not between adjacent levels,

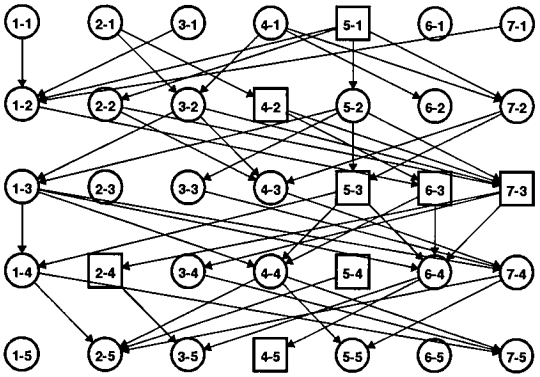


Figure 14: A directed graph with two types of nodes, circles and squares, used as the starting point for demonstrating dummy nodes and constraint problems.

additional nodes called dummy nodes with appropriate link pairs are added on the intermediate levels.

One example of the need for dummy nodes is shown in figure 15. As part of a constraint example where the circles and squares must be on separate ranks, the initial directed graph of figure 14 has been expanded. Now the circles and squares are on separate ranks, as required, but the directed graph is no longer proper.

Dummy nodes, designated by triangles, have been added to the graph in figure 16. Although dummy nodes may be added to the directed graph so that the revised graph will be both proper and homeomorphic, here a different approach has been taken. The dummy nodes needed to make the graph proper are placed in the nearest empty location within the appropriate rank on the implicit grid of the graph representation. This is highly likely to increase the number of edge crossings initially. No attempt is made to make the revised graph homeomorphic with the initial graph. Regardless of how the dummy nodes are inserted, homeomorphicity will disappear as nodes are reordered to reduce the edge crossing count.

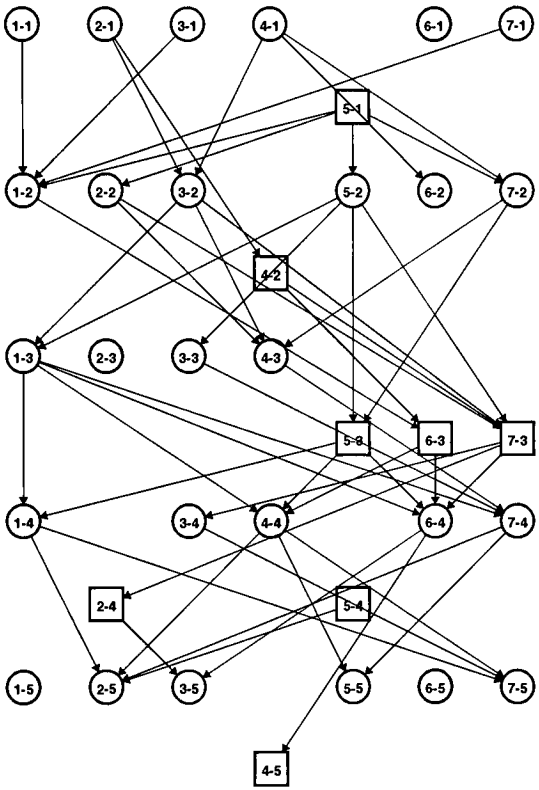


Figure 15: The directed graph of figure 14 after squares are separated onto different ranks from circles.

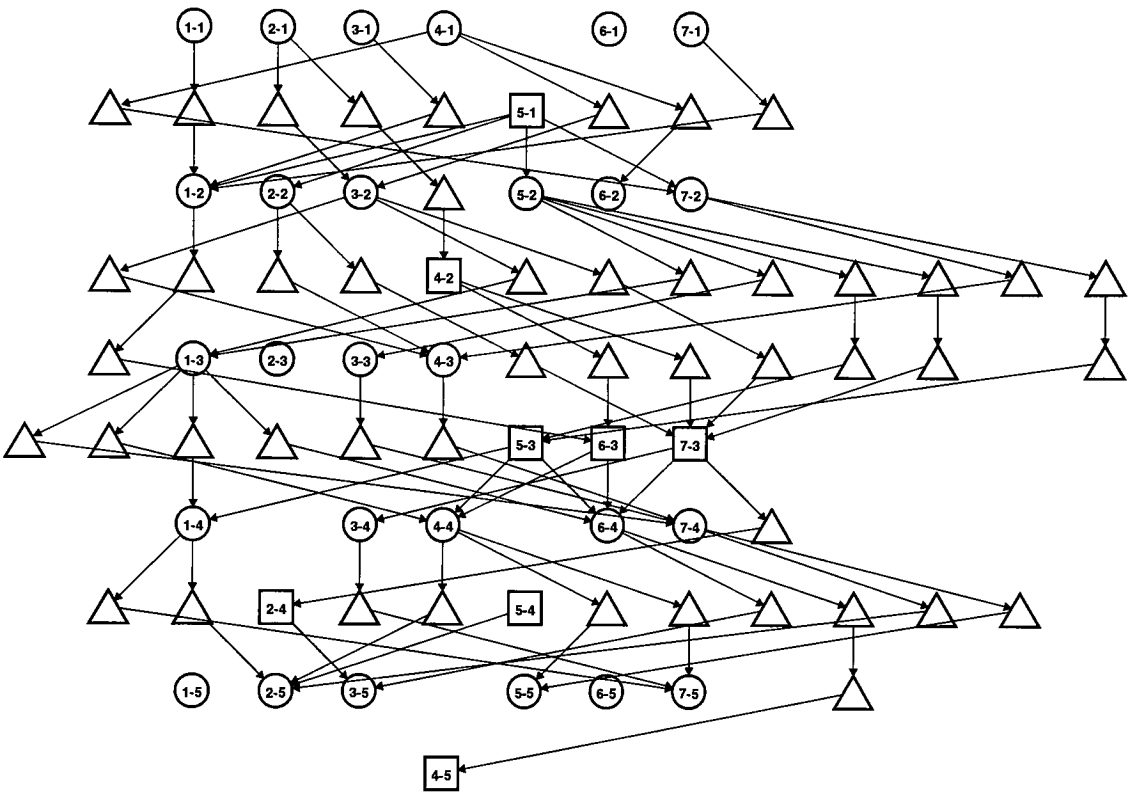


Figure 16: The directed graph of figure 14 after separation and then dummies (triangles) added to make it a proper directed graph again.

4.3 Directed Graphs with Constraints

4.3.1 Circles and Squares on Different Levels

The initial directed graph used to demonstrate constrained edge crossing reduction (where circles and squares are required to be on different ranks) is shown in figure 14. Intermediate steps are shown in figures 15 and 16. Only two HP passes of the recommended type ('*multiple*' to be discussed in section 2.5) were required to give the results shown in figure 17.

After the squares and circles are put on separate ranks and dummy nodes are inserted, the number of crossings has grown to 147. With the crossing reduction process complete, only 51 remain for a crossing reduction efficiency, ie. crossings eliminated divided by 147, of .65. The details of this edge crossing reduction efficiency metric are discussed in section 5.1.

As previously noted, further processing, outside the scope of this work, could now be applied to the graph in order to simplify the figure. Especially useful here would be minimizing the bends along edges as the triangles (ie. dummy nodes) were replaced with simple point vertices. The major question to understand is how to slide the nodes along each rank, minimizing bends yet maintaining a compact structure. The result, however, would not change the edge crossing count.

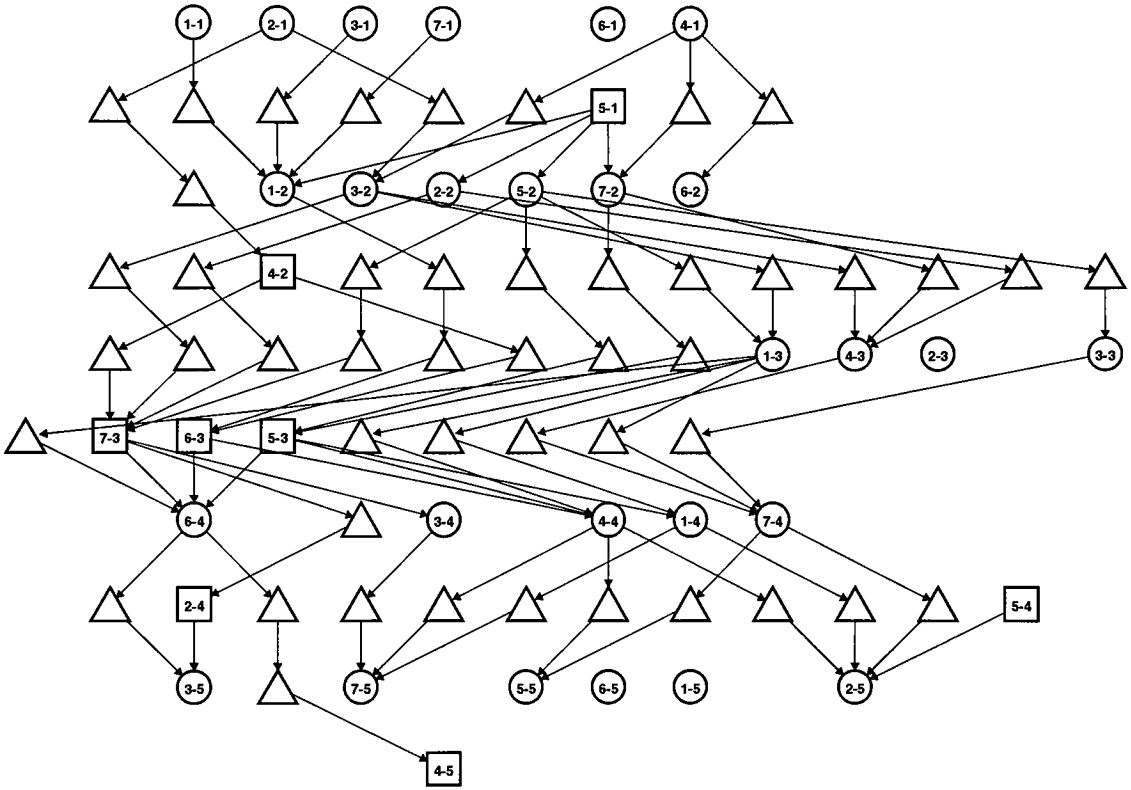


Figure 17: Completed edge crossing reduction of the directed graph of figure 14 under the constraint that circles and squares be on different ranks.

4.3.2 Squares before Circles on the Same Level

Figure 18 shows the result of constraining the graph in figure 14 to have the squares before the circles on the same level. Surprisely, in this particular example, reordering each rank so the squares come before the circles reduces the initial crossing count from 133 to 100. Since the graph is originally proper, no dummies are added as reordering nodes on each rank does not affect this graph property. The constraint is defined so that if dummy nodes were required, they could be interspersed within both square and circle nodes on each rank.

Figure 19 shows the directed graph with the squares first constraint after the crossing reduction process. Only a .42 efficiency was achieved (see section 5.1), reflective of the decreased flexibility in deciding node placement due to the constraint.

4.3.3 Nodes that have Fixed Locations

A common constaint is that some of the nodes must remain in fixed positions. For example, a graph layout might require that certain nodes be fixed in the middle of the graph in order to show the main flow of a set of tasks. Holding the square nodes in figure 14 fixed in position demonstrates this constraint form. Since the

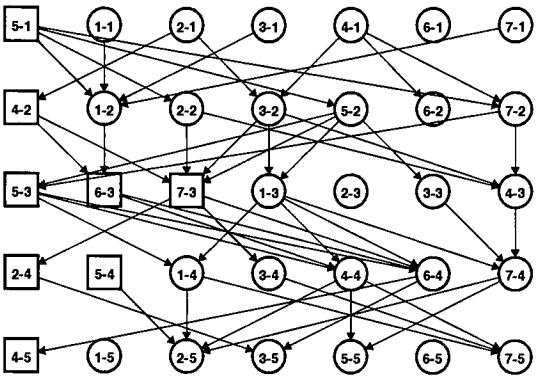


Figure 18: The squares first directed graph after rearrangement so that squares are to the left of circles.

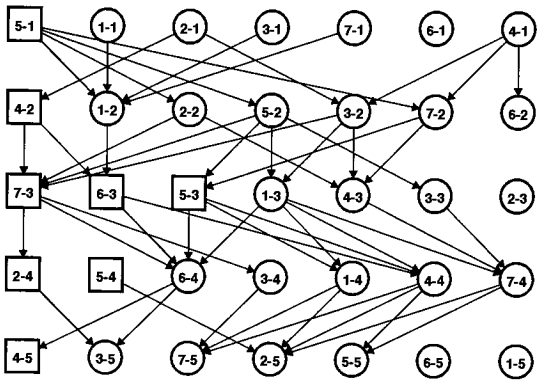


Figure 19: The squares first directed graph after crossing reduction is complete.

graph is initially proper, there is no need to add dummy nodes before reducing the number of edge crossings. The graph after the crossing count is minimized is shown in figure 20. The initial count of 133 crossings has been reduced here to just 61 even with the fixed square nodes constraint.

A careful examination of the final graph shown in figure 20, however, shows that the crossing count can be reduced by at least seven more crossings! If node 1-1 were at the right end of the first rank, three crossings would be eliminated and if node 6-2 were at the left edge of the second rank, four more crossings could be avoided. These changes look like they have been missed due to the directional properties of the algorithm which are discussed in the next section.

5 Analysis

In an earlier section it was shown for a pathological case that some HPs became effectively useless in reducing crossing counts. But 'real world' instances may not be the same. Moreover, other complexities may occur which will obviate proposed preferred HP strategies. Two techniques will be used prior to making a formal recommendation for an HP. One will be a detailed performance study of a single graph and the other will look at graphs with different node densities.

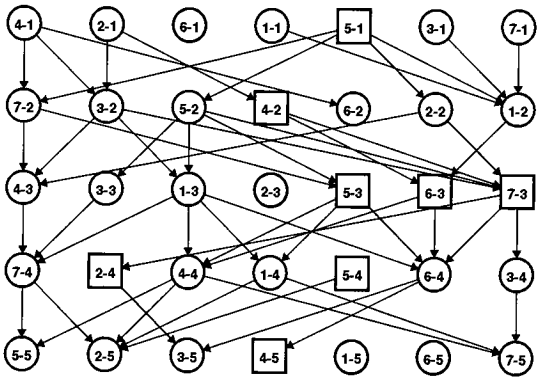


Figure 20: The keep squares fixed directed graph after crossing reduction is complete.

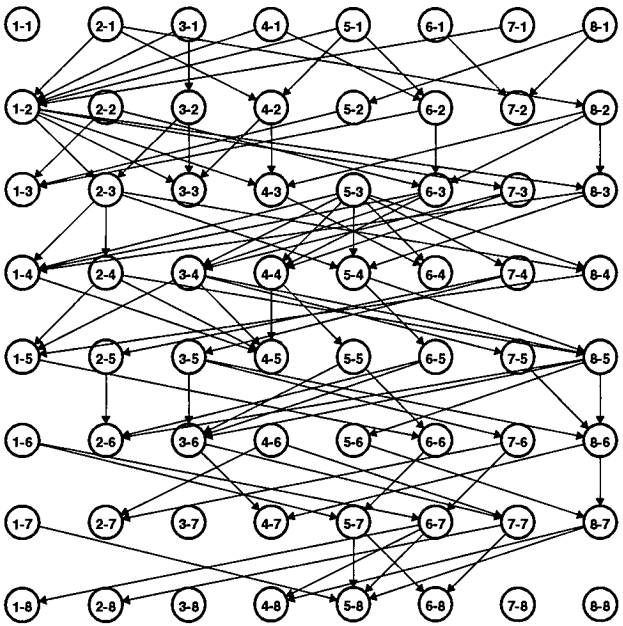


Figure 21: The base test case with 64 nodes, 101 edges and 282 crossings as a result of a .2 random link density.

5.1 A Crossing Reduction Efficiency Metric

A metric is needed to evaluate different HP strategies. Although using the number of crossings eliminated would suffice as the metric for a single graph analysis, the raw number is not meaningful for comparisons that include multiple graphs. If the absolute minimum number of crossings for each graph were known or calculable, it could be used to normalize the results for each graph. However, this is not generally possible. As noted earlier, finding the minimum number of edge crossings of a directed graph is an NP-complete problem [EMW85].

An alternative is to normalize using the inital crossing count, ie. an edge crossing reduction efficiency metric. As a practical metric, this will avoid the raw score scaling problem allowing comparisons among similar types of graphs. But a metric alone is not sufficient. It must be used with an appropriate set of test graphs. As did Eades and Kelly [EK86], a sequence of graphs with increasing edge density was used for each of the graph types. In order to show trends rather than possibly abnormal examples, a set of graphs was used for each density.

The validity of the metric is still open to question until it is compared to a metric based on the absolute mininmum number of crossings for each graph. If T is the total number of crossings in a graph, C the absolute minimum number of

crossings in the graph, and E the number of crossings eliminated, the relationship between T, C and E is always

$$0 \leq C \leq T - E \leq T \tag{10}$$

When $E \simeq 0$, ie. few crossings have been eliminated, $T - E \simeq T$. The relationship between E and C is unknown. The validity of the metric is still indeterminate. However, when $E \simeq T, T - E \simeq 0$ and therefore $C \simeq 0$ must also be true which implies $E \simeq C$. Therefore, when the crossing efficiency is close to 1.0, the metric approximates one which uses the absolute number of crossings for normalization.

An example of using this metric is shown in figure 23. Since the results of testing a set of graphs are plotted, rather than the result for a single test, the metric becomes the average crossing efficiency. The efficiency is higher for graphs with few edge crossings (ie. low edge densities) and becomes 1.0 when all crossings are eliminated.

If the graph initially had the minimum number of crossings possible, then there would be no further improvement and the crossing efficiency reduction would be 0.0. The metric measures effectiveness in eliminating crossings, ie. the quality of

the HP strategy, not the quality of the graph.

The initial crossing count always is calculated after dummy nodes, if any, are added.

5.2 Comparison of Hierarchical Pass Strategies

5.2.1 A Single Example

In order to illustrate the differences between HP strategies, a typical hierarchical graph, generated by the `dater` program, was used. Figure 21 shows the initial hierarchy. The `dater` parameters for this example included eight nodes on a rank and a total of eight ranks. Since each node could potentially have an edge to each of the nodes in the rank below it, there are potentially $8 \times 8 \times 7 = 448$ edges for this hierarchy. Using equations 5 - 9, such a matrix would have 5488 crossings. Here a .2 random link density produced 101 edges and 282 crossings.

A single $forward/down$ HP reduced the crossing count only to 271. Even after the sixth successive $forward/down$ HP the crossing count had only gone down to 159. More HPs of this form did not reduce the count any further.

Surprisingly, successive $back/up$ HPs resulted in a minimum of only 117 crossings after only six passes. The resulting graph is shown in figure 22. This result is

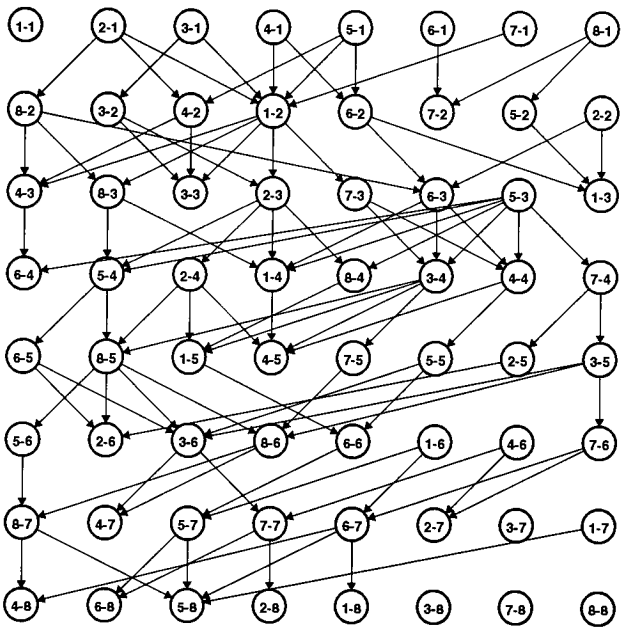


Figure 22: The base test case after six $back/up$ HPs with only 117 edge crossings.

surprising because this type of HP has the same problem as the $forward/down$ HP strategy. Both disturb the node order that the previous level pass had used to reduce its crossing count. Unlike the pathological case described earlier, this time this strategy does work. As different HP strategies are looked at for this test case, the result from the $back/up$ successive HPs strategy looks quite good but it requires six HPs to reach a minimum.

As indicated in section 2.4 there are only four basic level pass methods $forward/down$, $forward/up$, $back/up$ and $back/down$. Two HP strategies that might yield better or equal results quicker were hinted at in section 2.5. Both are based on the toothpaste tube squeezing analogy. The first additional strategy is the method $middle$, which is a single pass algorithm. The second strategy, called $multiple$, is the best combination of the four basic single level pass methods.

The $middle$ algorithm starts at an appropriate middle rank and squeezes crossings off the top of the directed graph by going $forward/up$ and off the bottom by going $back/down$. This is a single pass HP strategy. As discussed earlier, repeated $forward/up$ or $back/down$ passes produce no changes over the first $forward/up$ or $back/down$ pass.

The $multiple$ strategy recognizes that multiple HPs are likely to give superior results than any single pass of the four basic types. It avoids the pathologi-

cal problems of the repeated $forward/down$ or the equivalent repeated $back/up$ HP strategy. This new algorithm also recognizes that several of the alternating $forward$ and $back$ strategies incorporated in previous work are likely to be non-optimal combinations due to the weaknesses with $forward/down$ and $back/up$ passes.

Starting with a $forward/down$ level pass just between level 1 and level 2, the multiple algorthim does an inital squeeze of edge crossings off the top of the graph. This is followed by a full set of $forward/up$ HPs down through the graph starting at level 2. Next a full $back/down$ HP completes the first cycle. Further cycles repeat the $forward/up$ and $back/down$ parts of the first cycle.

Comparison of the four basic HP strategies plus repeated $forward/down$, simple alternating (ie. $forward/down$ and $back/up$) and the two new methods, $middle$ and $multiple$ on a single directed graph did not indicate a clearly preferred single strategy. With this example, for instance, the repeated $back/up$ worked much better than expected. What is needed is a much larger test over a number of directed graphs of a variety of types.

5.2.2 Sets of Test Directed Graphs

Six HP strategies were tested on a variety of directed graph types. The six strategies were the previously used repeated $forward/down$, repeated $back/up$, alternating and the new $middle$ and $multiple$ HP strategies as well as a single cycle of the $multiple$ HP strategy. Four directed graph types were tested. The first was a simple eight node $\times$ twelve rank directed graph with edge density factors ranging from near zero (.01 or .05) to .35 in .05 step increments. At each edge density, ten different sets of randomly produced edges were tested for the collection of 96 nodes. As a result, data was accumulated from over 1500 directed graph edge crossing reduction tests. In order to concentrate on the different HP strategies and keep the individual graphs down to a tractable size, the random edges were only drawn between adjacent levels; therefore the generated graphs were all initially proper.

The second directed graph type was tree-like with one node at the top and linearly increasing to eight nodes in the twelfth rank. Once again, sets of graphs of varying edge densities were tested. The parameters used were similar to those used with the first graph type.

The third and fourth directed graph types included constraints. Both of these

graph types used the eight node × twelve rank size used in the first two graph types. Similarly, the same edge density range and number of test graphs for each density were generated. However, for these two types, the test graphs had about a quarter of the nodes defined as squares and the rest defined as circles. For the type three graph, the constraint required that squares be fixed in place. For the type four graph, the constraint required that squares had to come to the left of (ie. before) circles on each rank.

For each test, where the HP strategies included the possibility of multiple cycles, the strategy was allowed to cycle up to 20 times or until the cycle failed to reduce the crossing count. The process was monitored for average edge crossing reduction efficiency and average number of cycles required. Although statistics were not kept, the number of level passes and maximum cycles required to terminated were observed.

The closest results among the HP strategies tested were on the first directed graph type. Figure 23 plots the average crossing reduction efficiency for the sets of graphs tested at different edge densities. All four graph types had similar relative orderings of the tested HP strategies as that of graph type one. Only in the single case of the type one graph at extremely low edge density are all the HP strategies close in efficiency. When there are very few crossings and no constraints, any

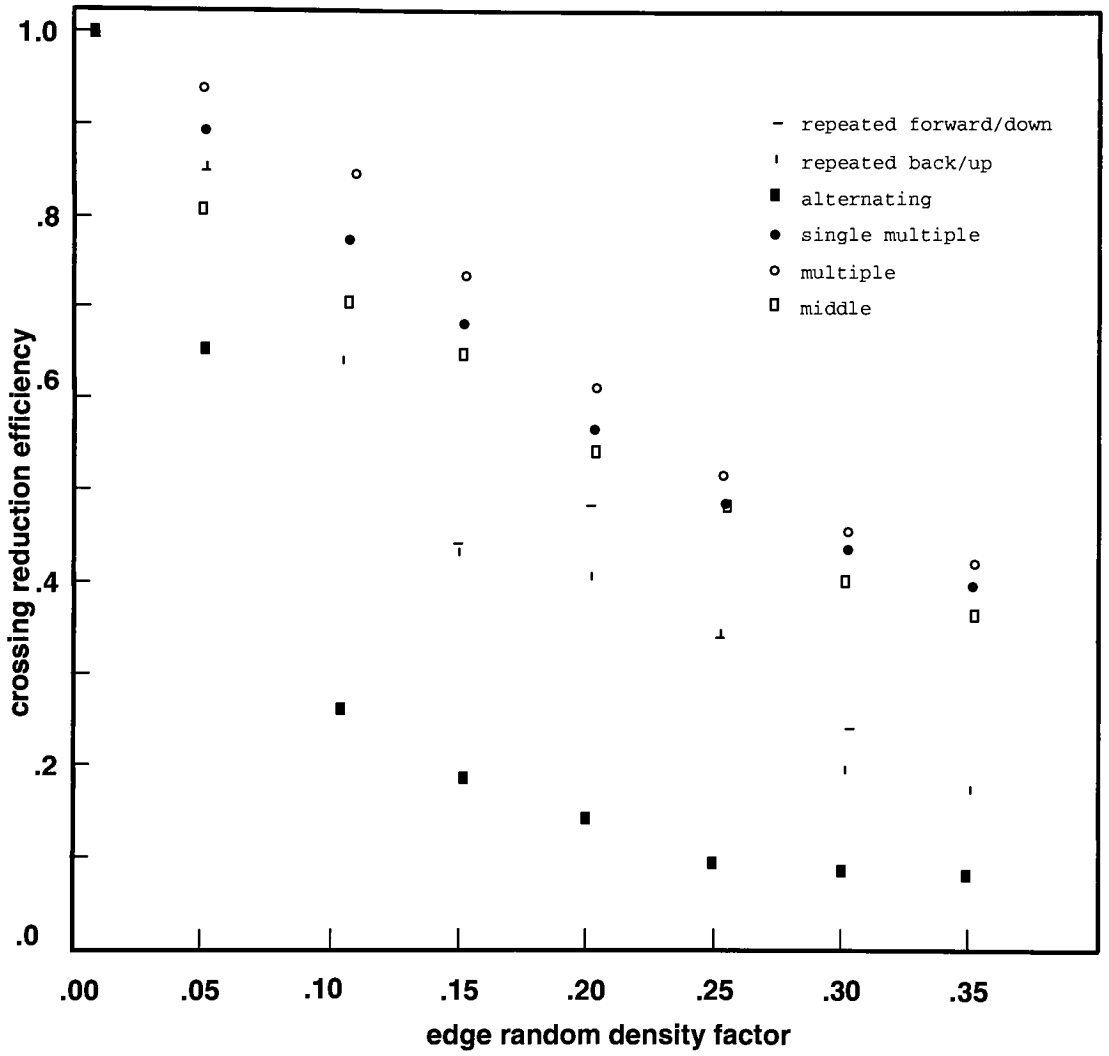


Figure 23: Edge crossing reduction efficiency results for the eight node × 12 rank graph type without any constraints.

method will work!

The alternating strategy always is dramatically worse than the other tested HP strategies even though it seems to be the most common method in use in the literature references. Not surprisingly, $forward/down$ and $back/up$ on average give very similar results. The one exception is in the type two graph where $forward/down$ is distinctively better, especially at higher edge densities. Even though $forward/down$ does not 'push' the crossing complexities ahead of its level passes as well as do the new strategies, it is better able to take advantage of the top to bottom asymmetry of the graph. Of course, one expects that if the root of the tree were at the bottom instead of the top, $back/up$ would be expected to be the better of the two.

The biggest surprise is that $middle$ is not even as good as a single cycle of the $multiple$ HP strategy. Apparently having to untwist the crossings in only half the levels of the graph as done by each phase of the $middle$ algorithm is not as useful as the two complete passes done during a single $middle$ style cycle.

In every case tested the $multiple$ HP strategy is best. The computing cost is that it takes on the order of 2.5 HPs before it completes. Since even for the worst case only a few seconds of wall clock time is required on a Sun Sparcstation 1, from this data we conclude that $multiple$ is the HP strategy of choice.

Among the strategies that allow multiple cycles, alternating takes the fewest HPs on average before it stops. It still is not a useful strategy since a single *multiple* cycle or using the *middle* HP strategy always produces fewer edge crossings. The highest number of average HPs before failing to lower the crossing count any further is taken by the repeated $forward/down$ or repeated $back/up$ HP strategies. Sometimes they averaged more than 6 HPs while on the same graphs *multiple* averaged less than 2.5 HPs.

For all of the strategies tested, the method of crossing count reduction for a single pair of levels is identical. Any two nodes within the level are eligible to be swapped. Checking all pairs of nodes was a single iteration. As part of the analysis of the HP strategies, data was generated on the number of iterations required before there was no more improvement in the crossing count reduction for a single pair of levels. Typically, only two or three iterations were required. Four iterations probably made up less than five percent of the cases and five passes were rarely seen.

5.3 Crossing Reductions Missed

It is instructive to examine figure 22 to see if there are any rank node swaps that would further reduce the crossing count and to understand why they were not swapped. Notice that moving node $(5-2)$ to the right of $(2-2)$ would reduce the crossing count by 1. Similarly $(7-1)$ could be moved one place to the left. Since the only HP strategy used was $back/up$, these opportunities were missed. Either a $back/down$ or a $forward/down$ level pass within a HP strategy are required for these moves to occur.

Similar problems were mentioned in section 4.3.3 in conjunction with figure 20 which is the result of the "keep square nodes fixed in place" constraint problem. This would appear to be the same problem as in figure 22 suggesting that modifying the HP strategy is the solution. Node 1-1 cannot be swapped with either 3-1 or 7-1 (5-1 is a square and must be not be moved) for a reduced edge crossing count. 1-1 must be inserted to the right of 5-1, a possibility not allowed by the greedy swap algorithm. A similar problem results in the inability to move 6-2 to the far left edge of its rank.

5.4 Level Pass Anomalies

Further examination of figure 20 indicates that even more crossing reductions are possible. If both nodes 2-4 and 3-5 together were moved to the far right edge, 14 more crossings could be eliminated. But this would require a much more subtle algorithm that would look at both the current and previous level pairs or at the current and next level pairs and count their crossings before deciding whether to swap nodes. Alternatively, we could consider moving all pairs of nodes in adjacent levels as suggested for the joint move of 2-4 and 3-5.

6 Conclusions

The algorithm developed here for directed graph edge crossing reduction has been demonstrated to be equal to and usually superior to previous methods published for the set of tested graph types. It is also superior to many in that it directly tests edge crossing counts in evaluating node placement. None of the methods previously published incorporates the constraint problem at all, let alone the variety of constraints that this method will allow. For the first time, an analysis of hierarchical pass strategies has been done and a new strategy is shown to more effective than all previous ones on the series of random tests evaluated. Unlike

several of the other methods, this technique explicitly allows for cycles within the graph.

6.1 Further Improvements

This technique will not reduce the edge cross count to the minimum number possible; occasionally even some obvious reductions are missed but this could be improved by additions to the level pass technique and/or additions to the HP strategy of choice. For all HP strategies tested, adding a constaint to the graph layout seems to increase the likelihood of a missed opportunity for edge crossing reduction.

The major shortcoming of the greedy swapping algorithm is that two nodes must be interchanged. Although a greedy insertion method [EK86] has been shown to be less effective than greedy swapping, insertion has a strong appeal. Given the crossing count with a node in one rank configuration, then testing it before, after and between the other nodes in the rank seems an appropriate method. This would be done repetitively for all nodes in the rank until no further reduction in count were possible. This suggested method would eliminate many of the missed opportunities previously described in section 5.3.

If the directed graph had square nodes all along one vertical edge of the graph and the constraint was that the square nodes must be fixed, another aspect of the problem with the swapping method would surface. All node movements would be along one side of the squares. The crossing reduction scheme would not take advantage of trying node placements 'outside' the squares. The insertion method above would avoid this artifact of the swapping method.

The method could be further extended, as already indicated, to include more than a single pair of levels as the incremental segment of the graph for edge crossing reduction. For instance when swapping nodes within a level, the crossing count for both the level before and the level after could be minimize together. It is not at all obvious how much of an improvement would result from the increase in computational effort.

Beyond the algorithm itself are additions such as the bend straightening previously alluded to. Other possible additions are routines to create a ranking given just the nodes and links [Wir76]. If this technique were to be part of a hypertext browser as initially conceived, an interactive display of the graph built using windowing toolkits such as Sun's Xview would be very useful.

6.2 The general graph

This new algorithm specifically ignores link directionality. An immediate consequence is that there is no limitation on cycles within the graph. An more significant result is that the method is therefore directly applicable to general graphs. As examples of using it for general graphs, just ignore the arrowheads on the links in all the directed graphs used here. The algorithm never noticed the link directionality.

The general technique for an arbitrary graph is as follows:

```
organize nodes into ranks
use method already described
```

General graphs with the nodes fixed on a rectangular grid can be considered to have ranks in either the horizontal or vetical direction. Testing would be necessary in order to evaluate which gives the best results in edge crossing reduction. One strategy might be to try both and use the result of the lesser number of edge crossings. Another technique might be to alternate cycles of HPs between the two orthogonal directions. If there is a predominate directionality to the edges, it is likely to be helpful to align the grid used with the grain. Once again, an open question is whether the top/bottom direction should be parallel or perpendicular to the apparent grain.

References

[Car80] Marie-Jose Carpano. Automatic display of hierarchized graphs for computer-aided decision analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10(11), November 1980.

[Dav85] Michael B. Davis. A layout algorithm for a directed graph browser. Master's thesis, Dept. of EE & CS, UC-Berkeley, May 1985.

[EK86] Peter Eades and David Kelly. Heuristics for drawing 2-layered networks. *Ars Combinatoria*, 21-A:89–98, 1986.

[EMW85] Peter Eades, Brendan D. McKay, and Nicholas C. Wormald. An np-hard crossing number problem for bipartite graphs. Tech. Rep. 60, Department of Computer Science, Univ. of Queensland, 1985.

[ET88] Peter Eades and Roberto Tamassia. Algorithms for drawing graphs: An annotated bibliography-revised version september 1989. Technical Report 82, Department of Computer Science, Univ. of Queensland, 1988.

[EW89] Peter Eades and Nicholas C. Wormald. Edge crossings in drawings of bipartite graphs. Technical Report 108, Department of Computer Science, Univ. of Queensland, March 1989.

[GNV88] E. R. Gansner, S. C. North, and K. P. Vo. DAG-A Program that draws directed graphs. *Software-Practice and Experience*, 18(11), November 1988.

[HMT87] F. G. Halasz, F. G. Moran, and R. H. Trigg. Notecards in a nutshell. In *Proceedings of the ASM CHI + GI Conference*, pages 45–52. ACM, 1987.

[LENW82] Daniel E. Lipkie, Steven R. Evans, John K. Newlin, and Robert L. Weissman. Star graphics: An object-oriented implementation. *Computer Graphics*, 16(3), July 1982.

[Mey83] Carl Meyer. A browser for directed graphs. Master's thesis, Dept. of EE & CS, UC-Berkeley, December 1983.

[RDM+87] Lawrence A. Rowe, Michael Davis, Eli Messsinger, Carl Meyer, Charles Spirakis, and Allen Tuan. A browser for directed graphs. *Software-Practice and Experience*, 17(1), January 1987.

[RT81] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, SE-7(2), March 1981.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2), February 1981.

[Sug87] Kozo Sugiyama. A cognitive approach for graph drawing. *Cybernetics and Systems: An International Journal*, 18, 1987.

[Sym88] Symantec Corporation, Living Videotex Division, 10201 Torre Avenue, Cupertino, CA 95014. *More ll-Planning Write and Desktop Presentations*, September 1988.

[TDBB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1), February 1988.

[Vau80] Jean G. Vaucher. Pretty-printing of trees. *Software-Practice and Experience*, 10, 1980.

[War77] John N. Warfield. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-7:505–523, July 1977.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[Woo81] Donald Roy Woods. *Drawing Planar Graphs*. PhD thesis, Stanford University, 1981.

[WS79] Charles Wetherell and Alfred Shannon. Tidy drawing of trees. *IEEE Transactions on Software Engineering*, SE-5, September 1979.

[Xer85] Xerox Artifical Intelligence Systems, Pasadena, CA 91109. *InterLispD Lisp Library Packages*, April 1985.