Interactive Graph Layout

Tyson R. Henry   Scott E. Hudson
Department of Computer Science
University of Arizona, Tucson AZ 85721 *

Abstract

This paper 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. An architecture is present to support graph exploration. It contains methods for building custom layout algorithms hierarchically, interactively decomposing large graphs, and creating interactive parameterized layout algorithms. As a proof of concept, examples are drawn from a working prototype that incorporates this methodology.

1 Introduction

Directed and undirected graphs provide a natural notation for describing many fundamental structures of computer science. Data base schema [2], for example, can be described using connected graphs. Visual programming notations [13] rely heavily on graphs—viewing a visual program is a similar problem to viewing general graphs. Hypertext navigation [5] can also be modeled with connected graphs. Unfortunately graphs are hard to lay out in an easy to read fashion.

Traditional Graph layout algorithms have tried to increase the readability of graphs by focusing on the task of minimizing specific fixed properties. The following list contains some common graph aspects traditional algorithms focus upon [3].

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

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

Unfortunately, minimizing these kinds of aspects tends to be a very difficult computational problem. In fact finding a layout which simply minimizes edge crossings in a general graph is NP-complete [9]. Approximate solutions can be found by heuristics that run in polynomial time. Nearly 200 papers dealing with graph layout are listed in [3].

Nonetheless, even when traditional layout algorithms do a good job minimizing such properties, they do not always produce the most readable graphs or make the best use of available resources such as screen space. The reason for this is that the "best" layout depends on what information the user is currently focused upon. Consequently, a single canonical layout algorithm cannot always produce the best results. Users must be able to interact with the layout process so they can customize it to meet their current needs and interests.

For example, users should be able to customize the layout so it will focus on the components or aspects of the graph they think are the most important [10]. 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 his 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 build lots of graphs, each of which focuses on a few substructures that the user thinks are important [6]. This method allows the user to explore the graph and learn about it in comprehendable pieces.

This paper presents a novel methodology for viewing large graphs. The basic concept is to provide the users with interactive tools that allow him to iteratively dissect large graphs into manageable pieces. The goal is to provide a system that is powerful enough to allow non-programmers to break up the graph into pieces that match his current focus. In addition to partitioning the graph, the user must be able to lay out the portions in a manner which clearly reflects his current focus.

Three new concepts for building interactive graph layout toolkits will be presented. The first concept is an architecture for building new simple 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 to parameterize graph layout algorithms to give the user control over the layout process. Finally, a highly interactive mechanism for selecting portions of the graph that match the user's current focus will be presented.

2 Composing Graph Layout Algorithms Hierarchically

An interactive graph layout system must afford the user as much power as possible to create customized layouts. In order to explore large or overly complex graphs, the user needs to create custom layouts that highlight the particular aspects of the graph he is currently exploring.

Composing graph layout algorithms hierarchically allows the user to create simple[1] new layout algorithms by plugging together existing layout algorithms. Layout algorithms can be plugged together simply by modifying them to layout a set of nodes and a set of layouts produced by other layout algorithms [7].

A similar hierarchical layout solution is used by the Compose system [12]. Compose uses a divide-and-conquer algorithm to subdivide large graphs into subgraphs. Each subgraph is laid out separately using the same algorithm, and the individual layouts are pasted together to create a layout of the total graph.

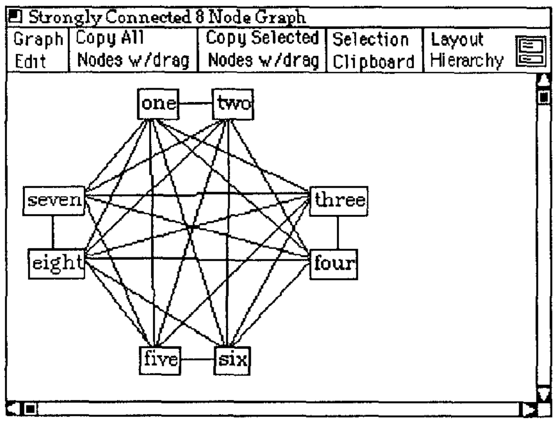

Figure 1: 8 Node Connected Graph

Composing layout algorithms gives non-programming users the power to change the focus of a layout by building a new layout algorithm. The first part of this section introduces the concept of composing graph layout algorithms hierarchically by way of example. Then the simple architecture used to hierarchically combine heterogeneous algorithms is presented. Section 5 presents interactive structures to handle composing layout algorithms.

As an example of composing layout algorithms, consider a completely connected graph. One good way to layout a connected graph is by placing the nodes on the perimeter of a circle. Figure 1 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 1 was created by combining several in-

[1]This method cannot be used to create highly mathematical or generalized layout algorithms.

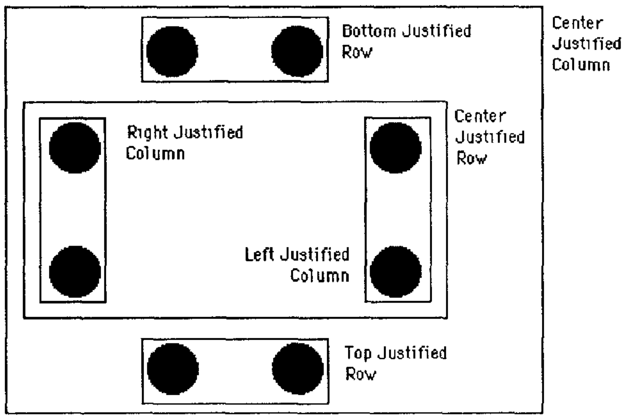

Figure 2: Hierarchy of Layouts for Figure 1

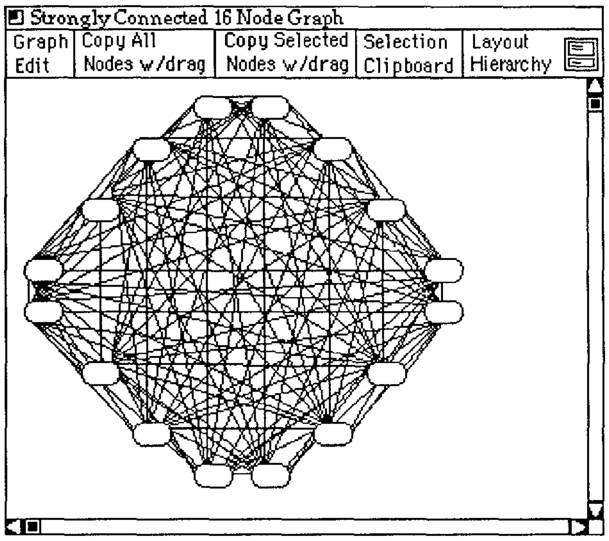

Figure 3: 16 Node Connected Graph

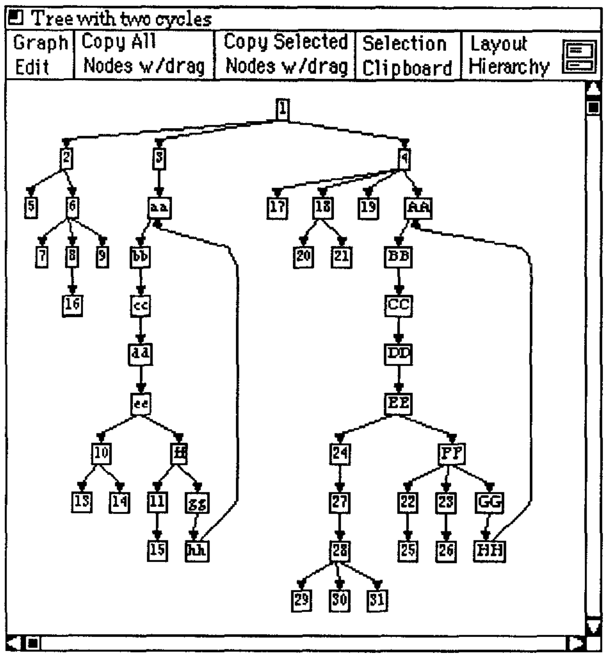

Figure 4: Tree with Two Cycles

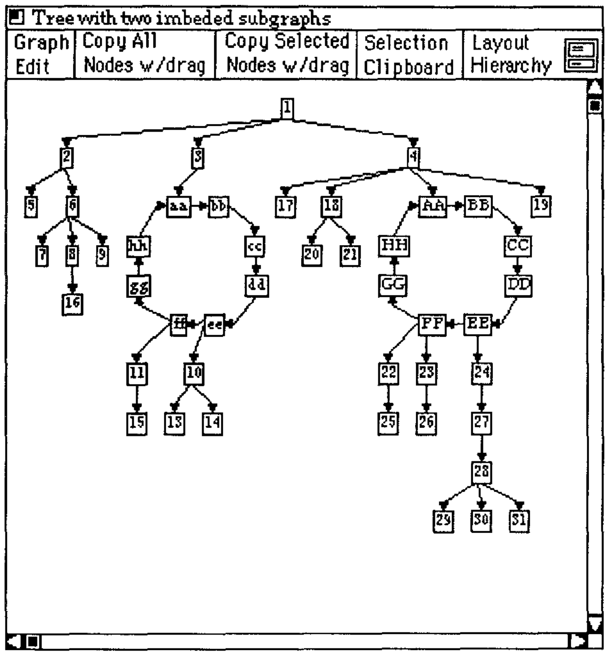

Figure 5: Same Graph as Figure 4 Laid Out Using the Composed Cycle Algorithm

stantiations of a row layout algorithm. This row layout algorithm simply positions nodes in a row (optionally a column) with nodes justified on the top, center or bottom (left, center, or right for columns). Figure 2 shows the hierarchy of layout algorithms used to produce this layout. Figure 3 shows the same concept applied to a larger graph. (Since the individual row algorithms lay out their nodes in the order in which they are passed to it, the nodes must be manually ordered).

Using hierarchical layout algorithms is particularly simple in very regular graphs. The example shown above is highly regular and thus the layout algorithm could be composed out of multiple copies of a single layout algorithm. More interesting layouts can be created by combining different algorithms. The graph shown in Figure 4 contains two cycles. These cycles are visible but not entierly obvious. The graph is laid out by a hierarchical layout algorithm based on ideas presented in [14, 4]. Figure 5 is the same graph shown in Figure 4. It is laid out using the composed algorithm shown in Figure 2 (the same algorithm used to layout the graph shown in 1) to position the cyclic components and the hierarchical layout algorithm to position the remainder of the graph.

Neither of the layouts shown in Figures 4 and 5 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 4 clearly shows the width 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 5 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 he explores the graph. Hierarchies of layout algorithms provide great versatility and provide the user with the power to create new layout algorithms.

Important graph substructures can often be obscured by the layout algorithm. In the example above, the hierarchical layout somewhat obscured the cycles while the combination algorithm clearly highlighted the cycles. Hierarchical combinations of layout algorithms can draw important graph structures clearly, while allowing the algorithm that best presents the overall structure to layout 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 he is currently interested in. Without this power, the graph features the user is focused upon may not be clearly presented in the layout.

Figure 6 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 concen-

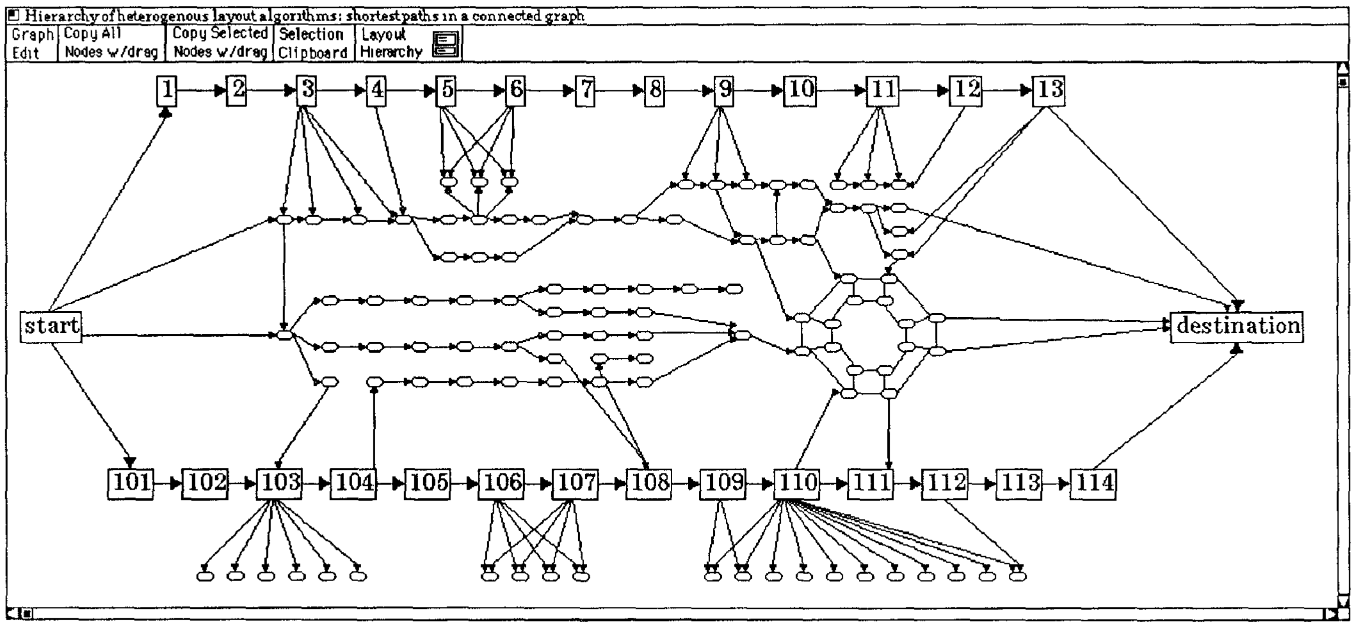

Figure 6: Highlighted Shortest Paths

tric rings—are laid out in such a manner to convey their actual structure. This composed layout algorithm was built to layout this particular graph. Several interactive iterations were necessary to fine tune the graph so it would clearly highlight the two shortest paths. While this "new" layout algorithm does a good job of laying out this particular graph, it clearly is not a general layout algorithm. On the other hand, compared to programming a new layout algorithm this one was trivial to create.

Thirty two instantiations of two different layout algorithms–row and hierarchical–were used to lay out the graph in Figure 6. While each of these instantiations is fairly simple and was easy to create for the example, the structure of the layout algorithms is nontrivial. Section 5 introduces an interactive architecture for displaying and editing the algorithm hierarchy.

The technical problem of combining heterogeneous layout algorithms is solved by requiring each layout algorithm to create a uniform interface. Because of their simplicity, convex hulls are used as a uniform interface. Each layout algorithm calculates the smallest convex hull that completely contains all the nodes it positions. Since nodes can be arbitrary size and shape, they must also calculate the smallest convex hull that contains them. Since all layout algorithms and all nodes create convex hulls, layout algorithms can treat nodes and sublayouts uniformly. Traditional layout algorithms can be converted to be used in a hierarchical layout system simply by incorporating the concept of laying out convex hulls.

Before any layout algorithm positions its nodes and sublayouts, it requests each of its sublayouts to layout all of their nodes. Thus, the layout process is done strictly bottom up–the layout algorithms furthest from the root calculate their layout first. Performing the layout process bottom up has the disadvantage that sublayouts must be laid out in isolation and thus cannot incorporate layout aspects of neighboring layouts. To avoid the problems associated with cycles, layout algorithms are composed in a strict hierarchy.

Hierarchical structuring of layout algorithms has shown to have several advantages over traditional monolithic layout algorithms. Firstly, and most importantly, it allows users to compose new highly specialized layout algorithms without programming. Secondly, hierarchically composed layout algorithms are particularly good at clearly presenting the structures of subcomponents of the graph. Finally, users can directly and easily interact with individual layout algorithms, 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.

3 Parameterized Layout Algorithms

Users can often perceive changes to generated layouts that would help highlight the information they are cur-

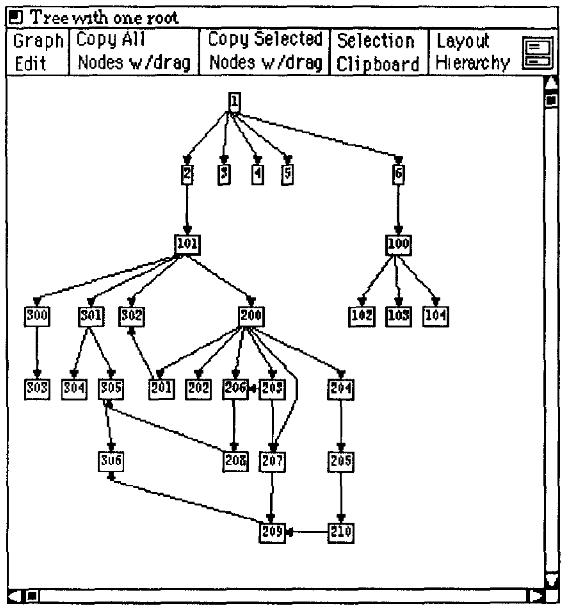

Figure 7: Tree with One Root

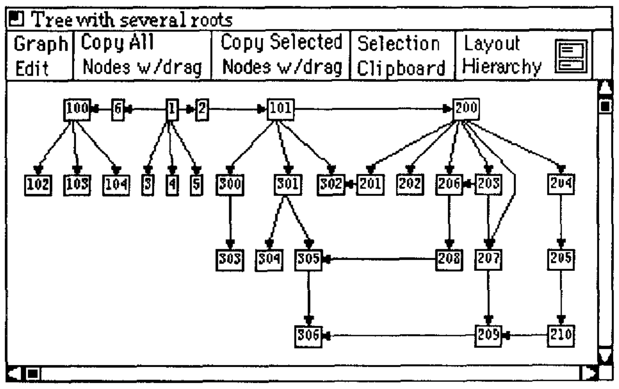

Figure 8: Tree with Multiple Roots—Same Graph Shown in Figure 7

rently focused upon. For example, changing the root node used in calculating a hierarchical layout can drastically change the graph, and possibly change the focus of the layout to reflect the user's interests.

Changes to the layout must be incorporated back into the layout process, or the next iteration of the algorithm will reverse the changes. One possible solution is to create constraints that reflect the changes and incorporate them back into the layout process [1]. This method has two disadvantages. The first is that the user can create a set of inconsistent constraints forcing the constraint solver to delete constraints. The second disadvantage is that while the user may be able to see the system of constraints that resulted from his changes, it would be very hard for him to interpret them and understand how they will affect the layout.

Another method to support customization of layouts is to directly interact with the layout algorithm through a set of parameters. This has the advantages that the changes can always be visible through an interactive interface and can easily be incorporated back into the layout process [7].

The most versatile type of parameters are sets of nodes. Parameterizing layouts with sets of nodes provides a powerful mechanism for specifying the focus of a graph, giving the user great control over the layout process. As a simple example consider the task of laying out a connected graph as a tree. The layout shown in Figure 7 was produced by the same hierarchical layout algorithm used in the previous examples. In this layout the algorithm was parameterized by the single root "1." Figure 8 shows the same graph laid out by the same algorithm, but parameterized by the set of roots "1","2","6","100","101","200".

The layouts shown in Figures 7 and 8 are significantly different views of the same graph. Parameterizing layout algorithms is a powerful graph customization tool. Users can modify the layout (sometimes drastically) simply by choosing new parameterization sets. This mechanism allows the user to customize layouts to produce a layout that illustrates the aspects of the graph he is currently interested in.

While it is not always obvious which set of nodes will produce the best graph, choosing sets can be simplified by providing an interactive interface. A quick and easy interface will make it possible for the user to iteratively try many different parameterization sets. This iterative approach allows the user to explore the graph at the same time he builds custom layouts. This process is explorative because as the user tries different layouts he discovers aspects of the graph that may not have been clear in the previous layouts.

The incorporation of parameterization sets into layout algorithms makes them considerably more general but does not necessarily complicate the implementation of the algorithm. The layout algorithm used to lay out the graph shown in Figures 7 and 8 was quite trivial to convert from a single root to a set of roots. This particular algorithm was modified by creating a dummy root connected to all the nodes in the parameterization set. The single root algorithm was then applied to the dummy root.

Layout algorithms can also be parameterized by including options in the algorithm. For example, the row layout algorithm used to compose the cycle layout algorithm used in Figure 5 contains the options to isolate or close its nodes. (An isolated layout draws the convex hull around its nodes and terminates all edges that cross

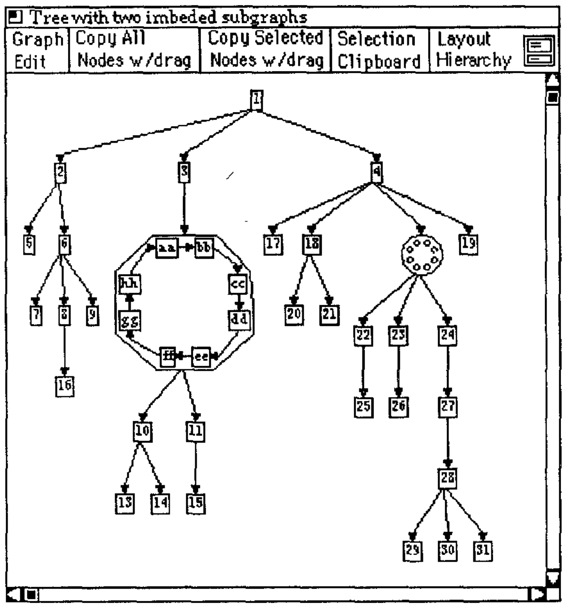

Figure 9: Tree with Two Cycles

this boundary at the boundary. Closed layouts draw a bitmap instead of all of its nodes and edges.) As an example of the close and isolate options, consider the layout shown in Figure 9. This is the same graph laid out by the same algorithms as that shown in Figure 5. The only differences are that left cycle has been isolated from the graph, the right cycle has been closed and the vertical spacing between the nodes has been increased. Even though the cycle layout algorithms has not been explicitly programmed to contain the close and isolate options, since it was built out of row layout algorithms, it inherited the options.

Providing direct access to layout options gives the user additional control over the layout process. Once again, this allows the user to use an iterative approach trying lots of different layouts as he explores the graph. As an example of some possible options, the following list contains a few options to the hierarchical layout:

- Orientation (vertical, horizontal)
- Search method (depth first, breadth first)
- Justification of children (center, right, left)

Layout algorithms can also be parameterized by changing constants used during the layout process. As a simple example, the vertical space between nodes in the hierarchical layout can be interactively changed. Each layout algorithm has a different set of variable parameters that allows the user to customize the layout.

Unlike using constraints to incorporate customizations, all three types of parameterization–sets, options, values–are trivial to incorporate into the layout process.

Parameterized layout algorithms are especially powerful when combined with a hierarchy of layout algorithms. The combination gives the user the ability to customize portions of the graph by changing the parameters to the algorithm responsible for that portion. It would be considerably more difficult to write monolithic layout algorithms that allow arbitrary portions of the layout to be customized. Not only would it be hard to keep track of which portions of the graph the user had modified, since these changes can be arbitrary, it would be hard to include these changes in the layout process.

4 Subgraph Selection

In order to successfully navigate through large graphs, the user must be able to select arbitrary pieces of the graph that are small enough to look at and reflect his current focus. For the system to be truly conducive to learning and exploration, the user must be able to quickly select subgraphs and apply a layout algorithm to them creating a view—an arbitrary subgraph coupled with a layout algorithm. The user must be able to iteratively create views of the graph until he finds a view, or set of views, that meets his current needs. The process of selection must be easy to use and versatile, otherwise it will interfere with the user's exploration.

There are two basic types of selection, manual selection and algorithmic selection. Manual selection allows the user to select nodes in the graph by pointing at them with the mouse, or by dragging a rubber-banded rectangle over some collection of nodes. This method works well if the nodes to be selected are few in number and are positioned near each other in one of the current views.

Selection algorithms traverse the graph marking nodes as being selected. They can work well even if the nodes to be selected are not currently visible or are spread throughout the graph–this is often the case with large graphs. They also allow for arbitrarily complex criteria to guide the selection.

As an example of selection, consider the graph shown in Figure 10. The nodes "root" and "destination" have been manually selected. Figure 11 shows the result of applying a shortest path selection algorithm to the graph. This algorithm which is parameterized by two nodes—"root" and "destination" in this example–selects all the nodes on the shortest path between the two parameter nodes.

Once a set of nodes is selected, it can be copied into a new view. Figure 12 shows a new view of the exact same graph shown in Figure 11. The selected nodes have been laid out by the row layout algorithm highlighting the shortest path while the remainder of the graph was laid out in relation to the shortest path.

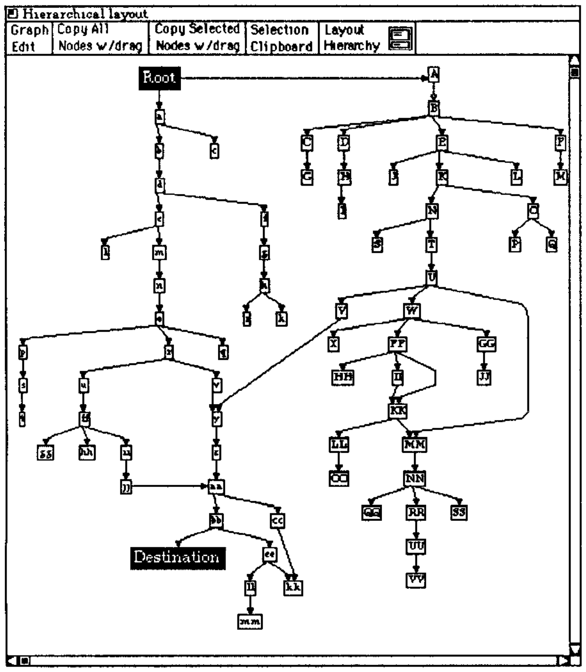

Figure 10: Sample Connected Graph

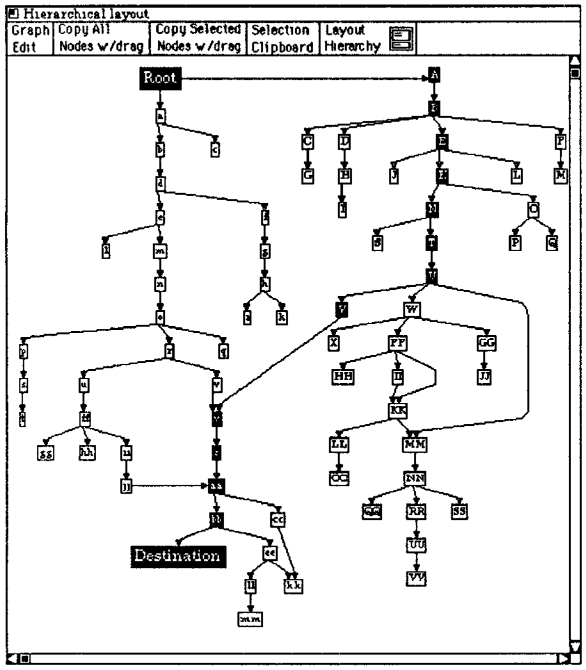

Figure 11: Shortest Path—Same Graph Shown in Figure 10

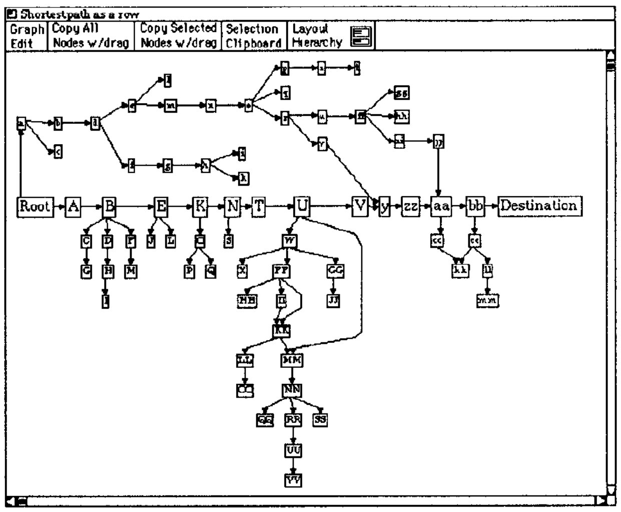

Figure 12: Modified Connected Graph—Same Graph Shown in Figures 10 and 11

Algorithmic selection only allows the user to apply selection algorithms that have been programmed into the system. It can however be expanded by allowing new selection algorithms to be built by plugging together existing selection algorithms. Figure 13 shows a selection algorithm built by plugging together existing algorithms. This algorithm selects the two most connected nodes in the graph and all of their immediate neighbors.

While plugging together selection algorithms is not a completely general mechanism, it does provide non-programmers with the ability to create new selection algorithms and thus gives them additional power to customize layouts.

5 Interactive Structures

This section describes the interactive structures used to build the interfaces to hierarchical compositions of layouts and the interface for the presented selection methodology.

It is imperative that the interface to the selection mechanism be quick and easy to use. Users must be able to create lots of views so they can home in on the aspects of the graph they are currently interested in. If the interface is too cumbersome, or difficult to use, the user could get bogged down in the task of using the system instead of exploring the graph. Since selection and layout algorithms can be parameterized by multiple sets of nodes, the interface must allow the user to create multiple sets.

Selection sets–sets of selected nodes–are represented in two different manners. The first is by drawing a copy of each selected node in a labeled box. This method

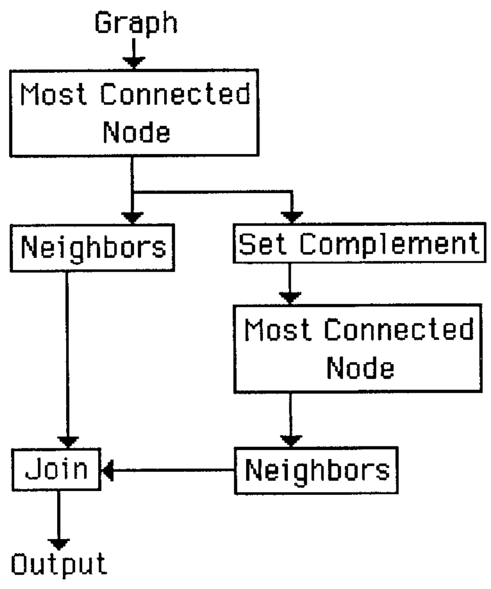

Figure 13: Composition of Selection Algorithms: Selects two Most Connected Nodes and their Immediate Neighbors

works well when the set is small and when the nodes are distinct. If all the nodes in the graph look the same, for example the graph shown in Figure 3, drawing copies of nodes is useless. The second method is to use an icon to represent the set and highlight all the selected nodes in the graph. Figure 11 shows a graph in which all the nodes in the current selection set are highlighted. In order not to confuse the selection sets, there can be only one current selection set at any time.

The user interacts with the selection sets through a direct manipulation dialogue box. Each set is represented by a labeled icon–this icon can be opened to display pictures of each node. The set can be marked as the current selection set simply by clicking on its icon. The icons can also be repositioned in the dialogue box simply by dragging them with the mouse.

Instantiations of selection algorithms are represented by icons. Each icon contains a connection port for each input and output set. Selection algorithms are composed by connecting together the selection set icons and the selection algorithms icons. Once the icons are connected, new selection sets can be calculated by pressing the "Execute" button.

Each view of the graph is displayed in its own window. New views are created by first creating a new window via menu selection. Nodes can then be copied into the new view by dragging the "Copy" or "Copy Selected Nodes" button from the source view into the destination view. The copy procedure only inserts unique node

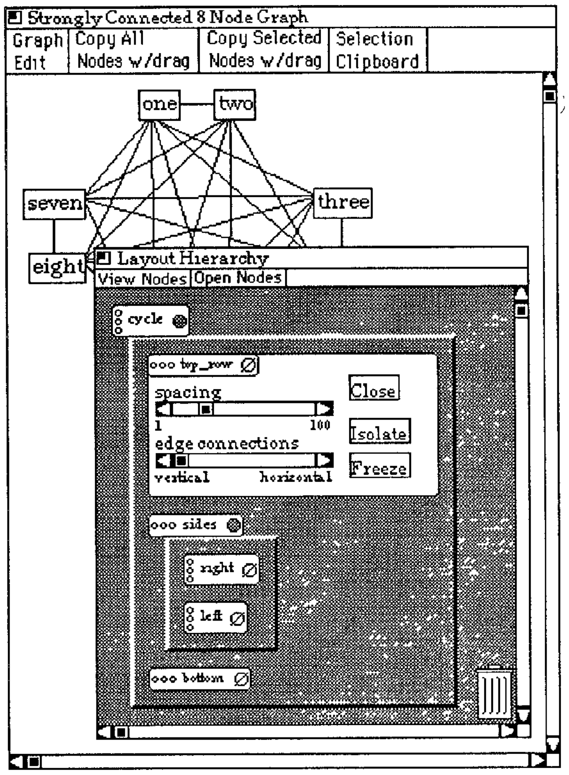

Figure 14: Metagraph of Layout Hierarchy

into the destination view. In other words, nodes cannot be copied into a view more than once. Edges are automatically copied into the new view if both of the nodes they connect to are in the new view.

While the hierarchical composition of layout algorithms gives the user great versatility to compose new layout algorithms, it can be difficult to determine the layout algorithm hierarchy. One solution would be to superimpose some information on top of the layout to portray which algorithm was positioning which nodes. This has the severe disadvantage of complicating and obscuring the actual layout. An alternative solution is to provide a *metagraph* to present the algorithm hierarchy. Figure 14 shows the metagraph for the layout shown in Figure 1.

There are three main functions of the metagraph. First, it conveys the structure of the hierarchy of layout algorithms to the user. Second, it allows the user to manipulate the hierarchy and finally it provides an interface for the parameters of individual layout algorithms.

Each layout algorithm in the current view is represented by a named *metanode*[2] in the metagraph. When

[2]The node titled "top_row" in Figure 14 is an example of a metanode. It represents the layout algorithm that lays out the top row.

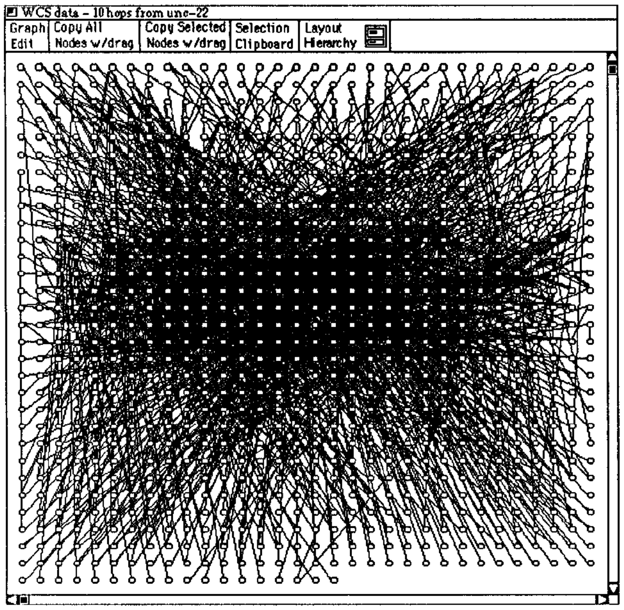

Figure 15: Nematode Reference Database

the metanode is completely closed, only the name of the node, an icon depicting the layout algorithm, and an indicator if it has children is shown. The metanode can be opened to show the algorithm's control panel and/or the metanodes of its children layouts. In the metagraph shown in Figure 14, only the topmost metanode "cycle" has children; it is shown in its open state. The metanode "top_row" is shown with its control panel open. The user can directly interact with this control panel to change the parameters to the row layout algorithm that positions the nodes in the top row of the layout. For example, if the user changed the value of the spacing parameter by interacting with the slider labeled "Spacing," the layout algorithm would recalculate the position of its nodes (nodes "one" and "two" in this example) and the new layout would be drawn.

Hierarchies of layout algorithms can be modified by rearranging the order of the hierarchy. This can be accomplished by changing the hierarchy of metanodes. Metanodes can be interactively dragged around the metagraph and positioned inside other metanodes. The actual layout is automatically updated to match the current structure of the metagraph. As soon as a metanode is moved, the structure of the graph is updated and the resulting new layout is displayed on the screen.

New compositions of layout algorithms can be created and new layout algorithms can be added to existing composition by selecting algorithms from a menu and positioning them within the metagraph. Graph nodes can be moved between layouts by selecting an active selection set and placing it into a specific metanode.

6 Conclusion

The examples shown in this paper are intentionally small to conserve space. It takes several iterations on large graphs before the user can create a reasonably sized layout—on the order of the larger examples shown in this paper. The results however, are considerably more dramatic with large examples. As an example of a large graph, Figure 15 shows a small portion of a 10,000 node graph of data relationships in a molecular biology information system application. This graph was laid out using a very naive grid layout algorithm so it would fit on the page. It could have been laid out with the hierarchical layout, but the resulting graph is so large it would take several pages to print it. Figure 16 shows a piece of this graph that was interactively selected from the total graph, and then laid out using the hierarchical layout algorithm.

The most global contribution of the presented work is that it puts the power to explore a graph and create layouts in the hands of non-programmers. Users who are experts about the data in a particular graph can navigate through the graph and create layouts without programming. Three new concepts were presented to provide non-programmers with this power:

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

There are two primary future directions: using domain specific graph semantics to guide the layout and the selection, and creating a methodology that can be used to build an interactive system for non-programmers to specify selection and layout algorithms.

General graph semantics have been used to help guide the layout process [11], domain specific semantics could provided even more information for the layout process. Subgraphs in domain specific graphs are often well known, and thus layout and selection algorithms can be created to recognize these subgraphs and automatically extract them and lay them out in such a way as to preserve their semantic significance.

Once graph semantics are incorporated in the system, it will be necessary to create many specialized selection and layout algorithms. Each specialized algorithm will encode domain specific semantic aspects of the graph. Since end-users know the most about the semantics of their graph, an architecture must be created that will allow non-programming end-users to specify new selection and layout algorithms and incorporate them into the toolkit.

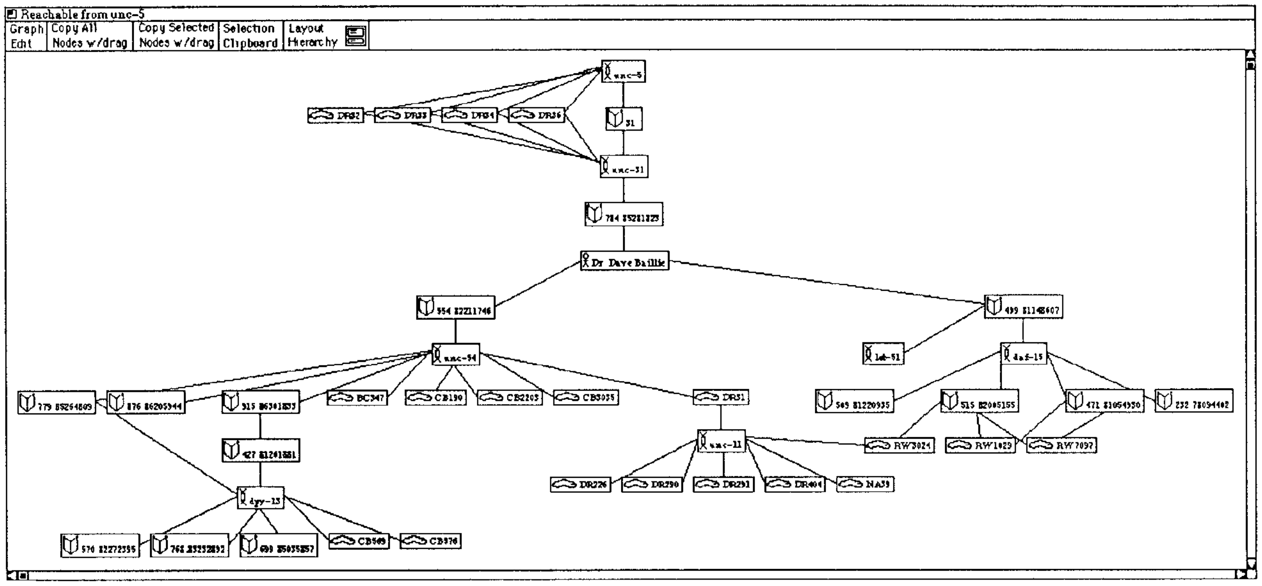

Figure 16: Nematode Reference Database

References

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

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

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

[4] Gansner, E. R., North, S. C., Vo, K. P., DAG—A program that Draws Directed Graphs, *Software-Practice and Experience*, **18** No. 1 (November 1988) pp. 1047-1062.

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

[6] 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).

[7] Henry, T. R., Interactive Graph Layout: the Exploration of Large Graphs, *Ph. D. Dissertation, Department of Computer Science, University of Arizona*, in preparation.

[8] 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*, pp. 112-121, (October 1990).

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

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

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

[12] Messinger, E. B., Lawrence, A. R., 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, (January 1991).

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

[14] 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).