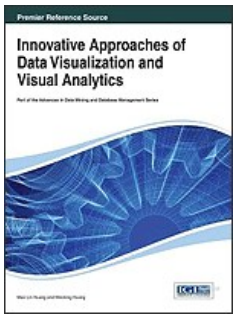

Innovative Approaches of Data Visualization and Visual Analytics

Chapter 16: A Framework for Developing Diagram Applications

Wei Lai,
Faculty of Information and Communication Technologies, Swinburne University of Technology
Australia

Weidong Huang,
CSIRO ICT Centre, Sydney
Australia

ABSTRACT

This chapter presents a framework for developing diagram applications. The diagrams refer to those graphs where nodes vary in shape and size used in real world applications, such as flowcharts, UML diagrams, and E-R diagrams. The framework is based on a model the authors developed for diagrams. The model is robust for diagrams and it can represent a wide variety of applications and support the development of powerful application-specific functions. The framework based on this model supports the development of automatic layout techniques for diagrams and the development of the linkage between the graph structure and applications. Automatic layout for diagrams is demonstrated and two case studies for diagram applications are presented.

1 INTRODUCTION

A graph typically consists of a set of nodes and a set of edges where a node is drawn as a point and an edge is drawn as a line. We are more interested in diagrams which refer to practical graphs. In a practical graph, nodes (e.g. boxes and circles) representing objects take some spaces. They are not abstract points in a classical graph. Diagrams are commonly used to model relations in information systems and software engineering. Examples are data flow diagrams, state transition diagrams, flow charts, PERT charts, organization charts, Petri nets, entity-relationship diagrams, and UML diagrams.

A number of graph drawing algorithms have been developed, these algorithms are listed in (Battista et al 1999). These algorithms produce aesthetically pleasing abstract graph drawings, where nodes are just points in the plane. In practice, nodes have many attributes; for example, in a UML diagram, a node has several textual labels in several fields. These attributes need to be represented graphically, for instance, as shapes, sizes, and colors. We call this kind of graph where nodes vary in shape and size a practical graph.

Most diagrams in applications are practical graphs (e.g. UML diagrams). Diagrams are widely used in information visualization. The motivation for information visualization is that relational information, once handled textually, is now commonly displayed and manipulated with diagrams. We call this kind of information visualization using diagram displays and manipulations *diagram applications*. Diagram applications can be divided into two categories:

1. Those which focus on translation of textual information into a diagram, such as translation of a program to a flow chart, and translation of URL relations to a web graph.
2. Those which focus on semantic interpretation of diagrams, such as interpretation of a flow chart to execute a program, code generation from a UML diagram, and using diagrams for data mining.

Both categories of applications need diagram layout interfaces. It is critical to have an efficient tool for creating diagram layout interfaces. Although there are some software tools (such as Visual Basic, Xlib, and Motif) for windows and GUI programming, they are very low-level and not efficient for interactive diagram applications. These tools support building those interface components, such as menus, scrolling bars, dialog boxes, and so on. However, they are not efficient to support building diagram layout interfaces.

Most existing CASE (Computer Aided Software Engineering) tools, such as Rational Rose (7), provide graphical editors for drawing UML diagrams. However, these tools were designed for UML diagram applications only and they cannot be used for various diagram applications. Also, these CASE tools do not efficiently support automatic diagram layout. Automatic layout can release the user from the time-consuming and detail-intensive chore of generating a readable diagram.

This chapter presents a framework for developing diagram applications. It includes the investigation of a robust model for diagrams to solve the problems mentioned above. This model can support automatic layout techniques for diagrams and the development the linkage between the diagram structure and applications.

The critical issue is that what kind of model is suitable to represent diagrams where nodes vary in size and shape and can support the development of diagram layout functions? Is current classical graph model sufficient in playing this role? Can current graph drawing algorithms be applied to diagram layout? In next section, the background is introduced for classical graph model and graph drawing algorithms, and the questions above will be answered.

2 BACKGROUND

Many graph drawing algorithms (Battista et al., 1999) have been developed for abstract graph automatic layout. They are based on the classical graph model, that is, G=(V, E), where V is a set of abstract nodes (points) and E is a set of edges (lines). However, there is little work have been done on automatic layout for practical graphs (i.e. diagrams) used in real world applications, especially for nodes varying in size and shape in practice.

The most difficult problem for diagram interfaces is *layout*-assigning a position for each node and a curve for each edge. The assignment must be chosen to make the resulting picture easy to understand and easy to remember. A good layout can be like a picture-worth a thousand words; a poor layout can confuse or mislead the user. This problem is called the *graph drawing problem*. A number of graph drawing algorithms have been developed. They produce aesthetically pleasing graph layouts (see Figure 1a). These algorithms can be applied to draw those graphs where the sizes of nodes are very small. This is because such algorithms were often originally designed for *abstract graphs* where nodes take up little or no space. However, in applications, the images of nodes are circles, boxes, diamonds and similar shapes, and may contain a considerable amount of text and graphics. Sometimes nodes may represent subgraphs, and may be quite unpredictable in size and shape. Applying such algorithms to *practical graphs* (i.e. diagrams) where nodes vary in shape and size may result in overlapping nodes and/or edge-node intersections (see Figure 1b).

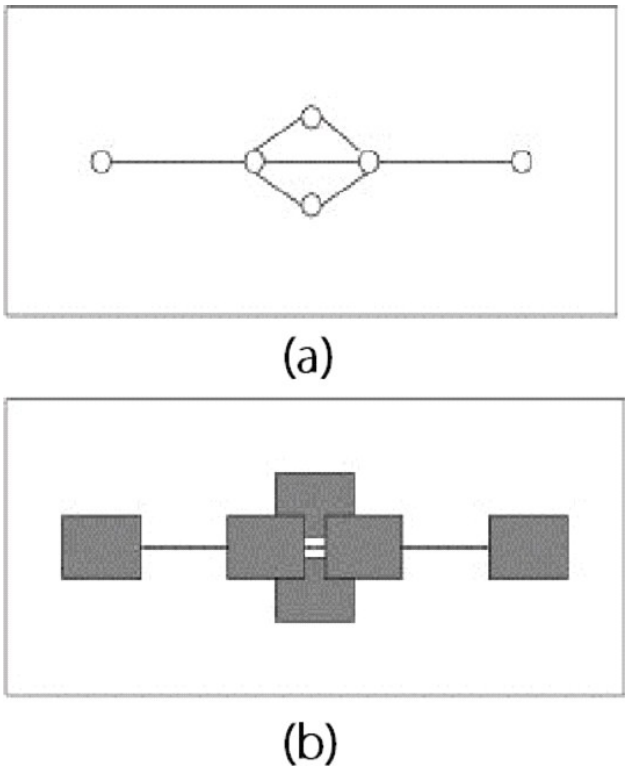

Figure 1: An example of overlapping nodes

Another example is illustrated in Figure 2; replacing the abstract nodes in Figure 2 (a) with rectangles gives the overlapping nodes in Figure 2 (b). But the layout we need should be Figure 2 (c). For practical graphs, the size of nodes should be taken into account in the development of automatic layout techniques. What kind of functions should be developed to suitable for diagram layout?

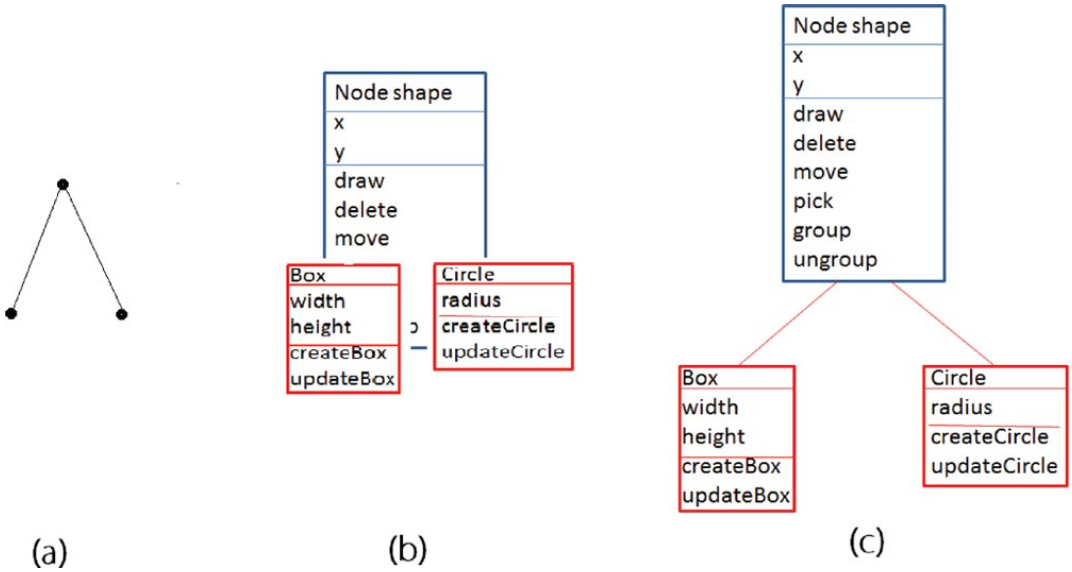

Figure 2: Layout of a class diagram

If we cannot make use of existing drawing algorithms for displaying diagrams in practice, we have to develop specific drawing approach for a

specific type of diagrams. For example, to display a UML diagram, a specific approach is developed (2012).

Based on the analysis above, it is obviously the classical graph model is not suitable for representing diagrams used in practice. A robust model for diagrams (layout and applications) should be investigated. In next section, the model for diagrams is presented.

3 A MODEL FOR DIAGRAMS

In this section, we present the model for diagrams and a generic approach for automatic diagram layout.

3.1 The Model

A robust structure for diagrams has been investigated. This structure can represent a wide variety of applications and support the development of powerful application-specific functions (including layout functions and the functions for the linkage between the graph structure and applications).

This kind of structure should include geometrical information for diagrams and support the development of automatic layout functions. We call this structure the *practical graph model*.

The model is combined with the approaches on generic graph grammars (Ong & Kurth, 2012; Rekers & Schurr, 1995) and cluster graphs (Huang & Lai, 2006). In order to represent a variety of diagrams in application, basic components of diagrams should be defined. These basic components can be used to construct compound components. A diagram component is defined as

$C = (r, L_r, P_r, I_r, E_r)$

which consists of a single *root* node r, a set of *inside* nodes I_r, and a set of relationships E_r between elements of I_r. The root node r *includes* the graph $G_r = (I_r, E_r)$.

If I_r is empty, the component is a basic component. Otherwise it is a compound component. For example, the root node r of the component *statement* is a rectangle with label "*if_statement*". Its inside node set I_r consists of an entry point node, an exit point node, a condition node, and two component nodes (corresponding to '*yes*' and '*no*' of the condition node); the set E_r is a set of relationships between the inside nodes.

L_r and P_r define a component's *logical part* and *physical part*. The logical part can be used to communicate with applications. The semantic connection between the application and the logical part can be set up. One connection is visualization in the application; for instance, the application data represented by syntactic rules in text can be translated into the logical part of the diagram component. Another is semantic interpretation of the diagram on the screen, which is achieved by semantically interpreting the logical part of the diagram component. For example, the mapping between a program in a specific language and the logical part can be achieved by invoking an application-specific function.

The purpose of the physical part is purely for illustration-it is only an image of reality. The semantic connection between the application and the physical part is rather loose. The application may or may not influence the physical part: the semantics of the application can be illustrated in the physical part, but the basic purpose of the physical part is to *clarify* meaning, not to *define* meaning. For example, a node image mapping function can easily be defined from geometric information for the node: position (x, y), size (w, h) and node image mode (like a box). In direct manipulation, when a node image is selected by the mouse, the corresponding node can be found by the position matched. The physical part defines a diagram's graphical appearance. For a diagram component, its physical part is related to a diagram layout function.

That is, diagram layout functions would operate on the diagram component's physical part; the application linkage mechanism would be related to diagram component's logical part.

The framework is based on this practical graph model. We have developed a prototype platform of the framework. This platform can allow the developers to interactively construct a diagram component by testing diagram layout/editing functions they developed.

3.2 Development of Automatic Layout for Diagrams

The development of layout functions should consider the node's shape and size. If a node's shape is a box, its size is decided by its width and height. If a node's shape is not a box, such as an oval, a circle, or a diamond, it can be bounded in a box (see Figure 3). That is, its bounding box decides its size. To draw a node's shape, a specific shape drawing function can be used. In the processing of solving overlapping nodes and edge-node intersections, a node's size is based on its bounding box.

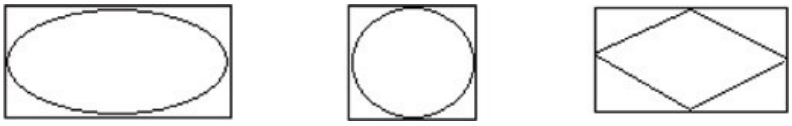

Figure 3: A node's size is based on its bounding box

Our approach to automatic layout for practical graphs has two steps, as follows.

1. Apply an abstract graph layout function to the graph.
2. Use post-processes to avoid overlaps of node images and edge-node intersections by rearranging the graph layout generated by step 1.

Step 1 is to make use of existing abstract graph drawing algorithms which are available in (Battista et al., 1999).

Step 2 refers to rearrange and adjust the graph generated by step 1. The rearrangement should also preserve the original "structure" of the graph. In the other words, the user's mental map of the graph (Eades et al., 1991, 1995) should be preserved. Step 2 needs to address the problems of overlapping nodes and edge-node intersections.

The critical part of our approach is to remove overlapping nodes and edge-node intersections. We use the techniques (Lai & Eades, 1992, 2002) for removing overlaps of node images and edge-node intersections. We have experimented with these techniques using many sets of overlapping nodes and found that it is quite effective. An example of using these techniques is shown below. Figure 4 shows a graph layout generated by an abstract graph layout algorithm - the "spring" algorithm (Eades, 1984). Figure 5 shows the result of replacing the nodes with rectangles, which gives not only the overlapping nodes but edge-node intersections. Figure 6 is the result of applying the force-scan algorithm (Lai & Eades, 2002) for removing overlaps of node images and edge-node intersections.

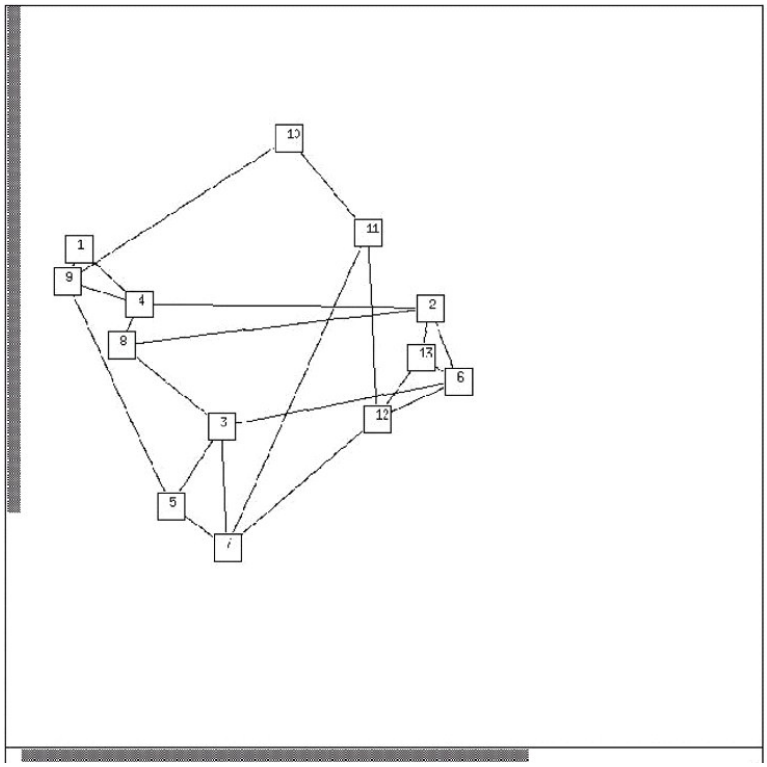

Figure 4: An abstract graph layout

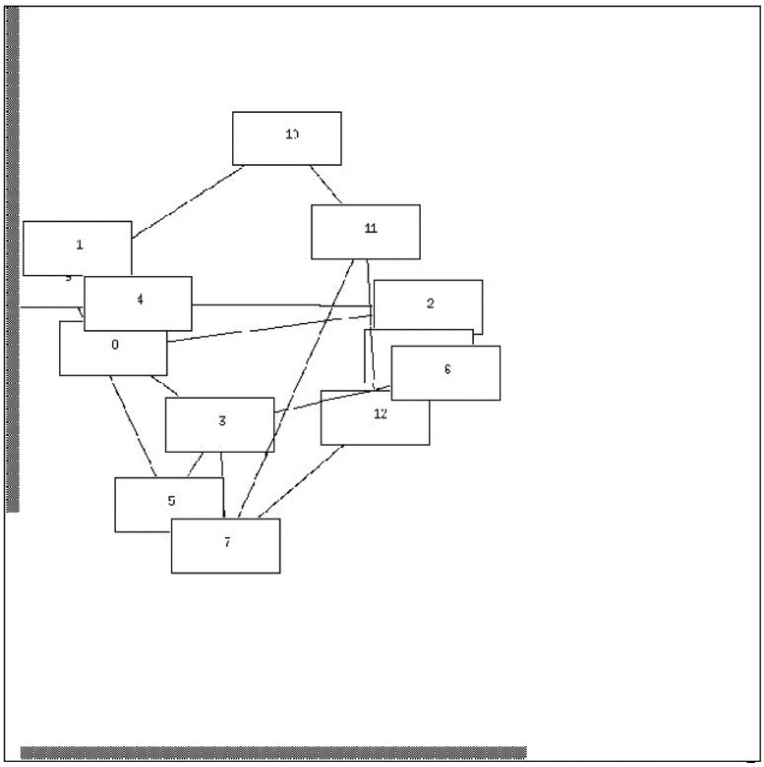

Figure 5: A practical graph

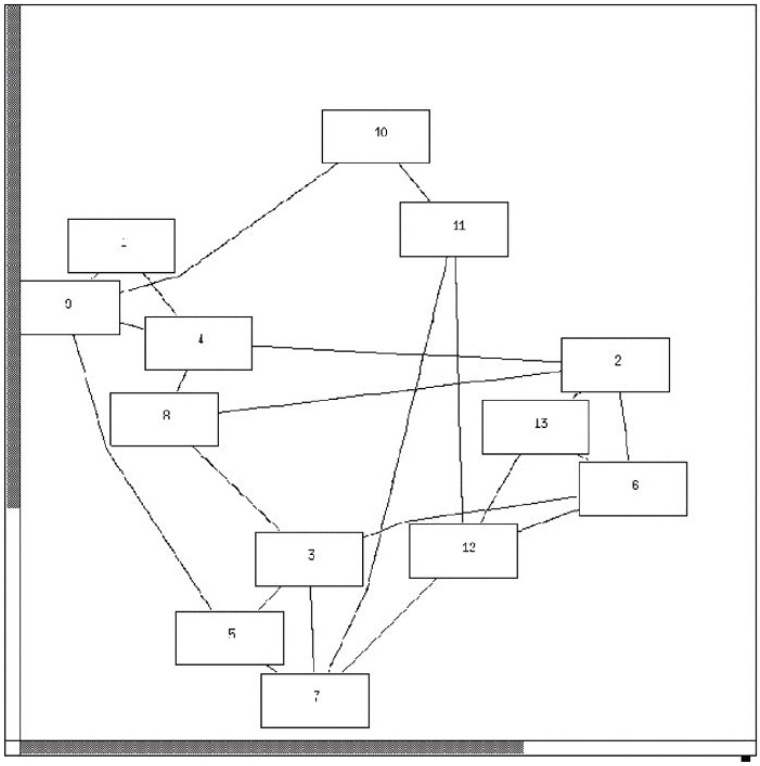

Figure 6: Layout adjustment

Layout adjustment and *Dynamic methods* for diagram editing have been studied (Eades et al., 1991, 1995; Lai & Eades, 1992, 2002). A change in the combinatorial graph (made either by the user or the application) should induce a change in the layout on the screen. However, the layout should not change so much as to disturb the user's "mental map" of the diagram. Some mathematical models of "mental map" have been defined and some algorithms (Lai & Eades, 1992, 2002) developed for rearranging a diagram while preserving its "mental map". We have developed some dynamic methods for practical graph layout without changing the "mental map". Some efficient algorithms have been developed for removing overlapping nodes and edge-node intersections.

The user interface includes abstract graph drawing algorithms and layout adjustment algorithms for removing overlapping nodes and edge-node intersections. We mainly focus on the development of automatic layout techniques for practical graphs based on this platform.

4 TWO CASE STUDIES

As mentioned above, a platform has been developed based on the framework. We use this platform to do some case studies for applications, as this platform can also allow the developers to develop functions for the linkage between the graph structure's logical part and applications. For example, different applications can be translated into abstract objects and relations in the graph structure by using different language parsers. To test and evaluate this framework for diagrams, two case studies have been developed.

4.1 Web Graph Visualization and Navigation

Each URL in the webspace is treated as a node and a link between two URLs is an edge. The graph using these nodes and edges is called web graph.

A specific function (web site parser) is developed to support the construction of Web sub-graphs by communicating with Web sites over the Internet. It can quickly search the entire neighbourhood of the focused node to form a Web sub-graph.

The Web site parser analyses the HTML file of the Web site corresponding to the focused node and extracts the hyperlinks embedded in the Web site to form nodes and edges for the Web sub-graph. To reduce the complexity of the Web graph, an information filter is developed to remove unnecessary information (edges and nodes) generated by the parser and only retains the essential part which the user requires. The web graph is mapped to the logical part of the practical graph model. The physical part of the model supports the web graph display. Actually, only the sub-graph is displayed based on the user's focus. The Web sub-graph user interface maintains the user's orientation for Web exploration and it also reduces the cognitive effort required to recognise the change of views. This is done by connecting successive displays of the subset of the Web graph and by smoothly swapping the displays via animation according to the user's navigation of the web graph. Figure 7 shows an example of a web site and its web sub-graph.

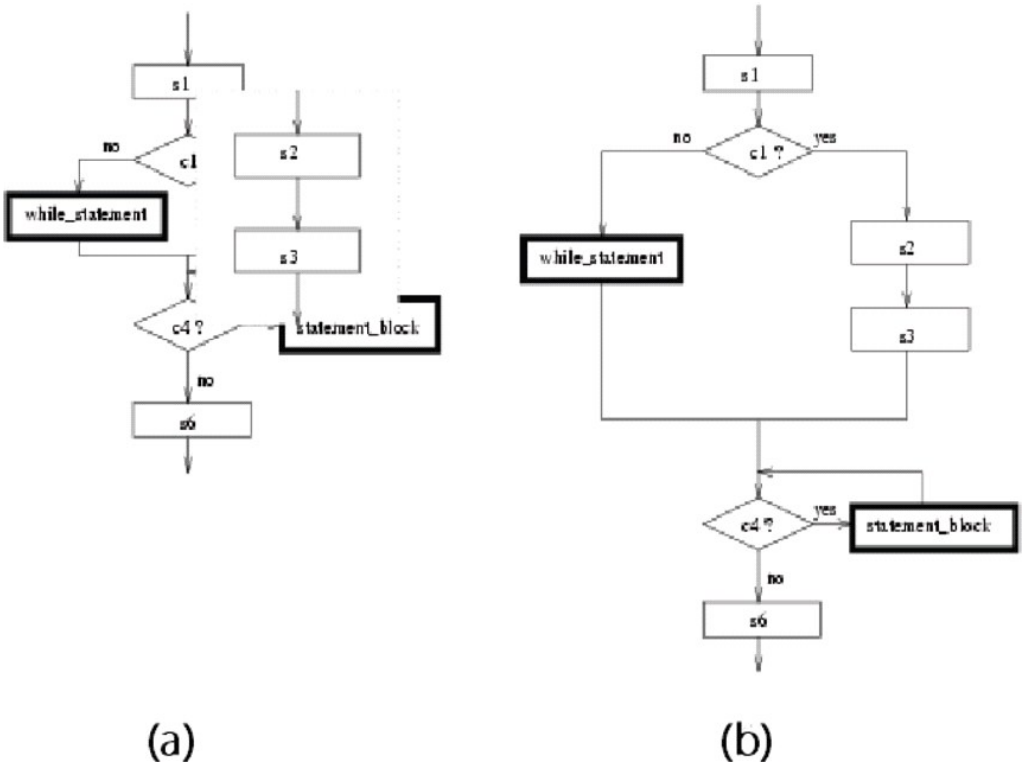

Figure 7: A web site and its web sub graph

4.2 Visualization of Program Flowcharts

A source code parser is developed for converting a Pascal program source code to the flowchart drawing structure. The parser scans the source code from top to bottom and convert each statement to corresponding flowchart component (i.e. the graph's logic part). The graph's physical part is responsible for its layout.

For example, suppose that the user wants to expand the contents of node, the right statement_block within the if_statement, to see more details. After a mouse action, the function DynamicLayout is invoked to operate on this node (statement_block). First the graph included by this node is displayed and it replaces the node statement_block in the diagram (see Figure 8a); then the whole diagram (the graph included by statement_block) is rearranged (see Figure 8b) - this is achieved by the function DynamicLayout.

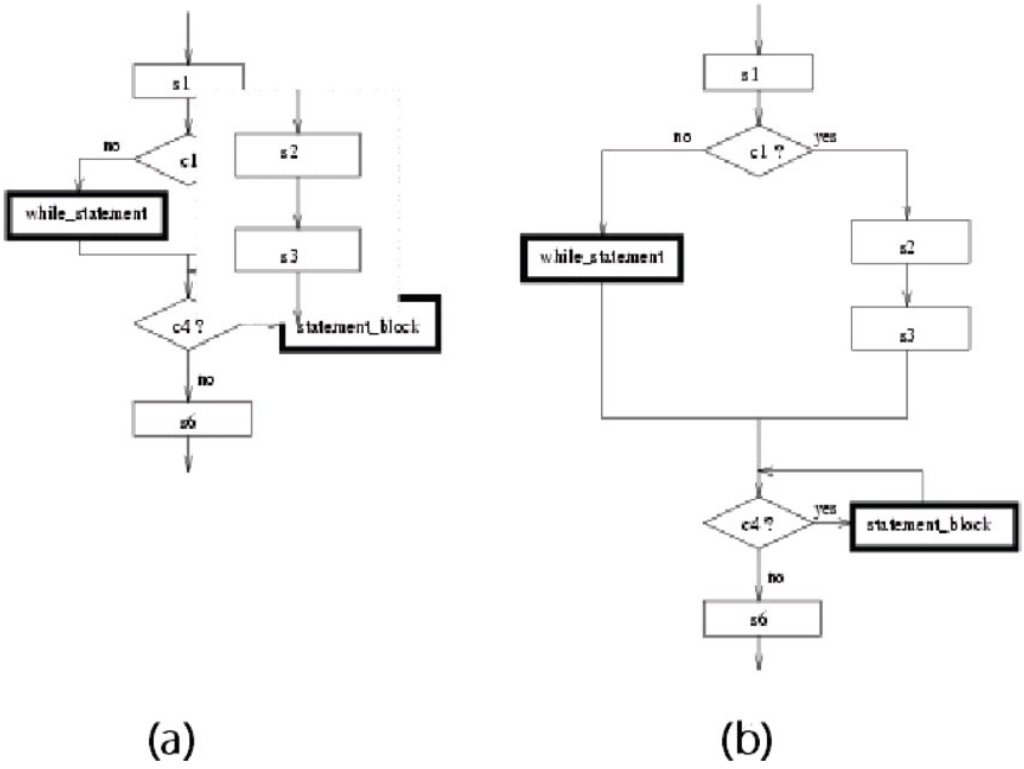

Figure 8: An example of supporting dynamic layout

Figure 9 shows the flowchart visualization for Pascal is in action.

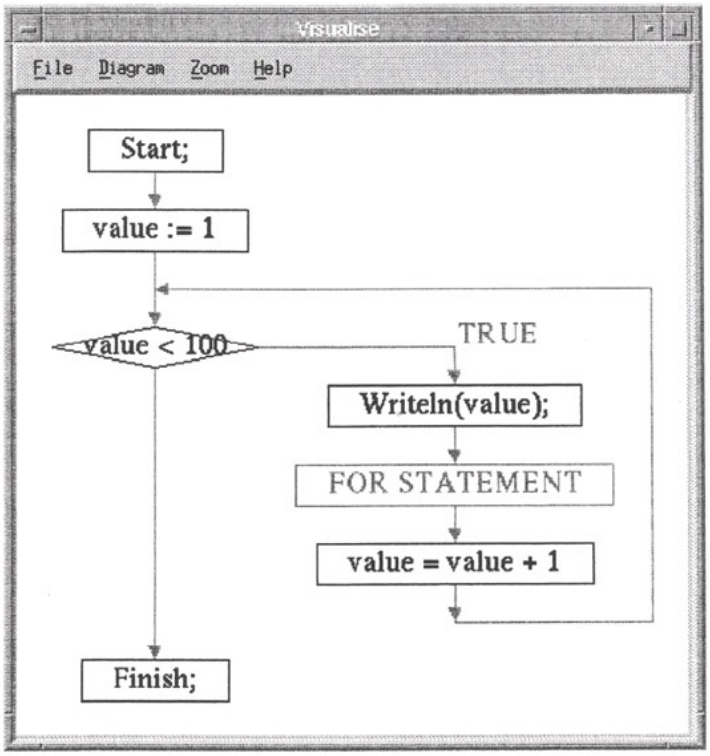

Figure 9: An example of flowchart visualization

5 FUTURE WORK

More efficient methods will be further investigated for the layout of diagrams used in real world applications and more case studies will be developed. In practically, an application will be developed for UML diagrams visualization and code generation. Through this case study, we can test the semantic interpretation of the practical graph model. That is, the logical part can be used as a linkage between UML diagrams and code generation.

REFERENCES

Battista, G. D., Eades, P. R., Tamassia, R., & Tollis, I. (1999). *Graph drawing: Algorithms for the visualization of graphs*. Upper Saddle River, NJ: Prentice Hall.

Eades, P. (1984). A heuristic for graph drawing. *Congressus Numerantium*, *42*, 149–160.

Eades, P., Lai, W., Misue, K., & Sugiyama, K. (1991). Preserving the mental map of a diagram. [Sesimbra, Portugal: Academic Press.]. *Proceedings of COMPUGRAPHICS*, *91*, 34–43.

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

Huang, X., & Lai, W. (2006). Clustering graphs for visualization through node similarities. *Journal of Visual Languages and Computing*, (17): 225–253. doi:10.1016/j.jvlc.2005.10.003

Lai, W., & Eades, P. (1992). Algorithms for disjoint node images. *Australian Computer Science Communications*, *14*(1), 253–265.

Lai, W., & Eades, P. (2002). Removing edge-node intersections in drawings of graphs. *Information Processing Letters*, *81*, 105–110. doi:10.1016/S0020-0190(01)00194-6

Ong, Y., & Kurth, W. (2012). A graph model and grammar for multi-scale modelling using XL. In *Proceedings of 2012 IEEE International Conference on Bioinformatics and Biomedicine Workshops* 1-8. Washington, DC: IEEE Press.

Rekers, J., & Schurr, A. (1995). A graph grammar approach to graphical parsing. In *Proceedings of IEEE Symposium on Visual Languages (VL'95)*. Darmstadt, Germany: IEEE Press.

Storrle, H. (2012). On the impact of layout quality to understanding UML diagrams: Diagram type and expertise. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing,* 49-56. Innsbruck, Austria: IEEE Press.