JUNG 2.0 Tutorial

Or how to achieve graph based nirvana in Java

1 Introduction

JUNG2 is a major revision of the popular Java Universal Network/Graph framework. This is a programming tutorial intended to demonstrate and illustrate individual features of JUNG2 with as little extra code as possible. Hence none of the code examples here will show off the combined full power of JUNG2, for that you should try out the very nice example programs that come with JUNG2.

Note that the examples in this document have been tested against JUNG2 version 2.0 and this document was last updated April 22nd, 2009.

1.1 Information on the non-JUNG Library Dependencies

JUNG 2 depends on three other libraries: *JUnit*, *Colt*, and *Common Collections*.

1. JUnit is strictly used for testing JUNG. This is hosted at source forge: http://junit.sourceforge.net/ , with a very good separate site at http://www.junit.org/ . This is a very nice methodology for *unit testing*. Note that JUNG2 unit test provide further coding examples.
2. Commons Collections: This stems from the Apache Jakarta Commons Collections http://jakarta.apache.org/commons/collections/. However, the current version of this library doesn't use generics and hence loses much of the type safety that JUNG2 provides. Hence others noticed this and produced a version with the same functionality based on generics. The folks at http://larvalabs.com/collections/ have a source forge project http://sourceforge.net/projects/collections/ . Note that JUNG2 uses this and you will need to use a few key interfaces as discussed later in this tutorial
3. Colt: Arguably the best numerical library for Java out there. This was developed at CERN in conjunction with various other research labs. See http://dsd.lbl.gov/~hoschek/colt/ If you hate to have to go back to C++ just to do some fast numerical work, this is the library for you!

2 JUNG2 Graph Basics

Though most of us are impressed by JUNG's visualization capabilities. Its easiest to understand what's going on at first without all the extra GUI stuff.

To start off, there is an interface *Graph<V,E>* defined in edu.uci.ics.jung.graph

. This interface defines the basic operations that you can perform on a graph. These include:

1. Adding and removing edges and vertices from the graph and getting collections of all edges and vertices in a graph.
2. Getting information concerning the endpoints of an edge in the graph.
3. Getting information concerning vertices in a graph including various degree measures (in-

degree, out-degree) and predecessor and successor vertices.

There are three other related interfaces:

1. DirectedGraph<V, E> -- This is a “marker” interface which is used to indicate that classes implementing this interface will only support directed edges.
2. UndirectedGraph<V,E> -- This is a “marker” interface which is used to indicate that classes implementing this interface will only support undirected edges.
3. SimpleGraph<V, E> -- This is a “marker” interface which is used to indicate that classes implementing this interface will only not support parallel edges or self loops. The commonly used definition of a *simple* graph.

For those interested, JUNG2 actually provides a generalization of the concept of a graph known as a hyper-graph. A hyper-graph is similar to a graph in that it contains vertices and edges, however unlike a graph, in a hyper-graph an edge maybe associated with more than two vertices. JUNG2's Graph<V,E> interface actually extends the Hypergraph<V,E> interface.

2.1 Creating a Graph and adding vertices and edges

In JUNG2 vertices and edges can be any object type. Note that the same vertices (and edges) can appear in more than one graph. This is very nice for looking at different topologies with the same nodes. Although you can create your own graph classes conforming to the Graph<V,E> interface JUNG2 provides a number implementations of various graph types. In our first example, from the file *BasicGraphCreation.java*, we'll use the SparseMultigraph<V, E> class.

The sequence of calls:

```
// Graph<V, E> where V is the type of the vertices
// and E is the type of the edges
Graph<Integer, String> g = new SparseMultigraph<Integer, String>();
// Add some vertices. From above we defined these to be type Integer.
g.addVertex((Integer)1);
g.addVertex((Integer)2);
g.addVertex((Integer)3);
// Add some edges. From above we defined these to be of type String
// Note that the default is for undirected edges.
g.addEdge("Edge-A", 1, 2); // Note that Java 1.5 auto-boxes primitives
g.addEdge("Edge-B", 2, 3);
// Let's see what we have. Note the nice output from the
// SparseMultigraph<V,E> toString() method
System.out.println("The graph g = " + g.toString());
// Note that we can use the same nodes and edges in two different graphs.
Graph<Integer, String> g2 = new SparseMultigraph<Integer, String>();
g2.addVertex((Integer)1);
g2.addVertex((Integer)2);
g2.addVertex((Integer)3);
g2.addEdge("Edge-A", 1,3);
g2.addEdge("Edge-B", 2,3, EdgeType.DIRECTED);
g2.addEdge("Edge-C", 3, 2, EdgeType.DIRECTED);
g2.addEdge("Edge-P", 2,3); // A parallel edge
System.out.println("The graph g2 = " + g2.toString());
```

Will produce graphs with two different topologies utilizing the same vertices and edges. The graph toString() method produces the following output:

```
The graph g = Vertices:2,1,3
```

```
Edges:Edge-B[2,3]Edge-A[1,2]
The graph g2 = Vertices:2,1,3
Edges:Edge-P[2,3]Edge-B[2,3]Edge-C[3,2]Edge-A[1,3]
```

Where does the name SparseMultigraph<V,E> come from? *Sparse* means that the implementation is efficient for large graphs with low average connectivity, *Multi* meaning that the graph supports parallel edges. If you do not want or need parallel edge support JUNG2 provides a SparseGraph<V,E> class.

2.2 Constructing a Directed Graph with Custom Edges

Have more information than a single number or string that you'd like to associate with an edge or a vertex? In JUNG2 you just specify your own classes for the vertex (V) and edge (E) classes in the declaration and constructors of graphs and algorithms.

Here is some sample code for our own vertex and edge classes from *BasicDirectedGraph.java*:

```
class MyNode {
    int id; // good coding practice would have this as private
    public MyNode(int id) {
        this.id = id;
    }
    public String toString() { // Always a good idea for debuging
        return "V"+id;         // JUNG2 makes good use of these.
    }
}

class MyLink {
    double capacity; // should be private
    double weight;   // should be private for good practice
    int id;

    public MyLink(double weight, double capacity) {
        this.id = edgeCount++; // This is defined in the outer class.
        this.weight = weight;
        this.capacity = capacity;
    }

    public String toString() { // Always good for debugging
        return "E"+id;
    }

}
```

To create the following directed graph using our own edge and vertex classes is fairly simple.

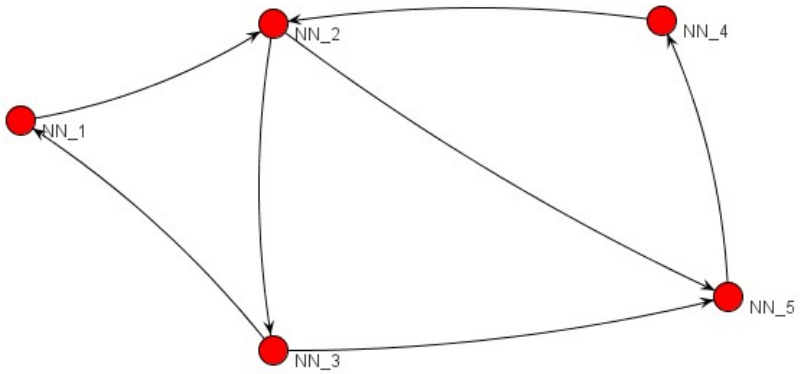

Figure 1: *A simple directed graph to be implemented in JUNG2*

The following partial code shows the graph construction:

```
g = new DirectedSparseMultigraph<MyNode, MyLink>();
// Create some MyNode objects to use as vertices
n1 = new MyNode(1); n2 = new MyNode(2); n3 = new MyNode(3);
n4 = new MyNode(4); n5 = new MyNode(5); // note n1-n5 declared elsewhere.
// Add some directed edges along with the vertices to the graph
g.addEdge(new MyLink(2.0, 48),n1, n2, EdgeType.DIRECTED); // This method
g.addEdge(new MyLink(2.0, 48),n2, n3, EdgeType.DIRECTED);
g.addEdge(new MyLink(3.0, 192), n3, n5, EdgeType.DIRECTED);
g.addEdge(new MyLink(2.0, 48), n5, n4, EdgeType.DIRECTED); // or we can use
g.addEdge(new MyLink(2.0, 48), n4, n2); // In a directed graph the
g.addEdge(new MyLink(2.0, 48), n3, n1); // first node is the source
g.addEdge(new MyLink(10.0, 48), n2, n5);// and the second the destination
```

Note that the addEdge call in a directed graph doesn't need the EdgeType to be specified. Trying to add an undirected edge to a directed graph should result in an exception (and will with the JUNG2 implementations).

Output for the above using the Graph<V,E> toString() method will give something like:

```
Vertices:V2,V5,V3,V4,V1
Edges:E5[V3,V1]E6[V2,V5]E4[V4,V2]E3[V5,V4]E2[V3,V5]E1[V2,V3]E0[V1,V2]
```

3 Working with Algorithms

Given that we can create undirected and directed graphs the next simplest thing to do is try running some algorithms on them. Along the way we'll see two encounter two important Java design idioms that come up frequently in JUNG2: (a) the *Transformer* idiom, (b) the *Factory* idiom.

3.1 An Unweighted Shortest Path

To find the shortest path assuming uniform link weights (e.g., 1 for each link) in the graph we created we can use the following code from *BasicDirectedGraph.java*:

```
DijkstraShortestPath<MyNode,MyLink> alg = new DijkstraShortestPath(g);
List<MyLink> l = alg.getPath(n1, n4);
System.out.println("The shortest unweighted path from" + n1 +
      " to " + n4 + " is:");
System.out.println(l.toString());
```

Nice and simple. Note that the shortest path algorithm is also "parameterized" and hence works with your vertex (V) and edge (E) classes.

The output looks like:

```
The shortest unweighted path fromV1 to V4 is:
[E0, E6, E3]
```

3.2 A Weighted Shortest Path and Transformer classes

The previous example found the shortest path with all link weights equal to 1, also known as the minimum hop count path for those of us in data networking. However, we frequently want to use other links weights and we designed our MyLink class with such a custom weight in mind. However, we now need a way to "feed" the link weight to the algorithm in a way that can work with any edge class.

This is where the *commons collections* ***Transformer<E, Number>*** interface comes into use. The only method in this interface is ***Number transform(E e),*** hence given an edge class of our creation, we will need to come up with an appropriate (and usually very simple) Transformer class to extract the weight for that edge.

The general signature for the Dijkstra algorithm constructor that we will use is DijkstraShortestPath(Graph<V,E> g, Transformer<E,Number> nev). The following code from *BasicDirectedGraph.java* illustrates how this is done with our MyLink edge class. Note that the graph *g* referenced in the code is the same as the one constructed in section 2.2.

```
Transformer<MyLink, Double> wtTransformer = new Transformer<MyLink,Double>() {
    public Double transform(MyLink link) {
        return link.weight;
    }
};
DijkstraShortestPath<MyNode,MyLink> alg = new DijkstraShortestPath(g,
    wtTransformer);
List<MyLink> l = alg.getPath(n1, n4);
Number dist = alg.getDistance(n1, n4);
System.out.println("The shortest path from" + n1 + " to " + n4 + " is:");
System.out.println(l.toString());
System.out.println("and the length of the path is: " + dist);
```

This *Transformer* idiom (or design pattern) will comes up many times in JUNG2. The output from this looks like:

```
The shortest weighted path from V1 to V4 is:
[E0, E1, E2, E3]
and the length of the path is: 9.0
```

3.3 Max-Flows and the Factory Idiom

Wait! You may still want to read this even if you don't care about calculating max-flows in a graph. The folks that brought us JUNG2 implemented the very efficient Edmonds-Karp algorithm for calculating the maximum flow in a graph where the edges have a capacity. For those of us in data communications this is usually related to the bandwidth of the communications link. The constructor for this algorithm is the slightly intimidating:

```
EdmondsKarpMaxFlow(DirectedGraph<V,E> directedGraph, V source, V sink,
Transformer<E,Number> edgeCapacityTransformer, Map<E,Number> edgeFlowMap,
```

```
Factory<E> edgeFactory)
```
In this constructor we furnish the graph of interest, the source and sink vertices, a *Transformer* to deliver the capacity (a Number) for each edge (just like the edge weight case before), a map structure that the algorithm will use in its calculations, and a *Factory<E>*. The ***Factory<E>*** interface comes from the *commons collections* and consists of just one method ***E create()*** whose job it is to create a new instance of the edge class. In the situation here, the algorithm utilizes the factory to create additional edges needed as part of its calculations. In the section on graphics you'll see this same idiom used when you want utilize the visualization classes to create edges and vertices graphically.

From *BasicDirectedGraph.java* the following creates and runs the Edmonds-Karp algorithm:

```
Transformer<MyLink, Double> capTransformer =
           new Transformer<MyLink, Double>(){
  public Double transform(MyLink link)  {
      return link.capacity;
  }
};
Map<MyLink, Double> edgeFlowMap = new HashMap<MyLink, Double>();
// This Factory produces new edges for use by the algorithm
Factory<MyLink> edgeFactory = new Factory<MyLink>() {
    public MyLink create() {
        return new MyLink(1.0, 1.0);
    }
};

EdmondsKarpMaxFlow<MyNode, MyLink> alg =
           new EdmondsKarpMaxFlow(g,n2, n5, capTransformer, edgeFlowMap,
           edgeFactory);
alg.evaluate();
System.out.println("The max flow is: " + alg.getMaxFlow());
System.out.println("The edge set is: " + alg.getMinCutEdges().toString());
```

The output from this looks like:

```
The max flow is: 96
The edge set is: [E6, E1]
```

4 Visualizing Graphs

So you now can build some graphs and run some algorithms. If you've run some of the JUNG demos then you know that JUNG has great visualization capabilities. Here we will be showing JUNG2's visualization capabilities from a programmers perspective trying to illustrate various features with as little extra code as necessary to illustrate the feature of interest.

4.1 The Basics of Viewing a Graph

To view a graph with JUNG2 we need two new concepts: (a) that of a graph layout, and (b) that of a visualization component. But don't worry the explanation is much longer than the code required by the user!

The basic class for viewing graphs in JUNG2 is the *BasicVisualizationServer* class (edu.uci.ics.jung.visualization). This implements the JUNG2 *VisualizationServer<V,E>* interface and inherits from Swing's *JPanel* class (javax.swing.JPanel) and hence can be placed anywhere in the Swing GUI hierarchy where you would use a similar component (see the Java tutorial's Swing trail for more information). This is the “canvas” on which the graph will be drawn.

Prior to creating this we need to a way to place the vertices of a graph at locations in the visualization space (i.e., need to assign locations to the vertices). This is achieved via JUNG2'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.

The minimal pieces we need to display a graph are:

1. The graph itself (we'll use our very first graph from section 2.1.)
2. A layout implementation (for our example here we'll use *CircleLayout*)
3. A *BasicVisualizationServer* on which to show our graph
4. A base GUI component. Here we'll just use a Swing *JFrame*.

The complete example is available in the file *SimpleGraphView.java*. The main program that does everything but create the graph is:

```
public static void main(String[] args) {
    SimpleGraphView sgv = new SimpleGraphView(); //We create our graph in here
    // The Layout<V, E> is parameterized by the vertex and edge types
    Layout<Integer, String> layout = new CircleLayout(sgv.g);
    layout.setSize(new Dimension(300,300)); // sets the initial size of the space
     // The BasicVisualizationServer<V,E> is parameterized by the edge types
     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);
}
```

The layout.setSize() call above sets the horizontal and vertical bounds on where the layout algorithm will place the vertices. The setPreferredSize() on the BasicVisualizationServer sets the size of the GUI (Swing) component. The resulting graph when run on my system looks something like:

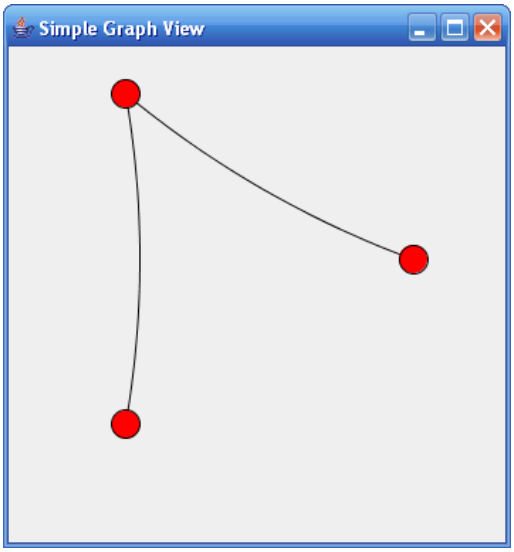

Figure 2: Our first view of a graph with JUNG2.

4.2 Painting and Labeling Vertices and Edges

We'll we can see our graph but now the marketing folks will start voicing their opinions over colors and line thickness, etc... In addition, in many types of graphs it is crucial to provide labels for the vertices and/or edges.

JUNG2 provides a very extensible framework for supporting about every graph display variation that your marketing department can think of (but never all!). At the same time this framework is easy to use with lots of very nice looking defaults. The two new sets of interfaces/classes that we'll want a bit of familiarity with are the (a) *RenderContext* (in edu.uci.ics.jung.visualization) and (b) *Renderer* (edu.uci.ics.jung.visualization.renderers).

The JUNG2 renderers are used to actually draw four different items: (a) edges, (b) edge labels, (c) vertices, and (d) vertex labels. JUNG2 supports the notion of pluggable renderers so that you can substitute different renderers for the default. However, you may not need to do this since each renderer was defined via interfaces to be fairly general and can accept a number of "parameters". How do we supply all these parameters (such as vertex color or edge label) to the graph renderers? Where are the default values (if any) kept? This is the job of the *RenderContext*. Each BasicVisualizationServer (the thing we use to actually display the graph) contains a *RenderContext* object that we can access to set these various rendering "parameter" values. Look at the JavaDoc for edu.uci.ics.jung.visualization.RenderContext to see the full set of "parameters" that you can set.

For now lets limit ourselves to the following:

1. Change the vertex color from the default to green.
2. Make the line used in the edges a dashed line.
3. Display a label for both the edges and vertices.
4. Center the vertex label within its corresponding vertex.

For the first three of these we will use the following calls to the RenderContext that we can get from the current BasicVisualizationServer class. These are: *setVertexFillPaintTransformer*(), *setEdgeStrokeTransformer*(), *setVertexLabelTransformer*(), and *setEdgeLabelTransformer*(). As you

could surmise from the names of these method each takes a *Transformer* class argument that converts an edge or vertex to the type of information needed by a renderer. This also means that it is easy for you as a programmer to change the visual aspects of the graph based on attributes of your custom edge and vertex classes.

The complete code is given in *SimpleGraphView2.java*, however the main new functionality is given by:

```
    public static void main(String[] args) {
        SimpleGraphView2 sgv = new SimpleGraphView2(); // This builds the graph
        // Layout<V, E>, BasicVisualizationServer<V,E>
        Layout<Integer, String> layout = new CircleLayout(sgv.g);
        layout.setSize(new Dimension(300,300));
        BasicVisualizationServer<Integer,String> vv =
              new BasicVisualizationServer<Integer,String>(layout);
        vv.setPreferredSize(new Dimension(350,350));
        // Setup up a new vertex to paint transformer...
        Transformer<Integer,Paint> vertexPaint = new Transformer<Integer,Paint>() {
            public Paint transform(Integer i) {
                return Color.GREEN;
            }
        };
        // Set up a new stroke Transformer for the edges
        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;
            }
        };
        vv.getRenderContext().setVertexFillPaintTransformer(vertexPaint);
        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 2");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(vv);
        frame.pack();
        frame.setVisible(true);
    }
```

The results from this new code is shown below:

The code uses a nice utility Transformer *ToStringLabeller* (from edu.uci.ics.jung.visualization.decorators) that calls the edge's or vertex's toString() method. For more information on the *BasicStroke* class see the Java2D trail in the Java Tutorial.

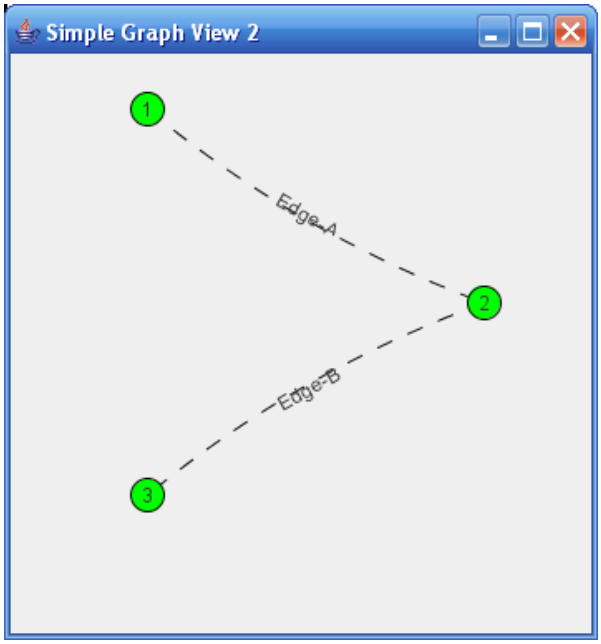

Figure 3: Changing the way edges and vertices are drawn.

5 Getting Interactive

JUNG2 provides GUI features to let users interact with graphs in various ways. Most interactions with a graph will take place via a mouse. Since there are quite a number of conceivable ways that users may want to interact with a graph, JUNG2 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, and more!

To deal with a multitude of mouse modes JUNG2 uses the concept of a "pluggable mouse", i. e., a mouse that accepts various plugins to implement the currently supported modes. For more details on the mouse and mouse plugin classes see section 5.3 The JUNG2 Mouse and Mouse Plugin Hierarchy.

5.1 Scaling (Zooming) and Transforming (pan, rotate and skew) a Graph Visually

Okay so what do we need to do to get started in providing interactivity? We'll we need a JUNG2 graph mouse. So we'll use the DefaultModalGraphMouse which provides a full set of picking and transforming capabilities. In addition we'll upgrade from the BasicVisualizationServer to the *VisualizationViewer* derived class that also provides for mouse functionality. The complete code is given in *InteractiveGraphView1.java* however the code for the graphics and interactivity is just:

```
public static void main(String[] args) {
    // I create the graph in the following...
    InteractiveGraphView1 sgv = new InteractiveGraphView1();
    // Layout<V, E>, VisualizationComponent<V,E>
    Layout<Integer, String> layout = new CircleLayout(sgv.g);
```

```
        layout.setSize(new Dimension(300,300));
        VisualizationViewer<Integer,String> vv =
                new VisualizationViewer<Integer,String>(layout);
        vv.setPreferredSize(new Dimension(350,350));
        // Show vertex and edge labels
        vv.getRenderContext().setVertexLabelTransformer(new ToStringLabeller());
        vv.getRenderContext().setEdgeLabelTransformer(new ToStringLabeller());
        // Create a graph mouse and add it to the visualization component
        DefaultModalGraphMouse gm = new DefaultModalGraphMouse();
        gm.setMode(ModalGraphMouse.Mode.TRANSFORMING);
        vv.setGraphMouse(gm);
        JFrame frame = new JFrame("Interactive Graph View 1");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(vv);
        frame.pack();
        frame.setVisible(true);
    }
```

Hey, there are only three new lines of code... But how do I (or the user) use this stuff? The default behavior is:

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

Sorry no pictures here, you really need to give this a try yourself ☺.

5.2 Controlling the Mouse Mode via Key Listeners

JUNG2 provides a couple extra ways to control the current mouse mode besides programmatically setting it yourself. Here we'll use 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 (pan) mode if the user types a "t" and to picking mode if the user types a "p". Now since key events are received by Java AWT components we need to register this key listener with the *VisualizationViewer* that displays our graph. See the file *InteractiveGraphView2.java* for the complete code but we'll only need to add the following line of code to get this nifty functionality:

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

Where *vv* is a *VisualizationViewer* object and *gm* is a *DefaultModalGraphMouse* object.

5.3 Making our own Simple Graph Mouse

Okay, you like the DefaultModalGraphMouse that we previously used. But you'd like something simpler for your users. Suppose while viewing the graph you only want the users to be able to pan (translate) or zoom the graph. No picking of vertices and no rotating or skewing of the graph.

The mouse hierarchy roughly consists of the interfaces: *VisualizationViewer.GraphMouse*, and *ModalGraphMouse.* Since we don't need a mouse that changes modes, e.g., picking versus transforming, we won't need to implement the *ModalGraphMouse* interface. In addition we don't need to start from scratch either since JUNG has the concept mouse plugins that do the heavy lifting for a particular purpose such as picking, translating, zooming, editing, etc... Since we just want translation and zooming we'll use the *TranslatingGraphMousePlugin* and *ScalingGraphMousePlugin*

*(*edu.uci.ics.jung.visualization.control) classes. What do these "plugins" plug into? A *PluggableGraphMouse* (edu.uci.ics.jung.visualization.control) object of course! So, the idea is to choose your plugins (and JUNG has a bunch) then add them to a PluggableGraphMouse.

The complete code is in the file *InteractiveGraphView3.java*. We'll only need the following three new lines of code in our main method:

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

If you run the sample code, you'll see that translating and zooming work but we no longer have rotation and skewing nor can we switch to the picking mode.

Later we'll look at customizing a modal graph mouse, but first let's see how to edit graphs interactively.

5.4 Graph Editing: Adding Vertices and Nodes Interactively

Many application require the interactive editing of a graph. To do this we JUNG2 we'll use a more fully featured graph mouse, the *EditingModalGraphMouse* (edu.uci.ics.jung.visualization.control). Since this mouse will be used to create new vertices and edges in response to user actions, its constructors requires that we furnish factory classes (derived from the Factory<E> and Factory<V> interfaces). The complete code is given in file *EditingGraphView1.java* The essential functionality not including the vertex and edge factories is given below. We'll need a simple static layout to hold our vertex locations. This code is shown highlighted in light blue .

```
    public static void main(String[] args) {
        EditingGraphViewer1 sgv = new EditingGraphViewer1();
        // Layout<V, E>, VisualizationViewer<V,E>
        Layout<Integer, String> layout = new StaticLayout(sgv.g);
        layout.setSize(new Dimension(300,300));
        VisualizationViewer<Integer,String> vv =
                  new VisualizationViewer<Integer,String>(layout);
        vv.setPreferredSize(new Dimension(350,350));
        // Show vertex and edge labels
        vv.getRenderContext().setVertexLabelTransformer(new ToStringLabeller());
        vv.getRenderContext().setEdgeLabelTransformer(new ToStringLabeller());
        // Create a graph mouse and add it to the visualization viewer
        //
        EditingModalGraphMouse gm =
                  new EditingModalGraphMouse(vv.getRenderContext(),
                  sgv.vertexFactory, sgv.edgeFactory);
        vv.setGraphMouse(gm);


        JFrame frame = new JFrame("Editing Graph Viewer 1");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(vv);

        // Let's add a menu for changing mouse modes
        JMenuBar menuBar = new JMenuBar();
        JMenu modeMenu = gm.getModeMenu(); // Obtain mode menu from the mouse
        modeMenu.setText("Mouse Mode");
        modeMenu.setIcon(null); // I'm using this in a main menu
        modeMenu.setPreferredSize(new Dimension(80,20)); // Change the size
```

```
        menuBar.add(modeMenu);
        frame.setJMenuBar(menuBar);
        gm.setMode(ModalGraphMouse.Mode.EDITING); // Start off in editing mode
        frame.pack();
        frame.setVisible(true);
    }
```

Since the user will be deciding via the mouse where vertices will be place we will utilize a particularly simple graph *Layout* called *StaticLayout*. The editing graph mouse creation code is highlighted in yellow. Note that the editing mouse constructor requires both a vertex factory and an edge factory.

Now the user has a number of modes in which they can use the mouse: editing, picking, and transforming. JUNG's *EditingModalGraphMouse* offers a nice somewhat built in way for the user to select between mouse modes, by providing either a pre-made JMenu for this purpose or a pre-made JComboBox. The code highlighted in light green above shows this along with some formating tweaks for using this in a main menu.

5.5 Graph Editor with Edge and Vertex Property Editing Menus

In this section we will describe a more complete graph editor application. The key differences between this and the previous examples are:

1. We will work with more realistic custom vertex and edge classes that will have several application specific properties that we'd like to edit.
2. We will implement a general mouse plugin that will bring up different popup menus based on whether the user clicks the right mouse button over a vertex or over an edge.
3. We will implement two generic entries for the above menu for deleting a user defined vertex or edge.
4. We will show how to use the above general mouse plugin by developing popup menus that can show information specific to the edge or vertex selected and allow editing of specific edge or vertex properties.

Before we go on lets look at three screen shots of what the user gets from this functionality.

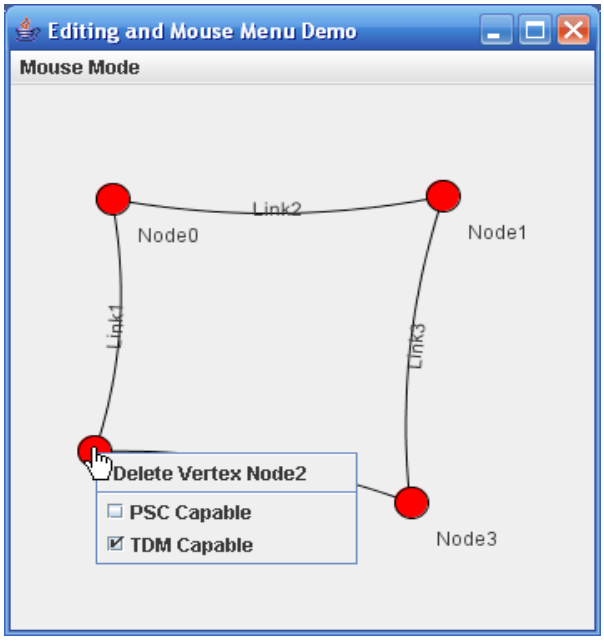

Figure 4: A context sensitive vertex menu with check boxes for changing properties.

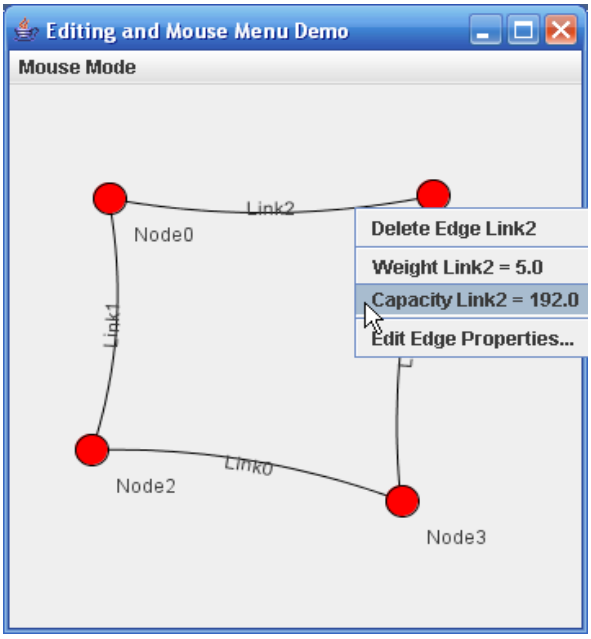

Figure 5: A context sensitive edge menu displaying properties and such.

In Figure 4 we show a vertex menu that has check boxes to show and change the state of the two vertex specific properties. In addition this menu contains the generic delete vertex menu item. In Figure 5 we show the context sensitive edge menu (context sensitive meaning that it shows the properties and acts on the specific edge selected). Note how we use menu items to show edge properties and utilize the generic edge delete menu item. Finally in Figure 6 we show the edge properties dialog that the user can bring up from the edge menu (once again its is furnished with the selected edge object).

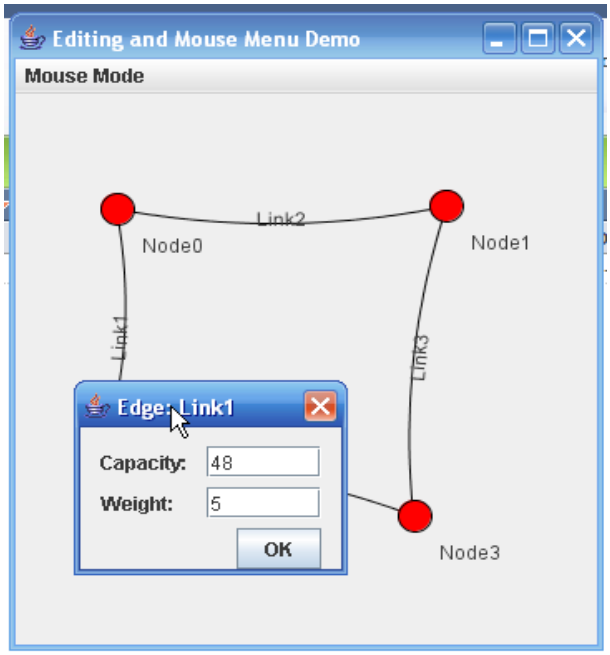

Figure 6: A context sensitive edge dialog brought up from the edge menu.

Unlike our previous single file examples we will break this project up into multiple files, but to avoid an explosion of files we will utilize inner classes to pack multiple classes into a single Java file. The files and their contents are:

1. *GraphElements.java* → contains our custom edge and vertex classes as static inner classes *MyVertex* and *MyEdge*. *MyVertex* contains two boolean valued properties, and *MyEdge* contains two double valued properties. Although simple these take up space with a full complement of getter/setter methods.
2. *PopupVertexEdgeMenuMousePlugin.java* → contains our new graph mouse plugin. This is given *JPopupMenus* to be updated and shown when an edge or vertex is selected with the right mouse button. Note that this is generic and can work with any edge or vertex types.
3. Three simple listener interfaces: *VertexMenuListener.java*, *EdgeMenuListener.java*, and *MenuPointListener.java* are implemented by menu items within the popups that wish to be updated prior to being shown with the selected vertex or edge. The point interface is for those that also want to know where in the graph the selection was made, e.g., for bringing up a dialog box where the item was selected.
4. *MyMouseMenus.java* → This contains two main static inner classes that extend JPopupMenu, *EdgeMenu* and *VertexMenu*. These in turn are constructed from classes that extend either JMenuItem or JCheckBoxMenuItem and one or more of the listener interfaces from above to update text or other information with context (edge or vertex) sensitive information.
5. Two generic deletion menu items: *DeleteEdgeMenuItem.java*, *DeleteVertexMenuItem.java* these are generic and can work with any edge or vertex types respectively. One very useful functionality of a popup mouse menu is to allow users to delete edges or vertices, hence we made these generic.
6. Finally we have the main program *EditorMouseMenu.java* which is very similar to previous examples and a dialog for edge properties, *EdgePropertyDialog.java*. Note that this dialog uses the swing layout extensions (https://swing-layout.dev.java.net/) since it was created via NetBeans.

5.6 The JUNG2 Mouse and Mouse Plugin Hierarchy

This stuff may be moved to an appendix as it was really an aid in writing the more complicated example of the previous section.

5.6.1 The Mouse Hierarchy

The JUNG2 graph mouse hierarchy is headed by the *VisualizationViewer.GraphMouse* interface.

Next in the hierarchy is *ModalGraphMouse* which provides an interface for setting and getting mouse modes.

The first class we get to is the *PluggableGraphMouse* which is a GraphMouse that accepts plugins.

Then we start getting some basic functionality for picking and transforming in the *AbstractModalGraphMouse* class.

The concrete classes provided by JUNG2 currently include:

Class Name	Functionality

DefaultModalGraphMouse	is a PluggableGraphMouse class that pre-installs a large collection of plugins for picking and transforming the graph. Additionally, it carries the notion of a Mode: Picking or Translating.
EditingModalGraphMouse<V,E>	A fully functional modal mouse similar to the previous class but adds functionality for adding and deleting vertices and edges to a graph.
AnnotatingModalGraphMouse<V,E>	a graph mouse that supplies an annotations mode
ModalLensGraphMouse	an implementation of the AbstractModalGraphMouse that includes plugins for manipulating a view that is using a LensTransformer.
ModalSatelliteGraphMouse	A subclass of the DefaultModalGraphMouse...

5.6.2 The Mouse Plugin Hierarchy

Interface GraphMousePlugin (edu.uci.ics.jung.visualization.control), Methods: boolean checkModifiers(MouseEvent), int getModifiers(), setModifiers(int).

The modifier constants are defined in interface edu.uci.ics.jung.visualization.event.Event. These are almost all "masks" which we can do bit comparisons against. Examples button 1, 2, and 3 masks, control key mask, alt key mask, shift key mask.

Mouse events let us know where the mouse was clicked, how many times, and whether other keys where held, etc...

MouseEvent Hierarchy: MouseEvent → InputEvent → EventObject (all in edu.uci.ics.jung.visualization.event) Methods in MouseEvent: int getButton(), int getClickCount(), Point getPoint(), int getX(), int getY(), boolean isPopupTrigger().

Methods in InputEvent used by MouseEvent: consume, getExtendedModifiers, getModifiers, getTime, isAltDown, isConsumed, isControlDown, isShiftDown

Class **AbstractGraphMousePlugin** implements the plugin interface and keeps the following information: Point2D down (the last point the mouse was down), Cursor (the current cursor), modifiers (the modifiers to use to check if the plugin should be activated).

And all the following extend the above class: **AbstractPopupGraphMousePlugin, AnimatedPickingGraphMousePlugin, AnnotatingGraphMousePlugin, EditingGraphMousePlugin, LabelEditingGraphMousePlugin, LensMagnificationGraphMousePlugin, PickingGraphMousePlugin, RotatingGraphMousePlugin, ScalingGraphMousePlugin, ShearingGraphMousePlugin, TranslatingGraphMousePlugin, ViewTranslatingGraphMousePlugin**