Visualizing Relational Data Using Graph Theory

by Jason Bellone and Daniel Lang

What better platform than Flex and ActionScript to create a rich information visualization experience? Can a few lines of code serve to paint a clear picture of your data? Let's give it a try...

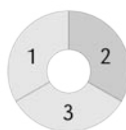
What you will learn...

- Building a graph theory based visualization application
- Customizing look and feel of nodes and edges to convey meaning
- Creating controls and events to filter and drill-down data

What you should know...

- Create a Flex Project using external libraries
- Basic ActionScript
- How to configure item renderers
- Working with XML

Level of difficulty



In the words of Edward Tufte, "Graphical excellence consists of complex ideas communicated with clarity, precision, and efficiency." Information visualization involves representing data in a form that facilitates understanding. The form of presenting relational data is the focus of this article, and specifically the potential of using Flex/ActionScript to support the visual representation of graphs and other characteristics of structured data.

Getting Started

This tutorial is based on an open source library called BirdEye and its Relational Analysis Visualization (RaVis) component. BirdEye is a community project to advance the design and development of a comprehensive open source information visualization and visual analytics library for Adobe Flex. The actionscript-based library enables users to create multi-dimensional data visualization interfaces for the analysis and presentation of information.

The RaVis component enables users to create complex data visualization interfaces for the analysis of relational data sets such as social networks, organization trees, navigation systems, taxonomies, db schemas, and other link-based phenomena. The compiled RaVis library (e.g. RaVis.swc) should be included in your Flex Project.

Basic Concepts

Relational analysis is based on the exploration and discovery of associations between objects. This method of analysis provides visibility of the relationships and associations between objects of different types that are not apparent from raw data. These methods have long standing history for applications to social sciences (e.g. social networks) and have developed through the contributions of multiple disciplines such as mathematics, graph theory, biology, physics, and other sciences. The common goal of these approaches is to provide visibility to structures that may be present as hidden patterns or links among seemingly unrelated items. To better understand the basis for relational analysis and the methods of graph theory, five basic points should be understood:

- A *node* is a single graphed object or item;
- An *edge* is a link between two nodes;
- A *graph* refers to a set of nodes and a set of edges that connect pairs of nodes;
- A *graph* drawing is based on a *layout* – a mathematically based method/algorithm for drawing nodes and edges in a calculated structure;
- The distance between interrelated nodes is often termed degrees of separation, e.g. a node-edge-node relation represents one degree of separation.

Data Source

We'll begin the project by defining a simple XML data source. This data source will specify the creation of nodes and edges. Nodes in the graph are declared by a node element where each node has a unique identifier. Edges in the graph are declared by the edge element where *from* and *to* relations are established between two nodes. Listing 1 illustrates the basic requirements for a graph drawing. We'll build on this structure throughout the tutorial to provide more rich and meaningful information reflective of real-life applications.

Graph Layout

The RaVis component provides for the layout of graphs using several algorithms, edge styles, and node renderings. The layout of node placements is determined by mathematical models based on radial, tree, physics and other computational logic. Figures 1-8 illustrate the different layout options which you can select from for the *best fit* presentation of your data.

For the purposes of the tutorial, we'll configure our application to use a *concentric radial layout*, based on a combination of methods known as radial layout and focus+context. This method provides an ability to interactively view a graph from different perspectives through the selection of a single node as the centre of focus. The user can

Listing 1. *Basic data source structure*

```
<graph>
<node id="n0"/>
<node id="n1"/>
<node id="n2"/>
<node id="n3"/>

<edge fromID="n0" toID="n1"/>
<edge fromID="n0" toID="n2"/>
<edge fromID="n0" toID="n3"/>
</graph>
```

navigate the graph by selecting any visible node as the focus node. The graph is then rearranged

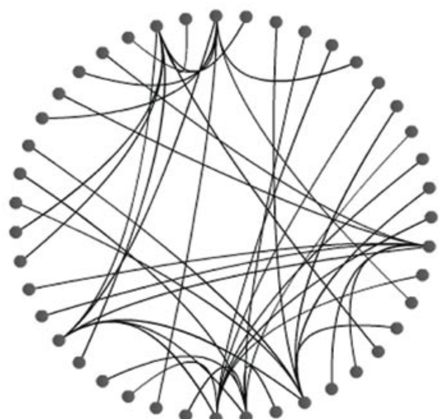


Figure 1. *Single-cycle circle*

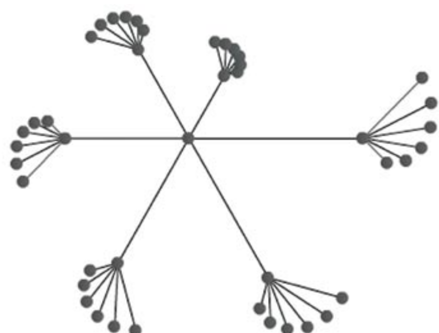


Figure 2. *Phyllotactic*

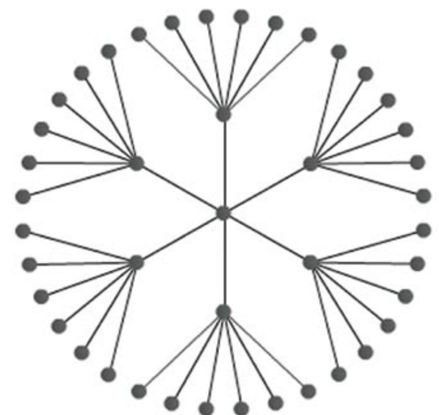


Figure 3. *Concentric radial*

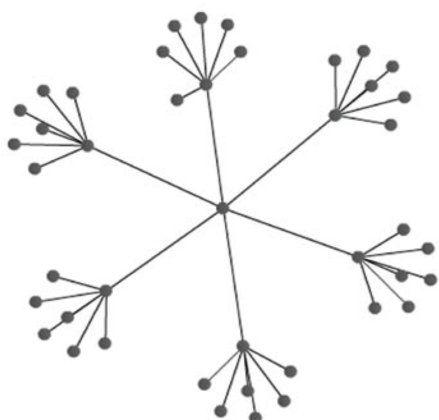


Figure 4. *Force-directed*

to reflect network distances from the newly chosen focus. Nodes are arranged on concentric rings around the focus node.

Let's begin by creating our main application file and adding the RaVis component. Listing 2 illustrates the configuration of a VisualGraph instance and the parameters required. The `script` body demonstrates the basic requirements for a graph with the following steps:

- Assign data source;
- Instantiate graph object;

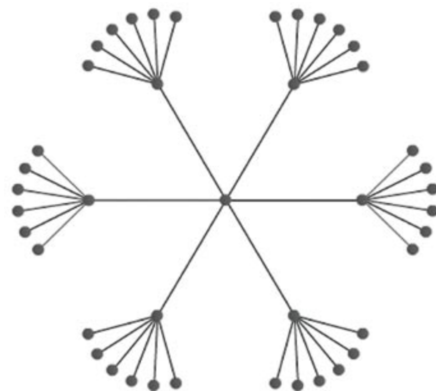


Figure 5. *Parent-centered radial*

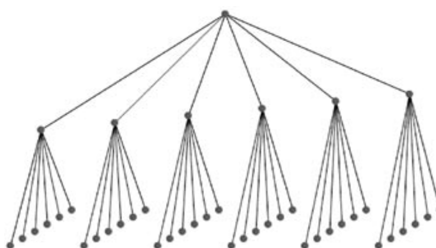


Figure 6. *Hierarchical tree*

Figure 7. *Hyperbolic*

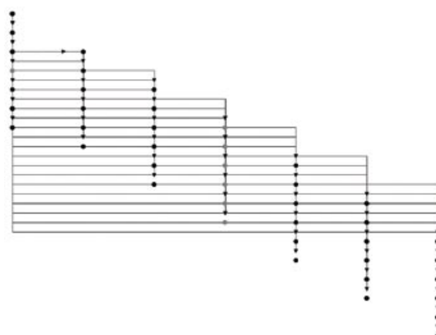


Figure 8. *Direct placement*

- Set layout type;
- Set edge renderer;
- Configure root node;
- Draw graph.

This code configures the VisualGraph component which we place like any other Flex mxml component, for example:

```
<ravis:VisualGraph id="vgraph" width="100%"
                   height="100%" ... />
```

You'll note that an `itemRenderer`, `edgeRenderer` and `edgeLabelRenderer` are specified. These objects can be customized to suit the visual design requirements of your application.

Node Renderer Configuration

Those familiar with the use of item renderers will understand quickly the design basis of configuring node renderers. A node renderer can be any Flex/ActionScript UI object such as a label, image or chart. For this tutorial we'll build a simple renderer based on a circle which we'll size and color based on data values; we'll also add a text label showing the name of the node.

Listing 3 illustrates the code for creating a basic node renderer component. Note the use of `nodeColor` and `nodeSize` as parameters taken from the data source.

Edge Configuration

A graph edge is the line drawn between two nodes. We can customize the look of an edge as well as add edge labels or graphics based on data parameters. We'll begin by configuring the edge renderer based on the preset options available from the RaVis library. Several edge renderers are included in the library such as straight, curved, flow, and orthogonal lines. Note the reference in Listing 2 for assigning the edge renderer:

```
vgraph.edgeRenderer = new
                   DefaultEdgeRenderer;
```

The `DefaultEdgeRenderer` is a simple straight line which can be additionally styled by a colour attribute from the data source. For example, the XML attribute `colour="0xFF000"` would colour the line red.

In addition to edge line style, it is often useful to include annotations or visual metaphors for characteristics specific to the relation. For example, is Node A *a friend of* Node B; or, does Node C have a flow value for it's link to Node D? The edge label renderer enables you to create a custom data renderer for displaying this data.

Listing 4 demonstrates the creation of a simple edge label component where a label is displayed at the half way point of the edge line:

```
<mx:Label text="{this.data.data.@edgeLabel
                        }" />
```

Similar to the node renderer, any UI object may be used as an edge label renderer where once again colour, size, text, or an icon can be specified via the source data.

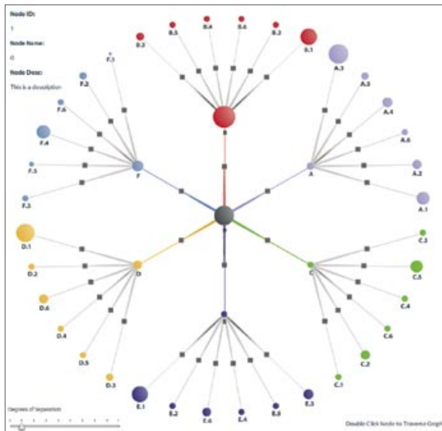


Figure 9. *Final Product*

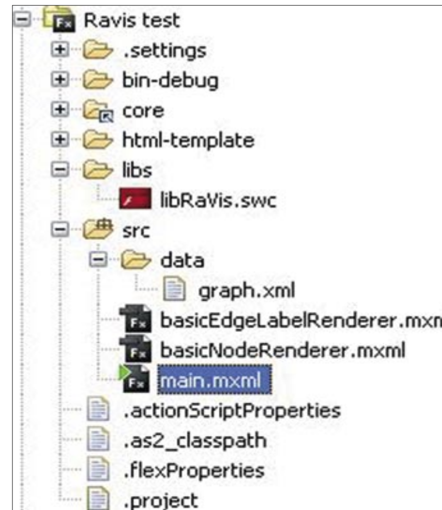


Figure 10. *Folder structure eclipse*

Controls and Events

Creating a dynamic and interactive visualization application should include the necessary controls and events to navigate, drill-down, and provide the means to explore, discover, and analyze. For this purpose we'll add a simple control to parameterize degrees of separation and an event to get details about a specific node.

Many of the necessary controls for RaVis are included as components. For example, to add a degrees of separation control, insert this code to your main application file:

```
<vc:DegreesOfSeparation id="myDOS" />
```

This control will enable users to specify the maximum distance to display among related nodes. This is useful for complex graphs with numerous links. Controlling degrees of separation is also useful for managing the performance of the application which is

Listing 2. *main.mxml*

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"

   xmlns:ravis="org.un.cava.birdeye.ravis.graphLayout.visual.*"
    creationComplete="init();" width="100%" height="100%" >

<mx:Script>
<![CDATA[
/* import the required classes */
    import org.un.cava.birdeye.ravis.graphLayout.visual.Default
                        EdgeRenderer;
    import org.un.cava.birdeye.ravis.graphLayout.layout.Concent
                        ricRadialLayouter;
    import org.un.cava.birdeye.ravis.graphLayout.data.Graph;

/* Init function, initiates the mapping of XML source data */

        private function init():void {
            initData(['Node','Edge','fromID','toID']);
        }

/* Build array of graph objects */

        private function initData(xN:Array):void {
            var xmlNames : Array= xN;

/* Init a graph object with the XML data */

        var graph:Graph= new org.un.cava.birdeye.ravis.graph
                        Layout.data.Graph("XMLAsDocsGraph"
                        ,false,xmldata,xmlNames);

/* Set the graph in the VGraph object, this automatically
                        initializes the VGraph items */

        /* vgraph is the id of the VisualGraph component */
        vgraph.graph = graph;

        /* set the default layouter type */
        var layouter:ConcentricRadialLayouter= new ConcentricRa
                        dialLayouter(vgraph);
        vgraph.layouter = layouter;

        /* set autofit */
        layouter.autoFitEnabled = true;

        /* set the edge renderer */
        vgraph.edgeRenderer = new DefaultEdgeRenderer;

        /* set the visibility limit options, default 2 */
        vgraph.maxVisibleDistance =  ;

        /* select a root node, most layouters requires a root
                        node */
        var startRoot = graph.nodeByStringId("1").vnode;

        /* set if edge labels should be displayed */
        vgraph.displayEdgeLabels = true;

        /* the following kicks it off .... */
        vgraph.currentRootVNode = startRoot;
        var initDone:Boolean = true;
        vgraph.draw();
    }
]]>
</mx:Script>

<ravis:VisualGraph
    id="vgraph" width="100%" height="100%" backgroundColor="#FF
                        FFFF" alpha="1"
    itemRenderer="basicNodeRenderer" edgeLabelRenderer="basicE
                        dgeLabelRenderer"
    visibilityLimitActive="true"
    >
</ravis:VisualGraph>
<mx:XML id="xmldata" source="data/graph.xml" />
</mx:Application>
```

Listing 3. basicNodeRenderer.mxml

```
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:local="org.un.cava.birdeye.ravis.assets.icons.primitives.*"
    creationComplete="setNodeCircle()"    >

    <mx:VBox verticalAlign="bottom" horizontalAlign="center" verticalGap="0" >
    <mx:Spacer height="11" />
    <local:Circle id="circle" toolTip="{this.data.data.@name}" />
<mx:Text id="nodeText" text="{this.data.data.@name}" width="75" height="11"
    textAlign="center" fontSize="10" />
    </mx:VBox>

    <mx:Script>
        <![CDATA[
            private function setNodeCircle():void {
            var nodeColor:int = this.data.data.@nodeColor;
            var nodeSize:int = this.data.data.@nodeSize;
            circle.color=nodeColor;
            circle.width=nodeSize;
            circle.height=nodeSize;
            }
        ]]>
    </mx:Script>
</mx:Canvas>
```

Listing 4. *basicEdgeLabelRenderer.mxml*

```
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    verticalAlign="middle" horizontalAlign="center" >

    <mx:Label width="50" height="8" text="{this.data.data.@edgeLabel}"
        color="#CCCCCC" />
</mx:VBox>
```

Listing 5. *graph.xml (sample data)*

```
<Node id="1" name="0" desc="Description for node 0" nodeColor="0x333333" nodeSize="30"
                       />

<Node id="2" name="A" desc="Description for node A" nodeColor="0xFF0000" nodeSize="25"
                       />
<Node id="3" name="B" desc="Description for node B" nodeColor="0x0000FF" nodeSize="15"
                       />
...

<Edge fromID="1" toID="2" edgeLabel="is friend of" color="0xFF0000" />
<Edge fromID="1" toID="3" edgeLabel="is associate of" color="0x0000FF" />
...
```

Listing 6. *getDetails Function*

```
    private function getDetails(event:Event):void {

    var nodeId:String = this.data.data.@id;
    var nodeName:String = this.data.data.@name;
    var nodeDesc:String = this.data.data.@desc

    // The main application detail pane is populated with the data

    parentDocument.nodeIdTxt.text=nodeId;
    parentDocument.nodeNameTxt.text=nodeName;
    parentDocument.nodeDescTxt.text=nodeDesc;
    }
```

heavily affected by the volume of displayed nodes.

It is often useful to offer a drill down of details and attributes for a specific node. We'll create a basic detail pane to display some information from a click event. Let's create a `getDetails` function to display additional attributes from the XML data source.

We'll start by editing the `basicNodeRender` where we'll add a linkbutton with a click event. This will be substituted with the current label:

```
<mx:LinkButton label="{this.data.d
   ata.@name}" click="getDetails()" />
```

Next we'll create the `getDetails` function which will trigger the function to populate the pane and display the id, name, and description attributes. Listing 6 illustrates some sample code for this function.

Final Product

Running the final application should yield a result similar to Figure 9. Double clicking any node will trigger an animated refocus of the selected node to center. Click the label link bar to view details about a given node. Try adjusting degrees of separation to restrict the depth of the graph to smaller or larger sets.

Next Steps

The possibilities for creating a visual representation of data within the Flex/ActionScript environment are endless. Check out the RaVisExplorer (included in the source project files) for a full demonstration of the possibilities. Using this tutorial as a base, try changing layouters, edge renderers, and customizing your nodes and edge labels using various components.

Readers interested in this field of development are invited to write the authors, and more importantly, to actively participate in the BirdEye library development and user group. Check out GeoVis, QaVis and the rest of the BirdEye library to explore new ways of visualizing your data.

And, stay tuned for Flex Information Visualization Part 2: Visualizing GeoSpatial Data where we'll explore the potential of map drawing.

http://code.google.com/p/birdeye/

JASON BELLONE

Jason leads the United Nations Centre for Advanced Visual Analytics (CAVA). The Centre sponsors the Flex-based BirdEye visualization library and develops visualization applications for the United Nations. http://cava.unog.ch

DANIEL LANG

Daniel is a staff member of the United Nations and is the principle developer of the BirdEye/RaVis component.