Graph Layout Adjustment Strategies

Margaret-Anne D. Storey[1 2] and Hausi A. Müller[2]

[1] School of Computing Science, Simon Fraser University, Burnaby, BC, Canada.
[2] Department of Computer Science, University of Victoria, Victoria, BC, Canada.
{mstorey,hausi}@csr.uvic.ca

Abstract. *When adjusting a graph layout, it is often desirable to preserve various properties of the original graph in the adjusted view. Pertinent properties may include straightness of lines, graph topology, orthogonalities and proximities. A layout adjustment algorithm which can be used to create fisheye views of nested graphs is introduced. The SHriMP (**S**imple **H**iera**r**ch**i**cal **M**ulti-**P**erspective) visualization technique uses this algorithm to create fisheye views of nested graphs. This algorithm preserves straightness of lines and uniformly resizes nodes when requests for more screen space are made. In contrast to other layout adjustment algorithms, this algorithm has several variants to preserve additional selected properties of the original graph. These variants use different layout strategies to reposition nodes when the graph is distorted. The SHriMP visualization technique is demonstrated through its application to visualizing structures in large software systems.*

1 Introduction

Although the computer screen is relatively small, it is easy to fill it with so much information and detail that it completely overwhelms the user. It is not the amount of information displayed that is relevant, but rather how it is displayed [17]. 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.

Misue *et al.* in [8] describe three properties which should be maintained in adjusted layouts to preserve the user's *mental map*: orthogonal ordering, clusters and topology. The orthogonal ordering between nodes is preserved if the horizontal and vertical ordering of points is 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 [9], and relative sizes of nodes.

It is impossible to allocate more space to a portion of a graph constrained to fit on 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. In a simple grid layout, it is preferable to preserve parallel and orthogonal relationships among nodes. This is important for the visualization of large circuit diagrams. For other layouts, such as subway routing maps, the proximity relationships among nodes is a more important property to preserve.

One layout adjustment problem is that of showing more detail (perhaps by increasing the size of nodes) without hiding the remainder of the graph. Approaches based on the 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. In addition, several techniques have the side effect of causing too much distortion in some areas of the graph. For some applications, it would be better to evenly distribute the distortion throughout the entire graph by uniformly scaling nodes outside the focal points.

This paper presents the SHriMP layout adjustment algorithm. This algorithm is suitable for creating fisheye views of nested graphs[3] by uniformly resizing nodes when requests for more screen space are made. It preserves straightness of lines and non-overlapping nodes in the original view will not overlap in the adjusted view. Moreover, this algorithm is flexible in its distortion technique as it can be altered to suit different graph layouts.

Several fisheye view methods are briefly discussed in Section 2. The subsequent section presents the SHriMP layout adjustment algorithm. Section 4 describes different layout strategies which are used to preserve important properties of various graph layouts. Finally, the SHriMP visualization technique is applied to the task of visualizing software structures.

2 Fisheye Views

Manipulating large graphs on a small screen can be problematic. Because of this, various methods have been proposed for displaying and manipulating large graphs. One approach partitions the graph into pieces, and then displays one piece at a time in a separate window. However, context is lost as detail is increased. Another approach makes the entire drawing of the graph smaller, thus preserving context, but the smaller details become difficult to read and interpret as the scale is reduced. A combination view can be given by providing context in one window and detail in another but this requires that the user mentally integrate the two—not always an easy task.

Techniques have been developed to view and navigate detailed information while providing the user with important contextual cues. Fisheye views, an approach proposed by Furnas [4], provide context and detail in one view. This display method is based on the fisheye lens metaphor where objects in the center of the view are magnified and objects further from the center are reduced in size. In Furnas' formulation, each point in the structure is assigned a *prior-*

[3] Nested graphs, in addition to nodes and arcs, contain composite nodes which are used to implicitly communicate the hierarchical nature of the graph [5].

ity that is calculated using a *degree of interest (DOI)* function. Objects with a priority below a certain threshold are filtered from the view.

In order to deemphasize information of lesser interest, several variations on this theme have been developed that use size, position, colour, or shading in addition to filtering. For example, *SemNet* uses three-dimensional point perspective that displays close objects larger than objects further away [3]. *Graphical Fisheye Views,* a technique developed by Sarkar and Brown [13], magnifies points of greater interest and correspondingly demagnifies vertices of lower interest by distorting the space surrounding the focal point. Therefore, nodes that are further away from the focal point appear smaller. The *Continuous Zoom Algorithm* by Ho *et al.* [2], suitable for interactively displaying hierarchically-organized, two-dimensional networks, allows users to view and navigate nested graphs by expanding and shrinking nodes. This algorithm uniformly resizes nodes to provide space for focal points. However, the zoom-out operation is not the reverse of the zoom-in operation.

Two methods based on a rubber sheet metaphor are described by Sarkar *et al.* in [14]. The first method, *Orthogonal Stretching*, uses handles to stretch an area of the graph in the x and y directions. Items which fall in these areas are stretched uniformly, while everything outside of these areas contract uniformly. The second method, *Polygonal Stretching*, allows a user to specify a polygonal region. Items inside the polygon are scaled as the polygon is stretched and the rest of the view is scaled smoothly to integrate it with the enlarged region. This method does not have an inverse mapping once a region is scaled.

Misue *et al.* describe three approaches in [9]. They are the *Biform Display Method* (BF), the *Fisheye Display Method* (FE) and the *Orthogonal Fisheye Display Method* (OFE). The BF method uses view areas, where items inside the view areas are uniformly magnified, and items outside of the areas are uniformly demagnified. The BF method is similar to the Continuous Zoom algorithm and the Orthogonal Stretching techniques; all three of these approaches preserve straightness of lines and orthogonal ordering in the distorted view. The FE uses an inverse tangent function to apply a fisheye lens to the view. Objects closer to the center of the lens appear increasingly larger. However, the orthogonal ordering of points is not maintained using this approach. OFE, a variant of FE, does maintain the orthogonal orderings, but this method (as well as FE) tends to have too much distortion for some parts of the graph. A survey of these approaches and others such as *Perspective Wall* and *Cone Trees* are described by Noik in [11].

3 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 1 gives an example where one node is enlarged. Figure 1(a) shows the graph before the node of interest (the center node) is scaled by the desired factor.

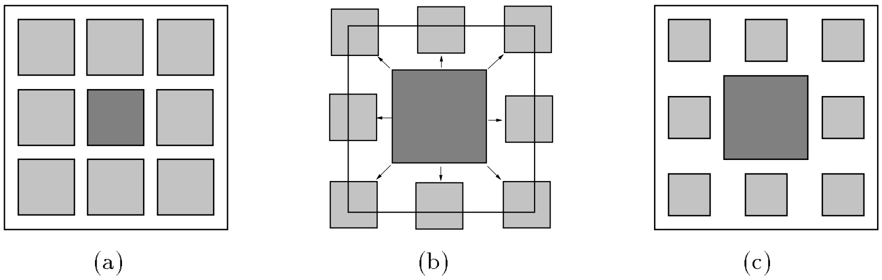


Fig. 1. (a) The graph before any scaling is done. (b) The node of interest (center node) grows by the desired scale factor and pushes its siblings outward. (c) Finally, the node and its siblings are scaled to fit inside the screen. This last step is the only step visible to the user of SHriMP, the other step is shown to describe the algorithm only.

The node grows by *pushing* its sibling[4] nodes outward as if there were infinite screen space, see Fig. 1(b). The node and its siblings are then scaled around the center of the screen so that they will fit inside the available space, see Fig. 1(c).

Each sibling is pushed outward by adding a translation vector $[\mathbf{T}_x, \mathbf{T}_y]$ to its coordinates. The scaling operation makes use of an equation which scales an object around an arbitrary fixed point [6]. In this case, the fixed point is the center of the screen, (x_p, y_p), and the scale factor, s, is equal to the size of the screen divided by the requested size of the screen. Equations (1) and (2) show the function applied to the coordinates (x,y) of the sibling nodes.

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

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

In a nested graph, the node of interest pushes the boundaries of its parent node outward also. The parent in turn pushes its siblings out and so on until the root is reached. As a final step, everything is scaled to fit inside the root.

To shrink a node that has previously been enlarged, the scale factor, s, will be < 1. The zoom-out operation will be the reverse operation of the zoom-in, and vice versa, when the scale factor is set appropriately. 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 siblings. Finally, nodes are scaled to fit inside the available space.

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. In most applications, k is much smaller than n. The next section describes how different translation vectors may be used for repositioning siblings when a node is resized.

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

4 Layout Strategies

When zooming a node in a graph layout it may be desirable to maintain pertinent properties of the layout such as orthogonality, proximity, straightness of lines and the overall topology of the graph. The layout strategies presented in this paper preserve straightness of lines and the graph topology of the nodes. However, it is difficult to preserve both orthogonality and proximity relations using a fixed screen size. Depending on the graph layout, it is often only necessary to preserve one of these properties.

In the SHriMP layout adjustment algorithm, a node grows (or shrinks) by pushing its sibling nodes outward along vectors. The translation vectors determine how the sibling nodes are repositioned after a request for more space is made. The following section describes three methods for setting the magnitude and direction of a vector. Figure 2 shows a simple grid layout of a graph. Figure 2(a) shows the grid before any scaling has been done. Parts (b),(c) and (d) show how different translation vectors can alter the appearance of the graph when it is distorted to allocate more space to the center node.

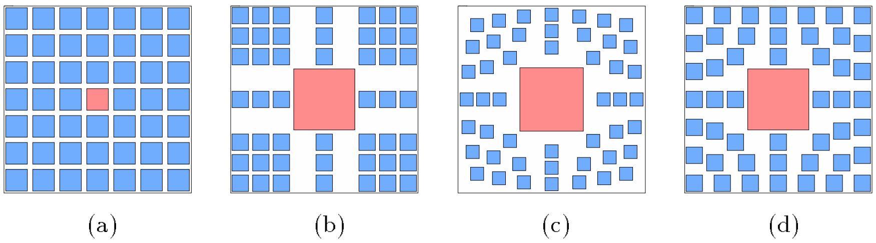


Fig. 2. (a) Grid before any scaling is done. (b) The center node is scaled using a layout strategy which preserves orthogonality in the graph. (c) and (d) The center node is scaled using layout strategies which are more suited to preserving proximities in the graph.

4.1 Preserving Orthogonality

One layout strategy, called **variant 1**, preserves the orthogonal relationships among nodes. The graph is partitioned into nine partitions by extending the edges of the scaled node. The translation vector for each sibling node is calculated according to the partition containing its center. Figure 3 shows the translation vectors for each of the nine partitions. For example, the translation vector for those nodes in partition 1 is $\mathbf{T} = [-d_x, -d_y]$, 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 sibling nodes above (below) the scaled node are pushed upward (downward) by the same amount, thereby maintaining the orthogonality relationships of these nodes with respect to the y axis. Similarly, nodes to the right (left) are pushed

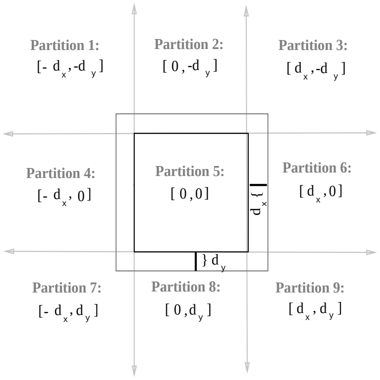


Fig. 3. In this layout strategy, the translation vector for each sibling node is determined by the partition containing its center.

right (left) by the same amount, maintaining the orthogonality relationships with respect to the x axis.

Figure 2(b) demonstrates how **variant 1** maintains the grid-like appearance of a graph when a node is resized. This variant is quite similar in appearance to the Continuous Zoom algorithm reported by Ho *et al.* and the Biform Display method described by Misue *et al.*

4.2 Preserving Proximities

Many layout algorithms position nodes in groups or clusters to depict certain relationships in the graph. For example, spring layout algorithms position nodes which are highly connected closer to one another [1]. Therefore, a layout strategy which keeps those nodes that are close in the original view close in the distorted view would be beneficial.

This subsection describes another variant, called **variant 2**, where proximity relationships are preserved by constraining each sibling node to stay on the line connecting its center to that of the node being resized. When a node is resized, it pushes a sibling node outward along this line. The direction of each sibling node's translation vector is equal to the direction of the line connecting the centers. The magnitude of this vector is equal to the distance that a corner point of the scaled node moves as it is enlarged.

In Fig. 4(a), 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. (x_b, y_b) is the center of B, a sibling 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 (3). Equations (4) and (5) are used to calculate the translation vector $\mathbf{T} = [\mathbf{T}_x, \mathbf{T}_y]$. Note that μ is constant for all sibling nodes, and need only be calculated once.

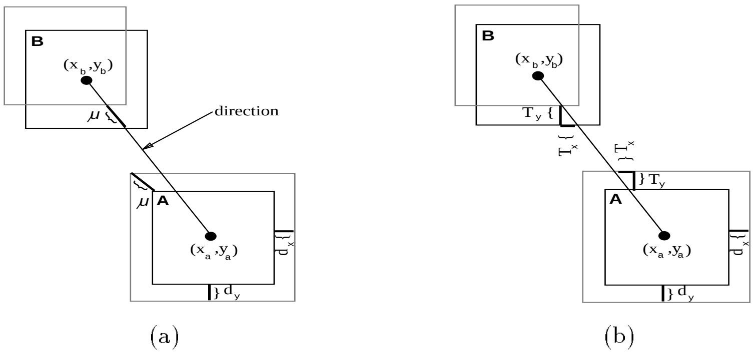


Fig. 4. (a) Sibling node, B, is pushed outward along the line connecting its center and the center of A, the node being scaled. Each node is pushed out by the distance μ. (b) A sibling node, B, is pushed along the vector between its center and that of A, the node being scaled. The distance it is pushed along this vector is determined by the displacement of the intersecting node's edge as it moved along the vector.

$$\mu = \sqrt{d_x^2 + d_y^2} \tag{3}$$

$$\mathbf{T}_x = \mu \frac{x_a - x_b}{\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}} \tag{4}$$

$$\mathbf{T}_y = \mu \frac{y_a - y_b}{\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}} \tag{5}$$

This strategy has been applied to the grid in Fig. 2(c). This figure demonstrates that this strategy does not preserve all of the orthogonal relationships of **variant 1**, but that it does appear to 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. **Variant 3**, described in the next subsection, makes better use of screen space while maintaining similar proximity relations.

4.3 An Alternative Proximity Preservation Strategy

In **variant 3**, the direction of the translation vectors is the same as in **variant 2**, but the magnitude (μ) is not the same for all sibling nodes. Instead, a node pushes out sibling nodes according to the displacement of the scaled node's edge as it moved along the line connecting their centers. This strategy makes use of the fact that nodes are drawn as rectangles in SHriMP views. **Variant 3** is applied to a grid layout in Fig. 2(d).

In Fig. 4(b), node A is being enlarged. m is equal to the slope of the line connecting the centers of A and B. The $\mathbf{T}_x$ and $\mathbf{T}_y$ components are calculated using (6) and (7). In this example, the $\mathbf{T}_y$ component of the translation vector is simply equal to d_y. Since the sibling node is above the scaled node, it is intuitive that it must be pushed upwards by at least this amount to provide room for A to grow in that direction. $\mathbf{T}_x$ is then 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).

Translation vectors for other sibling nodes are calculated similarly, where $-d_x$ is used in place of d_x when $x_b < x_a$, and similarly for d_y.

$$\mathbf{T}_x = \begin{cases} \frac{1}{m}\,(y_b \pm d_y - y_a) + x_a - x_b & \text{if } |m| >= 1 \\ 0 & \text{if } |m| = \infty \\ \pm\, d_x & \text{otherwise} \end{cases} \tag{6}$$

$$\mathbf{T}_y = \begin{cases} m\,(x_b \pm d_x - x_a) + y_a - y_b & \text{if } 0 < |m| < 1 \\ 0 & \text{if } m = 0 \\ \pm\, d_y & \text{otherwise} \end{cases} \tag{7}$$

Figure 5 shows a spring layout of a graph. Figure 5(b) shows the result of applying **variant 3** when scaling several nodes. The general appearance of the spring layout is maintained by retaining the proximity relationships between nodes in the adjusted view. Figure 5(c) shows the same nodes scaled using the orthogonality preservation layout strategy, **variant 1**, which distorts some of the clusters created by the spring layout and destroys the user's mental map in the process.

4.4 Hybrid Strategies

A graph layout may be composed from a variety of layout algorithms [7]. For example, the overall structure of the graph may be that of a tree, where subgraphs are laid out using a simple grid strategy. When zooming a node in any part of the graph, the overall layout as well as the subgraph layouts should be maintained. This is possible as the algorithm can apply different layout strategies to the sibling nodes when a request for more or less space is made. In other words, the method of calculating translation vectors need not be the same for all sibling nodes.

In a tree layout it may be preferable to preserve parallel relationships between levels in the tree hierarchy while repositioning children so that they remain close to their parent node. A hybrid strategy based on **variant 1** can retain both properties for these layouts. If we set the $\mathbf{T}_x$ component of the translation vector for children of the node of interest to 0, the children will be not be spread apart horizontally. Figure 6 shows the advantage of applying this hybrid strategy to a tree layout.

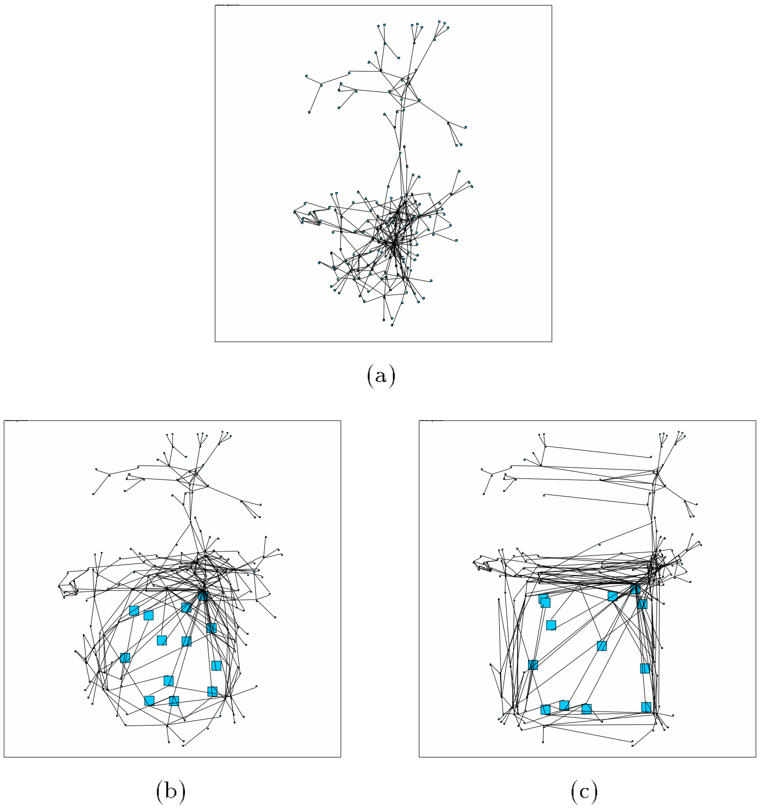
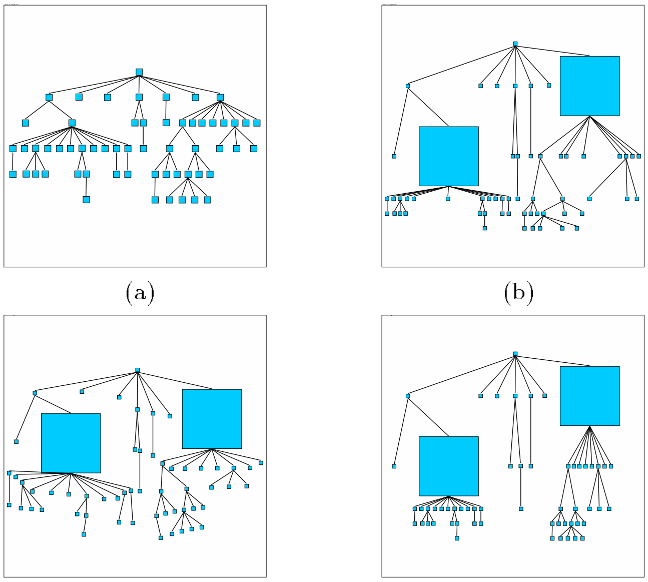


Fig. 5. (a) A spring layout of a graph before any scaling is done. (b) Several nodes are scaled using **variant 3**, which preserves proximities. The clusters are not distorted using this strategy. (c) The same nodes are scaled using the orthogonality preservation layout strategy, **variant 1**. Note how some of the clusters created by the spring layout are distorted.

5 Visualizing Software Structures Using SHriMP Views

The SHriMP visualization technique has been incorporated in the Rigi system for documenting and manipulating structures of large software systems [15]. The Rigi reverse engineering system is designed to analyze, summarize, and document the structure of large software systems [18]. The SHriMP visualization technique helps to alleviate problems of losing context while exploring the many relationships in a multi-million line legacy system.

For large software systems, understanding the structural aspects of a system's architecture is initially more important than understanding any single component [18]. The SHriMP visualization technique is 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



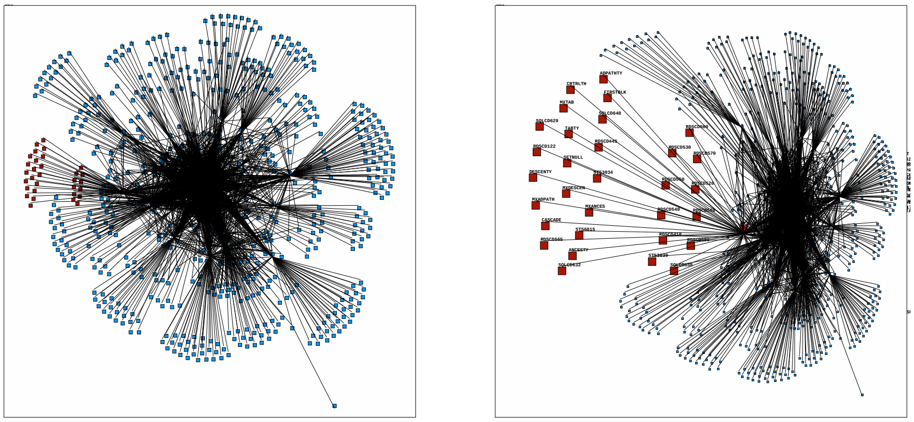
(a) (b)

(c) (d)

Fig. 6. (a) A tree layout before any scaling is done. (b) Two nodes are scaled using the orthogonality preservation layout strategy. Note how the children of these scaled nodes are spread apart. (c) Two nodes are scaled using the proximity preservation layout strategy. Note how the layout of the children of these nodes and the rest of the graph is distorted. (d) Two nodes are scaled using a hybrid strategy which preserves parallel relationships between levels in the tree, and keeps children close to their parents.

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 user may incrementally expose the structure of the software by magnifying subsystems of current interest. The SHriMP layout adjustment algorithm provides the ability to browse groups of nodes and arcs in large software systems. By zooming on 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 [10]. 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.



(a) (b)

Fig. 7. (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 clusters of nodes is enlarged to show more detail.

Figure 7(a) shows the result of applying a spring layout algorithm to the graph representation of the SQL/DS software [18]. This layout algorithm assisted in the identification of several candidates for subsystems, by clustering groups of nodes around the fringe of the graph. In Fig. 7(b), the user has selected and zoomed one of these clusters, in order to see more detail. By using the SHriMP layout adjustment algorithm which preserves the proximity relations, this structure was emphasized without adversely affecting the general layout of the graph.

In the Rigi system, a variety of tree layout algorithms are used for visualizing call graphs, data dependency trees and other hierarchies. For example, Fig. 8 shows a call dependency tree routed at the main function in a small program written in the C language. 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 the node in this fashion, a software engineer can read the source code of the mylistprint function and at the same time maintain his mental map of the location of this function in the call dependency tree.

Rigi is end-user programmable [16] through the RCL (Rigi Command Language) which is based on the Tcl/Tk language [12]. The SHriMP visualization technique is 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.

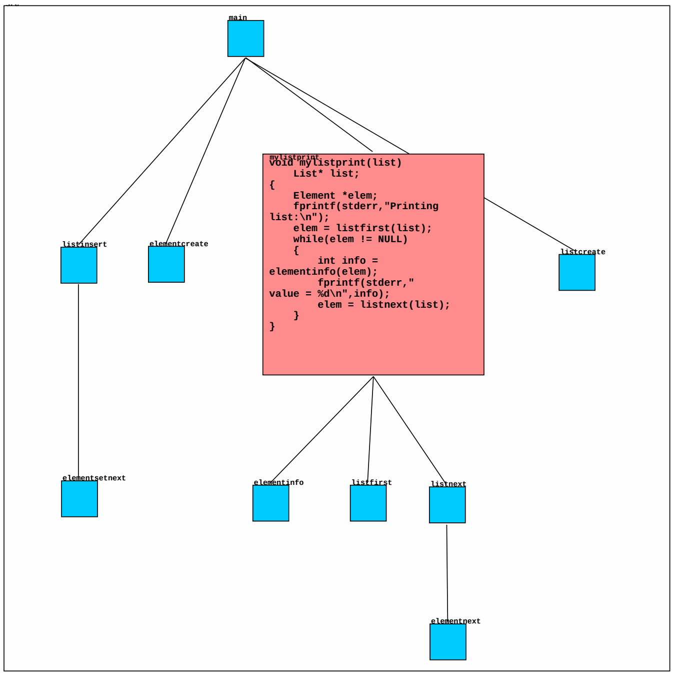


Fig. 8. Browsing software source code using SHriMP Views

6 Conclusions

This paper introduced the SHriMP layout adjustment algorithm suitable for uniformly resizing nodes when requests for more screen space are made. It preserves straightness of lines and graph topology in the adjusted views. Moreover, the SHriMP algorithm is flexible in its distortion technique and can be changed to suit different graph layouts. Several variants were presented 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.

This algorithm, 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. This approach can also be applied to the node disjointness problem.

We are currently investigating whether or not these methods will be useful in the design phase of graph layouts such as those found in hierarchical petri nets and user interfaces. Future work involves investigating other layout strategies and to analyze the strategies presented in this paper to determine which classes of proximity relations and orthogonal relations are preserved using the different strategies.

Acknowledgments The authors gratefully thank Bryan Gilbert, James McDaniel and Minou Bhargava for editing suggestions. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

1. G. Di Battista, P. Eades, R. Tamassia, and I. Tollis, *Algorithms for graph drawing: An annotated bibliography*, Comput. Geom. Theory Appl., 4 (1994), pp. 235–282.
2. 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, 1994.
3. K. M. Fairchild, S. E. Poltrock, and G. W. 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, 1988.
4. G. Furnas, *Generalized fisheye views*, in Proceedings of ACM CHI'86, Boston, MA, April 1986, pp. 16–23.
5. D. Harel, *On visual formalisms*, Communications of the ACM, 31(5) (May 1988).
6. D. Hearn and M. P. Baker, *Computer Graphics*, Prentice Hall, 1986.
7. T. R. Henry and S. E. Hudson, *Interactive graph layout*, in UIST, Hilton Head, South Carolina, November 11-13, 1991, pp. 55–64.
8. K. Misue, P. Eades, W. Lai, and K. Sugiyama, *Layout adjustment and the mental map*, Journal of Visual Languages and Comput., 6(2) (1995), pp. 183–210.
9. 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, September 1991, pp. 834–838.
10. H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, *A reverse engineering approach to subsystem structure identification*, Journal of Software Maintenance: Research and Practice, 5(4) (December 1993), pp. 181–204.
11. E. Noik, *A space of presentation emphasis techniques for visualizing graphs*, in Proceedings of Graphics Interface '94, (Banff, Alberta), May 1994, pp. 225–233.
12. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
13. M. Sarkar and M. Brown, *Graphical fisheye views*, Communications of the ACM, 37(12) (December 1994).
14. M. Sarkar, S. Snibbe, O. Tversky, and S. Reiss, *Stretching the rubber sheet: A metaphor for viewing large layouts on small screens*, in User Interface Software Technology, 1993, November 3-5, 1993, pp. 81–91.
15. M.-A. D. Storey and H. A. Müller, *Manipulating and documenting software structures using shrimp views.* To appear in *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95), Opio (Nice), France*, October 16-20, 1995.
16. S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller, *Programmable reverse engineering*, International Journal of Software Engineering and Knowledge Engineering, 4 (1994).
17. E. R. Tufte, *Envisioning Information*, Graphics Press, 1990.
18. K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey, *Structural redocumentation: A case study*, IEEE Software, 12 (1995), pp. 46–54.

This article was processed using the LaTeX macro package with LLNCS style