A FRAMEWORK FOR COMPLEXITY MANAGEMENT IN GRAPH VISUALIZATION

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Burkay Genç

September, 2002

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Uğur Doğrusöz (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Uğur Güdükbay

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Oya Ekin Karaşan

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

A FRAMEWORK FOR COMPLEXITY MANAGEMENT IN GRAPH VISUALIZATION

Burkay Genç
M.S. in Computer Engineering
Supervisor: Assist. Prof. Dr. Uğur Doğrusöz
September, 2002

In this thesis we describe a comprehensive and efficient framework for development of complexity management techniques in graph visualization tools. The presented architecture is capable of managing multiple associated graphs with navigation links and nesting of graphs as well as ghosting, folding and hiding of unwanted graph elements. The theoretical analyses show that the involved data structures and operations on them are quite efficient, and an implementation in a graph drawing tool has proven to be successful.

Keywords: Graph Drawing, Compound Graphs, Information Visualization, Software Engineering.

ÖZET

ÇİZGE GÖRÜNTÜLEMESİNDE KARMAŞIKLIĞIN YÖNETİMİ İÇİN BİR ÇERÇEVE ÇALIŞMA

Burkay Genç
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Assist. Prof. Dr. Uğur Doğrusöz
Eylül, 2002

Bu tezde, çizge görüntüleme uygulamalarında karmaşıklık yönetim teknikleri geliştirmek için etraflı ve verimli bir çerçeve çalışması sunuyoruz. Sunulan mimari birbiriyle ilişkili birden çok çizgeyi yönetme ve çizgeleri yuvalandırma yetisine sahip olduğu gibi, istenmeyen çizge elemanlarını bulanıklaştırma, dosyalama ve gizleme imkanı da tanımaktadır. Teorik incelemeler, kullanılan veri yapılarının ve bu yapılar üzerine bina edilen operasyonların oldukça verimli olduğunu göstermektedir. Bu yapılar ve operasyonlar bir çizge çizim uygulamasında kullanılmış ve başarısı ispatlanmıştır.

Anahtar sözcükler: Çizge Çizimi, Bileşik Çizgeler, Bilgi Görüntülemesi, Yazılım Mühendisliği.

Acknowledgement

. . . to . . .

my family, for their endless support

my advisor, for his trust

my friends, for making life bearable

my God, for the family, advisor and the friends

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are commonly used to model relational information that arises in world wide web analysis, relational databases, biochemical networks, telecommunication networks, financial analysis, software engineering and geographical studies. Objects are the *nodes* in a graph; relations or links are the *edges* in a graph. The usefulness of the relational model depends on whether the drawing, or the layout, of the graph effectively conveys the relational information to the users. A poorly drawn diagram with a large number of graph elements confuses the user of an application, while a well laid out diagram with a reasonable number of graph elements improves the user's comprehension of the data.

In this thesis we present a comprehensive framework for a variety of complexity management techniques in graph visualization. This framework meets the industry requirements for generality (works for all sorts of directed and undirected graphs), efficiency (works well within an interactive tool), and extendibility (can be easily customized). It supports complexity management via navigation links, association of multiple related graphs, nesting of graphs within others, ghosting, folding, and hiding. We firstly introduce the definitions used in the area in Chapter 2 and explain related work done on these topics in Chapter 3. Later, we introduce a new concept, the graph manager, to build a structure for supporting the techniques and visualization styles we have proposed, in Chapter 4. We keep on with the tips and tricks of drawing a graph manager, and we

discuss in detail the creation and transitions between an intergraph edge and a meta edge in Chapter 5. Chapter 6 discusses the ways of managing complexity using our framework and shows how one can implement the well known complexity management techniques using our framework. It shows that the techniques which have been separately invented by different researchers, may easily be inserted into our framework. The class structure scoping the proposed architecture is given in Chapter 7. Chapter 8 discusses an implementation of this structure within Tom Sawyer Software's Graph Editor Toolkit for Java. We conclude in Chapter 9 by discussing the results of our study and the concepts that have been introduced to the field with this thesis.

Chapter 2

Definitions

2.1 Base Definitions

Most of the terms and definitions below are from [3].

A *graph* G is defined by two finite sets V and E, such that $E \subseteq [V]^2$. The elements of V are the *nodes* (or *vertices*) of G, and the elements of E are the *edges* of G. An edge e is given as (u, v), where $u \in V$ is the *source node* of e and $v \in V$ is the *target node* of e.

A *path* P $(= x_1x_2x_3 \ldots x_i \ldots x_{n-2}x_{n-1}x_n)$ is defined as a non-empty graph of n vertices, such that all vertices in the sequence are distinct from each other and an edge exists between x_i and x_j, iff $i = j - 1$ where $i = 1, 2, \ldots, n - 1$ and $j = 2, 3, \ldots, n$. Given a path P, by adding the edge (x_n, x_1) we obtain a *cycle*. A graph is called *acyclic* if there are no cycles in it.

A non-empty graph $G = (V, E)$ is called *connected* if for any pair of $u \in V$ and $v \in V$ we have a path between u and v in G.

An acyclic graph is also named as a *forest*, and a connected forest is called a *tree*.

A *rooted tree* T is defined by a node set V, an edge set E, and a node r, such that for every node $u \in V - \{r\}$, there is a unique path P from r, the *root* of the tree, to u.

2.2 Compound Graphs and Graph Managers

A *compound graph* [6] CG is defined over a graph $G = (V, E_G)$ and a rooted tree $T = (V, E_T, r)$ with the same set of vertices (Figure 2.1). The vertex set V may be divided into two subsets V_B and V_S, where the nodes belonging to V_B are the leaves of T and denote the *base nodes* of the graph and the nodes belonging to V_S denote the *subgraphs* of the graph. An existing edge in E_T denotes a *nesting relation* between the source and target nodes of the edge. For example, if $e = (u, v) \in E_T$, this claims v is nested in u. Depending on the context it may also be called an *inclusion relation*, where node u includes node v.

A *graph manager* $M = (S, F)$ is a variant of compound graphs, defined by a *graph set* $S = \{G_1, G_2, \ldots, G_l\}$ where $G_i = (V^{G_i}, E^{G_i})$ and a *navigation forest* of rooted trees $F = \left(V^F, E^F\right) = T_1 \cup T_2 \ldots \cup T_k$. Each graph $G_i \in S$, each node $v \in V^{G_i}$, and each edge $e \in E^{G_i}$ is represented by a distinct node in V^F. For each node $v \in V^{G_i}$, there exists an edge $(G_i, v) \in E^F$ and for each edge $e \in E^{G_i}$, there exists an edge $(G_i, e) \in E^F$, representing ownership relations in the graph manager. Then G_i is called the *owner* of v (or e); conversely v (or e) is called a *member* of G_i.

A *navigation link* associates a member of a graph and another graph. Each such link is represented in the navigation forest by an edge $(m, G_i) \in E^F$ between a member (a node or an edge) m and a graph G_i, where G_i is not the owner of m. We say the graph member m *navigates* to the associated graph G_i; and G_i is said to be the *child graph* of the *parent member* m. Conversely, the owner of the graph member m is called the *parent graph* of G_i.

Another way of associating two different graphs in a graph manager $M = (S, F)$ is via the intergraph edges. Let $u \in V^{G_i}$ and $v \in V^{G_j}$ be two nodes where

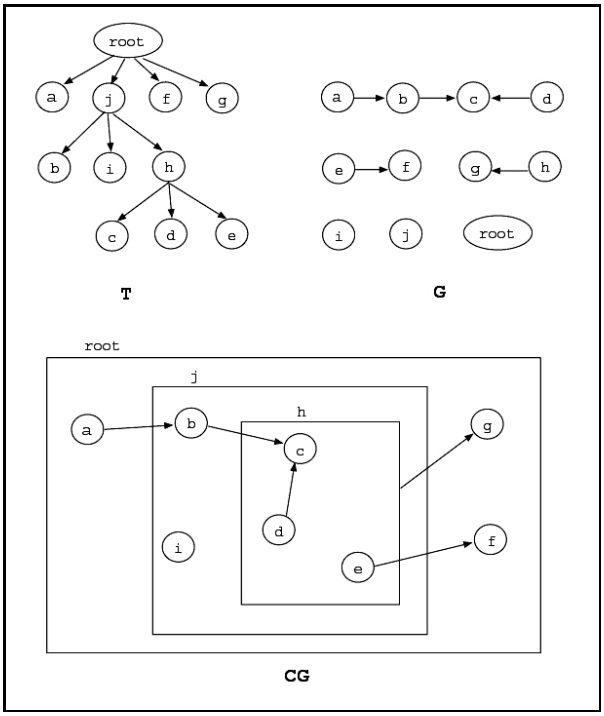

Figure 2.1: A compound graph CG, and its corresponding components: graph G and rooted tree T.

$G_i \neq G_j$ and $G_i, G_j \in S$. Then the edge (u, v) is called an *intergraph edge*, representing a relation between objects (nodes) that belong to different entities, graphs G_i and G_j in this case. An intergraph edge $e = (u, v)$ is owned by the lowest common ancestor graph of the two respective owner graphs of end-nodes u and v in the navigation forest F.

The major differences between compound graphs and graph managers are the graph manager's capability of handling multiple graphs and intergraph edges.

Chapter 3

Related Work

A great deal of research in graph drawing [2, 4] has been done over the past couple of decades. As graphical user interfaces have improved, and more state-of-the-art software tools have incorporated visual functions, interactive graph editing and diagramming facilities have become important components in visualization systems. The increase in the size of the information (e.g., size of information databases and the complexity of their structures) to be visualized forced the demand for more sophisticated complexity management techniques for many applications.

Many researchers in the area have incorporated different techniques and solutions. For instance, VCG [10] is a tool with ability to fold nodes and edges under certain cases and hide edges of specific types. D-ABDUCTOR [8] is another tool with support for compound graphs [13] and information hiding via expand and collapse operations. Higres [9], on the other hand, is a visualization system for clustered graphs and has support for handling compound graphs. Many other tools have incorporated complexity management techniques according to their application specific needs, none providing a comprehensive set. As a result, the following techniques have been introduced to the graph drawing community:

- **Zooming / Panning**: Zooming and panning are the most straightforward approaches for managing complexity in graph visualization [7]. The

zoom and pan interface may be used for selectively displaying a part of the compound graph to allow detailed visualization. Zooming may have two forms: geometric zooming and semantic zooming [7]. Geometric zooming is based on changing the drawing transformations for obtaining a view with larger graph components. Semantic zooming on the other hand is used for altering the details displayed in graph components. For example, by semantically zooming in the graph, one can make the labels of the nodes viewable, which were not available in the current zoom level. Panning helps zooming by providing a mechanism to select the part of the graph to be zoomed in/out.

- **Fisheye Distortion**: Fisheye distortion [5, 11, 12] is another well known technique for obtaining a detailed view of a certain part of the compound graph. Generally the graph visualization is focused on a certain point and the environment around this point is zoomed depending on the distance from this point. A curved view of the graph components around this point is obtained as a result of this technique. Figures 3.1 and 3.2 show the effect of fisheye distortion on a graph [11].

- **Nesting**: Nesting is the most common technique used for visualizing compound graphs. It is simply based on drawing a group of nodes inside another one for emphasizing an inclusion relation. However, nesting alone does not help decrease the complexity of the compound graph, since all the nodes of the compound graph are still viewable. One can interpret nesting as a way of grouping similar information for easy referencing purposes.

- **Expanding / Collapsing**: Expanding and collapsing operations are defined as extensions on the nesting technique which make it more powerful by adding complexity management possibility. A nesting relation may be avoided at a certain state of the drawing by collapsing the node which is the base for nesting. This makes all the content, which is immediately or deeply nested under the node to be collapsed, unviewable upon collapse, thus reduces the number of visual elements in the compound graph. Expand is the inverse of the collapse operation and it makes all the content nested into a node visible.

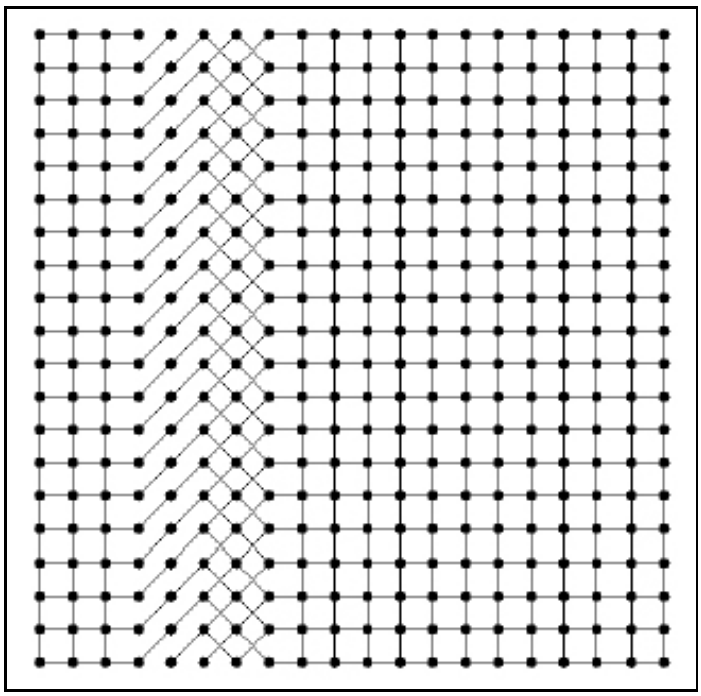

Figure 3.1: Graph before applying fisheye distortion.

- **Folding**: Folding is a technique used for creating nesting relations between a group of graph members and a folder node. The affected compound graph elements are grouped together and placed under a single node. The default nesting state may allow or disallow visual representation of the folded elements. In other words the default nesting state depends on the implementation of the framework. Either an expanded folder or a collapsed folder may be accepted as an initial parent.

- **Hiding**: Hiding is another solution for reducing the number of elements that require a visual representation. Different than folding, hiding totally makes the grouped elements invisible. They are neither visible by themselves nor as a folder node anymore. The only way to make them visible again is to undo the hiding operation.

- **Ghosting**: Ghosting is the least effective way of complexity management, since it only decreases the visual attractiveness of a group of compound graph members by reducing color intensity of them. Zooming/Panning,

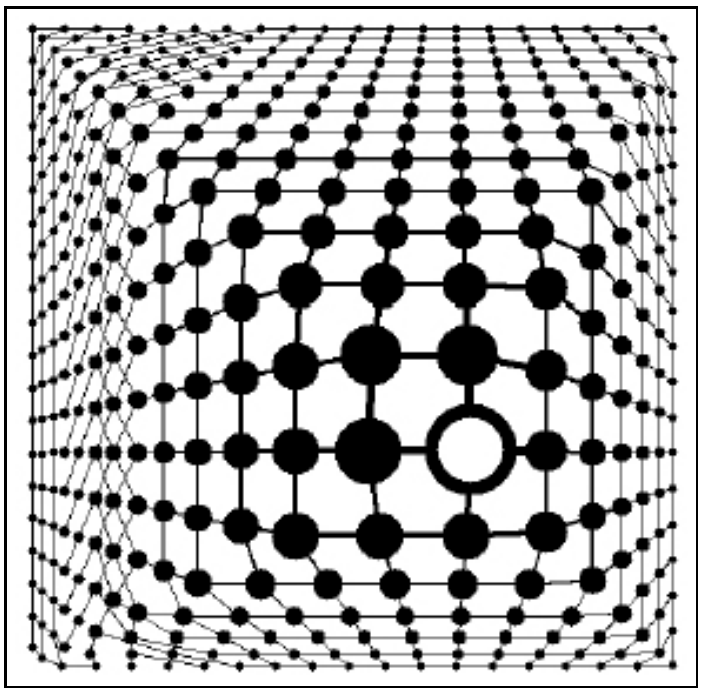

Figure 3.2: Graph after applying fisheye distortion.

Fisheye Distortion, and Ghosting are display layer features and are relatively easy to implement while other operations may be used on abstract graph topologies.

All techniques described above have been implemented in various contexts. However, they have always focused on a single type of problem or environment. Also all implementations only applied a few of these techniques and lacked the others. Although implementing all techniques is not necessary in most applications, we believe describing a framework supporting most of these well known techniques and providing a basis of study for future extensions is a major requirement for the current state of complexity management in graph drawing.

Chapter 4

Graph Managers

A graph manager is an extension of the compound graph previously defined and used in graph drawing [1]. A compound graph is restricted in many ways when compared with the capabilities of a graph manager. First of all, a graph manager is designed to keep multiple graphs rather than a single flat graph structure which is employed by most compound graph implementations. This allows further abstraction levels by more rigid boundaries. In a single flat graph structure, all the nodes and edges are placed in the same graph environment and abstraction is provided virtually via hierarchy trees. However, a graph manager allows placement of graph members into different graphs, thus permits solid boundaries between members of each graph. Since each graph has all the functionality of the flat graph of compound graph, recursively many graphs may be defined dynamically nested into each other. Also a graph manager is capable of managing intergraph edges between nodes which belong to different graphs. Compound graphs do not provide direct support for intergraph edges; instead one needs to use the hierarchy tree to figure out whether an edge in the flat graph is an intergraph one.

Intergraph edges have a very important property which makes them different from normal edges: intergraph edges do not belong to a certain graph. A normal edge, which is defined between two nodes of a certain graph, is assumed to belong to that graph. However, there is no obvious graph that an intergraph edge belongs to. So we use a special mechanism for storing the intergraph edges. Since they

are not bounded to a single graph and require data from more than one graph, we keep them in a special graph in the graph manager, which is called the *intergraph*. The intergraph is special in many ways. First, it is not in direct use of the user. It is hidden from the user and only accessed by the graph manager for its own purposes related with the intergraph edges. Second, it does not own any nodes, since the edges in the intergraph are always intergraph edges and the nodes of these edges are already placed in other graphs. At the time of intergraph edge creation these nodes are not moved, but a new edge is inserted into the intergraph to store the relation among these two nodes. Third, there is a special reference to one of the graphs in the graph manager for each intergraph edge, which is called the lowest common ancestor graph. A *lowest common ancestor graph* for an intergraph edge is the graph furthest to a root in the navigation forest, which is an ancestor of both end-nodes.

Overall, a graph manager is not only responsible for maintaining a set of graphs but also their interrelations (through navigation links and intergraph edges). The contents of the graphs along with navigation links in a graph manager can be represented by a directed navigation forest, defining the "skeleton" of the graph manager. An example navigation forest is given in Figure 4.1 and a pictorial representation of the graph manager with the same navigation forest is given in Figure 4.2.

We will now classify the elements and relations in these figures considering our definitions. $G1$, $G2$, $G3$, $G4$, $G5$ and $G6$ are our graphs. The nodes are $N1$, $N2$, ..., $N11$, $N12$. The edges are $E1$, $E2$, $E3$ and $E4$. The intergraph edges are $I1$ and $I2$. The ownership relations may easily be classified in the skeleton. $G1$ is the first graph in our manager and it is the root of our skeleton. It has 3 nodes ($N1$, $N2$, $N3$), a normal edge ($E1$) and an intergraph edge ($I1$). $G1$ is the owner of all these members. Similarly, $G2$ is the owner of nodes $N4$ and $N5$ and the intergraph edge $I2$. The other ownership relations are defined similarly. As given in the definition, an ownership relation is shown as a directed edge from a graph to a member. A navigation link is defined as an edge in the skeleton from a member to a graph, where the graph is not the owner of the member. Then the edges $(N1, G2)$, $(N4, G4)$, $(N3, G3)$, $(E2, G5)$ and $(I1, G6)$ are our navigation links for

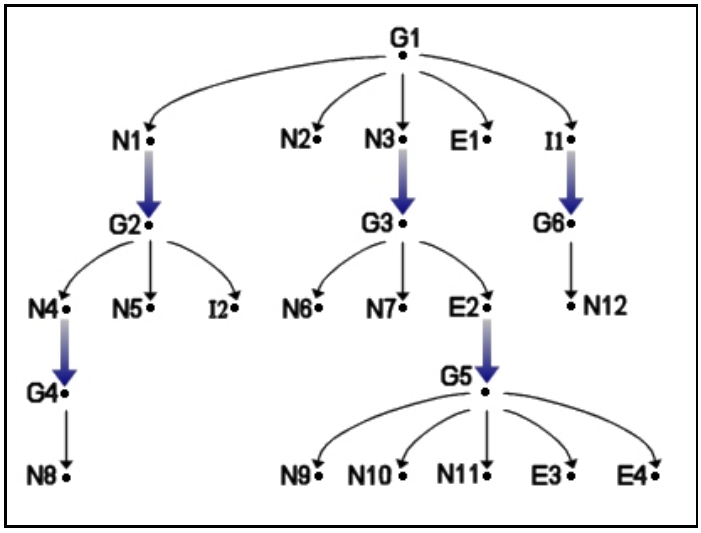

Figure 4.1: The navigation forest of an abstract level graph manager with six graphs, representing ownership (solid) and navigation (gradient) links.

this manager. The navigation link between $N1$ and $G2$ allows a *node-to-graph* navigation. The link between $E2$ and $G5$ allows an *edge-to-graph* navigation and the link between $I1$ and $G6$ allows an *intergraph edge-to-graph* navigation.

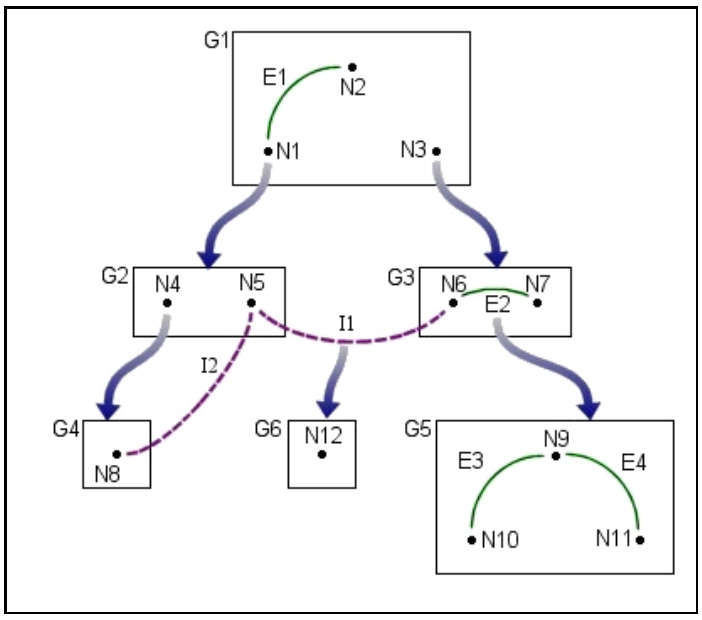

Figure 4.2: A pictorial representation of the graph manager with a navigation forest same as in Figure 4.1. The gradient arrows show the navigation links and the dashed edges are the intergraph edges.

Chapter 5

Drawing Graph Managers

So far we have described a new structure built on top of the compound graphs. Chapter 4 described the basics related with the proposed structure and gave details on the abstract side of the framework. We are now going to explain techniques related with drawing of a graph manager. At the drawing layer every graph member has a pair of coordinates. These coordinates describe the location of the item in its owner graph. Some elements like the node and edge labels, have their coordinates maintained relative to their owner graph members. Section 5.1 will discuss the geometry related issues in detail and introduce a technique to handle relative coordinates among graphs. Section 5.2 will explain how the nesting relations are handled in this framework. Section 5.3 will introduce details of drawing an intergraph edge and Section 5.4 will end up this chapter by giving a detailed analysis of transitions among intergraph edges and their representative meta edges when intergraph edges are unviewable.

5.1 Main Graph and Navigation

A *main graph* is the drawing and transformation base for all other items in a graph manager. The architecture supports viewing the elements of the manager from different points of the structure. Initially, only the main graph and its

immediate members are displayed. We can draw the nodes and edges in the main graph, using a simple transformation from the graph coordinates to the device coordinates. If the graph manager has multiple graphs associated with navigation links, the user might want to change the main graph by following the navigation links (e.g., navigate to the parent or child graph), displaying one graph at a time. For example, assume the current main graph is $G1$ for the graph manager in Figure 4.1. If we use the navigation link $(N1, G2)$, we can change our main graph to $G2$. Then we can use the link $(N4, G4)$ to change the main graph to $G4$. We can also traverse back using the navigation links. So, after setting the main graph to $G4$, we can reuse the link $(N4, G4)$ to set main graph to $G2$. This way, we can navigate among graphs of the graph manager, displaying one graph at a time.

5.2 Nesting

A *nesting* of a graph in its parent node, is a visual representation of an inclusion relation, facilitating the drawing of multiple graphs simultaneously. The node within which a graph is nested is said to be *expanded*. In our architecture, a navigation link is simply an abstract symbolic relation between a node and a graph, which may be realized in a drawing using nesting.

When multiple graphs are to be displayed simultaneously, only the main graph's coordinates are directly used, the other graphs' member coordinates should first be transformed into the coordinate system of the main graph. For example, in the graph manager of Figure 4.2, $N8$ is not a member of the main graph and in order to be drawn, it should transform its coordinates to main graph coordinates. For efficient handling of these transformations, a unique transformation matrix is maintained by each graph to transform its members' coordinates to the coordinates of its parent graph. The graph manager has a special transformation stored, which transforms the main graph's coordinates to device coordinates for displaying the main graph on device. So, assuming $G1$ is the current main graph, when we want to draw $N8$ of Figure 4.2, we first need to transform the

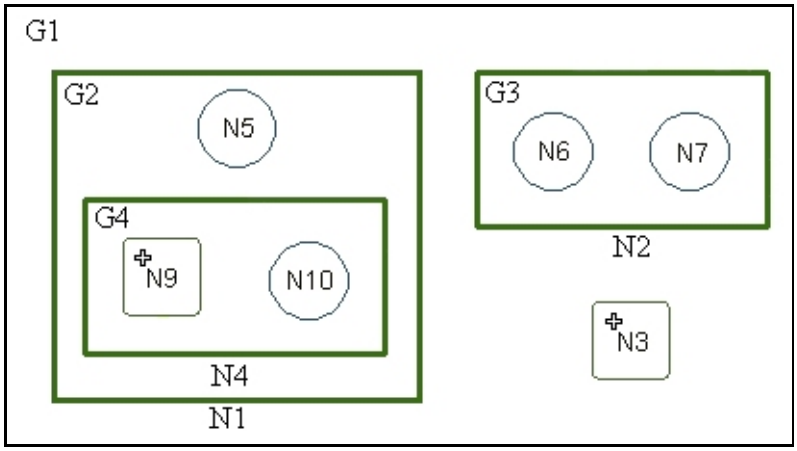

Figure 5.1: Drawing of a graph manager with multiple levels of nesting realized.

coordinates of $N8$ to the coordinate system of $G2$ using the transformation matrix stored in $G4$. Then we need to transform them to the coordinate system of $G1$ using the matrix of $G2$ and finally we use the device transformation matrix of the graph manager to transform the coordinates in $G1$ to device coordinates for rendering. This is much more efficient since changes in the nesting structure affects only the transformation per graph as opposed to the coordinates of each and every graph member.

The structure of the nesting relations are stored in a *nesting forest*. In a nesting forest, the nodes and edges represent nested graphs and expanded nodes in which these graphs are nested, respectively. Figure 5.1 shows a graph manager and Figure 5.2 shows the associated nesting forest. The plus sign on a node indicates that the node has a navigation link to another graph, but the child graph is not currently nested. Thus there is an edge between the node and the graph in the navigation forest, but not one in the nesting forest. Note that, a graph is represented in the nesting forest, iff it is nested in a node or one of the nodes of it nests another graph. So, initially we do not have a representing node for a graph in the nesting forest. Notice that edges may not be expanded; nevertheless they allow navigation through them.

Our framework supports editing a nested graph directly from an ancestor

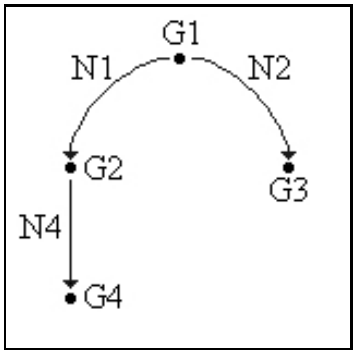

Figure 5.2: The corresponding nesting forest.

graph view without navigating to it through the links, a feature called *in-place editing*. This helps the user work on the "big picture".

5.3 Intergraph Edges

Maintenance and drawing of intergraph edges present a difficult issue since normally an edge's coordinates are stored in the coordinate system of its owner graph but intergraph edges are not directly a member of any graph. Since their end-nodes are placed in different graphs, they partly belong to different coordinate systems. Thus we store the bend points of an intergraph edge in the coordinate system of the lowest common ancestor graph of the intergraph edge as this is most prone to changes in coordinates (e.g., a change in the transformation of the owner graph of an end-node). For the graph manager in Figures 4.1 and 4.2, $I1$ is an intergraph edge and its lowest common ancestor graph is $G1$. This relation is shown in the navigation forest as an ownership relation, although the lowest common ancestor graph is not directly the owner of that intergraph edge.

5.4 Meta Edges

Not all graph elements that belong to a graph manager are viewable at all times. A graph is said to be *viewable* if it is immediately or deeply nested within the current main graph; *unviewable* otherwise. A graph element is viewable if it is owned by a viewable graph. Thus an intergraph edge will be unviewable unless both its end-nodes are viewable. Alternatively, an intergraph edge is said to be viewable if and only if the lowest common ancestor graph of the intergraph edge is a descendant of the main graph in the nesting forest. Additionally, an intergraph edge is said to be *reachable* if and only if the lowest common ancestor graph of the intergraph edge is connected to the main graph in the navigation forest of the graph manager. Note that when an intergraph edge is removed (might be a temporary removal such as one during a hide operation), the edge is also considered to be unreachable. If an intergraph edge is reachable but unviewable, then a meta edge is created between the two respective nodes in which the source and target of this intergraph edge is hidden (Figures 5.3 and 5.4). This way the user can still realize the relationship between the underlying objects.

One can clearly identify conditions under which an intergraph edge should be represented by a meta edge, and conditions under which a meta edge should be discarded to reveal the associated intergraph edge(s). Firstly, we can only visualize a reachable and viewable intergraph edge. In addition, it is impossible to have a viewable but unreachable intergraph edge. Table 5.1 summarizes all different states and transition conditions among those states for an intergraph edge (for the table, V, R, UV and UR stand for "viewable", "reachable", "unviewable" and "unreachable", respectively).

Actions defined in the table:

1. Do nothing.
2. A corresponding meta edge is created whenever a reachable and viewable intergraph edge becomes unviewable but stays reachable. An example case is when the graph that is the owner of one of the end nodes of the intergraph

Initial State	Ending State	Action Taken
R-V	R-V	(1)
	R-UV	(2)
	UR-UV	(1)
R-UV	R-V	(3)
	R-UV	(4)
	UR-UV	(5)
UR-UV	R-V	(1)
	R-UV	(6)
	UR-UV	(1)

Table 5.1: Conditions and transitions for an intergraph edge.

edge has been unnested under its parent member.

3. The intergraph edge now becomes viewable so we do not need to represent it via a meta edge anymore. If this intergraph edge is the only edge represented by the meta edge, then remove the meta edge and display the intergraph edge. Otherwise, display the intergraph edge as well as the meta edge. An example case is reversing what is done in (2), so nesting the owner graph once more to reveal the details under it, such as the intergraph edge.

4. The intergraph edge was already being represented by a meta edge, but now we may need a new meta edge to represent the intergraph edge. So, if needed, create a new meta edge for the intergraph edge, otherwise, the old meta edge keeps representing the intergraph edge. Assume A nests B and C, B nests D, and C nests E, and we have a node $u \in D$ and a node $v \in E$ and an edge $e = (u, v)$. If we unnest E, the edge e is replaced by a meta edge m_1 between u and the parent member of E. If we then unnest C, now the meta edge m_1 is replaced with a new meta edge m_2 between u and the parent member of C. This is a simple case of this transition.

5. The intergraph edge was already being represented with a meta edge, but now the intergraph edge is out of the topology. If the meta edge was representing only this intergraph edge, then discard the meta edge, otherwise, keep the meta edge. An easy example of this transition is the deletion of

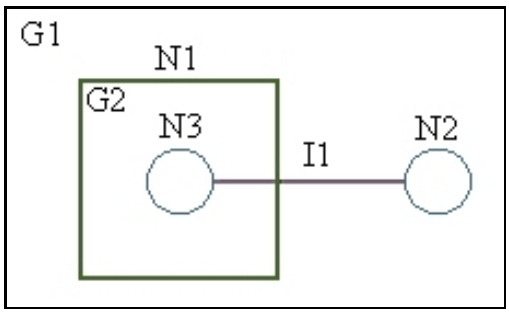

Figure 5.3: A simple intergraph edge between $N2$ of $G1$ and $N3$ of $G2$.

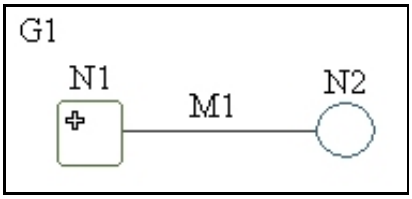

Figure 5.4: When $G2$ is unnested within $N1$, the intergraph edge $I1$ gets replaced with a newly created meta edge $M1$.

one of the end nodes of the intergraph edge. Thus the intergraph edge becomes unreachable and the representing meta edge of this intergraph edge may need to be discarded if it was only holding this intergraph edge.

6. The intergraph edge was inserted (back) to the topology, but it is still unviewable by default, so we create a meta edge for it. Undo of the operation defined in case (5) is a good example for this transition.

For efficiency reasons, a meta edge is created only when it will be viewable. Furthermore, a single (compressed) meta edge may represent multiple edges, helping in the reduction of the complexity of the manager. This is especially useful when we have a significantly smaller nesting forest compared to the size of the navigation forest.

Chapter 6

Complexity Management Operations

In the preceding chapters, the abstract structure of our framework and the way it handles multiple associated graphs with nesting relations have been described. Details on visualization of a graph manager have been given and an emphasized discussion on meta edge creation has been made. In this chapter, we will detail out the complexity management operations built on this structure. Section 6.1 will describe ways of hiding and revealing information using the expand and collapse mechanisms of our framework. Section 6.2 will explain another complexity management technique for creating folders and groupings inside the graph manager. Section 6.3 will discuss three more techniques, namely invisibility, hiding and ghosting. Section 6.4 will talk about the algorithms for each operation defined in this chapter and detailed complexity analysis for each algorithm. We will come to an end for this chapter by discussing on experimental analysis results in Section 6.5.

6.1 Expand/Collapse

An *expand* operation is applied to a node which has a navigation link to a child graph, to form a nesting relation between the node and the graph. A *collapse* operation is applied to a node which has a nesting relation with a graph, to undo the nesting relation. Note that collapsing a node does not break its navigation link.

Most applications require multiple levels of abstraction, where the user would like to visualize the information with varying levels of abstraction for different parts of the drawing. In order to structurally keep those different levels away from each other we use the multiple graph handling structure, the main graph corresponding to the highest level of abstraction. The visual support for reduction in complexity is realized through the expand and collapse operations. At any time during visualization, we may want to view a single portion of the whole manager. If this portion defines a tree in the skeleton of our graph manager, then we can set our main graph as the root of this tree and view only that part. When the parts we want to view are describing a forest instead, we are to do a selective collapse operation where the nodes we should collapse are from different trees of the navigation forest. Via the collapse operation, we can hide the nested graph of a node (avoiding inclusion) and draw it as a normal node. This way, we can collapse all nodes with nesting relations to hide unnecessary information embedded into them and expand all nodes with nesting relations to graphs which we want to display. In a graph manager with many graphs, expanding only a selected set of nodes may drastically reduce the amount of graph elements to be displayed and improve performance.

In Figure 5.3, $N1$ is expanded and collapsing $N1$ results in the drawing in Figure 5.4. The nesting relation is broken between $N1$ and $G2$, thus $G2$ and all underlying graph structures are no more visible. The navigation link is still there indicated with a plus sign on $N1$.

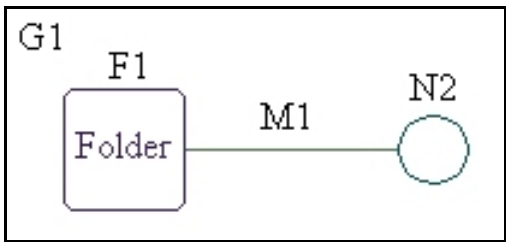

Figure 6.1: Folding $N1$ and all underlying structures.

6.2 Folding/Grouping

A *fold* operation is applied to a group of graph members, and results in a new (folder) node and its new child graph with these members. The folder node is created in collapsed state and may subsequently be expanded to nest this child graph with folded graph contents. At any time, an *unfold* operation may be applied on a folder node to reverse the effects of the fold operation.

The most common way of creating a child graph is to first create the graph and then fill it with required nodes and edges. However, it may be the case that the members are in a different graph and we want to transfer them to a new graph with a link from their current owner graph. This is an important means for decreasing complexity since you can temporarily avoid displaying many unnecessary members of a graph this way.

Often times members of a graph need to be put together according to some criteria to emphasize certain *grouping*. This can be achieved through folding followed by an expand operation, enabling all the group members to be gathered in the newly created child graph.

Currently the framework does not support overlapping groups, since a node may not exist in two or more different graphs' topologies. However, during the implementation of the framework, multiple views for a single node may be allowed. Although, this will not let us draw overlapping groups, a node would be able to have multiple views in multiple folders at the same time.

Figure 6.1 shows $N1$ and all underlying structures of the graph in Figure 5.3 folded. This results in a folder node $F1$ and a meta edge $M1$, since the intergraph edge $I1$ is no more visible after the folding. A folder node is in fact a normal node with a navigation link, but one may prefer to display them with a different user interface to explicitly show it is a folder node.

6.3 Invisibility/Hiding/Ghosting

A graph member is said to be *invisible* when it is not rendered on the display yet it is part of the graph topology. *Hiding*, on the other hand, is used to avoid any means of user interaction on a set of graph members, and temporarily removes graph contents from the graph topology as well. When a member of a graph is hidden, it is removed from its owner graph and placed in a special graph called the *hide graph*. Each graph in the graph manager has a hide graph associated with it. Any set of graph members may later be *unhidden* reversing the hide operation's effects. These members are removed from the hide graph and transferred back to the original owner graph. So, one can think of a hide graph as a special folder to hold the hidden graph elements, where the folder node is neither included in the graph topology nor has a visual representation.

Ghosting can be used to visually decrease the importance of a graph member by means of changing its color and/or brightness of its skin, and sending it to the background, "behind" other members. Unlike hiding, ghosting only tries to lose focus on the ghosted member but the member is still there both visually and topologically. You can still interact with that member and apply operations on it.

All three techniques help decrease the visual complexity of the graph manager and hiding additionally decreases the topological complexity at the cost of leaving hidden contents out of the reach of any operations on the owner graph.

Hide and ghost operations are demonstrated in Figures 6.2 and 6.3, using the graph manager in Figure 5.3. In Figure 6.2, we are hiding $N2$ and $I1$ is being

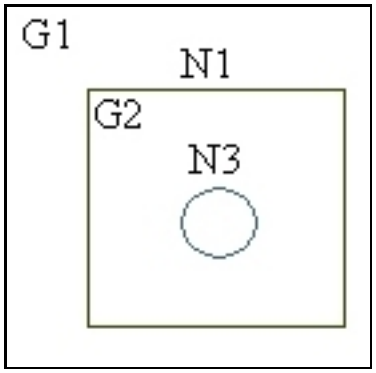

Figure 6.2: Hiding $N2$.

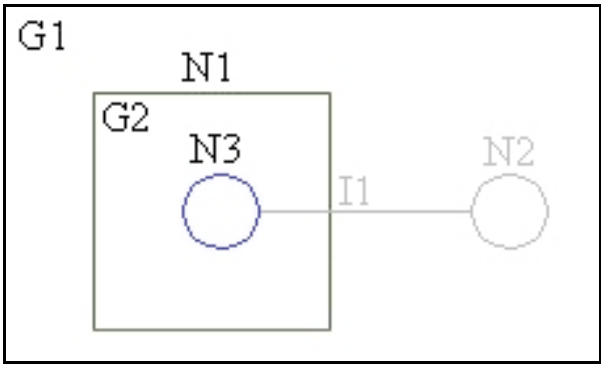

Figure 6.3: Ghosting $N2$.

hidden with it. Any incoming or outgoing edge or intergraph edge is also hidden when hiding a node. Note that, no meta edge or any other representing body is created for $I1$, it is out of the graph topology temporarily. In Figure 6.3, we are ghosting $N2$ which in turn ghosts $I1$. Both the node and the edge are still in the graph topology and viewable by the user, however their visual attractiveness has been decreased to let the user focus on other components of the graph.

6.4 Computational Complexity Analysis

It is crucial that such operations are efficient enough to be used as part of an interactive graph drawing and editing tool. Our framework allows one to efficiently implement the complexity management operations described previously. With the help of the nesting forest, basic operations such as finding a graph which is deeply nested into a node takes time linear in the order of the number of graphs in the graph manager. Similarly, building a list of graphs or nodes deeply nested into a node is linear in the number of graphs.

One of the basic operations is to find the *least viewable ancestor* of a node, which is done in $O(d_{NF})$ time, where d_{NF} is the depth of the tree of the main graph in the navigation forest. We simply navigate over the navigation tree, starting from the node under consideration to the root of the tree, until we find a viewable ancestor.

Finding the least common ancestor of two nodes (or an edge) is again a basic operation with $O(d_{NF})$ time complexity as follows, where both steps have time complexity of $O(d_{NF})$:

algorithm LEASTCOMMONANCESTOR($node_1, node_2$)
(1) Mark all ancestors of $node_1$ as traversed
(2) Iterate the ancestors of $node_2$ to find a previously marked ancestor

Node insertion is straightforward and can be performed in constant time complexity. However, edge insertion is a little bit tricky. If the edge is a normal edge, then the insertion again takes constant time. If the edge to be inserted is an intergraph edge, on the other hand, then $O(d_{NF})$ time is required as described below:

algorithm INSERTINTERGRAPHEDGE($node_1, node_2$)
(1) **if** $node_1$ and $node_2$ are both viewable **then**
(2) Insert edge as a normal intergraph edge
(3) **else**
(4) v_1 = FINDVIEWABLEANCESTOR($node_1$)

(5) v_2 = FindViewableAncestor($node_2$)
(6) Create a meta edge between v_1 and v_2
(7) **endif**

Step (2) and (6) are of $O(1)$ time, and Steps (4) and (5) are of $O(d_{NF})$ time, resulting in an overall time complexity of $O(d_{NF})$.

Edge removal is favorably easier than insertion assuming each edge keeps a reference to its meta edge. However, this brings extra space requirement and may be omitted to give up speed and reduce space. The algorithm is as follows:

algorithm RemoveIntergraphEdge(*intergraphEdge*)
(1) Remove edge as a normal intergraph edge
(2) **if** *intergraphEdge* was not viewable before removal **then**
(3) **if** meta edge of the *intergraphEdge* holds only this edge
(4) Discard meta edge
(5) **else**
(6) Remove *intergraphEdge* from the associated edge list of the meta edge
(7) **endif**
(8) **endif**

Steps (1), (4) and (6) require $O(1)$ time to complete. Thus the overall time complexity of the algorithm is $O(1)$.

Collapse operation is one of the more sophisticated operations:

algorithm Collapse(*node*)
(1) Mark all graphs to be affected from the operation
(2) Build affected intergraph edges list
(3) Create/assign meta edges for all affected intergraph edges

Step (1) is handled in $O(g)$ time, where g denotes the number of graphs in the graph manager, by simply navigating over the nesting forest. Step (2) can be finished in $O(m_{IG})$ time, by iterating over the intergraph and checking whether

the owners of both end nodes of all intergraph edges are marked, where m_{IG} denotes the number of edges in the intergraph. Step (3) has $O(m_{IG} \cdot d_{NF})$ time complexity. Thus the overall time complexity is $O(g + m_{IG} \cdot d_{NF})$. Assuming g is much smaller than $m_{IG} \cdot d_{NF}$ we can conclude that the overall time complexity of the operation is $O(m_{IG} \cdot d_{NF})$.

Expand operation is defined as follows:

algorithm EXPAND(*node*)

(1) Build a list of affected meta edges
(2) Discard affected meta edges
(3) Insert revealed intergraph edges

Step (1) may be handled in $O(m_{IG})$ time. However, using more space, we can decrease the time complexity if we keep a meta edge list for each collapsed node. This lowers the time complexity to the order of the number of meta edges connected to the expanded node, m_{meta}, which is in the worst case equal to m_{IG}, but on the average far smaller than m_{IG}. Step (2) is trivial and requires $O(m_{meta})$ time. Step (3) is normal edge insertion which is $O(d_{NF})$ per single edge. Since we have m_{meta} edges the overall complexity is $O(m_{meta} \cdot d_{NF})$.

Fold is another operation defined as follows:

algorithm FOLD(*nodeList*)

(1) Transfer all nodes in the *nodeList* to newly created child graph of folder node
(2) Build a list of all affected edges using *nodeList*
(3) **foreach** edge in the affected edges list **do**
(4) Remove edge from its old owner graph
(5) Insert edge to its new owner graph
(6) **endfor**

Step (1) consumes $O(n_{GM})$ time, where n_{GM} is the total number of nodes in the graph manager. Step (2) is trivially $O(m_{GM} + m_{IG})$ by iterating over *nodeList* and building a combined edge list from their connected edges. Step (3) iterates

$m_{GM} + m_{IG}$ times in the worst case, Step (4) is of $O(1)$, and Step (5) has $O(d_{NF})$ time complexity. Thus the overall time complexity of the algorithm is $O(n_{GM} + (m_{GM} + m_{IG}) \cdot d_{NF})$. Assuming m_{GM} is larger than both n_{GM} and m_{IG}, we can simplify the time complexity as $O(m_{GM} \cdot d_{NF})$.

Unfold has almost the same algorithm with fold, except *nodeList* is the set of all nodes of the child graph of the unfolded node, and the transfer occurs from the child graph to the owner graph of the unfolded node. Simplified time complexity is again $O(m \cdot d_{NF})$.

Other supplementary operations can also be implemented efficiently using our framework. For instance, the average time complexity of hit testing of objects in a graph is $O(d_{NT} \cdot n)$ where d_{NT} is the depth of the nesting tree and n is the number of objects in the graph. Assuming graph objects are uniformly distributed to the graphs and nesting forest is a balanced tree with constant non-leaf vertex degrees, the average complexity turns out to be $O((n_{GM} + m_{GM})/g \cdot \lg g)$, where n_{GM}, m_{GM}, and g stand for the total number of nodes, edges, and graphs in the graph manager, respectively.

6.5 Experimental Analysis

In this section we will discuss some experimental results obtained by executing the complexity management operations on different graph manager data. The operations to be examined are *collapse*, *expand* and *fold*. Each test is executed ten times and the average time is used as the result. The graph managers used in the tests are created randomly and uniformly to avoid time complexity differences that may arise from different graph manager topologies. All graph managers have a binary navigation tree, where the number of graphs in the graph manager is equal to $2^{d_{NF}} - 1$. Since the graph manager is uniform, the total number of nodes and edges is divided by the number of graphs in the graph manager and that many nodes and edges are assigned to each graph. Also intergraph edges are distributed uniformly among graphs, such that any two arbitrary graphs in the

graph manager have almost equal number of intergraph edges connected to their nodes. The computer used for experiments has a Pentium III 733MHz CPU with a 256MB memory.

The implementation used for experimentation is within Tom Sawyer Software's Graph Editor Toolkit for JAVA. This implementation does not keep a meta edge list specific to each node, so the expand operation will run in $O(m_{IG} \cdot d_{NF})$ instead of proposed $O(m_{meta} \cdot d_{NF})$ time complexity. Also the implementation creates the meta edges whenever the represented intergraph edges are to be drawn instead of creating them whenever the intergraph edge becomes unviewable. This allows inserting an intergraph edge in $O(1)$ time and eliminates the d_{NF} multiplier for all complexity management operations, but burdens the rendering of intergraph edges.

The first operation to be examined is *collapse*. The base graph manager data for each test has 1000 nodes, 2000 edges and 100 intergraph edges. All graph managers have a navigation tree depth of 3, which means there are exactly 7 graphs in each.

Figure 6.4 displays the effects of changing n_{GM}. From the table it is seen that n_{GM} has no effect on the running time of the *collapse* operation.

Figure 6.5 displays the effects of changing m_{GM}. The table shows that m_{GM} has no effect on the running time of the *collapse* operation, although a small effect due to indirect implementation specific details has been observed.

Figure 6.6 displays the effects of changing m_{IG}. The table shows that m_{IG} has a linear contribution to the running time of the *collapse* operation.

The second operation to be examined is *expand*. The base graph manager data for each test has 1000 nodes, 2000 edges and 100 intergraph edges. All graph managers have a navigation tree depth of 3, which means there are exactly 7 graphs in each.

Figure 6.7 displays the effects of changing n_{GM}. From the table it is seen that n_{GM} has no effect on the running time of the *expand* operation.

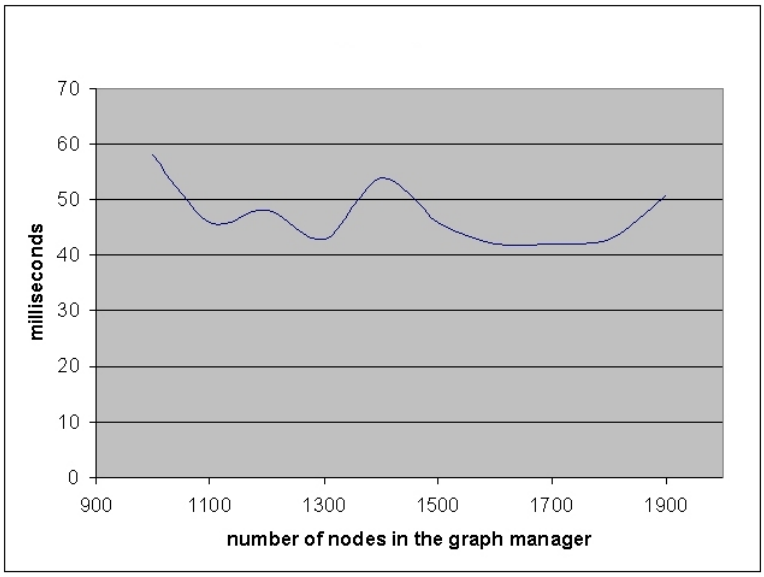

Figure 6.4: Affect of n_{GM} on *collapse*. ($m_{GM} = 2000, m_{IG} = 100$)

Figure 6.8 displays the effects of changing m_{GM}. The table shows that m_{GM} has no effect on the running time of the *expand* operation.

Figure 6.9 displays the effects of changing m_{IG}. The table shows that m_{IG} has a linear contribution to the running time of the *expand* operation.

The last operation to be examined is *fold*. The base graph manager data for each test has 1000 nodes, 2000 edges and 100 intergraph edges. All graph managers have a navigation tree depth of 3, which means there are exactly 7 graphs in each.

Figure 6.10 displays the effects of changing n_{GM}. From the table it is seen that n_{GM} has a linear contribution to the running time of the *fold* operation.

Figure 6.11 displays the effects of changing m_{GM}. The table shows that m_{GM} has a linear contribution to the running time of the *fold* operation.

Figure 6.12 displays the effects of changing m_{IG}. The table shows that m_{IG} has a linear contribution to the running time of the *fold* operation.

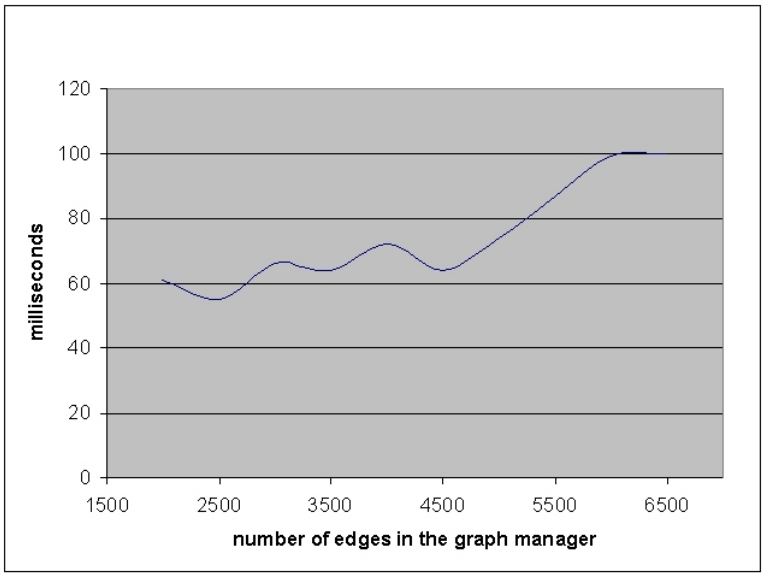

Figure 6.5: Affect of m_{GM} on *collapse*. $(n_{GM} = 1000, m_{IG} = 100)$

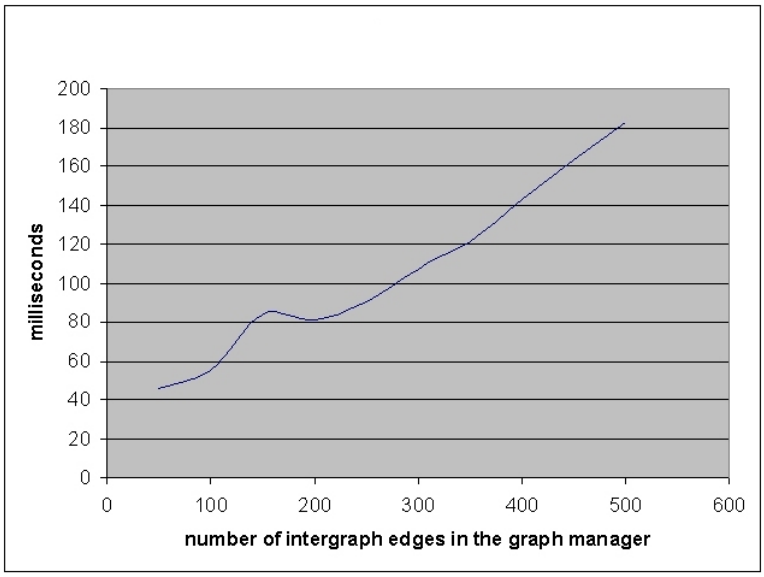

Figure 6.6: Affect of m_{IG} on *collapse*. $(n_{GM} = 1000, m_{GM} = 2000)$

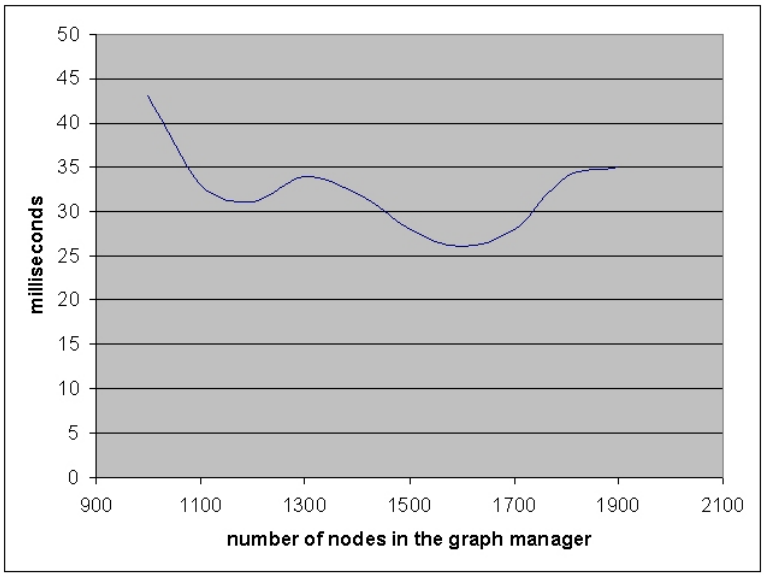

Figure 6.7: Affect of n_{GM} on *expand*. $(m_{GM} = 2000, m_{IG} = 100)$

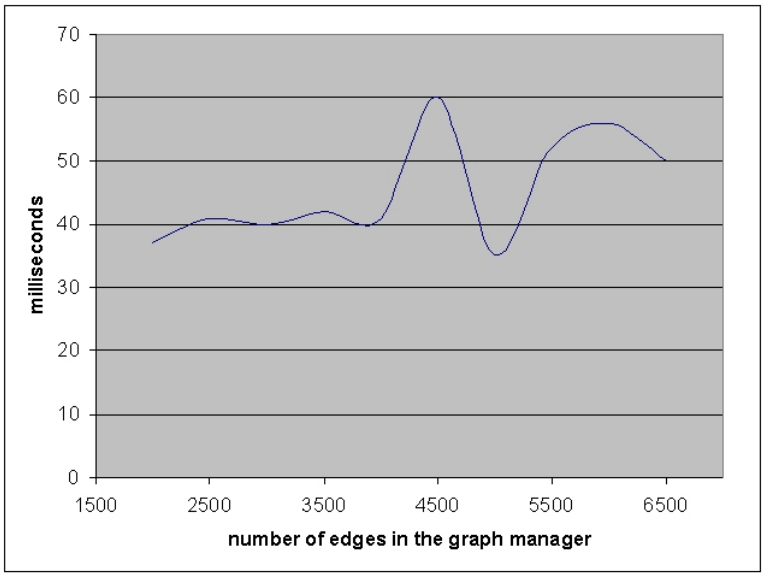

Figure 6.8: Affect of m_{GM} on *expand*. $(n_{GM} = 1000, m_{IG} = 100)$

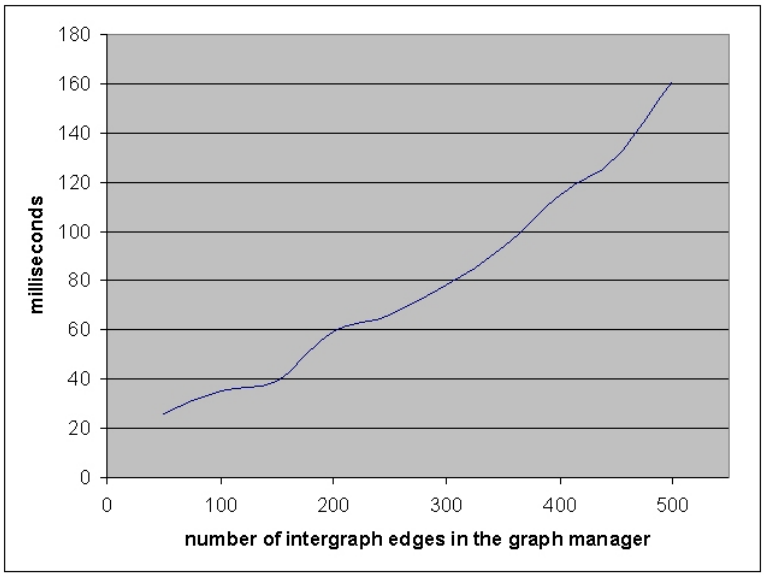

Figure 6.9: Affect of m_{IG} on *expand*. $(n_{GM} = 1000, m_{GM} = 2000)$

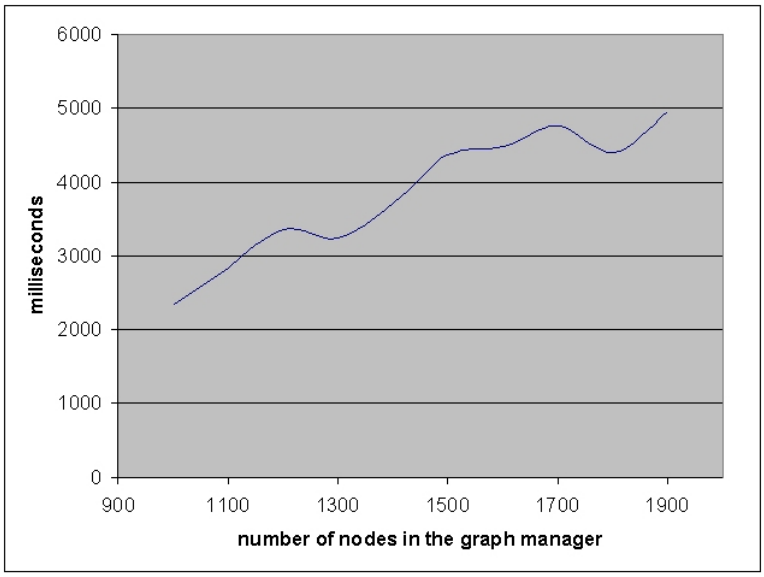

Figure 6.10: Affect of n_{GM} on *fold*. $(m_{GM} = 2000, m_{IG} = 100)$

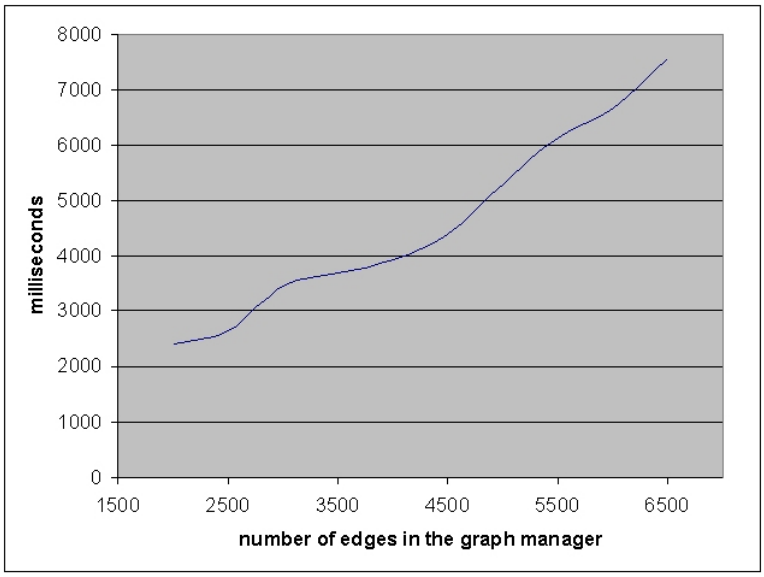

Figure 6.11: Affect of m_{GM} on *fold*. ($n_{GM} = 1000, m_{IG} = 100$)

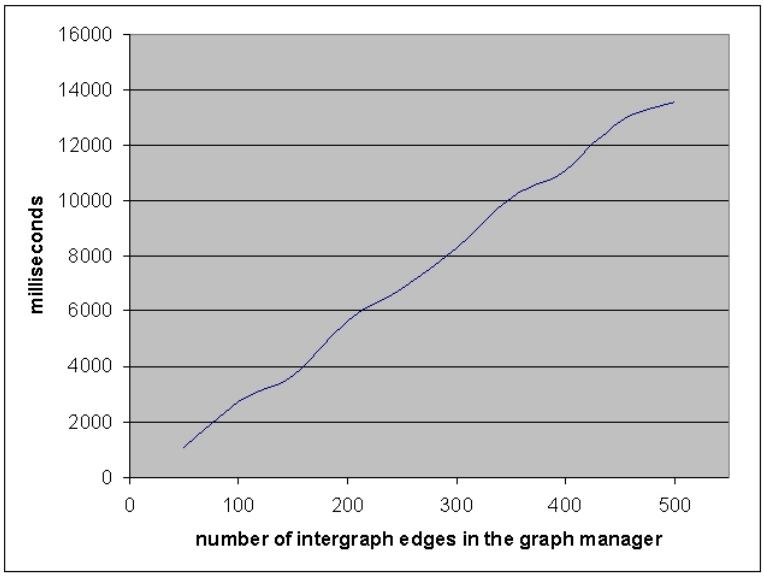

Figure 6.12: Affect of m_{IG} on *fold*. ($n_{GM} = 1000, m_{GM} = 2000$)

Chapter 7

Class Structures

The previous chapters introduced the graph manager concept, described challenges and solutions of drawing graph managers and explained ways of integrating complexity management techniques on top of the proposed structure. Now, we are going to give a brief class diagram that will reveal the basic structures behind a successful implementation of the framework. The class diagram discussed in this chapter is not a complete one, but should be treated as a starting design for an implementation.

Figure 7.1 is a class diagram summarizing the framework architecture. In this diagram, only the major inheritance and aggregation relations have been shown. Similarly, only the significant data and functionality of each class have been included. The underlying classes might be classified into two: abstract level graph manager and its components, and the corresponding drawing level classes.

As seen from the figure, *GraphManager*, *Graph*, *GraphMember*, *Node* and *Edge* classes are in the abstract layer, where we do not have any information related with drawing. Class *Node* and class *Edge* inherits from class *GraphMember*, which makes both of them available for child graph handling. A *GraphManager* object is capable of inserting and removing *Graph* objects. Additionally, it may insert and remove *Edge* objects. The reason for that is the intergraph edges. Class *GraphManager* acts like a filter for edge insertion and decides whether a

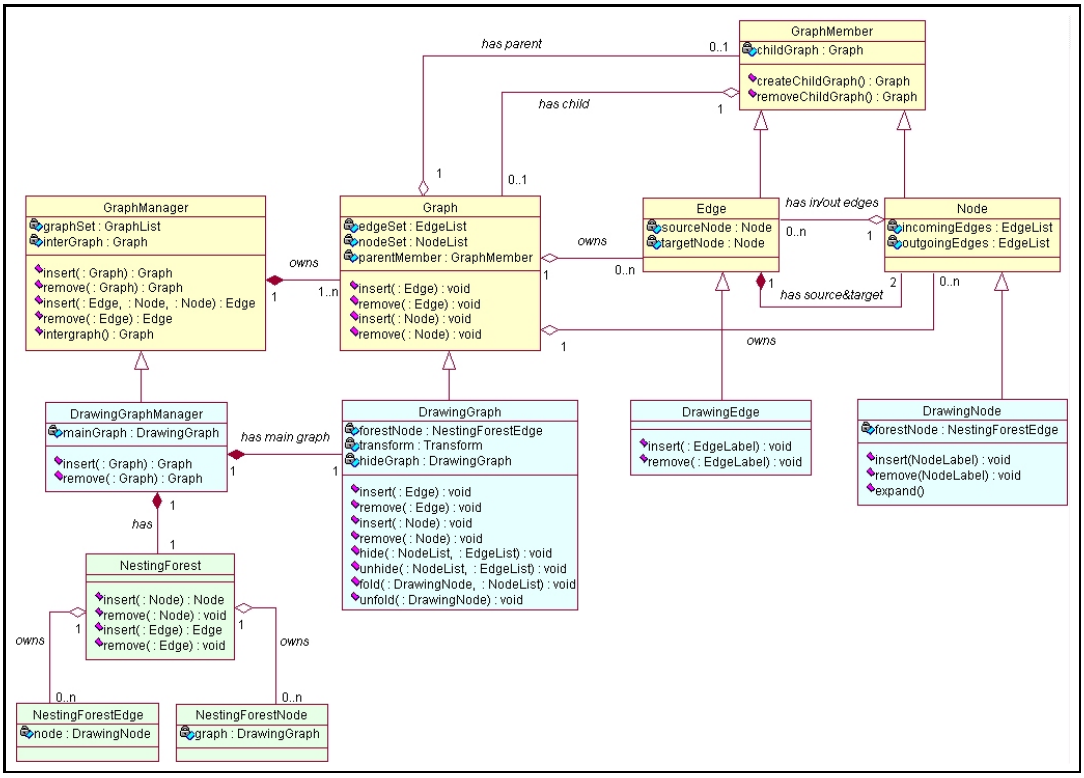

Figure 7.1: Class diagram summarizing the framework architecture.

new edge is a normal edge that should be inserted by the owner graph of the source and target nodes, or it is an intergraph edge which should be inserted by the intergraph.

Class *Graph* is capable of inserting and removing nodes and edges to its topology. Its topology consists of an edge list and a node list. It also keeps a reference for the parent member, if it is the child graph of a member.

Class *GraphMember* is a basis for the navigation link operations and is the ancestor of class *Node* and class *Edge* classes. An *Edge* object keeps references for its source and target nodes and a *Node* object knows its incoming and outgoing edges.

The drawing layer is built on the abstract layer and has the additional properties related with drawing. Class *DrawingGraphManager* has the additional

mainGraph reference, which was described in Section 5.1. It also has a *NestingForest* to keep track of the nesting relations. The nesting forest is in fact an extended structure built on the *Graph*, *Node* and *Edge* classes.

Class *DrawingGraph* has additional properties like a transformation matrix and the unique hide-graph. A *DrawingGraph* object supports folding and hiding related operations.

Class *DrawingEdge* is very similar to class *Edge* and have nothing special added on top of what class *Edge* has. However, class *DrawingNode* is now capable of visualizing a nesting relation via inclusion, and thus has the additional expand and collapse methods.

Chapter 8

Implementation

This chapter will show an example implementation of the framework. Tom Sawyer Software is an industry leading company in graph drawing and editing tools with their product series GLT and GET [14] for various platforms. Our framework has been taken as the architecture basis for GET for Java, version 5, and has been successfully implemented. The complexity management operations have proven to work efficiently in terms of both speed and stability.

Figures 8.1, through 8.5 show the complexity management operations in action within GET for Java, version 5. Figure 8.1 shows a simple network drawing where there are four routers, serving other network devices (PCs). In Figure 8.2, we have grouped all routers and the PCs connected to them together and attained a much clearer visualization of the overall network. We can expand only the folders corresponding to two routers as in Figure 8.3. In order to exclude a device which is broken, we can hide it as in Figure 8.4. To input varying levels of details of the main server, deep nesting of graphs may be used as in Figure 8.5.

A real life application will typically be of much higher complexity than this simple network. For instance, a call graph, or the local area network of a university campus usually has hundreds if not thousands of nodes and edges. In a graph of that kind, benefits of complexity management techniques become much clearer.

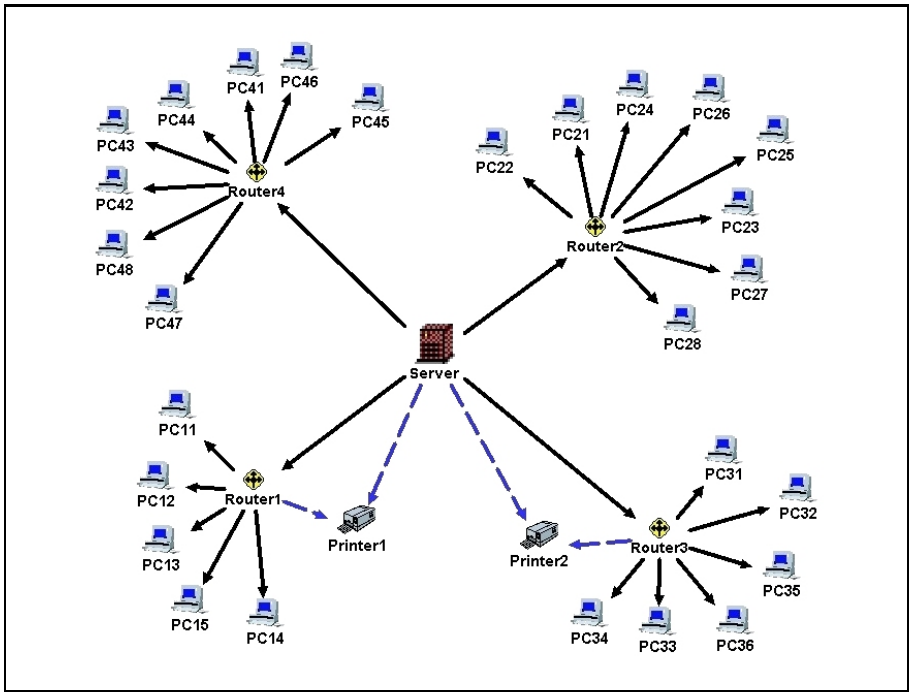

Figure 8.1: Map of a small network drawing in GET for Java.

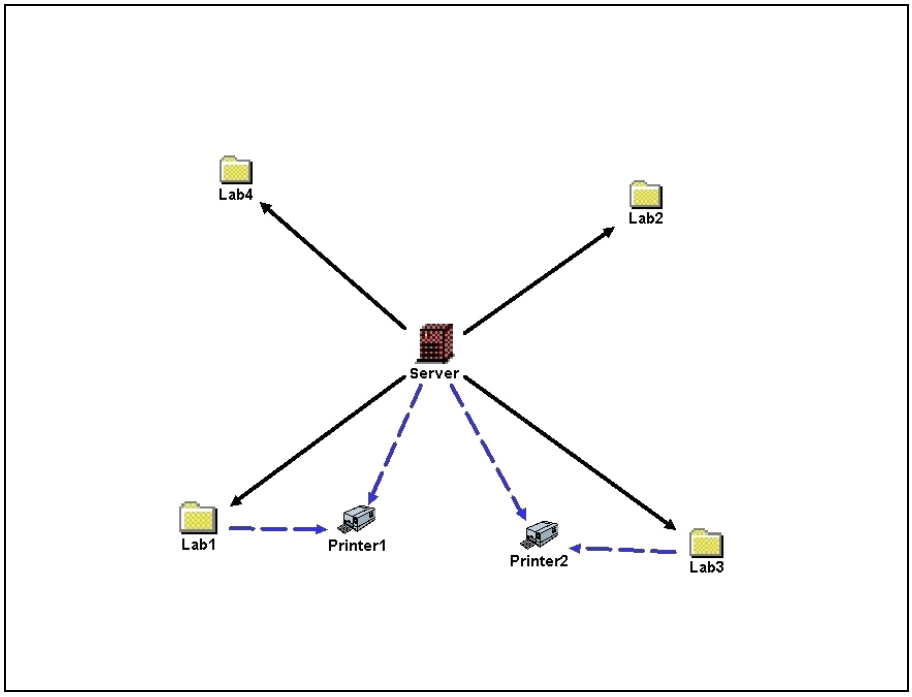

Figure 8.2: All related network devices are grouped together under a folder.

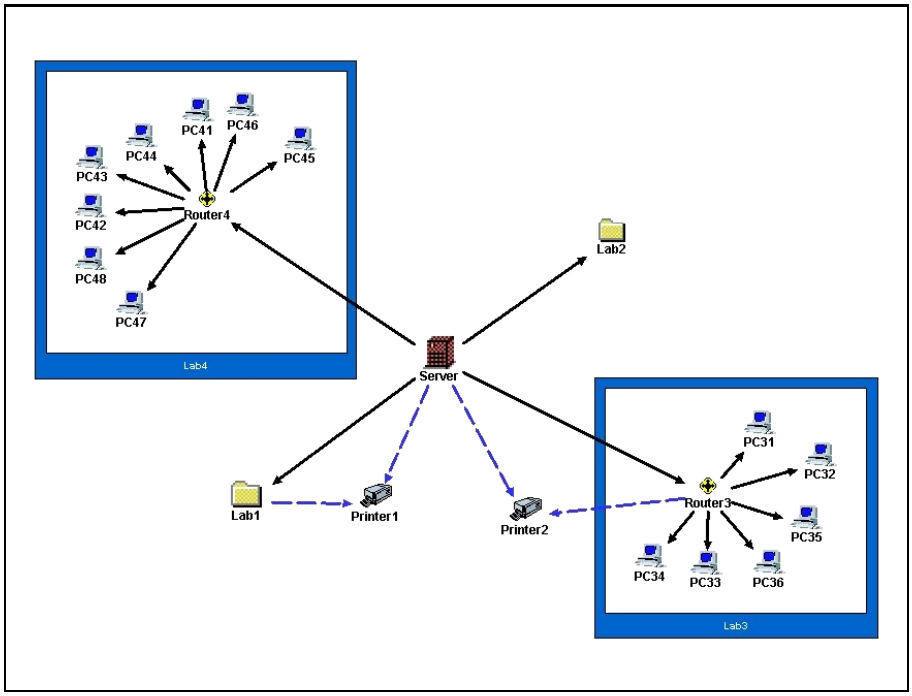

Figure 8.3: Two of the groups have been expanded to reveal details.

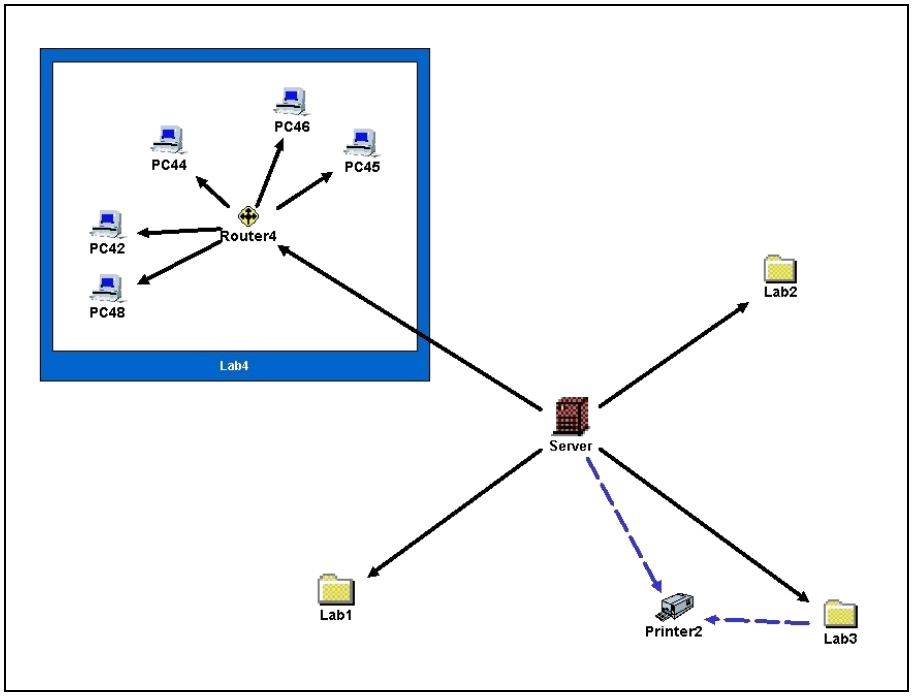

Figure 8.4: Printer1, PC41, PC43 and PC47 are unavailable, so they are hidden.

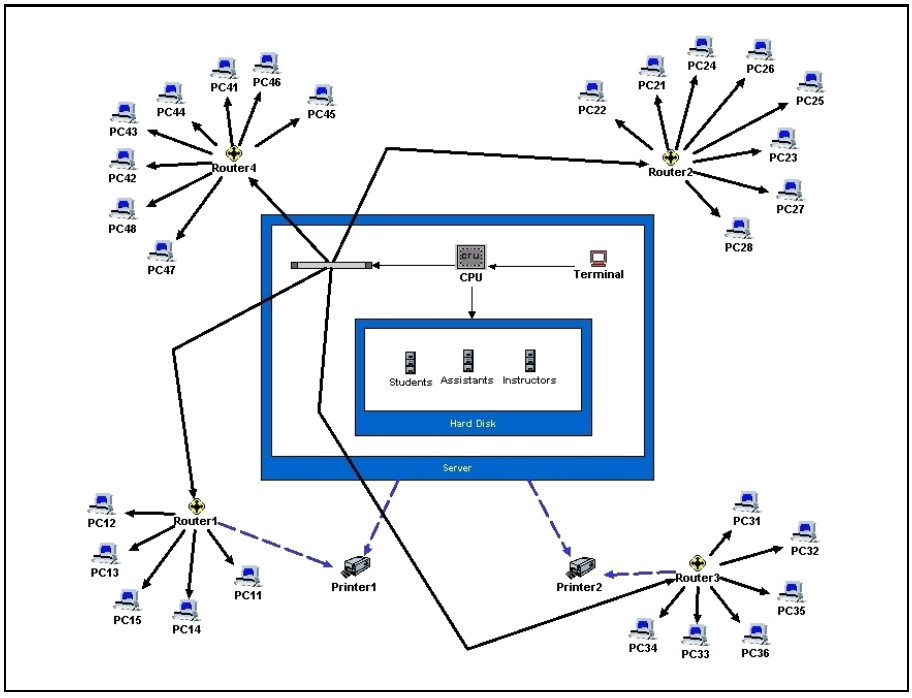

Figure 8.5: The varying levels of the details of the server has been input as deeply nested graphs.

Chapter 9

Conclusion

In this thesis the concept of a *graph manager* is introduced, which is built on the idea of a compound graph, but is extended to be able to include multiple associated graphs and intergraph edges. A normal compound graph can only handle a single graph with all the members and child-parent hierarchies embedded into it. This is an acceptable way of managing the relation as long as pure abstraction layers are not required. However, a graph manager supports multiple graphs which makes pure abstraction layers possible by dividing the graph manager's members among these graphs. In this way, the members are placed into different graphs and the abstraction layers are easily manipulated using these graphs. It is much easier to work on the partial data of a graph manager, since each graph in the graph manager has all the functionality of a normal graph.

Support for intergraph edges is another important advantage our framework supplies over a compound graph. A compound graph, as discussed in detail, has a unique graph for holding all members and hierarchy, thus any edge in the compound graph is a normal edge. An edge of the compound graph should be analyzed further to be exactly identified: is it a normal edge connecting two nodes in the same abstraction layer, is it an edge connecting two nodes in different abstraction layers or even is it a meta edge that represents some hidden edges. Introduction of intergraph edges into our framework allows clear identification of edge types and relations among different abstraction layers. This helps in easier

visualization of edges connecting nodes from different layers and easier operations defined over such edges.

The proposed framework clearly defines the transitions over intergraph edges and representative meta edges. It identifies all situations that an intergraph edge starts being represented by a meta edge and all situations a meta edge is discarded to reveal the associated original intergraph edge(s).

Many complexity management operations - such as expanding, collapsing, folding, grouping, hiding, ghosting and invisibility - that have been previously defined in the literature have been integrated into a single framework, which can be implemented and extended easily, by this study. The framework has been successfully integrated into a commercial graph visualization tool. Experimental analysis verify the efficient theoretical computational complexity of the framework as well.

In brief, we have described a comprehensive framework for development of complexity management techniques in graph visualization tools. The architecture supports management of multiple, associated graphs on which various complexity management operations such as navigation, folding, and nesting may be applied. The implementation as well as the theoretical analysis of this framework show that the involved data structures and algorithms are efficient enough to be used within an interactive graph drawing and editing tool.

Bibliography

[1] A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proceedings of the eleventh annual ACM-SIAM Symposium on Discrete Algorithms*, pages 566–575. ACM Press, 2000.

[2] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing, Algorithms for the Visualization of Graphs.* Prentice-Hall, 1999.

[3] R. Diestel. *Graph Theory.* Springer-Verlag, New York, 2000.

[4] U. Dogrusoz, Q. Feng, B. Madden, M. Doorley, and A. Frick. Graph visualization toolkits. *IEEE Computer Graphics and Applications*, 22(1):30–37, January/February 2002.

[5] A. Formella and J. Keller. Generalized fisheye views of graphs. In F. J. Brandenburg, editor, *Proc. 3rd Int. Symp. Graph Drawing, GD*, pages 242–253, Berlin, Germany, 20–22 1995. Springer-Verlag.

[6] K. Fukuda and T. Takagi. Knowledge representation of signal transduction pathways. *Bioinformatics*, 17(9):829–837, 2001.

[7] I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[8] K. Sugiyama and K. Misue. A Generic Compound Graph Visualizer / Manipulator: D-ABDUCTOR. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 500–503. Springer-Verlag, 1995.

[9] I. A. Lisitsyn and V. N. Kasyanov. Higres - Visualization system for clustered graphs and graph algorithms. In J. Kratochvíl, editor, *Graph Drawing (Proc. GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 82–89. Springer-Verlag, 1999.

[10] G. Sander. Graph Layout Through the VCG Tool. In R. Tamassia and I. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 194–295. Springer-Verlag, 1995.

[11] M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. In P. Bauersfeld, J. Bennett, and G. Lynch, editors, *Human Factors in Computing Systems, CHI'92 Conference Proceedings: Striking A Balance*, pages 83–91. ACM Press, Mai 1992.

[12] M. Sarkar and M. H. Brown. Graphical fisheye views. *Communications of the ACM*, 37(12):73–84, 1994.

[13] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):876–892, 1991.

[14] Graph Layout Toolkit and Graph Editor Toolkit User's Guide and Reference Manual. Tom Sawyer Software, Oakland, CA, USA, 1992-2002. http://www.tomsawyer.com.