INTERACTIVE GRAPH LAYOUT: THE EXPLORATION OF LARGE GRAPHS

(Ph.D. Dissertation)

Tyson Rombauer Henry

TR 92-03

June 30, 1992

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

This work was supported in part by the National Science Foundation under grants IRI–8702784, CDA–8822652, and IRI–9015407.

INTERACTIVE GRAPH LAYOUT: THE EXPLORATION OF LARGE GRAPHS

by

Tyson Rombauer Henry

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 2

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: _____

ACKNOWLEDGMENTS

I wish to thank the following people who have all helped me complete my Ph.D.

My advisor Scott Hudson for teaching me how to do research. Scott has provided me with both the freedom to develop my own research and research style and the guidance to keep me moving forward. He has been a good friend in addition to a good advisor—he made it possible to have fun along the way.

The University of Arizona graphic user interface research group, Chenning Hsi, Shamim Mohamed, Gary Newell, Robert Simms, and Andrey Yeatts, for their valuable feedback and for providing the forum in which I presented and debugged the ideas that grew into this dissertation.

Curtis Dyreson, Shamim Mohamed, Vic Thomas, Nick Kline, Mike Soo, Mark Abbott, Patrick Homer, and Shivakant Mishra, for thousands of lunches, countless trips into the back country, mountain bike rides in the desert, and especially for 5 years of spending Thanksgiving in the Chiricahau Wilderness Area—a tradition that helped me keep graduate school in perspective.

Herman Rao for talking me into getting a PhD—"Title is for whole life"—and for all his support and tremendous encouragement from my first day of graduate school to my last. Shamim Mohamed for answering thousands of systems questions saving me hundreds of hours looking up little details. Andrey Yeatts for listening to new ideas, proof reading old ideas and teaching me how to be politically aware. Robert Simms for arriving in the department one week before me, for his encouragement, and for pointing out that the first person that fails the prelims or defense will be immortalized in the department history book. Phyllis Pradd for always listening and providing on-site support. Vic Thomas, Shamim Mohamed and Peter Bigot for holding down TEX while I attempted to make it to things it did not want to—I guess now I will have to buy and *read* the book.

My major committee members, Larry Peterson and Rick Snodgrass for their help in developing these ideas and their contributions to the dissertation. Rick Schlichting for filling in when my advisor left Arizona and for telling me to "Get to work" every time he has seen me during the last 2466 days. Russ Ferrell for introducing me to a different view of human computer interactions and for sitting on my minor committee. Suvrajeet Sen for sitting on my minor committee.

The NSF for their financial support under grants IRI–8702784, CDA–8822652, and IRI–9015407 and the department for filling the financial gaps between NSF grants and providing the office space and computing equipment that made this research possible. The department lab and office support staff for helping with the hardware details and the office paperwork.

Finally I want to thank Ann Parker for her great encouragement, support and understanding during these last two year—the most difficult and grueling years of the entire Ph.D. process. She clearly helped me over these last final peaks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Directed and undirected graphs provide a natural notation for describing many fundamental structures of computer science. Unfortunately graphs are hard to draw in an easy to read fashion. Traditional graph layout algorithms have focused on creating good layouts for the entire graph. This approach works well with smaller graphs, but often cannot produce readable layouts for large graphs.

This dissertation presents a novel methodology for viewing large graphs. The basic concept is to allow the user to interactively navigate through large graphs, learning about them in appropriately small and concise pieces. The motivation of this approach is that large graphs contain too much information to be conveyed by a single canonical layout. For a user to be able to understand the data encoded in the graph she must be able to carve up the graph into manageable pieces and then create custom layouts that match her current interests.

An architecture is presented that supports graph exploration. It contains three new concepts for supporting interactive graph layout: interactive decomposition of large graphs, end-user specified layout algorithms, and parameterized layout algorithms.

The mechanism for creating custom layout algorithms provides the non-programming end-user with the power to create custom layouts that are well suited for the graph at hand. New layout algorithms are created by combining existing algorithms in a hierarchical structure. This method allows the user to create layouts that accurately reflect the current data set and her current interests.

In order to explore a large graph, the user must be able to break the graph into small, more manageable pieces. A methodology is presented that allows the user to apply graph traversal algorithms to large graphs to carve out reasonably sized pieces. Graph traversal algorithms can be combined using a visual programming language. This provides the user with the control to select subgraphs that are of particular interest to her.

The ability to Parameterize layout algorithms provides the user with control over the layout process. The user can customize the generated layout by changing parameters to the layout algorithm. Layout algorithm parameterization is placed into an interactive framework that allows the user to iteratively fine tune the generated layout.

As a proof of concept, examples are drawn from a working prototype that incorporates this methodology.

CHAPTER 1
INTRODUCTION

Graphs[1] are a fundamental structure in computer science. They are particularly well suited for representing sets of objects and the relationships between them. Graphs thus provide a very natural notation for many areas of computer science.

Data base schemas [11], for example, can be described using graphs. Visual programming notations [53] rely heavily on graphs—viewing visual programs is a similar problem to viewing general graphs. Hypertext navigation maps [22, 57] can also be modeled with connected graphs. Petri nets [56], pert charts, and schematic diagrams are all graph based.

Not only do graphs provide a good data structure for certain types of data, they also provide a powerful visual notation [13, 31, 33, 35, 50]. This is because graphs can simultaneously convey many relationships and people are good at interpreting graphs—a graph may also be worth a thousand words.

Even though graphs provide a powerful visual notation, it is difficult to create a "good" presentation of a general graph. A graph does not contain any information about placement of its nodes and edges. It consists only of a set of nodes and a set of edges. In order to draw a graph, screen coordinates must be assigned to all the nodes and edges. This task of calculating these coordinates is called *graph layout.*

Arbitrarily positioning the nodes of a graph will produce a *legal* layout of the graph, but an arbitrary positioning of nodes would produce a layout that would be very difficult to read. Thus the goal of graph layout is to generate readable layouts of graphs. The limited resources of screen space, screen resolution, and the limits of the user's cognitive power make this a difficult goal to achieve.

1.1 Traditional Graph Layout Algorithms

Traditional graph layout algorithms have tried to increase the readability of graphs by focusing on improving specific graph properties [16, 63]. The following list contains several goals for improving generated layouts.

- Avoid edge crossings
- Maximize display symmetry
- Avoid bends in edges
- Keep edge lengths uniform
- Distribute vertices uniformly

[1] A directed graph G is defined to be a pair $(V(G), E(G))$, where $V(G)$ is a non-empty finite set of elements called vertices or nodes, and $E(G)$ is a finite set of ordered pairs (not necessarily distinct) of elements of $V(G)$ called edges [67]. Undirected graphs differ from directed graphs in that edges are *unordered* pairs of nodes.

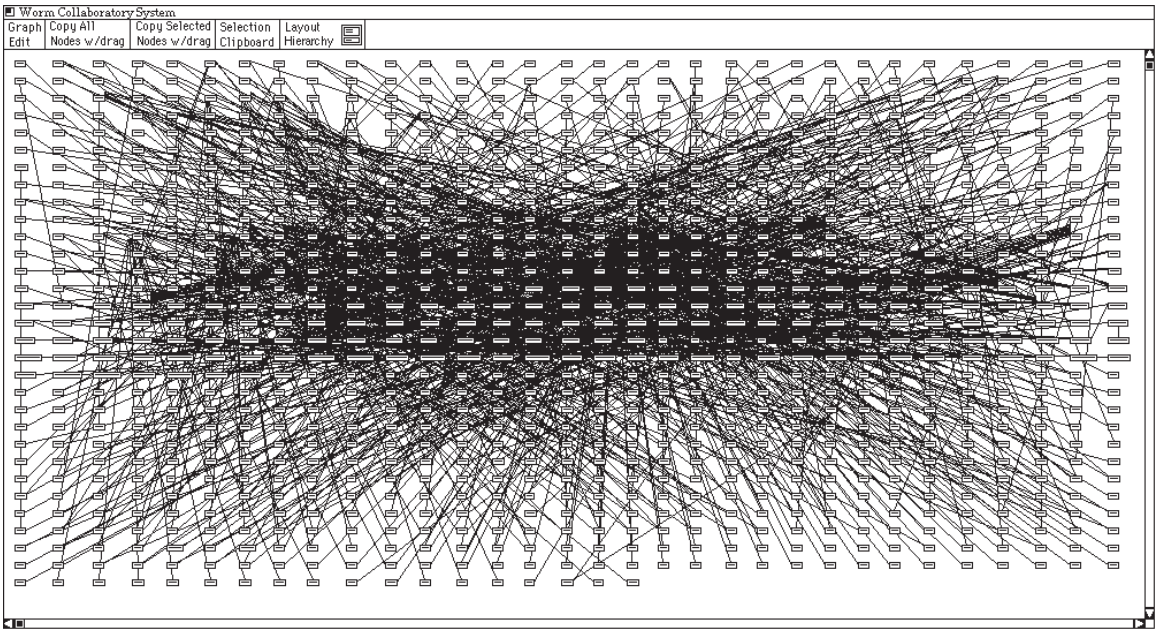

Figure 1.1: Single View of a Large Graph

Focusing on these kinds of aspects tends to be very computationally difficult. For example, minimizing the edge crossings in a general graph is NP-complete [19, 30]. There are however many approximate solutions that use heuristics to satisfy these properties and generate good layouts in polynomial time [16].

Nonetheless, even when traditional layout algorithms do a good job satisfying such goals, they do not always produce the most readable graphs or make the best use of available resources and thus don't always produce the "best" layout. There are two main factors that lead to the shortcomings of traditional algorithms.

The first factor is that these algorithms do not always scale well. It may not be possible to meet any of the above goals for very large or complicated graphs (e.g. one with a high degree of connectivity). The basic problem is that too much data is encoded in large and complicated graphs to be presented in a single generated layout. Figure 1.1 shows a naive layout of a large graph for which traditional layout algorithms may not be able to generate a good layout.

The second factor is that the "best" layout depends on what information the user is currently focused upon. Traditional layout algorithms do not take into account the user's current interests. Consequently, traditional algorithms must compromise by creating a single canonical layout that attempts to satisfy the goals and interests of all users.

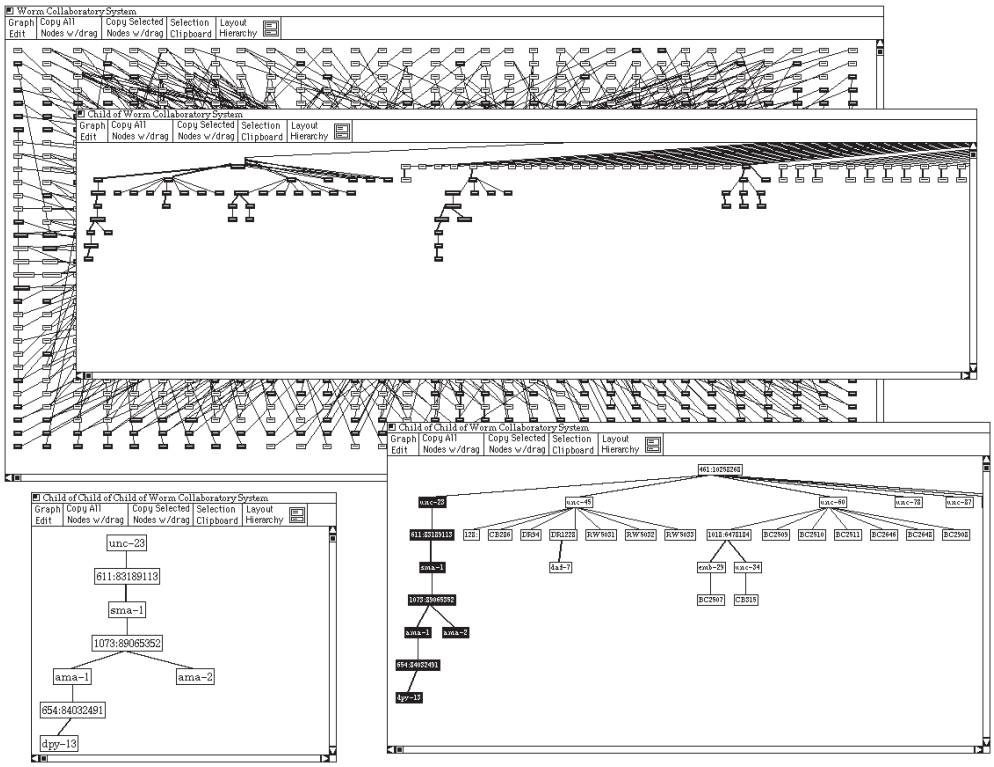

Figure 1.2: Multiple Views of a Single Large Graph

1.2 Interactive Approach

The shortcomings of traditional layout algorithms can be overcome by partitioning large graphs and generating layouts for each partition. The difficulties with this approach are how to partition the graph and how to lay out the partitions.

Since a good layout of the entire graph may not exist for large graphs. The user needs to be able to create multiple layouts or *views* of the graph. Figure 1.2 shows several views of the graph shown in Figure 1.1. If the user is in control of how the graph is subdivided into multiple views, she will be able to create views that reflect her current interests and that show manageable portions of the graph.

Traditional algorithms can't make the "best" use of the available resources for the all tasks because the user does not have any control over the layout process. If the user is given control over the layout process, he can customize the generated layout so it meets his current needs and interests.

For example, the user should be able to customize the layout so it will emphasize the components or aspects of the graph he thinks are the most important [36]. General layout algorithms try to produce layouts that do the best job possible on the overall graph. Unfortunately, good global layouts often obscure graph substructures. Layout algorithms should not always sacrifice substructures in order to clean up the overall layout of the

graph. The user should have control over the layout process so the resulting layout will reflect her current focus.

Creating layouts that cater to the user's focus is particularly important in large graphs. It can be very difficult for the user to recognize important structures in large graphs. One possible solution is to generate lots of layouts, each of which focuses on a few substructures that the user thinks are important [24]. This method allows the user to explore the graph and learn about it in comprehensible pieces.

This dissertation presents a novel methodology for viewing large and complex graphs. The basic concept is to provide the user with interactive tools that allow her to iteratively dissect large graphs into manageable pieces and then create custom layouts of these pieces. The goal is to provide a system that is powerful enough to allow a non-programming end-user to break up the graph and create custom layouts that match her current focus. The result of this approach is that the user can explore the graph and learn about the information encoded within it.

Three new concepts for building interactive graph layout toolkits will be presented. The first is an architecture for composing new graph layout algorithms out of existing algorithms. This gives the non-programming user the power to create customized layout algorithms. The second new concept is the parameterization of graph layout algorithms, which provides the user with control over the layout process. Finally, a mechanism for selecting portions of the graph that match the user's current focus will be presented.

1.2.1 Composing Graph Layout Algorithms

The basic concept of hierarchical composition of graph layout algorithms is to partition the graph into a set of unique subgraphs and lay out each subgraph in isolation from the remainder of the graph. Once layouts have been generated for all subgraphs, the generated layouts are combined into a single layout. The task of combining a set of generated layouts is similar to the task of laying out a set of nodes. The hierarchical layout process can thus be simplified by extending layout algorithms to lay out a set of generated layouts in addition to a set of nodes. The result of this extension is that a strict hierarchical ordering is imposed on the generated layouts—each generated layout is laid out by exactly one layout algorithm that may lay out any number of other generated layouts. Chapter 3 presents a methodology for composing graph algorithms.

1.2.2 Parameterized Graph Layout Algorithms

Much of the task of generating layouts can be performed automatically by algorithms, but the user often perceives changes to the generated layout that would improve it. If the user is allowed to interact with the layout process, he can fine-tune the generated layout by adjusting the layout process.

The basic concept of parameterizing layout algorithms is to extend the layout algorithms so they rely on parameters that can be imported. This method allows the user to influence the layout process, and thus fine-tune the generated layout. This method can be improved by providing an interactive interface to these parameters. Such an interface allows the user to quickly make changes to the parameters and thus experiment with how the parameters control the generated layout. The result is a system that gives the user the

feeling of directly manipulating the generated layout. Chapter 4 introduces the concept of parameterized layout algorithms and provides several examples. Section 6.2 introduces an interactive interface for algorithm parameters.

1.2.3 Subgraph Selection

The basic idea of subgraph selection is to provide the user with tools which allow her to specify arbitrary portions of a large graph. The simplest method is to allow the user to manually select nodes with the mouse. This methods has the disadvantage of not scaling well—it would be difficult to use the mouse to select meaningful sets of nodes from the graph shown in Figure 1.1.

An alternative to manual selection is to use graph traversal algorithms to specify sets of nodes. The basic idea is that the user specifies parameters to the traversal algorithms and then the algorithm traverses the graph marking nodes. This method has the advantage that it works well in both small and large graphs. Chapter 5 further discusses subgraph selection and Section 6.5 present an interactive methodology that provides the user with great control over the selection process.

1.3 Layout Examples

All of the ideas presented in this dissertation have been implemented in a prototype system and all the examples shown were generated by the prototype. Chapter 7 introduces the prototype and presents some implementation issues. Chapter 6 presents an interactive methodology that incorporates these new interactive layout concepts.

The prototype system is solely a layout editor. It allows the user to customize the layout—which may include removing nodes and edges—but does not provide a mechanism for adding new nodes and edges. Thus the prototype uses a textual language for specifying the connectivity of the graph. This language also allows an initial layout structure to be specified. Section 7.2 and Appendix A present the graph specification language.

The examples presented in this dissertation were created by interactively manipulating layouts. In order to replicate the examples and to provide an additional representation of the layout structure, scripts were created to describe the interactively created layouts. (The current prototype does not contain a dump mechanism for generating scripts from interactively created layouts). Appendix B lists the scripts for many of the examples—the scripts for the larger examples were excluded because of their size.

The related work on graph layout provides important building blocks upon which this research has been built. However, this work builds on a collection of several general research areas rather than improving a single area. The following chapter presents previous work in each of these related areas.

CHAPTER 2
RELATED WORK

Interactive graph layout is related to previous work in a number of areas:

- Automatic graph layout
- Graph editors and browsers
- Divide and conquer layout algorithms
- Constraint driven layout algorithms
- Stable layout algorithms
- Multiple view interfaces

While none of these areas directly address interactive graph layout, each provides important building blocks upon which the concepts behind interactive graph layout have been built. The remainder of this chapter addresses each of the above areas.

2.1 Automatic Graph Layout

The general methodology of interactive graph layout is to provide a framework for extending existing layout algorithms to give the user additional control over the layout process. Thus, while this work does not directly address automatic graph layout algorithms—in fact the layout algorithms used are quite simple—previous algorithms are crucial to its success.

There is a very rich set of related work on algorithms for drawing graphs. Nearly 200 papers on graph layout are presented in Eades and Tammasias' annotated bibliography [16]. Traditionally these algorithms have tried to increase the readability of graphs by focusing on specific graph properties.

It would be possible to directly incorporate many traditional algorithms into the interactive layout framework, although some may be too computationally expensive to provide quick enough feedback. However, a very simple set of layout algorithms has been implemented for the prototype system. The use of simple algorithms has allowed the research to focus on interactive issues rather than the implementation of layout algorithms.

2.1.1 Type of Graphs

Much of the literature on graph layout algorithms divides graphs into two categories: planar and non-planar graphs. A plane graph is one which can be laid out in a plane without any edge intersection. A planar graph is one which is isomorphic to a plane graph. For example, all the graphs in Figure 2.1 are planar, but only Figures 2.1.b and 2.1.c are plane graphs [67].

Another prominent aspect used to categorize graph layout algorithms is the method used for drawing the edges within the generated layout. There are four such methods:

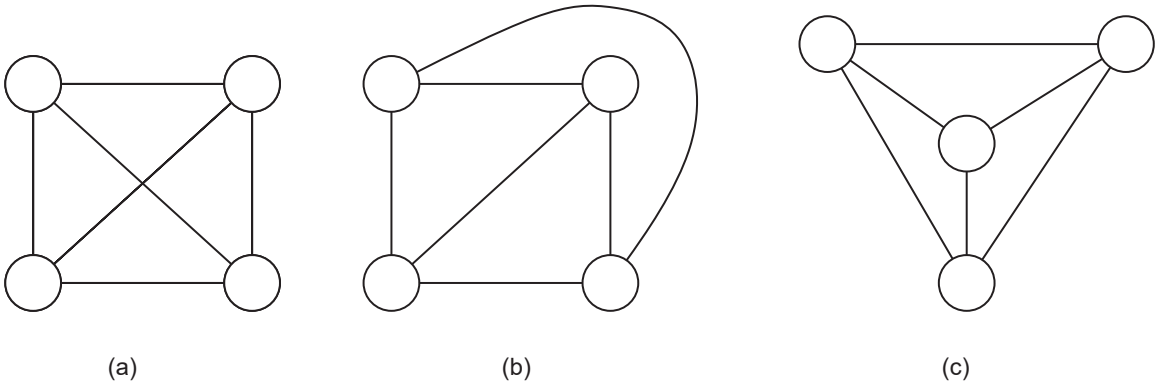

Figure 2.1: Example Planar Graphs

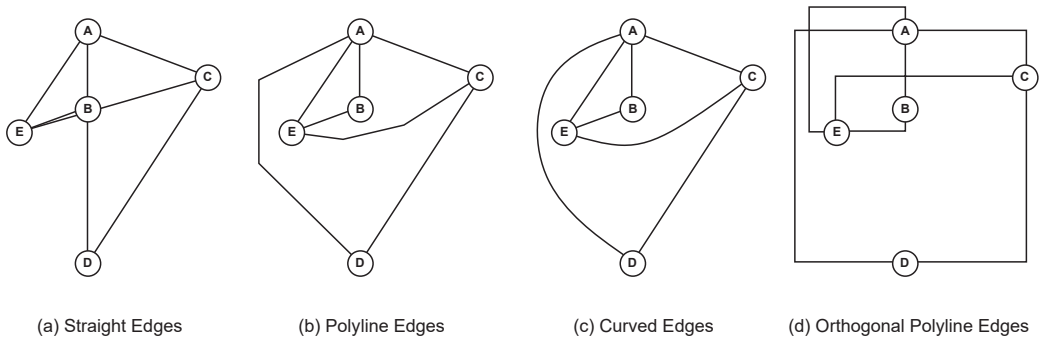

Figure 2.2: Edge Drawing Methods

straight lines, polylines, curves and orthogonal polyline. Figure 2.2 illustrates these methods by using each to draw the same graph. The simplest method is to draw all edges with straight lines, Figure 2.2.a. This method has the two disadvantages: coincidental node intersections, and overlapping edges. For example, the edge between nodes "A" and "D" in Figure 2.2.a lies on top of the edges between nodes "A" and "B." The result is that there appears to be an edge between "A" and "B" and a second edge between "B" and "C."

Polyline edges and curved edges (see Figures 2.2.b and 2.2.c) are considerably more difficult to calculate, but provide layout algorithm with more degrees of freedom resulting in generated layouts with potentially fewer edge crossings.

The area of circuit design has motivated the graph layout research using orthogonal polyline edges, Figure 2.2.d. Minimizing the number of bends in wires and the total area of the circuit are crucial to circuit design. Thus, much of the research in orthogonal polyline graph layout has focused on reducing edge bends and the total area of the graph [55, 62].

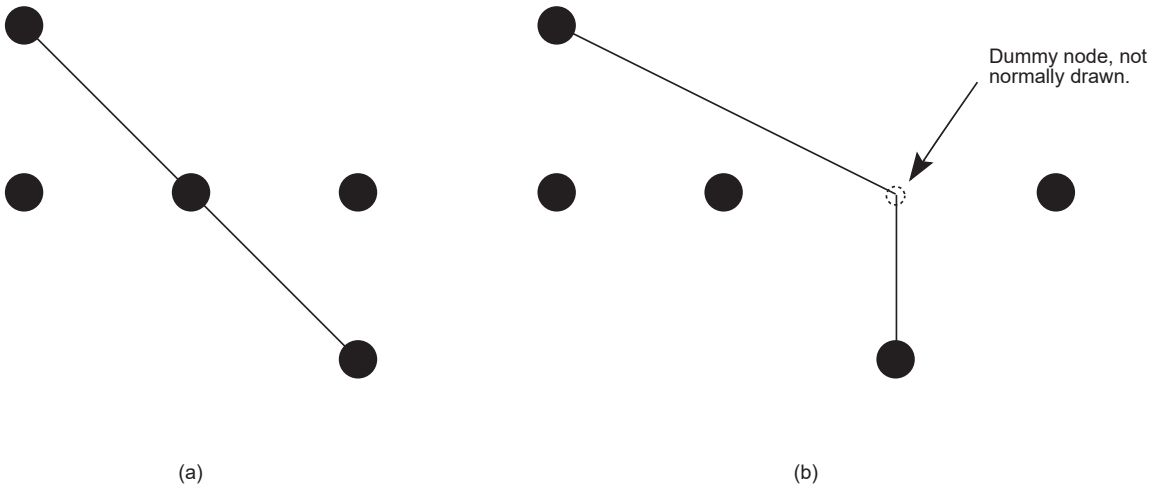

Figure 2.3: Using Dummy Nodes to Bend Edges

This work does not restrict the incorporation of a layout algorithm based on either its planarity or the method it used for drawing edges. However, in order to draw non-straight edges, artificial nodes will have to be introduced to the graph (Section 2.1.2.1 presents this technique).

2.1.2 Methods for Automatically Generating Graph Layouts

There are many different types of algorithms used to generate graph layouts [8, 18, 60, 63]. This section presents several methods in order to illustrate the great breadth of approaches that have been applied to this problem

2.1.2.1 Sugiyama Type Layout Algorithms

Minimizing the number of edge crossings in the layout of a general graph is an NP-complete problem [30]. One early heuristic solution (due to K. Sugiyama [60]) approached the problem by ordering the graph into distinct levels and using an iterative type approach to minimize the crossings between adjacent levels in order to reduce the overall number of edge crossings. Later work [18, 51] improved this method by introducing new algorithms for partitioning the graph into levels and for reducing the edge crossings.

The first step in the algorithm is to break all cycles by reversing edges found by a depth-first search to create cycles. The next step is to partition the nodes into distinct levels. The naive algorithm simply orders nodes by their distance from the root node. More advanced algorithms find optimal rank assignments for the nodes by computing integer ranks for nodes so that the sum of the costs of the edges is minimized (cost is the product of the edge weight and the edge length where length is defined as the number of levels traversed by the edge).

The next step is to create dummy nodes for every instance of an edge crossing a level. For example, Figure 2.3.a shows a graph with its nodes partitioned into three levels. There is single edge in this graph from the node in the top level to the node in the bottom level. However, this edge intersects a node in the middle level making it appear to be two separate edges. Whenever an edge traverses more than one level there is the potential for it to accidentally intersect a node. Figure 2.3.b shows the same graph with a dummy node (drawn as a white circle) inserted into the middle level. The edge is first drawn from the node in the first level to the dummy node. Then it is drawn from the dummy node to the node in the bottom level. If dummy nodes are created for each level that each edge traverses, and the edge is broken into two halves as in the example, edges never traverse multiple levels and thus no edge can accidentally intersect any nodes.

The graph shown in Figure 2.3.b was created simply to illustrate the concept of inserting dummy nodes. Actual layout algorithms would not draw the dummy node; it is just a means to reserve space for drawing edges through levels.

After the nodes have been ordered into levels and dummy nodes have been created, the algorithm attempts to reduce the number of edge crossings in the graph. The essential idea of this method is to reorder the rows to reduce the total crossing in the layout. The algorithm iterates over the layout considering pairs of rows. During each iteration the nodes in a single row are reordered to reduce the number of crossing between one of the neighboring rows. Reducing the number of crossing between adjacent rows tends to reduce the number of crossings in the entire graph.

The fourth step of the algorithm assigns drawing coordinates to the nodes. This pass does not reorder the nodes, but respects the ordering created by the first and third passes. Algorithms that create curved edges include a final pass that calculates spline control points for the edges.

2.1.2.2 Simulated Annealing

Simulated annealing [12] is a novel approach to producing graph layouts. The basic idea is to randomly improve a generated layout until a given cost function accepts it as a "good" layout. This section first introduces the concept of simulated annealing and briefly introduces the physical process it is modeled after. Then the simulated annealing graph layout algorithm is presented.

Simulated annealing is an iterative optimization process. It is especially well suited for large combinatorial optimization problems. The main difference between simulated annealing and standard iterative improvement methods is that simulated annealing allows intermediate changes to the solution that actually degrade the solution. This approach sometimes works better than methods that always improve the solution because paths towards better solutions sometimes have "hills" (intermediate results that are worse than the previous solution).

This method was derived from the physical world, in which systems sometime enter non-optimal unstable states, before reaching more stable optimal states. In the physical process of annealing, objects are heated and then cooled slowly. The molecules in hot objects are constantly moving around. If an object is cooled quickly, the molecules are frozen into a somewhat random structure. However, if the object is cooled slowly, the

molecules tend to line up and form a more defined crystalline structure. With many materials this process changes physical properties of the material.

The simulated annealing graph layout algorithm proceeds as follows [12]:

1) choose an initial configuration σ and an initial temperature T
2) repeat the following (usually some fixed number of times)
   (a) choose a new configuration σ' from the neighborhood of σ
   (b) let E and E' be the values of the cost function at σ and σ'
      if E' < E or *random* $< e^{(E-E')/T}$ then set $\sigma \leftarrow \sigma$'
3) decrease the temperature
4) if the termination rule is satisfied
   then stop
   else go to step 2

(In clause 2(b), *random* stands for a real number between 0 and 1, selected randomly.)

The basic idea behind this algorithm is to randomly change the generated layout and if that change leads to a better layout then keep the new layout. The only twist is the annealing notion of allowing changes that produce a worse layout—the notion of making uphill changes.

The temperature **T** in the above algorithm models the energy level in the physical world model. The higher the temperature, the more freedom the molecules have to move into less optimal states. In the above algorithm, the temperature **T** limits how much of degradation (or how large of an uphill move) will be accepted as an improvement. As the algorithm progresses **T** decreases. Thus as the algorithm progresses, smaller and smaller uphill changes will be accepted. Eventually, changes that degrade the solution will not be accepted.

The difficulties with this approach are how to compare two generated layouts to determine which is better (the cost function) and how to determine when the generated layout is good enough to stop (the termination rule).

The cost function evaluates how good the generated layout is or how nice it looks. The desired aspects of the generated layout must all be translated into a weighted contribution to the cost function. Cost functions usually rely on measurable aspects such as node distribution, edge length, edge crossing, avoiding bends in edges, and node-edge distances. Since the repeated calculate of the cost function is the largest computational part of the layout algorithm, great care should be taken to make it efficient.

The termination rule can take on several different forms. The simplest is to terminate after a fixed number of iterations. Another option is to terminate after several iterations that did not yield an improvement in the value of the cost function for the generated layout. Finally, more complex mathematical mechanisms [65] can be used to evaluate the progress of the algorithm.

2.1.2.3 Genetic Algorithms

Genetic algorithms have been used to generated graph layouts [20, 32, 46]. The basic idea behind genetic algorithms is similar that of simulated annealing. There are however,

two differences. First, genetic algorithms do not allow the uphill changes that simulated annealing allows. Second, the rules for determining the next iteration (or generation) of the generated layout are considerably different.

There are three basic rules used by genetic algorithms to calculate the next generation: mutation, crossover and reproduction. Mutation is accomplished by randomly altering the generated layout according to mutation rules. The bits used to encode the generated layout are grouped into packets called chromosomes. The rules for mutation allows for the bits used to encode the chromosomes to be randomly altered.

In the biological world, crossover, or recombination, is the process in which portions of different chromosomes may be exchanged. Genetic graph layout algorithms model this physical process by swapping bit substrings between two of the bitstrings that encode the layouts. New layouts are created through the process of reproduction. Two generated layouts reproduce according to a schedule dependent on their relative quality.

Similarly to the simulated annealing graph layout approach, the most important factor to the success of genetic algorithms is their ability to evaluate the generated layouts. Unlike the simulated annealing approach, it is important that the evaluation metric vary continuously in such a way that the evaluation improves as the various factors become more closely satisfied. This is because uphill changes are not allowed by the genetic algorithms and thus discontinuities of the evaluation function cannot be tolerated by the algorithm.

2.1.2.4 Force-Directed Placement

Force-directed placement [15, 17] uses a physical model from Newtonian mechanics. The idea is to model vertices as subatomic particles or celestial bodies. In this model, vertices exert attractive and repulsive forces upon one another just like subatomic particles. Like physical systems, the magnitude of the forces between vertices is related to the distance between them. In order to simplify the problem, only the forces of neighboring vertices are taken into account.

The attractive and repulsive forces are designed to reflect good graph layout qualities. For example, a repulsive force is used to prevent vertices from being drawn to close to each other and an attractive force is used to prevent edges from being too long.

Once the definitions of the forces is assigned to all the vertices, the simulation is begun. Just like physical systems, the forces induce movement of the vertices, which in turn causes the vertices to orient themselves so the net force upon them is zero. When this occurs the system will be in a static state. This state represents the generated layout of the graph. If the criteria used to define the forces accurately models good layout aspects, the generated layout will have these good aspects.

2.2 Graph Editors and Browsers

Graph editors provide the user with a tool for creating and changing the structure of graphs. Graph browsers provide tools for interactively examining, or *browsing*, graphs [26, 29, 44, 49, 51, 58]. Traditionally, graph browsers include graph edit functions.

Early graph editors/browsers required the user to manually specify the a layout of the nodes and edges. The GRAB system [51] introduced the idea of using a layout algorithm

to automatically generate the layout. Automatically calculating the layout relieves the user of the task of manually positioning nodes and edges on the screen allowing them to concentrate on editing and browsing tasks. Most recent work in graph editors/browsers have incorporated automatic graph layout algorithms.

Interactive graph layout is a new type of graph browser. The basic idea behind interactive graph layout is to provide interactive tools that allow the user to explore the graph. The major improvement over traditional graph browsers is the amount of control the user is given over both the layout and exploration processes.

Graph browsers traditionally provide the user with a single layout algorithm and thus produce a single layout. Typically, the only method the user has for viewing a layout too large to fit on the screen is to pan and zoom the display window around the virtual window containing the entire generated layout.

Interactive graph layout has extended traditional graph browsers by giving the user control over the exploration process and the layout process. Interactive tools are provided that allow the user specify portions of the graph he is interested in. This allows the user to carve a manageable size piece out of a large graph. Once the user has carved out a piece, he can create a custom layout for it.

Some graph editors limit the graphs that can be created by forcing the user to create only graphs that are legal in the specific application. For example, drawing programs such as diagram editors [14], database schema editors [4, 7, 64], and computer-aided design tools [2] use domain specific semantic to control the editing process. Incorporating the application semantics into the editor aids the user in the creation of legal graphs; however, building domain specific editors is difficult. The Unidraw system [66] solves this problem by providing a toolkit for creating domain specific editors.

General graph semantics have been used to help guide the layout process [38]. Section 9.2 discusses using domain specific graph semantics to guide interactive graph layout.

2.3 Divide and Conquer Layout Algorithms

One of the key mechanisms that allows users to create custom generated layouts is the hierarchical composition of layout algorithms. Users build new layout algorithms by plugging together existing algorithms.

The Compoze [40] system uses a method similar to the hierarchical composition method. Compoze uses a divide and conquer method that generates layouts by first partitioning the graph into subgraphs and then laying out the subgraphs in isolation. The final step is composing the generated layouts for the subgraphs into a single layout. This is similar to the hierarchical composition method in that layouts of portions of the graph are computed in isolation and then combined. There are however, two major differences. The first is that the Compoze system uses a fixed algorithm to subdivide the graph instead of allowing the user to specify the partitions. The other difference is that Compoze uses the same layout algorithm to lay out each piece of the graph. Since the hierarchical method uses a collection of algorithms to lay out the pieces, it can be thought of as a generalization of the Compoze divide and conquer method.

Hierarchical composition of layout algorithms requires that layout algorithms be able to

calculate a layout for a portion of the entire graph. This generated layout of a subgraph is in turn laid out by another layout algorithm. In order to encapsulate the generated layouts of subgraphs, a convex hull is calculated that completely encloses the generated layout. The idea of using convex hulls to encapsulate generated layouts is related to the more limited technique of *contour lines* [41].

Contour lines are polylines that describe the borders of subtrees within a larger tree layout. The idea is to lay out subtrees and then calculate contour lines for each side of the generated layout. The generated layouts of the subtrees can then be combined by using the contour lines as boundaries. This method actually produces a better layout for some trees. For example, the contour line method would avoid the problem in which two subtrees are not positioned as close as possible because of overlapping convex hulls, see Figure 3.19. However, the convex hull method is more general in that it allows generated layouts of any size or shape to be combined in any direction, not just trees placed side to side.

2.4 Constraint Driven Layout Algorithms

Constraint driven layout algorithms assign graphical relations (or *constraints*) between nodes and then attempt to position the nodes so that the relations are satisfied. There are three types of graphical relations: absolute positioning, relative positioning and clusters.

Absolute position relations constrain the node's position to a fixed coordinate. Relative positioning constrains the node's position in relation to other nodes. For example, `node A is above node B`. Clusters group a set of nodes which can further be constrained. For example, `cluster C must have a height of 10 units`.

The first step of the algorithms is to choose constraints that describe characteristics of a good layout. Constraints are then expressed as algebraic constraints among the variables that characterize the layout (i.e. the (x,y) values for the nodes). Finally the system of algebraic constraints is solved and drawing coordinates are assigned to the nodes. If all the constraints cannot be satisfied, an approximate solution is found.

The graph-theoretic approach of GIOTTO [63] uses a method similar to the variable parameterization of layout algorithms that will be presented in Chapter 4. The GIOTTO approach has incorporated constraints into the layout algorithm and allows the user to customize the graph by specifying new values for the constraints.

The main differences between this approach and variable parameterization are the directness and the functionality. The variable parameterization approach provides the user with simple but very direct parameters—there is a high correspondence between changes to the parameters and the resulting change to the generated layout.

On the other hand, constraint parameters provide the user with greater influence over the layout process than variable parameters. However, since they are parameters to heuristic algorithms they are more abstract and thus less direct. The result is that the relationship between changing a constraint and the generated layout is not always obvious.

The variable parameters method trades functionality for directness, simplicity and speed. Variable parameters affect the graph in an easier to understand manner that can always be reversed. In addition variable parameters tend to be independent of each other. Individual constraints in a system of constraints tend to rely heavily on each other.

2.5 Stable Layout Algorithms

One of the main motivations behind the parameterized layout algorithm methodology is to provide generated layouts that are *stable* (layouts that do not change drastically when the layout algorithm is reapplied to the graph) [5, 39]. Previous research has explored using constraint systems to augment automatic graph layout algorithms to achieve stable layouts.

Stable layouts are achieved by using constraints to control the layout process and allowing the user to modify these constraints and to add new constraints to the system. For example, if a user were to rearranged two nodes in the generated layout, the system would create new constraints to describe the new arrangement of nodes. These constraints would then be added to the system of constraints used to generate the layout. Since the user's changes become part of the system of constraints, the changes are incorporated back into the layout process making it a stable layout.

The main disadvantage of this approach is the complexity of systems of constraints. It can be difficult for a user to understand how his action was translated into a set of constraints and how these constraints affected the layout process.

2.6 Multiple View Interfaces

The interactive graph layout framework allows the user to create simultaneous multiple presentations, called *views*, of a given graph. Each view provides a direct manipulation interface [28, 52] that allows the user to interact with the generated layout.

Direct manipulation interfaces provide the illusion of directly manipulating data objects. This illusion is most often created by representing the data objects with graphical interaction objects that can be directly manipulated by the user. One of the main goals of direct manipulation interfaces is to increase the directness felt by the user—so the user feels she is interacting directly with the data and not with an interface.

There are two important aspects of a user interface that contribute to its directness; how closely the interface objects match the data objects and how closely the interface structure matches the user's mental model. However, all users do not always have the same mental model. Mental models may vary from user to user and may even depend on the user's current interests and the current data set. Thus there is not always a single "best" structure for any given interface.

Since each view acts as a direct manipulation interface to the layout system, views are analogous to general multiple view interfaces. For example, the Rooms [24, 9] system allows the user to create multiple views or *rooms* for an application. Each room is a custom view of the interface that caters to the user's current interests.

The Macintosh Finder interface also provides the user with multiple views of the data [1]. The finder interface normally depicts files as icons on the desktop. However, the user can optionally select from a small set of alternative textual views. Each of these views has advantages and disadvantages, and each supports some tasks better than others. By providing alternative views, the supported range of tasks increases and the interface becomes more useful.

As a more subtle example of multiple view interfaces, consider the Cone Tree [48] system. The Cone Tree system is designed to visualize hierarchical data structures. It uses a 3D representation of the data that can be interactively animated. Each level of the hierarchy is represented by a 2D projection of a 3D cone. The root node for each level is placed at the top of the cone and the children nodes are evenly distributed along the perimeter of the base. The diameter of the base of the cone depends on the number of children and the size of the children. The result is that the cones representing the higher levels of the hierarchy are larger than those that represent the lower levels.

The interface to the Cone Tree allows the user to rotate all the cones simultaneously to bring specific paths to the front of the display. Thus while the user cannot create new views of the interface, he can customize the interface to meet his current interests. In addition to rotating the structure, the user can create animation sequences that do a good job at illustrating the relationships between different paths.

The Perspective Wall [36] is another example of an interface that gives the user the control to specify which aspects of the interface should be viewed in greater detail than the rest of the interface. The Perspective Wall uses the physical metaphor of folding to distort a 2D layout into a 3D visualization. The result is that the center of the 2D layout appears undistorted while its sides are folded away from the user. This folding distorts the sides in order to conserve screen space but still provide contextual information.

The advantage of the Perspective Wall is that it allows more information to be presented on the screen than an undistorted interface. This is accomplished by providing detailed information of part of the interface, and contextual information for the remainder of the interface. The interface to the Perspective Wall allows the user to shift the 2D interface back and forth along the wall. Thus the user can select which portion of the interface is to be displayed undistorted and which portion is to be displayed distorted. Similarly to the Cone Tree, the Perspective Wall gives the user control over the interface so it can be customized to meet their current interests.

CHAPTER 3
COMPOSITION OF GRAPH LAYOUT ALGORITHMS

The basic strategy of composing graph layout algorithms is to partition the graph into a set of unique subgraphs and lay out each subgraph in isolation from the remainder of the graph. Once layouts have been generated for all subgraphs, the generated layouts are combined into a single layout. The task of combining a set of generated layouts is same to the task of laying out a set of nodes. Generated layouts are treated as large nodes and the edges that go between two generated layouts are treated like edges between any other nodes. The hierarchical layout process can thus be simplified by extending layout algorithms to lay out a set of generated layouts in addition to a set of nodes. The result of this extension is that a strict hierarchical ordering is imposed on the generated layouts—each is laid out by exactly one layout algorithm and may include any number of other generated layouts.

Since each subgraph is laid out in isolation from the remainder of the graph, each can be laid out by a different layout algorithm. Thus a heterogeneous set of layout algorithms can be used to lay out a single graph. The term *layout object* will be used to refer to a subgraph coupled with an instantiation of a layout algorithm.

As an example of a very simple hierarchically composed layout algorithm, consider the generated layout shown in Figure 3.1. This layout was created by a two level hierarchy. The left part of the graph was laid out by a tree layout algorithm. The right part was laid out by a grid layout algorithm. The tree and the grid were then positioned next to each other by a row layout algorithm.

Since each node is drawn only once, nodes can belong to exactly one layout object. In order to simplify the layout process, layout objects can belong to no more than one other layout object. The result is a strict hierarchy of layout objects in which all but the root layout object have a parent layout object. While each node may belong to only one layout object, edges may connect nodes in different layout objects. For example Figure 3.2 shows a generated layout in which edges go between layout objects. This layout was generated by two tree layout objects—one oriented top-to-bottom, the other oriented bottom-to-top. The tree layout objects are in turn laid out by a column layout object.

The algorithm used to generate hierarchically composed layouts proceeds from the bottom to the top of the hierarchy: it starts with the layout objects that contain no other layout objects and proceeds towards the root. This can be thought of as an inside-out ordering. The generated layouts that are laid out by another layout algorithm can be thought of as inside layouts because they are encapsulated by other generated layouts. Since these layouts are generated first, the layout process proceeds from the inside to the outside of the generated layout.

This approach does not allow for communication between layout algorithms, thus the algorithms used to lay out the two tree components of the layout shown in Figure 3.2

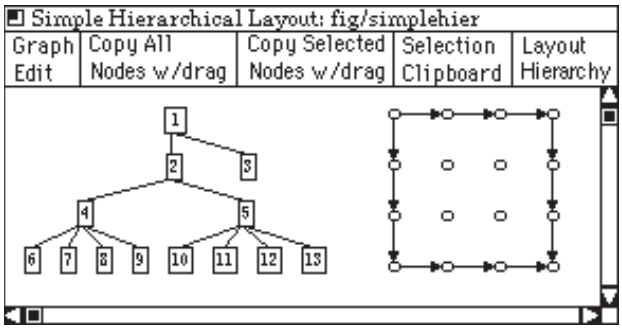

Figure 3.1: Simple Two Level Hierarchically Composed Layout: Row Containing Tree and Grid

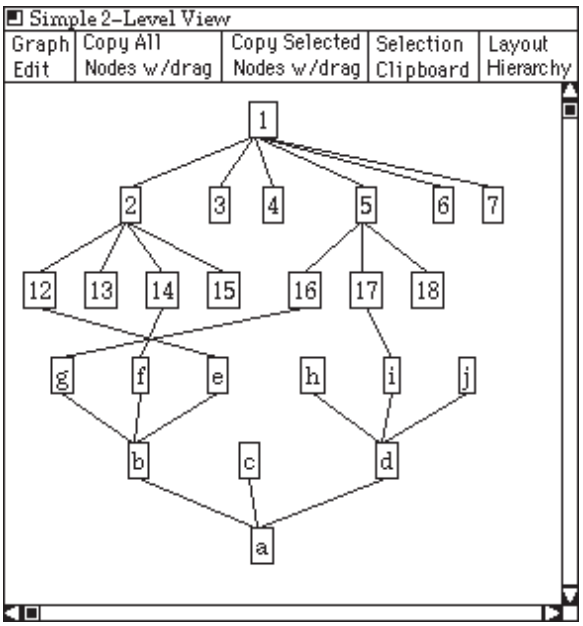

Figure 3.2: Two Level Hierarchically Composed Layout: Column Containing Two Trees

---

cannot communicate and position their nodes to eliminate the edge crossings. Figure 3.3 shows the same graph laid out with the same algorithms. The only difference was that the ordering of nodes "e" and "g" have be interactively changed. Section 6.4 will introduce a mechanism for manually interacting with the layout and changing node orderings. Section 3.5 discusses alternatives to the inside-out ordering of the layout process.

After a layout object lays out its nodes, it calculates the minimum convex hull that encloses the generated layout. This hull is then passes up the hierarchy so the generated layout can be laid out by its parent layout object. Section 3.2 will elaborate on the layout process. The edge layout mechanism is discussed in Section 3.3 and Section 3.4 presents

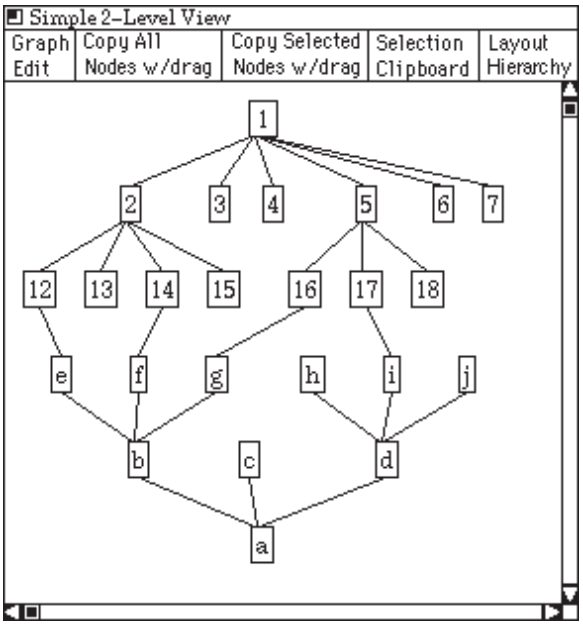

Figure 3.3: Manually Improved Layout of Column Containing Two Trees

---

the algorithms used to create, manipulate and position convex hulls.

3.1 Creating New Layout Algorithms

Composing layout algorithms hierarchically provides an abstraction for creating new layout algorithms without programming. End-users can create new layout algorithms by combining existing algorithms. Thus composing layout algorithms provides the end-user with the power to create customized layout algorithms which generate layouts that reflect her current interests. As a result, the user can create layout algorithms that highlight a very specific portion or aspect of the graph.

As an example of creating a new layout algorithms, consider a completely connected graph. One good way to lay out a connected graph is by placing the nodes on the perimeter of a circle. Figure 3.4 shows an 8 node connected graph. While programming an algorithm to layout a connected graph in a circle would not be too difficult for an experienced programmer, it would be very difficult for non-programmers. However, the layout shown in Figure 3.4 was created by combining several instantiations of a row layout algorithm. This row layout algorithm positions nodes in a row (optionally a column) with nodes justified on the top, center or bottom (left, center, or right for columns).

Figure 3.5 shows the hierarchy of layout algorithms used to generate this layout. The top and bottom pair of nodes are laid out by a row layout algorithm. The top row is bottom justified—it aligns the bottom of both nodes—and the bottom row is top justified. The left and right pair of nodes are each laid out by a column layout algorithm. These columns are in turn laid out by a row layout algorithm. All three rows are then laid out

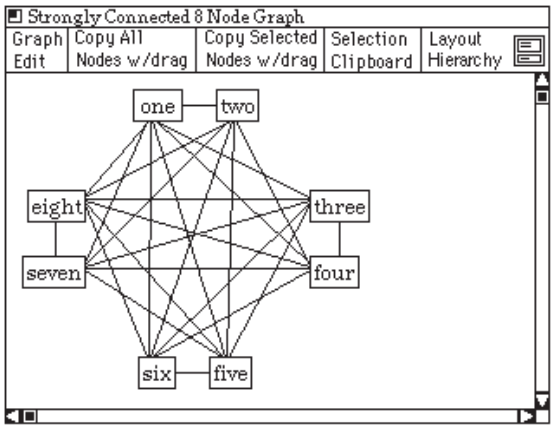

Figure 3.4: 8 Node Connected Graph

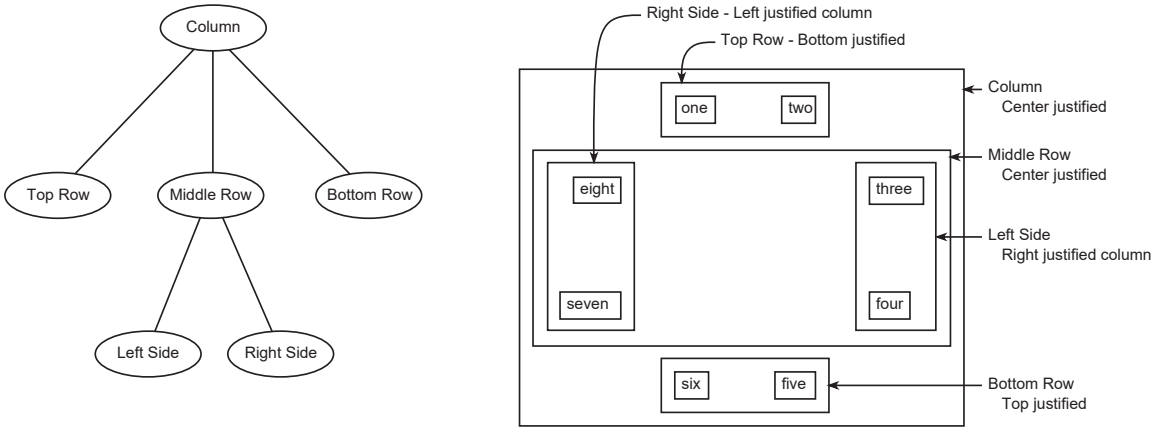

Figure 3.5: Hierarchy of Layout Objects for Figure 3.4

---

by a column layout algorithm. Since an individual row algorithm lays out its nodes in the order in which they are passed to it, the nodes were manually ordered for this example.

While the composition of layout algorithms in the above example may not be immediately obvious, it not overly complicated. Once the basic structure was discovered, it was not difficult—although it was repetitious—to expand the composed layout algorithm to handle larger connected graphs. Figure 3.6 shows a layout of a 16 node connected graph that was generated by a hierarchy of layout algorithms similar to that used in the 8 node example.

Using hierarchically composed layout algorithms is particularly simple in very regular graphs. The examples shown above are highly regular and thus the new layout algorithms could be composed out of multiple copies of a single layout algorithm. More interesting

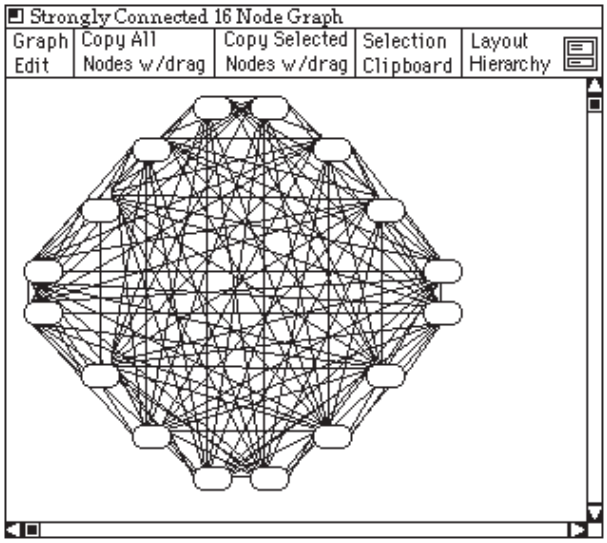

Figure 3.6: 16 Node Connected Graph

layouts can be created by combining different algorithms. For example, consider the graph shown in Figure 3.7. It contains two cycles that are visible but not entirely obvious. The graph is laid out by a simple hierarchical layout algorithm based on ideas presented in [60, 18, 51]. Figure 3.8 shows a different layout of the same graph shown in Figure 3.7. It is laid out using the composed algorithm shown in Figure 3.5 (the same algorithm used to lay out the graph shown in Figure 3.4) to position the cyclic components and the hierarchical layout algorithm to position the remainder of the graph.

Neither of the layouts shown in Figures 3.7 and 3.8 is categorically better than the other. Each however highlights different aspects of the graph and thus could prove more useful in different situations. The layout in Figure 3.7 clearly shows the height of the graph—the distance between the root node "1" and the furthest nodes "hh" and "HH." On the other hand, the layout in Figure 3.8 highlights the existence of the cycles. Since both layouts are valuable, the user must be able to see both views and have the freedom to alternate between them. The user must also have the power to create new layouts as she explores the graph. Hierarchically composing layout algorithms provides the user with the power to create new layout algorithms and thus is very versatile.

Layouts that clearly show the global structure of the graph often obscure important graph substructures. In the example above, the hierarchical layout somewhat obscures the cycles while the combination algorithm clearly highlights the cycles. Hierarchical combinations of layout algorithms can generate layouts that draw important graph structures clearly, while allowing the algorithm that best presents the overall structure to lay out the overall graph. This is a particularly important feature when dealing with large graphs. The user must have the power to focus the layout on the aspects of the graph she is currently interested in. Without this power, the graph features the user is interested in may be obscured by the remainder of the graph.

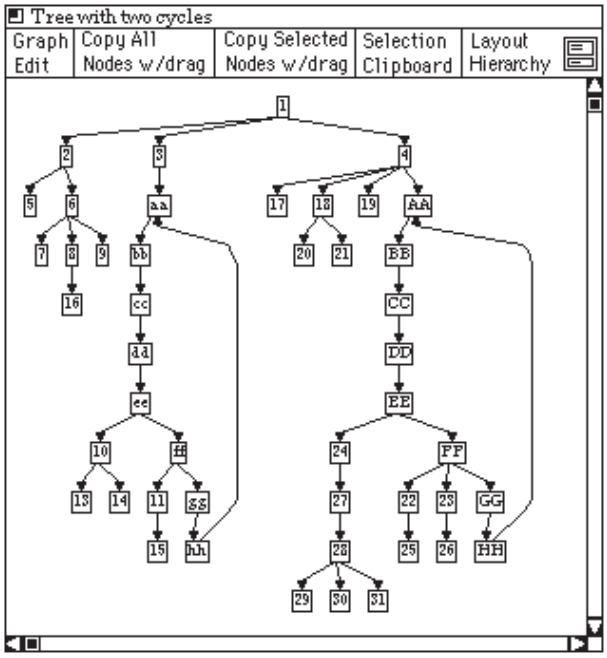

Figure 3.7: Tree with Two Cycles

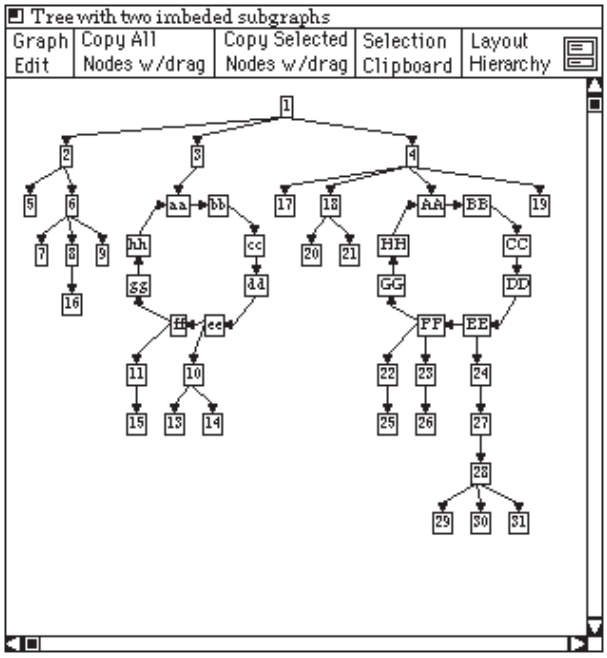

Figure 3.8: Same Graph as Figure 3.7 Laid Out Using the Composed Cycle Algorithm

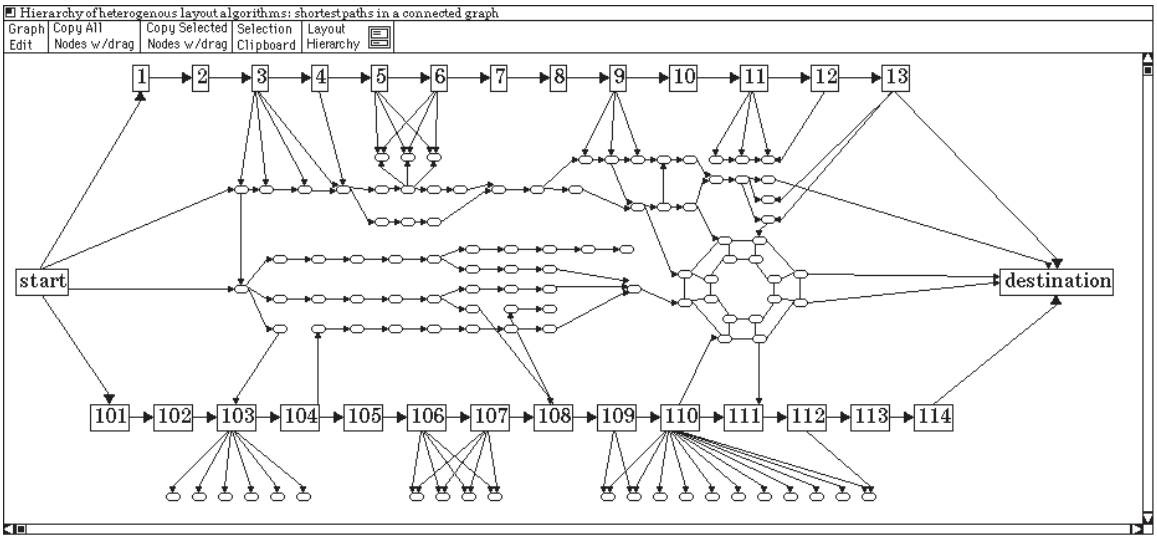

Figure 3.9: Highlighted Shortest Paths

Figure 3.9 shows a graph laid out using a collection of layout algorithms. The focus of this layout is the two shortest paths between the nodes "start" and "destination." Each of these shortest paths is laid out using the row layout algorithm. The result is that the user can instantly recognize the shortest paths and see how the remainder of the graphs connects to these paths. Other features of the graph—for example the concentric rings—are laid out in such a manner to convey their actual structure. This composed layout algorithm was interactively created to lay out this particular graph. It consists of thirty two instantiations of the row and hierarchical layout algorithms.

After the hierarchy was created, several interactive iterations were necessary to fine tune the graph so it would clearly highlight the two shortest paths. However, each iteration is so quick that the user can afford to make many iterations. Chapter 6 introduces an interactive architecture for displaying and editing the generated layout.

The result of the specialized hierarchy and the interactive fine tuning is a "new" layout algorithm specifically designed to lay out this graph. There is however, a trade off between the functionality and the creation time of layout algorithms. General layout algorithms can handle a large class of graphs, but are difficult and time consuming to program. Composed layout algorithms, on the other hand, can handle a small set of graphs—maybe only one—but are quick and easy to create. The advantage of composed algorithms is that they often do a far better job on specific graphs than general algorithms.

3.2 The Layout Process

Table 3.1 summarizes the recursive algorithm used to calculate hierarchically composed layouts. The layout process starts with the outermost layout algorithm (the root of the

```
calculate_layout(graph)
    ∀ layout_object ∈ graph.children_layout_objects
        hull = calculate_layout(layout_object)
        hull_set = hull_set ∪ {hull}
    ∀ node ∈ graph.my_nodes
        hull = calculate_hull(node)
        hull_set = hull_set ∪ {hull}
    calculate layout for all hulls ∈ hull_set
    new_hull = calculate convex hull enclosing generated layout
    return new_hull
```

Table 3.1: Hierarchical Composition Layout Algorithm

hierarchy). It requests the convex hull for each of its node and layout objects. Nodes simply return their enclosing convex hull, layout objects must generate their layout and then return the convex hull that encloses this layout.

When a layout object receives a request for its convex hull, it must first calculate a layout for all of its nodes and layout objects. Since the first step in generating a layout is to request the convex hulls from all of its nodes and layout objects, the process recursively descends the layout hierarchy until it reaches the innermost layout object. Since this layout object contains only nodes and no layout objects, the recursion stops and this innermost layout object can proceed to calculate its layout.

Nodes that do not change size or shape calculate their convex hull at creation time and return this hull when they receive a request for their hull. Dynamic nodes calculate a new convex hull if their size or shape changes—this can only happen if one of their parameters that control their size or shape have changed. Methods for calculating convex hulls are described in Section 3.4.

In isolation from the remainder of the generated layout, the innermost layout object lays out its nodes and sublayouts using their convex hulls as boundaries. Since the generated layout will in turn be repositioned by the parent layout object the final positions of the nodes cannot yet be determined. Thus the layout algorithm assigns $\mathbf{x}$ and $\mathbf{y}$ offset values to each of the nodes instead of absolute screen coordinates.

After all the nodes have been positioned—have been assigned an offset—the layout object calculates the minimum convex hull that encloses the generated layout and returns it to the parent layout object allowing the parent to proceed with the layout of its objects.

When the parent layout object positions the generated layout, an offset will be assigned to the layout object. Thus the actual screen coordinates of nodes can be found by adding the node's offset to all the offsets of the layout objects above it in the layout hierarchy.

The only restriction on layout algorithms are that they request hulls from all of their children and that they calculate a hull that encloses the generated layout. Thus layout algorithms are free to position these hulls using any method. This method could even

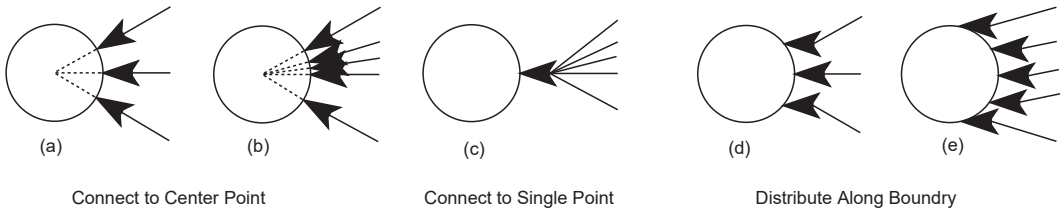

Figure 3.10: Connecting Edges to Nodes

include input from the user allowing the user to manually lay out a subgraph.

The hierarchical composition layout algorithm does not explicitly lay out edges. The following section introduces mechanisms for edge layout.

3.3 Edge Layout

One of the most difficult problems in generating graph layouts is to prevent or minimize edge crossings and accidental edge intersections with nodes. Traditional graph layout algorithms have focused on minimizing the number of edge/edge and edge/node intersections. In the general case, it is not always possible to prevent all edges intersections. Additionally, it is a NP-complete problem to minimize edge crossings in a general graph [19, 30].

Heuristic algorithms have been created that do a good job of minimizing edge crossings [18, 38, 40, 60, 63]. There are no restrictions on what layout algorithms can be incorporated into the interactive framework. However, very computationally intensive algorithms may be too slow to generate layouts quick enough for an interactive framework.

The main focus of interactive graph layout is to examine how to give the user control over the layout process. Thus complicated layout algorithms that reduce edge crossings are not explicitly addressed. In fact, the layout algorithms implemented in the prototype are computationally very simple, but allow the user to interact with the layout process.

A basic component to all graph layout systems is how to calculate the best point to connect edges to nodes. The following section presents several methods for calculating these connection points. Traditionally the connection points are calculated by the graph layout algorithm. A novel approach is presented that allows the node to calculate the connection points for all of its edges.

3.3.1 Connecting Edges to Nodes

There are three common methods for connecting edges to nodes: connect all edges to the center of the node [1] (Figures 3.10.a, 3.10.b), connect edges to the midpoint of the nearest side (Figure 3.10.c), and distribute edges along the node's boundary (Figures 3.10.d, 3.10.e)

[1]Since nodes are drawn on top of the edges, the part of the edge that is within the node will not be seen.

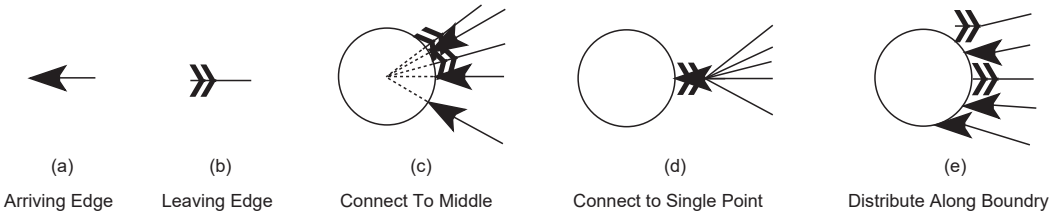

Figure 3.11: Connecting Edges with Different Endpoint Images

While connecting edges to the center of the nodes is the easiest to calculate, it has two problems. The first is that endpoint images must be rotated. While this is not a problem on high resolution devices, some images are difficult to rotate in screen resolution. The second problem is that if many edges are connected to a single node, the endpoint images may overlap (Figure 3.10.b). This problem becomes even more noticeable when edges have different endpoint images as in the example shown in Figure 3.11.c.

The next method is to connect the edges to the midpoint of the nearest side. For example all edges that originate from the right of the node should be connected to the right side of the node (Figure 3.10.c). This method is simple to calculate, and does not require rotating the endpoint images. However, it does not work well when edges have different images (Figure 3.11.d).

Distributing edges along the boundary works well with edges that have different endpoint images and does not require rotation of the images (Figure 3.11.e). There are however, two problems associated with it. The first problem is when too many edges are connected to the nodes. Since each edge—if it has an endpoint image—occupies the width of their image of the boundary, there is only room for so many edges. The result is that the edges do not actually connect with the node. This problem can be avoided by reverting to the simpler solution of overlapping edges images.

The interactive framework allows each of the three edge connection methods. In order to provide this flexibility an edge ordering mechanism has been defined in the methodology—distributing edges along the boundary requires an ordering of edges.

Traditional graph layout algorithms consider all the nodes in the graph and can thus calculate how to draw the edges. In the hierarchically composed model, an edge may connect two nodes that belong to two different layout objects. Thus neither of the layout algorithms has the complete information needed to position the edge.

This problem can be solved by leaving the calculation of edge connection points up to the nodes. After all the nodes have been laid out by all the layout algorithms, each node calculates the best connection point for each of its edges. This approach has the disadvantage of limiting the complexity of edge layout, but the layout algorithm can still control edge layout by creating dummy nodes that have the affect of bending the edge [60]. See Figure 3.7 for an example of edges bent by using dummy nodes.

In addition, this method allows custom node types to specialize the connection algorithm. For example, the visual programming interface presented in Section 6.5 uses

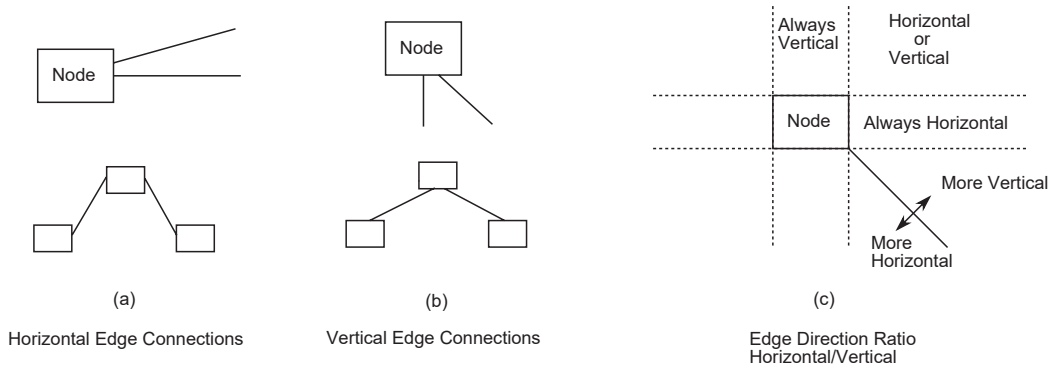

Figure 3.12: Vertical vs. Horizontal Edges

a customized node definition to implement objects in its visual programming language. These nodes have explicit ports to which edges (dataflow links) can be connected. Since nodes have control over where edges are connected, the node definition can assure that edges are only connected to the explicit ports.

There are situations in which it is ambiguous to which side of node an edge should be connected. When an edge connects two nodes directly above and below each other, it is clear that the connections should be vertical—the edge should connect to the top and bottom of the two nodes. Figure 3.12.b illustrates *vertical edge connections*. Similarly, an edge between two node that line up horizontally should have horizontal connections. Figure 3.12.a illustrates *horizontal edges connections* in which the edges are connected to the sides of the nodes.

If nodes are neither directly above/below or left/right, it is ambiguous if edge connections should be vertical or horizontal. Figure 3.12.c illustrates when it is ambiguous which side to connect the edge to. Edges that connect to nodes that are directly above or below this node should always have vertical connections. Edges that connect to nodes that are directly left or right should have horizontal connections. The remaining edges could have either vertical or horizontal connections.

The manner in which the edges are connected to the nodes (vertical vs. horizontal connections) is important because it influences the appearance of the generated graph. Horizontal versus vertical edge connections change the entire look of the graph and thus the entire graph must be taken into account when determining where to connect edges. Figure 3.13 shows the same graph laid out twice. The left hand layout uses vertical edge connections and the right hand layout uses horizontal edge connections.

Individual nodes cannot always determine which connection points would be best for a given edge because they cannot use the context of the entire graph. Consider the three nodes enclosed by dashed lines in Figures 3.14 and 3.15. The orientation of these three nodes is the same in all four graphs, but the context and the edge connection method varies.

Ultimately the user should decide which looks best for the current graph. In the work

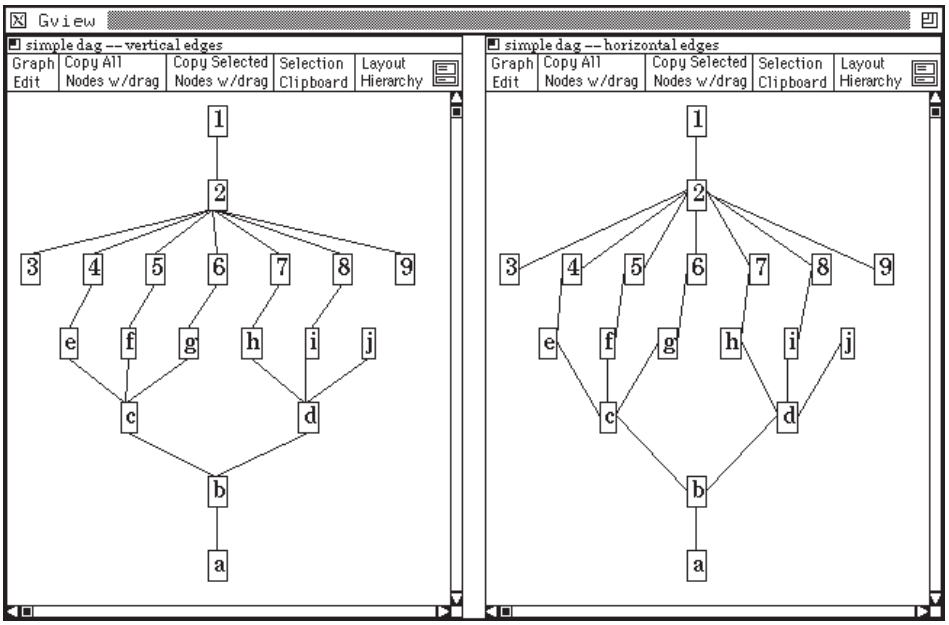

Figure 3.13: Vertical Edge Connections vs. Horizontal Edge Connections

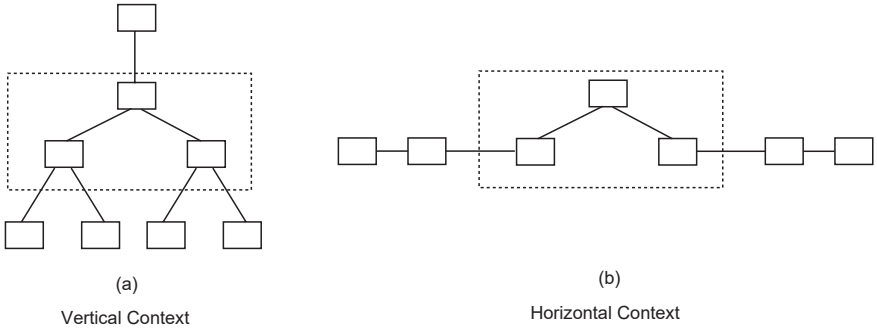

Figure 3.14: Vertical Edges in Vertical and Horizontal Contexts

---

described here, the user is given control over the choice between vertical and horizontal edge connections by associating a ratio of vertical to horizontal connections with each layout object. This ratio is analogous to the horizontal/vertical dividing line shown in the lower right of Figure 3.12 (the line is drawn at the 45 degree default value). Edges that connect to nodes that lie above the dividing line will be connected horizontally, edges that lie below the line will be connected vertically. Changing the ratio shifts this line and thus changes how edges are connected for nodes that lie in the four corner quadrants.

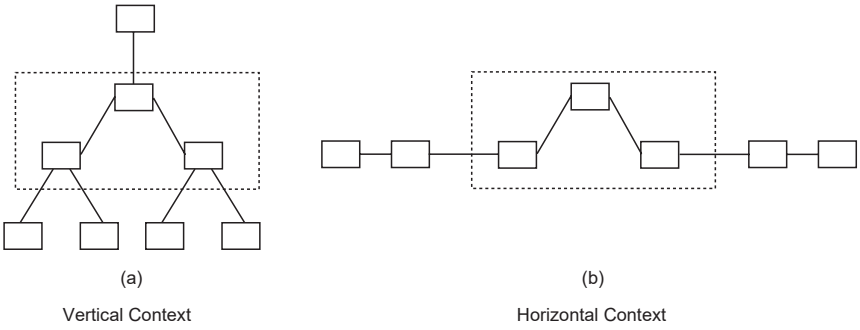

Figure 3.15: Horizontal Edges in Vertical and Horizontal Contexts

3.4 Using Convex Hulls to Encapsulate Layout Objects

The *convex hull* of a set of points is defined as the smallest *convex polygon* enclosing all the points. A convex polygon is a polygon such that any line segment connecting two points inside the polygon is itself inside the polygon [37].

Convex hulls are used as a graphical approximation of nodes and generated layouts. If all node types and all layout algorithms calculate the minimal convex hull that encloses the node or generated layout, all node types and generated layouts become standardized. Layout algorithms can thus deal with a homogeneous set of convex hulls instead of a heterogeneous set of nodes and layout objects.

Nodes must calculate convex hulls at creation time and whenever they change their size or shape. Layout algorithms must calculate the convex hull that encloses the generated layout after each time the algorithm is applied to its set of nodes and layout objects.

Section 7.1 presents some implementation issues of calculating convex hulls for nodes and generated layouts. The remainder of this chapter presents some advantages and disadvantages of using convex hulls and explores the alternative of non-convex hulls.

Once layouts and nodes are standardized by requiring them to create a convex hull, parts of the layout process can be extracted from individual layout algorithms. For example, an algorithm to lay out a tree can be recursively stated as follows:

> Layout the root's subtrees (each will return a convex hull).
> Position each subtree close to each other with all roots in a row.
> Position the root over the middle of all the subtrees.

The difficult part of this algorithm is calculating how close the subtrees can be positioned—the minimum separation. Figure 3.16.a illustrates this step. The subtrees need to be positioned so as to minimize the distance d while keeping d' non-zero. Figure 3.16.b shows the result of moving the subtrees in Figure 3.16.a close as possible to each other without overlapping.

The task of calculating the minimum separation can be performed by a hull positioning algorithm thus simplifying the tree layout algorithm. The advantage of separating the hull

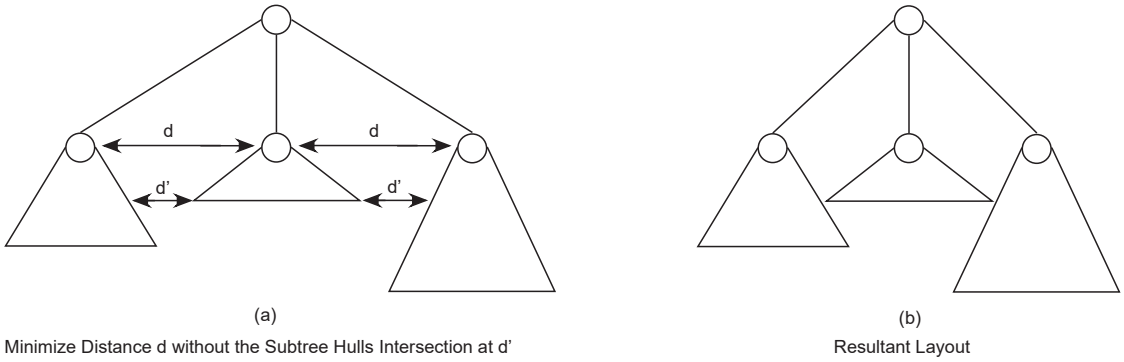

Figure 3.16: Using Convex Hulls in the Layout Process

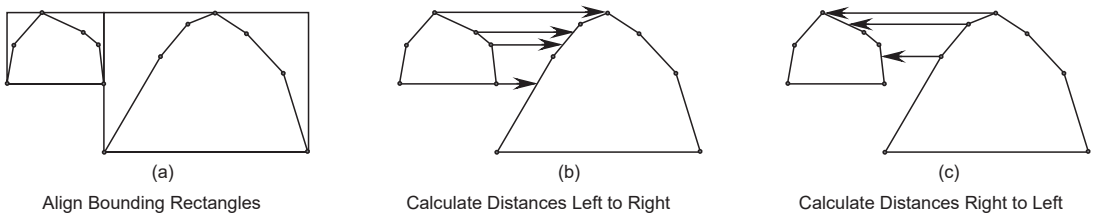

Figure 3.17: Calculating Minimum Horizontal Separation Between Hulls

---

position is that the algorithm can be used by all other layout algorithms. There are two basic hull positioning tasks, *fit on right*, and *fit on bottom*. The fit on right algorithm calculates the minimum horizontal separation for two hulls oriented such that their top bounds intersect the same horizontal line. The fit on bottom algorithms does the same for hulls oriented along a vertical line.

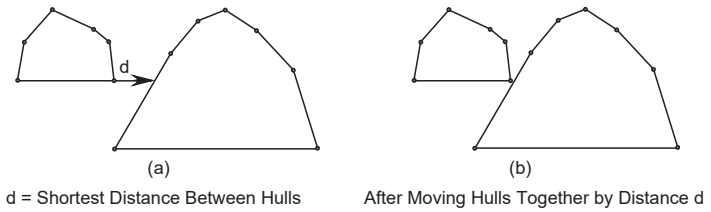

Figure 3.18: Fitting Hulls Together Horizontally

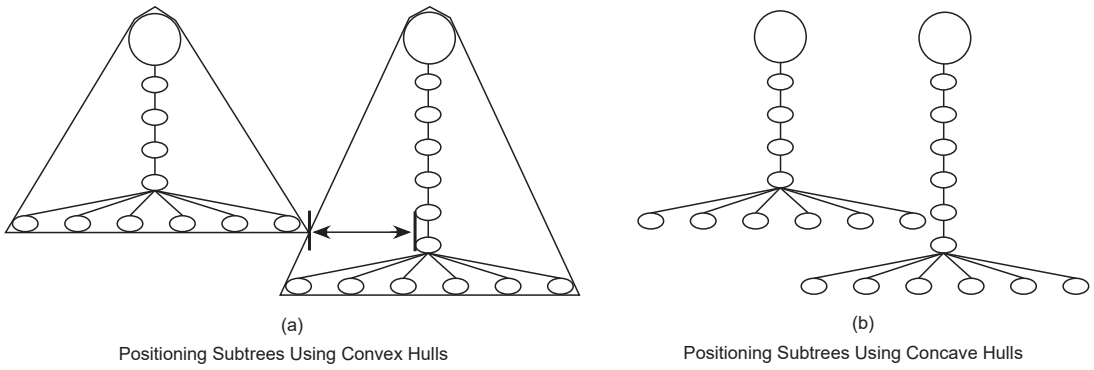

Figure 3.19: Disadvantage of Convex Hulls vs. Concave Hulls

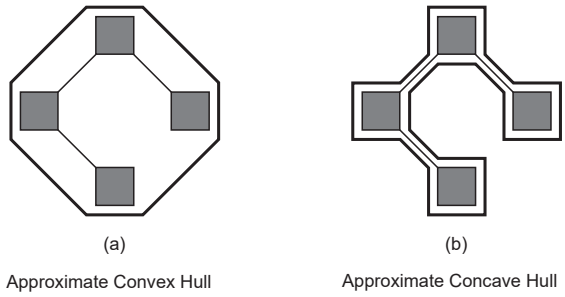

Figure 3.20: Convex and Concave Hulls of Connected Graph

Figure 3.17 illustrates how the minimum horizontal separation distance is calculated. In the first step, the hulls are aligned as shown in Figure 3.17.a. The next step calculates the distance from each of the vertices to the neighboring hull, see Figures 3.17.b and 3.17.c.

Since all the hulls are convex, the shortest distance between the hulls must originate at a vertex on one of the hulls. Figure 3.18.a highlights the shortest distance between these two hulls. Figure 3.18.b shows the result of moving these two hulls towards each other by this distance.

Since convex hulls only provide an approximation of a subgraph, there are situations in which they do not perform well. Figure 3.19 presents an example in which the convex hull tree layout algorithm does not perform well. Figure 3.19.a shows two subtrees with their convex hulls. These subtrees are positioned as close to each other as possible without their hulls overlapping. Figure 3.19.b shows the same two subtrees positions as close to each other as possible without the nodes overlapping. The resulting graph in Figure 3.19.b is smaller than the convex hull version in 3.19.a.

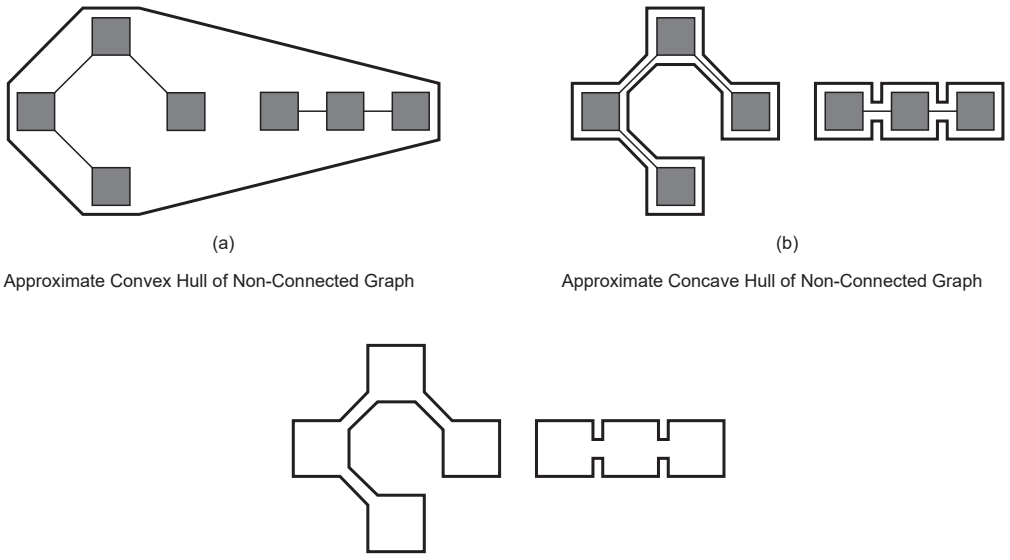

Figure 3.21: Convex and Concave Hulls and Disconnected Graph

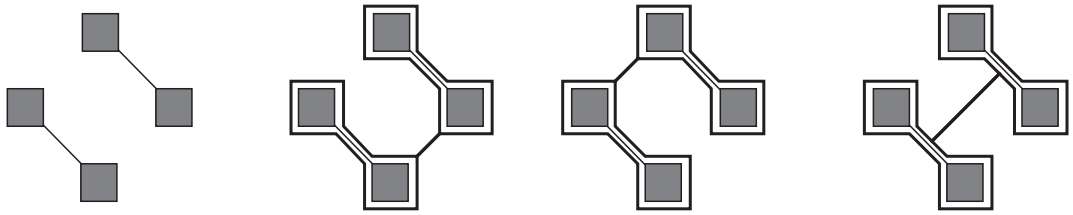

Figure 3.22: Disconnected Graph and Three Possible Concave Hulls

---

The approximation problem associated with using convex hulls can be avoided if *concave hulls* are used. A concave hull of a set of points is defined as the smallest *closed path* (polygon) enclosing all the points. Concave hulls are however considerably more complicated and have several problems associated with them. Figure 3.20 shows approximate[2] convex and concave hulls of a four node connected graph.

Concave hulls can contain pockets and thus can have many sides that face a single direction. This complicates the positioning algorithms presented above, especially when it is possible to put one of the hulls inside of a pocket or hole in the other hull.

[2]The hulls shown in Figure 3.20 are enlarged so they will be more apparent in the figure.

The method of using convex hulls presented above does not explicitly take edges into consideration. However, since convex hulls are created around nodes, and edges are straight[3] line segments between nodes, by the definition of convex hulls, the hull is guaranteed to include all edges within the subgraph—that is all edges that originate and terminate within the subgraph. However, when using concave hulls one must take into account all the subgraph's edges.

The concave hull shown in Figure 3.20.b was drawn around all the edges in the subgraph, not just the nodes. However, if the subgraph is connected, then there always exists a single unique concave hull that encloses all the nodes and edges. If the subgraph is not connected, then a unique hull no longer exists. Figure 3.21.c shows the concave hull that encloses a disconnected graph. It is possible to create a single concave hull that encloses a disconnected graph, but the hull is no longer unique and it is not clear how to create the best hull. Figure 3.22 shows three valid concave hulls for a disconnected graph. While all three of the hulls shown in Figure 3.22 are valid hulls, each could prove better than the others in different situations.

The alternative is to use a set of disjoint hulls to describe non-connected graphs. As the hierarchical layout process proceeds, sets of concave hulls will have to be passed up the hierarchy. Each step will have to consider edges that cross between its subgraphs and recalculate a new set of concave hulls. While using concave hulls may result in better generated layouts, the layout process would be considerably more complex and thus the implementation of new layout algorithms would be complicated. In fact, some layout algorithms may not even be able to handle laying out concave hulls.

3.5 Inside-Out Nature of the Layout Process

The hierarchical layout process proceeds from the bottom of the hierarchy to the top or from the inside of the generated layout to the outside. This strict inside-out nature allows the user to focus on specific pieces of the graph and highlight small details and structures of the graph. For example, if the user were interested in a particular structure, she could put it in a well suited layout object at the bottom of the hierarchy. This has the result of the graph being laid out around the important structure.

The biggest advantage of the inside-out ordering of layout algorithms is its simplicity. Since each algorithm proceeds in complete isolation from the other layout algorithms, the algorithms don't have to include procedures to communicate with the other layout algorithms. Thus it is considerably easier to add new layout algorithms to the system.

In addition to not complicating the implementation of new layout algorithms, the inside-out ordering simplifies the programming model. The user does not need to worry about how layout algorithms communicate or which algorithms are compatible. All he has to do in order to create a new layout algorithm is to impose a hierarchical ordering on a set of layout algorithms.

The disadvantage of the inside-out model is that layout algorithms cannot communicate with other layout algorithms. Thus it is difficult to position edges that cross layout object boundaries. Since a layout object cannot find out information about the layout

[3]Bent edges can be simulated by using dummy nodes [60].

objects above it on the hierarchy, the layout algorithm cannot find out the the other endpoints of edges the cross the boundary. Thus the algorithm cannot position its nodes to reduce these edge crossing.

Figure 3.2 show a layout generated by a hierarchy of layout algorithms. Since neither of the layout algorithm used to generated this layout can take into account the position of nodes laid out by the other algorithm, neither algorithm can position its nodes to eliminate the crossings. The edge crossings in this example can easily be manually eliminated (see Figure 3.3), but since the algorithms can't communicate, the crossings can't be eliminated automatically.

An alternative model would be an outside-in ordering. An outside-in model has the advantage that the user can focus on the entire structure of the graph. It has the disadvantage that layout algorithms will have to generate layouts that meet the criteria prescribed by the parent layout algorithm. In other words, an outside-in algorithm will have to assign a region for each of its children layout objects before any of them have calculated how large their generated layout will be. Thus all layout algorithms will have to be able to scale their generated layout to fit any arbitrary region that might be assigned by their parent.

Another possible model would be a multiple pass algorithm that allowed the different layout algorithms to communicate in both directions along the hierarchy. This approach would provide a more powerful programming model in which layout algorithms could take their neighboring layout algorithms into account. The problem with this method is that it would complicate the layout algorithms. It is not currently clear what information would be communicated between layout algorithms, how this information would be used by the algorithms, and if the algorithms would still be efficient enough for an interactive framework.

3.6 Advantages of Hierarchical Composition of Layout Algorithms

Hierarchical structuring of layout algorithms has several advantages over traditional monolithic layout algorithms. First, and most importantly, it allows users to compose new highly specialized layout algorithms without programming. Second, hierarchically composed layout algorithms are particularly good at clearly presenting the structures of subcomponents of the graph. Finally, it partitions the graph so that users can interact with individual layout algorithms [4], thus the user can manipulate portions of the graph customizing the layout to meet his current focus. Each of these advantages contributes to the global goal of allowing non-programming users to explore arbitrary graphs, creating meaningful layouts of subgraphs.

[4]Chapter 4 presents a mechanism that allows the user to interact with portions of the graph

CHAPTER 4
PARAMETERIZED LAYOUT ALGORITHMS

The goal this research is to provide a methodology that will allow the user to create custom layouts. Much of the task of generating layouts can be performed automatically by algorithms, but users often perceive changes to the generated layout that would improve it. The solution is to allow the user to edit the generated layout.

The changes made to a generated layout must be incorporated into the layout process or they will be lost the next time the layout algorithm is applied to the graph. Not incorporating the changes would force the user to repeatedly make the same edits. Thus any mechanism that allows the user to edit the generated layout should incorporate the changes. However, incorporating these changes back into the layout process is one of the most difficult problems with editing a generated layout [5, 39]. See Section 2.5 for a discussion of previous work on incorporating edits into the layout process.

There are two general approaches for allowing the user to customize a generated layout: edit the generated layout and interact with the layout algorithm. Editing the generated layout allows the user to make very fine grain adjustments, but does not necessarily allow him to make global changes—this is especially a problem in large layouts. It is also difficult to incorporate general edits back into the layout process.

An important goal of composing hierarchical layout algorithms is the ability to easily add new layout algorithms to the system. The problem with allowing general edits of the generated layout is that all layout algorithms would have to be able to handle all types of edits—recall that a single layout may be generated by a collection of layout algorithms. Otherwise the user would not be able to tell what edits were allowed in which parts of the generated layout. Handling general edits would significantly complicate the implementation of new layout algorithms contradicting the goal of making it easy to add new algorithms.

An alternative to editing the generated layout is to allow the user to interact with the layout process: customize the generated layout by modifying the layout process. This method has the advantages that the user can make global changes to the generated layout and can iteratively fine tune the layout. In addition, it is easier to implement algorithms that incorporate these type of changes into the layout process.

The remainder of this chapter introduces the concept of using parameters to control the layout process. Section 6.2 discusses a direct manipulation interface to the layout algorithm parameters which allows the user to interactively change the parameters making it possible for him to fine tune the layout by making several quick interactions.

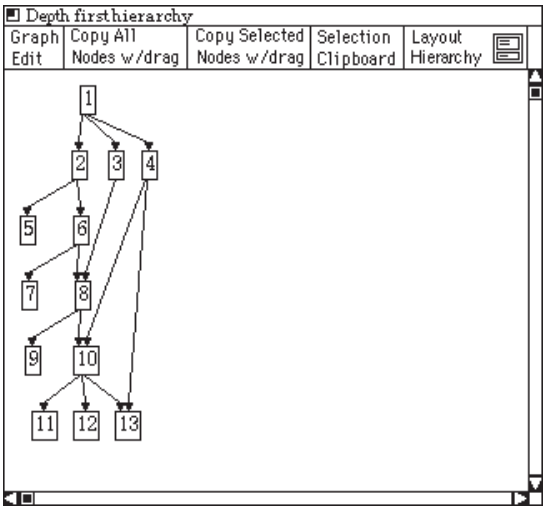

Figure 4.1: Depth First Search Ordering

4.1 Using Parameters to Control Layout Algorithms

The basic concept of parameterized layout algorithms is to extend them so they rely on parameters that can be imported. By allowing the user to change these parameters, she can customize the generated layout. The result is that instead of editing the generated layout, the user interacts with the layout algorithm which in turn generates a new layout.

The first step to parameterizing a layout algorithm is to identify aspects of the layout process that can be isolated. For example, the spacing between nodes in the generated layout is usually hardwired into the layout algorithm. This spacing, however, can easily be isolated from the layout process. The spacing parameter can be further refined by splitting it into two parameters; vertical spacing and horizontal spacing.

Once parameters have been identified, the layout algorithm can be extended to import them. The parameterization makes the layout algorithm more general and thus provides the end-user with more control over the final layout.

The process of converting a layout algorithm to rely on parameters is not fundamentally complicated. Most layout algorithms can easily be converted to rely on at least a few parameters. The general strategy is to find which parameters make the most sense in the context of the given layout algorithm. For example, the original implementation of the the hierarchical algorithm that generated the layout shown in Figure 4.1 was written to use a depth first search to order the nodes. The algorithm was later expanded to use either a depth first or a breadth first search—a parameter was added to determine which search method to use. Figure 4.2 shows a different layout of the the same graph as shown Figure 4.1. Both layouts were generated by the same algorithm, the only difference was that the search parameter for Figure 4.1 was set to "depth first" and it was set to "breadth first" for Figure 4.2.

There is very little overhead for incorporating a new parameter into a layout algorithm.

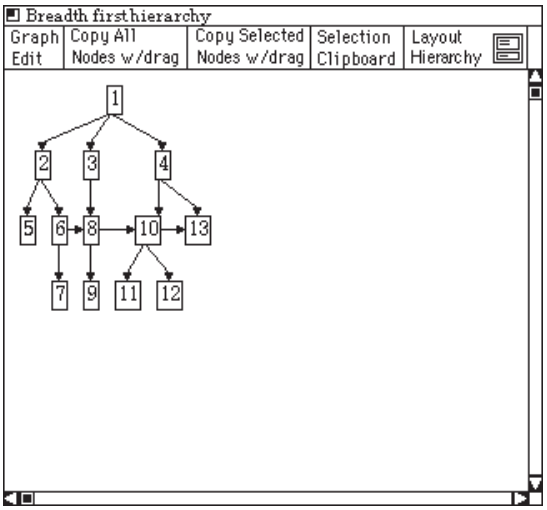

Figure 4.2: Breadth First Search Ordering

For example, adding the parameter of spacing to a layout algorithm is trivial. It only requires the addition of a call to the interface to get any changes to the spacing parameter. The majority of the work involved in the addition of the breadth first search to the hierarchical algorithm was in creating the search algorithm. Once the search algorithm was created, it was trivial to add it to the layout algorithm.

There are two classes of parameters: *scalar* parameters and *set* parameters. Scalar parameters can have numerical values like the spacing example, or can have enumerated types like the type of search algorithm used in the previous example.

Set parameters—sets of nodes and/or edges—allow the user to set the focus of the layout on a specific object or set of objects. As an example, the layout shown in Figure 4.3 was produced by the same hierarchical layout algorithm used in the previous examples. It allows any number of nodes to be specified as first-level nodes, also called root nodes. When the layout algorithm is parameterized by multiple root nodes, the result is a forest of trees. The layout shown in Figure 4.3 was generated by parameterizing the algorithm with the single element set {"1"}. Figure 4.4 shows the same graph laid out by the same algorithm parameterized by the root set {"100", "6","1","2", "101", "200"}. These two layouts are significantly different and each may be better than the other in different situations.

In order to give the user more control over the layout process, parameter sets are ordered—thus the ordering of the elements in a parameter set can be significant to the layout algorithm. For example, the hierarchical layout algorithm used in the last example uses the ordering of the nodes in the root parameter set to position the roots in the generated graph. Specifically, the first node in the parameter set is the leftmost node in the generated graph. The second node in the parameter is the second leftmost node, and so on. The generated layout can thus be customized further by changing the ordering of the nodes in the parameter set.

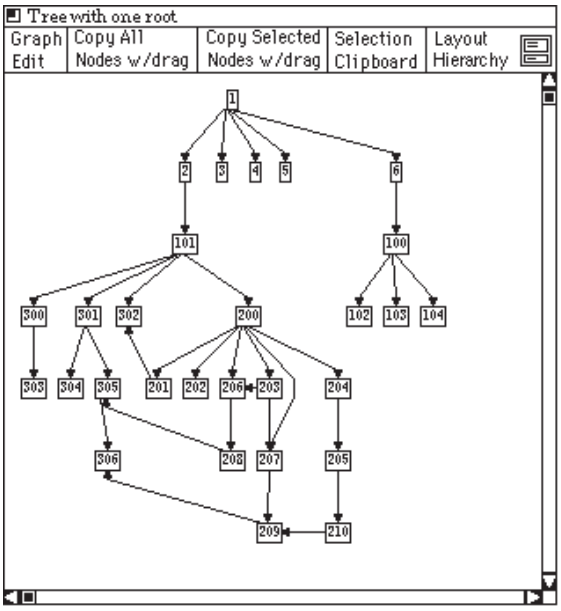

Figure 4.3: Tree with One Root

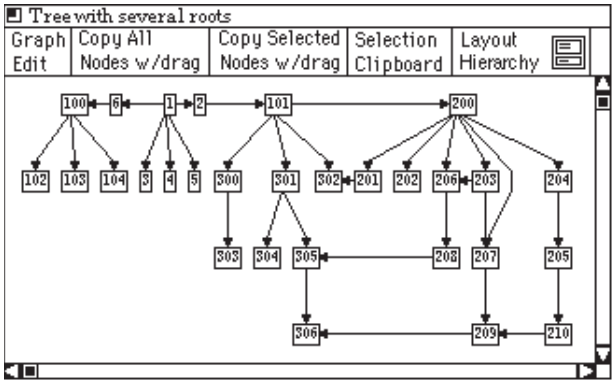

Figure 4.4: Tree with Multiple Roots—Same Graph Shown in Figure 4.3

4.2 Advantages of Parameterized Layout Algorithms

Parameterized layout algorithms allow the user to make local and global edits to the generated layout. In the hierarchical composition framework, the user can interact with the parameters of any individual layout algorithm and thus fine tune a small portion of the generated layout, while at the same time make global changes by adjusting parameters to the outer layout algorithms.

Parameterizing layout algorithms has the advantage of simplicity over a more com-

prehensive system of general edits. The algorithm parameterization approach does not require that all layout algorithm have a given set of parameters—each defines a unique set and an interactive interface to its parameters. Furthermore, incorporating changes back into the layout process is algorithm specific, thus global editing functions do not have to be incorporated into all algorithms added to the system. Thus instead of complicating layout algorithm implementations with interactive functionality, parameterizing layout algorithms provides support for the implementation of new algorithms.

CHAPTER 5
SUBGRAPH SELECTION

While traditional graph layout algorithms may do a good job laying out small graphs, they often don't scale well. Large graphs with hundreds of nodes are considerably harder to lay out. The main problem is that good layouts of a large graph may not exist. For example, if the connectivity is high, it might be impossible to create a readable layout—the minimum number of edge crossing could be so great that the graph is unreadable. The problem is that large graphs have too much information to be conveyed by a single canonical layout.

One solution is to divide the graph into smaller pieces and lay out the pieces independently from the remainder of the graph. The difficultly with this solution is how to divide the graph into meaningful subgraphs. Since the focus may vary from user to user, each user must be able to guide the subdivision process so she can partition the graph into subgraphs that reflect her current focus. The process of selecting a subgraph will simply be called *selection.*

A subgraph is traditionally defined as a set of nodes and a set of edges. In order to simplify the problem, this work will consider a subgraph to be only a set of nodes. This is an acceptable approximation because the subgraph can be defined as containing the explicit set of nodes and the implicit set of edges defined as all the edges—that exist in the total graph—between nodes in the subgraph. The following methodology thus only considers the selection of nodes and subsequently *selection sets* of nodes. Analogous methods could be used to explicitly include a set of edges in the subgraph selection process.

The following section presents the uses of subgraphs. Then some simple manual selection methods are discussed. Finally, the concept of algorithmic selection is introduced. Section 6.5 will present an interactive interface for selection.

5.1 Uses of Subgraphs

There are two processes in which the user needs to specify subgraphs: reorganizing/modifying an existing generated layout and creating a new, smaller generated layout.

The structure of a generated layout is defined by the layout hierarchy and by the distribution of nodes amongst the layout objects in the hierarchy. Moving nodes between layout objects will result in changes to the generated layout. Subgraph selection provides a tool for specifying the set of nodes to be moved from their current layout into a different one.

As an example of changing the distribution of nodes consider the layout shown in Figure 5.1. Suppose the user wishes to focus on in the ordering of the leaf nodes—the nodes drawn highlighted. Figure 5.2 shows a different layout of the graph shown in Figure 5.1. All the leaf nodes have been removed from the tree layout object and inserted into a

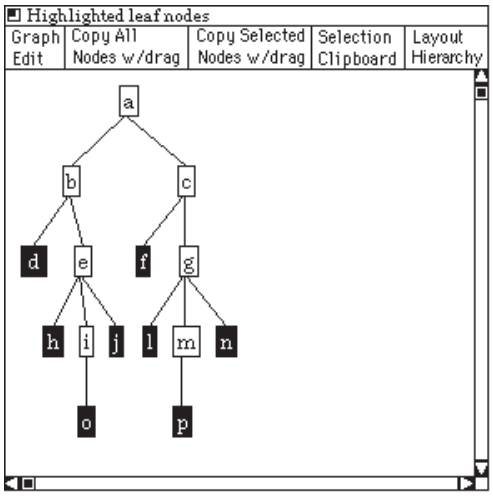

Figure 5.1: Tree with Highlighted Leaf Nodes

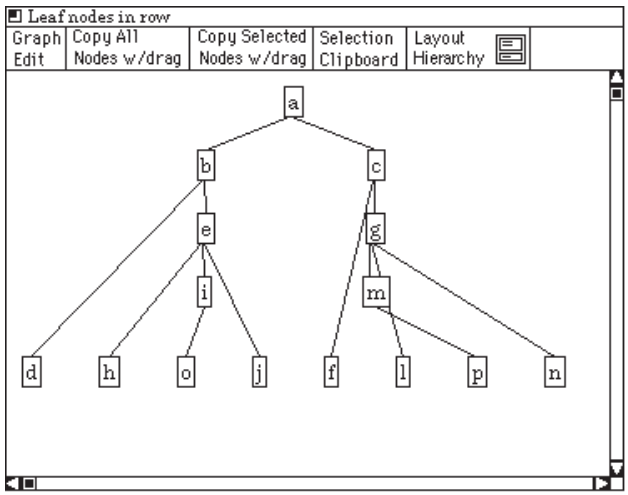

Figure 5.2: Tree in Figure 5.1 with Leaf Nodes in a Row

---

row layout object.

This graph was created by first creating a new column and a new row layout objects. The existing tree layout object and the new row layout object were placed inside of the new column layout object. Then the leaf nodes were selected and moved from the tree layout object to the row layout object. Finally the spacing for all three objects was interactively adjusted to reduce the edge crossings.

When evaluated using the metrics of traditional layout algorithms, the layout shown in Figure 5.1 is clearly a "better" layout than the one shown in Figure 5.2. There are no

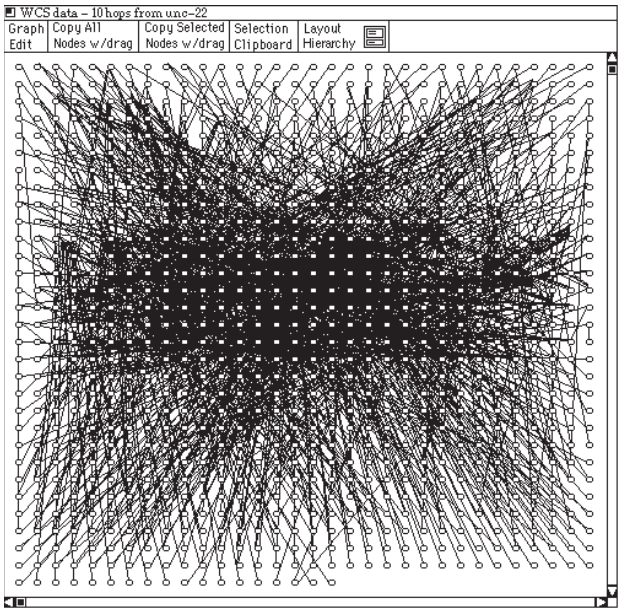

Figure 5.3: Nematode Reference Database Structure

edge crossings in the first layout, it is more symmetrical, and requires less screen space. However, the second layout focuses on the leaf nodes and does a better job presenting their ordering. Overall qualities of the layout have been traded for a better representation of a specific aspect of the graph.

Selection is crucial to the process for creating new views of a graph. (A view is an interface window coupled with a generated layout. Section 6.1 introduces the concept of views.) The main goal in creating a new view is to carve out a meaningful section of a large graph—a graph that is too large to lay out in its entirety. Selection provides the user with the mechanism to specify the part of the graph she is interested in.

Figure 5.3 shows a graph that is so large that it cannot be laid out in a readable fashion. (This graph represents the structure of a biology bibliographic database [27].) This layout was generated by a naive square grid layout algorithm.

Figure 5.4 shows a generated layout of small piece of the graph shown in 5.3. This layout was created by selecting a piece of the original graph and copying it into a new view. The new layout was then interactively edited to create the layout shown in Figure 5.4. (The methodology used to edit the new view will be discussed in Chapter 6). The following two sections present the selection process in detail.

5.2 Manual Selection

The simplest method for selecting a set of nodes is to select individual nodes with the mouse. This can be expanded by allowing the user to specify a region on the screen—drag a rectangle, draw a polygon, *etc.*—and then have the system automatically select all the

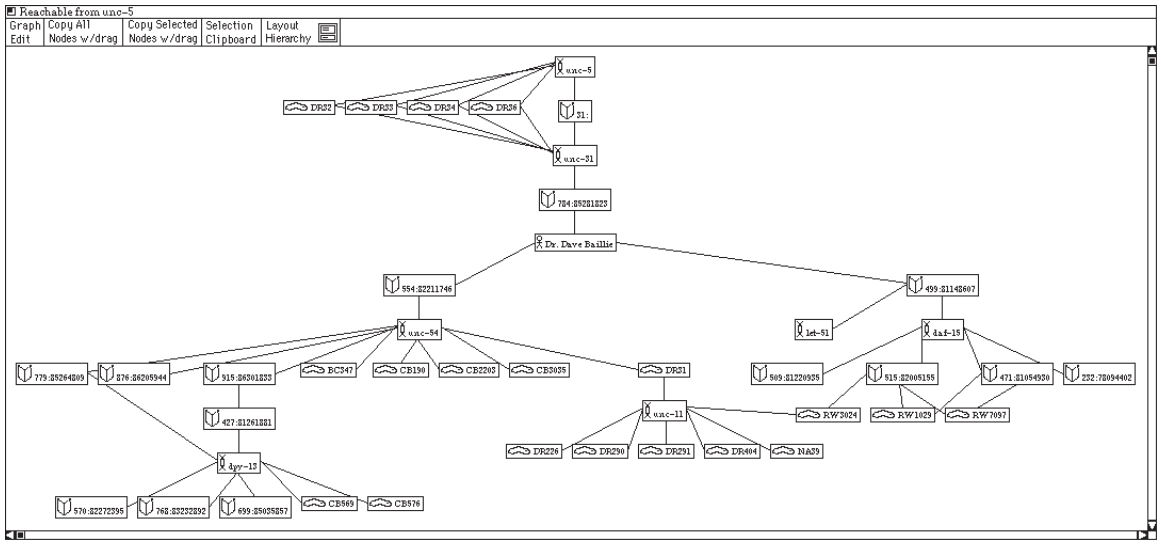

Figure 5.4: Subgraph of Graph Shown in Figure 5.3

nodes within the region.

This method, called *manual selection*, works well in small graphs in which the user can quickly find all the nodes he wants to select. It also works well when the target nodes are geographically grouped in the generated layout in such a manner that they can be enclosed in a rectangle or simple polygon.

Manual selection does not work well when the graph is large. For example, it would be difficult to manually pick out the nodes in Figure 5.4 from the layout shown in Figure 5.3. The following section introduces the concept of *algorithmic selection* which uses graph traversal and set operation algorithms to select nodes.

5.3 Algorithmic Selection

The idea of algorithmic selection is to use graph traversal algorithms to specify the selection set. One such algorithm—reachable nodes—selects all the nodes within a given distance (number of edges that must be traversed) from a given node. The nodes in the layout shown in Figure 5.4 were selected from the graph shown in Figure 5.3 by applying the reachable algorithm from a single node of interest (the root of Figure 5.4).

In this example the reachable selection algorithms was parameterized by the node labeled "unc-5" (used by the algorithm as the root of the traversal) and the integer 10 (the distance). The result is all the nodes within 10 edge traversals of the node "unc-5" were marked as selected and could then be copied into the new view.

As another example of algorithmic selection, consider the graph shown in Figure 5.5. Suppose the user is interested in the shortest path between the highlighted nodes "root" and "destination." The shortest path selection algorithm can be applied to this graph

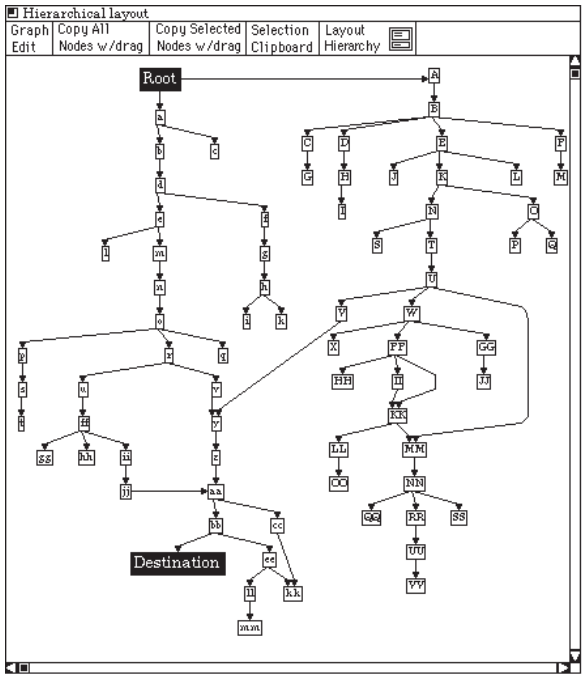

Figure 5.5: Sample Connected Graph

to find the shortest path between these nodes. The result of applying the shortest path algorithm parameterized by "root" and "destination" is shown in Figure 5.6.

Once the shortest path has been selected, the user can create a new layout that highlights it. Figure 5.7 shows a new layout of the exact same graph shown in Figure 5.6. In this new layout the selected nodes—those on the shortest path—have been moved into a row layout object. The layout shown in Figure 5.7 highlights the shortest path between the "root" and "destination" nodes. On the other hand, the layout shown in Figure 5.6 clearly presents the graph as two adjacent trees.

The reachable and shortest path algorithms used in the above example were parameterized by nodes in the graph—the two endpoints in the path example and the root node in the reachable example. Before these algorithms can be applied, their parameters have to be selected. The graph in the path example is small enough that the user can manually select the endpoints using the mouse. On the other hand, the graph in the reachable example is so large that it would even be hard to manually selected the single node "unc-5."

To alleviate this problem, selection algorithms can be used to select the parameters for other selection algorithms. In the reachable example, the root parameter node was selected using a regular expression selection algorithm. This algorithm takes a regular expression as a parameter and searches the title strings of all nodes in the graph selecting nodes with titles that match the regular expression.

The notion of using one selection algorithm to select the parameters for another can be

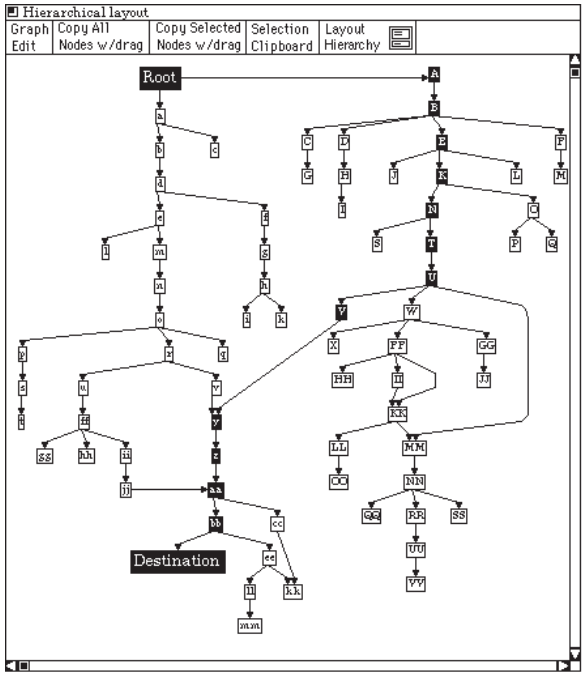

Figure 5.6: Shortest Path—Same Graph Shown in Figure 5.5

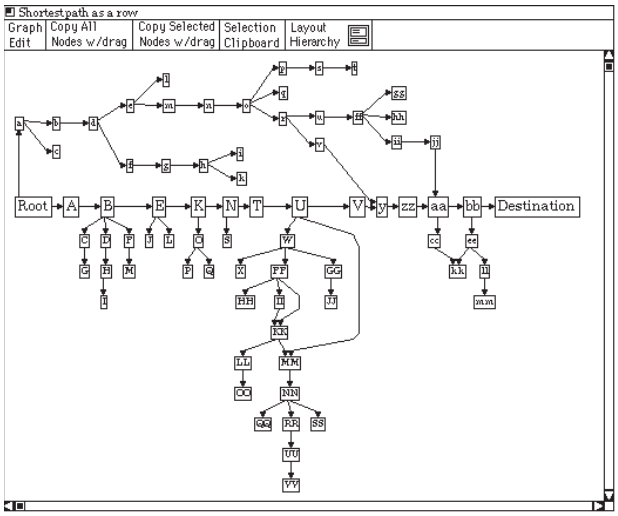

Figure 5.7: Modified Connected Graph—Same Graph Shown in Figures 5.5 and 5.6

generalized by allowing any combination of selection algorithms to be connected together in arbitrary configurations. The result is a simple dataflow programming language that allows the non-programming end-user to create new selection algorithms by plugging together existing algorithms.

In order to reduce runtime type checking, all selection algorithms are designed to accept sets of nodes as their input parameters. This simplifies the programming model and reduces the syntactic checking that has to be performed at program creation time. However, forcing all algorithms to accept sets as inputs may change the semantics of some algorithms.

For example, the shortest path algorithms must be implemented as a shortest paths algorithm. This is because the algorithm must accept two sets of nodes as its parameters—instead of the traditional two nodes. Thus it becomes a shortest paths algorithm in that it must calculate the set of shortest paths between each of its input sets.

In addition to selection algorithms, set operation algorithms can be used. Some set operators, such as set complement, require the entire graph as a parameter. These operators have both explicit parameters and the entire graph as an implicit parameter. A special set operator that returns all nodes in the graph has the entire graph as an implicit operator and no explicit parameters.

Figure 5.8 shows an example selection program that contains both selection algorithms and set operations. This algorithm selects the two most connected nodes in the graph and their immediate neighbors. The first selection algorithm selects the `most connected node`. The left branch of the program then finds all the `neighbors` for this node (the nodes within a single edge traversal). The right branch of the program first finds the second most connected node in the graph. This is done by applying the `most connected node` selection algorithm to the graph after the first most connected node was removed (it was removed by the `set compliment` operation). The results of the left and right branches are then combined using the *join* operation.

Section 6.5 presents a visual programming interface for creating selection programs.

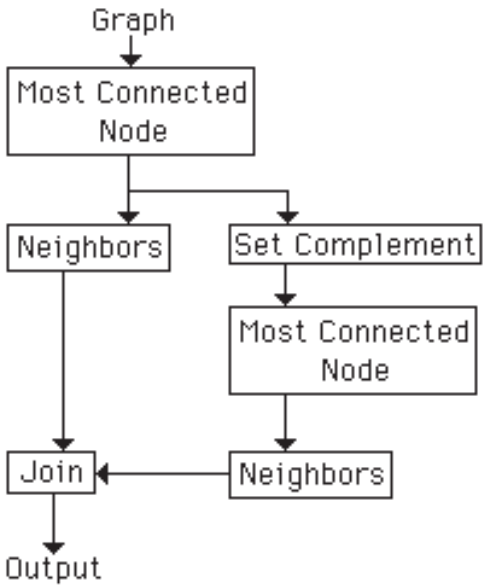

Figure 5.8: Composition of Selection Algorithms: Selects the Two Most Connected Nodes and Their Immediate Neighbors

CHAPTER 6
INTERACTIVE GRAPH LAYOUT

The goal of interactive graph layout is to provide tools that allow the user to explore large graphs and learn about them in reasonably sized pieces. In order to create a learning/exploration environment, the graph layout system must not create a barricade between the user and the graph, but allow the user to easily and quickly try many different alternatives.

This section presents an interactive methodology that incorporates the concepts presented in the previous three chapters—hierarchical composition of layout algorithms, parameterized layout algorithms, and subgraph selection. The main goal of this methodology is to allow the user to create and modify multiple layouts of a single graph. This set of layouts can be used to guide the user in the creation of new layouts and ultimately to help them explore the graph and understand the data encoded within it.

Throughout this section, examples are drawn from the interfaces used in the prototype system. The individual interfaces are not as important as the system's overall functionality. The prototype was created to provide a proof of concept that graph layout can be interactive and that an interactive graph layout system would provide the end-user with previously unobtainable power to create customized graph layouts.

The following section discusses methods for creating multiple layouts and copying nodes between them. Interactive methods that allow the user to modify layouts will then be introduced.

6.1 Multiple Views

The best method for creating a good layout of a given graph is not always initially obvious to the user. Thus she needs the power to experiment with many different layouts until she finds one that looks good to her and meets her current needs.

Providing the user with simultaneous multiple layouts will help guide him in the creation of new and potentially better layouts. The standard workstation windowing model can be used to manage the multiple layouts.

In the prototype system, individual layouts of the graph are combined with an interface and associated with a window. This encapsulated layout and interface is called a view of the graph, or *view* for short. All the current views are managed by a window manager that allows the user to create new empty views via menu selection and handles copying nodes and edges between views. Figure 6.1 shows four views of a single graph generated in an interactive fashion. Each view is more specific and thus shows more details.

In the prototype interface, nodes are copied between views by dragging a button (labeled "copy node") from the source view into the destination view. There are two modes for copying nodes. The first is to copy all nodes from the source view into the destination

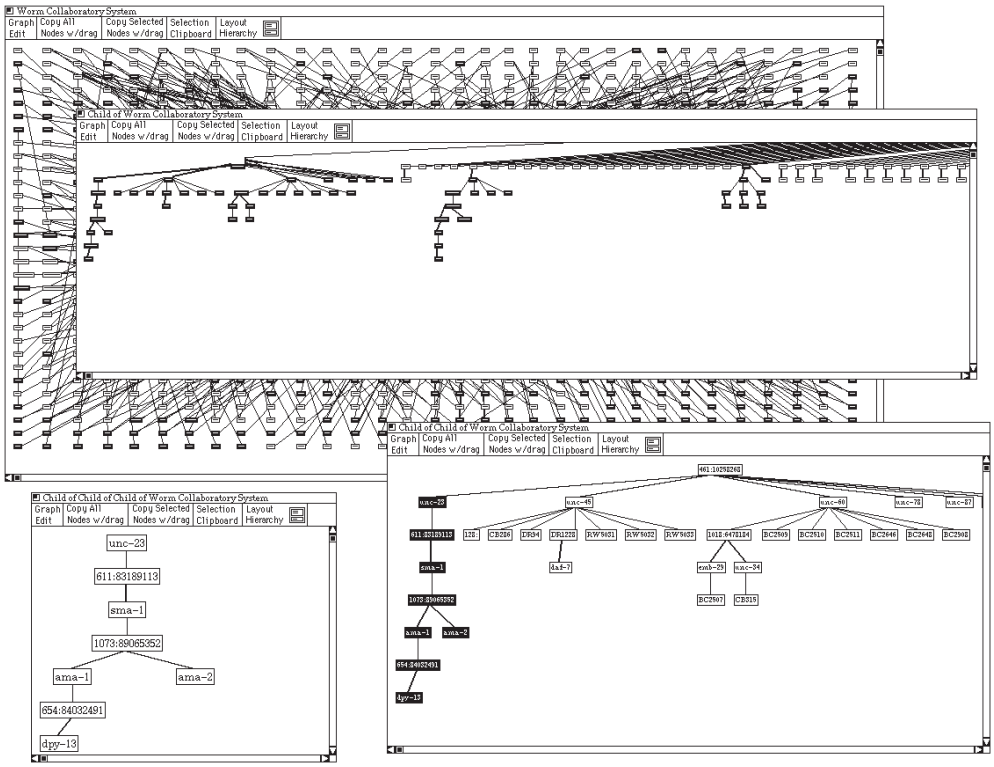

Figure 6.1: Multiple Views of a Single Large Graph

---

view. The second is to copy only the *currently selected* nodes.

There is a potential problem with replicating node objects; multiple copies of the same node could exist in a single layout. The goal of multiple views is to allow the users to simultaneously examine different generated layouts of a graph. If nodes are allowed to be multiply copied into a single view, the view will no longer be a legal subgraph of the original graph. This problem can be avoided by using set union to paste objects into a layout. Thus only one copy of any node or edge can be pasted into a single view.

In order to avoid explicit specification of which edges are to be pasted into a view, the set of edges to be pasted is calculated automatically. After all the nodes have been pasted, all edges in the graph are considered for insertion into the new view. If copies of both of the nodes the edge is connected to have copies in the new view and a copy of the edge is not already in the view, the edge will be copied into the view.

A view is more than a collection of nodes and edges. It also contains a hierarchy of layout objects. If nodes and edges were simply pasted into an empty view, all the layout information would be lost. Thus before a node or edge can be copied into a new view, the layout object they belong to must be copied.

Layout objects—except the root layout object—are nested inside of a layout object. Thus before a layout object can be copied into a new view, its parent layout object must be copied. The result is that before a node or edge can be copied into a view, all of the

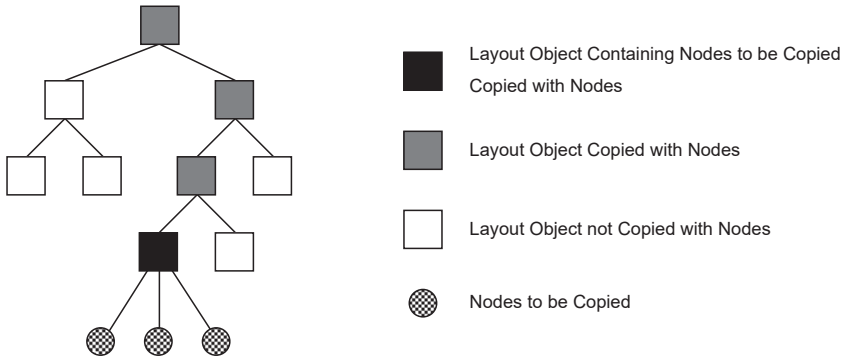

Figure 6.2: Sample Hierarchy of Layout Objects: Copying Layout Objects when Nodes are Copied

---

layout objects above it in the hierarchy must also be copied. Figure 6.2 illustrates this point. It shows a sample hierarchy of layout algorithms with a couple of nodes marked to be copied. All the layout objects that exist above the nodes must be copied into the new view before the nodes can be copied. These layout objects are shaded (black and grey) in Figure 6.2.

The node pasting algorithm proceeds as follows. First all the layout objects that will have nodes in the new view are marked. This is done by having each node marked "to be copied." The layout objects in turn mark their parent layout objects to be copied. This process continues up the hierarchy until the root layout object is reached.

The next step is to copy all the layout objects that have been marked into the new view—only layout objects that do not already exist in the destination view are copied. This process must start at the root layout object because it is the only layout object without a parent layout object. After all the layout objects have been copied and inserted into the destination view, all the node objects are copied and inserted. As mentioned above, the edges are then copied.

Each view has exactly one root layout object. If nodes are copied from two distinct views, then the root layout objects from both original views will be pasted into the destination view. The destination view will then have two different root layout objects. This problem is solved by creating a new root layout object—by default a grid layout algorithm—and inserting all the root layout objects into it. Thus pieces from different graphs can be combined into a single layout.

6.2 Interactive Parameterization

Parameterization of layout algorithms gives the user some control over the layout process and thus allows them to customize the generated layout. Since graph layout can be complicated, it is not always initially clear to the user how to best customize the generated layout. Providing an interactive interface to the algorithm's parameters creates an envi-

ronment in which the user can experiment with different parameters until the resulting generated layout meets her needs.

Each layout algorithm has a set of parameters associated with it. An interface to the layout algorithm parameters must be created for each unique set of parameters. Similar algorithms often have the same set of parameters and can share the interface definition. For example the **row** and **column** layout objects share an interface definition.

In order to reduce the distance between changes made to the interface and the resulting changes to the generated layout, the layout is recalculated after every change to the interface. For example, the row layout algorithm allows the user to specify the spacing between the nodes. The interface for the row layout algorithms thus provides a slider for changing the spacing. As soon as the user makes a change to the spacing slider, the layout is recalculated and the new generated layout is immediately drawn on the screen—Section 6.6 presents a screen update methodology that improves performance.

The result of interactive parameterization is that the user can easily change the generated layout and immediately see the result. Since changes are so easy to make and the results immediately seen, the user can quickly make many changes and thus iteratively fine tune the layout. The advantage of this approach is that several small changes can result in drastic changes in the generated layout. Thus the user has great freedom to customize the generated layout.

The drawback of instantly changing the screen is that the user may be surprised by drastic changes to the layout. However, if an edit drastically changes the generated layout, the user will see the current layout jump in a single step to the new layout. If this change is not what the user wanted, he can simply undo the edit. On the other hand, if the user likes the new layout but does not understand how his edit caused the change, he can repeatedly undo and redo the edit so he can better understand how it affected the generated layout.

Since each layout object has an interface associated with it, there may not be enough screen space to draw all the interfaces. The following section introduces a method for managing the interfaces and presents a sample interface to the hierarchical layout algorithm.

6.3 Providing an Interface to Layout Algorithms

The hierarchical compositions of layout algorithms provides the end-user with a tool for creating new layout algorithms. In order to understand and customize a composed layout algorithm, the user must be able to view, edit the structure of the layout algorithms. However, the hierarchical structure of layout algorithms is not always explicitly apparent in the generated layout.

In the prototype interface an interactive structure called a *metagraph* is provided to explicitly present the structure of layout algorithms and to furnish a direct manipulation interface to this structure. Figure 6.3 shows a sample composed layout with the metagraph interactor open.

The layout shown in Figure 6.3 is composed out of six layout objects. The outermost object, named "row_of_trees," is a row layout that contains two tree layout objects named

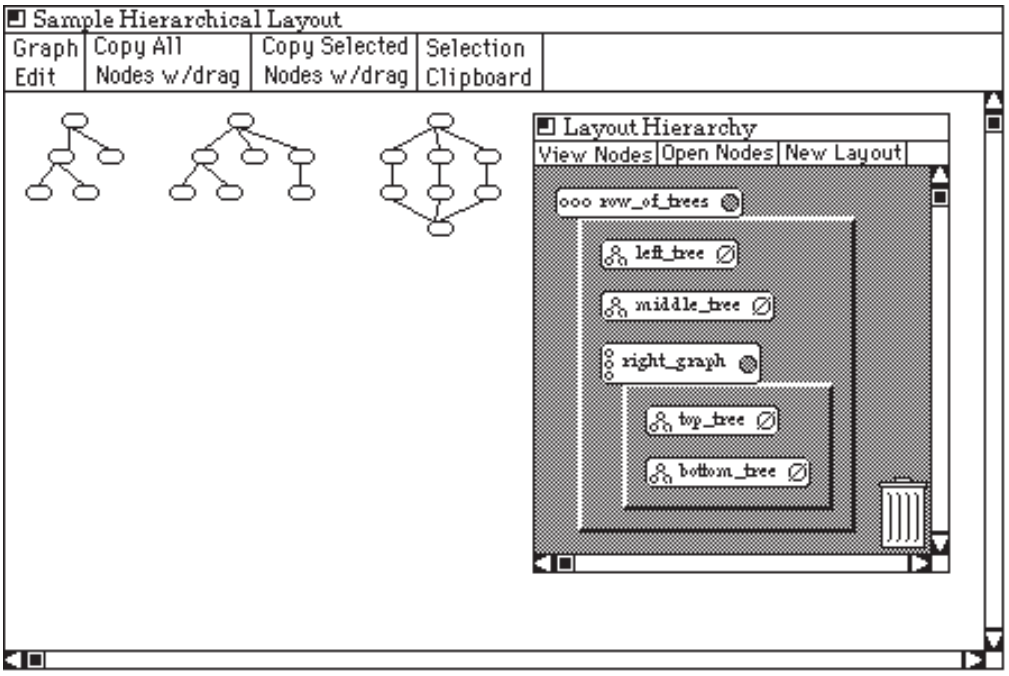

Figure 6.3: Sample Hierarchical Layout with Open Metagraph

"left_tree," "middle_tree" and one column layout object named "right_graph." The column layout in turn contains two more tree layout objects named "top_tree" and "bottom_tree."

Each layout object in the view is represented by a node—called a *metanode*—in the metagraph. The six nodes in the metagraph shown in Figure 6.3 represent the six layout objects in this example graph. The structure of the metagraph represents the structure of the layout objects. Metanodes are positioned inside of a box bounded by the metanode it belongs to. For example, the layout objects "top_tree" and "bottom_tree" in Figure 6.3 belong to layout object "right_graph." Thus the metanodes for "top_tree" and "bottom_tree" are drawn inside the metanode for "right_graph."

There are four parts to each metanode. The first is an icon that depicts the layout algorithm. The topmost metanode in Figure 6.3 represents a row layout algorithm and thus its icon is a few nodes laid out in a row. Each layout algorithm has a unique icon that attempts to represent the general structure of the layout generated by the algorithm. The second part of the metanode is the layout algorithm instantiation name—each instantiation has a unique name that is either explicitly specified or otherwise generated by the system. For example, the names `row_of_trees` and `left_tree` in Figure 6.3 are layout algorithm instantiation names.

The next part is the circular interactor on the far right of the metanode. This interactor acts as a "has children" indicator and allows the user to open and close the metanode so it does or does not display its children—the metanodes below it in the hierarchy. If the metanode is open, it is enlarged and its children are drawn inside of it representing the

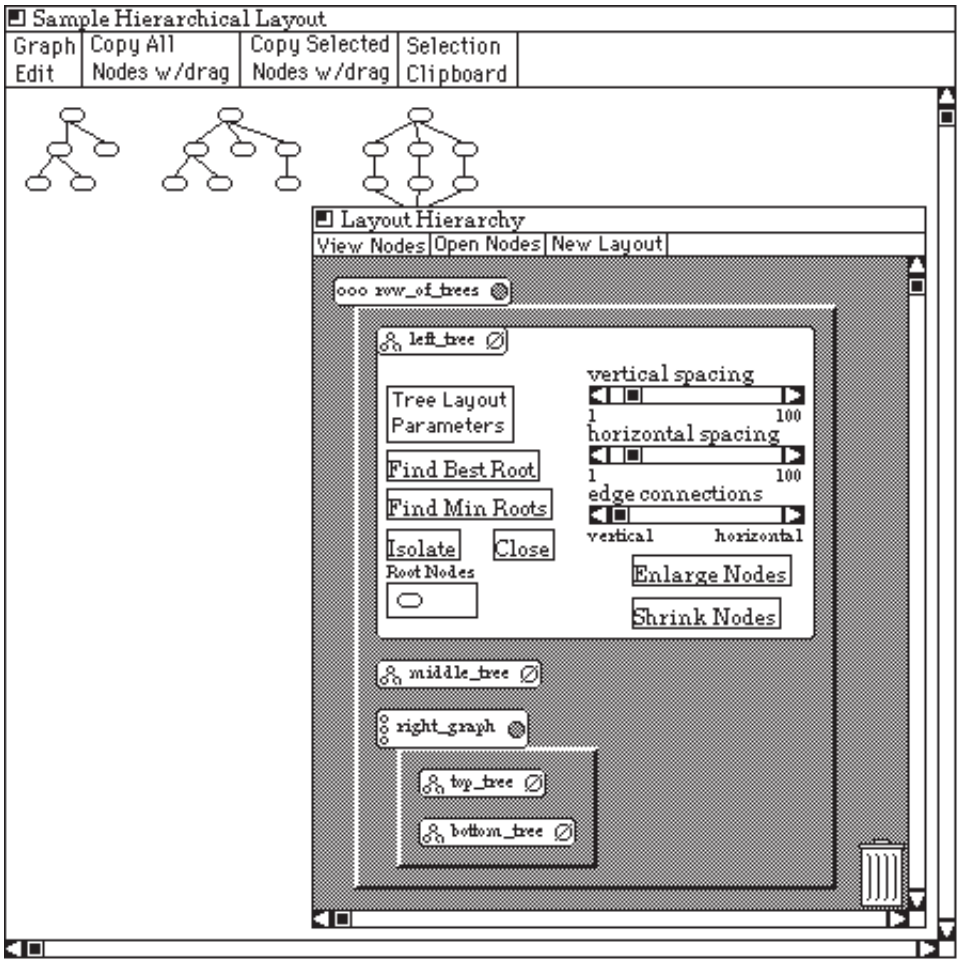

Figure 6.4: Metagraph with an Open Control Panel

---

structure of the hierarchy of layout algorithms.

Each instantiation of a layout algorithm has a direct manipulation interface associated with it. The metanode provides access to this interface by allowing the user to open and close it by clicking anywhere within the metanode. Figure 6.4 shows the metagraph shown in Figure 6.3 with the control panel for the leftmost layout algorithm opened.

Different instantiations of the same layout algorithm can optionally be specified to share the same interface. This allows the user to manipulate a set of instantiations of layout algorithm with a single interaction. Since different layout algorithms have unique interfaces, different algorithms cannot share interfaces. Shared interfaces look just like non-shared interfaces—the user opens and closes them from the metanode and uses them just like other layout algorithm interfaces. However, only one copy of a shared interface is drawn on the screen at any one time.

The opening and closing of layout algorithm interfaces and the optional displaying

of the metanode's children changes the size of the metanode. Thus the metagraph must rearrange the metanodes whenever any of them change size. This problem is handled by using a graph layout algorithm to position the metanodes. Metanodes are simply a custom node type, and the metagraph is a generated layout. In other words, the interface to the metagraph is built out of the same mechanism used to generate the graph layouts.

In addition to illustrating the structure of layout algorithms and managing the interfaces to the algorithms, the metagraph allows the user to manipulate the structure. There are three basic functions: reorder the hierarchy, add/delete layouts, and move nodes between layouts.

The user can reorder the hierarchy of layout algorithms by rearranging the metanodes in the metagraph. Nodes can be dragged with the mouse to any position in the hierarchy. As soon as the user releases the mouse button, the hierarchical ordering of layout algorithms and the metagraph are reorganized. The resulting new layout is immediately calculated and redrawn.

Instantiations of layout algorithms can be deleted from the hierarchy by dragging the corresponding metanode to the trash can icon. New instantiations can be created through menu selection. The metanode for the new instantiation is placed into the hierarchy by dragging it from the creation menu. The associated instantiation is then inserted into the algorithm hierarchy.

All nodes belong to exactly one layout object. The metagraph provides an interface for moving nodes from one layout algorithm to another. Nodes are moved by first selecting the nodes to be moved. Once a set of nodes has been selected, all the nodes in the set—even if the nodes don't start in the same layout algorithm—can be moved to a different layout object by dragging the "Copy Node" button into the metanode for the destination layout object.

There are two shortcomings of the layout hierarchy creation method. First, there is no mechanism for reuse of layout algorithm hierarchies. Second, graphs must be manually partitioned and manually inserted into the layout algorithm hierarchy. Section 9.2.3 presents some possible solutions to these shortcomings.

6.4 Interaction with Individual Node Objects

Several aspects of the generated layout cannot be controlled by the global interface method presented in the previous section. For example, while the size of all the nodes in a layout could be changed via the layout algorithm's interface, changing the size of a single node would be difficult. The solution is to provide an interface, called a *node control panel*, for every node.

Figure 6.5 show a sample control panel to a basic text node—the node labeled "1". In this example, the ordering of edges (see below), size, font, and shape of the border, can be interactively changed by the user. Since nodes can be arbitrarily defined (see Section 7 for details about node definitions) they can have different variable parameters. Thus each node type has an arbitrarily defined interface.

The order of the arriving (and leaving) edges of a node can be significant. Simple layout algorithms may use the ordering of the edges to guide the layout process. Nodes

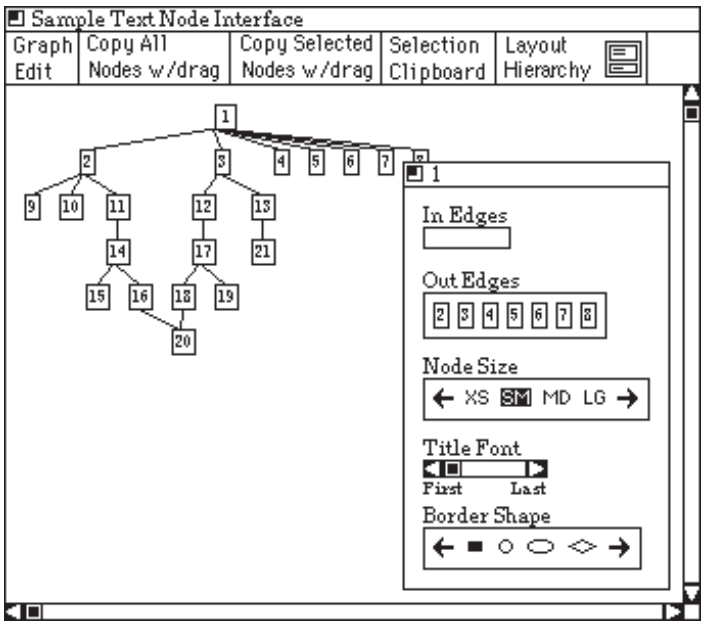

Figure 6.5: Interface to a Text Node

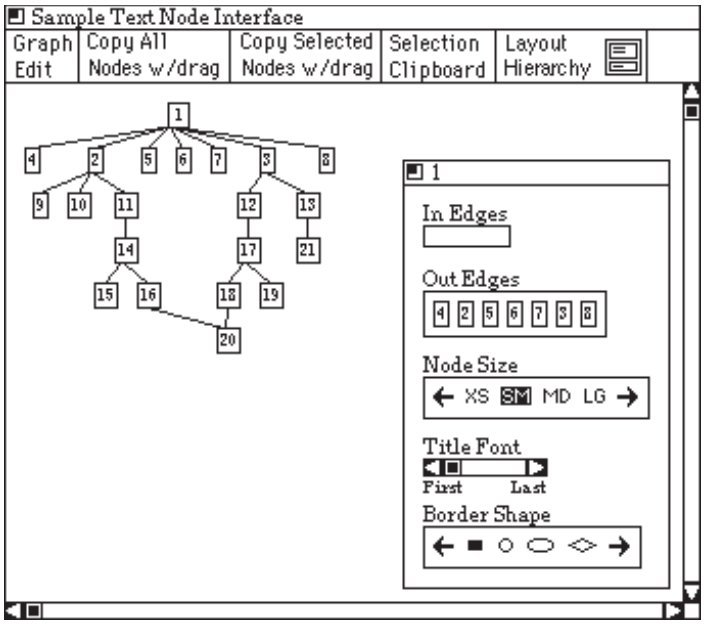

Figure 6.6: Using Node Interface to Reorder Out Edges of Node "1"

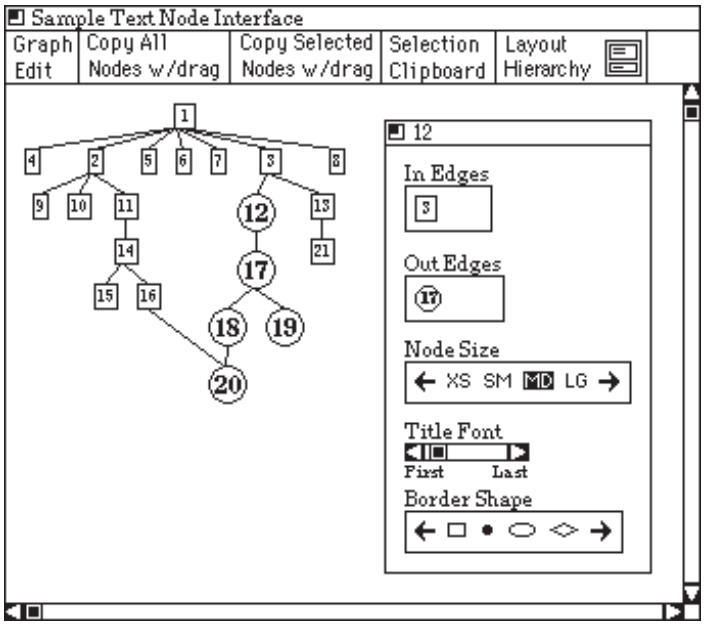

Figure 6.7: Changing the Size, Shape and Font of Nodes

thus have ordered lists for their arriving and leaving edges. The order of edges corresponds to the order in which they were declared in the graph specification script.

Since the order of edges can affect the generated layout, the user must be allowed to change it. In the sample interface shown in Figure 6.5, the interactors labeled "In Edges" and "Out Edges" allow the user to change the ordering of the arriving and leaving edges by rearranging the icons—since edges usually all look alike, an icon of the node at the other end of the edge is used to represent the edge. In this example, the represented node has eight leaving edges and no arriving edges. The layout shown in Figure 6.6 was created by rearranging the out edges for the node labeled "1" in Figure 6.5. Note that for both graphs the ordering of the children of node "1" is the same in the generated layout and the node's control panel.

While some layout algorithms use the ordering of edges to guide the layout process, others change the ordering. For example, Sugiyama [60] type layout algorithms rearrange the ordering of edges to reduce edge crossings. If the user were to manually change the edge ordering, the algorithm would simply change the ordering back. A locking mechanism allows the user to mark a manual edge ordering so layout algorithms will not undo the ordering. While some layout algorithms can be modified to abide by manually specified orderings, locking the ordering is only a suggesting and thus the layout algorithms don't have to abide by it. In order to prevent confusing the user, the edge reordering interactor can be disabled for nodes that belong to a subgraph that is laid out by an algorithm that can't handle locked edge orderings.

In addition to reordering the edges, the node control panel allows the user to change aesthetic aspects of individual nodes. For example, the text node control panel shown

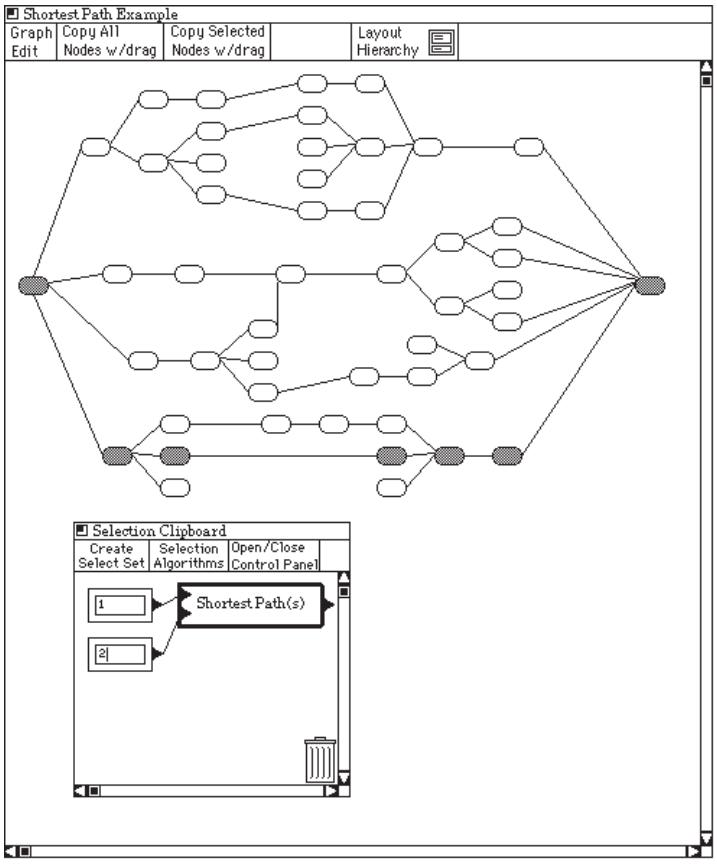

Figure 6.8: Selecting the Shortest Path

above, allows the user to change the size, shape and font used to draw the node. Figure 6.7 shows the same layout as in Figure 6.6 except the aesthetic aspects of several nodes have been changed.

6.5 Interactive Subgraph Selection

This section presents a visual language interface to subgraph selection. Using a visual language provides the non-programming end-user with an environment in which she can create new selection algorithms by plugging together existing algorithms. This section first introduces the components of the selection visual language and presents the programming interface. Then the program evaluation mechanism is presented.

Associated with each view is a pop-up interactor called the *selection clipboard*. It provides a palette for building the visual program. Figure 6.8 shows an open selection clipboard that contains a program to calculate the shortest path between two nodes.

There are three types of programming objects in this language; manual selection objects, selection algorithm objects, and dataflow links. Manual selection objects represent

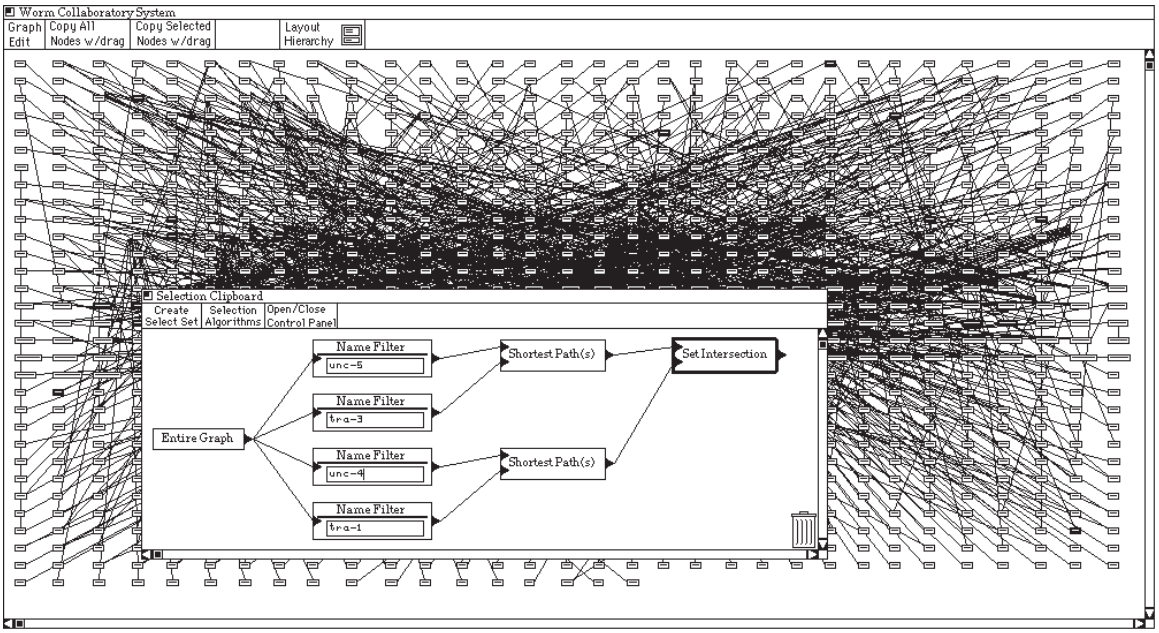

Figure 6.9: Intersecting Paths

a set of nodes that can be specified by manually selecting nodes with the mouse. Manual selection sets are used as input parameters for selection algorithms.

Selection algorithm objects apply graph traversal algorithms to the graph. They are parameterized to allow the user to control the traversal. Dataflow links simply connect the input and output parameters—represented by ports drawn as black triangles—of manual and algorithmic objects.

In the selection program shown in Figure 6.8, the two objects labeled "1" and "2" are manual selection objects. (Manual objects can be renamed. The numerical names are the default names generated by the system). In this example, the first manual selection set contains the leftmost node in the graph. The second contains the rightmost node. The two manual selection objects are connected to the input ports of the **Shortest Path(s)** selection object. This object is associated with a selection algorithm that calculates the shortest paths between the nodes specified as its inputs.

Each selection object also has a set of nodes associated with it. This set is calculated by applying the object's algorithm to the input parameters. In the above example, the **Shortest Path(s)** object contains the set of all node on the shortest path between the leftmost node—the node specified in the manual object "1"—and the rightmost node—the node specified in "2."

Associated with every view is a current selection set—manual or algorithmic. In the prototype system, the current selection set is specified by picking the corresponding selection object in the selection clipboard. In the previous example, the **Shortest Path(s)** object has been picked with the mouse. The result is that the selection object is high-

lighted with a thicker border and the nodes in the corresponding set are draw highlighted. Thus since the shortest path selection object is highlighted, all the nodes one the shortest path are drawn highlighted.

As a more complicated example of a selection program, Figure 6.9 shows a program that calculates the intersection of two paths. Each path is calculated by the **Shortest Path(s)** selection object. Since the graph is so large, the input parameters for the path algorithms are calculated by the **Text Filter** selection object. This object searches all the nodes in its input set for titles that match the regular expression the user has typed in. The resulting paths are then passed to the **Intersection** selection object. Any node that exist in both paths will be selected.

In order to simplify the language, there is only one data type: set of nodes. Thus the input parameters and output values are all standardized to accept and produce a set of nodes. This reduces the syntactic analysis, and simplifies the programming model. Only two syntactic checks have to be performed: assure that all connections are between an input port and an output port of different objects, and that a cycle is not completed by the new link.

Links are created by selecting a port and dragging a rubber banded line to another port. When the new link is dragged near a valid connection point, the link will snap to that point. The snapping performs the syntactic check and provides the user with the semantic feedback that it is a legal connection point for that link [25]. When the user releases the mouse button, a link is created only if it has snapped to a valid port.

Just like the metagraph interface, the selection interface is built using the system's own graph layout algorithms. Thus the selection visual program is automatically laid out after every change to the program. In fact it is laid out by the same hierarchical layout algorithm used to lay out many of the examples in this dissertation.

Whenever an object or link is added to the visual program, the position of all the objects in the program is recalculated and the program is redraw using the new layout. This approach has the advantage of relieving the user of the task of repositioning all the programming objects as the program grows. The disadvantage is that the user no longer has complete control over the layout of the program. The user can however, interact with the parameters to the layout algorithm.

The interactive interface to the selection mechanism allows the user to quickly create a subgraph which can then be copied into a new view. The first example in this section (Figure 6.1) shows four views of a single large graph. Each of these views was created by first selecting a set of nodes and then copying the nodes into a new empty view. The process is quick enough that several views can be created in a few minutes.

The following section introduces an efficient algorithm for determining when a layout must be recalculated in order to keep the screen up to date.

6.6 Screen Update

There are two aspects associated with screen updates, generating a new layouts and redrawing the screen. Input events are the only mechanism that can change a layout. All layouts are thus known not to have changed between input events. Therefore is it adequate

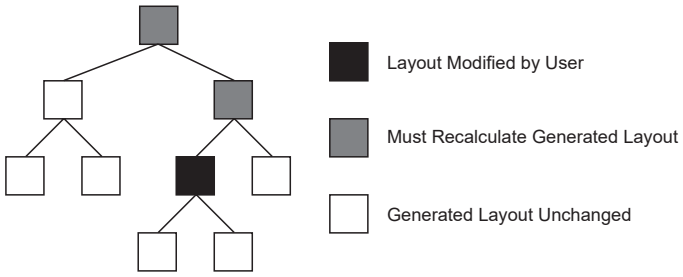

Figure 6.10: Propagating Changes in Generated Layouts

---

to check each layout only after input events are consumed by a screen object.

After every new input event that is consumed, the system queries all root layout algorithms for changes to the generated layout—changes to its parameters. When queried for changes the layout algorithm recursively queries its child layouts and nodes for changes. This is similar to the actual layout process described in Section 3.2. Since all layouts exist in a strict hierarchical structure, it is sufficient to query only the root layouts for changes.

If a node object is changed by the user, it may affect the layout it belongs to. If a generated layout is changed directly by the user or if one of its node changes, the change may affect all the layouts above it in the layout algorithm hierarchy. It is thus necessary to recalculate all the generated layouts in the hierarchy above the changed node or changed layout. On the other hand, it is not necessary to recalculate any generated layouts that are below the changed layout. Figure 6.10 illustrates the propagation of changes to generated layouts. Each box in this figure represents a layout algorithm in the algorithm hierarchy. In this example, the user has changed the layout represented by the black box. All the layout algorithms represented by the grey boxes may rely on the changed layout and thus must recalculate their generated layouts. The layouts represented by the white boxes cannot be affected by the change and thus do not have to recalculate their generated layouts.

The above algorithm proceeds in a top down fashion from the root of the layout algorithm hierarchy looking for changed layouts. As soon as it reaches a layout algorithm who's children layouts have not changed, that layout algorithm is applied to produce a new generated layout. As the algorithm progresses backup up the layout algorithm hierarchy, each algorithm calculates a new layout.

CHAPTER 7
PROTOTYPE IMPLEMENTATION

The prototype is implemented in approximately 25,000 lines of C++. It uses the Arizona Retargetable Toolkit [25] for graphics support and input event handling. The majority of the code handles the interactive aspects of the system: only 10% of the code is dedicated to graph layout algorithms.

The prototype has been designed as a framework to which layout algorithms, node definitions, edge definitions, and selection algorithms can be added without any changes to the prototype code. This has been achieved by using the inheritance mechanism in C++[59]. New objects and algorithms can be created by inheriting behavior from existing objects and algorithms.

For example, to add a new node type the user must specify the following:

- Constructor function[1].
- Destructor function.
- Drawing routine.
- Customized control panel (optional, default is used otherwise).
- Convex hull description.

Much of the functionality of a new node can be inherited from the existing base class of nodes. The following list contains some of the functionality provided by the base node class.

- Adding edges.
- Deleting edges.
- Connecting edges to the convex hull bound.
- Selection mechanism.
- Calculating the convex hull.
- Changing the size, shape, and shading.
- Default control panel.
- Opening/closing control panel.

In addition to these node specific functions, the system provides a library of drawing functions and a collection of interactor definitions. Thus the creation of a customized node control panel is merely a collection of interactor declarations.

The only restriction on node definitions is that they must be able to return a convex hull. Thus nodes can take on a large variety of shapes and functionality. Since the each node type definition includes drawing routine, nodes can take on any graphical appearance.

The metanode (see Section 6.3) is an example of a complex node. Metanodes contain interactors to open their children and to open the interface for the layout object they

[1] See [59] for the definitions of constructor and destructor functions.

represent. As the metanode opens and closes it simply reports the appropriate convex hull to the algorithm laying it out.

A collection of functions that facilitate the layout process is also associated with the base node class. These functions perform connectivity tasks such as transitive closure and returning the set of nodes within a given distance. In addition to these functions, each nodes has two registers that can be set and read by the layout algorithm. This allows the layout algorithm to associate data with individual nodes during the layout process.

Layout algorithms are also implemented as objects. The base class for layout objects contains functions and data structures to facilitate the layout process. For example, the routines for creating the enclosing convex hull for the generated layout are built into the base layout object class and don't have to be implemented when a new layout algorithm is added to the system.

As an example of how the built-in functions facilitate the addition of a new layout algorithm, consider the hierarchical layout algorithm used for many of the previous examples. The algorithm can be summarized by the following list. In this list, the "•" symbol indicates the task is performed by a built-in function. The "○" symbol indicates the task was implemented explicitly for this layout algorithm.

- • Mark all nodes unseen (done only once).
- • Mark self seen.
- • Request enclosing convex hull from each child object.
- • Position the children's hulls **vspacing** apart.
- ○ Position self over children's hulls.
- • Create and return a convex hull.

The tasks of marking nodes, marking self, getting the children of a node and creating a convex hull are all built into the system. The most useful tools are the convex hull creation and positioning tools. They allow the implementor to abstract away the details of positioning pieces of the layout, especially the problem of how close two pieces can be placed.

7.1 Efficient Use of Convex Hulls

The prototype system uses the Graham's Scan algorithm [21, 37] to calculate the smallest enclosing convex hulls of nodes and generated layouts. The complexity of this algorithm is O(n log n) where n is the total number of points considered by the algorithm.

After all the nodes and sub layouts are positioned by a layout algorithm, a new hull that encloses the generated layout is calculated. This hull is calculated by applying the scan algorithm to all the points in the convex hulls for all the nodes and sub layouts in the new generated layout.

Each node in the graph must calculate an enclosing convex hull at creation time and any time the node changes size or shape. There are basically three types of graphic representations for nodes: line drawings, text, and bitmaps. The bounding box of text is used to approximate the convex hull of the text. If the node consists of only text, the four corners of the bounding box can be used as the convex hull. The endpoints of all the lines

```
new tree top(justification = center)
{
    new text_node(size = large) : 1,2,3,4;
    1 to 2;
    1 to 3;
    2 to 4;
};
```

Table 7.1: Sample Graph Specification Script

used to draw the node are passed to the scan algorithm to calculate the enclosing convex hull of nodes that contain line drawings.

It is possible to apply the scan algorithm to all the points in a bitmap, but this can result in hulls with far more edges than are useful. Since node bitmaps tend to be small, a hull with eight sides usually provides a sufficient approximation to the smallest enclosing hull and is thus used by the system.

Convex hulls are stored as an ordered set of points. Since many of the algorithms that position convex hulls must first calculate several minimum and maximum points, eight such points are precalculated and stored with the convex hull. The size of the convex hull data structure is thus increased and the hull positioning algorithms become more efficient because the extremes are calculated only once.

7.2 Graph Specification Language

The prototype system uses a hierarchically structured language to specify graphs. This language allows the user to specify both the connectivity of the graph and the hierarchical structure of layout objects used to lay it out. The connectivity of the graph is merged with the structure to provide support for textually defining hierarchically composed layout algorithms. Appendix A gives the complete grammar for the language and Appendix B gives scripts that can be used to recreate many of the interactively generated examples graphs presented in this dissertation. The remainder of the section presents an overview of the graph specification language.

The language is organized as a set of declarations. Nodes and layout objects are placed inside of a layout object by placing their declaration inside of the parent object's declaration. For example, the script shown in Table 7.1 declares a new tree layout object named `top`. It contains four nodes named `1`, `2`, `3`, `4`, (nodes can have string or integer names) and three edges.

Parameters to all object declarations—nodes, edges and layout objects—are enclosed in parentheses following the objects type name or the optional instantiation name. Parameters are set by following the name of the parameter by an equal sign and then the value of the parameter. For example, in the script shown in Table 7.1, the tree layout object named `top` has the parameter `justification` set to the value `center`. Similarly,

```
new tree top(justification = center)
{
    nodes = text_node(size = large);
    edges = arrow_edge;

    1 to 2,3;
    2 bendedge(head = "icons/arrowhead") 4;
};
```

Table 7.2: Default Node and Edge Types

the size parameter for all the new nodes is set to `large`—since the parameter followed the type name, the parameter will be applied to all nodes declared in that statement.

Edges are declared by placing the keyword `to` after the name of the origin node and before the name of the destination node. For example, the statement `1 to 2;` creates an edge between the nodes named `1` and `2`. Multiple edges can be declared in a single statement by specifying multiple destination node names. For example, the first edge declaration in the script shown in Table 7.2, declares two separate edges, one between the nodes `1` and `2` and one between the nodes `1` and `3`.

When edges are created using the keyword `to` (as shown in the first example), the default edge type is used. This default type can be changed using the `edges =` command as shown in Table 7.2. All edges in this example created after the `edges = arrow_edge` command that are declared using the `to` keyword will be of type `arrow_edge`.

Edges can also be declared by using the edge type name instead of the keyword `to`. The final edge in the above example is of type `bendedge` and has its bitmap argument `head` set to the filename `icons/arrowhead`.

There is also a default node type. Notice that the nodes in the second example were not declared before edges were created between them. When a node that has not been explicitly declared appears in an edge declaration, the node is automatically declared using the default node type. Similarly to the edge type default, the node type default can be changed. In this example, the node type default has been set to `text_node`.

Nodes and edges belong to the layout object within which they are declared. In the first two script examples, all the nodes and edges belong to the single tree layout object. Table 7.3 shows a two level hierarchy of layout objects. The row layout object named `row_of_trees` contains two tree layout objects named `left` and `right`. Each of these contains several node and edge declarations.

```
new row row_of_trees(justification = center)
{
    new tree left
    {
        a to b;
        b to c,d;
        d to e;
    };
    new tree right
    {
        1 to 2;
        2 to 3,4;
        3 to 5,6,7;
    };
};
```

Table 7.3: Hierarchical Layout Object Declaration

CHAPTER 8
INTERACTIVE CONTROL OF USER INTERFACES

This chapter presents an application of interactive graph layout in which graph layout techniques are used for creating flexible user interfaces that allow the end-user to customize the structure of the interface. The basic idea behind this extension is to define node types that are general interaction objects, and use interactive graph layout to position them.

Traditional user interface management systems have concentrated on providing the interface designer with very general tools for specify the interface. These tools provide the interface designer with the ability to precisely specify exact interfaces. The cost for this power is that the specification must be very exact—each interaction object must be considered.

Using interactive graph layout algorithms to format an interface provides a flexible mechanism that empowers the end-user with the tools to customize the structure of an interface. This is an improvement of existing interface management systems because there is not always a single "best" structure for the interface.

In addition, graph layout algorithms can handle interfaces to applications with dynamic data sets. The result is that the end-user can create interfaces that more closely match her expectations. This reduces the gap between the user and the actual data, and increases the directness of the interface.

8.1 Data-Rich User Interfaces

The structure of an interface is more important for interfaces that concentrate on the manipulation and exploration of large data sets—*data-rich* interfaces—because the data begins to predominate the interface. Not only does the structure become more important, the large number of interaction objects are more difficult to lay out on the screen.

Unlike many interfaces, a data-rich interface may be used for the general exploration of data or for other poorly structured tasks that cannot be well characterized in advance. In addition, multiple aspects of the data may need to be addressed simultaneously and multiple relationships between individual data items may need to be depicted. Finally, data-rich applications normally cannot control the number and properties of most of the user interface components in advance—since these are derived from a dynamic data set.

8.2 Using Interactive Graph Layout to Generate Interfaces

As a sample data-rich interface, consider a spreadsheet. Each cell in a spreadsheet is an interaction object that allows the user to manipulate part of the data—usually a name, an equation and a value. Traditional spreadsheet interfaces are structured in a rectangular matrix—all the cells are positioned on the screen in a rectangular grid, as shown in the

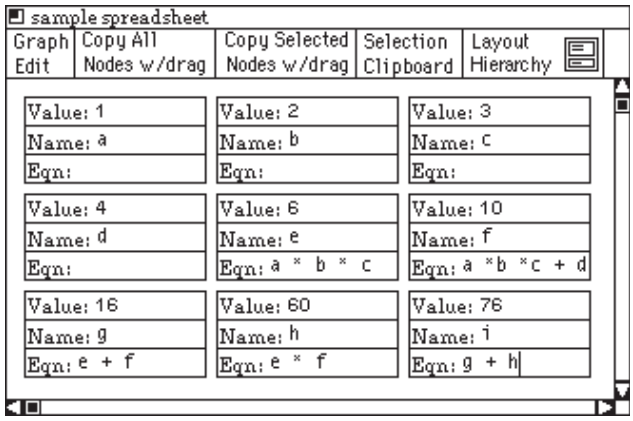

Figure 8.1: Traditional Spreadsheet Interface

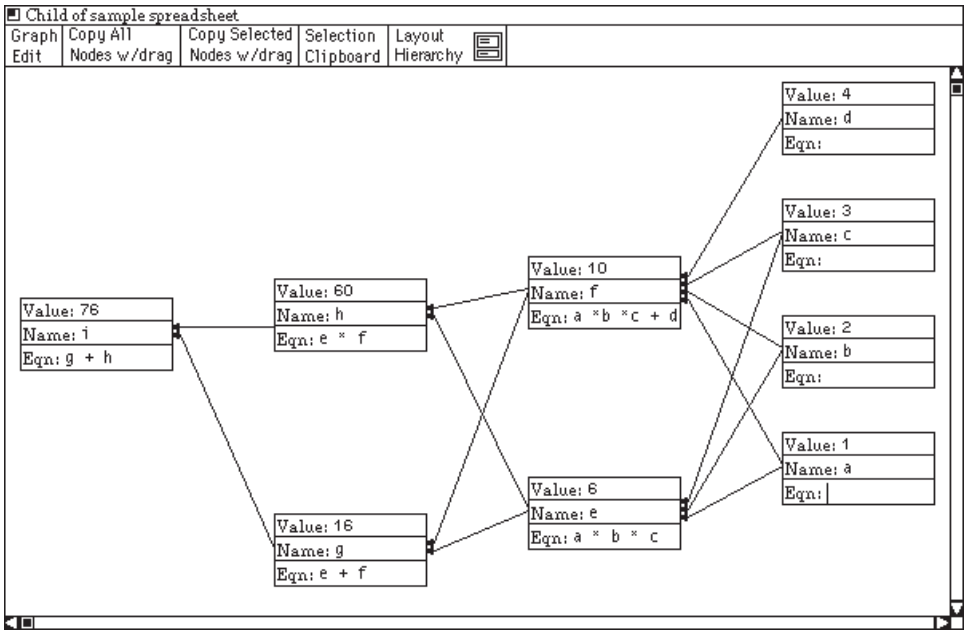

Figure 8.2: Spreadsheet Interface Depicting Cell Dependencies for Example in Figure 8.1

---

sample spreadsheet in Figure 8.1. The position of each cell is permanent and cannot be changed by the user.

Figure 8.2 shows an alternative interface to the sample spreadsheet shown in Figure 8.1. The cells in this interface where positioned by a hierarchical graph layout algorithm. The algorithm used the equation in each cell to determine the dependency ordering of all the cells in the spreadsheet. It then positioned the cells in a pattern that represents the dependency DAG. In addition to positioning the cells, the algorithm created dependency links between cells (the edges shown in Figure 8.2).

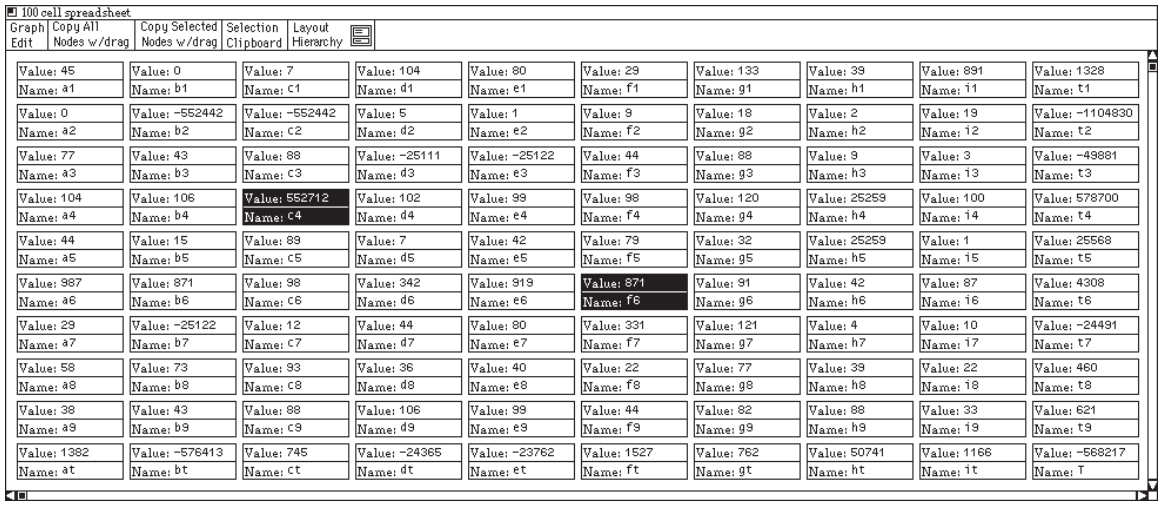

Figure 8.3: Spreadsheet with two Cells of Interest Highlighted

---

The interfaces in Figures 8.1 and 8.2 are live interfaces; they provide different views of the spreadsheet. Both are concurrently shown on the screen and the user can interact with either to change the spreadsheet data. This allows the user to look at both interfaces to understand the data and to interact with the interface that best fits his current task. In the example above, if the user wanted to edit all the cells that a given cell depends on, he might use the dependency oriented interface. On the other hand, if the user wanted to change all the cells grouped in a column, he would use the interface organized as a matrix.

The interactive graph layout methodology allows the user to create custom interfaces. For example, consider the interface shown in Figure 8.3 . Assume the user has discovered that when the value for cell "f6" is changed, the value in cell "c4" changes unexpectedly—these cells have been highlighted in Figure 8.3. The dependency graph for the spreadsheet would help the user examine the dependencies between the two cells of interest. However, the complete dependency graph is quite large and would not fit well on the screen. The solution is to allow the user to select the portion of the spreadsheet she is currently interested in.

Figure 8.4 is an interface that includes only a portion of the original interface. It includes all cells in the spreadsheet that the cell "c4" depends on. Even this graph is a bit large and contains some information that does pertain to the user's current interest. Figure 8.5 shows another interface that includes only the cells that depend on "f6" and that "c4" depends on—all the paths between these cells. Since there is more space available, cells have been expanded in Figures 8.4 and 8.5 to include the equations allowing the user to interpret the dependency graph. Several of the cells have longer equations have been enlarged to show the entire equation. The resulting interface, while no longer general purpose, is dramatically better for the particular task at hand (debugging the connection between cells "f6" and "c4").

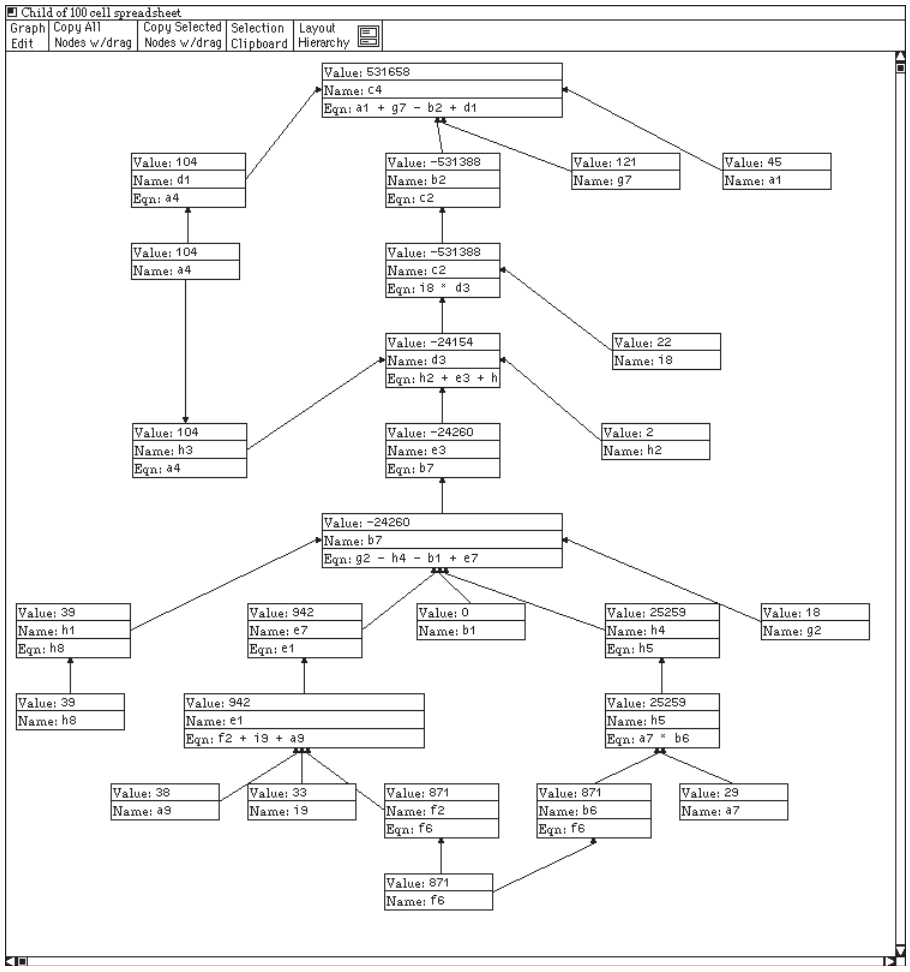

Figure 8.4: Dependency Graph for Cell "c4" in Figure 8.3

Additional views of the interface are created just like additional views of a graph. Empty views and empty layout objects are created via menu selection. Interaction objects are copied into the new view just like all other nodes.

8.3 Extending the Interactive Graph Layout Prototype

The example interfaces shown in this chapter were built on top of the prototype interactive graph layout system. No changes were made to the prototype to enable it to lay out interfaces However, the spreadsheet interface did required the specification of the cell node type. The cell node definition included functions for handling changes to the three text fields and included a general equation solving system for was accessed calculating and updating the value fields. In addition to the cell node definition, the reachable selection algorithm and the hierarchical layout algorithm were modified to use the cell equations

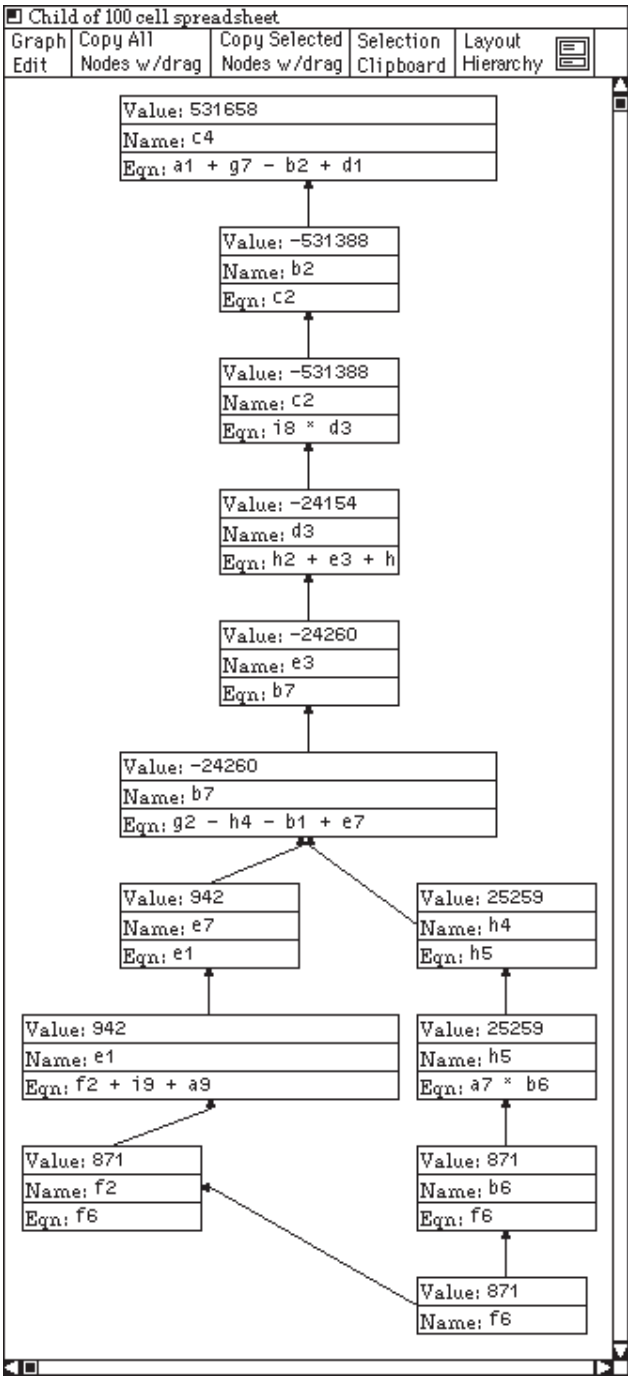

Figure 8.5: Pruned Dependency Graph

for calculating the dependencies between cells.

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

This dissertation has presented a novel approach to automatic graph layout. The basic idea is to provide the non-programming end-user with the power to interactively explore a graph and create many custom layouts.

9.1 Contributions

This work has redefined the traditional graph layout problem of trying to generate the best layout for a given graph to the problem of how to provide the end-user with the tools that allow him to explore graphs and learn about the encoded information. The result of this new approach is that non-programming end-users can create meaningful layouts of large graphs and thus learn about the data encoded within them.

Four new concepts were presented to provide the user with this power:

- Building new layout algorithms hierarchically
- Interactive parameterization of layout algorithms with sets of nodes and variables
- Algorithmic selection of subgraphs
- Using an interactive environment to iteratively explore large graphs.

9.1.1 Hierarchical Composition of Graph Layout Algorithms

Hierarchically composing layout algorithms allows the non-programming end-user to create new layout algorithms. This mechanism allows the user to create layout algorithms that are especially well suited for the graph at hand. It allows her to customize the layout so it fits her current focus.

This is particularly important with large graphs. A single layout cannot meet all needs of all users. The user must be able to carve out important subgraphs and create custom layouts for them. Since each graph is different, only the user can create an appropriate layout algorithm.

The algorithms created by composing existing algorithms are often graph-specific—they do a good job of laying out a particular graph. Even though these new algorithms may only work on a small set of graphs, they are easy to create. There is a tradeoff between the creation difficulty and the generality of layout algorithms. Layout algorithms that can generate layouts for a general set of graphs are very hard to create, but since they work on lots of graphs the time is justified. On the other hand, it is easy to create new graph specific algorithms by composing existing algorithms. Since these algorithms are so easy to create, it is reasonable that they work on only a small set of graphs.

Creating new layout algorithms allows the user to generate layouts that focus on details within a large graph. While general graph layout algorithms are good at depicting the

overall structure of the graph, they often obscure local structures and details. The power to create new layout algorithms allows the user to focus on the details of the graph—to learn about the graph in appropriately small pieces.

9.1.2 Parameterized Layout Algorithms

Often the user can perceive improvements to a generated layout. The problem with editing a generated layout is that it is very hard to incorporate the edits back into the layout process. If the changes are not incorporated then the next iteration of the layout algorithm will undo the changes.

The proposed solution is to provide a mechanism that allows the user to interact directly with the layout algorithms. The idea is to export parameters of the layout process so the user can manipulate them. An interactive interface to these parameters allows the user to make quick adjustments and thus affords them the ability to iteratively fine tune the layout. In the hierarchical composition framework, the user can interact with the parameters of any individual layout algorithm and thus fine tune a small portion of the generated layout, while at the same time make global changes by adjusting parameters to the outer layout algorithms.

Directly editing the generated layout has the disadvantage that it is difficult to specify global changes. On the other hand, the parameterization method has the advantage that the user can make both local and global changes.

Individual layout algorithms have their own set of parameters. The user is limited to changing only the parameters implemented for a given layout algorithms. While this method is limiting, it has the advantage of simplicity over a more comprehensive system of general edits.

A mechanism for general edits should require that all layout algorithms support all the available edits. If it didn't the user would not know to which portions of the generated layout he could apply which edit functions. Thus the implementation of new layout algorithms would be complicated by forcing the implementor to support all the edit functions. Since layout algorithm parameterization does not require all algorithms to provide the same set of parameters, the system provides support for the implementation of new layout algorithms, instead of complicating it.

9.1.3 Algorithmic Subgraph Selection

Algorithmic selection is a crucial component of the interactive graph layout methodology. It allow the user to break up large graphs into pieces that are both meaningful and manageable.

Previous selection techniques forced the user to manually specify the portions of the graph of interest to him. While these methods are adequate for small graphs, they do not scale.

Algorithmic selection provides the user with a tool that works equally well on small and large graphs. Thus the size of a graph is no longer an impassable barricade. Regardless of size, the user can apply selection algorithms to carve out useful subgraphs.

9.1.4 Interactive Graph Exploration

An important contribution of this methodology is that it allows end-users to explore large graphs and learn about the data encoded within them piece by piece. This work has unlocked graphical data that was previous lost in graphs too large for traditional graph layout methods to view.

A crucial component to the success of this work is the novel use of interactive techniques. This interactive approach allows the user to do more than read a generated layout, but to interact with the graph and thus explore it in real time. Interactive exploration is a considerably better learning method than viewing a single static layout; giving the user control allows her to freely examine the data and allows her to follow her intuition asking questions and finding answers. The method of interactive exploration is analogous to the basic human learning model in which people interact with object in the real world.

9.2 Future Work

There are three primary directions for future research: using domain specific graph semantics to guide the layout process and selection, creating a methodology that can be used to build an interactive system for non-programmers to specify base level selection and layout algorithms, and automatically partitioning graphs and inserting the partitions into a hierarchy of layout algorithms.

9.2.1 Using Graph Semantics to Control the Layout Process

General graph semantics have been used to help guide the layout process [38]. Domain specific semantics could provided even more information for the layout process. For example, domain specific graphs often have well know components that are described by the domain semantics. If this semantic information were incorporated into the layout process, the resulting generated layouts could preserve the semantic significance of the important components.

In addition to using domain specific semantics to guide the layout process, they could be used to guide selection algorithms. Semantically sensitive selection algorithms would provide the user with a more powerful mechanism for selecting pertinent portions of the graph. An algorithm that took semantics into account would be able to traverse the graph in manners that are well suited for the given graph.

9.2.2 Layout and Selection Algorithm Specification

The hierarchical composition of layout algorithms has shown to provide the user with more control over the generated layout than traditional layout algorithms. However the user is limited to using layout algorithms from a given set.

While implementing graph layout algorithms is not too difficult for experienced programmers, it is clearly an impossible task for non-programming end-users. However, many layout algorithms are simple enough that an end-user might be able to specify them given a proper framework. Such a mechanism would provide more freedom to the user if he could

create new base level layout algorithms—the building blocks for hierarchically composed layout algorithms.

While it would be hard to provide a mechanism that allows the user to create generalized layout algorithms, tools could be created that allow the user to specify simply layout algorithms that would layout a specific graph.

End-users could specify a large number of layout algorithms simply by drawing templates and then choosing a method for distributing nodes within the template. For example, a system that allowed the user to draw any arbitrary polygon and then allowed her to select how the nodes are to be distributed along the edges or within the boundary would allow the user to create very specific layout algorithms. This approach could be enhanced by providing the user with a set of convex hull positioning algorithms and allowing them to specify the relative positioning of templates.

9.2.3 Automatic Partitioning of Graphs

There are two shortcomings of the layout hierarchy creation method. First, there is no reuse mechanism for layout hierarchies. The user has to manually specify entire hierarchies. Second, the mechanism for partitioning graphs and inserting them into layout hierarchies is completely manual.

The first problem is not very difficult to overcome. Layout hierarchies can be encapsulated and named. A database with a graphical interface could be used to display existing hierarchies. The metagraph interface would have to be modified to allow the user to extract a layout hierarchy and to allow insertion and deletion of entire layout hierarchies.

It is less clear how to overcome the second problem. Each layout hierarchy requires a different partitioning. For example, consider the layout shown in Figure 3.8. The hierarchy that positions the nodes on the circle requires the eight nodes in the cycle to be divided into four sets: top, bottom, left, and right. Not only must the nodes be partitioned into four sets, the order is crucial. The first and second nodes of the cycle must be in to top row, the third and fourth must be in the second and so on. This partitioning is unique for this layout hierarchy.

The hierarchical layout algorithm composition method would clearly be more useful if graphs were automatically partitioned into layout hierarchies. However, since each hierarchy can be different, it is unclear how to perform this task automatically. It seems that each layout hierarchy will require a unique partitioning algorithm. Furthermore, since the user can specify layout hierarchies, the user must also be able to specify a partitioning algorithm for each layout she creates.

APPENDIX A
GRAPH SPECIFICATION LANGUAGE GRAMMAR

<graph_file> ::=
    <window_decl> <graph_or_default_list>

<window_decl> ::=
    WINDOW <params> SEMICOLON
    | ϵ

<graph_or_default_list> ::=
    <graph_or_default_list> <graph_decl> SEMICOLON
    | <graph_or_default_list> <defaults> SEMICOLON
    | ϵ

<graph_decl> ::=
    NEW GRAPH <params> LBRACKET <decl_list> RBRACKET

<decl_list> ::=
    <decl_list> NEW <new_layout> LBRACKET <decl_list> RBRACKET SEMICOLON
    | <decl_list> <node_decl> SEMICOLON
    | <decl_list> <edge_decl> SEMICOLON
    | <decl_list> <defaults> SEMICOLON
    | <decl_list> <selections> SEMICOLON
    | ϵ

<new_layout> ::=
    ID <params>
    | ID ID <params>
    | ID STRING <params>

<node_decl> ::=
    ID <params> COLON <node_decl_list>

<node_decl_list> ::=
    <node_decl_list> COMMA <node>
    | <node>

```
<node> ::=
      STRING <params>
      |  ID <params>
      |  ID DOTS ID <params>

<params> ::=
      LPAREN <arg_list> RPAREN
      |  LPAREN RPAREN
      |  ϵ

<arg_list> ::=
      <arg_list> COMMA <arg>
      |  <arg>

<arg> ::=
      STRING EQUAL STRING
      |  STRING EQUAL ID
      |  ID EQUAL STRING
      |  ID EQUAL ID

<edge_decl> ::=
      <nodes> ID <params> <nodes>
      |  <nodes> TO <nodes>

<nodes> ::=
      LPAREN <node_name_list> RPAREN
      |  <node_name_list>

<node_name_list> ::=
      <node_name_list> COMMA <node>
      |  <node>

<defaults> ::=
      EDGES EQUAL ID <params>
      |  NODES EQUAL ID <params>

<selections> ::=
      SELECT <nodes>
```

APPENDIX B
SPECIFICATION SCRIPTS FOR EXAMPLE GRAPHS

The examples presented in this dissertation were created by interactively manipulating layouts. In order to replicate the examples and to provide an additional representation of the layout structure, scripts were created to describe the interactively created layouts. (The current prototype does not contain a dump mechanism for generating scripts from interactively created layouts). This appendix lists the scripts for many of the examples—the scripts for the larger examples were excluded because of their size.

Script for layout shown in Figure 3.1:

```
new graph(x = 0, y = 0, w = 290, h = 150,
  title = "Simple Hierarchical Layout: fig/simplehier"){

    nodes = text_node(size = small);
    new row(justification = centered, spacing = 30) {
        new tree(panel = t, edge_direction_ratio = vertical) {
            1 to 2,3;
            2 to 4,5;
           4 to 6-9;
            5 to 10-13;
        };
        new grid(vspacing = 20, hspacing = 20) {
            dnode(size = xsmall) : 100-115;
            edges = dedge(head = "icons/arrowheads/small_arrow");
            100 to 101;
            101 to 102;
            102 to 103;
            100 to 104;
            104 to 108;
            108 to 112;
            103 to 107;
            107 to 111;
            111 to 115;
            112 to 113;
            113 to 114;
            114 to 115;
        };
    };
};
```

Script for layout shown in Figure 3.3:

```
new graph(x = 0, y = 0, w = 290, h = 310,
  title = "Simple 2-Level View"){
    nodes = text_node(size = med);
    new col(justification = center, spacing = 25) {
        new tree(edge_direction_ratio = vertical,
          vspacing = 25, hspacing = 15){
            1 to 2,3,4,5,6,7;
            2 to 12-15;
            5 to 16-18;
        };
        new tree(edge_direction_ratio = vertical,
          direction = bottom_to_top, vspacing = 25,
          hspacing = 30) {
            a to b,c,d;
            b to e,f,g;
            d to h,i,j;
        };
        12 to e;
        14 to f;
        16 to g;
        17 to i;
    };
};
```

Script for layout shown in Figure 3.4:

```
new graph(x = 5, y = 5, w = 314, h = 240,
  title = "Strongly Connected 8 Node Graph") {
    nodes = text_node(size = medium);
    new col circut(justification  = center, spacing = 40) {
        new row top(justification = bottom, spacing = 20,
          edge_direction_ratio = vertical) {
            one to two;
        };
        new row sides(spacing = 130) {
            new col right(justification = right, spacing = 20,
              edge_direction_ratio = horizontal) {
                eight to seven;
            };
            new col left(justification = left, spacing = 20,
              edge_direction_ratio = horizontal) {
                three to four;
            };
        };
        new row bottom(justification = top, spacing = 20,
          edge_direction_ratio = vertical) {
            six to five;
        };
    };
    one to three, four, five, six, seven, eight;
    two to four, five, six, seven, eight;
    three to five, six, seven, eight;
    four to six, seven, eight;
    five to seven, eight;
    six to eight;
};
```

Script for layout shown in Figure 3.6:

```
new graph(x = 5, y = 5, w = 314, h = 280,
  title = "Strongly Connected 16 Node Graph") {
    nodes = dnode(size = medium);
    new col circut(justification  = center, spacing = 20) {
        new col top(justification = center) {
            new row topmiddle(justification = bottom) {
                dnode(size = medium) : bb, cc;
            };
            new row topoutside(spacing = 75,
              justification = bottom) {
                dnode(size = medium) : aa, dd;
            };
        };
        new row sides(spacing = 130) {
            new row right(justification = center) {
                new col rightinside(justification = right,
                  edge_direction_ratio = vertical) {
                    dnode(size = medium) : oo, nn;
                };
                new col rightoutside(spacing = 75,
                  justification = right,
                  edge_direction_ratio = horizontal) {
                    dnode(size = medium) : pp, mm;
                };
            };
            new row left(justification = center,
              edge_direction_ratio = horizontal) {
                new col leftoutside(spacing = 75) {
                    dnode(size = medium) : ee, hh;
                };
                new col leftinside(edge_direction_ratio =
                  vertical ){
                    dnode(size = medium) : ff, gg;
                };
            };
        };
        new col bottom(justification = center) {
            new row bottomoutside(spacing = 75) {
                dnode(size = medium) : ll, ii;
            };
            new row bottommiddle {
                dnode(size = medium) : kk,jj;
            };
        };
    };
    aa to bb,cc,dd,ee,ff,gg,hh,ii,jj,kk,ll,mm,nn,oo,pp;
    bb to cc,dd,ee,ff,gg,hh,ii,jj,kk,ll,mm,nn,oo,pp;
    cc to dd,ee,ff,gg,hh,ii,jj,kk,ll,mm,nn,oo,pp;
    dd to ee,ff,gg,hh,ii,jj,kk,ll,mm,nn,oo,pp;
    ee to ff,gg,hh,ii,jj,kk,ll,mm,nn,oo,pp;
    ff to gg,hh,ii,jj,kk,ll,mm,nn,oo,pp;
    gg to hh,ii,jj,kk,ll,mm,nn,oo,pp;
    hh to ii,jj,kk,ll,mm,nn,oo,pp;
```

```
    ii to jj,kk,ll,mm,nn,oo,pp;
    jj to kk,ll,mm,nn,oo,pp;
    kk to ll,mm,nn,oo,pp;
    ll to mm,nn,oo,pp;
    mm to nn,oo,pp;
    nn to oo,pp;
    oo to pp;
};
```

Script for layout shown in Figure 3.7:

```
new graph(x = 5, y = 5, w = 314, h = 340,
  title = "Tree with two cycles") {
    nodes = text_node;
    edges = bendedge(head = "icons/arrowheads/small_arrow");
    new sugiylo tree(search = depth_first, vspacing = 15,
      edge_direction_ratio = vertical){
        aa to bb;
        bb to cc;
        cc to dd;
        dd to ee;
        ee to 11;
        ee to ff;
        ff to 10;
        ff to gg;
        gg to hh;
        hh to aa;
        AA to BB;
        BB to CC;
        CC to DD;
        DD to EE;
        EE to 24;
        EE to FF;
        FF to 22,23;
        FF to GG;
        GG to HH;
        HH to AA;
        1 to 2,3,4;
        2 to 5,6;
        3 to aa;
        10 to 13,14;
        11 to 15;
        6 to 7,8,9;
        8 to 16;
        4 to 17,18,19,AA;
        18 to 20,21;
        22 to 25;
        23 to 26;
        24 to 27;
        27 to 28;
        28 to 29,30,31;
        select 1;
        };
};
```

Script for layout shown in Figure 3.8:

```
new graph(x = 5, y = 5, w = 314, h = 340,
  title = "Tree with two imbeded subgraphs") {
    nodes = text_node;
    edges = bendedge(head = "icons/arrowheads/small_arrow");
    new tree(edge_direction_ratio = vertical, vspacing = 15,
      search = breadth_first) {
        new col circut(justification  = center) {
            new row top {
                text_node : aa,bb;
            };
            new row sides(spacing = 50) {
                new col right(edge_direction_ratio = vertical) {
                    text_node : hh,gg;
                };
                new col left(edge_direction_ratio = vertical) {
                    text_node : cc,dd;
                };
            };
            new row bottom(edge_direction_ratio = horizontal) {
                text_node : ff,ee;
            };
            aa to bb;
            bb to cc;
            cc to dd;
            dd to ee;
            ee to ff;
            ff to gg;
            gg to hh;
            hh to aa;
        };
        new col circut2(justification  = center,
          bitmap = "icons/nodes/layouts/cycle") {
            new row top2 {
                text_node : AA,BB;
            };
            new row sides2(spacing = 50) {
                new col right2(edge_direction_ratio = vertical) {
                    text_node : HH,GG;
                };
                new col left2(edge_direction_ratio = vertical) {
                    text_node : CC,DD;
                };
            };
            new row bottom2 {
                text_node : FF,EE;
            };
            AA to BB;
            BB to CC;
            CC to DD;
            DD to EE;
            EE to FF;
            FF to GG;
            GG to HH;
```

```
            HH to AA;
        };
        1 to 2,3,4;
        2 to 5,6;
        3 hidden_edge circut;
        3 to aa;
        circut hidden_edge 11,10;
        ff to 11;
        ee to 10;
        10 to 13,14;
        11 to 15;
        6 to 7,8,9;
        8 to 16;
        4 to 17,18;
        4 hidden_edge circut2;
        4 to AA,19;
        18 to 20,21;
        circut2 hidden_edge 22,23,24;
        FF to 22,23;
        EE to 24;
        22 to 25;
        23 to 26;
        24 to 27;
        27 to 28;
        28 to 29,30,31;
        select 1;
    };
};
```

Script for layout shown in Figure 4.1:

```
new graph(x = 0, y = 0, w = 323, h = 299,
  title = "Depth first hierarchy") {
    nodes = text_node(size = medium);
    edges = dedge(head = "icons/arrowheads/small_arrow");
    new tree(search = depth_first,
      edge_direction_ratio = vertical, vspacing = 22) {
        1 to 2,3,4;
        2 to 5,6;
        3 to 8;
        6 to 7,8;
        8 to 9,10;
        4 to 10,13;
        10 to 11,12,13;
    };
};
```

Script for layout shown in Figure 4.2:

```
new graph(x = 0, y = 0, w = 323, h = 299,
  title = "Breadth first hierarchy") {
    nodes = text_node(size = medium);
    edges = dedge(head = "icons/arrowheads/small_arrow");
    new tree(search = breadth_first,
      edge_direction_ratio = vertical, vspacing = 22) {
        1 to 2,3,4;
        2 to 5,6;
        3 to 8;
        6 to 7,8;
        8 to 9,10;
        4 to 10,13;
        10 to 11,12,13;
    };
};
```

Script for layout shown in Figures 4.3, and 4.4:

```
new graph(x = 0, y = 0, w = 314, h = 340,
  title = "Tree with one root") {
    nodes = text_node;
    edges = bendedge(head = "icons/arrowheads/small_arrow");
    new sugiyama "one to ten"(panel = tree1,
      search = depth_first, vspacing = 30,
      hspacing = 7,edge_direction_ratio = vertical) {
        text_node : 100,6;
        1 to 2,3,4,5,6;
        2 to 101;
        6 to 100;
        100 to 102,103,104;
        101 to 300,301,302;
        101 to 200;
        200 to 201,202,206,203,207,204;
        203 to 206,207;
        204 to 205;
        205 to 210;
        206 to 208;
        207 to 209;
        208 to 305;
        209 to 306;
        210 to 209;
        201 to 302;
        300 to 303;
        301 to 304,305;
        305 to 306;
        select 1;
    };
};
```

Script for layout shown in Figure 5.1:

```
new graph(x = 0, y = 0, w = 290, h = 290,
  title = "Highlighted leaf nodes"){
    nodes = text_node(size = med);
    new tree(edge_direction_ratio = vertical, vspacing = 30) {
        text_node(size = med) : d,h,o,j,f,p,l,m;
        a to b,c;
        b to d,e;
        c to f,g;
        e to h,i,j;
        g to l,m,n;
        i to o;
        m to p;
    };
};
```

Script for layout shown in Figure 5.2:

```
new graph(x = 0, y = 0, w = 370, h = 290,
  title = "Leaf nodes in row"){
    nodes = text_node(size = med);
    new col(justification = center, spacing = 30){
        new tree(edge_direction_ratio = vertical,
          vspacing = 20, hspacing= 90){
            a to b,c;
            b to e;
            e to i;
            c to g;
            g to m;
        };
        new row(justification = center, spacing = 35,
          edge_direction_ratio = vertical){
            text_node(size = med) : d,h,o,j,f,l,p,m;
            b to d;
            e to h,j;
            i to o;
            c to f;
            g to l,n;
            m to p;
        };
    };
};
```

Script for layout shown in Figures 6.3 and 6.4:

```
new graph(x = 5, y = 5, w = 314, h = 440,
  title = "Strongly Connected 8 Node Graph") {
    nodes = text_node(size = medium);
    new col cycle(justification  = center, spacing = 40) {
        new row top_row(justification = bottom, spacing = 20,
          edge_direction_ratio = vertical) {
            one to two;
        };
        new row sides(spacing = 130) {
            new col right(justification = right, spacing = 20,
              edge_direction_ratio = horizontal) {
                seven to eight;
            };
            new col left(justification = left, spacing = 20,
              edge_direction_ratio = horizontal) {
                three to four;
            };
        };
        new row bottom(justification = top, spacing = 20,
          edge_direction_ratio = vertical) {
            five to six;
        };
    };
    one to three, four, five, six, seven, eight;
    two to four, five, six, seven, eight;
    three to five, six, seven, eight;
    four to six, seven, eight;
    five to seven, eight;
    six to eight;
};
```

Script for layout shown in Figures 8.1 and 8.2:

```
new graph(x = 0, y = 0, w = 328, h = 215,
  title = "sample spreadsheet") {
    new grid(vspacing = 6, hspacing = 6)  {
        cell : a(size = med, value = 1);
        cell : b(size = med, value = 2);
        cell : c(size = med, value = 3);
        cell : d(size = med, value = 4);
        cell : e(size = med, eqn = "a * b * c");
        cell : f(size = med, eqn = "a *b *c + d");
        cell : g(size = med, eqn = "e + f");
        cell : h(size = med, eqn = "e * f");
        cell : i(size = med, eqn = "g + h");
    };
};
```

Script for layout shown in Figures 8.3, 8.4 and 8.5:

```
new graph(x = 0, y = 0, w = 980, h = 420,
  title = "100 cell spreadsheet") {
    new grid(vspacing = 6, hspacing = 6)  {
        cell : a1(size = small, value = 45);
        cell : b1(size = small, value = 0);
        cell : c1(size = small, value = 7);
        cell : d1(size = small, eqn = "a4");
        cell : e1(size = small, eqn = "f2 + i9 + a9");
        cell : f1(size = small, value = 29);
        cell : g1(size = small, value = 133);
        cell : h1(size = small, eqn = "h8");
        cell : i1(size = small, value = 891);
        cell : t1(size = small,
          eqn = "a1 + b1 + c1 + d1 + e1 + f1 + g1 + h1 + i1");
        cell : a2(size = small, value = 0);
        cell : b2(size = small, eqn = "c2");
        cell : c2(size = small, eqn = "i8 * d3");
        cell : d2(size = small, value = 5);
        cell : e2(size = small, value = 1);
        cell : f2(size = small, value = 9);
        cell : g2(size = small, value = 18);
        cell : h2(size = small, value = 2);
        cell : i2(size = small, value = 19);
        cell : t2(size = small,
          eqn = "a2 + b2 + c2 + d2 + e2 + f2 + g2 + h2 + i2");
        cell : a3(size = small, value = 77);
        cell : b3(size = small, value = 43);
        cell : c3(size = small, value = 88);
        cell : d3(size = small, eqn = "h2 + e3 + h3");
        cell : e3(size = small, eqn = "b7");
        cell : f3(size = small, value = 44);
        cell : g3(size = small, value = 88);
        cell : h3(size = small, value = 9);
        cell : i3(size = small, value = 3);
        cell : t3(size = small,
          eqn = "a3 + b3 + c3 + d3 + e3 + f3 + g3 + h3 + i3");
        cell : a4(size = small, value = 104);
        cell : b4(size = small, value = 106);
        cell : c4(size = small, eqn = "a1 + g7 - b2 + d1");
        cell : d4(size = small, value = 102);
        cell : e4(size = small, value = 99);
        cell : f4(size = small, value = 98);
        cell : g4(size = small, value = 120);
        cell : h4(size = small, eqn = "h5");
        cell : i4(size = small, value = 100);
        cell : t4(size = small,
          eqn = "a4 + b4 + c4 + d4 + e4 + f4 + g4 + h4 + i4");
        cell : a5(size = small, value = 44);
        cell : b5(size = small, value = 15);
        cell : c5(size = small, value = 89);
        cell : d5(size = small, value = 7);
        cell : e5(size = small, value = 42);
        cell : f5(size = small, value = 79);
```

```
cell : g5(size = small, value = 32);
cell : h5(size = small, eqn = "a7 * b6");
cell : i5(size = small, value = 1);
cell : t5(size = small,
  eqn = "a5 + b5 + c5 + d5 + e5 + f5 + g5 + h5 + i5");
cell : a6(size = small, value = 987);
cell : b6(size = small, eqn = "f6");
cell : c6(size = small, value = 98);
cell : d6(size = small, value = 342);
cell : e6(size = small, value = 919);
cell : f6(size = small, value = 871);
cell : g6(size = small, value = 91);
cell : h6(size = small, value = 42);
cell : i6(size = small, value = 87);
cell : t6(size = small,
  eqn = "a6 + b6 + c6 + d6 + e6 + f6 + g6 + h6 + i6");
cell : a7(size = small, value = 29);
cell : b7(size = small, eqn = "g2 - h4 - b1 + e7 + h1");
cell : c7(size = small, value = 12);
cell : d7(size = small, value = 44);
cell : e7(size = small, eqn = "e1");
cell : f7(size = small, value = 331);
cell : g7(size = small, value = 121);
cell : h7(size = small, value = 4);
cell : i7(size = small, value = 10);
cell : t7(size = small,
  eqn = "a7 + b7 + c7 + d7 + e7 + f7 + g7 + h7 + i7");
cell : a8(size = small, value = 58);
cell : b8(size = small, value = 73);
cell : c8(size = small, value = 93);
cell : d8(size = small, value = 36);
cell : e8(size = small, value = 40);
cell : f8(size = small, value = 22);
cell : g8(size = small, value = 77);
cell : h8(size = small, value = 39);
cell : i8(size = small, value = 22);
cell : t8(size = small,
  eqn = "a8 + b8 + c8 + d8 + e8 + f8 + g8 + h8 + i8");
cell : a9(size = small, value = 38);
cell : b9(size = small, value = 43);
cell : c9(size = small, value = 88);
cell : d9(size = small, value = 106);
cell : e9(size = small, value = 99);
cell : f9(size = small, value = 44);
cell : g9(size = small, value = 82);
cell : h9(size = small, value = 88);
cell : i9(size = small, value = 33);
cell : t9(size = small,
  eqn = "a9 + b9 + c9 + d9 + e9 + f9 + g9 + h9 + i9");
cell : at(size = small, value = 1 ,
  eqn = "a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9");
cell : bt(size = small, value = 1 ,
  eqn = "b1 + b2 + b3 + b4 + b5 + b6 + b7 + b8 + b9");
cell : ct(size = small, value = 1 ,
  eqn = "c1 + c2 + c3 + c4 + c5 + c6 + c7 + c8 + c9");
```

```
        cell : dt(size = small, value = 1 ,
          eqn = "d1 + d2 + d3 + d4 + d5 + d6 + d7 + d8 + d9");
        cell : et(size = small, value = 1 ,
          eqn = "e1 + e2 + e3 + e4 + e5 + e6 + e7 + e8 + e9");
        cell : ft(size = small, value = 1 ,
          eqn = "f1 + f2 + f3 + f4 + f5 + f6 + f7 + f8 + f9");
        cell : gt(size = small, value = 1 ,
          eqn = "g1 + g2 + g3 + g4 + g5 + g6 + g7 + g8 + g9");
        cell : ht(size = small, value = 1 ,
          eqn = "h1 + h2 + h3 + h4 + h5 + h6 + h7 + h8 + h9");
        cell : it(size = small, value = 1 ,
          eqn = "i1 + i2 + i3 + i4 + i5 + i6 + i7 + i8 + i9");
        cell : T(size = small, value = 1 ,
          eqn = "t1 + t2 + t3 + t4 + t5 + t6 + t7 + t8 + t9");
    };
};
```

REFERENCES

[1] Apple Computer Company, *Inside Macintosh*, Addison-Wesley Publishing Company, Inc., 1982.

[2] Aoudja, F, Laborie, M, Saint-Paul, A., CASE: Automatic Generation of Electrical Diagrams, *Computer-Aided Design*, Vol. 18, No. 7, Pages 356-360, 1986.

[3] Asente, P. J., Editing Graphical Objects Using Procedural Representations, *Digital Equipment Corporation Western Research Laboratory Research Report*, 87/6, November 1987.

[4] Binder, R. V., Graphic Notation for Relational Design, *Database Programming and Design*, Pages 50-59, April 1990.

[5] Bohringer, K., Paulisch, F. N., Using Constraints to Achieve Stability in Automatic Graph Layout Algorithms, *Proceedings of ACM/SIGCHI*, Pages 43-51, April 1990.

[6] Brooks, K. P., A, Two-view Document Editor with User-definable Document Structure, *Ph. D. Dissertation, Department of Computer Science, Stanford University*, May 1988.

[7] Bryce, D., Hull, R., SNAP: A Graphics-based Schema Manager, *Proceedings of the IEEE Conference on Data Engineering*, Pages 151-164, February 1986.

[8] Cahn, R., An Algorithm to Draw Networks and Graphs, *Technical Report, IBM T.J. Watson Research Center*, RC 14126 (#63287), October 1988.

[9] Card, S. K., Robertson, G. G., Mackinlay, J. D., The Information Visualizer, an Information Workspace, *Proceedings of ACM/SIGCHI*, Pages 181-188, April 1991.

[10] Cardelli, L., Building User Interfaces by Direct Manipulation, *Proceedings of the ACM/SIGGRAPH Symposium on User Interface Software and Technology*, Pages 152-166, October 1988.

[11] Chen, P., The Entity Relationship Model—Towards a Unified View of Data, *ACM Transactions on Database Systems*, Vol. 1, No. 1, March 1976.

[12] Davidson, R., Harel, D., Drawing Graphs Nicely Using Simulated Annealing, *Technical Report, The Weizmann Institute of Science, Department of Applied mathematics and Computer Science*, CS98-13, July 1989.

[13] Dertin, J., *Semiology of Graphics*, The University of Wisconsin Press, 1983.

[14] Di Battista, G., Pietrosanti, E., Tamassia, R., Tollis, I., Automatic Layout of PERT Diagrams with XPERT, *Proceedings of IEEE Workshop on Visual Languages*, Pages 171-176, October 1989.

[15] Eades, P., A Heuristic for Graph Drawing, *Congressus Numerantium*, Vol. 42, Pages 149-160, 1984.

[16] Eades, P., Tamassia, R., Algorithms For Drawing Graphs: An Annotated Bibliography, *Technical Report, Brown University, Department of Computer Science*, October 1989.

[17] Fruchterman, T., Reingold, E., Graph Drawing by Force-Directed Placement, *Technical Report, University of Illinois at Urbana-Champaign, Department of Computer Science*, UIUCDCS-R-90-1609, June 1990.

[18] Gansner, E. R., North, S. C., Vo, K. P., DAG—A Program that Draws Directed Graphs, *Software–Practice and Experience*, Vol. 18, No. 1, Pages 1047-1062, November 1988.

[19] Garey, M. R., Johnson, D.S., Crossing Number is NP-complete, *SIAM Journal of Algebraic and Discrete methods*, Vol. 4, No. 3, Pages 312-316, 1983.

[20] Goldbert, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[21] Graham, R. L., An Efficient Algorithm for Determining the Convex Hull of a Planar Set, *Information Processing Letters*, Vol. 1, Pages 132-133, 1972.

[22] Halasz, F. G., Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems, *Communications of the ACM*, Vol. 31, No. 7, Pages 836-852, July 1988.

[23] Heath, L., S., Rosenberg, A., L., Graph Layout Using Queues, *Technical Report, University of Massachusetts, Department of Computer and Information Science*, TR 98-45, December 1989.

[24] Henderson, D. A. Jr., Card, S. K., Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface, *ACM Transactions of Graphics*, Vol. 5, No. 3, July 1986.

[25] Henry, T. R., Hudson, S. E., Newell G. L., Integrating Gesture and Snapping into a User Interface Toolkit, *Proceedings of the ACM/SIGGRAPH Symposium on User Interface Software and Technology*, Pages 112-121, October 1990.

[26] Himsolt, M., Graphed: An Interactive Graph Editor, *Technical Report, Universitat Passau, Lehrstuhl fur Informatik*, 1988.

[27] Hudson, S., Peterson, L., Schatz, B., Systems Technology for Building a National Collaboratory, *Technical Report, University of Arizona, Department of Computer Science*, TR 90-24, September 1990.

[28] Hutchins, E. L., Holland, J. D., Norman, D. A., Direct Manipulation Interfaces, in *User Centered Systems Design*, Norman, D. A., Draper, S. W. (eds.), Lawrence Erlbaum Associates, Hillsdale, New Jersey, Pages 87-124, 1986.

[29] Jablonowski, D., Guarna, V. Jr., GMB: A Tool for Manipulating and Animating Graph Data Structures, *Software Practice and Experience*, Vol. 19, No. 3, Pages 283-301, March 1989.

[30] Johnson, D. S., The NP-Completeness Column: an Ongoing Guide, *Journal of Algorithms*, Vol. 3, No. 1, Pages 89-99, 1982.

[31] Kamada, T., Kawai, S., A General Framework for Visualizing Abstract Objects and Relations, *ACM Transactions of Graphics*, Vol. 10, No. 1, Pages 1-39, January 1991.

[32] Kosak, C., Marks, J., Shieber, S., A Parallel Genetic Algorithm for Network-Diagram Layout, *Proceedings of the Fourth International Conference on Genetic Algorithms*, Pages 458-465, July 1991.

[33] Kaufman, L., *Sight and Mind: An Introduction to Visual Perception*, Oxford University Press, New York, 1974.

[34] Linton, M. A., Vlissides, J. M., Calder, P. R., Composing User Interfaces with InterViews, *IEEE Computer*, Vol. 22, No. 2, Pages 8-22, February 1989.

[35] Mackinlay, J.D., Automating the Design of Graphical Presentations of Relational Information, *ACM Transactions of Graphics*, Vol. 5, No. 2, Pages 499-512, September 1985.

[36] Mackinlay, J. D., Robertson, G. G., Card, S. K., The Perspective Wall: Detail and Context Smoothly Integrated, *Proceedings of ACM/SIGCHI*, Pages 173-179, April 1991.

[37] Manber, U., *Introduction to Algorithms a Creative Approach*, Addison Wesley, Reading, Massachusetts, 1989.

[38] Marks, J., A Syntax and Semantics for Network Diagrams, *Proceedings of IEEE Workshop on Visual Languages*, Pages 104-110, October 1990.

[39] Messinger, E., Automatic Layout of Large Directed Graphs, *Technical Report, University of Washington, Department of Computer Science*, TR88-07-08, July 1988.

[40] Messinger, E. B., Rowe, L. A., Henry, R. R., A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs, *IEEE Transactions of Systems, Man, and Cybernetics*, Vol. SMC-21, No. 1, Pages 1-12, January 1991.

[41] Moen, S., Drawing Dynamic Trees, *IEEE Computer*, Pages 21-28, July 1990.

[42] Myers, B. A., Creating User Interfaces by Demonstration, *Ph. D. Dissertation, University of Toronto*, CSRI-196, May 1987.

[43] Myers, B. A., *et. al.*, The Garnet Toolkit Reference Manuals: Support for Highly–Interactive, Graphical User Interfaces in Lisp, *Technical Report, Carnegie Mellon University, School of Computer Science*, CMU-CS-98-196, November 1989.

[44] Newbery, F., An Interface Description Language for Graph Editors, *Proceedings of IEEE Workshop on Visual Languages*, Pages 144-149, October 1988.

[45] Olsen, D. Jr., MIKE: The Menu Interaction Kontrol Environment, *ACM Transactions on Graphics*, Vol. 17, No. 3, Pages 43-50, 1986.

[46] Pazel, D., A Graphical Interface for Evaluating a Genetic Algorithm for Graph Layout, *Technical Report, IBM T.J. Watson Research Center*, RC 14348 (#64209), January 1989.

[47] Paulisch, F., Tichy, W., EDGE: An Extendible Graph Editor, *Software Practice and Experience*, Vol. 20, No. S1, Pages S1/63-S1/88, June 1990.

[48] Robertson, G. G., Mackinlay, J. D., Card, S. K., Cone Trees: Animated 3D Visualizations of Hierarchical Information *Proceedings of ACM/SIGCHI*, Pages 189-194, April 1991.

[49] Robins, G., The ISI Grapher: A Portable Tool for Displaying Graph Pictorially, *Proceedings of Symboliikka '87*, Helsinki, Finland, August 1987.

[50] Roth, S. F., Mattis, J., Data Characterization for Intelligent Graphics Presentation, *Proceedings of ACM/SIGCHI*, Pages 193-200, April 1990.

[51] Rowe, L. A., Davis, M. Messinger, E. Meyer, C., A Browser for Directed Graphs, *Software Practice and Experience*, Vol. 17. No. 1, Pages 61-76, January 1987.

[52] Shneiderman, B., The Future of Interaction Systems and the Emergence of Direct Manipulation, *Behaviour and Information Technology*, Vol. 1, Pages 57-69, 1982.

[53] Shu, N. C., *Visual Programming*, Van Nostrand Reinhold, New York, New York, 1988.

[54] Singh, G., Green, M., Chisel: A system for Creating Highly Interactive Screen Layouts, *Proceedings of the ACM/SIGGRAPH Symposium on User Interface Software and Technology*, Pages 86-94, November 1989.

[55] Storer, J., A., On Minimal Node-Cost Planar Embeddings, *Networks*, Vol. 14, Pages 181-212, 1984.

[56] Stotts, P., Expressing High-Level Visual Concurrency Structures in the PFG Kernel Language, *Proceeding of the IEEE Workshop on Visual Languages*, Pages 168-174, October 1988.

[57] Stotts, P., Furuta, R., Petri-Net-Based Hypertext: Document Structure with Browsing Semantics, *ACM Transactions of Information Systems*, Vol. 7, No. 1, Pages 3-29, January 1989.

[58] Stotts, P. D., Furuta, R., Browsing Parallel Process Networks, *Journal of Parallel and Distributed Computing*, Vol. 9, Pages 224-235, 1990.

[59] Stroustrup, B., *The C++ Programming Language*, Addison Wesley, Reading, Massachusetts, 1987.

[60] Sugiyama, K., Tagawa, S., Mitsuhiko, T., Methods for Visual Understanding of Hierarchical System Structures, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-11, No. 2, February 1980.

[61] Szekely, P., Template–Based Mapping of Application Data to Interactive Displays, *Proceedings of the ACM/SIGGRAPH Symposium on User Interface Software and Technology*, Pages 1-9, October 1990.

[62] Tamassia, R., On Embedding a Graph in the Grid with the Minimum Number of Bends, *SIAM Journal of Computing*, Vol. 16, No. 3., Pages 421-444.

[63] Tamassia, R., Di Battista, B., Batini, C., Automatic Graph Drawing and Readability of Diagrams, *IEEE Transactions of Systems, Man, and Cybernetics*, Vol. 18, No. 1, Pages 61-79, January/February 1988.

[64] Ullman, J., *Principles of Database Systems*, Computer Science Press, 1982.

[65] van Laarhoven, P. J. M., Aarts, E. H. L., *Simulated Annealing: Theory and Applications*, D. Reidel Publishing Company, 1987.

[66] Vlissides, J. and Linton, M., Unidraw: A Framework for Building Domain-Specific Graphical Editors, *ACM TOIS*, Vol. 8, No. 3, Pages 237-268, July 1990.

[67] Wilson, R. J., *Introduction to Graph Theory*, Third Edition, Longman Inc., New York, New York, 1985.