AD NUMBER
ADA166622
LIMITATION CHANGES
TO: **Approved for public release; distribution is unlimited.**
FROM: **Distribution authorized to U.S. Gov't. agencies and their contractors; Administrative/Operational Use; 30 DEC 1985. Other requests shall be referred to Defense Advanced Research Project Agency, 675 North Randolph Street, Arlington, VA 22203-2114.**
AUTHORITY
DARPA ltr, 11 Apr 1986

A Browser for Directed Graphs

Technical Report

S. L. Graham

(415) 642-2059

Contract No. N00039-82-C-0235

November 15, 1981 - December 30, 1985

Arpa Order No. 4031

A Browser for Directed Graphs

by

Carl Meyer

Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Table of Contents

Chapter 1

Introduction

1.0. Graphics as a Tool

The availability of low-cost workstations with a bit-mapped display and mouse has made possible widespread use of graphics software. User interfaces can display data through a convenient graphics interface rather than the textual presentations necessitated by older technology.

One of the most applicable structures of computer science is a graph (i.e., a graph with vertices and edges as opposed to a drawing). Graphs can be used to represent finite state machines, source code dependencies, the structure of a hierarchical file system, dynamic or static procedure call graphs, an office organization, network systems, or parse trees. In general, any system which can be described by the dependencies among its components can be represented concisely with a graph.

In the past, graphs have been represented by a list of vertices and incident edges, as seen in table 1.1. The most natural representation of a graph is a

Sample Graph		
From Node	To Node	Label
q_0	q_1	a
q_0	q_2	b
q_1	q_3	d
q_1	q_0	c
q_1	q_2	a

Table 1.1: Tabular listing of a graph.

drawing depicting the vertices as circles and the edges as arcs connecting the vertices, as shown in figure 1.1. From table 1.1 and figure 1.1 it is clear that a graphical presentation of a graph is preferable to a tabular one. However, the problem of drawing a graph is a non-trivial one. Indeed, the notion of a "nice presentation" is far more subjective than objective. The complexity of this problem combined with the previously prohibitive expense of graphics equipment has resulted in very little literature in this area.

If the graph drawing problems could be overcome, a tool could be written to browse through graphs using the natural graphical display. Such a tool, or *graph browser*, would display graphs in a reasonable format and allow users to query or edit the data represented by the graphs. There are many examples where a graph browser would be useful: the call-graph of a software package could be browsed to become familiar with the structure of the package; the parse trees of a compiler could be browsed while debugging the parser; a finite automaton could be displayed, edited, and queried. The many uses of graphs in computer science provide an unlimited number of graph browser applications.

This report describes the design of a general graph browser based on our experiences with an earlier prototype. Chapter II discusses the prototype and

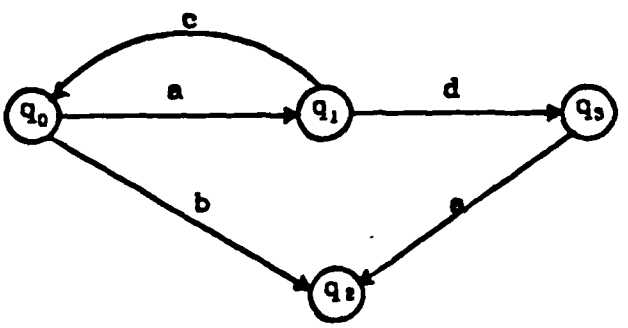

Figure 1.1. Graphical representation of a graph.

the limitations encountered with it. Chapter III describes the design of a general graph browser, called GRAB, and the user interface it provides. Chapter IV describes an implementation design of GRAB, including the database design and the heuristics used to draw reasonable-looking graphs. Chapter V concludes with a summary and a discussion of future extensions to GRAB.

Chapter 2

An Early Version of GRAB

2.0. Description of GRAB0

Before implementing the first version of GRAB (hereafter called GRAB0), two approaches to graph display were considered: 1) An attempt could be made to draw the entire graph with arcs connecting the nodes, or 2) A portion of the graph could be drawn with arcs not drawn but implicitly represented.

The first method was considered too formidable a task, with problems dealing mainly with node placement so as to minimize arc crossings. In contrast, the absence of arcs provided by the second method would allow for a simple node placement algorithm. Consequently, GRAB0 displayed a portion of a graph without arcs.

A graph was displayed by focusing attention on a particular node in the graph (hereafter called the "focus node"). The portions of the graph relevant to the focus node are the node's parents (predecessors) and children (successors). The focus node could be drawn in the center of the screen, its parents above it, and its children below. Figure 2.1 shows a sample display produced by GRAB0. The nodes were drawn in rectangles just large enough to encompass their labels. Nodes were grouped in rows above or below the focus node. The nodes grouped above the focus node are its parents and the nodes grouped below it are its children. This representation avoided the problem with arc crossings, since no arcs need be drawn at all: arcs from parents to focus node to children are implicit in the representation.

Graphic oriented browsers traditionally provide facilities for *travel* and *selection*. Travel commands allow the user to move around the entire object

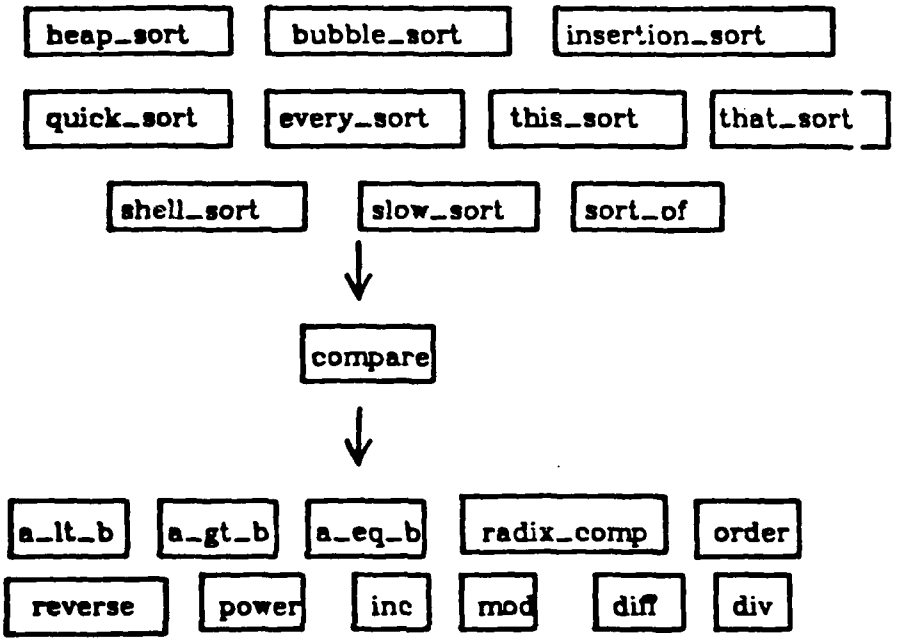

Figure 2.1. Sample GRAB0 display.

being displayed and make different portions of the object visible on the screen. Selection commands allow the user to pick various parts of the display. Items which have been picked can then be used in conjunction with other commands (e.g. to delete an item one would first select the item or items to delete and then invoke the delete command). GRAB0 provides for a limited form of travel by allowing parent and children rows of nodes to be scrolled up or down. Figure 2.2 illustrates a graph view where the number of parents is too large to fit on the screen. When this occurs, the user is presented with "scroll boxes" at the top and bottom (if the children don't fit) of the display. A scroll box consists of up and down selections (arrows). The user can scroll parent or children rows up or down by picking the appropriate arrow in the corresponding scroll box.

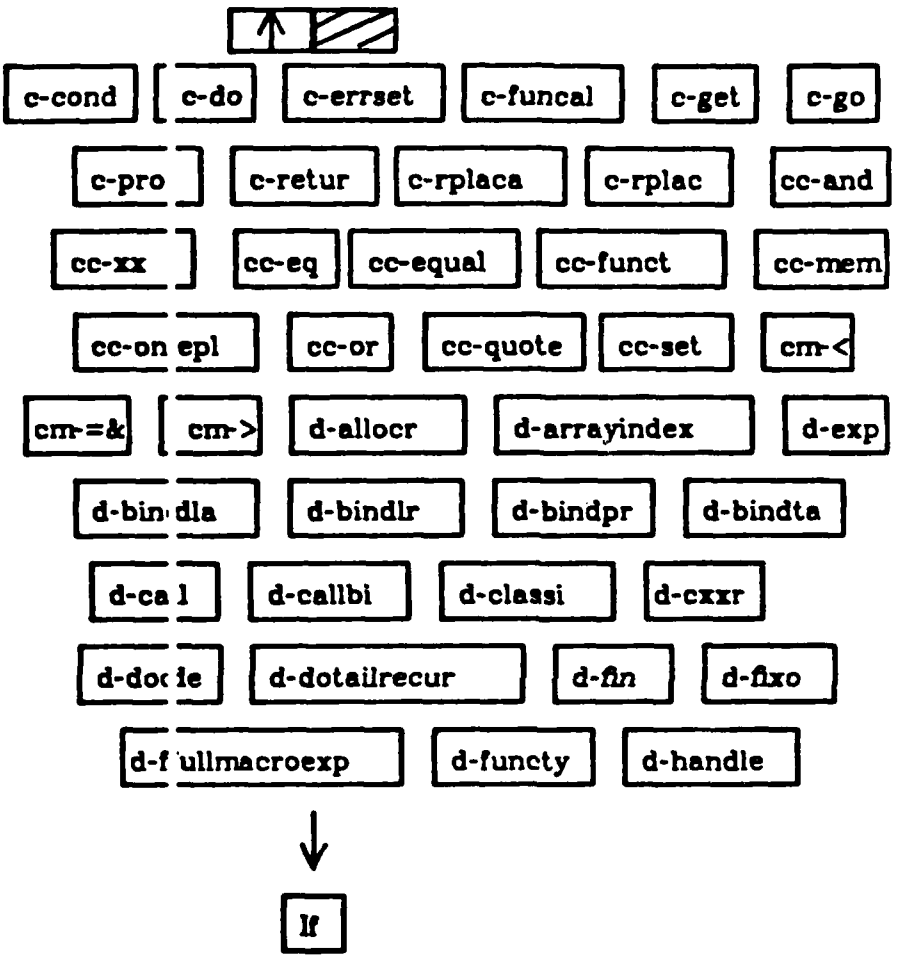

Figure 2.2. GRAB0 display of a large graph.

Selection is accomplished by picking a node of interest. The selected node becomes the new focus node and the display is cleared and redrawn to show the parents and children of the new focus node. For example, selecting the node named "cc-equal" in figure 2.2 results in the display shown in figure 2.3.

Color was used in GRAB0 to indicate the fruitfulness of selection. Parent nodes were painted green if they themselves had parents. Children nodes were painted green if they had children. This was deemed useful since it was

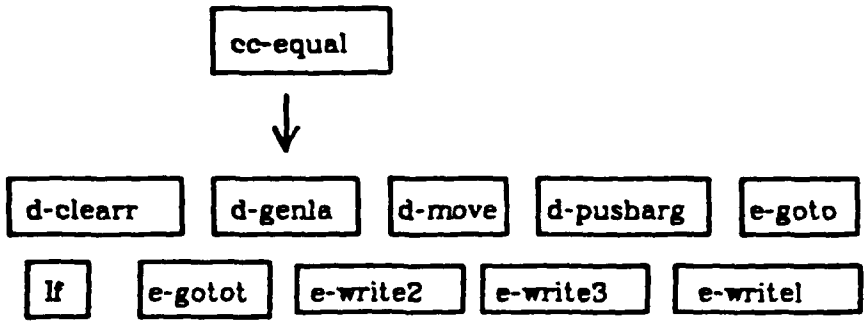

Figure 2.3. Result of selecting cc-equal from figure 2.2.

anticipated that traversal of parent/children chains would be a common operation. Thus the user could see which node selections would yield "grandparents" or "grandchildren".

2.1. Problems with this Early Version

GRAB0 was easy to implement. While the system provided a tool that could be used to browse a graph, it was hard to learn because it did not display graphs in the normal way, and it had limited usefulness because it did not provide operations to edit graphs and selectively display large graphs.

One problem with the display is the absence of arcs. The basic notion of a graph is a representation of relationships among entities -- the arcs being the relationships and the nodes the entities. GRAB0 essentially displays only two implicit relationships: that of "parent-of" and "child-of". This is fine for applications such as call-graphs: the parents "call" the focus node and the children are "called-by" the focus node. However, applications which do not display this binary symmetry do not lend themselves to this model. An example is the display of a finite automaton. Figure 1.1 illustrates a sample finite automaton. Here the label on an arc defines a transition relationship between incident

nodes. An attempt to display this finite automaton is shown in figure 2.4, with node q3 as the focus node. From figure 2.4 it is clear that the arcs are essential for understanding the underlying graph structure.

The second display problem is one of orientation. When a graph is presented in its entirety and drawn in the conventional manner, a user is given a global sense of the entity relationships depicted by the graph. With GRAB0 the concept of a focus node provides no global sense. Moreover, users would tend to get lost in the sea of nodes without having a feeling for where they were in the graph.

The third display problem is that of scope. With GRAB0 the user is given only one level of look-ahead and look-behind. So it is not possible, for example, to view an entire calling sequence in a call graph.

The problem with the GRAB0 operations is that only two were provided: scroll and select. The data represented in the graph could not be updated. The

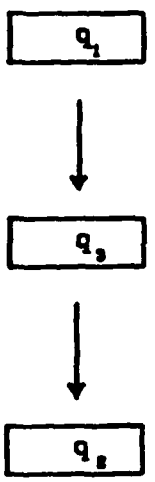

Figure 2.4. Finite automaton from figure 1.1 as drawn by GRAB0.

user could not make arbitrary queries about the graph data. Finally, there was no operation which would allow a user to jump to a node in the graph that was not on the display (i.e. only displayed nodes could be selected).

In conclusion, GRAB0 was much too simple. A more complete system was needed. In fact, the people who used GRAB0 were excited by the prospect of a general purpose graph browser. The next chapter describes the design for such a browser.

Chapter 3

Using GRAB

This chapter illustrates the functionality of GRAB. A call-graph application is used to demonstrate the type of graphs drawn and the various operations that can be performed.

3.0. Overview of GRAB

The data for the graph to be displayed is stored in a relational database. The database provides a standard format for the graph data that can be used by applications other than GRAB. The information in the database describes the connectivity of the graph as well as node-specific information such as a display icon, a name, etc. GRAB is simply a front-end to browse the data in the data base. The database schema is described in the next chapter.

The user interface provided by GRAB is similar to that of the Lisa Desktop Interface [Apple83]. Figure 3.1 illustrates a sample (empty) display. A GRAB display consists of a window with a pull-down menu bar at the top and horizontal and vertical scroll bars. Multiple windows can be overlapped on the screen during the session. Various commands open up a new window to display their results. A window can be positioned anywhere on the screen simply by dragging the corner of the window to the desired position. The *current window* has its title highlighted; another window can be selected by clicking the mouse within the desired window.

3.1. Displaying a Graph

Figure 3.2 shows a GRAB display of a call-graph. Graph nodes are drawn within the window with their respective icons. Arcs are displayed as labeled straight lines with an implied top-to-bottom direction. The entire graph is

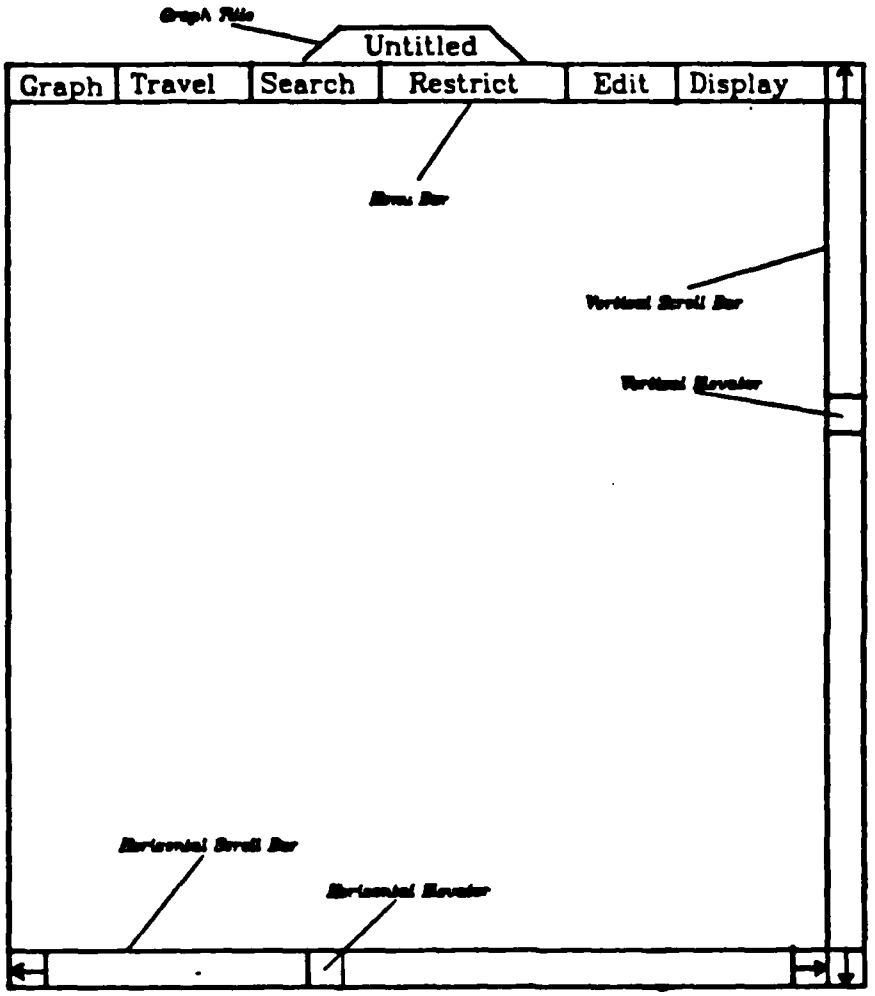

Figure 3.1. Sample empty GRAB display.

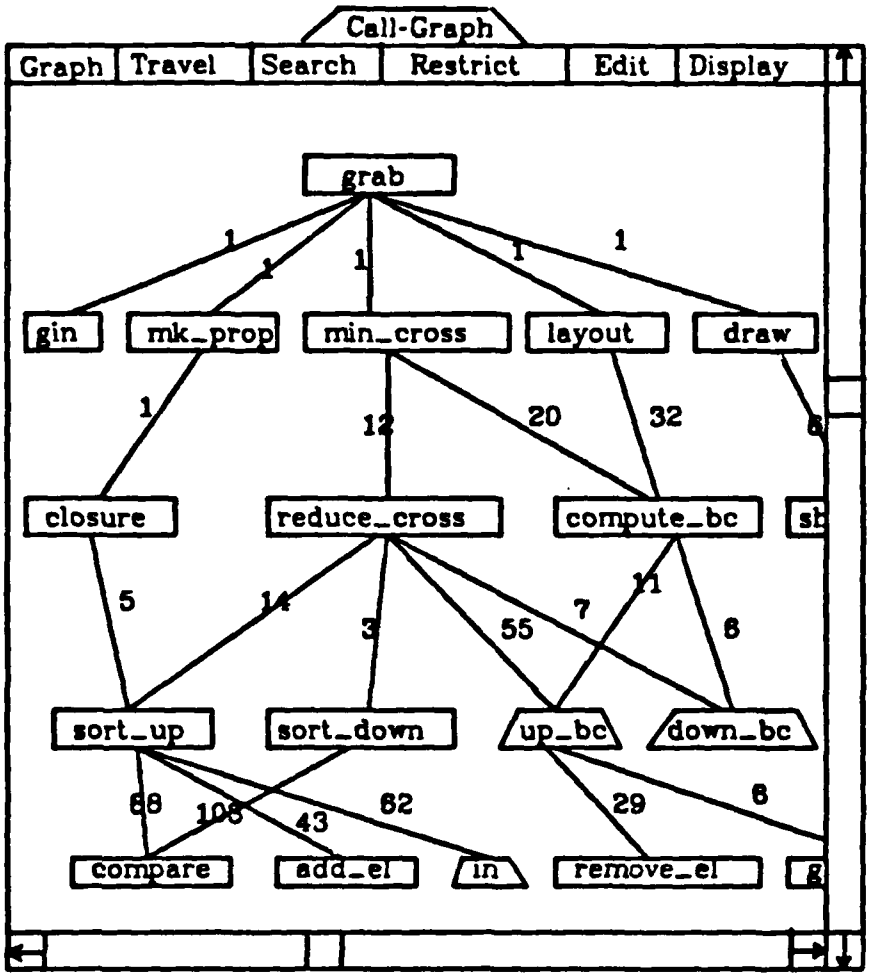

Figure 3.2. GRAB display of a call-graph.

clipped to fit within its window.

Figure 3.3 displays the contents of the GRAB menus. The remainder of this chapter is devoted to a discussion of a subset of the various menu commands.

3.2. Selection

Many of the menu commands operate on the *current nodes* or *current arcs*. The user can select a node or group of nodes by clicking a mouse button once

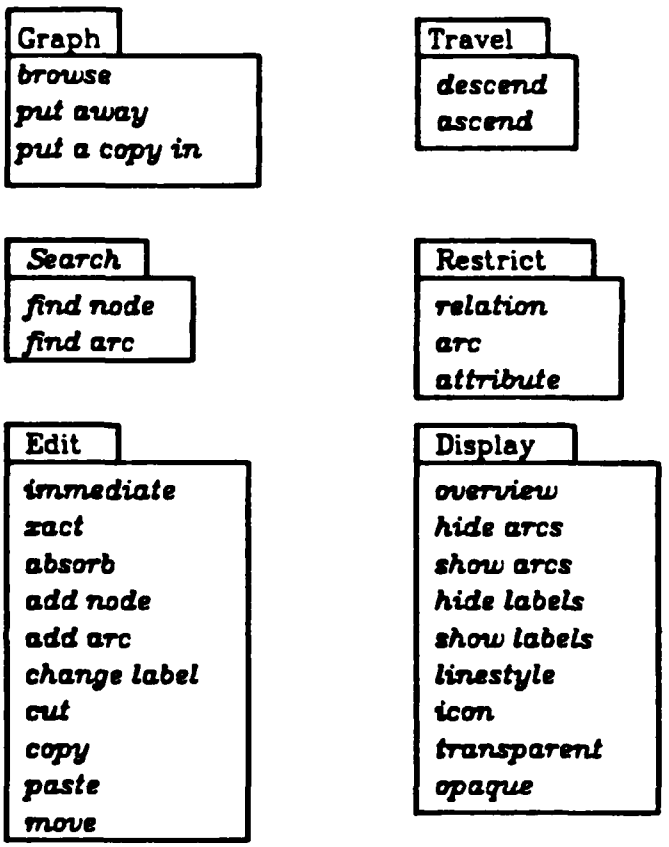

Figure 3.3. GRAB menus and their contents.

on each desired node. The selected nodes will be highlighted. Similarly, arcs or groups of arcs can be selected by clicking the desired arcs.

3.3. Moving Around the Graph

Many graphs are very large and hence do not fit within the window. The user can browse through other parts of such a graph by either *scrolling* or *traveling*. The scroll bars on the current window contain *page scroll boxes* and *scroll elevators*. Scrolling is accomplished by either clicking the page scroll boxes or dragging the scroll elevators. The page scroll boxes scroll a window-

full-at-a-time, and the scroll elevators allow arbitrary scrolling.

Another method for moving about is to select one of the commands in the search menu. For example, suppose we want to display the portion of the call-graph containing the procedure "reduce_cross." This can be done with the *find node* command in the search menu. GRAB will prompt for the name of the desired node to search for. Once this is entered, the graph will be scrolled to center "reduce_cross" in the window, as was shown in figure 3.2. The node which was searched for becomes the new current node (and hence need not be selected before using some of the other menu commands pertaining to it).

When the graph is very large, an overview of the desired graph may be desirable. An overview is simply a version of the graph reduced to convey the global sense of connectivity. The overview can be displayed by selecting the overview command in the display parameters menu. For example, figure 3.4 shows the overview for the call-graph. GRAB creates the overview by reducing all nodes to points and hiding all node names and arc labels. The overview can be used to "hop" around the graph by clicking the mouse on the desired node to hop to.

3.4. Selective Display

Often times a graph will display many relations among its various nodes. This can be confusing, since the display will tend to be cluttered with the many arcs of each relation. Which relations to display on the graph can be controlled by the *relation* command in the restrict menu. For example the arcs in the call-graph shown in figure 3.2 are labeled with the number of times each procedure is called. If we wish to only see those procedures which are in the relation "A calls B n times", we can invoke the restrict-by-relation command and provide the name of the desired relation (either by selecting an arc labeled with the relation, or entering it from the keyboard). The display will be restricted to only those arcs with the desired label (and their incident nodes).

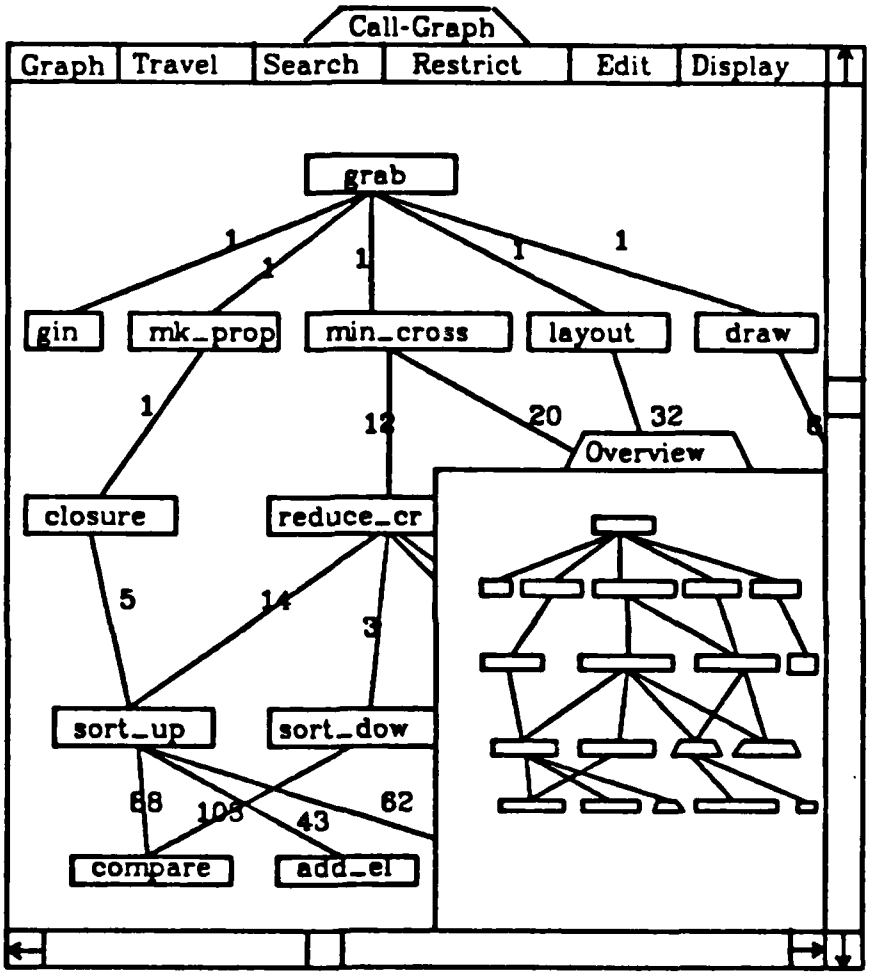

Figure 3.4. Overview for figure 3.2.

GRAB predefines three common relations: parent, child, and sibling. These are useful when a particular node is referenced. Parents correspond to predecessors of a particular node, children are the successors, and siblings are the children of the parents. In the call-graph example, it is very useful to restrict the display to only the parents and children of a particular node. When the user chooses to restrict the display in this manner, an option is available to specify the level of ancestry desired (e.g. level 1 for parents, level 2 for parents and

grandparents, etc.). This allows users to view any portion of the call chain for a particular procedure. Figure 3.5 shows an example of restricting the display to only the parents and children (level 1) of "compute_bc".

The arc labels can be treated as data as well as relation names. For example, suppose that we wish to only see those procedures which call some procedure or are called more than five times. Since the arcs are labeled with the

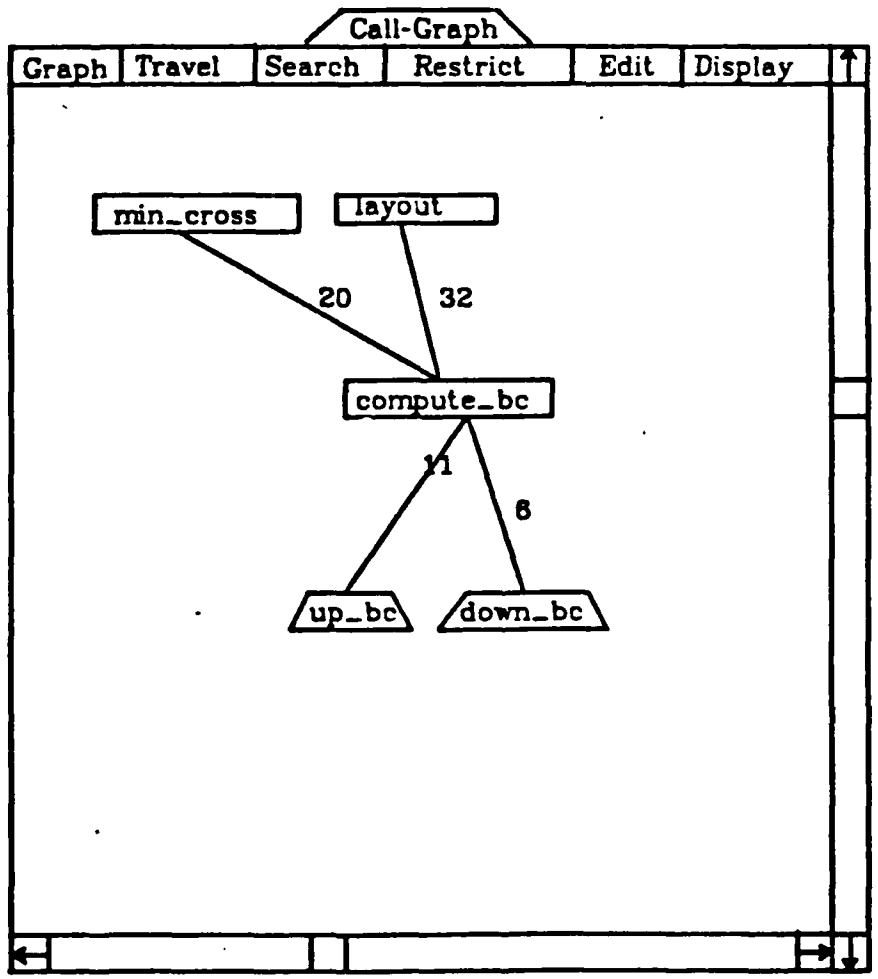

Figure 3.5. Call-graph restricted to parents and children of compute_bc.

number of times a call is made, we can do this by using the restrict-by-arc command. This command will prompt for the desired class of arc labels to which the display should be restricted. A class can be defined by giving either an absolute value such as "5" or an inequality such as ">= 5". The result of restricting to the class ">= 5" is shown in figure 3.6. Note that this is not the same as restrict-by-relation, since the restriction is simply used to hide nodes that are not incident with arcs of the desired class, rather than restrict the display to only one relation.

3.5. Editing the Graph

The graph can be changed by invoking one of the commands in the edit menu. For example, suppose now that we wish to delete from the graph all of those nodes which are currently hidden from the display by the previous restriction command. This can be done by selecting the *absorb* command in the edit menu. The hidden nodes will be removed from the database, and will not reappear should the previous restrictions be turned off. Changes made to the graph can be committed immediately to the database by selecting the *immediate mode* in the edit menu; or they may be deferred (until the user invokes the commit command) by choosing the *xact mode*.

Delete, change, or add node or arc operations are also available. In addition the user may *cut* and *paste* parts of the graph. Cutting corresponds to removing a subgraph of the graph and placing it in a special window called the *scrap*. If the user does not wish to remove the subgraph from the graph, the *copy* command may be used instead. Pieces of the graph in the scrap may be pasted back into the graph with the paste command, or they may be pasted into a graph in a different window.

If the user is not satisfied with GRABs placement of the graph nodes, the *move* command may be used to position particular nodes manually. The user simply selects the desired node and drags the mouse to its new position. The

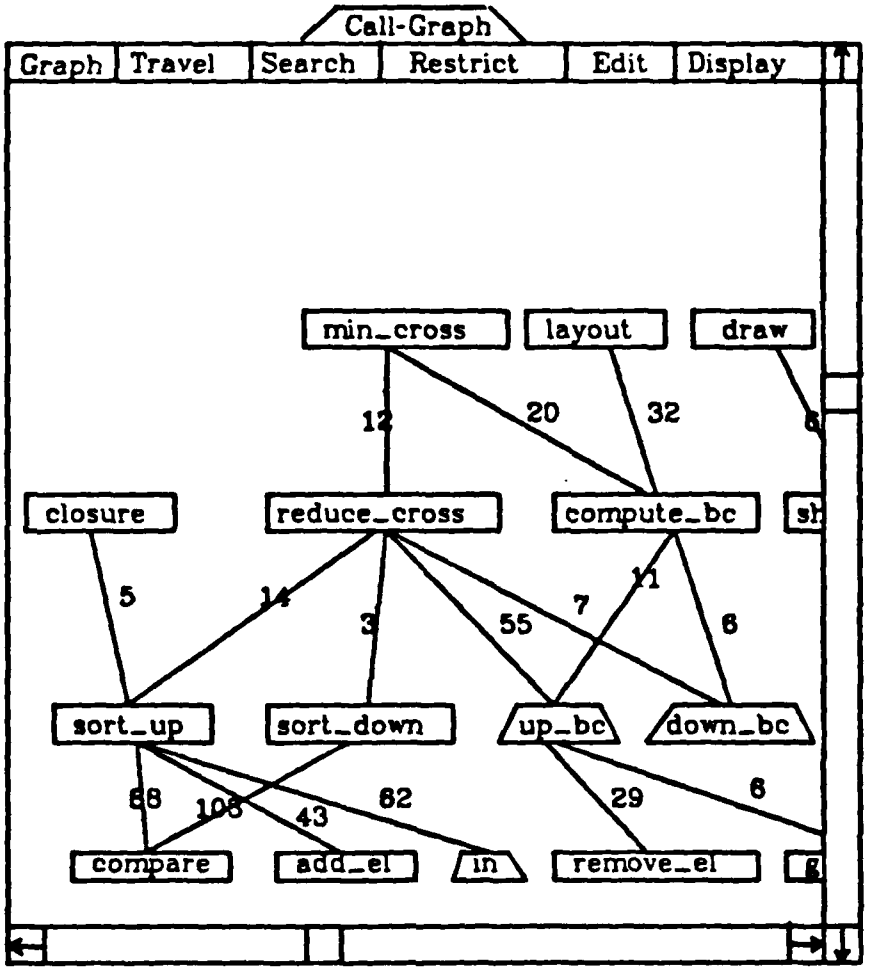

Figure 3.6. Call-graph with arcs restricted to those with labels >= 5.

arcs attached to the node will flex in a rubber-band fashion as the node is moved.

After a series of graph updates, the graph may be displayed in a haphazard fashion. This can be corrected by invoking the *position* command. GRAB will rerun the heuristic graph displaying algorithm and automatically position all of the nodes in a hierarchical format. The graph displaying algorithm will be dis-

cussed in chapter 4.

3.6. Customizing the Display

GRAB is a general graph browser and can be used to browse graphs for many different types of data. Each type of data may require a somewhat different display format. This is taken into account with the commands in the display menu.

Arcs can be hidden or unhidden, labeled or not, and can be drawn in different line styles. Changing line styles is useful when many relations are depicted in one graph. For example, figure 3.7 illustrates the effect of changing the call-graph so that arcs labeled with a value of twenty or more are highlighted.

Nodes can be displayed with various icons. If no icon is specified in a node's database entry, the default icon is a rectangle. This can be changed to any one of a number of predefined icons, or the user may define a new icon. Figure 3.2 shows the call-graph with rectangles used for procedures and trapezoids for functions. This adds to the information conveyed simply by looking at the graph.

3.7. Hierarchical Graphs

A hierarchical graph is one in which nodes in the graph may contain subgraphs. An example of a hierarchical graph can be seen with the call-graph which we have been using. The nodes in this graph represent procedures or functions which are contained in files. The files themselves are dependent upon each other. This description is typically given in a *makefile* description file [Feldman79]. The graph defined in the makefile for procedures and functions of the call-graph is shown in figure 3.8. Each node defines a file which contains one or more procedures or functions. So we have a two-level hierarchy: a file graph and a procedure-function graph. This can be extended further if we note that subprograms contain local variables that are defined and used. This def-use

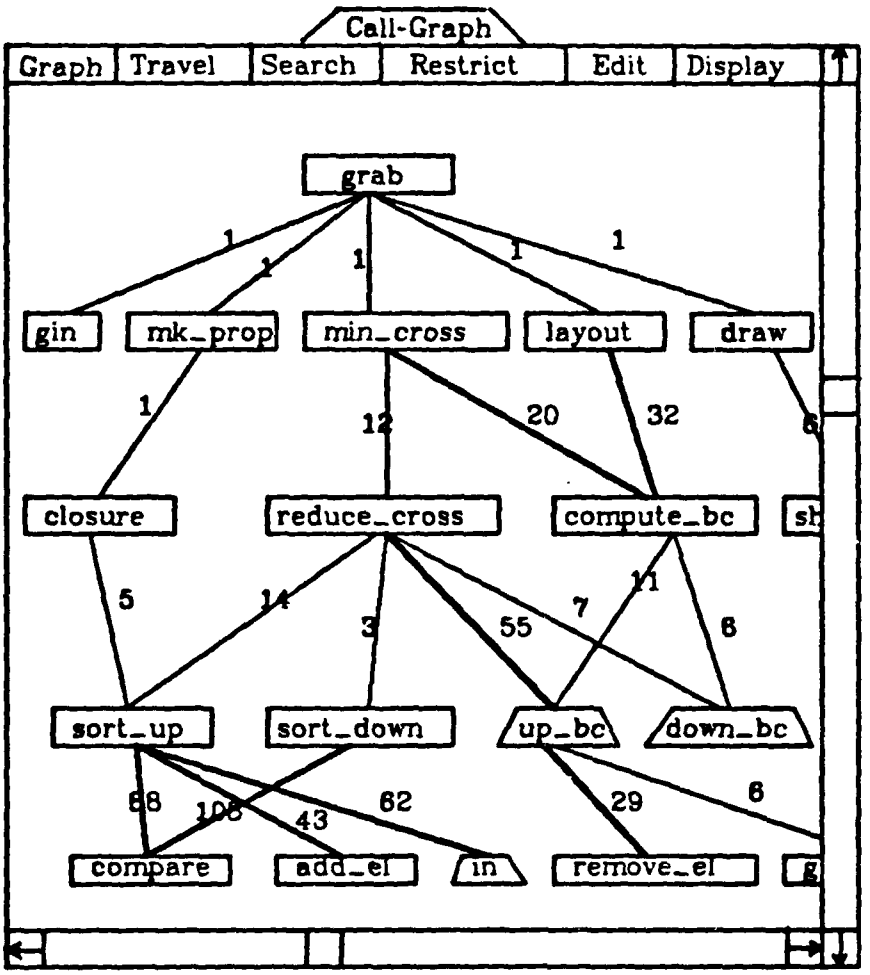

Figure 3.7. Example of display versatility: arcs labeled with a value of twenty or more are highlighted.

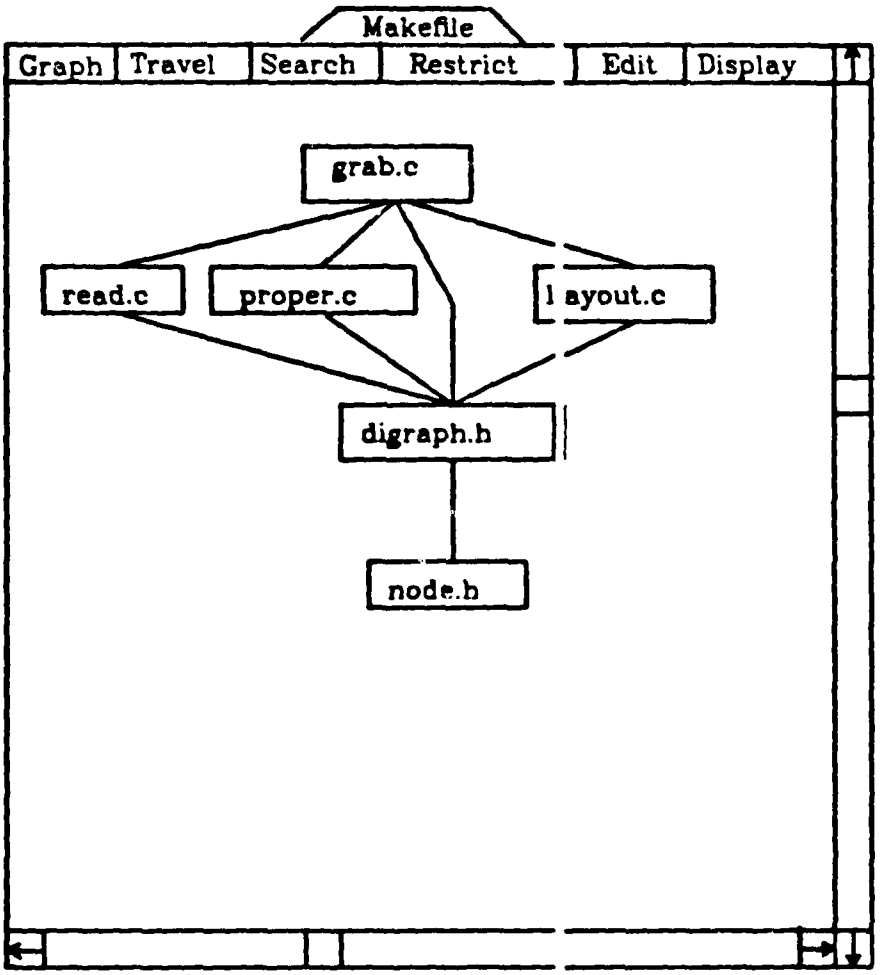

Figure 3.8. Hierarchical graph defined by a makefile.

relationship defines a graph. Thus our call-graph is simply a special case of a three-level hierarchical graph:

FILES → PROCEDURES → VARIABLES.

3.7.1. Hierarchical Operations Hierarchical graphs can be browsed with the *descend* and *ascend* commands in the travel menu. The descend command clears the display and draws the subgraph contained in the current node. Simi-

larly, the ascend command redraws the graph in which the current node is contained. An additional display parameter allows hierarchical nodes to be displayed as either *transparent* or *opaque*. Transparent nodes are drawn with their contained subgraph visible (nodes in the subgraph may themselves be transparent). Opaque nodes do not display their subgraphs.

3.8. Summary

In summary, GRAB provides a convenient interface for browsing and editing directed graphs. The ability to select pieces of the graph simplifies the interaction with various commands. Several commands are provided to move around the graph, selectively display various parts, and edit the graph. Finally, the appearance of the graph can be altered by choosing a series of display parameters.

In the next chapter the GRAB database schema is discussed along with the heuristics used to display reasonable-looking graphs.

Chapter 4

Implementation

This chapter describes the design of a database schema for GRAB, an algorithm for drawing graphs, and performance of a prototype implementation of the graph drawing algorithm.

4.0. The Database Schema

Before GRAB can be used to browse a particular graph, a *map* must be given from the user's database schema to that of GRAB. GRABs database schema is shown below.

GRAPH(graph_name, node_relation, arc_relation,
icon_relation, type_relation, attribute_relation)

A graph is defined with a tuple in the GRAPH relation. The attributes in the tuple are described below.

graph_name

The name of the graph.

node_relation

The definition of the relation that contains the nodes of the graph. The schema of this relation is

node_relation(node_name)

where *node_relation* is the name of the relation containing the nodes and *node_name* is the attribute in node_relation with the node names.

arc_relation

The definition of the relation that contains the arcs of the graph. The schema for this relation is

arc_relation(from, to, label)

where *arc_relation* is the name of the relation that contains the arcs, *from* is the name of the attribute in arc_relation defining the tail of each arc, *to* is the attribute defining the head of each arc, and *label* is the attribute defining the arc labels.

icon_relation

The definition of the relation specifying the icons for each node. The schema for this relation is

icon_relation(node_name, icon_value)

where *icon_relation* is the name of the relation that defines the icons, *node_name* is the name of the attribute in icon_relation with the node names, and *icon_value* is the name of the attribute with the icons.

type_relation

The definition of the relation specifying the type of each node. The schema for this relation is

type_relation(node_name, type)

where *type_relation* is the name of the relation that specifies the node types, *node_name* is the name of the attribute in type_relation with the node names and *type* is the name of the attribute with the type of each node. The type of each node is used to determine the attributes that may be queried or set while browsing.

attribute_relation

The name of the relation defining the attributes for each node type. The schema for this relation is

attribute_relation(type, attribute_name)

where *attribute_relation* is the name of the attribute relation, *type* is the name of the attribute in attribute_relation with a node type, and *attribute_name* is the name of the attribute in attribute_relation that

contains the attribute value.

The use of relation maps frees the user from having to use a predefined schema. Data stored in arbitrary relations can be browsed without making changes to the relations. As an example, the relation maps for the call-graph used in the previous chapter are shown below.

```
GRAPH(
    graph_name = "call-graph",
    node_relation = "proc_funcs(pfname)",
    arc_relation = "calls(caller, callee, num_times)",
    icon_relation = "pficons(pfname, icon)",
    type_relation = "pftypes(pfname, type)",
    attribute_relation = "pfattr(type, attribute)"
)
```

4.1. The Graph Drawing Algorithm

The graph drawing algorithm proposed for GRAB was first described in [Warfield76] and [Sugiyama81]. The remainder of this chapter is devoted to describing this algorithm, exposing some of its limitations, and proposing some extensions to overcome these limitations.

The graph drawing algorithm is composed of two major parts: an algorithm to transform a graph into a structure know as a *proper hierarchy*, and a set of heuristics to draw a proper hierarchy so as to satisfy a series of *readability constraints*. A dramatic example of this algorithm can be seen in figures 4.1 and 4.2. Here we see a sample graph hand-drawn in simple lexicographic order (figure 4.1) and the same graph as drawn by GRAB (figure 4.2). The the nodes in the hand-drawn graph are not positioned with any particular regularity; the many arc crossings give the graph a messy look and make the graph difficult to read. In contrast, the nodes in the GRAB-drawn graph are positioned in a systematic fashion (a hierarchy) and the arc crossings have been completely removed.

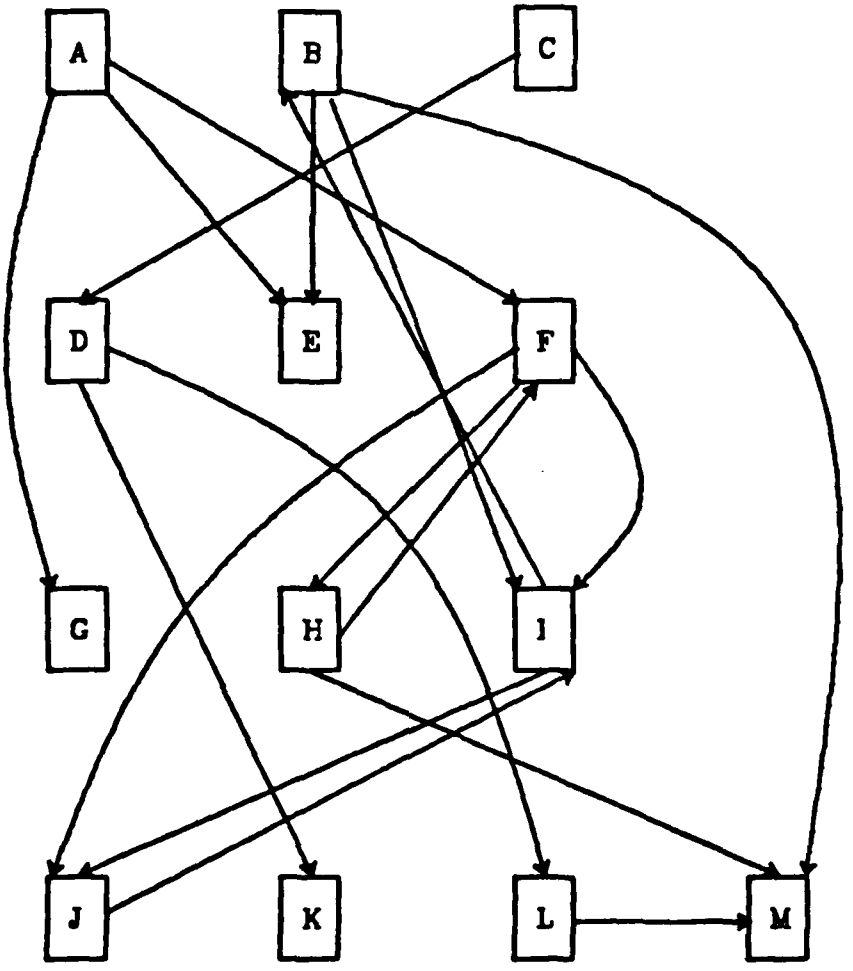

Figure 4.1. Sample hand-drawn graph.

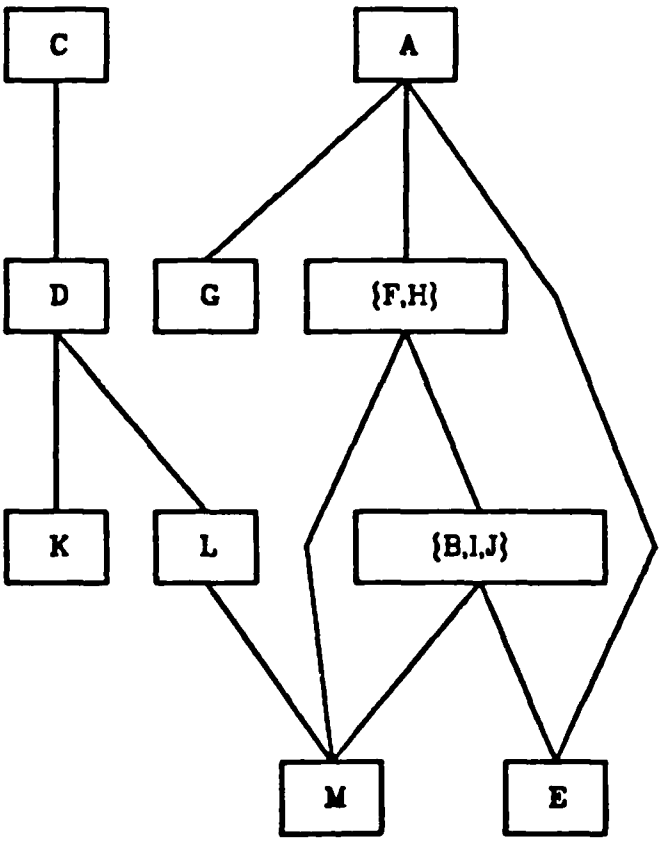

Figure 4.2. Graph from figure 4.1 as it would be drawn by GRAB.

In the next section the definition of a proper hierarchy is given and the proper hierarchy algorithm is explained. This is followed by a description of the readability constraints and the heuristics used to draw a proper hierarchy.

4.1.1. Creating a Proper Hierarchy

A graph to be displayed by GRAB is partitioned into levels, with arcs extending from an upper level to the next lower level. Cycles within a level are

resolved by creating new nodes called *proxies* to represent the nodes in the cycle. Long spans (arcs that span more than one level) are shortened by adding nodes at each intermediate level (dummy nodes). For example, in figure 4.2 one can see two proxies and two long spans. The nodes labeled {F,H} and {B,I,J} are proxies for cycles composed of the original nodes F and H, and B, I, and J, respectively. The resulting display is known as a *proper hierarchy*.

To help clarify the proper hierarchy algorithm, the graph shown in figure 4.1 will be run through each phase of the algorithm and the results will be displayed. Since arc labels do not play a part in the algorithms, none are included in our sample graph.

In the next few sections the methods for determining the graph levels, collapsing cycles, and reducing long spans are described.

4.1.1.1. Determining the Graph Levels

In order to partition the graph into levels, the transitive closure is first computed. Figure 4.3 depicts the transitive closure of our sample graph. Next the *bottom level* is determined. The bottom level consists of those nodes for which the intersection of their set of successor nodes and predecessor nodes is equal to their successor nodes (i.e. any successors must also be predecessors). Table 4.1 lists the successors, predecessors, and intersections for each node in the sample graph. From table 4.1 we see that only nodes K, M, G, and E have the intersection of their successor sets and predecessor sets equal to their successor sets. These nodes comprise the bottom level for the sample graph. Figure 4.4 demonstrates this by grouping the bottom level nodes in a row at the bottom of the display. Since the transitive closure graph (figure 4.3) is somewhat complicated, only the original arcs are drawn in figure 4.4.

Note that the bottom level nodes either have no successors (i.e no outgoing arcs) or have successors on the same level. Also, if a bottom level node has suc-

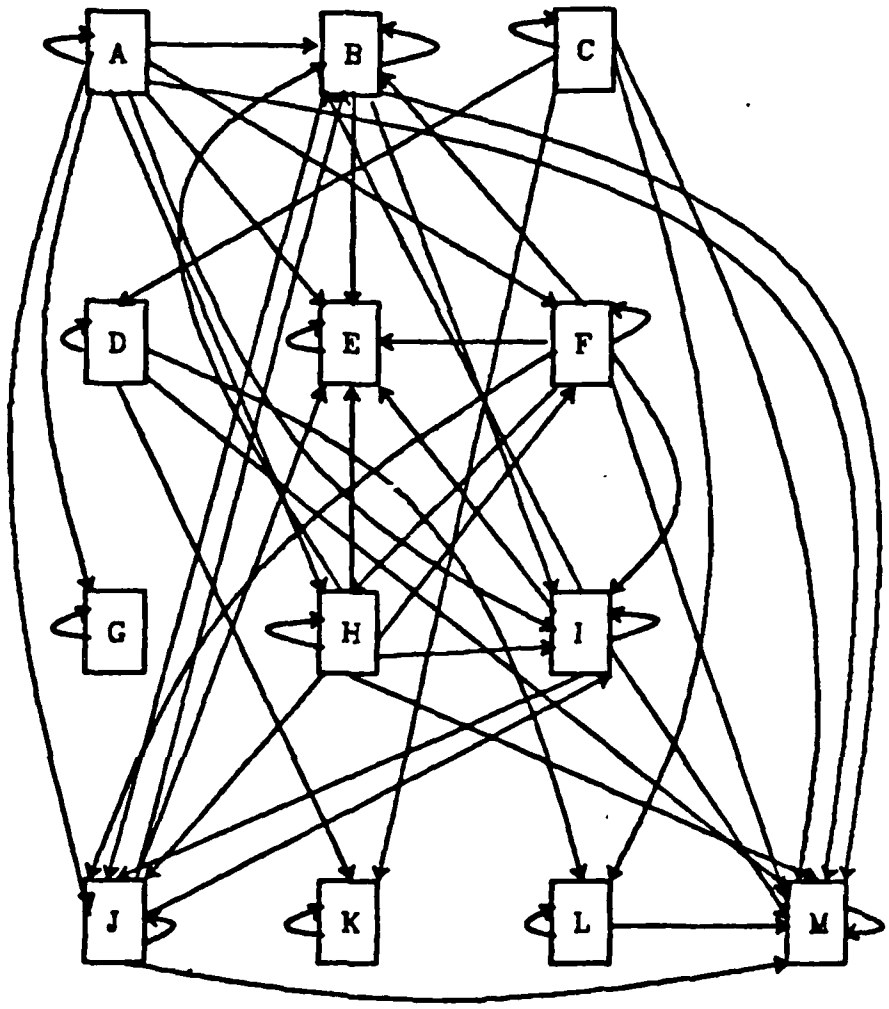

Figure 4.3. Transitive closure of sample graph.

Bottom Level Computation			
Node	Predecessors	Successors	Intersection
A	A	A B E F G H I J	M
B	A B F H I J	B E I J M	B I J
C	C	C D K L M	C
D	C D	D K L M	D
E	A B E F H I J	E	E
F	A F H	B E F H I J M	F H
G	A G	G	G
H	A F H	B E F H I J M	F H
I	A B F H I J	B E I J M	B I J
J	A B F H I J	B E I J M	B I J
K	C D K	K	K
L	C D L	L M	L
M	A B C D F H I J L M	M	M

Table 4.1. Bottom level computation indicates that nodes E, G, K, and M should be on the bottom level.

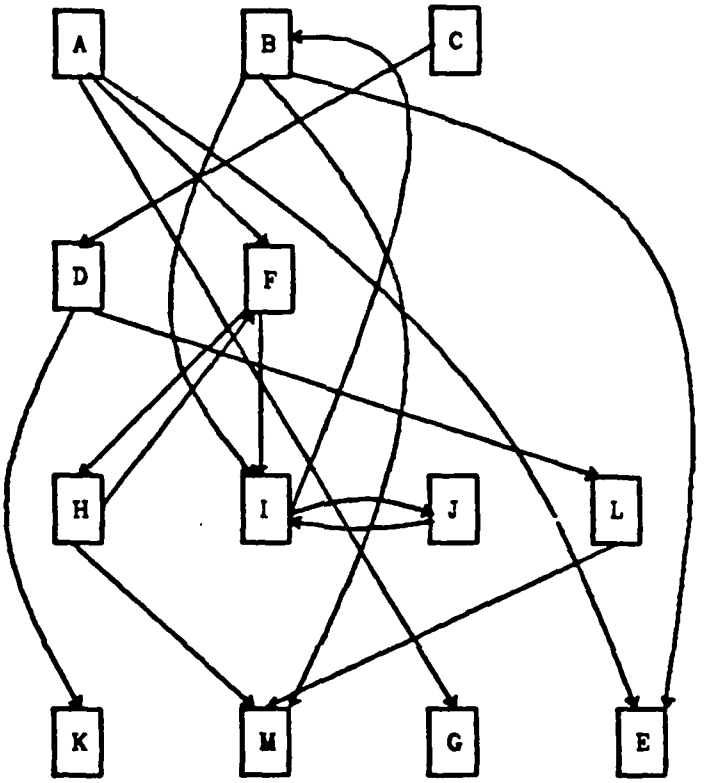

Figure 4.4. Sample graph with computed bottom level.

cessors, then that node and its successors must be in a cycle, since any successors of the node must also be its predecessors.

Once the bottom level has been determined, the next level is found by removing the nodes in the bottom level from the graph and recomputing the bottom level. This is repeated until the entire graph is partitioned into levels. The resulting level-partioned graph is shown in Figure 4.5.

Once the graph levels have been determined, the cycles within each level (caused by same-level arcs) must be collapsed.

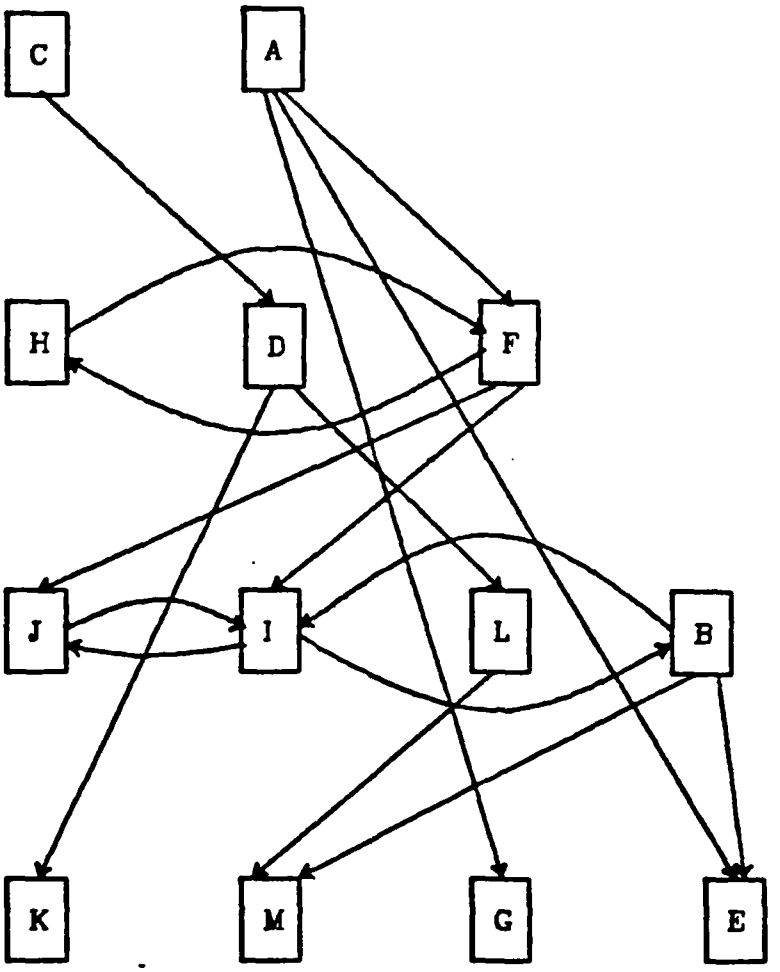

Figure 4.5. Sample graph partitioned into levels.

4.1.1.2. Collapsing Cycles

In order for the graph to be a proper hierarchy, the successors of each node on each level must be in a lower level (i.e. arcs must flow from the top of the hierarchy to the bottom). As was described above, some of the nodes on the levels may have successors on their same level. These nodes are said to be in a *cycle*. Each cycle on each level is replaced with a *proxy*. A proxy is a new node composed of the nodes in a cycle. For example, levels two and three in figure

4.5 each have cycles. Figure 4.6 displays the cycles collapsed into proxies. Proxies are named with a comma-separated list of the names of the nodes which they represent. By removing arcs within a level, proxies serve to simplify the arc crossing and node positioning heuristics.

Once the nodes have been partitioned into levels and the cycles on each level have been collapsed, the transitive closure of the graph is no longer

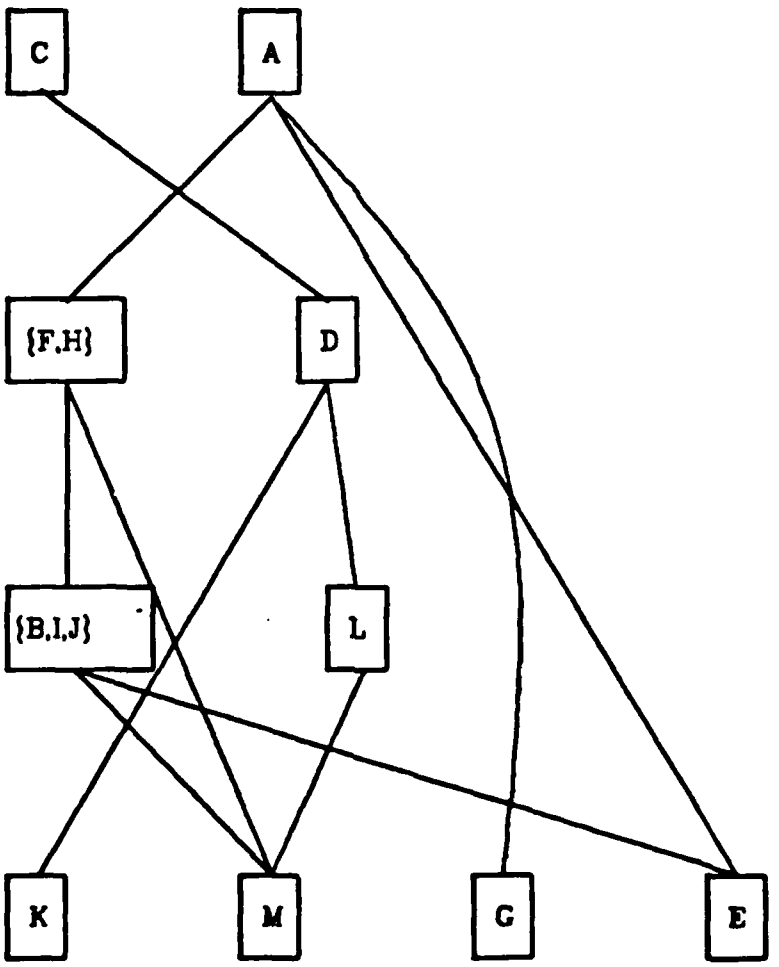

Figure 4.6. Figure 4.5 with cycles collapsed into proxies.

needed. The closure is "undone" by restoring the original arcs in the graph. The next step in the proper hierarchy algorithm is the reduction of long spans with the addition of dummy nodes.

4.1.1.3. Reducing Long Spans

The proper hierarchy constraint also requires each arc to flow from a higher level to *the next lower level*. This is accomplished by adding *dummy nodes* at each intermediate level of a long arc. A dummy node acts as a tie point for a long arc, and simplifies the arc crossing and node positioning heuristics. Dummy nodes are not drawn on the display, but appear as inflection points for long arcs. For example, the graph in figure 4.6 has long spans from nodes A to G, A to E, {F,H} to M and D to K. The result of adding dummy nodes is shown in figure 4.7. Note that the A to G long span was resolved by moving node G up to the second level rather than adding intermediate dummy nodes. This was possible since G has no successors.

The reduction of long spans concludes the proper hierarchy algorithm. In summary, the three parts of the proper hierarchy algorithm are:

- The original graph is partitioned into levels such that arcs either flow from a higher level to a lower level or to nodes on the same level.
- Cycles caused by same-level arcs are collapsed into new nodes called proxies.

and

- Arcs which traverse more than one level are reduced by adding new nodes called dummy nodes to each intermediate level.

In the next section we describe the graph drawing heristics for displaying a proper hierarchy and the readability constraints they must satisfy.

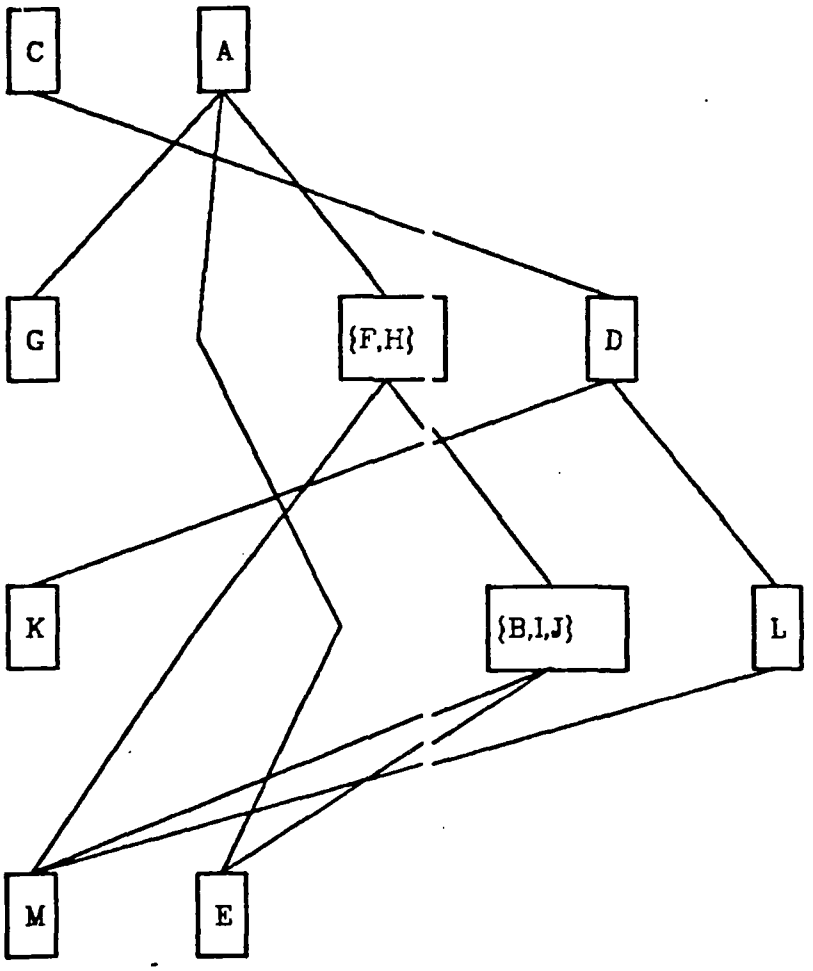

Figure 4.7. Figure 4.6 with dummy nodes added to shorten long spans.

4.1.2. The Graph Drawing Heuristics

The graph drawing heuristics are designed to satisfy several readability constraints:

- Arc crossings should be minimized
- Long lines should be as straight as possible
- Connected nodes should be positioned close to each other (i.e. paths should be short)

- Incoming and outgoing arcs of a node should be positioned in a balanced fashion

These constraints are satisfied by reducing the number of arc crossings on each level and properly positioning the nodes of each level on the display.

4.1.2.1. Minimization of Arc Crossings

In figure 4.7 we see that the graph has a total of ten arc crossings. This tends to make arc labels unreadable and arc paths difficult to follow. Arc crossings are reduced by a heuristic algorithm called the *barycentric method*. Each node in the hierarchy has an *up* and a *down* barycenter. The barycenter of a node corresponds to the best position for the node, according to the positions of the node's predecessors (for the up-barycenter) or successors (for the down-barycenter). Each node on a level has a position corresponding to its order on the level and the width of its icon. The up-barycenter of a node is computed as the average position of its predecessors (i.e. the sum of each position divided by the total number of predecessors). Similarly, the down-barycenter of a node is the average position of its successors. Note that since nodes on the first level have no predecessors, they do not have up-barycenters. Similarly, nodes on the last level do not have down-barycenters.

The barycentric algorithm begins by computing the up and down-barycenter of each node in the hierarchy. Figure 4.8 shows our sample graph with the computed up and down-barycenters for each level. As a reference, a labeled x-axis is provided at the top and bottom of the figure. The barycenters for dummy nodes are also displayed. The up-barycenter is printed above each node, and the down-barycenter below. For example, node D has an up-barycenter of 8.0 and a down-barycenter of 39.0. This indicates that, given the current positions of D's predecessors, the best position for D would be at 8.0 (i.e. $x = 8.0$) However, the current position of D's successors indicate that D would best be positioned at 39.0. This conflict is resolved using a heuristic

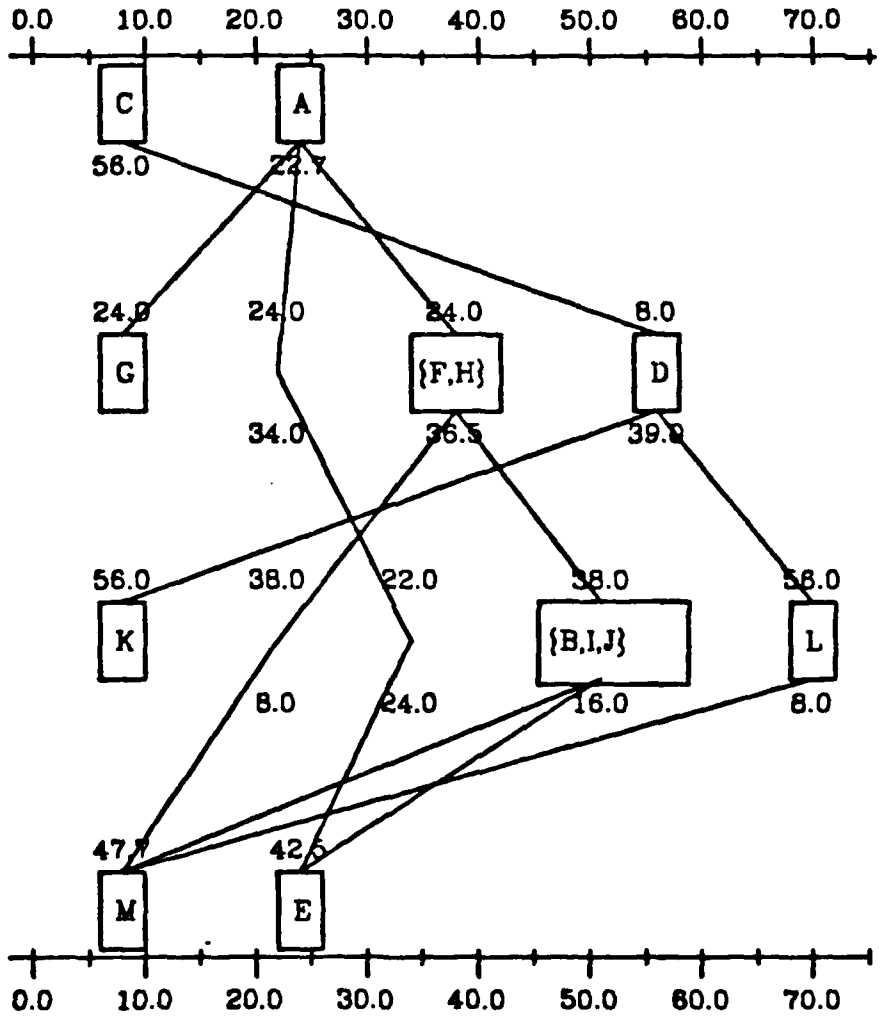

Figure 4.8. Sample graph with computed up and down barycenters.

which makes two passes over the hierarchy.

In the first pass, the nodes on levels 2..n (for an n-level hierarchy) are sorted by up-barycenters. Figure 4.9 displays the sample graph after sorting the second level by up-barycenters. Here we see that node D has been moved to the first position on level two. The sorting by up-barycenters has reduced the

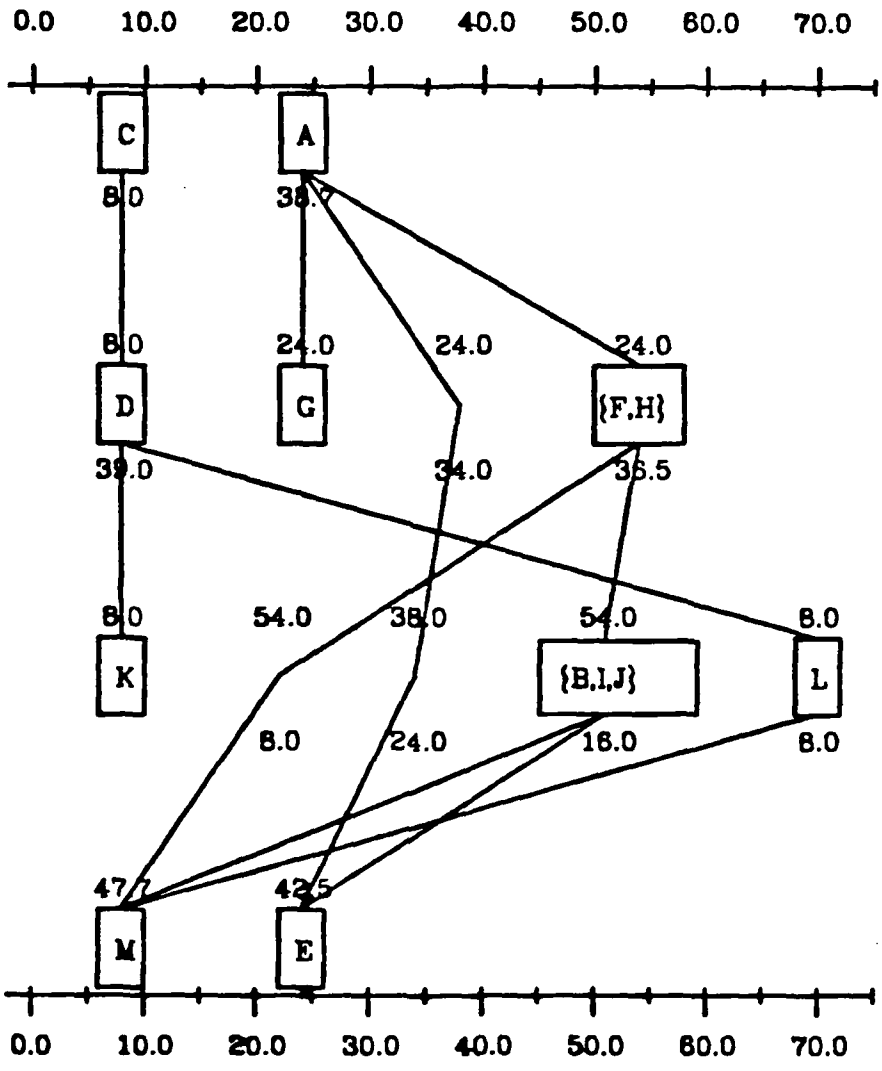

Figure 4.9. Sample graph after sorting level two by up-barycenters.

number of crossings in the graph from ten to seven. Note that sorting a particular level will change the down-barycenters of the previous level and the up barycenters of the next level. Figure 4.10 illustrates the result of sorting the remaining levels by up-barycenters. From this we see that after the first pass

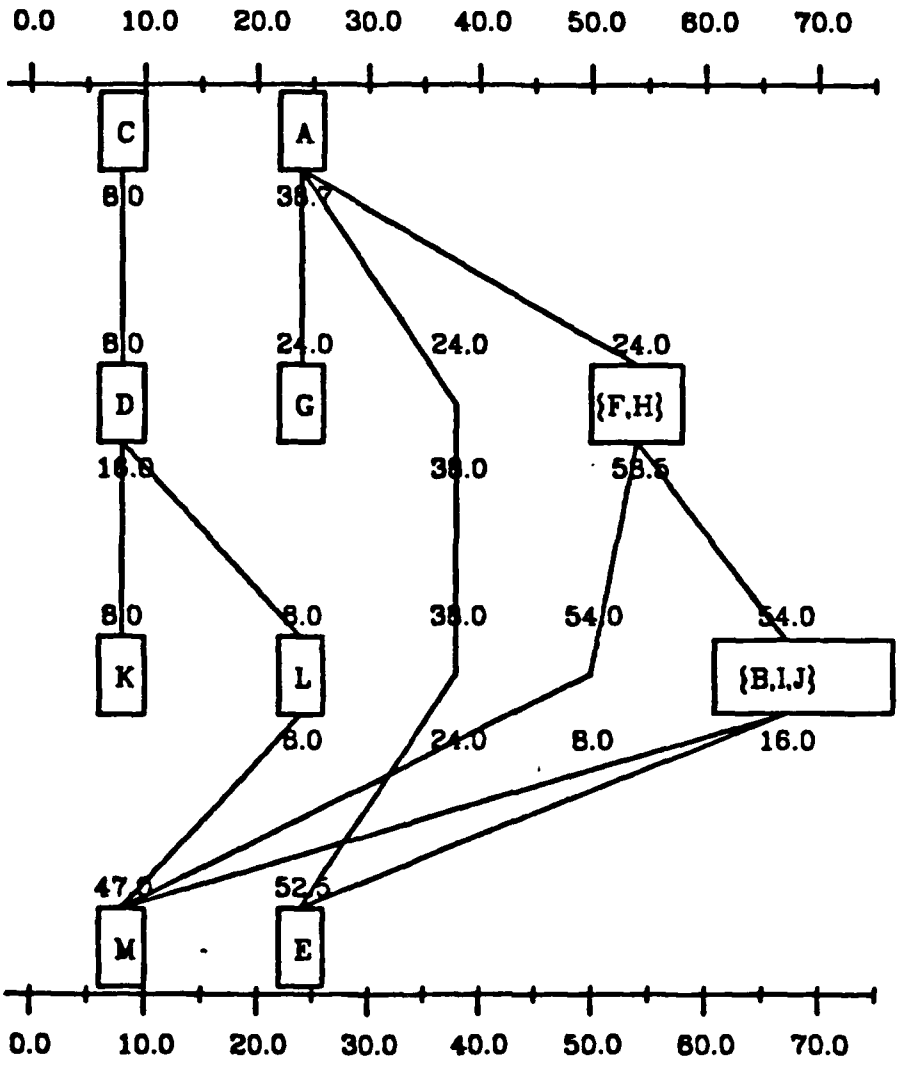

Figure 4.10. All levels sorted by up-barycenter.

we have reduced the total number of crossings from ten to two.

In the second pass, we attempt to further reduce the number of arc crossings by sorting levels n-1..1 by down-barycenters. Figure 4.11 illustrates the results of this pass.

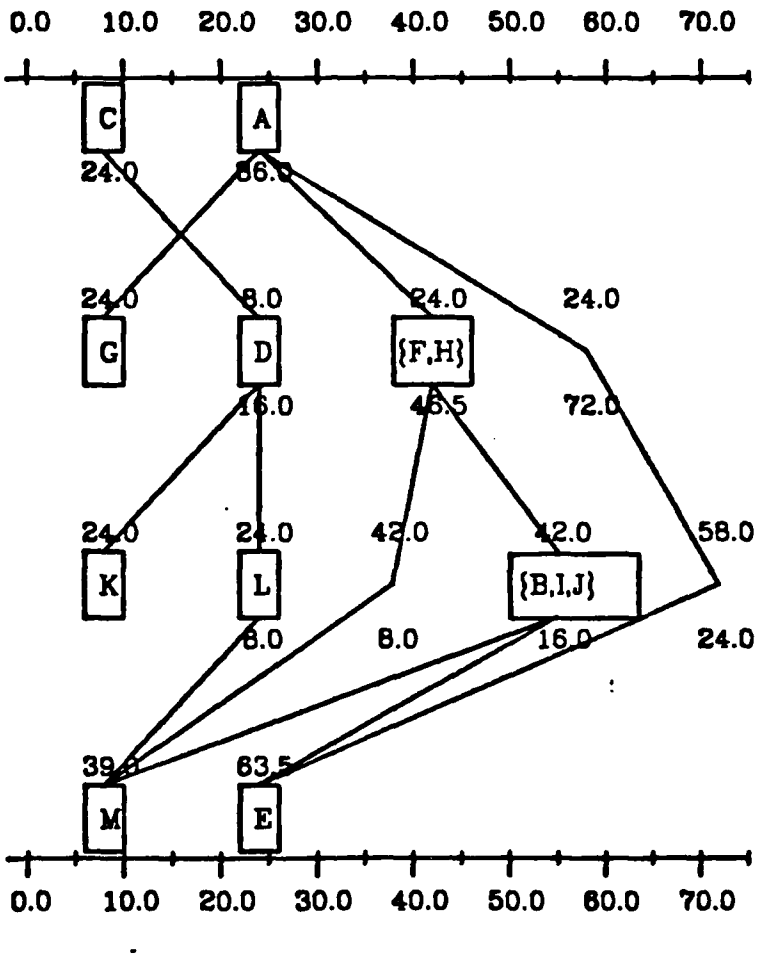

Figure 4.11. All levels sorted by down-barycenter.

When sorting the nodes on a level, the original order of nodes with equal barycenters is maintained. A separate pass is made to reverse the order of the equal-barycenter-runs. This refinement in the algorithm reduces the number of arc crossings in some cases.

After each pass through the hierarchy, the total number of arc crossings is computed. The up and down passes over the hierarchy are repeated until either an acceptable number of arc crossings is attained or a maximum number of

iterations is reached. Currently, the maximum number of iterations is set to five. Initially, the algorithm is invoked with zero as the only acceptable number of arc crossings. If, after iterating five times, more than zero crossings remain, the iteration is repeated until the minimum number of crossings seen is attained. Figure 4.12 displays the final ordering for each level of the sample graph. The number of arc crossings were reduced from ten to zero after three passes over the hierarchy.

Once the nodes in the hierarchy have been positioned on each level, the spacing between nodes on a level must be determined.

4.1.2.2. Positioning the Nodes

The nodes on each level are positioned on the display using an algorithm called the *priority positioning method*. Each node in the hierarchy has an *up* and a *down* priority. The priority of a node corresponds to its upward or downward connectivity. In other words, the up priority of a node is the number of predecessors it has and the down priority is its number of successors. The dummy nodes (i.e. tie points for long arcs) are given the highest priorities so as to keep long arcs as straight as possible. Figure 4.13 shows the sample graph with the computed up and down priorities and barycenters of each node.

The positions of the nodes on each level are improved by placing the nodes as close as possible to their up or down-barycenter. Since the up or down-barycenters may not be the same, two passes are made over the hierarchy.

In the first pass, the nodes in levels 2..n (for an n-level hierarchy) are positioned by their up barycenters. In this pass the nodes with the highest up priority on each level are positioned first. Once a node has been positioned, it cannot be moved by a lower priority node. From figure 4.13 we see that the dummy node has the highest up priority on level two. Nodes D and G are shifted left so as to position the dummy node at its up-barycenter, as shown in figure

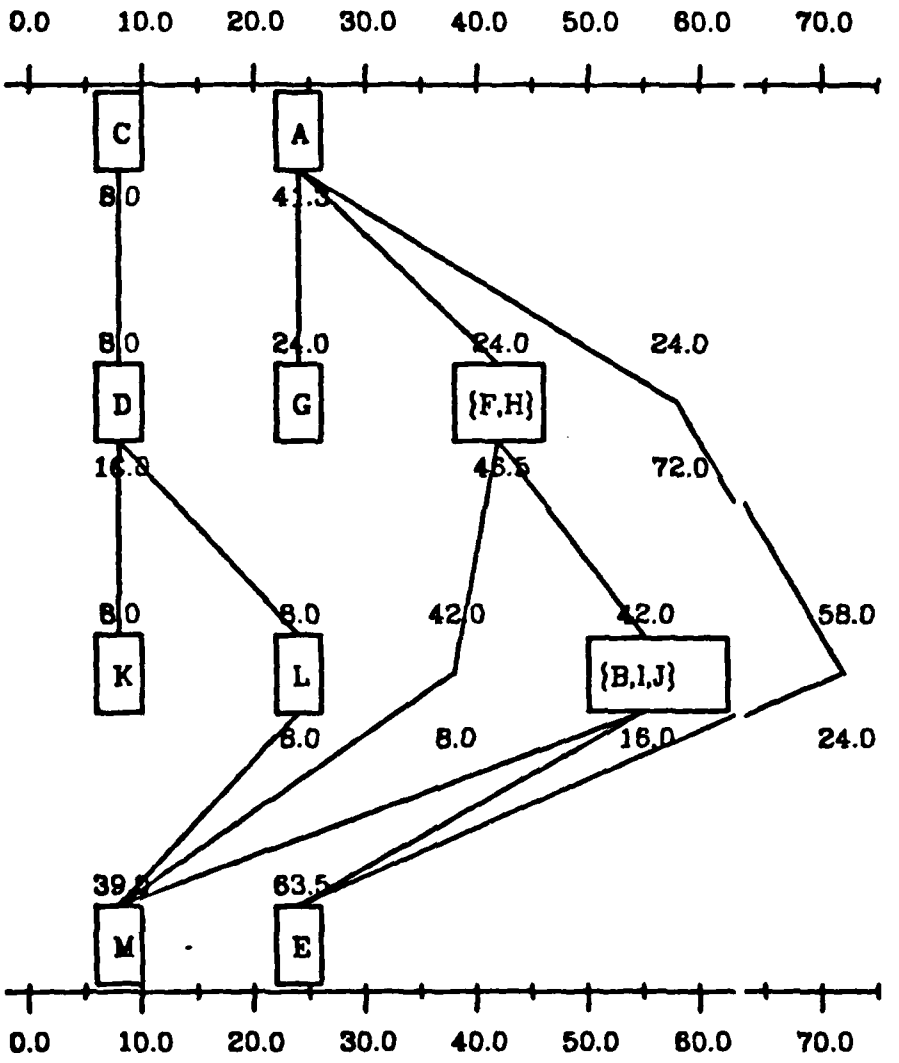

Figure 4.12. Final ordering for each level.

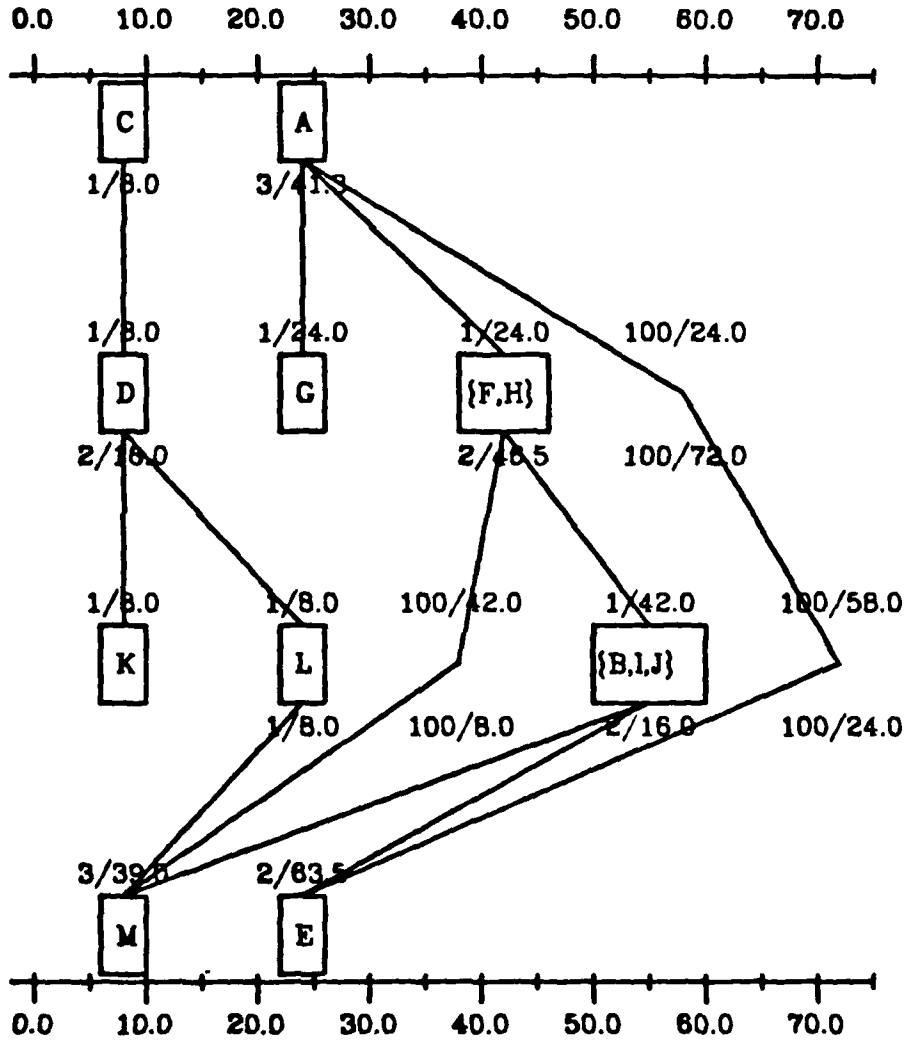

Figure 4.13. Sample graph with computed up and down priority/barycenter for each node.

4.14. Since only the up priorities and barycenters of level two are significant here, the other priorities and barycenters are not shown. From figure 4.14 we can see that the dummy node's new position allows for a straight arc between A and the dummy node. The remaining nodes on level two have equal up-priorities and each would like to be placed at position 24.0 (i.e. each have up-barycenters

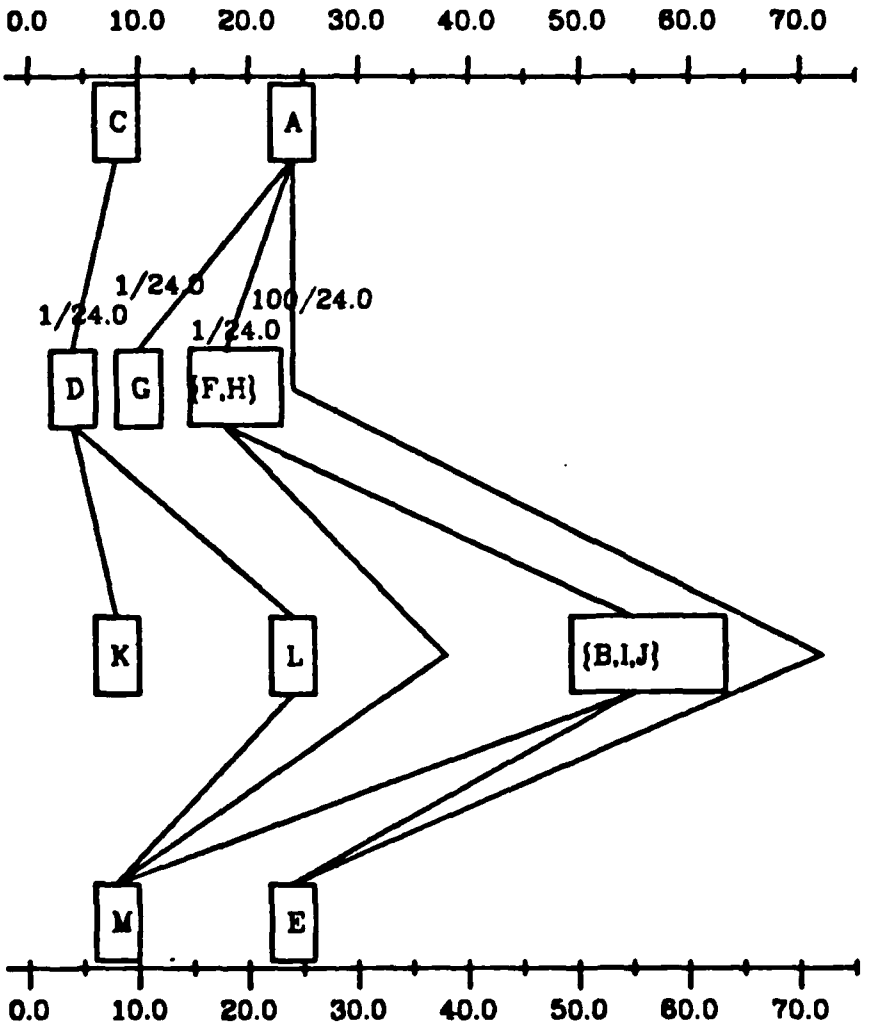

Figure 4.14. Dummy on level two positioned by up-barycenter.

of 24.0). Since the dummy node has already occupied position 24.0, the positions of nodes D, G, and {F,H} cannot be changed. Note that the repositioning of a node will change the down-barycenters of the nodes on the previous level and the up-barycenters of those on the next level. However, the priority of a node does not change, since the number of incoming and outgoing arcs remains the

same. Figures 4.15, 4.16, and 4.17 display the sample graph after positioning the remaining nodes according to their up priorities and up-barycenters.

In the second pass, the positions of the nodes in levels n-1..1 are determined by their down-barycenters, positioning the nodes with the highest down priority on each level first. Figures 4.18, 4.19, and 4.20 show the sample graph during this pass.

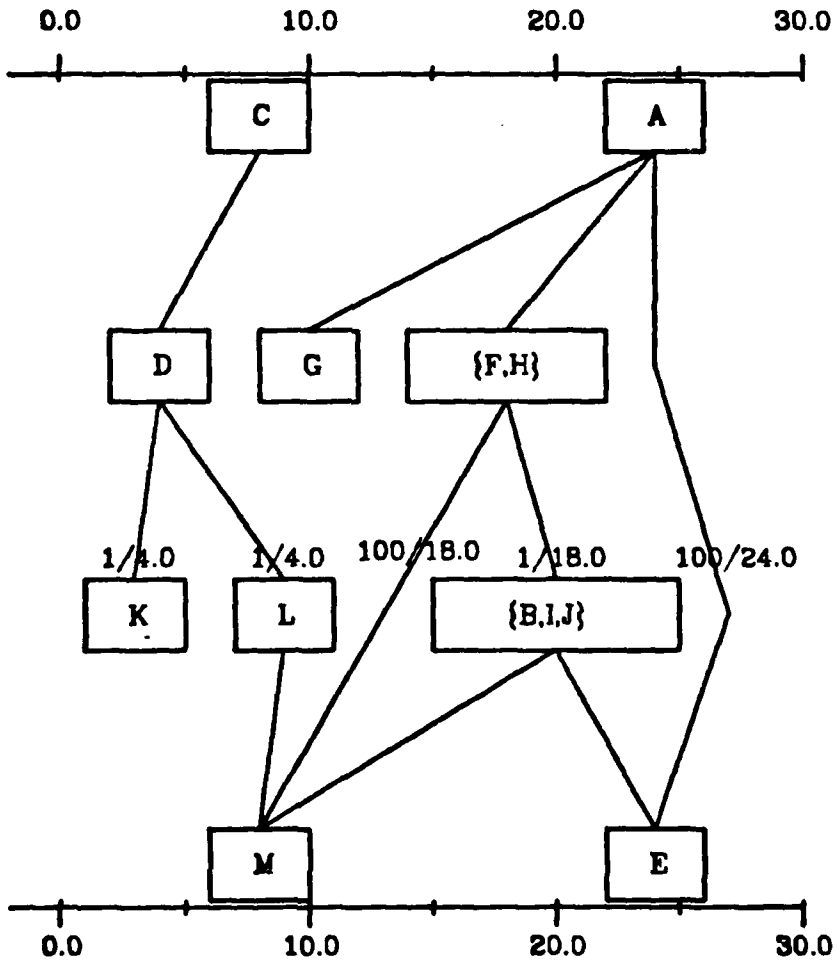

Figure 4.15. Positioned rightmost dummy node on level three by up-barycenter.

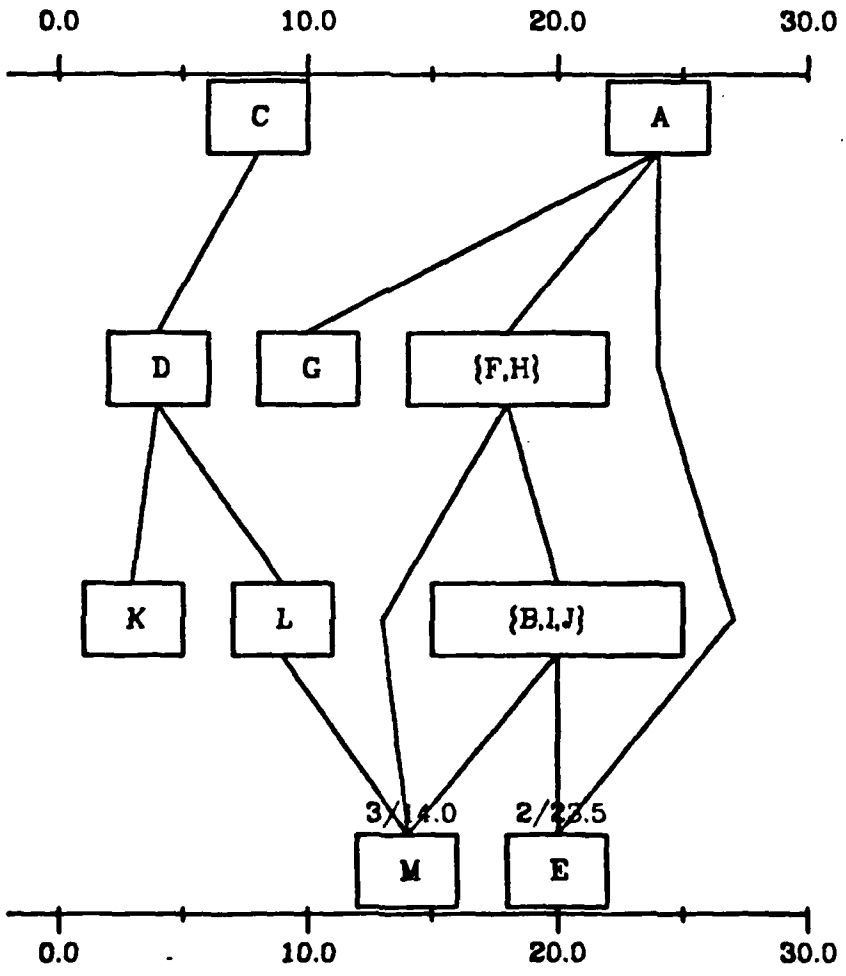

Figure 4.16. Positioned M by up-barycenter.

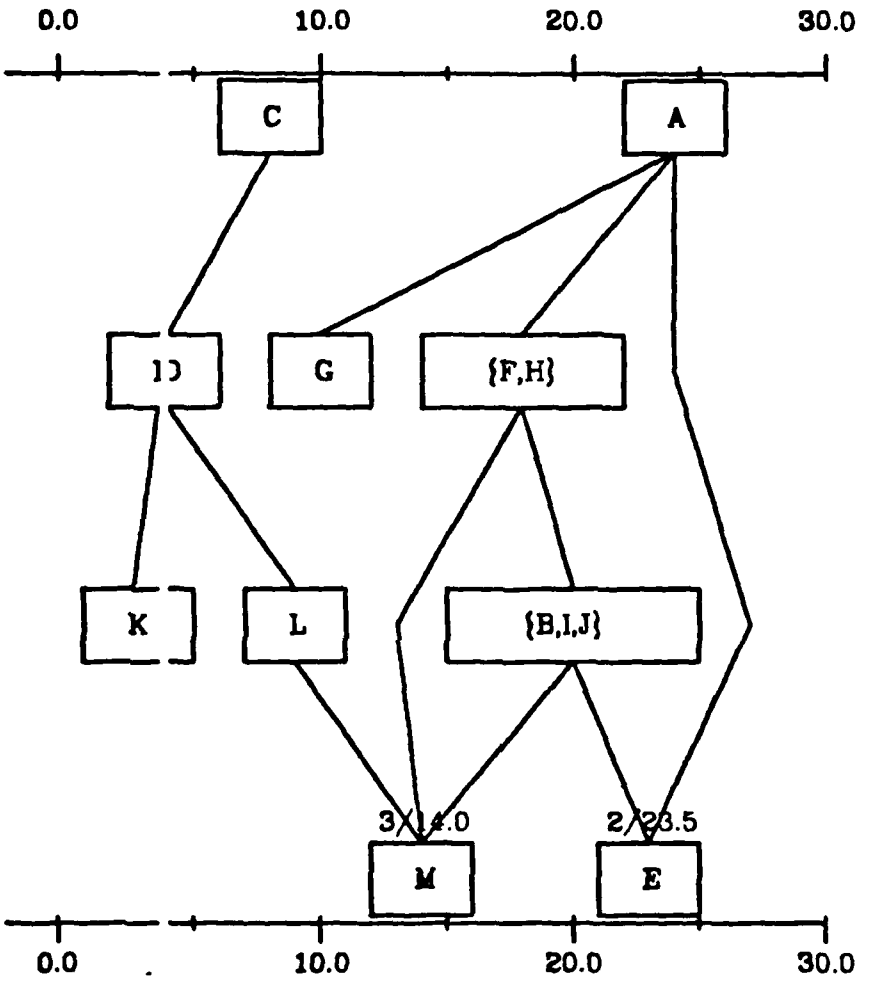

Figure 4.17. Positioned E by up-barycenter.

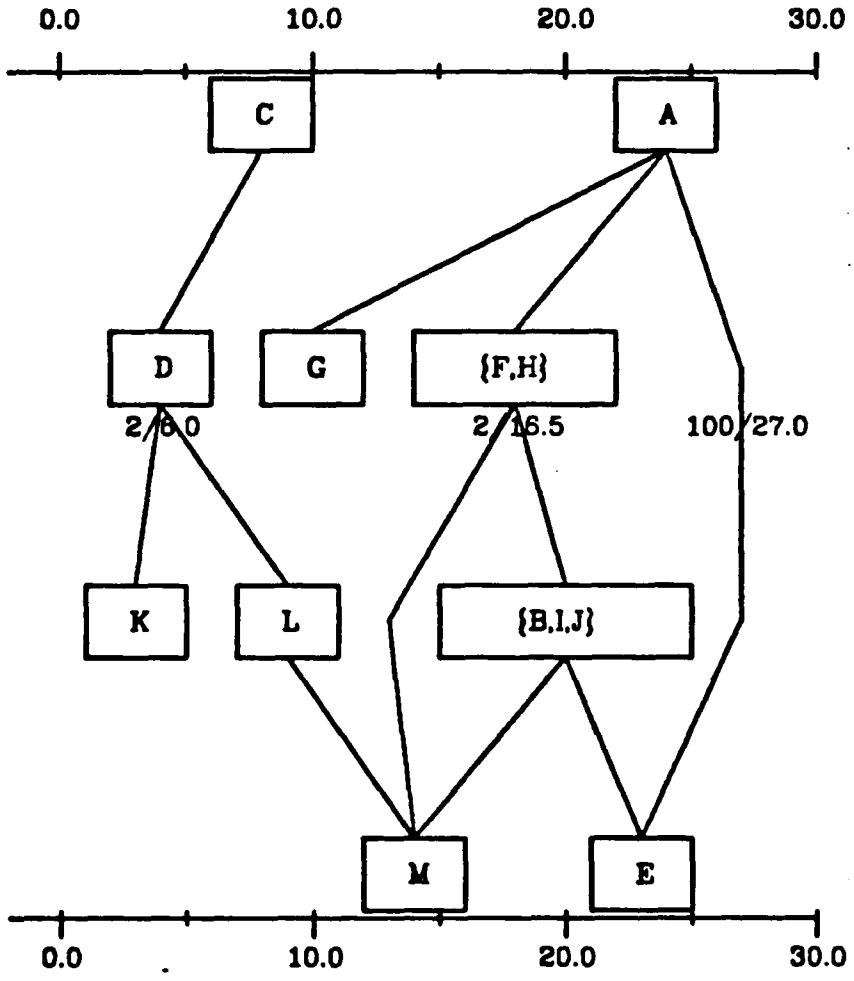

Figure 4.18. Positioned dummy node on level two by down-barycenter.

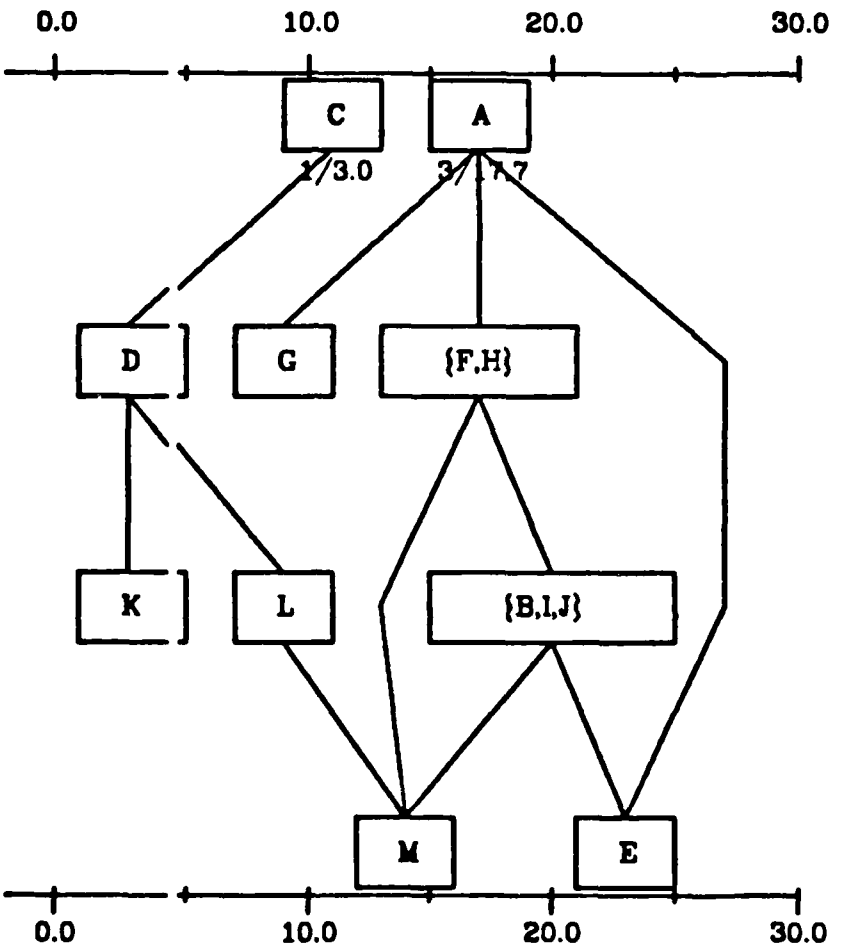

Figure 4.19. Positioned A by down-barycenter.

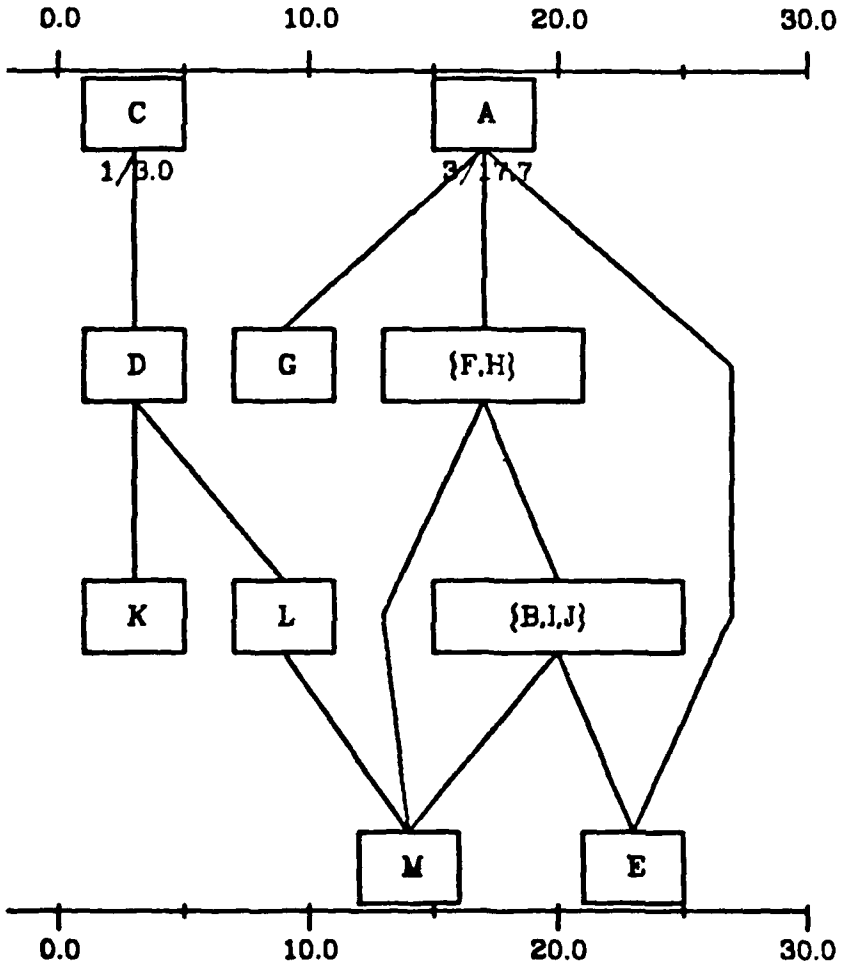

Figure 4.20. Positioned C by down-barycenter.

In a final pass, the nodes in levels (n/2)..n are again positioned according to their up-barycenters, using their up priorities. This final pass tends to favor the up-barycenters for the lower half of the hierarchy, as shown in figure 4.21.

A recap of what we have done:

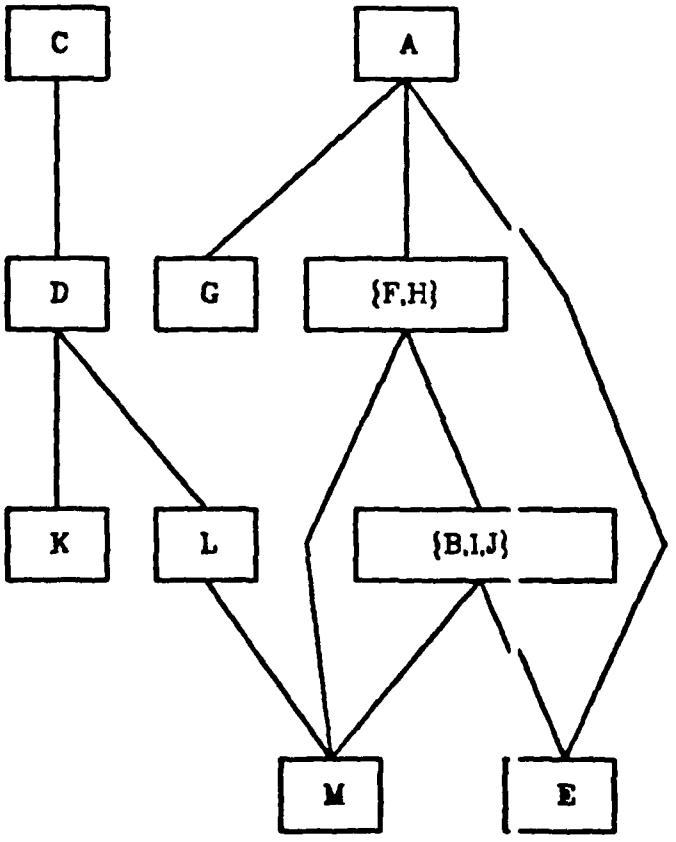

Figure 4.21. Final GRAB layout of sar mple graph.

- Arc crossings were minimized by permuting the nodes on each level of the hierarchy.
- Arcs were straighted by positioning the nodes on each level according to their up and down berycenters and connectivities.

Thus figure 4.21 represents the final output of the graph drawing algorithm. By comparing the original hand-drawn graph (figure 4.1) to the GRAB-drawn graph (figure 4.21) we see a striking difference in readability. The minimization of arc

crossings combined with the closeness of connected nodes in the GRAB-drawn graph help to clearly convey the relationships depicted by the graph. In addition, the hierarchical layout of the graph allows one to quickly perceive the degree of dependencies of each node (e.g. nodes on the first level are least depended on, while those on the bottom level are most depended on).

In the next section we look at some of the short-comings of the graph-drawing algorithm and explore some possible solutions.

4.1.3. Problems with the Graph Drawing Algorithm

This algorithm has several problems. First, no provision is made to expand the proxies into their constituent nodes. This could be solved by drawing the proxy constituents around the circumference of a circle. For example, figure 4.22 illustrates the result of expanding the {F,H} and {B,I,J} proxies from figure 4.21. Thus proxy expansion can be accomplished simply by allowing room between adjacent levels and nodes for the resulting circular layout. However, placing the nodes around the circle so as to minimize arc crossings appears to be an additional complication. Another solution to the proxy expansion problem would be to note that each proxy can be divided into cycles of two, three, or more nodes. The optimal layout of the different size cycles can be precomputed and used to layout the constituents of each proxy.

A second problem with the graph drawing algorithm is its inability to handle same-level arcs (i.e. arcs from one node to another node on the same level). The restriction to down-level arcs tends to add more dummy nodes and levels to the graph. For example, in figure 4.21 we see that both nodes M and E were forced down one level because of same-level arcs from nodes L and {B,I,J}. If we allow same-level arcs in the hierarchy, we are left with the graph in figure 4.23. Here we see that the introduction of same-level arcs from L to M, {B,I,J} to M, and {B,I,J} to E has removed the need for a fourth level. Moreover, by moving M and E up one level, we have done away with two of the three dummy nodes in the

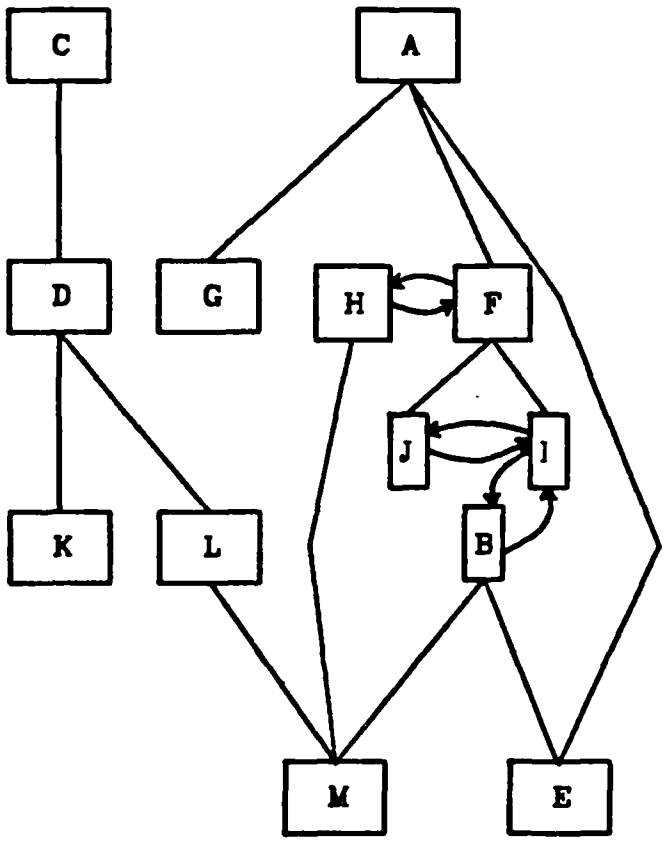

Figure 4.22. GRAB-drawn graph with expanded proxies.

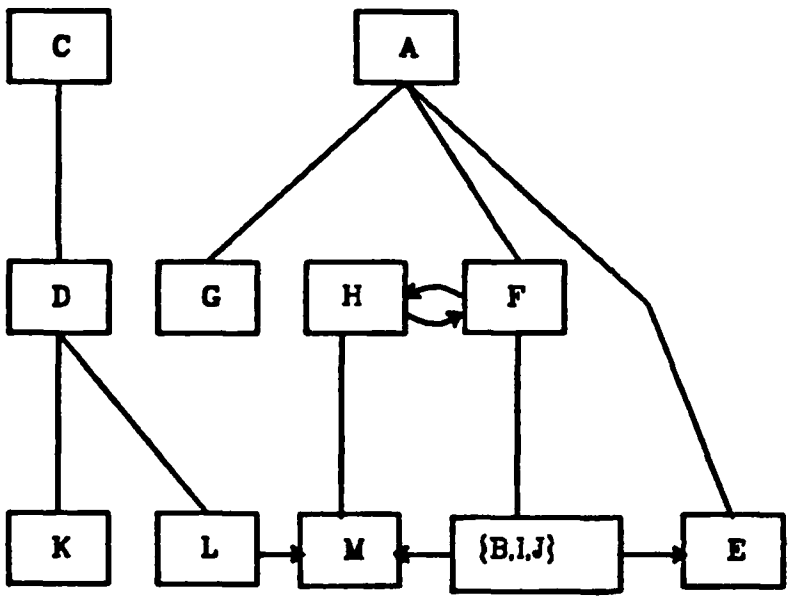

Figure 4.23. Figure 4.21 with same-level arcs.

graph.

An additional benefit of same-level arcs is that cycles with only two nodes need not be collapsed into proxies. For example, the proxy expansion of {F,H} in figure 4.22 simply involved the introduction of same-level arcs on the second level of the graph. If we recall that cycles were caused by a node having successors on the same level, we might be tempted to do away with proxies altogether

in favor of same-level arcs. However, it is clear that with cycles of more than two nodes, the circular layout provided by proxy expansion is superior to the flat layout provided by same-level arcs.

In the next section a prototype implementation of the graph drawing algorithm is discussed.

4.2. Implementation of the Graph Drawing Algorithm

In this section, a prototype implementation of the graph drawing algorithm will be summarized and its performance analyzed.

The graph drawing algorithm has been written in C on a VAX 11/780 running UNIX® 4.2 BSD. Currently, there is a total of approximately 2500 lines of code. Table 4.2 lists the performance of the algorithm on various size graphs. Several points of interest can be seen here. First, the sudden explosion in cpu time for the 10 node, 45 edge graph (340.51 seconds) is due to the nature of the graph: the graph is split into ten levels, one node per level. The edges are such that each node has an edge to each node in the levels below it. This results in a large number of dummy nodes generated to shorten the many long spans in the

Graph Algorithm Performance		
Nodes	Edges	User CPU Seconds
10	0	0.34
10	10	1.01
10	20	2.51
10	45	340.51
20	25	14.94
20	50	2.09
40	50	64.50
60	10	2.31
60	60	175.61
150	160	412.37

Table 4.2. Graph algorithm performance on various graphs.

graph. The net result is that the graph actually contains 215 nodes and 249 edges! The second surprise statistic is the extremely low cpu time for the 20 node, 50 edge graph (2.09 seconds). Further examination of this graph reveals that a large portion of the graph is in a cycle. Hence, most of the nodes were collapsed into a single proxy resulting in a drastic reduction in the number of edges to deal with. The statistics for two 60 node graphs (10 edges and 80 edges) clearly show that the running time is more closely related to the number of edges in the graph than the number of vertices.

As can be seen from this first experiment, the algorithm works reasonably well on small graphs. However, it is much too slow for large graphs. The next experiment examined the various phases of the algorithm to see which, if any, caused the performance problem.

Table 4.3 breaks the times from table 4.2 into the running time of each of the major parts of the graph drawing algorithm. Here we see that the majority of the time is spent executing the arc minimization heuristics. Further examination of these results has revealed that 90% of this time is spent dynamically

Graph Algorithm Performance Breakdown					
Nodes	Edges	Proper Hierarchy	Minimize Crossings	Position	Total
10	0	0.14	0.07	0.13	0.34
10	10	0.53	0.13	0.35	1.01
10	20	1.82	0.40	0.29	2.51
10	45	1.46	335.27	3.78	340.51
20	25	0.94	12.99	1.01	14.94
20	50	1.58	0.15	0.36	2.09
40	50	2.48	59.77	2.25	64.50
60	10	0.65	0.65	1.01	2.31
60	80	2.13	170.77	2.71	175.61
150	160	5.62	399.94	6.81	412.37

Table 4.3. Breakdown of CPU time.

computing the number of arc crossings in the graph. This time could be greatly reduced by adding some bookkeeping to the arc minimization heuristics to keep track of changes in the number of arc crossings after each pass over the graph.

This second experiment indicates that modifications in the implementation can yield significant improvements in the graph drawing algorithm. Nevertheless, the algorithm is probably too slow for real-time use. Users are not likely to tolerate waiting several minutes to display a 1000 node graph.

One solution to this problem is to recognize that for very large graphs, only a portion of the graph can fit on the display screen (i.e. for the node and arc labels to be readable, the graph must be clipped). Thus, the time spent running the graph drawing algorithm could be greatly reduced by identifying the portion of the graph to appear on the display and running the algorithm on that portion of the graph. Once the visible portion of the graph has been drawn, the graph drawing algorithm can continue to run in the background on the remainder of the graph.

A second solution is to compute the entire graph layout and save it in the database so that subsequent executions of the browser on the graph do not have to recompute the layout. The provision of an explicit *redraw graph* command allows users to rerun the graph drawing algorithm when necessary.

Chapter 5

Conclusion

This chapter summarizes the major features of GRAB.

The many uses of graphs in computer science provide a wide application base for a general purpose graph browser. An early prototype of such a graph browser has shown that:

- graphs should be presented in the usual fashion
- arcs and arc labels should be drawn
- convenient browsing and editing operations are required

Taking these features into account, we have proposed a design for GRAB, a general purpose graph browser.

GRAB will allow users to browse graphs stored in a relational database. It will use a user-friendly window-mouse paradigm. The combination of a graphical presentation of a directed graph and the mouse as a pointing device will allow users to browse large graphs conveniently.

The database schema used by GRAB allows users to specify a map from the user's database schema so that the interface can be used for various kinds of data without much effort.

Finally, GRAB will draw graphs in a systematic form know as a proper hierarchy. In this form, graphs are partioned into levels, with arcs directed from an upper level to the next lower level. Long arcs which traverse more than one level are shortened with the addition of dummy nodes at each intermediate level. Cycles within levels are collapsed into new nodes called proxies. Heuristics are used to minimize the arc-crossings in the graph and position the nodes on each level. The net result is a reasonable-looking graph which clearly

conveys the intended entity-relationships.

The proper hierarchy layout of graphs does have a few problems. The major problem is that no provision is made for the expansion of the proxies in the graph. We have seen that this could be solved by either expanding proxy constituents around the circumference of a circle or by precomputing a layout for proxies with a limited number of nodes. In addition, the introduction of same-level arcs to the hierarchy can reduce the number of levels and dummy nodes in the graph.

In its current implementation, the performance of the graph drawing algorithm deteriorates somewhat for large graphs. A major part of this is caused by a lack of bookkeeping in the arc minimization heuristics.

In order improve the performance of GRAB on large graphs, some consideration will be given to a precomputation and incremental update scheme. This would involve precomputing the proper hierarchy and layout for a graph and updating this computation whenever the graph is changed. The tradeoff between time spent running the graph drawing algorithm anew and that spent updating an old computation has yet to be examined.

References

[Apple83] *Lisa Owner's Guide*. Sep. 1983.

[Feldman79] S. I. Feldman, "Make -- A Program for Maintaining Computer Programs," *Software -- Practice and Experience* 9 pp. 255-265 (1979).

[Sugiyama81] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures.," *IEEE Transactions on Systems, Man, and Cybernetics* SMC-11(2) pp. 109-125 (Feb. 1981).

[Warfield76] J. N. Warfield, *Societal Systems*, John Wiley & Sons, Inc. (1976).