Customizing a Fisheye View Algorithm to Preserve the Mental Map

Margaret-Anne D. Storey, [†‡] F. David Fracchia[†] and Hausi A. Müller[‡]

March 25, 1999

[†]School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6.
[‡]Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6.
E-mail: {mstorey}@csr.uvic.ca

Abstract

Frequently large knowledge bases are represented by graphs. Many visualization tools allow users or other applications to interact with and adjust the layouts of these graphs. One layout adjustment problem is that of showing more detail without eliding parts of the graph. Approaches based on a fisheye lens *paradigm seem well suited to this task. However, many of these techniques are non-trivial to implement and their distortion techniques often cannot be altered to suit different graph layouts. When distorting a graph layout, it is often desirable to preserve various properties of the original graph in an adjusted view. Pertinent properties may include straightness of lines, graph topology, orthogonalities and proximities. However, it is normally not possible to preserve all of the original properties of the graph layout. The type of layout and its application should be considered when deciding which properties to preserve or distort. This paper describes a fisheye view algorithm which can be customized to suit various different graph layouts. In contrast to other methods, the user can select which properties of the original graph layout to preserve in an adjusted view. The technique is demonstrated through its application to visualizing structures in large software systems.*

Keywords: visualization, nested graphs, fisheye views, node disjointness problem, graph layouts, software visualization.

1 Introduction

Although the computer screen is relatively small, it is possible to display so much information and detail that it would completely overwhelm the user. It is not the amount of information displayed that is relevant, but rather how it is displayed [41]. Frequently large knowledge bases are represented by graphs. Layout algorithms are often used to present graphs in a more meaningful format. Many visualization tools allow a user or other applications to interact with and adjust these graph layouts.

One layout adjustment problem is that of showing more detail (perhaps by increasing the size of certain nodes) without obscuring other portions of the graph. Approaches based on a *fisheye lens* paradigm seem well suited to this task. However, many of these techniques are non-trivial to implement and their distortion techniques often cannot be altered to suit different graph layouts.

Misue *et al.* describe three properties which should be maintained in adjusted layouts to preserve the user's *mental map*: orthogonal ordering, clusters and topology [19]. Orthogonal ordering of nodes is preserved if the horizontal and vertical ordering of points are maintained. Clusters are preserved by keeping nodes close in the distorted view if they were close in the original view. The topology is preserved if the distorted view of the graph is a homeomorphism of the original view. Other properties which are important to preserve for some applications include straightness of lines, orthogonality of lines parallel to the x and y axes [20], and relative sizes of nodes.

It is impossible to allocate more space to a portion of a graph constrained by a fixed screen size without distorting one or more of the properties described above. The type of layout and its application should be considered when deciding which properties to preserve or distort. For example, in a simple grid layout it may be preferable to preserve parallel and orthogonal relationships among nodes. This could be important for the visualization of large circuit diagrams. For other layouts, such as subway routing maps, the proximity relationships among nodes may be a more important property to preserve.

This paper presents the **S**imple **H**ie**r**arch**i**cal **M**ulti-**P**erspective (SHriMP) layout adjustment algorithm. This algorithm is suitable for creating fisheye views of nested graphs by uniformly resizing nodes when requests for more screen space are made. The SHriMP algorithm is flexible in its distortion technique as it can be altered to suit different graph layouts.

The rest of this paper is structured as follows. Section 2 provides some background on graph drawing. Methods for drawing and visualizing large graphs are briefly discussed in Section 3. Section 4 describes the SHriMP layout adjustment algorithm. Section 5 presents three layout strategies used by SHriMP to preserve important properties of various graph layouts. Section 6 describes the customization of these strategies for more sophisticated layouts. The SHriMP visualization technique is applied to the task of visualizing software structures in Section 7. Finally, Section 8 concludes the paper.

2 Graph Drawing

A *graph* $G = (V, E)$ is a set V of nodes and a set E of arcs where each arc is an unordered pair of nodes [10]. G is *directed* when the arc set E is a set of ordered pairs. In this paper the term graph is used to denote a directed graph. A *nested graph*, in addition to nodes and arcs, contains *composite* nodes which are used to implicitly communicate the hierarchical nature of the graph [11]. In a nested graph a *composite arc* abstracts arcs to lower level nodes in the hierarchy. Nested graphs are often called *inclusion* or *compound* graphs. Figure 1 shows how the structure of a tree may be expressed through spatial containment in a nested graph.

A *graph drawing* is a visual representation of the geometrical description of a graph [17]. Nodes are usually drawn as boxes, points or circles. Arcs may be drawn as one or more straight-line segments connecting nodes. A high number of line segments may give the impression of curved lines connecting nodes. A graph where arcs are drawn as single line segments are called *straight-line* drawings. In a *polyline* drawing, arcs are drawn with multiple straight line segments where segments connect at *bend points*. An *orthogonal-line* drawing constrains arc segments to be parallel to the horizontal or vertical axes of a grid.

A *graph layout* specifies coordinates in the plane for each of the nodes and arc bend points [4]. One example is a grid layout where nodes are positioned on a rectangular grid. A *graph drawing algorithm* automatically calculates a layout for an input graph [5]. Graph layouts for nested or compound graphs are presented in [23, 28, 38]. Graph drawing algorithms use additional information or constraints to compute a layout to improve the readability of a graph. Graph layout algorithms may be designed to minimize the number of arc crossings, minimize the total area occupied by the drawing, display symmetries in the graph or avoid node and arc intersections.

An original graph layout may be adjusted for several reasons: to add or delete nodes in the graph; to reposition selected nodes so that the graph is more informative; to abstract a subgraph into a composite node; or to allocate more space to certain structures in the graph.

The following section describes various solutions to the problem of displaying large graphs on a small computer display.

3 Dealing with Large Graphs

Manipulating large graphs on a small display can be problematic. Various methods have been proposed for displaying and manipulating large graphs. Some of these techniques rely on multiple views which segment the graph into subgraphs, while others display the graph in a single view.

3.1 Multiple Views

For a large information space, a representative graph may contain thousands of nodes and arcs. Some tools partition large graphs into pieces and then display each piece in a separate window [32]. These windows may be cascaded or arranged so that the user can selectively choose which part of the graph to examine further. Frequently, an overview or map window is provided to show a less detailed, global view of the entire graph. A rectangle marker in an overview window is sometimes used to show which area is enlarged in a more detailed view. For many applications, this *multiple view* approach is less than satisfactory since the user has the difficult task of accurately conceptualizing and integrating implicit relationships among the contents of the individual windows.

3.2 Single View: Pan+Zoom

Several approaches display large graphs using a single, unified view. In a pan+zoom view, the user may *pan* (drag) the view using scroll bars or by dragging the mouse over the view. Also, the user may explore the graph in varying detail by *zooming in* (enlarging) or *zooming out* (shrinking) the view. Pad++ [1] uses optimized graphics to achieve smooth panning and zooming, making out-of-view parts of the graph quickly accessible. One problem with the pan+zoom technique is that a user can only enlarge one area of interest at a time in a given view. For some applications, it is often necessary to look at several disjoint parts of the graph in detail at the same time in order to compare them.

3.3 Single View: Context+Detail

Several approaches have been devised to present detailed information while also providing the user with important contextual cues. The term *fisheye view*, coined by Furnas [9], is commonly used to describe views that simultaneously display both context and detail. A fisheye lens has the distorting optical effect that objects closer to the center of the view appear increasingly larger. Furnas' original technique applied a degree of interest function to two measures for each object: its *a priori* relevance to the user and its distance from a focal point. When applied, this function produces a set of priority values. Objects with priorities below a certain threshold are filtered from the view. Furnas applied these ideas to display program source code and to display calendars.

The *Treemap* [14], *Interactive Graph Layout* [12] and *Layout Independent Fisheye View* [23] systems use a static, context+detail display technique. There is no stability in the graph layouts generated by these systems. A reissued layout might change dramatically due to a small change in requested node sizes. For many applications, such as software visualization, a graph layout is often designed to assist the programmer in building a mental map of the information structures presented. Drastically changing the whole layout due to a change of focus would disturb the visualizer's mental map.

Dynamic context+detail display techniques allow the user to change the area(s) of interest interactively and incrementally. For example, *SemNet* was developed for exploring and modifying large information spaces

[7]. SemNet uses 3-D point perspective (closer objects are displayed larger than objects farther away) to create a fisheye view effect. An advantage of 3-D drawing is that the view can be rotated until arcs of interest do not overlap. However, these graph displays tend to be cluttered and unfortunately impede understanding of the structure [27].

Bifocal Displays present objects on the screen using two magnification factors [16]. This algorithm splits the screen into nine regions, where the central region is the focus, and the surrounding regions are uniformly scaled to accommodate the larger center panel. Nodes that are orthogonal in the original view remain orthogonal in the distorted view.

Misue *et al.* [20] described three interactive methods for visualizing nested graphs: *Biform*, *Fisheye* (FE), and *Orthogonal Fisheye* (OFE). The *Biform* technique uniformly demagnifies nodes outside the areas of focus while preserving orthogonal relationships between nodes. The Biform method uses view areas, where items inside these areas are uniformly magnified, and items outside are uniformly demagnified. This method preserves the straightness of lines and the orthogonal ordering of nodes in the distorted view. The *Fisheye Display* (FE) method uses an inverse tangent function to distort the view in a fisheye lens manner. This method does not maintain orthogonal ordering. The *Orthogonal Fisheye Display* (OFE) method, a variant of FE, preserves both straightness of lines and orthogonal orderings. The FE and OFE display methods both tend to distort the graph too much, making the objects near the boundaries appear compressed.

Sarkar and Brown extended Furnas' technique to create interactive fisheye views of graphs [29]. Points of greater interest are magnified and points of lesser interest are demagnified by distorting the space around the focal point. Nodes farther away from the focal point appear increasingly smaller. For many applications, however, this use of distance to derive interest or importance may not be entirely suitable. Related approaches include *Perspective Wall* [18], *Document Lens* [26], *Rubber Sheet* [30], *Hyperbolic Fisheye Views* [15], and *3DPS* [3].

The *Continuous Zoom* algorithm by Dill *et al.* [6], is suitable for interactively displaying hierarchically-organized, two-dimensional networks. This approach allows users to view and navigate nested graphs by expanding and shrinking nodes. The underlying algorithm uniformly resizes nodes to provide space for focal points and uses a budgeting process to distribute space among the nodes in the network. However, this algorithm adversely distorts certain layouts, such as spring [8] and tree [25] layouts. The SHriMP fisheye algorithm, first described in [33], can adjust an existing layout while preserving selected constraints in various layouts.

The next section describes the SHriMP algorithm.

4 The SHriMP Layout Adjustment Algorithm

The SHriMP layout adjustment algorithm is elegant in its simplicity. Nodes in the graph uniformly *give up* screen space to allow a node of interest to grow.

Figure 2 shows an example where one node is enlarged in a grid layout of nine nodes. Figure 2(a) shows the graph before the node of interest (the center node) is scaled by the desired factor. The node grows by *pushing* its neighbor nodes outward as if there were infinite screen space (Fig. 2(b)). The node and its neighbors are then scaled around the center of the screen so that they fit inside the available space (Fig. 2(c)).

Each neighbor is pushed outward by adding a translation vector $[\mathbf{T} = \mathbf{T}_x, \mathbf{T}_y]$ to its coordinates, it is then scaled by a factor s around the center of the screen (x_p, y_p). The scale factor s is equal to the ratio of the size of the bounding box divided by the size of the bounding box after applying $\mathbf{T}$. Equations (1) and (2) show the functions applied to the coordinates (x,y) of the neighbor nodes to determine the new position (x', y'):

$$x' = x_p + s\,(x + \mathbf{T}_x - x_p) \tag{1}$$

$$y' = y_p + s\,(y + \mathbf{T}_y - y_p) \tag{2}$$

To shrink a node that has previously been enlarged, the following inverse equations are used:

$$x = (x' - x_p)/s + x_p - \mathbf{T}_x \tag{3}$$

$$y = (y' - y_p)/s + y_p - \mathbf{T}_y \tag{4}$$

In a nested graph, the node of interest pushes the boundaries of its parent node outward also. The parent in turn pushes its siblings[1] out and so on until the root is reached. As a final step, everything is scaled to fit inside the root. In Figure 3, the node labeled A is enlarged causing its siblings, B and C to be pushed out. Its parent node, D, also grows in response to A's request for more space. D in turn pushes its siblings outward. As a final step, everything is scaled to fit in the root node (H) since H cannot grow any larger due to limited screen space.

A simple extension allows for multiple focal points of varying scaling factors. To scale multiple nodes, each node in turn may grow (or shrink) pushing outward (or pulling inward) their neighbors. Finally, nodes are scaled to fit inside the available space (cf. Fig. 4). This is more efficient since the scaling step does not need to be repeated for every focal point. In addition, this ensures that the set of multiple focal points can be scaled uniformly.

This algorithm is simple, fast and effective. When considering only one focal point, the algorithm is linear with respect to the number of nodes in the graph. When scaling multiple nodes, it is $O(kn)$ where k equals the number of focal points and n is the total number of nodes in the graph. For most applications, k will be very small. The next section describes how different translation vectors may be used for repositioning neighbors when a node is resized.

5 Preserving the Mental Map

When zooming a node in a graph layout it is often desirable to maintain pertinent properties of the layout such as orthogonality, proximity, straightness of lines and the overall topology of the graph. It is impossible to preserve all of a graph's layout properties using a fixed screen size. Depending on the graph layout and its application, it is often only necessary to preserve some of these properties.

With the SHriMP layout adjustment algorithm, a node grows (or shrinks) by pushing (pulling) its neighbor nodes outward (inward) along vectors. The translation vectors determine how the neighbor nodes are repositioned when a request for more space is made. This section describes three methods for setting the

[1]Nodes which have the same parent in the nested graph are called siblings.

magnitude and direction of a vector.

5.1 Preserving Orthogonality

One layout strategy, called **ORTHO**, preserves orthogonal relationships among nodes. The graph is divided into nine partitions by extending the edges of the scaled node. The translation vector for each neighbor node is calculated according to the partition containing its center. Figure 6 summarizes the translation vectors for each of the nine partitions. For example, the translation vector for those nodes in the top left partition is **T**$= [-d_x, d_y]$, where d_x and d_y are the x and y differences between the new size of the scaled node and its previous size. All neighbor nodes above (below) the scaled node are pushed upward (downward) by the same amount, thereby maintaining the orthogonal relationships of these nodes with respect to the y axis. Similarly, nodes to the right (left) are pushed right (left) by the same amount, maintaining the orthogonal relationships with respect to the x axis.

The following equations show the translation vectors to be applied to a neighbor node **B** when the node of interest **A** is scaled, (x_a, y_a) and (x_b, y_b) are the origins of the nodes **A** and **B** respectively:

$$\mathbf{T}_x = \begin{cases} 0 & \text{if } x_b = x_a \\ -d_x & \text{if } x_a > x_b \\ d_x & \text{if } x_a < x_b \end{cases} \qquad (5)$$

$$\mathbf{T}_y = \begin{cases} 0 & \text{if } y_b = y_a \\ -d_y & \text{if } y_a > y_b \\ d_y & \text{if } y_a < y_b \end{cases} \qquad (6)$$

In Fig. 5, a simple grid layout of a graph is displayed. Figure 5(a) shows the grid before any nodes have been scaled. Figure 5(b) demonstrates how **ORTHO** maintains the grid-like appearance of a graph when a node is resized. This strategy is similar in result to the Continuous Zoom algorithm [6], the Biform Display Method [20] and the Bifocal Display [16].

5.2 Preserving Proximities

Many layout algorithms position nodes in groups or clusters to depict certain relationships in the graph. For example, a spring layout algorithm positions nodes which are highly connected closer to one another [5]. In interactive systems, where the user is allowed to manually position nodes, the ORTHO strategy would not be suitable as minor node position adjustments could be badly distorted in the boundaries of the nine regions surrounding a scaled node. Therefore, a layout strategy which keeps those nodes that are close in the original view close in the distorted view would be beneficial.

A layout strategy, called **PROX1**, preserves clusters by constraining each neighbor node to stay on the line connecting its center to that of the node being resized. When a node is resized, it pushes a neighbor node outward along this line. Therefore, the direction of each node's translation vector is equal to the direction of the line connecting the centers. The magnitude (μ) of this vector is equal to the distance that a corner point of the scaled node moves as it is enlarged.

In Figure 7, the node A is enlarged. d_x and d_y are the x and y differences between the new size of A and its previous size. (x_a, y_a) is the center of A, and (x_b, y_b) is the center of B, a neighbor of A. The direction of B's translation vector is equal to the *direction* of the connecting line, and its magnitude is equal to μ as per Equation (7). Equations (8) and (9) show how to calculate the translation vector $\mathbf{T} = [\mathbf{T}_x,\mathbf{T}_y]$. Note that μ is constant for all neighbor nodes, and need only be calculated once.

$$\mu = \sqrt{d_x^2 + d_y^2} \tag{7}$$

$$\mathbf{T}_x = \mu \frac{x_b - x_a}{\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}} \tag{8}$$

$$\mathbf{T}_y = \mu \frac{y_b - y_a}{\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}} \tag{9}$$

This strategy has been applied to the grid in Figure 5(c). This figure demonstrates that this strategy does not preserve all of the orthogonal relationships of **ORTHO**, but it does keep those nodes which were close in the original view, close in the transformed view. However, the screen space is not being used effectively by this method, valuable space is wasted in the corners of the screen. **PROX2**, described in the next subsection, makes better use of screen space while maintaining similar proximity relations.

5.3 An Alternative Proximity Preservation Strategy

An alternative proximity preserving strategy, **PROX2**, makes use of the fact that nodes are often drawn as rectangles.[2] As in **PROX1**, the direction of the translation vectors is determined by the relative location of the sibling node to the node of interest. However, the magnitude (i.e. the amount they are pushed away from the node of interest) is not the same for all sibling nodes. Instead, a node pushes out sibling nodes only as much as is necessary to make room for the node of interest to grow. Therefore, the magnitude of the translation vector is also dependent on the relative location of the sibling node to the node of interest.

For example, in Fig. 8, node A is to be enlarged. The $\mathbf{T}_y$ component of the translation vector is set to d_y. Since the sibling node is above A, it must be pushed upwards by at least this amount to provide room for A to grow in that direction. The $\mathbf{T}_x$ component is calculated by solving for $\mathbf{T}_x$ in a point-line equation of the line through (x_b, y_b) and (x_a, y_a).

For square nodes, the $\mathbf{T}_x$ and $\mathbf{T}_y$ components may be calculated using the following simplified equations. m is equal to the slope of the line connecting the centers of A and B and the negative values are selected for $\mathbf{T}_x$ and $\mathbf{T}_y$ when $x_b < x_a$ and $y_b < y_a$ respectively.

$$\mathbf{T}_x = \begin{cases} 0 & \text{if } |m| = \infty \\ \pm\min(d_x, \frac{d_y}{|m|}) & \text{otherwise} \end{cases} \tag{10}$$

$$\mathbf{T}_y = \begin{cases} 0 & \text{if } |m| = 0 \\ \pm\, min(d_y, |m| * d_x) & \text{otherwise} \end{cases} \tag{11}$$

Figure 9 shows the benefits of using this approach to enlarge several nodes in a spring layout. Figure 9(a) shows the original spring layout. In Fig. 9(b), several nodes were enlarged using the **ORTHO** strategy. Note that this strategy distorts some of the clusters in the spring layout and destroys the user's mental map

[2] The **ORTHO** and **PROX1** strategies can also be applied to circular nodes.

in the process. The same nodes are scaled in Fig. 9(c) using the proximity relationships preserving strategy, **PROX2**. The general appearance of the spring layout is maintained by retaining the proximity relationships between nodes in the adjusted view.

5.4 Preserving the Topology of a Graph

The orthogonal and proximity preserving strategies also preserve the topology of the graph nodes. That is, nodes that do not overlap in the original view, will not overlap in the adjusted view. However, additional arc intersections may be introduced in an adjusted view. This could be avoided by drawing arcs as curved line segments instead of straight line segments [2]. However, the extra overhead for translating arc segment end points would be quite significant.

The next section in this paper describes how the strategies just described may be customized and blended to preserve important properties of more sophisticated layouts.

6 Hybrid Strategies

To preserve the mental map of more sophisticated layouts other considerations may need to be taken into account. By developing hybrid strategies based on the ORTHO and PROX strategies, it is possible to preserve the layouts of more sophisticated graph layouts. Several examples are described next.

6.1 Tree Layouts

In a tree layout of an acyclic graph it is desirable to preserve orthogonal relationships between levels in the hierarchy while repositioning a node's descendants so that they remain close to one another. This can be achieved by applying a variation of the orthogonal preserving strategy, **ORTHO**. If the $\mathbf{T}_x$ component of the translation vector for descendants of the node to be scaled is set to 0, the descendants' relative horizontal positions do not change. The following equations describe the translation vector for this constrained variation of the **ORTHO** strategy:

$$\mathbf{T}_x = \begin{cases} 0 & \text{if } x_b = x_a \text{ or if } \mathbf{B} \text{ is a descendant of } \mathbf{A} \\ -d_x & \text{if } x_a > x_b \text{ and } \mathbf{B} \text{ is not a descendant of } \mathbf{A} \\ d_x & \text{if } x_a < x_b \text{ and } \mathbf{B} \text{ is not a descendant of } \mathbf{A} \end{cases} \tag{12}$$

$$\mathbf{T}_y = \begin{cases} 0 & \text{if } y_b = y_a \\ -d_y & \text{if } y_a > y_b \\ d_y & \text{if } y_a < y_b \end{cases} \tag{13}$$

Figure 10 illustrates the advantage of applying this hybrid strategy to a tree graph. Figure 10(a) shows the tree layout before any nodes are scaled. In Fig. 10(b) two nodes are scaled using the orthogonality preservation layout strategy. Note how the children of these scaled nodes are spread apart. The same nodes are scaled using the proximity preservation layout strategy in Fig. 10(c). Here the layout of the node's children and the rest of the graph are distorted. The hybrid strategy described above is applied in Fig. 10(d) which preserves orthogonal relationships between levels in the tree, but keeps children close to their parent nodes. This hybrid strategy more effectively preserves the mental map of a tree layout.

6.2 Directed Acyclic Graph Layouts

The Sugiyama algorithm produces a nice layout for directed acyclic graphs [39]. For Sugiyama layouts it is desirable to preserve orthogonal relationships between levels in the hierarchy when adjusting the layout. However, since nodes may have more than one ancestor in the hierarchy, the strategy just described for tree layouts is not suitable. A better approach is to maintain the relative spacing between nodes in the horizontal direction.

A hybrid of **ORTHO** and **PROX2** can be used to preserve both of these properties for Sugiyama layouts. The orthogonality preservation strategy is used to preserve the parallel relationships between levels in the tree, and the proximity preserving strategy is applied to nodes on the same level.

For this hybrid strategy, the equation for the $\mathbf{T}_x$ component is calculated using Equation 10 from **PROX2** and the equation for the $\mathbf{T}_y$ component is calculated using Equation 6 from **ORTHO**. Figure 11 demonstrates the benefits of using this approach for scaling nodes in a Sugiyama layout.

6.3 Nested Graph Layouts

In a nested graph layout, different layout algorithms may be used to draw subgraphs for different layers in the hierarchy. For example, in Fig. 12, the top level in the hierarchy was laid out in a grid. Tree, spring and grid layouts were used to draw subgraphs in other layers of the hierarchy. When scaling a node in this graph the mental map of each of these layouts can be preserved by applying appropriate translation vectors to the nodes. Each node is repositioned according to its original layout. When a node is scaled, it pushes (pulls) its siblings outward (inward) using a strategy to preserve the layout with which they were drawn. When this change propagates to a parent, the parent pushes (pulls) its siblings outward (inward) using a strategy which preserves the layout for that layer of the hierarchy. Children nodes simply move according to their parent, thus preserving the layouts in lower layers.

6.4 Orthogonal Line Drawings

An orthogonal line drawing constrains arc segments to be parallel to the horizontal or vertical axes of a grid [4]. The ORTHO strategy may be extended to preserve the mental map of orthogonal line drawings. Edge bend points are also translated and scaled when a node is enlarged. Figure 13(a) shows an orthogonal line drawing with edge bend points drawn as small black circles. A single node is selected in Fig. 13(b) so that it may be enlarged. This node pushes out its neighbors and the edge bend points. Finally, the entire drawing is scaled to fit the available space (cf. Fig. 13(c)).

6.5 Node Disjointness

The SHriMP layout algorithm can also be used to make adjustments to a graph with overlapping nodes. The goal is to redraw the graph so that node images will be disjoint. This task is referred to as the *node disjointness problem* [19] or *cluster busting* [17].

According to Misue *et al.* [19], an algorithm which addresses this problem should satisfy three requirements:

1. nodes in the new drawing should be disjoint;
2. the new drawing should be compact and should fit in the available space; and
3. the user's mental map (influenced by the original layout) should be preserved.

The SHriMP layout adjustment algorithm may be applied as follows. An initial pass through the set of graph nodes should be made to determine which nodes (if any) overlap. Sorting the nodes first will improve the efficiency of this step. If two nodes overlap, one of these nodes should *push* out the rest of the graph by the amount of the overlap so that they no longer overlap (Requirement 1). Finally, if it is necessary, the graph is scaled to fit in the available screen space (Requirement 2). Appropriate selection of the various layout preserving strategies, will help to maintain the user's mental map (Requirement 3). Figure 14 demonstrates how the SHriMP algorithm is used to reposition nodes in a graph with overlapping nodes.

7 Visualizing Software Structures using SHriMP Views

SHriMP views have been added to the Rigi system for documenting and manipulating structures of large software systems [37]. The Rigi reverse engineering system is designed to analyze, summarize and document the structure of large software systems [44]. The Rigi system is centered around a language-independent graph editor for presenting software artifacts. The first phase of reverse engineering a subject software system is fully automatic and involves parsing the software and storing the extracted artifacts. This first phase results in a flat resource-flow graph which can be manipulated using the Rigi graph editor.

Various visualization tools are available in the Rigi editor to aid the reverse engineer in discovering and documenting system design information. Some of these tools include selection algorithms, filtering (elision) algorithms, software metrics, and graph layout algorithms [22, 40].

For large software systems, understanding the structural aspects of a system's architecture is initially more important than understanding any single component [44]. The SHriMP visualization technique has been integrated in the Rigi system for exploring legacy systems. SHriMP views are particularly well suited to showing different levels of abstraction in a system's architecture concurrently. Nodes are used to represent artifacts in the software, such as functions or data variables. Arcs represent dependencies among these artifacts, such as call dependencies. Composite nodes correspond to subsystems in the software. The nesting feature of nodes communicates the hierarchical structure of the software (e.g. subsystem or class hierarchies). The SHriMP layout adjustment algorithm provides the ability to browse pertinent information in large software systems. By zooming different areas in a large graph, a software engineer can quickly identify important features such as highly connected nodes and candidate subsystems.

Reverse engineering a system involves information extraction and information abstraction [21]. One objective of a reverse engineer is to obtain a mental model of the structure of a subject software system and to communicate this model effectively. A reverse engineer uses visualization techniques to facilitate the identification of candidate subsystems and to assist in the visualization of structures and patterns in the graph. The application of graph layout algorithms play a key role in communicating the reverse engineer's mental model, and in the identification of structures and patterns in the software.

Figure 15(a) shows the result of applying a spring layout algorithm to the graph representation of the SQL/DS software [44]. This layout algorithm assisted in the identification of several candidates for subsystems, by clustering groups of highly connected nodes around the fringe of the graph. In Figure 15(b), the user has selected and enlarged one of these clusters, in order to see more detail. By using the SHriMP layout adjustment algorithm which preserves proximity relationships, this structure was emphasized without adversely affecting the general layout of the graph.

In Rigi, tree layout algorithms are commonly used for visualizing call graphs, data dependency trees and other hierarchies. For example, Fig. 16 shows a call dependency tree routed at the main function in a small C program. This program implements a list data structure. One of these nodes, mylistprint has been expanded by the SHriMP layout adjustment algorithm using the hybrid layout strategy suitable for tree layouts. By zooming nodes in this fashion, software engineers can read the source code of the mylistprint function and at the same time maintain their mental map of the location of this function in the call dependency tree.

Rigi is end-user programmable [40] through the RCL (Rigi Command Language) which is based on the Tcl/Tk language [24]. This feature is very powerful and allows for significant experimentation with different visualization approaches in Rigi. The first implementation of SHriMP views was also implemented in the Tcl/Tk language and was therefore easily integrated in the Rigi system. Since SHriMP (through Rigi) is end-user programmable, the layout strategy can be dynamically changed for one or more nodes. The user can experiment with a variety of hybrid strategies based on the graph layout hierarchy. This may be done by writing a series of RCL scripts and invoking these scripts dynamically through the RCL interpreter in Rigi.

Although the SHriMP layout adjustment algorithm is linear for a small number of focal points, scaling many graphical nodes (which may have source code displayed inside them) can be quite slow in Tcl/Tk. A more recent version of SHriMP [35] uses Pad++ [1], a graphics extension for Tcl/Tk. Pad++ is a substrate for building multiscale, dynamic user interfaces. It is optimized for efficiently displaying large numbers of objects and smoothly animating the motions of panning and zooming on standard PC or workstation hardware.

In this newer prototype of SHriMP, the nested graph views and fisheye-view algorithm are implemented in Tcl/Tk using the Pad++ widgets. Since Pad++ supports HTML items in its canvas, we used this feature to display HTML'ized source code inside the nodes, as opposed to displaying ordinary text. This latest version is currently being evaluated through user testing [36].

8 Conclusions

This paper describes the SHriMP layout adjustment algorithm. This algorithm is suitable for uniformly resizing nodes when requests for more screen space are made. It preserves straightness of lines and the graph topology of nodes in adjusted views. Moreover, the SHriMP algorithm is flexible in its distortion technique and can be customized to suit various graph layouts. Several variants were described for preserving orthogonal and proximity relationships. Hybrid strategies were also shown to be feasible, and are useful when trying to preserve the mental map of more sophisticated layouts. The effects of the different layout adjustment strategies on various graph layouts are summarized in Fig. 17. The SHriMP algorithm can also be applied to the node disjointness problem.

This technique, due to its simplicity, can be easily integrated with existing graph drawing tools. This has been demonstrated through its integration with the Rigi system, where it was used for creating fisheye views of nested graphs representing software systems. The SHriMP algorithm is also currently being used in medical image analysis for viewing and analyzing large sets of medical images on a computer screen [43].

Graphical fisheye views have become increasingly popular in recent years. It is generally accepted by many in the visualization community that fisheye views are a viable approach for presenting large amounts of information on a small screen. However, there is little empirical evidence that clearly indicates their value. There has been a lack of evaluation in this area (notable exceptions are [7, 9, 13, 16, 31, 42]). We have compared SHriMP views to other visualization methods, such as a multiple window approach and pan+zoom, in two user studies [34, 36]. Initial observations indicate that fisheye views may be useful in certain scenarios, but further studies are needed to evaluate the benefits of the various layout strategies and to determine if fisheye views are a useful medium for visualizing large software programs.

Acknowledgements

This work was supported by the Rigi lab, Department of Computer Science, University of Victoria, the Graphics and Multimedia Research Lab, School of Computing Science, Simon Fraser University and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] B. Bederson and J. Hollan, *Pad++: A zooming graphical interface for exploring alternate interface physics*, in Proceedings of ACM UIST'94, (Marina del Rey, California), November, 1994, pp. 17–26.

[2] M. Carpendale, D. Cowperthwaite, M.-A. Storey, and F. Fracchia, *Exploring distinct aspects of the distortion viewing paradigm*, Tech. Rep. TR 97-08, School of Computing Science, Simon Fraser University, September 1997.

[3] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia, *3-dimensional pliable surfaces: For effective presentation of visual information*, in Proceedings of ACM UIST'95, November, 1995, pp. 217–227.

[4] I. F. Cruz and R. Tamassia, *How to visualize a graph: Specification and algorithms, a graph drawing tutorial*, in Graph Drawing '94, DIMACS International Workshop, Princeton, New Jersey, October, 1994.

[5] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis, *Algorithms for graph drawing: An annotated bibliography*, Computational Geometry Theory and Applications, 4 (1994), pp. 235–282.

[6] J. Dill, L. Bartram, A. Ho, and F. Henigman, *A continuously variable zoom for navigating large hierarchical networks*, in Proceedings of the 1994 IEEE Conference on Systems, Man and Cybernetics, October 1994, pp. 386–390.

[7] K. Fairchild, S. Poltrock, and G. Furnas, *SemNet: Three-dimensional graphic representations of large knowledge bases*, in Cognitive Science and its Applications for Human-Computer Interaction, R. Guindon, ed., Lawrence Erlbaum Associates, Publishers, 1988, pp. 201–233.

[8] T. Fruchtermann and E. Reingold, *Graph drawing by force-directed placement*, Tech. Rep. UIUC CDS-R-90-1609, Department of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[9] G. Furnas, *Generalized fisheye views*, in Proceedings of ACM CHI'86, (Boston, MA), April 1986, pp. 16–23.

[10] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1989.

[11] D. Harel, *On visual formalisms*, Communications of the ACM, 31(5) (May 1988), pp. 514–530.

[12] T. Henry and S. Hudson, *Interactive graph layout*, in Proceedings of UIST '91, Hilton Head, South Carolina, November 1991, pp. 55–64.

[13] J.G.Hollands, T. Carey, M. Matthews, and C. McCann, *Presenting a graphical network: A comparison of performance using fisheye and scrolling views*, in Designing and using human-computer interfaces and knowledge-based systems, G. Salvendy and M. Smith, eds., Elsevier, 1989.

[14] B. Johnson and B. Shneiderman, *Tree-maps: A space-filling approach to the visualization of hierarchical information structures*, in Proceedings of Visualization '91, (San Diego, California), October 1991, pp. 284–291.

[15] J. Lamping and R. Rao, *Laying out and visualizing large trees using a hyperbolic space*, in Proceedings of ACM UIST'94, (Marina del Rey, California), November, 1994, pp. 13–14.

[16] Y. Leung, R. Spence, and M. Apperley, *Applying bifocal displays to topological maps*, International Journal of Human-Computer Interaction, 7(1) (1995), pp. 79–98.

[17] K. A. Lyons, *Cluster busting in anchored graph drawing*, in Proceedings of the 1992 CAS Conference, vol. II, November, 1992, pp. 327–337.

[18] J. Mackinlay, G. Robertson, and S. Card, *The perspective wall: Detail and context smoothly integrated*, in Proceedings of ACM CHI'91, (New Orleans,Louisiana), April 1991, pp. 173–179.

[19] K. Misue, P. Eades, W. Lai, and K. Sugiyama, *Layout adjustment and the mental map*, Journal of Visual Languages and Computing, 6 (1995), pp. 183–210.

[20] K. Misue and K. Sugiyama, *Multi-viewpoint perspective display methods: Formulation and application to compound graphs*, in 4th Intl. Conf. on Human-Computer Interaction, Stuttgart, Germany, vol. 1, Elsevier Science Publishers, September 1991, pp. 834–838.

[21] H. Müller, M. Orgun, S. Tilley, and J. Uhl, *A reverse engineering approach to subsystem structure identification*, Journal of Software Maintenance: Research and Practice, 5(4) (December 1993), pp. 181–204.

[22] H. Müller, S. Tilley, M. Orgun, B. Corrie, and N. Madhavji, *A reverse engineering environment based on spatial and visual software interconnection models*, in Proceedings of the Fifth ACM

SIGSOFT Symposium on Software Development Environments (SIGSOFT '92), (Tyson's Corner, Virginia; December 9-11, 1992), December 1992, pp. 88–98. In *ACM Software Engineering Notes*, 17(5).

[23] E. Noik, *Layout-independent fisheye views of nested graphs*, in VL'93: IEEE Symposium on Visual Languages, (Bergen, Norway), August 1993, pp. 336–341.

[24] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[25] E. Reingold and J. Tilford, *Tidier drawing of trees*, IEEE Transactions on Software Engineering, SE-7(2) (March 1981), pp. 223–228.

[26] G. Robertson and J. Mackinlay, *The document lens*, in Proceedings of ACM UIST'93, (Atlanta, Georgia), November 3-5, 1993, pp. 101–107.

[27] G. Robertson, J. Mackinlay, and S. Card, *Cone trees: Animated 3d visualizations of hierarchical information*, in Proceedings of ACM CHI'91, (New Orleans,Louisiana), April 1991, pp. 189–194.

[28] G. Sander, *Graph layout for applications in compiler construction*, Tech. Rep. A/01/96, Universität des Saarlandes, February, 1996.

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

[30] M. Sarkar, S. Snibbe, O. Tversky, and S. Reiss, *Stretching the rubber sheet: A metaphor for viewing large layouts on small screens*, in Proceedings of ACM UIST'93, (Atlanta, Georgia), November, 1993, pp. 81–91.

[31] D. Schaffer, Z. Zuo, S. Greenberg, L. Bartram, J. Dill, S. Dubs, and M. Roseman, *Navigating hierarchically clustered networks through fisheye and full-zoom methods*, ACM Transactions on Information Systems, (1995), pp. 162–188.

[32] M.-A. Storey, F. Fracchia, and H. Müller, *Cognitive design elements to support the construction of a mental model during software exploration*, Journal of Software Systems, 44 (1999), pp. 171–185. special issue on Program Comprehension.

[33] M.-A. Storey and H. Müller, *Graph layout adjustment strategies*, in Proceedings of Graph Drawing 1995, (Passau, Germany, Springer Verlag, September 1995, pp. 487–499. Lecture Notes in Computer Science.

[34] M.-A. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. Müller, *On designing an experiment to evaluate a reverse engineering tool*, in Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE96), Monterey, California, November 1996, pp. 31–40.

[35] M.-A. Storey, K. Wong, F. Fracchia, and H. Müller, *On integrating visualization techniques for software exploration*, in IEEE Symposium on Information Visualization (INFOVIS '97), Phoenix, Arizona., October 1997, pp. 38–45.

[36] M.-A. Storey, K. Wong, and H. Müller, *How do program understanding tools affect how programmers understand programs*, in Proceedings of the 4th Working Conference on Reverse Engineering (WCRE97), Amsterdam, Holland, October 1997, pp. 12–21.

[37] M.-A. D. Storey, H. Müller, and K. Wong, *Manipulating and documenting software structures*, in Software Visualization, P. Eades and K. Zhang, eds., World Scientific Publishing Co., 1996, pp. 244–263.

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

[39] K. Sugiyama, S. Tagawa, and M. Toda, *Methods for visual understanding of hierarchical systems*, IEEE Transactions on Systems, Man, and Cybernetics, 11(4) (1981), pp. 109–125.

[40] S. Tilley, K. Wong, M.-A.Storey, and H. Müller, *Programmable reverse engineering*, International Journal of Software Engineering and Knowledge Engineering, 4 (1994), pp. 501–520.

[41] E. Tufte, *Envisioning Information*, Graphics Press, 1990.

[42] D. Turo and B. Johnson, *Improving the visualization of hierarchies with treemaps: Design issues and experimentation*, in Proceedings of Visualization '92, (Boston, Masachusetts: 19-23 October 1992), Oct 1992, pp. 124–131.

[43] J. van der Heyden, M. Carpendale, K. Inkpen, and M. Atkins, *Visual presentation of magnetic resonance images*, in IEEE Conference on Visualization (VIS'98), October 1998, pp. 423–426.

[44] K. Wong, S. Tilley, H. Müller, and M.-A. Storey, *Structural redocumentation: A case study*, IEEE Software, 12 (1995), pp. 46–54.

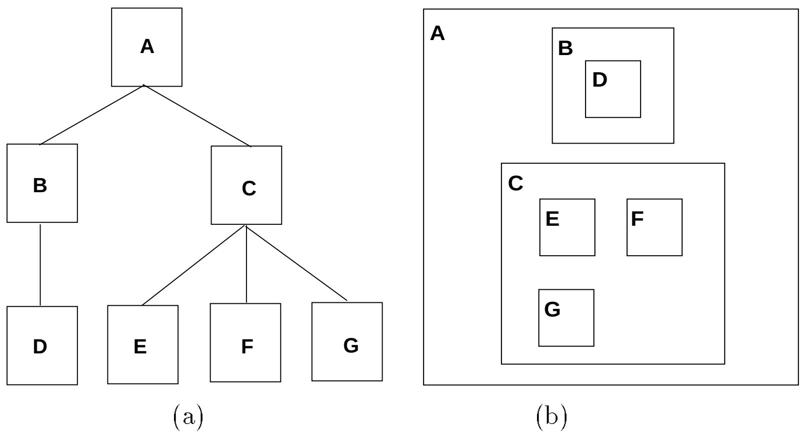


Figure 1: (a) A tree view of a hierarchy. (b) A nested graph view of the same hierarchy with composite nodes A, B, and C.

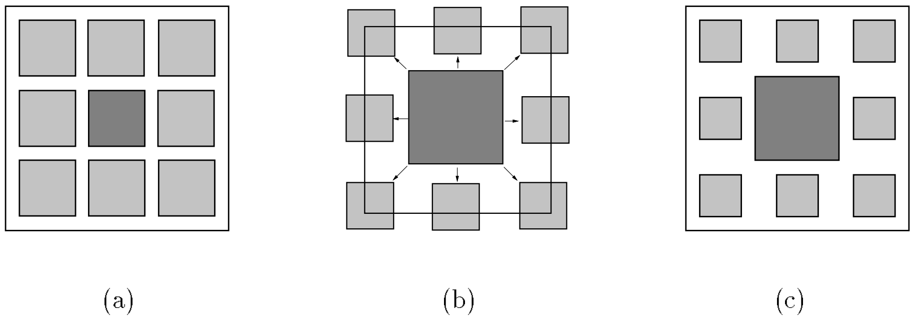


Figure 2: (a) The graph before any scaling is done. (b) The node of interest (center node) grows by the desired scale factor and pushes its neighbors outward. (c) Finally, the node and its neighbors are scaled to fit inside the parent. This last step is the only step visible to the user of SHriMP.

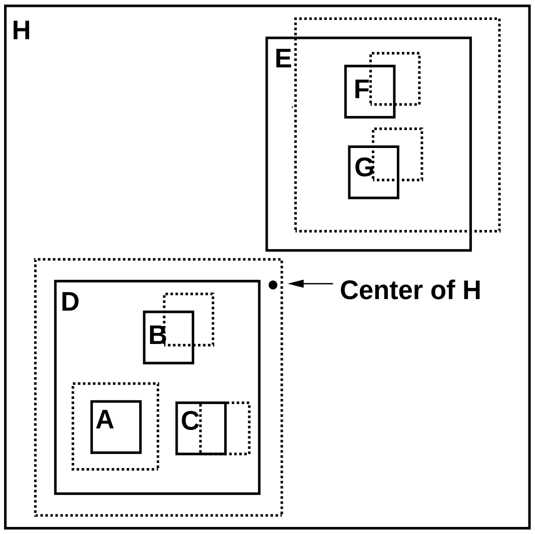


Figure 3: The node A is enlarged. This change is propagated from child to parent. Because H is fixed in size, its descendants are scaled to fit inside H. Dotted lines denote the new positions of the nodes.

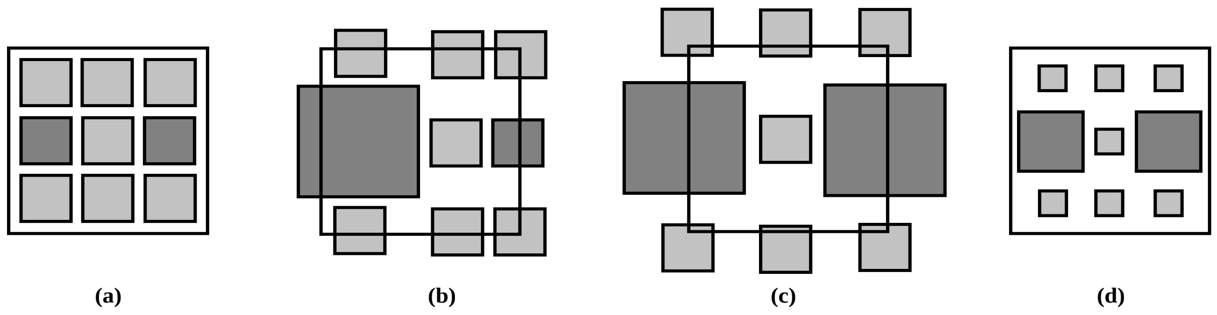


Figure 4: (a) Grid before any scaling is done; two nodes are selected to be enlarged. (b) The first node pushes the other nodes away to provide room for it to grow. (c) Similarly, the second node pushes away nodes to allow it to grow. (d) Finally, everything is scaled to fit in the available space.

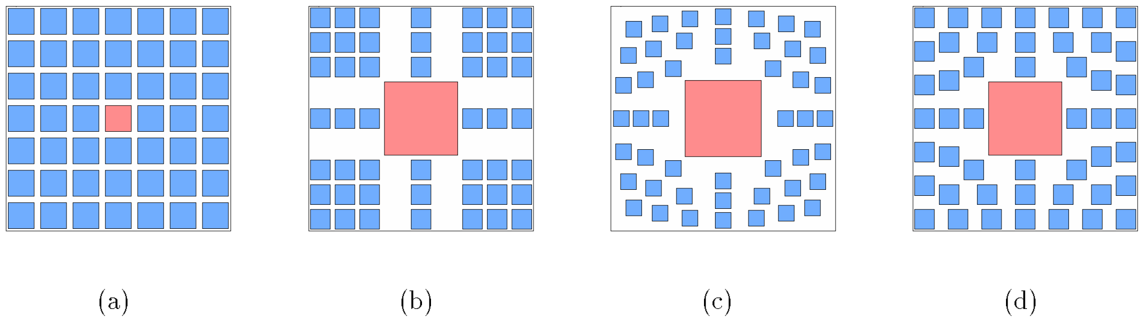


Figure 5: (a) Grid before any scaling is done. (b) The center node is scaled using a layout strategy which preserves orthogonality in the graph. (c) and (d) The center node is scaled using layout strategies which are more suited to preserving proximities in the graph.

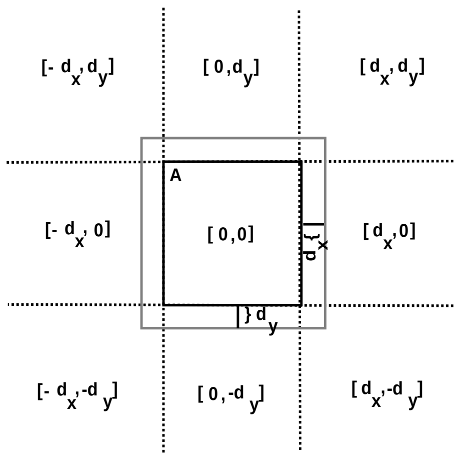


Figure 6: In the ORTHO layout strategy, the translation vector for each neighbor node is determined by the partition containing its center.

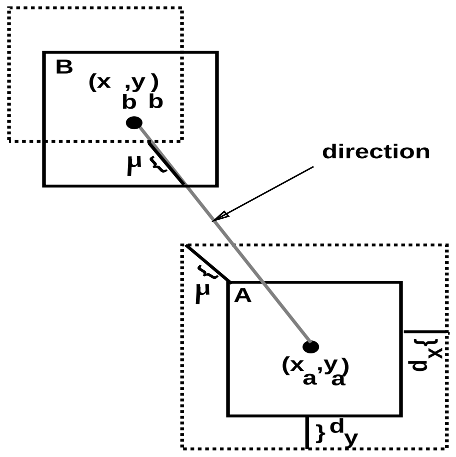


Figure 7: Sibling node, *B*, is pushed outward along the line connecting its center and the center of *A*, the node being scaled.

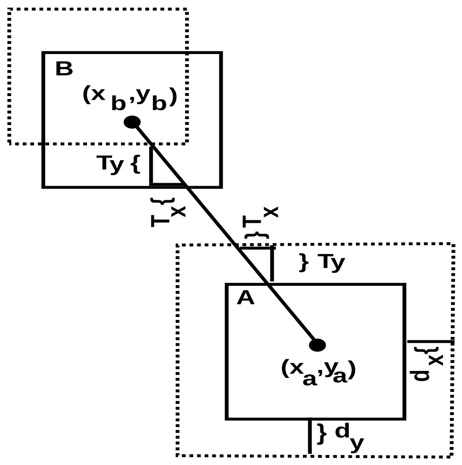
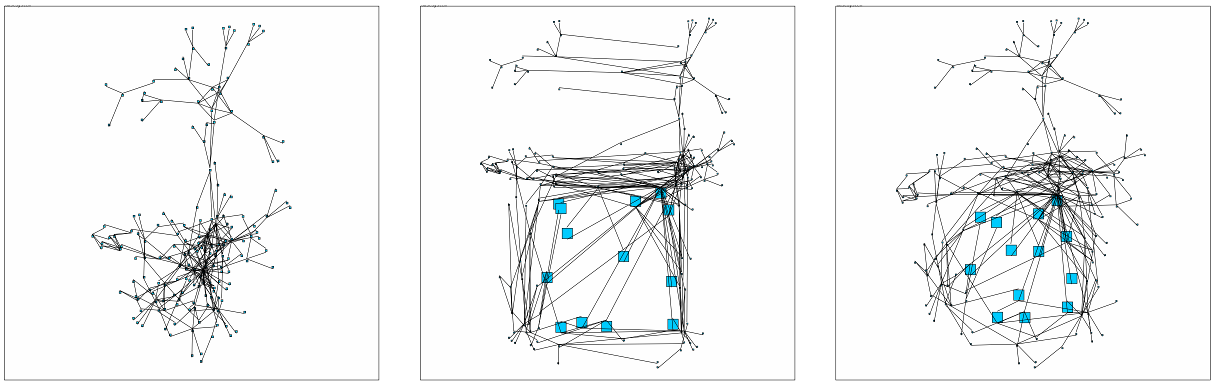


Figure 8: The node, B, is pushed along the vector between its center and that of A, the node being scaled. The distance it is pushed along this vector is determined by the displacement of the intersecting node's edge as it moved along the vector.



(a) (b) (c)

Figure 9: (a) A spring layout of a graph before any nodes are scaled. (b) Several nodes are enlarged using **ORTHO**, which preserves orthogonal relationships. Note how some of the clusters created by the spring layout are distorted. (c) The same nodes are scaled using the proximity preservation layout strategy, **PROX2**. The clusters are better preserved using this strategy.

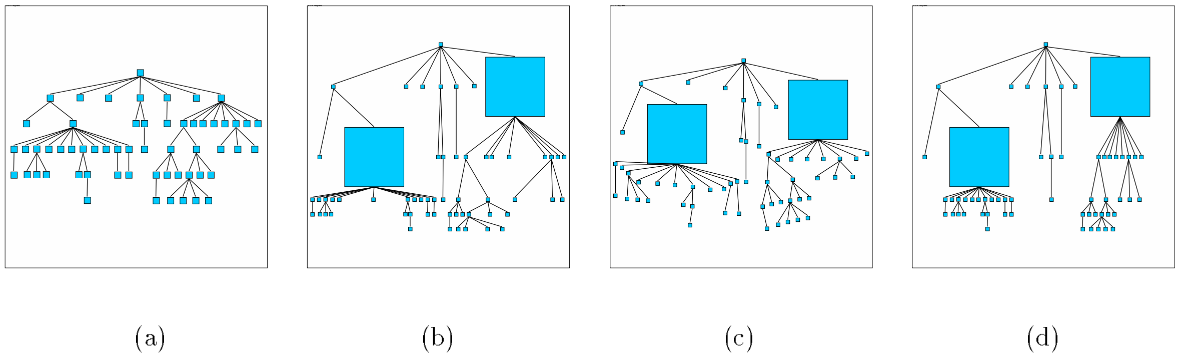


Figure 10: (a) A tree layout before any scaling is done. (b) Two nodes are scaled using the orthogonality preservation layout strategy. (c) The same two nodes are scaled using the proximity preservation layout strategy. (d) Here a hybrid strategy based on **ORTHO** is used to preserve parallel relationships between levels in the tree, and keep siblings close to one another.

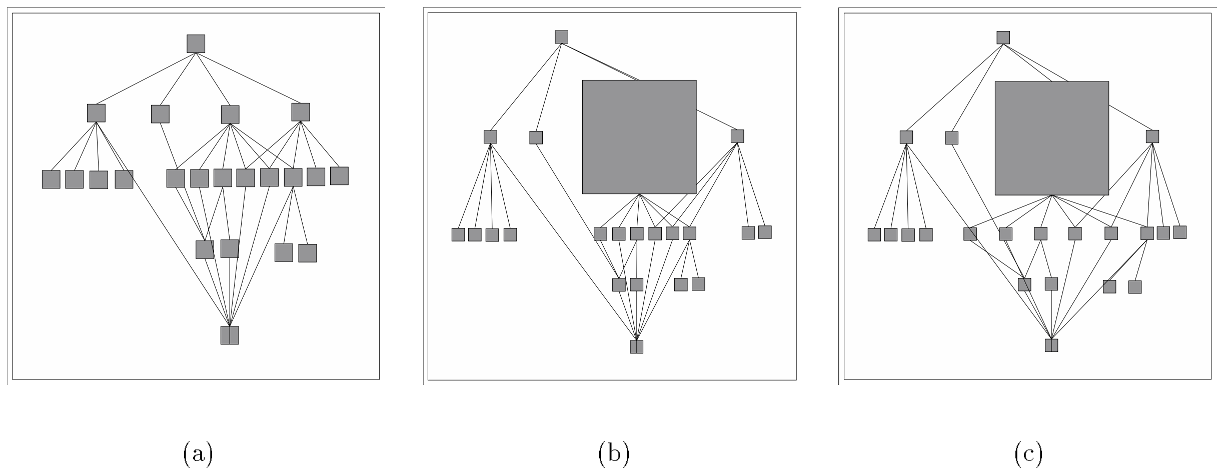


Figure 11: (a) A Sugiyama layout. (b) A node is scaled using the acyclic tree layout preservation strategy, causing the layout of children with more than one ancestor to be badly distorted. (c) Here, the node is scaled using a hybrid strategy which preserves orthogonal relationships between levels in the tree and proximity relationships between nodes on a single level.

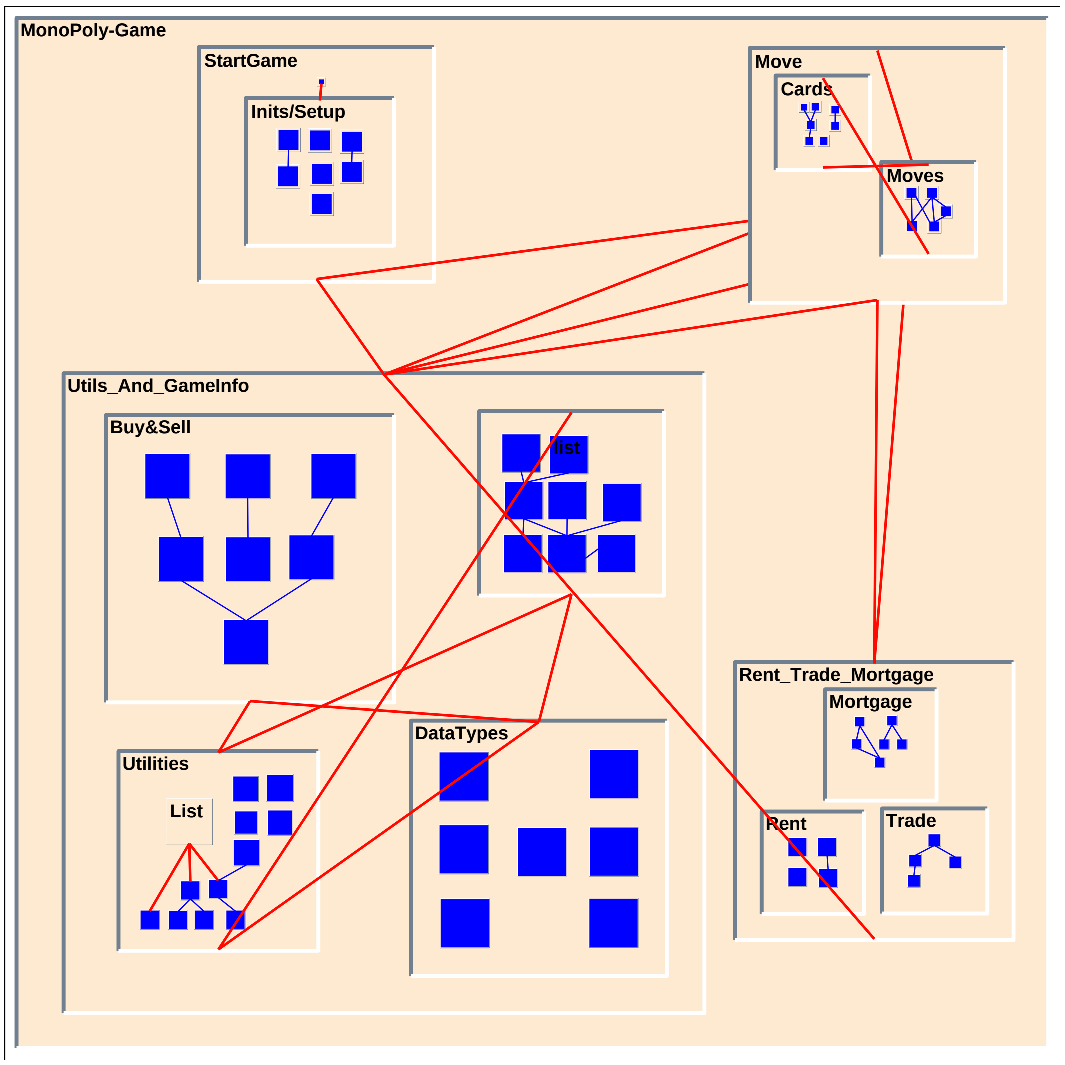


Figure 12: A Nested Graph Layout

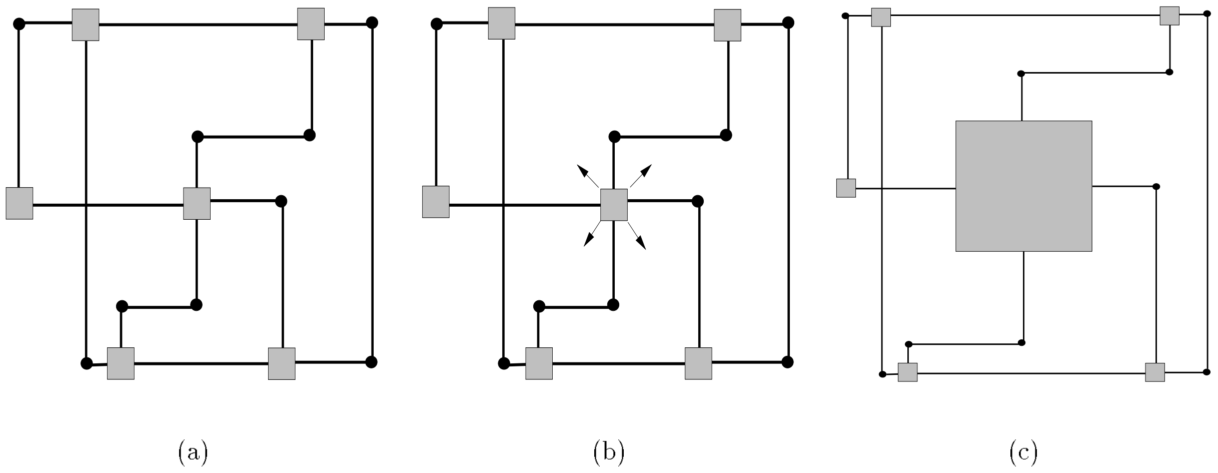
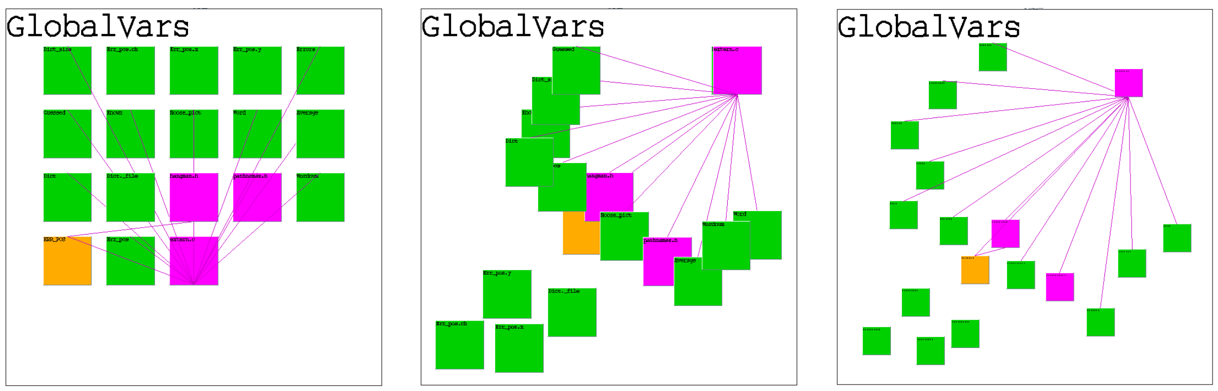
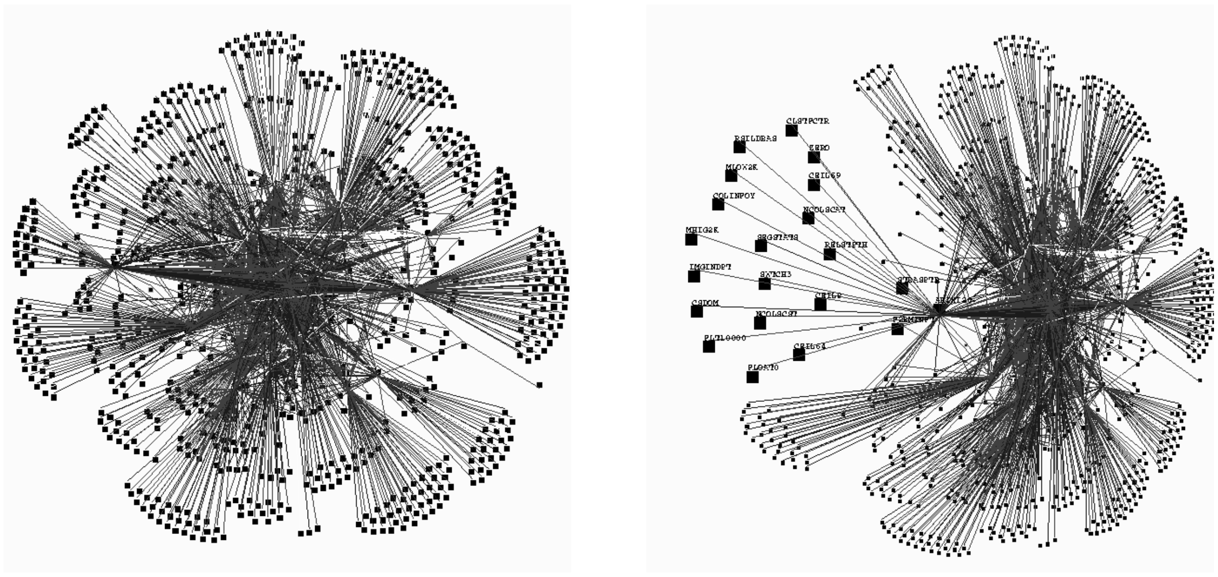


Figure 13: Extending the ORTHO strategy to preserve the mental map of an orthogonal line drawing.



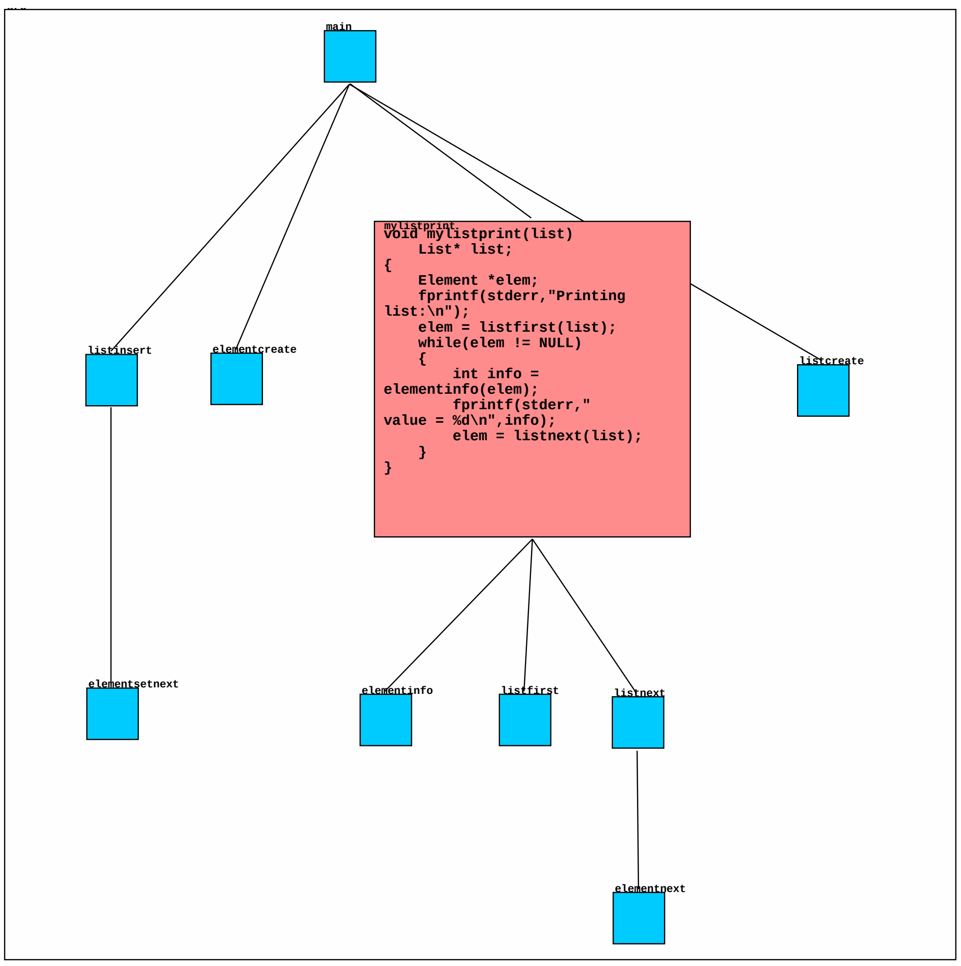
(a) (b) (c)

Figure 14: (a) A grid layout. (b) Here the user has manually repositioned the nodes in a more meaningful layout, resulting in overlapped nodes. (c) The user runs the node disjointness feature (using a strategy to preserve proximity relationships between nodes) so that nodes will not overlap.



(a) (b)

Figure 15: (a) The spring layout algorithm has been applied to the SQL/DS software system. This algorithm helped to expose clusters of nodes on the fringe of the graph, which are candidates for subsystems. (b) One of the node clusters is enlarged, so that it can be examined in more detail. By enlarging these nodes, the user was able to read the node labels on the screen and was able to determine that all of the node labels were prefixed with the same alphanumeric string (a programming convention).



```
void mylistprint(list)
    List* list;
{
    Element *elem;
    fprintf(stderr,"Printing
list:\n");
    elem = listfirst(list);
    while(elem != NULL)
    {
        int info =
elementinfo(elem);
        fprintf(stderr,"
value = %d\n",info);
        elem = listnext(list);
    }
}
```

Figure 16: Browsing software source code using SHriMP Views

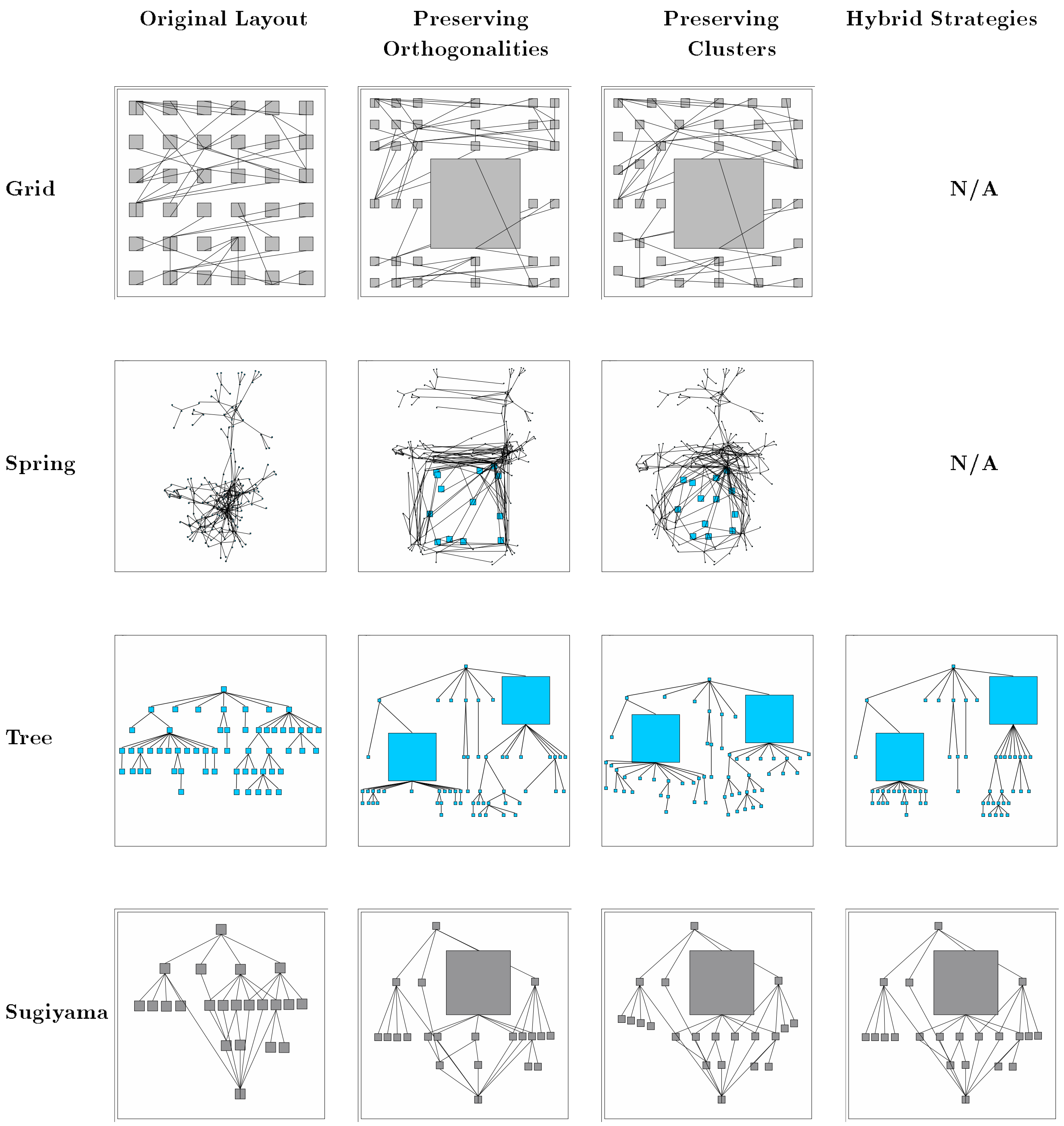


Figure 17: This figure summarizes how different strategies distort or preserve the mental map of various graph layouts.