AI-MAS Group

Artificial Intelligence and Multi-Agent Systems Laboratory

University "Politehnica" of Bucharest

Master of Science in Artificial Intelligence

Visualization of Context Graphs - JUNG and Zest

1^{st} *MsC Research Report*

Nihal Ablachim

MsC Research Title: Using context graphs to help users in their daily activities
Supervisor: **Ş.l. dr. ing. Andrei Olaru**
AI-MAS Laboratory, Computer Science Department, University "Politehnica" of Bucharest

February 2013

Contents

1 Introduction

The context of a user can be represented as a graph $G = (V, E)$ in which the values stored in vertices and edges are Strings or URI's that shows relations, concepts, people's name, place names etc. This kind of graphs are called *context graphs*.

The purpose of the main research is to develop an application that allows the user to edit his context graph and that automatically detects the situation of the user and proposes appropriate action, based on pre-existing context graphs.

In this circumstances, a starting point could be to implement a graphical interface which permits the user to visualize and dinamically edit his context graphs. Since there are already free and open-source softwares that provide the manipulation and visualization of the graphs there is no need to reinvent the wheel again and implement another framework but to make use of what already exists. In this paper two of the most common graph visualization toolkits will be described: *JUNG* and *Zest* each toolkit with its own features.

The document is divided into two main parts each part representing one of the two above mentioned toolkits. For each of the toolkits a brief introduction is made and then a subsection is allocated for each of the features. In subsection *Graphs, Vertices and Edges* it is described the fundamental properties and operations of graph, vertex, and edge objects. Subsection *Graph Visualization* outlines each toolkit's architecture for creating graph visualizations. In *Getting Interactive* subsection is presented how to dynamically change the graphs and subsection *Working with algorithms* lists some of the algorithms used for graphs. In the last section, conclusions are drawn.

2 JUNG Visualization Toolkit

2.1 Introduction to JUNG

JUNG(Java Universal Network/Graph Framework) is a software library that provides a language for the modeling, analysis, and visualization of data that can be represented as a graph or network. It is written in Java, which allows JUNG-based applications to make use of the extensive built-in capabilities of the Java API.

The JUNG architecture is designed to support a variety of representations of entities and their relations, such as directed and undirected graphs, multi-modal graphs, graphs with parallel edges, and hypergraphs.

JUNG includes implementations of algorithms from different fields such as graph theory, data mining, and social network analysis, including clustering, filtering, random graph generation, blockmodeling, calculation of network distances and flows, and a wide variety of metrics (PageRank, HITS, betweenness, closeness, etc.).

JUNG also provides a visualization framework that makes it easy to construct tools for the interactive exploration of data. Users can make use of one of the layout algorithms provided, or use the framework to create their own custom layouts.

2.2 Graphs,Vertices and Edges

The basic type implemented in JUNG is the graph. The base interface for all JUNG graph types, *Hypergraph* defines a graph to be a container of vertices and edges, with several methods for accessing and modifying these sets, for defining vertex and edge constraints. The *Graph* interface extends the *Hypergraph* interface, and is specialized for graphs whose edges connect exactly two vertices. *Graph* in turn has subinterfaces for graphs that have only directed edges, only undirected edges, or that are k-partite. The graph type hierarchy is shown in Figure 10 from the Appendix section.

For visualizing and editing context graphs, *Graph* interface with its subinterfaces

will be used which is defined in *edu.uci.ics.jung.graph* package. So, in what follows *Graph* interface will be described.

2.2.1 Basic Properties and Operations

Graphs, vertices and edges each have several properties that can be extracted, and operations that they can perform. These properties and operations will be presented in this subsection.

JUNG graphs are analogous to Java collections (such as List, Set, Map, and so on) in the way that just as collections may specify the type of their elements in the declaration (e.g., Set<Integer> or Map<String, SomeClass>), JUNG graph declarations may specify the type of each of their element categories, that is, vertices and edges.

As mentioned above JUNG has an interface $Graph < V, E >$ which extends the $Hypergraph < V, E >$ interface and allows basic operations we can perform on a graph such as:

- adding and removing vertices and edges to a graph.
- getting collections of all edges and vertices in a graph.
- getting information concerning the endpoints of an edge in the graph.
- getting information concerning vertices in a graph including various degree measures(indegree,out-degree) and predecessor and successor vertices.

The $Graph < V, E >$ interface allows working with both directed and undirected edges, this being possible while adding edges (see next subsection). But $Graph < V, E >$ also has as subinterfaces: $DirectedGraph < V, E >$, $Forest < V, E >$, $KPartiteGraph < V, E >$, $Tree < V, E >$, $UndirectedGraph < V, E >$ which allow working with only directed, undirected, k-partite, tree or forest graphs as the name of the interfaces specifies.

In JUNG the same vertices and edges can appear in more than one graph. In our case since we are dealing with context graphs that have in common nodes this can be really useful. Also vertex and edge objects must be unique to a graph: there cannot be two vertices, or two edges, such that vertex1.equals(vertex2), or edge1.equals(edge2), neither two vertices/edges can have the same labels.

Working with graphs with parallel edges is also possible in JUNG; this can be done using one of the classes in which names appear "multi"(e.g *SparseMultigraph*, *DirectedSparseMultigraph*, *UndirectedSparseMultipraph* etc.).

2.2.2 Creating, Adding and Removing

The simplest way to create a graph is by calling the constructor for the desired type of graph that is wanted, as in the following example:

```
Graph<Integer, String> g = new SparseMultigraph<Integer, String>();
```

Once the graph is created, vertices may be added to the graph:

```
g.addVertex((Integer) 1);
g.addVertex((Integer) 2);
g.addVertex((Integer) 3);
```

and once vertices exist, they may be connected by edges:

```
g.addEdge("Edge-A", 1, 2, EdgeType.DIRECTED);
g.addEdge("Edge-B", 2, 3);
```

Also if a vertex or an edge from a graph is no more useful we can remove it as follows:

```
g.removeVertex(1);
g.removeEdge("Edge-A");
```

Please note that if any removal operations are necessary, these should be done after all the addition operations are finished. For example if the following code is written:

```
g.addVertex((Integer) 1);
g.addVertex((Integer) 2);
g.addVertex((Integer) 3);
g.removeVertex(1);
g.addEdge("Edge-A", 1, 2);
g.addEdge("Edge-B", 2, 3);
```

it won't remove the vertex 1 because next in the code, an edge between 1 and 2 is added.

Removing an edge from a graph will not affect any other part of the graph but removing a vertex from a graph will remove all the incident edges to that vertex.

2.3 Graph Visualization

JUNG provides mechanisms for laying out and rendering graphs. The current renderer implementations use the Java Swing to display graphs. Some basics for visualizing a graph will be presented in this subsection.

2.3.1 The Basics for viewing a graph

In general, a visualization requires one of each of the following:

- A Graph to be visualized.
- A Layout, which takes the graph and determines the location at which each of its vertices will be drawn. This is achieved via JUNG's Layout interface and related classes (*edu.uci.ics.jung.algorithms.layout*). First of all the Layout interface's job is to return a coordinate location for a given vertex in a graph. It does this and more by extending the $Transformer < V, Point2D >$ interface, which when given a vertex v will return an object of type Point2D that encapsulates the vertex's (x, y) coordinates. This Layout interface provides additional mechanisms to initialize the locations of all vertices in a graph, set the location of a particular vertex, lock or unlock the position of a vertex, etc. JUNG provides many different layout algorithms for positioning the vertices of a graph(e.g. CircleLayout, RadialTreeLayout, SpringLayout, TreeLayout etc.). To visualize these layouts see figure 1.
- A (Swing) Component, which provides a drawing area upon which the data is rendered. The basic class for viewing graphs in JUNG is the BasicVisualizationServer class(*edu.uci.ics.jung.visualization*). This implements the JUNG $VisualizationServer < V, E >$ interface and inherits from Swing's JPanel class (javax.swing.JPanel).
- A Renderer, which takes the data provided by the Layout and paints the vertices and edges into the provided Component.Implementations of this class can set specific renderers for each element, allowing custom control of each.

In figure 2 is presented the code for the simplest way to visualize a graph(it is used the last version of the graph obtained above, after all the operations presented were performed on the graph) and in figure 3 can be seen the graph when run the code.

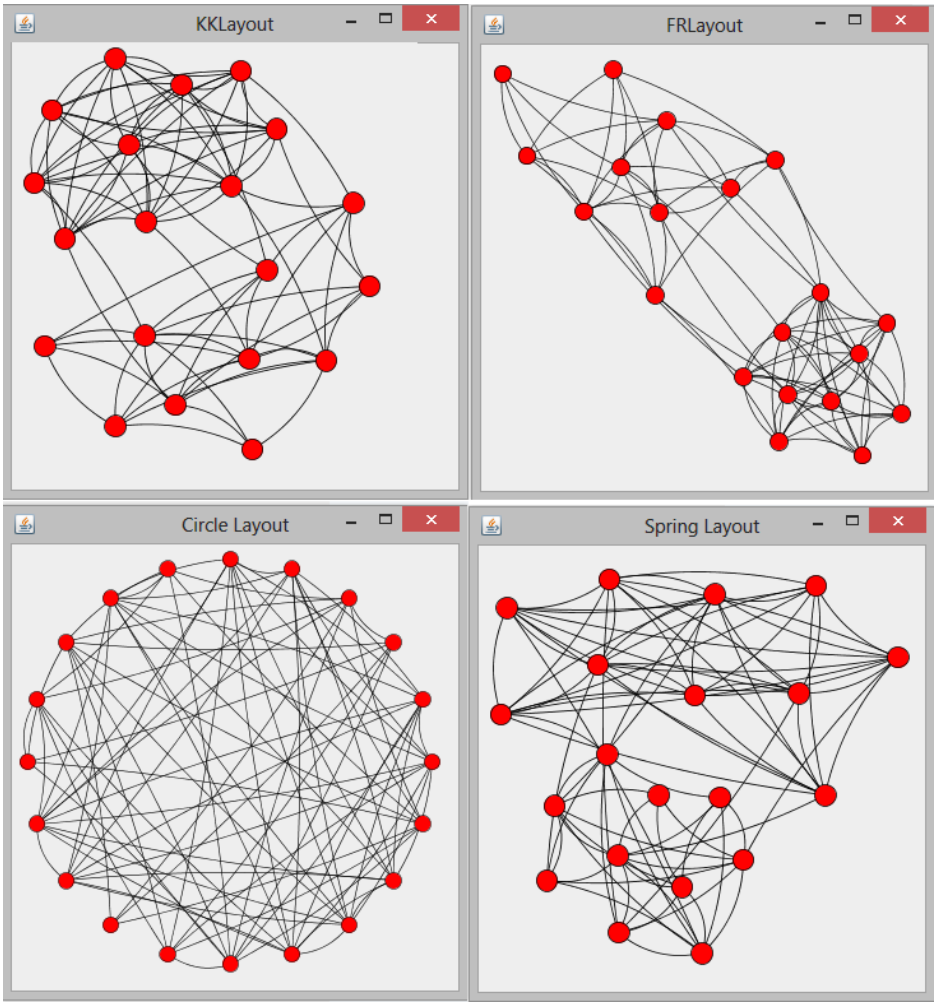

Figure 1: Diffrent type of layouts in JUNG

```
public static void main(String[] args) {
Layout<Integer, String> layout =
  new CircleLayout<Integer,String>(g);
layout.setSize(new Dimension(350, 350));
  // sets the initial size of the space
BasicVisualizationServer<Integer, String> vv =
  new BasicVisualizationServer<Integer, String>(layout);
vv.setPreferredSize(new Dimension(350, 350));
        // Sets the viewing area size
JFrame frame = new JFrame("Simple Graph View");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(vv);
frame.pack();
frame.setVisible(true);
}
```

Figure 2: Example of code for visualizing a graph

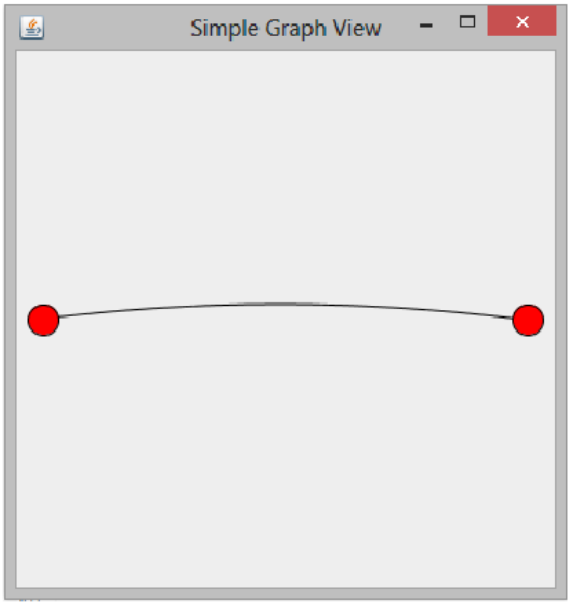

Figure 3: Simple view of a graph with JUNG

2.3.2 Painting and Labeling Issues for Vertices and Edges

The default implementation fetches the location of each vertex from the Layout, paints each one with the Renderer inside the Swing Component, and paints each edge as a straight line between its vertices. Users may customize this behavior as desired; JUNG includes utilities and support classes that facilitate such customization. The two new sets of interfaces/classes that one should be familiar with are the RenderContext (in edu.uci.ics.jung.visualization) and Renderer (from edu.uci.ics.jung.visualization.renderers).

The JUNG renderers are used to actually draw four different items: edges, edge labels, vertices, and vertex labels. Each BasicVisualizationServer (the interface used to actually display the graph) contains a RenderContext object that one can access to set various rendering parameter values such as vertex/edge color, vertex/edge label,vertex/edge shape etc; these being possible using *setVertexFillPaintTransformer()*, *setEdgeStrokeTransformer()*, *setVertexLabelTransformer()*, and *setEdgeLabelTransformer()*. As one could assume from the names of these methods each takes a Transformer class argument that converts an edge or vertex to the type of information needed by a renderer. Implementing the code from figure 11 from the Appendix section one can obtain the customized view of a graph from figure 4. This is a simple example, but of course that playing with a little bit of imagination one can obtain whatever graph display variation is wanted.

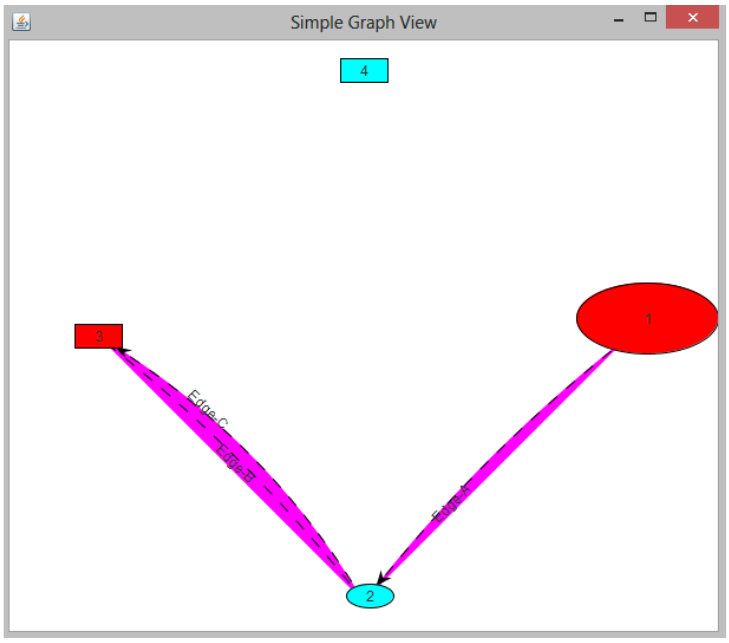

Figure 4: Customized view of a graph in JUNG

2.4 Getting Interactive

JUNG provides GUI features to let users interact with graphs in various ways. Most interactions with a graph will take place via the mouse. Since there are quite a number of conceivable ways that users may want to interact with a graph, JUNG has the concept of a modal mouse, i.e. a mouse that will behave in certain ways based on its assigned mode. Typical mouse modes include: picking, scaling (zooming), transforming (rotation, shearing), translating (panning), editing (adding/deleting nodes and edges), annotating etc. In this subsection these kind of mouse interactions will be presented.

In providing interactivity *DefaultModalGraphMouse* class from *edu.uci.ics.jung.visualization.control* can be used to provide a full set of picking and transforming capabilities. In addition to the code from figure 2 it will be used the derived class *VisualizationViewer* instead of *BasicVisualizationServer* that also provides mouse functionality and it will be added the following code:

```
DefaultModalGraphMouse gm = new DefaultModalGraphMouse();
gm.setMode(ModalGraphMouse.Mode.TRANSFORMING);
vv.addKeyListener(gm.getModeKeyListener());
vv.setGraphMouse(gm);
```

Running this code the default behavior is:

- Left mouse click and mouse drag in the graph window allows to translate(pan) the whole graph.
- Shift, left mouse click and drag in the graph window allows to rotate the graph.
- Control, left mouse click and drag in the graph window allows to shear the graph.
- Mouse wheel or equivalent allows to scale(zoom) the graph.

Also JUNG provides a couple extra ways to control the current mouse mode besides programmatically setting it ourselves. In this case it can be used a method based on key listeners which listen for buttons pressed or keys typed. In particular the *DefaultModalGraphMouse* provides a key listener that will change to translate mode if the user types a t and to picking mode if the user types a p. To get this functionality one only needs to add the following line of code to the code from figure 2 using the above mentioned code too:

```
vv.addKeyListener(gm.getModeKeyListener());
```

DefaultModalGraphMouse has a default behavior as presented previously, but in JUNG one can make its own behavior of mouse, for example while viewing the graph it is wanted only to be able to translate or zoom the graph(no picking of vertices and no rotating or skewing of the graph). For this to happen JUNG provides the concept of mouse plugins that do the heavy lifting for a particular purpose such as picking, translating, zooming, editing, etc. For the above example just translation and zooming is wanted so it can be used the *TranslatingGraphMousePlugin* and *ScalingGraphMousePlugin* (*edu.uci.ics.jung.visualization.control*) classes. These classes are used together with *PluggableGraphMouse* object from *edu.uci.ics.jung.visualization.control*. So, the idea is to choose the plugins wanted then add them to a mouse that is defined as being *PluggableGraphMouse*. In the main method from figure 2 there is only need of the following four new lines of code:

```
PluggableGraphMouse gm = new PluggableGraphMouse();
gm.add(new TranslatingGraphMousePlugin(MouseEvent.BUTTON1_MASK));
CrossoverScalingControl cs=new CrossoverScalingControl();
gm.add(new ScalingGraphMousePlugin(cs, 0, 1.1f, 0.9f));
```

If run the code, it can be seen that translating and zooming work but there is no longer rotation and skewing nor can be switched to the picking mode.

2.5 Working with Algorithms

JUNG includes implementations of algorithms from different fields such as graph theory, data mining, and social network analysis, including clustering, filtering, random graph generation, blockmodeling, calculation of network distances and flows, and a wide variety of metrics (PageRank, HITS, betweenness, closeness, etc.). The algorithms from JUNG that can be useful for working with context graphs will be presented in this section.

2.5.1 Paths problem solving algorithms

The algorithms that JUNG provides for finding the shortest paths within graphs are stored in *edu.uci.ics.jung.algorithms.shortestpath* package. These algorithms include: *DijkstraShortestPath*(Dijkstra (1959)), which calculates the length of the shortest paths from a specified vertex to other vertices in the same graph; *UnweightedShortestPath* that computes the shortest path distances for graphs whose edges are not weighted(using BFS);*BFSDistanceLabeler*, which labels each vertex in a graph with the length of the shortest unweighted path from a specified vertex in that graph.

2.5.2 Transformation algorithms

Sometimes it can be necessary to convert a graph of one type to another; this can happen in a few different circumstances. For example:

- Certain algorithms operate only on directed (or undirected) graphs. The *DirectionTransformer* class(*edu.uci.ics.jung.algorithms.transformation*) can transform any *Graph* into either an *DirectedGraph* or an *UndirectedGraph*.
- The process that resulted in the creation of a graph may not have identified all the details of the graph type; for example, a graph may be a k-partite graph in terms of its connectivity, but not have been created as an implementation of KPartiteGraph. The KPartiteSparseGraph class can construct a KPartiteSparseGraph that is a copy of an existing Graph, given an appropriate set of partition specifications that are known to apply to the graph. (That is, the original graph is not modified structurally; this construction only works if the original graph is actually k-partite.)

3 Zest Visualization Toolkit

3.1 Introduction to Zest

Zest Visualization Toolkit is a set of visualization components built for Eclipse. It has been developed in SWT / Draw2D. Zest supports the viewer concept from JFace Viewers and therefore allows to separate the model from the graphical representation of the model.

3.2 Graphs, Vertices and Edges

For visualizing and editing context graphs, Eclipse Zest has the following components(in *org.eclipse.zest.core*):

- GraphNode - Node in the graph with the properties
- GraphConnections - Arrow / Edge of the graph which connections to two nodes
- GraphContainer - Use for a graph within a graph
- Graph - holds the other elements (nodes, connections, container)

In figure 5 is shown the simplest way to create a graph in Zest; this is done by declaring the graph as a *Graph* type and then calling the constructor for *Graph*. Since the graph is created then vertices may be created too, by calling the constructor for *GraphNode* class in which should be specified the graph which belongs to each vertex. And finally these vertices can be connected by edges, declared as being of type *GraphConnection*. In the constructor of *GraphConnection* should be specified the source vertex and the destination vertex(in this order) and the type of edge wanted(directed or undirected).

```
1  // Graph will hold all other objects
2     graph = new Graph(parent, SWT.NONE);
3     // Now a few nodes
4     GraphNode node1 = new GraphNode(graph, SWT.NONE, "1");
5     GraphNode node2 = new GraphNode(graph, SWT.NONE, "2");
6     GraphNode node3 = new GraphNode(graph, SWT.NONE, "3");
7     GraphNode node4 = new GraphNode(graph, SWT.NONE, "4");
8     GraphNode node5 = new GraphNode(graph, SWT.NONE, "5");
9     GraphNode node6 = new GraphNode(graph, SWT.NONE, "6");
10    GraphConnection c1=new GraphConnection(graph, ZestStyles.NONE, node1,node2);
11    c1.setText("2");
12    GraphConnection c2=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node1,node6);
13    c2.setText("1");
14    GraphConnection c3=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node1,node4);
15    c3.setText("4");
16    GraphConnection c4=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node2,node3);
17    c4.setText("1");
18    GraphConnection c5=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node2,node5);
19    c5.setText("1");
20    GraphConnection c6=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node4,node3);
21    c6.setText("2");
22 }
```

Figure 5: Example of code for creating a graph in Zest

Graphs, vertices and edges each have several properties that can be extracted, and operations that they can perform.

As above mentioned, Zest provide working with both directed and undirected edges, this being possible while declaring the edges (see the code from figure 5).

In Zest the same vertices and edges can not appear in more than one graph. For example, in code from figure 5, *node6* is declared as a *GrahNode* and being part of *graph*, first it can not be declared another *node6*(because there is already one) nor to declare it as being part from another graph. Instead, in Zest two vertices and/or edges can have the same labels.

Working with graphs with parallel edges is not possible in Zest; in code from figure 5 *c2* was declared as being an edge which connects node1 with node2, if one may wish to add another edge between the two nodes, for example *c10*, Zest will take in consideration the last edge between these nodes.

If a vertex or an edge from a graph is no more useful we can remove it as follows:

```
node3.dispose();
c3.dispose();
```

Same as in JUNG, if any removal operations are necessary, these should be done after all the vertices are declared and all the connections between vertices are established.

Removing an edge from a graph will not affect any other part of the graph but removing a vertex from a graph will remove all the incident edges to that vertex.

TreeLayoutAlgorithm	Graph is displayed in the form of a vertical tree
HorizontalTreeLayoutAlgorithm	Similar to TreeLayoutAlgorithm but layout is horizontal
RadialLayoutAlgorithm	Root is in the center, the others nodes are placed around this node
GridLayoutAlgorithm	
SpringLayoutAlgorithm	Layout the graph so that all connections should have approx. the same length and that the edges overlap minimal
HorizontalShift	Moves overlapping nodes to the right
CompositeLayoutAlgorithm	Combines other layout algorithms, for example HorizontalShift can be the second layout algorithm to move nodes which were still overlapping if another algorithm is used

Figure 6: Zest Layout Manager

3.3 Graph Visualization

The Zest project contains a graph layout package which can be used independently. The graph layout package can be used within existing Java applications (SWT or AWT) to provide layout locations for a set of entities and relationships. So, Eclipse Zest provides graph layout managers. A graph layout manager determines how the nodes (and the edges) of a graph are arranged on the screen. In figure 6 are presented the layout managers of Zest.

The simplest way to visualize a graph in Zest is by declaring an instance of one of the layouts from figure 6 and then calling the method setLayoutAlgorithm() of the graph. So adding the below lines to the code from figure 2 one can be able to visualize a graph with Zest framework. Figure 7 shows the graph when run the code.

```
SpringLayoutAlgorithm SLA=
  new SpringLayoutAlgorithm(LayoutStyles.NO_LAYOUT_NODE_RESIZING);
graph.setLayoutAlgorithm(SLA, true);
```

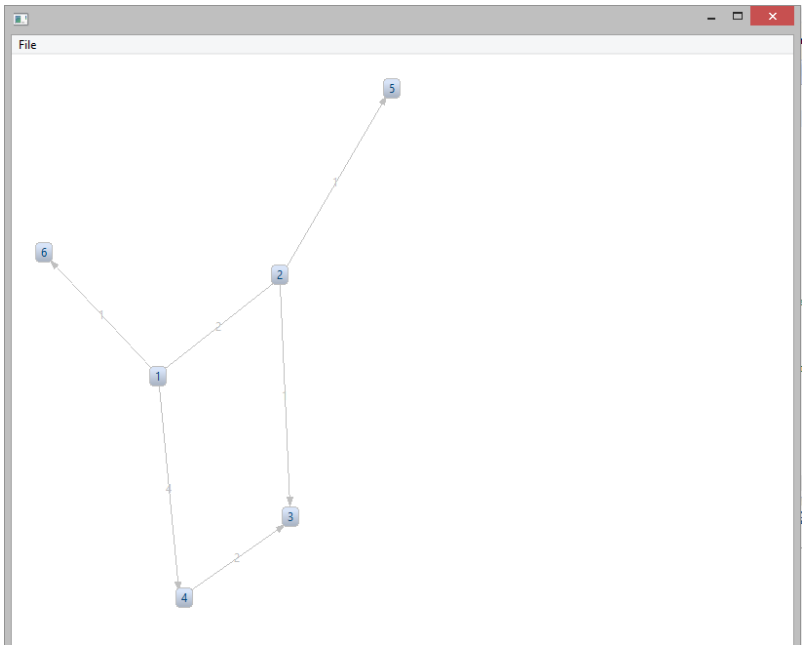

Figure 7: Simple view of a graph with Zest

3.3.1 Painting and Labeling Issues for Vertices and Edges

The default implementation fetches the location of each vertex from the Layout, paints each one with the background color as being a medium blue, and then paints each edge as a straight line between its vertices. Users may customize this behavior as desired; Zest supports methods that facilitate such customization.

In Zest each of the *GraphNode* and *GraphConnection* have methods that permit to user to set the desired characteristics(colour,shape,size etc.) for the graph's vertices and edges. Adding the code from figure 8 to the code from figure 5 one can obtain the customized view of a graph from figure 9. One more time, this is a simple example, but of course that playing with a little bit of imagination one can obtain whatever graph display variation is wanted.

By default, Zest layout algorithms (see Subsection Layout Algorithms) shrink the size of each node to a square based upon the overall size of the window. To override this behavior, it can be passed

```
LayoutStyles.NO_LAYOUT_NODE_RESIZING
```

when specifying the layout algorithm. Once this statement is added, the nodes are sized based upon their content rather than the overall size of the window. The background color for each node is a medium blue, in the example provided in figure

```
    c1.setLineColor(parent.getDisplay().getSystemColor(SWT.COLOR_CYAN));
    c4.setWeight(4);
    node1.setBackgroundColor(parent.getDisplay().getSystemColor(SWT.COLOR_DARK_YELLOW));
    node1.setBorderWidth(10);
    node2.setBackgroundColor(parent.getDisplay().getSystemColor(SWT.COLOR_CYAN));
    node2.setForegroundColor(parent.getDisplay().getSystemColor(SWT.COLOR_DARK_RED));
    node3.setBackgroundColor(parent.getDisplay().getSystemColor(SWT.COLOR_RED));
    node3.setLocation(0, 0);
    node4.setBackgroundColor(parent.getDisplay().getSystemColor(SWT.COLOR_CYAN));
    node5.setBackgroundColor(parent.getDisplay().getSystemColor(SWT.COLOR_MAGENTA));
    node6.setBackgroundColor(parent.getDisplay().getSystemColor(SWT.COLOR_YELLOW));
```

Figure 8: Example of code for customize a graph in Zest

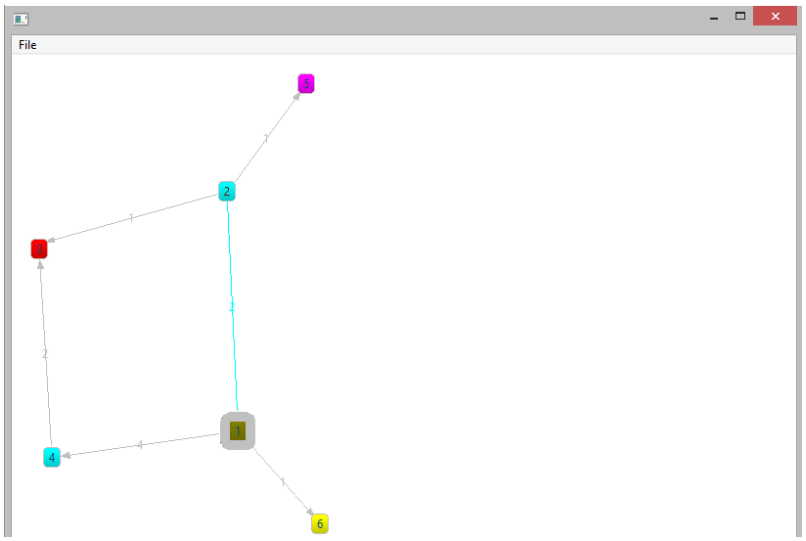

Figure 9: Customized view of a graph in Zest

8 the foreground text color of *node2* we set to be dark red rather than dark blue, or the background color for each node it was changed to different colors.

3.4 Layout Algorithms

Zest has only layout managing algorithms, unlike JUNG which has a large variety of algorithms. The default Zest layout algorithm, which was used up to this point, is a spring layout(see figure 6) where the edges should have about the same length. In these subsection will be described some of Zest layout algorithms.

Zest provides several different layout algorithms:

- GridLayoutAlgorithm positions nodes in a grid filled left to right,then top to bottom(see figure 12).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
graph.setLayoutAlgorithm(new GridLayoutAlgorithm(style),true);
```

- HorizontalLayoutAlgorithm positions all nodes in a single row(see figure 13).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
graph.setLayoutAlgorithm(new HorizontalLayoutAlgorithm(style),true);
```

- HorizontalTreeLayoutAlgorithm similar to a TreeLayoutAlgorithm but positions root nodes in the first column, child nodes in the, grandchild nodes in the third, and so on (see figure 14).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
graph.setLayoutAlgorithm(new HorizontalTreeLayoutAlgorithm(style),true);
```

- RadialLayoutAlgorithm positions nodes similarly to the TreeLayoutAlgorithm except with the roots at the center, child nodes in a circular fashion around the root nodes, grandchild nodes in a circular fashion around the child nodes, and so on (see figure 15).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
graph.setLayoutAlgorithm(new RadialLayoutAlgorithm(style),true);
```

- TreeLayoutAlgorithm positions root nodes in the first row, child nodes in the second row, grandchild nodes in the third row, and so on (see figure 16).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
graph.setLayoutAlgorithm(new TreeLayoutAlgorithm(style),true);
```

4 Conclusions

JUNG is a powerful framework for working with graphs that provides algorithms from different fields for graphs and also provides visualization framework that makes it easy to construct tools for the interactive exploration of data.

Zest is a framework built on top of Draw2D that enables developers to graphically show graph contents. The behavior of the nodes and edges of the graph can be highly configured(changing details as color,connection types in the graph etc.). Finally, multiple layout algorithms are provided, which can also be combined to make more complex and specific layout processes.

So both of the frameworks presented in this document are toolkits for creating and visualizing graphs each with its own structure. Even it may seem easier to work with Zest, JUNG has more features than Zest and it is better documented which helps a lot when time for implementing comes. Regarding the features, one difference between the two of them would be that Zest has only algorithms for layout manager while JUNG provides algorithms for calculating the shortest paths, maximum flows, clustering algorithms and so on. The only advantage(if can call it so) for working with Zest would be that has recent releases than JUNG;the last release of Zest(Zest 3.8.0) was in 2012 while JUNG's last release(JUNG2) was in 2010.

In this paper two of the most common graph visualization toolkits were presented. Next in our work we are going to choose one of these tools to implement a graphical user interface(GUI) which permits the visualization and interactively editing of context graphs.

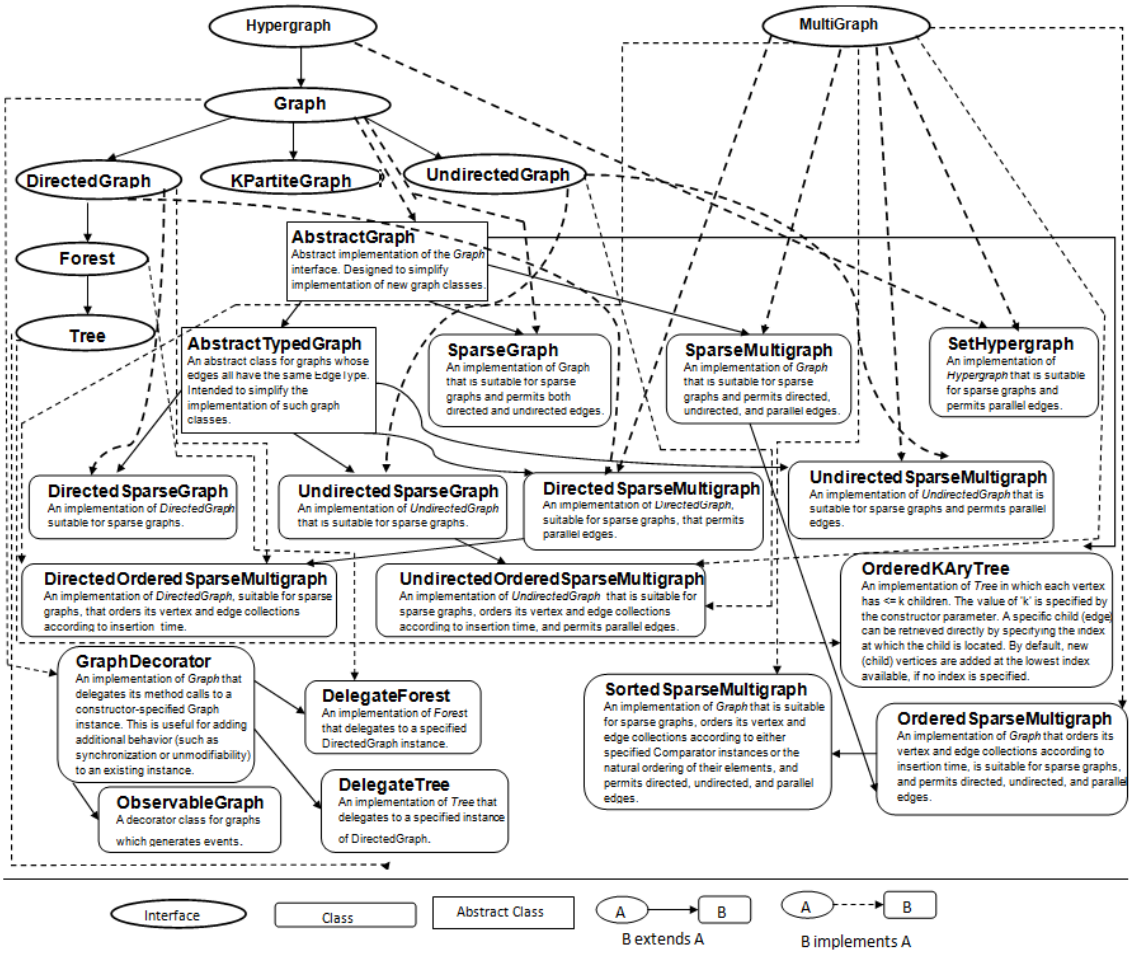

Figure 10: Graph types hierarchy in JUNG

Appendices

```
public static void main(String[] args){
        g.addVertex((Integer) 1);
        g.addVertex((Integer) 2);
        g.addVertex((Integer) 3);
        g.addVertex((Integer) 4);
        g.addEdge("Edge-A", 1, 2,EdgeType.DIRECTED);
        g.addEdge("Edge-B", 2, 3);
        g.addEdge("Edge-C", 2, 3,EdgeType.DIRECTED);
        layout.setSize(new Dimension(600,500));
        BasicVisualizationServer<Integer,String> vv =
        new BasicVisualizationServer<Integer,String>(layout);
        vv.setPreferredSize(new Dimension(600,500));
        vv.setBackground(Color.white);
        Transformer<Integer,Paint> vertexPaint =
        new Transformer<Integer,Paint>() {
                public Paint transform(Integer i) {
                if(i%2==0)return Color.CYAN;
                else return Color.RED;
                }
                };
        Transformer<String,Paint> edgePaint =
        new Transformer<String,Paint>() {
                public Paint transform(String i) {
                return Color.MAGENTA;
                }
                };
        float dash[] = {10.0f};
        final Stroke edgeStroke = new BasicStroke(1.0f, BasicStroke.CAP_BUTT,
        BasicStroke.JOIN_MITER, 10.0f, dash, 0.0f);
        Transformer<String, Stroke> edgeStrokeTransformer =
        new Transformer<String, Stroke>() {
        public Stroke transform(String s) {
        return edgeStroke;
        }
        };
        Transformer<Integer,Shape> vertexSize = new Transformer<Integer,Shape>(){
                 public Shape transform(Integer i){
                    Ellipse2D circle = new Ellipse2D.Double(-15,-15,40, 20);
                    if(i==1)
                    return AffineTransform.getScaleInstance(3, 3).createTransformedShape(circle);
                    else if(i==2) return circle;
                    else return new Rectangle(-20, -10, 40, 20);
                  }
              };
        vv.getRenderContext().setVertexFillPaintTransformer(vertexPaint);
        vv.getRenderContext().setVertexShapeTransformer(vertexSize);
        vv.getRenderContext().setEdgeFillPaintTransformer(edgePaint);
        vv.getRenderContext().setEdgeStrokeTransformer(edgeStrokeTransformer);
        vv.getRenderContext().setVertexLabelTransformer(new ToStringLabeller());
        vv.getRenderContext().setEdgeLabelTransformer(new ToStringLabeller());
        vv.getRenderer().getVertexLabelRenderer().setPosition(Position.CNTR);
        JFrame frame = new JFrame("Simple␣Graph␣View");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(vv);
        frame.pack();
        frame.setVisible(true);
}
```

Figure 11: Example of code for customized view of a graph in JUNG

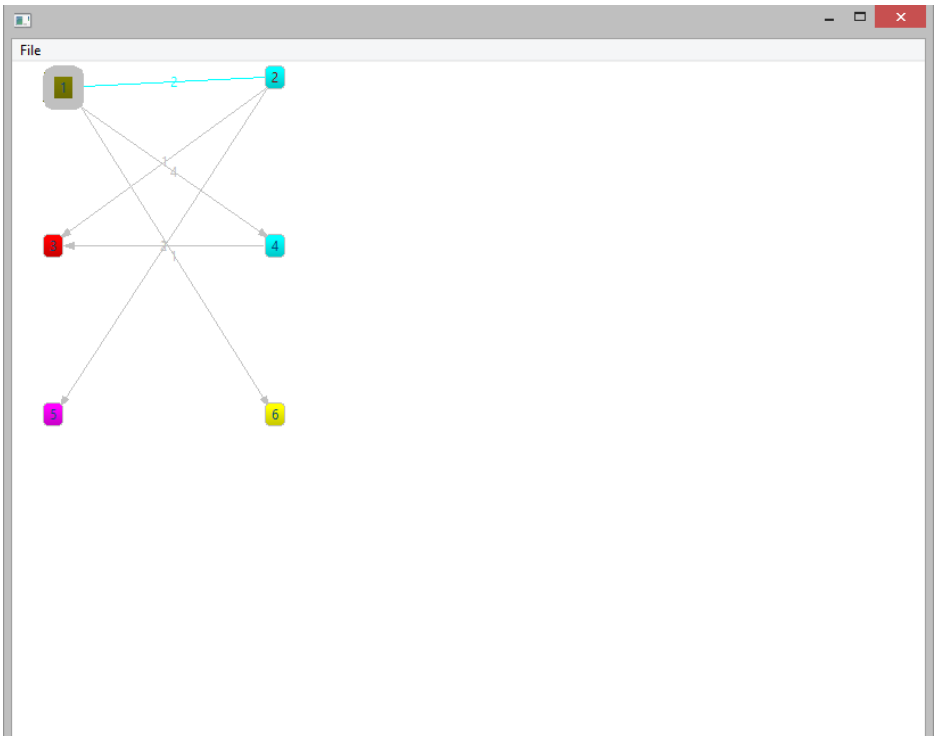

Figure 12: View of a graph in Zest with grid layout algorithm

Figure 13: View of a graph in Zest with horizontal layout algorithm

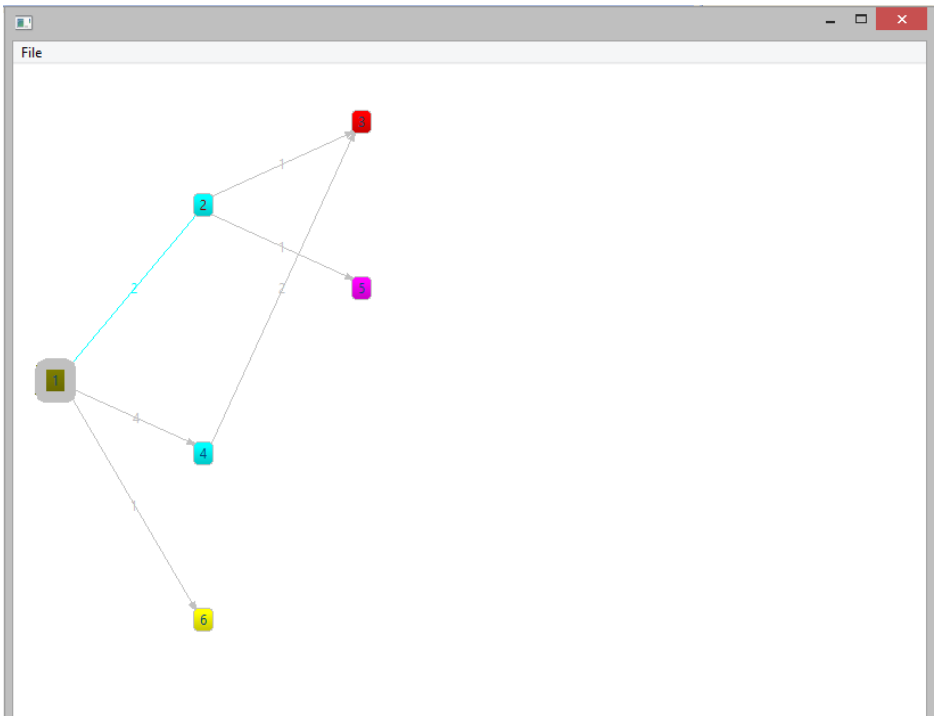

Figure 14: View of a graph in Zest with horizontal tree layout algorithm

References

[1] Andrei Olaru and Adina Magda Florea and Amal El Fallah Seghrouchni, *Graphs and Patterns for context awareness*, 2011.

[2] http://jung.sourceforge.net/

[3] http://stackoverflow.com/

[4] http://www.vogella.com/articles/EclipseZest/article.html

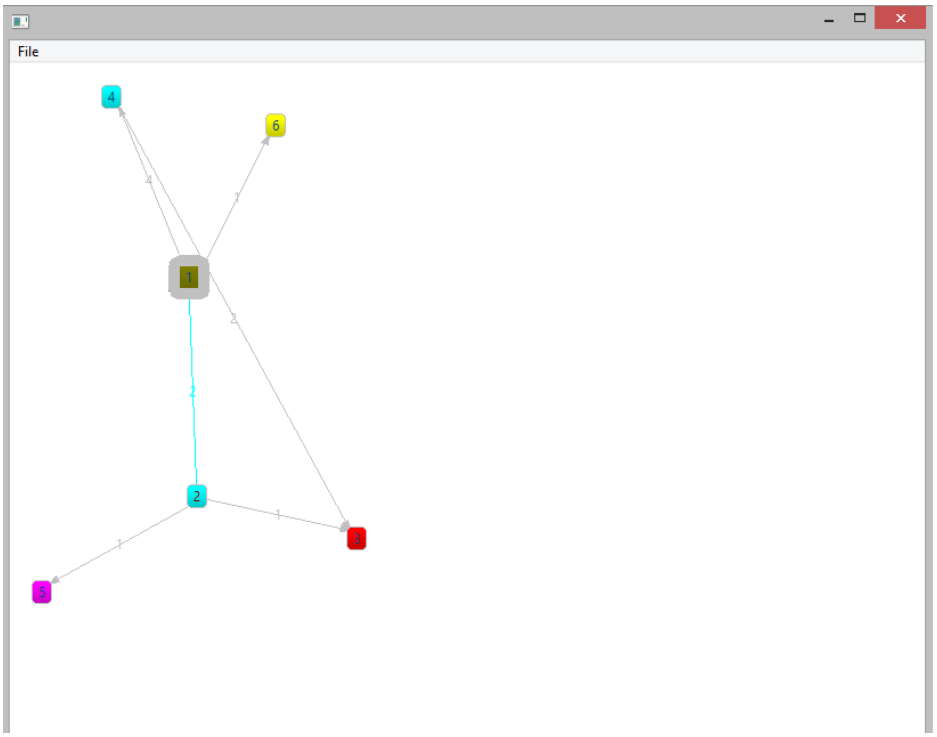

Figure 15: View of a graph in Zest with radial layout algorithm

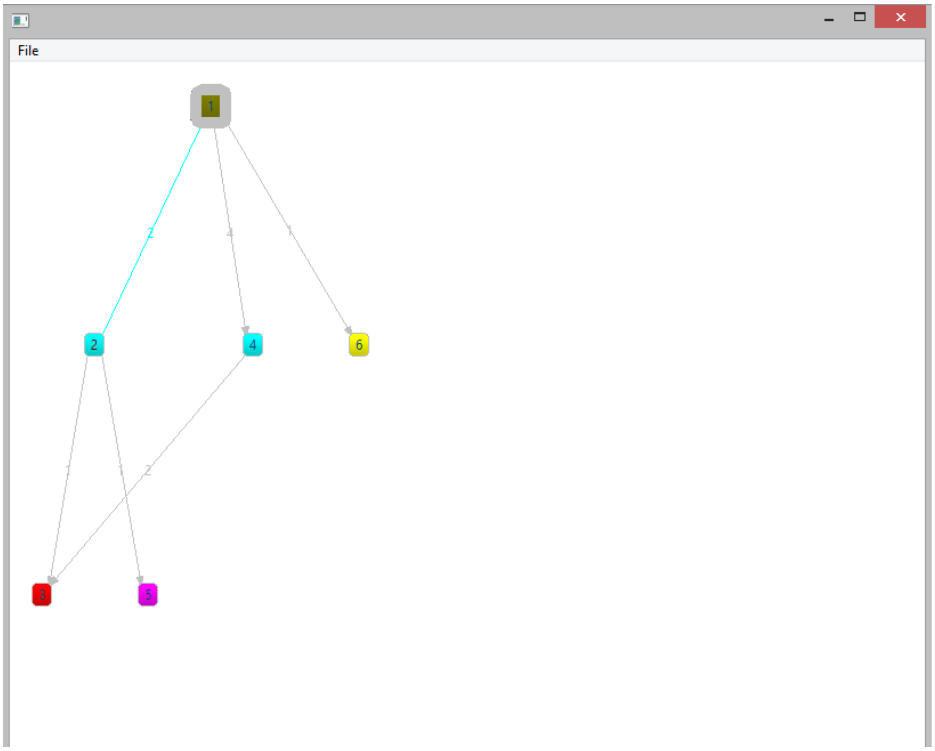

Figure 16: View of a graph in Zest with radial layout algorithm