Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 30 (2006) 86–97
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Technical Section

A multi-graph approach to complexity management in interactive graph visualization

U. Dogrusoz[a,b,*], B. Genc[a,c]

[a]Computer Engineering Department, Bilkent University, Ankara 06800, Turkey
[b]Tom Sawyer Software, 1625 Clay Street, Oakland 94612, CA, USA
[c]Computer Science Department, University of Waterloo, Waterloo, ON, Canada

Abstract

In this paper we describe a new, multi-graph approach for development of a comprehensive set of complexity management techniques for interactive graph visualization tools. This framework facilitates efficient implementation of management of 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: Information visualization; Graph drawing; Complexity management; Compound graph; Software engineering

1. Introduction

Graphs are commonly used to model relational information that arises in numerous areas including Web analysis, relational databases, biochemical networks, telecommunication networks, financial analysis, software engineering and geographical studies. Elements 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.

Considerable amount of research in graph drawing [1,2] 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 (see Fig. 1 for examples of complex real-life graphs).

In this paper, we present a comprehensive framework for visualizing complex graphs with the help of a variety of techniques. 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).

The base structure of our framework is a graph manager, which is an extension of the well-known

*Corresponding author. Computer Engineering Department, Bilkent University, Ankara 06800, Turkey.
Tel.: +90 312 290 1612; fax: +90 312 266 4047.
E-mail address: ugur@cs.bilkent.edu.tr (U. Dogrusoz).

0097-8493/$ - see front matter
doi:10.1016/j.cag.2005.10.015

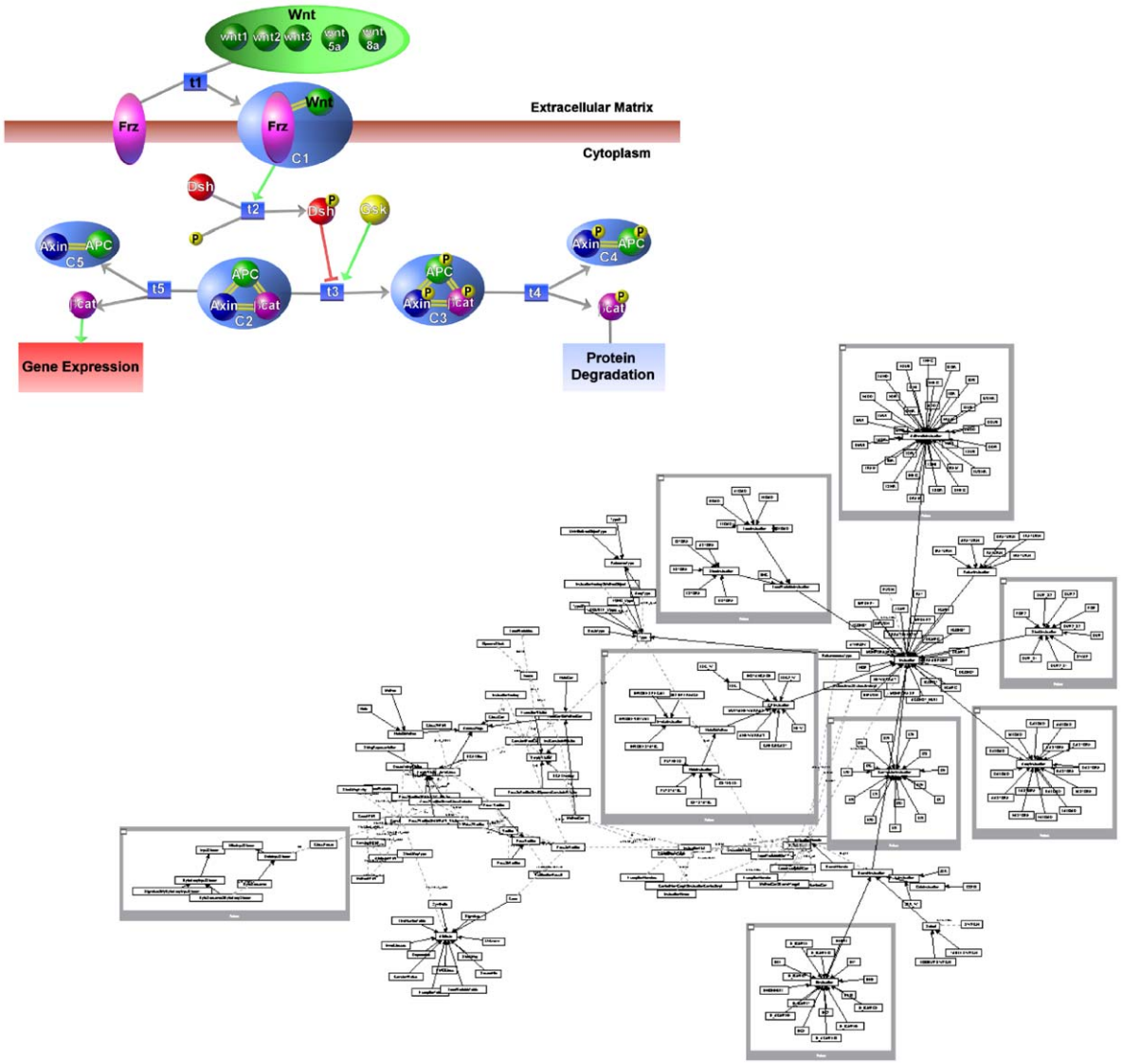

Fig. 1. Examples of complex real-life graphs with nesting and intergraph relations from biology and software modeling (courtesy of Tom Sawyer Software).

compound graphs [3,4]. A graph manager is defined as a collection of related simple graphs (i.e., graphs without any nesting or inclusion) and methods to apply changes on these graphs. Therefore, it represents both the underlying data structures and the operations defined on these structures. The most significant and novel features of this framework can be summarized as follows:

- It is based on a new structure, namely *graph manager*, which consists of multiple, possibly independent simple graphs. Multiple graphs allow us to define isolated abstraction levels.
- It is mainly designed for interactive use and dynamically changing data. The topology of a graph manager may be efficiently edited. The clear separation of abstraction levels increase the efficiency of interactive operations performed locally on a single graph.
- Most common complexity management operations (e.g., expanding and collapsing a node, folding a group of nodes, and hiding parts of a graph manager) can be efficiently implemented on top of the graph manager structure.
- Empirical results of an implementation of the system verifies that the framework satisfies industry standards for generality and efficiency.

2. Related work

A good deal of research has been conducted and results have been incorporated into various frameworks and tools to solve the complexity management problem for complex and/or large graphs. Some studies [5,6,3] describe how to extend graphs with a hierarchical structure. Some frameworks were designed to specifically create clusters based on a given data set [7,8]. HGV [9] is a framework with support for multiple views and hierarchies. Systems for efficient layout of compound graphs have also been proposed [10].

There are many techniques proposed for navigating and visualizing very large graphs. Sarkar and Brown describes the fisheye view [11,12] approach for complexity management in large graphs. In [13,14] authors explain techniques of expanding and collapsing certain nodes of a hierarchical graph representation to obtain different views of the compound graph, with extensive technical analysis of each operation using different data structures. A survey of techniques for graph visualization and navigation can be found in [15].

VCG [16] is a tool that uses the compound graph structure, with the ability to fold nodes and edges under certain cases and hide edges of specific types. D-ABDUCTOR [17] is another tool based on compound graphs [3]. It supports information hiding via expand and collapse operations. Higres [18], on the other hand, is a visualization system for clustered graphs and handles compound graphs. Huang and Eades describes DA-TU [19], another interactive system based on compound graphs, designed for clustering and navigating huge graphs.

A graph manager consists of multiple associated graphs whereas other tools based on compound graphs usually consist of a single flat graph with an inclusion tree built on it. This allows us to isolate abstraction levels from each other when needed. This way operations on a certain part of the topology might be handled with minimal interference to other parts.

Previous studies have been mostly based on visualization of static graphs. However, we propose an interactive framework where the user is capable of dynamically changing the topology of each graph independently or the graph manager as a whole as well as changing his/her viewpoint as desired. The supporting structures inside a graph manager facilitate much simpler implementations of complexity management operations.

Most previous frameworks focus on a certain limited set of complexity management operations. Some of them only offer visual methods like panning, zooming and fisheye views; others allow expanding and collapsing the nodes of the compound graph, while others propose hiding unwanted parts of the topology. Our framework offers mechanisms for efficiently implementing the most comprehensive complexity management tools.

3. Graph managers

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

A *graph manager* $M = (S, I, F)$ is a structure based on compound graphs, defined by a *graph set* $S = \{G_1, G_2, \ldots, G_l\}$, an *intergraph* I, and a *navigation forest* of rooted trees $F = (V^F, E^F) = T_1 \cup T_2 \cup \cdots \cup T_k$. 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 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 $M = (S, I, F)$ is via the intergraph edges, maintained by the intergraph I. Let $u \in V^{G_i}$ and $v \in V^{G_j}$ be two nodes where $G_i \neq G_j$ 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.

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 and a pictorial representation of the graph manager with this navigation forest are given in Figs. 2 and 3, respectively. The navigation forest provides a clean, dynamic way of navigating through the graph manager contents.

Graph managers provide a better representation and clean separation of abstraction levels. In a typical compound graph representation, all the nodes and edges are placed in a single flat graph structure, and abstraction is provided virtually via inclusion trees. A graph manager, on the other hand, allows placement of graph members into different graphs, thus permitting solid boundaries between members of each graph.

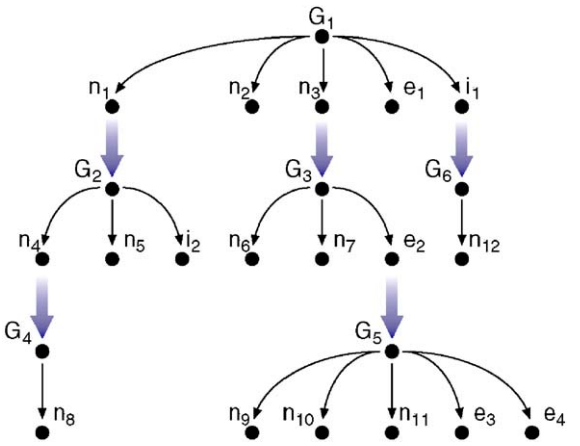

Fig. 2. The navigation forest of a graph manager, representing both ownership (solid) and navigation (gradient) links.

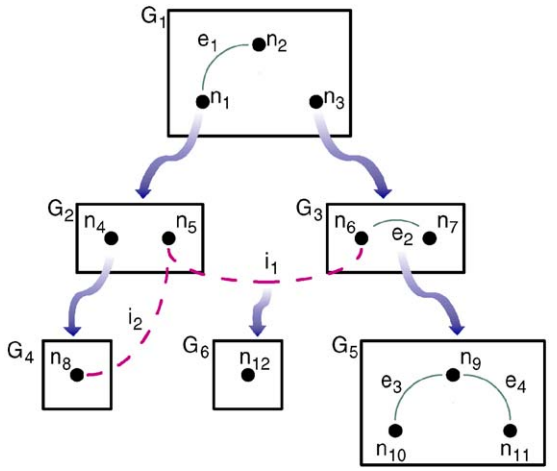

Fig. 3. A pictorial representation of the graph manager with the navigation forest in Fig. 2. The gradient arrows show the navigation links and the dashed edges are the intergraph edges.

Interactive graph editing operations are transparent to a graph manager. Whenever we add or remove a node or an edge to one of the graphs of a graph manager, only that graph is directly affected from this operation; others are not notified of this change. This allows simpler local manipulations of a graph.

Accessing the edge set of a single graph is much easier with this structure as each graph keeps its own list of nodes and edges. In a compound graph structure, access of edges in a certain abstraction can be obtained by first collecting a list of nodes belonging to that abstraction, and then traversing over all the edges to determine whether both its ends belong to this abstraction.

In our framework, the intergraph is a special graph which only stores intergraph edges (no nodes), and is not placed in the graph list of a graph manager. This facilitates traversal over intergraph edges in time, linear in the number of intergraph edges. A node incident with an intergraph edge stores the intergraph edge locally but in an incident edge list different from regular edges. Thus, we can selectively iterate over regular, intergraph, or both types of incident edges of a given node as desired, in time linear in the number of desired incident edges.

4. Drawing managed graphs

Drawing the structures inside a graph manager requires many additional structures on top of the ones described earlier. At the drawing layer every graph member has a geometry describing the location, dimension or routing of the member in its owner graph. Some elements like the node and edge labels, have their coordinates maintained relative to their owner graph members. Briefly the major design criteria are as follows:

- Ability to define flexible viewports from any graph of the graph manager.
- Capability to visualize inclusion relations among graphs through navigation or nesting.
- Support for editing any part of the graph manager which is currently visible to the user.
- Fast and efficient structures for local graph drawing operations like moving and resizing nodes inside the graph bounds.
- Ability to layout each graph with possibly different styles and parameters.
- Efficient visualization of intergraph edges.
- A representation for unviewable intergraph edges.

4.1. 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. The nodes and edges in the main graph are rendered 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.

4.2. 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 *optionally* realized in a drawing using nesting.

The structure of the nesting relations are stored in a *nesting forest*, where the nodes represent nested graphs and edges represent expanded nodes in which these graphs are nested. Fig. 4 shows a graph manager and 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. Notice that edges may not be expanded; nevertheless they allow navigation through them.

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 efficient handling of these transformations, a unique transformation matrix is maintained by each nested graph to transform its members' coordinates to the coordinates of its parent graph. This facilitates separate zoom and scroll levels for each graph. Fig. 5 shows a drawing with three nested graphs. A node in Graph A is expanded to provide a viewport for its child graph's (Graph B) contents. Because of its current zoom and scroll levels Graph B contents are only partially visible.

This design brings certain benefits during visualization of the graph objects. For instance, when the coordinates of a node with a child graph nested into it is updated, the coordinates of the child graph objects remain unchanged; it suffices to change the coefficients of the associated transformation matrix so that each child graph object reports the right transformed coordinates with respect to the main graph for operations such as rendering and hit-testing. On the other hand, when working within a single nested graph, local coordinates may be used without any transformation, forming an isolated workspace for operations such as layout. In addition, this separation of graphs and use of separate

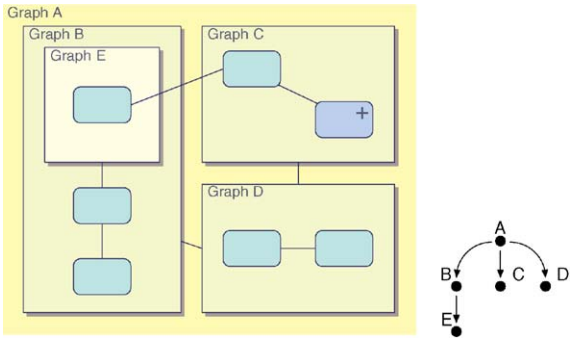

Fig. 4. *Left*: Drawing of a graph manager with multiple levels of nesting realized. *Right*: The corresponding nesting forest.

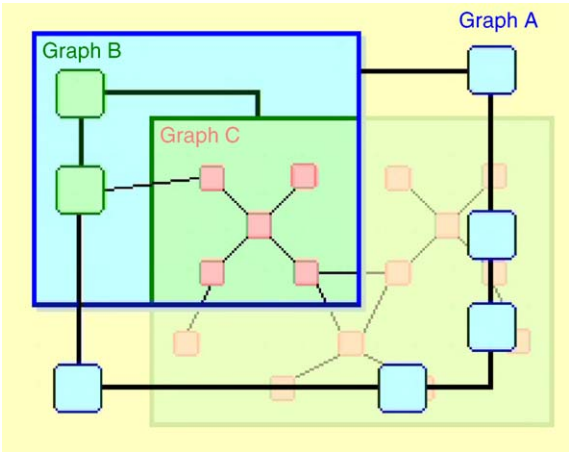

Fig. 5. Three graphs of a graph manager nested with varying scroll and zoom levels. Currently unviewable parts of the nested graph C is shown faded for clarity.

independent coordinate systems allows us to apply different layout techniques to each abstraction. For instance, a naive implementation may layout each graph at the leaves of the navigation forest independently in its own coordinate system. Then the parent nodes of each such graph may be treated as single (larger) nodes in their owner graphs, and can be laid out possibly in a different style, resulting in a bottom-up layout approach for multiply nested graphs. Of course the intergraph edges need to be routed in a post-processing step as well. An example may be found in Fig. 6.

Our framework also supports editing a nested graph directly without navigating to it through the links, a feature called *in-place editing*. This is simply done by first applying an *inverse* transformation (inverses of matrices used for drawing and hit-testing graph objects) from the input device coordinates to the local graph coordinates. This helps the user work on the "big picture" without having to focus on a limited part of a graph manager.

4.3. 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. 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 matrix of the owner graph of an end-node). 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.

4.4. Meta edges

Not all graph elements that belong to a graph manager will be 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 (Fig. 7). This way the user can still realize the relationship between the underlying objects.

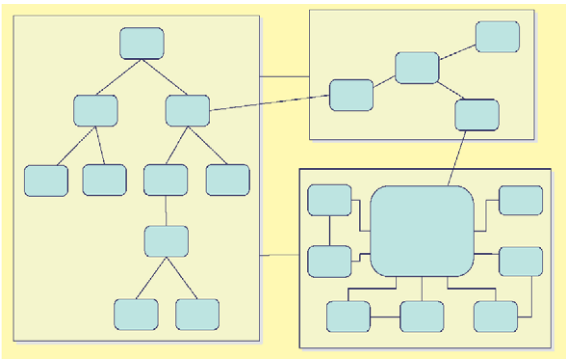

Fig. 6. Different layout techniques applied to each graph inside a graph manager.

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 1 summarizes all different states and transition conditions among those states for an intergraph edge; actions referred to in the table are as follows:

(1) Do nothing.
(2) A corresponding meta edge is created whenever a reachable and viewable intergraph edge becomes unviewable but stays reachable.
(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.
(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.

Table 1
Conditions and transitions for an intergraph edge; V, R, UV and UR stand for "viewable", "reachable", "unviewable" and "unreachable", respectively

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)

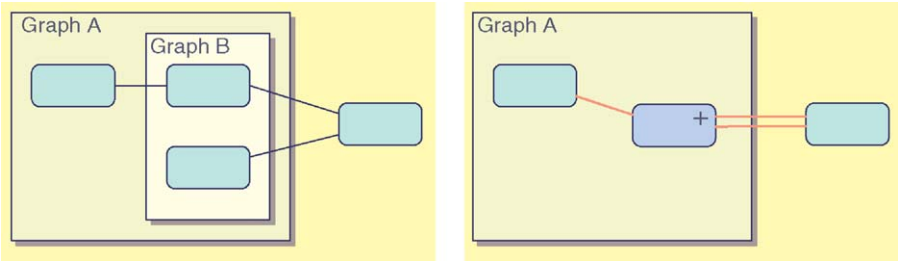

Fig. 7. *Left*: Graph B nested inside a node has incident intergraph edges. *Right*: These intergraph edges become unviewable and therefore not drawn when graph B is unnested. The meta edges (in red) represent the underlying intergraph relations.

(5) The intergraph edge was already being represented with a meta edge, but now the intergraph edge is removed. If the meta edge was representing only this intergraph edge, then discard the meta edge, otherwise, keep the meta edge.
(6) The intergraph edge was inserted back, but it is still unviewable by default, so we create a meta edge for it.

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 reduction of complexity. This is especially useful when we have a significantly smaller nesting forest compared to the size of the navigation forest.

5. Complexity management operations

In preceding sections, the structure of our framework and the way it handles multiple associated graphs with nesting relations have been described. This section will detail out and present an analysis of the complexity management operations built on this structure.

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 that 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.

Next we look into some important basic and supplementary operations followed by a discussion of complexity management operations.

One basic operation is finding the *lowest viewable ancestor* of a node, which is done in $O(d_{NF})$ time, where d_{NF} is the depth of the tree the main graph belongs to 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 *lowest common ancestor* of two nodes (or an edge) is again a basic operation with $O(d_{NF})$ time complexity as follows. In this algorithm, both steps have time complexity of $O(d_{NF})$:

Algorithm LowestCommonAncestor ($node_1, node_2$)

(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 requires more effort. 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) Insert edge as a normal intergraph edge
(2) **if** $node_1$ or $node_2$ is unviewable **then**
(3) v_1 = FindViewableAncestor($node_1$)
(4) v_2 = FindViewableAncestor($node_2$)
(5) Create a meta edge between v_1 and v_2
(6) **endif**

Steps (1) and (5) are of $O(1)$ time, and Steps (3) and (4) 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, maintenance of this reference brings an extra space requirement and may be omitted, sacrificing speed for space. The algorithm is as follows:

Algorithm RemoveIntergraphEdge(*intergraphEdge*)

(1) Remove *intergraphEdge* from *intergraph*
(2) **if** *intergraphEdge* was unviewable before removal **then**
(3) **if** its meta edge is only for this *intergraphEdge* **then**
(4) Discard its meta edge
(5) **else**
(6) Remove *intergraphEdge* from associated edge list of its 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 $O(1)$.

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 the nesting forest is a balanced tree with constant non-leaf vertex degrees, the average complexity turns out to be $O((n_{GM} + m_{GM})/g \lg g)$, where n_{GM}, m_{GM}, and g stand for the total number of nodes, edges, and graphs in the graph manager, respectively.

5.1. Expand/collapse

An *expand* operation is applied on 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 on 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. At any time during visualization, we may want to view a single portion of the whole manager. If this portion defines a subtree in the skeleton of our graph manager, then we can set our main graph as the root of this subtree and view only that part. Via the collapse operation, we can hide the nested graph of a node (avoiding inclusion) and draw it as a normal node.

In Fig. 8, there are two different views of the same graph manager. On the left, the graphs A and B are nested into their parent nodes, whereas on the right these parent nodes are collapsed to unnest graphs A and B, respectively.

Collapse 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 all edges of the intergraph and checking whether the owners of both end nodes of the 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 $O(m_{IG} \cdot d_{NF})$, assuming g is much smaller than $m_{IG} \cdot d_{NF}$.

Expand operation can be 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 node to be expanded, 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 $O(m_{meta} \cdot d_{NF})$.

5.2. 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 (reducing the graph's size) and may subsequently be expanded to nest the folded contents of the child graph. Another common way of using this facility is by first creating an empty folder and consequently filling its contents. At any time, an *unfold* operation may be applied on a folder node to reverse the effects of the fold operation.

Often times members of a graph need to be put together according to some criteria to emphasize certain

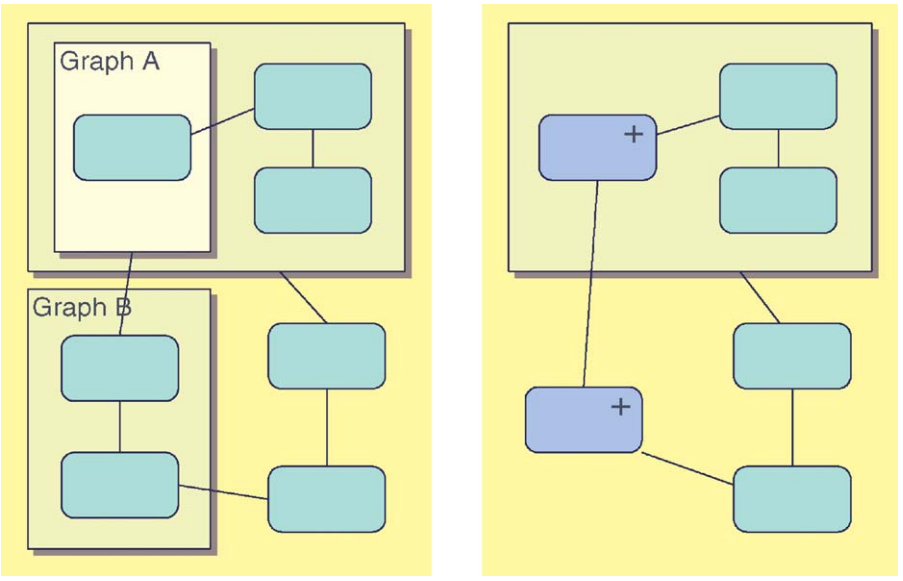

Fig. 8. Two different nestings of the same graph manager realized with expand and collapse operations.

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 could be allowed. Although, this will not let us draw overlapping groups, a node could have multiple views under multiple folders at the same time.

Fig. 9 shows a single graph with 15 nodes in it. A fold operation on the selected node set (with highlighted borders) results in the graph manager on the right. A new folder node is created inside the graph and a new child graph is created for this new node. Then the selected nodes and their edges are transferred into this newly created child graph. The edges of A, B and C whose one end is now inside the folder are converted to intergraph edges, and represented by meta edges since they are not viewable.

Following is the pseudo code for the fold operation:

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. Overall time complexity is again $O(m_{GM} \cdot d_{NF})$.

5.3. 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

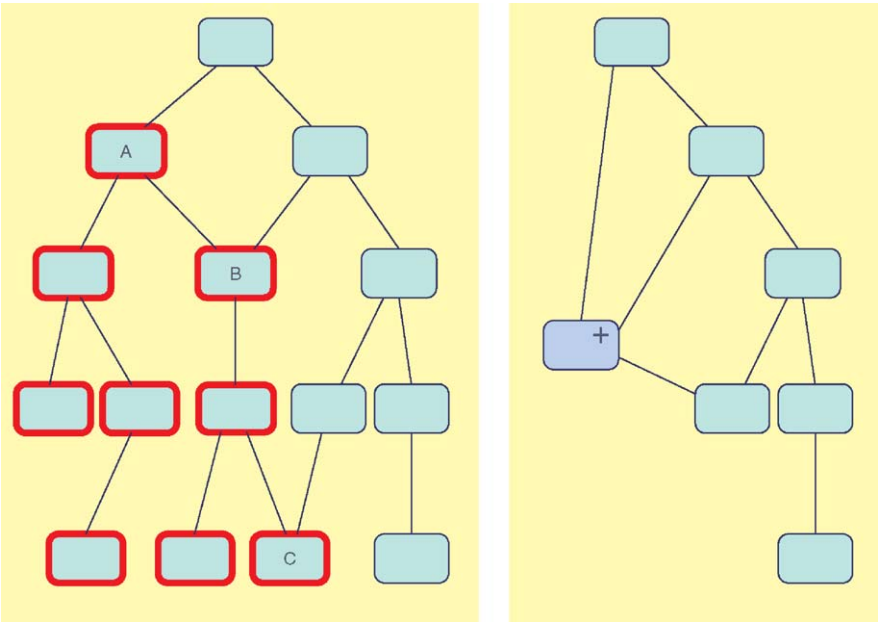

Fig. 9. An example of the fold operation.

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; that is, the user may still interact with it.

These operations are demonstrated in Fig. 10. The graph manager on the left has eight nodes. On the right, we have the same graph manager after hiding E, ghosting A, and setting D invisible. Upon hiding E, E and its incident edges are moved to the hide graph of its owner graph. Ghosting A makes A along with its contents ghosted. Also, all intergraph edges which has one end-node under A are also ghosted. Setting D invisible does not move it to the hide graph but it is not drawn in its owner graph anymore.

6. Implementation

Fig. 11 shows a class diagram summarizing the framework architecture. In this diagram, only the major inheritance and aggregation relations along with 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.

Major parts of our framework and most complexity management operations discussed earlier on have been successfully implemented and integrated into Tom Sawyer Software's Graph Editor Toolkit for Java, version 5. Left of Fig. 12 shows a network (around a hundred PCs, several printers, a few network devices such as servers and routers, etc.) after several steps of complexity management operations including folding of PCs in a lab together, hiding printers and PCs currently unavailable for use. On the right is the same network with varying levels of details of the server revealed through nesting as well as the PCs in certain labs being unfolded for detailed analysis. Of course, some real life applications such as a call graph or a network of a large university campus could be of much higher complexity. In a graph of that kind, benefits of complexity management techniques become much clearer.

We have performed experiments on execution time of the complexity management operations (including expand/collapse, and fold) on different data set (on a PC with Pentium III 733 MHz CPU and 256 MB memory). Each test was executed 10 times and the average time is used as the result. The graph managers used in the tests are created randomly and uniformly. We have assumed a binary navigation tree for these graph managers with total number of nodes and edges uniformly distributed among the graphs. Also intergraph edges were distributed uniformly among the graphs such that any two arbitrary graphs in the graph manager have almost an equal number of intergraph edges connected to their nodes.

All test results are in accord with the theoretical bounds. However, due to space considerations we will

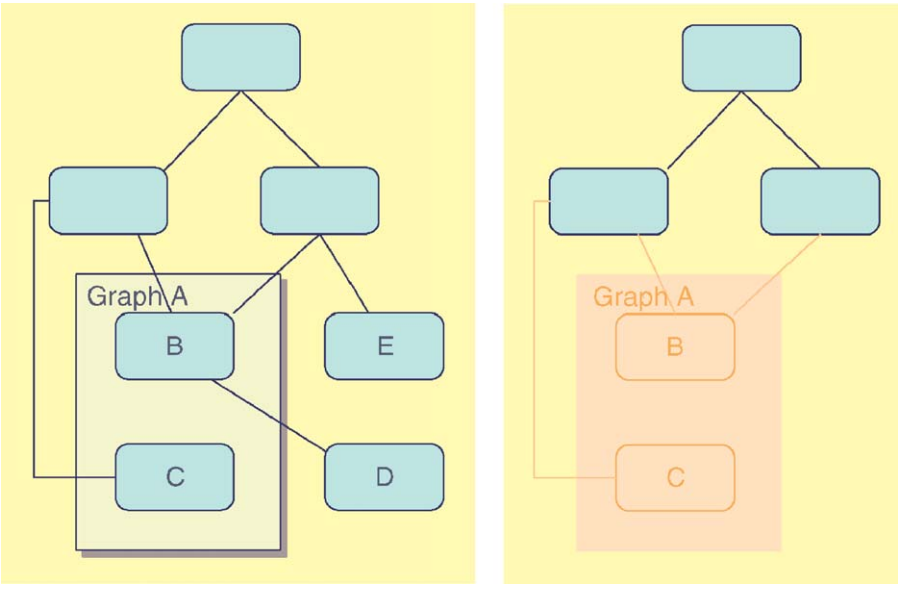

Fig. 10. Hiding, ghosting and invisibility realized on a graph manager.

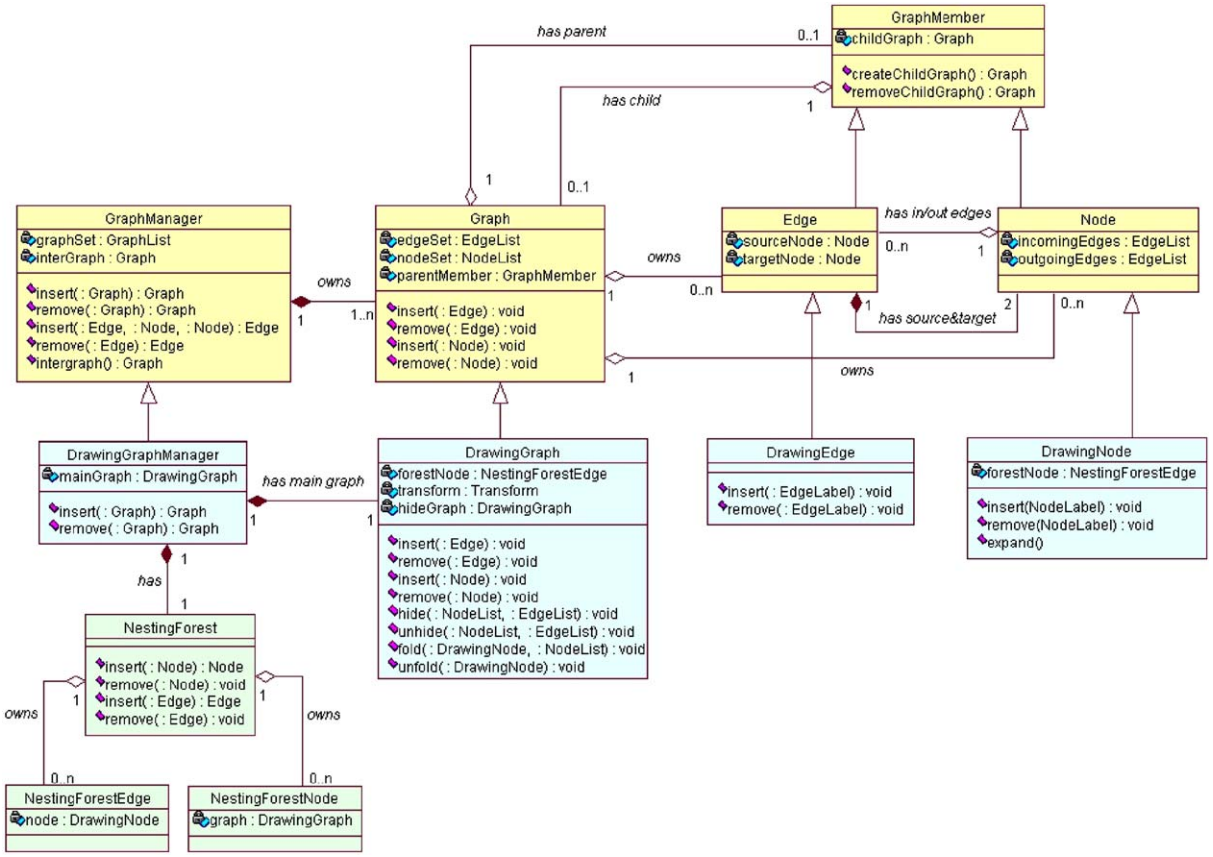

Fig. 11. A class diagram showing important parts of our framework design.

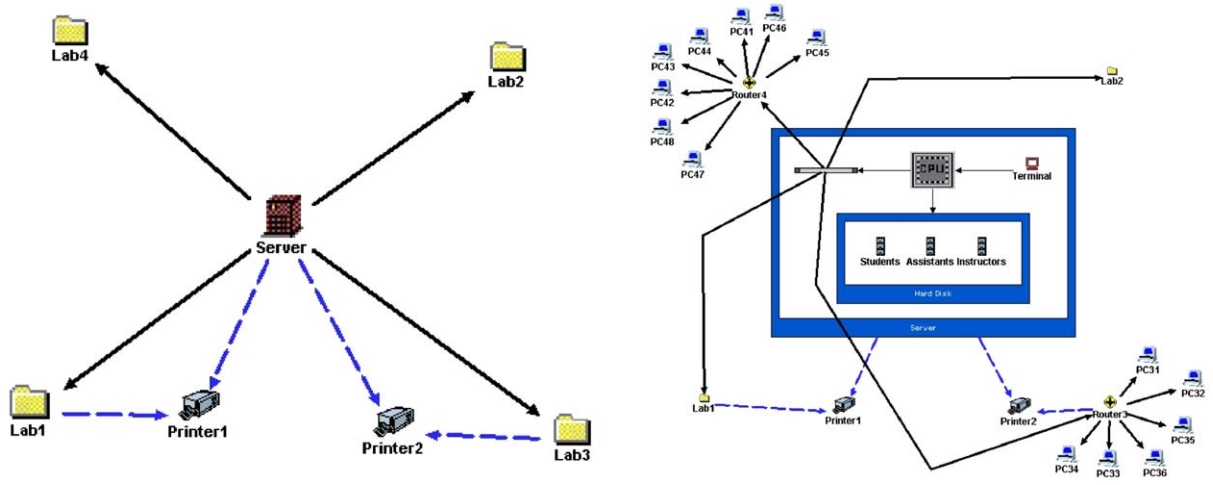

Fig. 12. *Left*: A map of a computer network after a series of complexity management operations applied. *Right*: The same network with certain desired parts revealed for detailed analysis.

give a sample test case where the operation to be tested is Fold and the parameter to be tested is n_{GM}. We started the test with a graph of 1000 nodes, 2000 edges and 100 intergraph edges. Hence a navigation tree of depth 3 is constructed with exactly 7 graphs. Then at each step the number of nodes in the graph manager is increased all

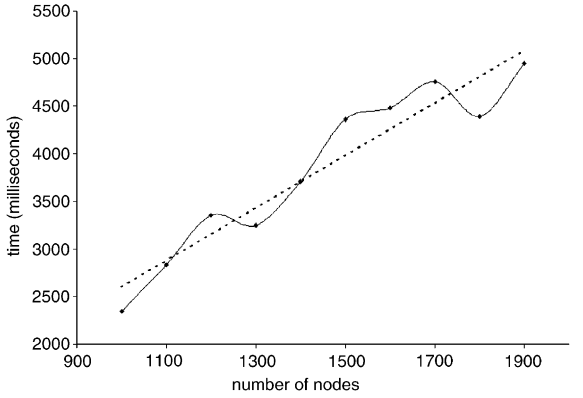

Fig. 13. Affect of n_{GM} on fold. ($m_{GM} = 2000$, $m_{IG} = 100$).

the way up to 2000, without changing the number of edges and intergraph edges. The plot obtained from these tests is given in Fig. 13. It clearly shows the contribution of n_{GM} in the Fold operation to the execution time is linear (the theoretical time complexity of Fold was found to be $O(n_{GM} + (m_{GM} + m_{IG}) \cdot d_{NF})$).

7. Conclusion

We have described a comprehensive framework for development of complexity management techniques for interactive 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. Clear separation of abstractions facilitates more efficient manipulation of both the topology and the geometry of graphs. 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.

References

[1] Di Battista G, Eades P, Tamassia R, Tollis IG. Graph drawing, algorithms for the visualization of graphs. Englewood Cliffs (NJ): Prentice-Hall; 1999.

[2] Dogrusoz U, Feng Q, Madden B, Doorley M, Frick A. Graph visualization toolkits. IEEE Computer Graphics and Applications 2002;22(1):30–7.

[3] Sugiyama K, Misue K. Visualization of structural information: automatic drawing of compound digraphs. IEEE Transactions on Systems Man and Cybernetics 1991;21(4): 876–92.

[4] Fukuda K, Takagi T. Knowledge representation of signal transduction pathways. Bioinformatics 2001;17(9):829–37.

[5] Eades P, Feng Q. Multilevel visualization of clustered graphs. In: North S, editor. Graph drawing, Proceedings of the GD '96, Lecture notes in computer science, vol. 1190. Berlin: Springer; 1997. p. 101–12.

[6] Harel D. On visual formalisms. Communications of the ACM 1988;31(5):514–30.

[7] Brockenauer R, Cornelsen S. Drawing graphs: methods and models. New York: Springer; 2001.

[8] Duncan C, Goodrich M, Kobourov S. Balanced aspect ratio trees and their use for drawing very large graphs. In: Whitesides S, editor. Graph drawing, Proceedings of the GD '98, Lecture notes in computer science, vol. 1547. Berlin: Springer; 1998, p. 111–24.

[9] Raitner M. HGV: a library for hierarchies, graphs and views. In: Goodrich SKMT, editor. Graph drawing, Proceedings of the GD '02, Lecture notes in computer science, vol. 2528. Berlin: Springer; 2002. p. 236–43.

[10] Lai W, Eades P. A graph model which supports flexible layout functions, Technical Report 96-15, Callaghan 2308, Australia; 1996, <URL citeseer.nj.nec.com/189844.html>.

[11] Sarkar M, Brown MH. Graphical fisheye views of graphs. In: Bauersfeld P, Bennett J, Lynch G, editors. Human factors in computing systems, CHI'92 conference proceedings: striking a balance. New York: ACM Press; 1992. p. 83–91.

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

[13] Buchsbaum AL, Westbrook JR. Maintaining hierarchical graph views. In: Proceedings of the 11th annual ACM–SIAM symposium on discrete algorithms. New York: ACM Press; 2000. p. 566–75.

[14] Buchsbaum AL, Goodrich MT, Westbrook J. Range searching over tree cross products. In: European symposium on algorithms, 2000. p. 120–31, <URL citeseer.nj.nec.com/buchsbaum00range.html>.

[15] Herman I, Melançon G, Marshall MS. Graph visualization and navigation in information visualization: a survey. IEEE Transactions on Visualization and Computer Graphics 2000;6(1):24–43.

[16] Sander G. Graph layout through the VCG tool. In: Tamassia R, Tollis I, editors. Graph drawing, Proceedings of the GD '94, Lecture notes in computer science, vol. 894. Berlin: Springer; 1995. p. 194–295.

[17] Sugiyama K, Misue K. A generic compound graph visualizer/manipulator: D-ABDUCTOR. In: Brandenburg FJ, editor. Graph drawing, Proceedings of the GD '95, Lecture notes in computer science, vol. 1027. Berlin: Springer; 1995. p. 500–3.

[18] Lisitsyn IA, Kasyanov VN. Higres—visualization system for clustered graphs and graph algorithms. In: Kratochvíl J, editor. Graph drawing, Proceedings of the GD '99, Lecture notes in computer science, vol. 1731. Berlin: Springer; 1999. p. 82–9.

[19] Huang M, Eades P. A fully animated interactive system for clustering and navigating huge graphs. In: Whitesides S, editor. Graph drawing, Proceedings of the GD '98, Lecture notes in computer science, vol. 1547. Berlin: Springer; 1998. p. 376–83.