

JUNG: Java 平台网络/图应用开发的一种通用基础架构

编者按：众多需要处理多个对象间复杂关系的应用软件，例如网络管理、人际关系网络、思维脑图、地理导航以及多种游戏等，在本质上都是对“图”的计算，JUNG 就是专为图（Graph）的计算和可视化提供的一个通用的可扩充的 Java 编程平台。本文深入浅出地以多个实例对 JUNG 编程做了较为详细的讲解。您可以在本刊网站下载本文全部的样例源码。

摘要：以循序渐进的实例说明在 JUNG 架构下进行网络/图编程的方法。

关键词：JUNG, Java, 图, 网络, 树, 数据结构

1. 概述

JUNG (Java Universal Network/Graph framework) 是一个 Java 开源项目，其目的在于为开发关于图或网络结构的应用程序提供一个易用、通用的基础架构。使用 JUNG 功能调用，可以方便的构造图或网络的数据结构，应用经典算法（如聚类、最短路径、最大流量等），编写和测试用户自己的算法，以及可视化的显示数据的网络图。本文使用尽可能简明的代码示范基于 JUNG 应用开发方法，希望对有开发复杂网络/图应用需求的编程人员有所帮助。

2. 开发环境配置

本文使用 MyEclipse 7.5 作为开发环境，文中的例子皆在 JUNG 2.0.1 下调试通过。

JUNG 包可以在 SourceForge 下载，其地址为：

<http://sourceforge.net/projects/jung/files/>

JUNG 包中包含所有需要的 jar 文件，将其解压到一个目录中即可。

在 MyEclipse 中新建一个 Java 工程，并使用 Configure Build Path 菜单命令激活“Libraries”选项卡，然后使用“Add External JARs”按钮将 JUNG 包中的 Jar 文件全部设为系统类库，这样在该工程中建立的 Java 程序就可以访问 JUNG 所提供的接口方法及数据了；本文中的程序均在该工程内编写。

3. 创建图的数据结构

JUNG 提供的最基本的图结构的接口是 $Graph<V,E>$ ，其中 V 和 E 分别为顶点和边的数据类型。该接口提供创建图结构的基本方法，包括：

添加或者删除顶点和边；

获取所有顶点和边到集合类；

获取边和顶点的属性（如边的权重等）；

JUNG 支持一系列不同类型的图，以子接口的形式出现，例如：有向图 $DirectedGraph<V,E>$ ，森林 $Forest<V,E>$ ，树 $Tree<V,E>$ 和无向图 $UndirectedGraph<V,E>$ 等。

作为第一个简单例子，我们以实现 Graph 等接口的 $SparseGraph$ 为例，来构造一个图。 $SparseGraph$ 代表一个允许有向或无向边的稀疏图（如果你希望支持并行边，可以使用 $SparseMultigraph$ 类），使用 $addVertex$ 和 $addEdge$ 方法可以方便地添加顶点和边。下面的 `CreateGraph.java` 代码构造一个包含三个顶点和两个边的一个图（顶点和边的数据类型我们都定义为字符串；当然你也可以使用任何其它的数据类型）：

```
SparseMultigraph<String, String> g = new SparseMultigraph<String, String>();
g.addVertex("1");
g.addVertex("2");
g.addVertex("3");
g.addEdge("Edge-A", "1", "2");
g.addEdge("Edge-B", "2", "3");
System.out.println("The graph g = " + g.toString());
```

该段代码运行的结果为：

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

注意 `toString` 方法非常完整地显示出该图的所有信息。

如果要构造有向图，只需要在 `addEdge` 方法中包含 `edgeType` 参数即可，例如：

```
g.addEdge("Edge-C", "3", "2", EdgeType.DIRECTED);
```

添加一条从顶点“3”到顶点“2”的有向边。

4. 图的可视化表现

图的可视化表现是 JUNG 最为吸引人的地方。JUNG 可视化功能的基本组件是 `BasicVisualizationServer` 类（隶属 `edu.uci.ics.jung.visualization` 包），该类是 `Swing JPanel` 的一个子类，因此可以放置到任何一个 `Swing` 的容器对象中。

要构造一个 `BasicVisualizationServer` 类，首先需要了解 JUNG 中“布局（Layout）”的概念。一个 `Layout` 接口的实现，实际上就是要告诉 JUNG 如何来绘制图的顶点和边，即确定其在图形界面上的坐标位置(x,y)。使用预定义的 `Layout` 实现（例如 `edu.uci.ics.jung.algorithms.layout.CircleLayout`）可以方便的完成这个工作而不需要繁杂的手工编码。因此图 `g` 的可视化表现可以分以下 4 个步骤完成：

//1. 初始化图 `g`

//2. 使用该图创建布局对象

```
Layout<Integer, String> layout = new CircleLayout(g);
```

//3. 使用布局对象创建 `BasicVisualizationServer` 对象

```
BasicVisualizationServer<String, String> vv = new BasicVisualizationServer<String, String>( layout);
```

//4. 将上述对象放置在一个 `Swing` 容器中并显示之

```
frame.getContentPane().add(vv);  
frame.pack();  
frame.setVisible(true);
```

显示效果如下图所示，完整代码参见 `GraphView.java`。

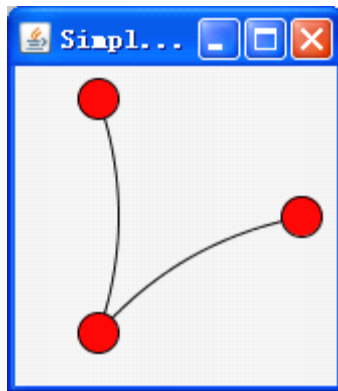


图 1 最简单的网络图显示

5. 网络图的修饰

图 1 的网络图中，没有关于顶点和边的描述信息。如果希望添加文字描述信息，或者修改边的线型和颜色等可视化属性，就需要理解变换器（`Transformer`）和渲染语境（`RenderContext`）两个概念。

变换器（`Transformer`）是来源于 `Apache Commons Collections` 开源项目的一个接口，它负责将进入集合中的对象变换成为另一种对象类型（例如提取关于这些对象的特定的信息）。例如，如果要确定顶点的绘图方式，可使用接口 `Transformer<V, Point2D>`，该接口负责将给定的顶点 `v` 转换成为一个 `Point2D` 类型的对象，其中包含了 `v` 的 (x, y) 坐标等信息。

`RenderContext` 负责 JUNG 渲染器（`Render`）的各项参数，并最终影响可视化的表现。`RenderContext` (定义在 `edu.uci.ics.jung.visualization`) 可以从 `BasicVisualizationServer` 对象得到，其影响的显示属性包括顶点颜色，边的线型，顶点和边的文本标签等。下面是几个常用的隶属于 `RenderContext` 的函数：

```
setVertexFillPaintTransformer()  
setEdgeStrokeTransformer()  
setVertexLabelTransformer()  
setEdgeLabelTransformer()
```

这些函数均需要使用相应的变换器作为参数，以将顶点和边转换为渲染器所需要的信息。以 `setVertexFillPaintTransformer()` 为例，它需要一个类型为 `Transformer<String, Paint>` 的参数，以根据顶点确定不同的 `Paint` 类型；注意：我们例子中的该变换器的第一个参数是 `String` 类型，这是因为它是由顶点的数据类型所决定的，而顶点的数据类型我们前面已经设为 `String` 了；系统可以根据这个 `String` 参数区分不同的顶点。假设我们希望让 John 节点显示为绿色，其它的节点显示黄色，则可以这样来构造这个变换器，注意观察其中的 `if` 代码是如何使用第一个参数来区别不同的顶点的：

```
Transformer<String, Paint> vertexPaint = new Transformer<String, Paint>() {  
    public Paint transform(String s) {  
        if (s.equals("John"))  
            return Color.GREEN;  
        else  
            return Color.YELLOW;  
    }  
};
```

`setVertexLabelTransformer()` 和 `setEdgeLabelTransformer()` 用于指定边和顶点的文本标签，其参数的数据类型分别为 `Transformer<V, String>` 和 `Transformer<E, String>`，其中 `V` 和 `E` 分别为顶点和边的数据类型。但是 JUNG 提供了一个非常简单的“工具变换器” `ToStringLabeller`（源于 `edu.uci.ics.jung.visualization.decorators`），用于利用顶点或者边的 `toString()` 方法得到一个相应的文本标签变换器；因此，下面的代码可以方便的设置顶点的标签而无需自行编码变换器：

```
BasicVisualizationServer 对象.renderContext().setVertexLabelTransformer(new ToStringLabeller());
```

完整的程序 `GraphView2.java` 的运行结果如图 2 所示。

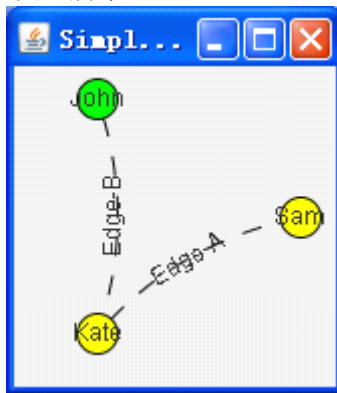


图 2 带颜色标记和文本标签的网络图

6. 自定义顶点和边的数据类型

上面的例子中顶点和边只有最简单的意义，也就是说，只有一个名字（一个字符串）。在实际应用中，节点和边往往具有复杂的数据意义，例如，在人际关系网络中，一个节点可能代表一个人或机构，具有名称、联系方式、隶属机构甚至包括照片等多媒体信息或附件。一个复合形式的顶点或边的数据结构可以是任何 `Java` 类，但一般说来起码应当提供 `id` 属性和 `toString()` 方法，前者用于区别不同的网络元素，后者一般用于为图形元素增加文本标签。作为一个例子，我们构造一个顶点为联系人（包含自增的 `id` 和名字属性）、边是联系方式的图；顶点类命名为 `People`，其中 `id` 是一个自增的属性，每次构造一个新的顶点其 `id` 会增 1；在构造函数中需要指定其姓名。

```
class People {  
    static int sid = 0;  
    int id;  
    String name;  
  
    People(String name) {  
        id = ++sid;  
        this.name = name;  
    }  
  
    public String toString() {
```

```

        return name + " - " + id;
    }
}

```

边的类命名为 `Link`，除了自增的 `id` 属性以外，`contact` 属性标记其联系方式。注意 `People` 和 `Link` 类都有 `toString` 方法以返回一个合适的文本标签。

```

class Link {
    static int sid = 0;
    int id;
    String contact;

    Link(String contact) {
        id = ++sid;
        this.contact = contact;
    }

    public String toString(){
        return contact+"(" +id+")";
    }
}

```

对于这种自定义的顶点和边，图和渲染类以及变换器的定义都需要做相应修改，以配合其数据类型。例如我们可以这样声明相关变量：

```

SparseGraph<People, Link> g;
BasicVisualizationServer<People, Link> vv;
Transformer<People, Paint> vertexPaint;
Transformer<Link, Stroke> edgeStrokeTransformer;

```

在增加 4 个顶点和四条边以后，样例程序 `ComplexGraph` 的输出如图 3 所示。

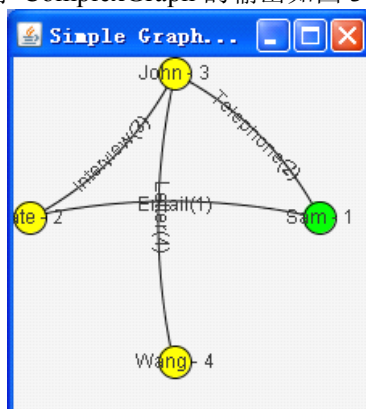


图 3 自定义顶点和边的数据类型

7. 从数据文件中读入图

如果图的连结信息存储在数据文件中，可以使用相应的函数将其转换为 `Graph` 类型。有多种存储图数据的方式，包括邻接矩阵与邻接表等；下面以邻接矩阵为例来读入图。

所谓邻接矩阵是这样定义的：设一个有 $n(n>0)$ 个顶点的图， $V(G)=\{v_1, v_2, \dots, v_n\}$ ，其邻接矩阵 AG 是一个 n 阶二维矩阵，该矩阵中如果 v_i 至 v_j 有一条边，则 (i, j) 项的值为 1，否则为 0。如下一个邻接矩阵：

```

0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 1
0 0 1 0 0 0 0 0 1

```



```

0000010000
0001000000
0100100100

```

对应的网络图如图 4 所示。

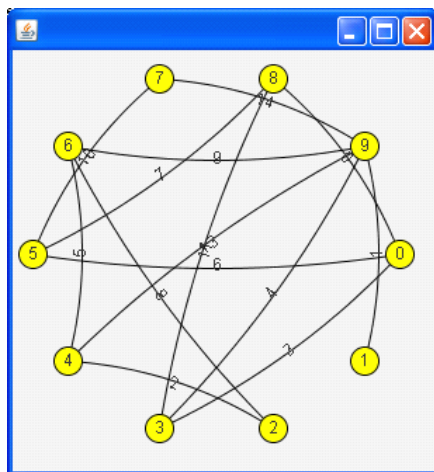


图 4 从邻接矩阵文件读入的图

下面我们就来从数据文件构造这个图。为了实现这个目的，我们首先要了解“工厂”（Factory）的概念。Factory 是定义在开源项目 `org.apache.commons.collections` 包下面的一个接口，用于自定义对象的创建过程（相当于一个简化的 Transformer），其核心是 `create()` 函数，此函数返回该工厂创建的对象。

读入邻接矩阵的函数是 `MatrixFile`，其原型为：

```

public MatrixFile(Map<E, Number> weightKey, Factory<? extends Graph<V, E>> graphFactory, Factory<V>
vertexFactory, Factory<E> edgeFactory)

```

该函数有四个参数，由于其 API 文档没有及时更新，**不能**提供正确的调用信息，我们只能通过分析其源码（主要代码在 `edu.uci.ics.jung.algorithms.matrix.GraphMatrixOperations.matrixToGraph()` 中）来研究其用法。首先看三个 Factory 参数：

graphFactory：用于创建邻接矩阵对应的图。假设我们需要一个 `SparseGraph` 对象，其顶点和边的数据类型分别为 `Integer` 和 `String`，则可以这样来初始化这个参数：

```

Factory<SparseGraph<Integer, String>> graphFactory;
graphFactory = new Factory<SparseGraph<Integer, String>>() {
    public SparseGraph<Integer, String> create() {
        return new SparseGraph<Integer, String>();
    }
};

```

上面的第一行代码，将 `graphFactory` 类型化（parameterize，或翻译为“参数化”）为 `SparseGraph` 类，而该 `SparseGraph` 的顶点和边分别为 `Integer` 和 `String` 类型。由于 `MatrixFile` 需要使用 `graphFactory` 来创建我们需要的图对象，因此在 `create` 函数（第三行）中，只需要简单的创建一个正确类型化的 `SparseGraph` 对象即可。

vertexFactory 用于创建顶点对象；由于在邻接矩阵中没有有关顶点的其他信息，我们只需要将顶点按顺序排序命名即可，因此我们的顶点采用 `Integer` 类型，并使用递增的静态变量做计数器：

```

Factory<Integer> vertexFactory;
Factory<String> edgeFactory;
static int vi = 0, ei = 0; // counter for creating vertices & edges
vertexFactory = new Factory<Integer>() {
    public Integer create() {
        return new Integer(vi++);
    }
};
edgeFactory = new Factory<String>() {
    public String create() {
        return "" + ei++;
    }
};

```

参数初始化完毕后，读入和创建图就非常简单了：

```
MatrixFile<Integer, String> mf  
    = new MatrixFile<Integer, String>(null, graphFactory, vertexFactory, edgeFactory);  
Graph g = mf.load("c:/g.txt");  
System.out.println(g);
```

注意 MatrixFile 的第一个参数暂且用不到，置为 null；后面处理有权图时需要设置这个参数；完整的代码请参见 ReadMatrixFile.java，控制台输出的运行结果为：

```
Vertices:0,1,2,3,4,5,6,7,8,9  
Edges:13[9,4] 14[9,7] 11[8,3] 3[3,0] 2[2,4] 1[1,9] 10[7,5] 0[0,8] 7[5,8] 6[5,0] 5[4,6] 4[3,9] 9[6,9] 8[6,2]
```

8. 应用关于图的算法

JUNG 中实现了多种关于图的算法，例如最短路径，k 近邻聚类，最小生成树等；当然你也可以编写自己的算法。下面以最短路径为例，演示如何使用 JUNG 进行图的计算和显示。

将 JUNG 中已经实现的内置算法，应用于一个图是很简单的。以上一节构造的无权图为例，应用 Dijkstra 算法求从顶点“0”到顶点“9”的最短路径的代码如下：

```
DijkstraShortestPath<Integer, String> alg = new DijkstraShortestPath<Integer, String>(g);  
List<String> l = alg.getPath(new Integer(0), new Integer(9));  
System.out.println("The shortest unweighted path from 0 to 9 is:");  
System.out.println(l.toString());
```

算法类 DijkstraShortestPath 的实例化同样是需要类型化为和图一致，然后使用 getPath 函数即可获取最短路径。控制台文本输出的信息为：

```
The shortest unweighted path from 0 to 9 is:  
[3, 4]
```

其意为通过边“3”和边“4”的路径是最短的。显然，如果能够使用图形化的表现方式来展现最短路径就更好了。为了达到这个目的，我们定义两个“笔画（stroke）”，其中 shortestStroke 使用粗笔画来显示最短路径：

```
final Stroke edgeStroke = new BasicStroke(1);  
final Stroke shortestStroke = new BasicStroke(4);//thick edge line!
```

然后创建一个笔画变换器，根据上面计算得到的最短路径来判断使用那种笔画：

```
Transformer<String, Stroke> edgeStrokeTransformer = new Transformer<String, Stroke>() {  
    public Stroke transform(String s) {  
        for(int i=0;i<l.size();i++){  
            if(l.get(i).equals(s))  
                return shortestStroke;  
        }  
        return edgeStroke;  
    }  
};  
vv.getRenderContext().setEdgeStrokeTransformer(edgeStrokeTransformer);
```

完整的代码请参见 ReadMatrixFile.java，如图 5 所示，最短路径的边使用粗线条表示。

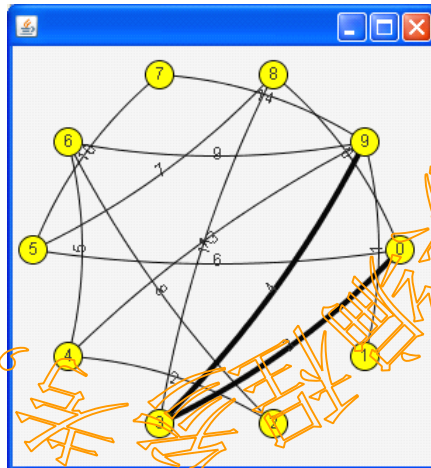


图 5 最短路径及其可视化表现

9. 从数据文件中读入带权值的图及最短路径计算

前文最短路径的计算中，没有考虑到边的权值；实际上，上面给出的邻接矩阵中也没有权值的信息（所有的边的权都默认为 1）。我们先看看计算带权的最短路径的方法：

构造一个将边变换为其权值的 Transformer，并将该变换器传递给算法的构造函数

```
DijkstraDistance(Hypergraph<V,E> g, Transformer<E, Number> nev)
```

即可，变换器的构造方法如下述代码所示（假设我们的边还是字符串类型，eWeights 是一个 Map，存储有边及其对应的权值的映射）：

```
// converts a string (of an edge) to the edge's weight
Transformer<String, Double> nev = new Transformer<String, Double>() {
    public Double transform(String s) {
        Number v = (Number) eWeights.get(s);
        if (v != null)
            return v.doubleValue();
        else
            return 0.0;
    }
};
```

除了可以获得最短路径的边之外，我们还可以获得最短路径的值：

```
Number dist = alg.getDistance(new Integer(0), new Integer(9));
```

```
System.out.println("and the length of the path is: " + dist);
```

上面的讨论是基于我们已经构造了有权图 g 和权值映射 eWeights。但是，如果希望以类似 MatrixFile 的形式从邻接矩阵中直接读入非 1 的权值，则需要一些更为技巧性的工作，直接使用下述代码是不行的：

```
MatrixFile<Integer, String> mf
```

```
= new MatrixFile<Integer, String>(eWeights, graphFactory, vertexFactory, edgeFactory);
```

因为直到目前的版本 (2.0.1)，JUNG 在 MatrixFile 读入数据时，并没有使用到它的第一个参数“Map<E, Number> weightKey”（可以认为这是一个还没有消除的 Bug）。

如何解决这个问题呢？通过研究其源码，我们使用如下的技术来间接使用 JUNG 的接口：

首先在程序开始时初始化一个 Map 用于存储边/权映射；因为 Map 是一个接口，为方便起见，我们使用 Hashtable 作为其实现：

```
Hashtable<String, Number> eWeights = new Hashtable<String, Number>();
```

然后使用如下代码从矩阵数据 C:/g2.txt 中读入有权图：

```
Graph<Integer, String> g;
try {
    BufferedReader reader =
        new BufferedReader(new FileReader("c:/g2.txt"));
    DoubleMatrix2D matrix = createMatrixFromFile(reader);!!!!
    g = GraphMatrixOperations.<Integer,String>matrixToGraph(matrix,
        graphFactory, vertexFactory, edgeFactory,eWeights);!!!!
    reader.close();
} catch (IOException ioe) {
    throw new RuntimeException("Error in loading file " + "c:/g2.txt", ioe);
}
```

矩阵数据 g2.txt 的内容假设为：

```
0 0 0 0 0 0 0 0 3 0
0 0 0 0 0 0 0 0 0 5
0 0 0 0 2 0 0 0 0 0
3 0 0 0 0 0 0 0 0 8
0 0 0 0 0 0 7 0 0 0
1 0 0 0 0 0 0 0 9 0
0 0 3 0 0 0 0 0 0 6
0 0 0 0 0 3 0 0 0 0
```

```
0002000000
0900500600
```

请注意上面代码中带下划线的两个部分。函数 `createMatrixFromFile()` 是从 `MatrixFile` 类的源码中复制过来的，因为该函数是一个 `private` 类型的方法，不能直接调用，我们只能采用“复制粘贴”的形式来使用它：

```
private DoubleMatrix2D createMatrixFromFile(BufferedReader reader)
    throws IOException
{
    List<List<Double>> rows = new ArrayList<List<Double>>();
    String currentLine = null;
    while ((currentLine = reader.readLine()) != null) {
        StringTokenizer tokenizer = new StringTokenizer(currentLine);
        if (tokenizer.countTokens() == 0) {
            break;
        }
        List<Double> currentRow = new ArrayList<Double>();
        while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            currentRow.add(Double.parseDouble(token));
        }
        rows.add(currentRow);
    }
    int size = rows.size();
    DoubleMatrix2D matrix = new SparseDoubleMatrix2D(size, size);
    for (int i = 0; i < size; i++) {
        List<Double> currentRow = rows.get(i);
        if (currentRow.size() != size) {
            throw new IllegalArgumentException(
                "Matrix must have the same number of rows as columns");
        }
        for (int j = 0; j < size; j++) {
            double currentVal = currentRow.get(j);
            if (currentVal != 0) {
                matrix.setQuick(i, j, currentVal);
            }
        }
    }
    return matrix;
}
```

而 `matrixToGraph` 则可以接收一个 `Map` 型的参数，以将各个边的权值存入该映射中。

最后，我们希望在可视化界面中显示边的权值，而不是在生成边时的顺序号；要做到这一点只需要增加一个 `EdgeLabelTransformer` 即可：

```
Transformer<String, String> ev = new Transformer<String, String>() {
    public String transform(String s) {
        Number v = (Number) myApp.eWeights.get(s);
        if (v != null) {
            return "----"+v.intValue();
        }
        else
            return "";
    }
};
vv.getRenderContext().setEdgeLabelTransformer(ev);
```

注意这个变换器和本节开始处的权值变换器非常类似，因为他们的功能都是通过边的到权值，只是对变换器的返回结果类型有不同要求。

预期在未来的 JUING 版本中，我们可以通过简单地功能调用：


```
MatrixFile<Integer, String> mf
    = new MatrixFile<Integer, String>(eWeights, graphFactory, vertexFactory, edgeFactory);
Graph g = mf.load("C:/g2.txt");
```

来实现上述读入有权邻接矩阵的目的。完整程序参见 ReadMatrixFile2.java，其显示结果为：

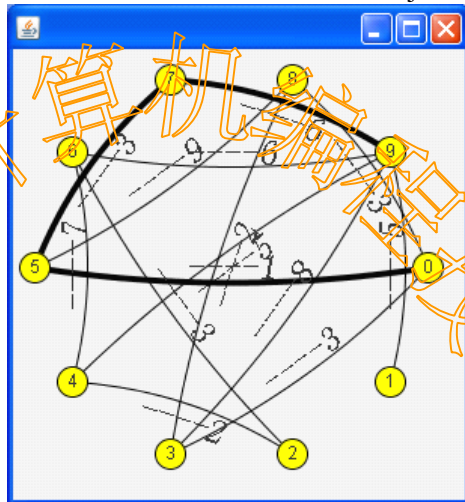


图 6 带权图的最短路径

为了清晰起见，图 6 在权值的前面加了些“----”以将不同的边的权值区别开来。

10. 响应鼠标输入对图进行变换

JUNG 支持的图的变换，包括缩放（Scalling）、移动（Transforming）、选择（Picking）、编辑（Editing）等。如果要使用这些默认的鼠标交互操作，只需要将 BasicVisualizationServer 升级为支持鼠标交互的 VisualizationViewer，并且添加下面代码即可：

```
// this is how we activate the mouse!
// Create a graph mouse and add it to the visualization component
DefaultModalGraphMouse<Integer, String> gm = new DefaultModalGraphMouse();
gm.setMode(DefaultModalGraphMouse.Mode.PICKING);
vv.setGraphMouse(gm);
```

这里面的核心概念是所谓的“模态图鼠标（ModalGraphMouse）”接口，定义在 edu.uci.ics.jung.visualization.control 包中。该接口定义了与图交互的鼠标操作应该支持的行为，其常用的实现包括 DefaultModalGraphMouse<V,E>，AnnotatingModalGraphMouse<V,E>，EditingModalGraphMouse<V,E>等；后两者用于交互式批注和编辑。如果只需要进行普通的视图变换，使用 DefaultModalGraphMouse 即可，如上例所示。

使用这样的代码，我们可以对图 6 那样 CircleLayout 自动生成的布局，用鼠标选择并拖动顶点，改变图的表现，使之变成图 7 所示的模样。你可以试试滚动鼠标的滚轮，看看有什么别的效果，或者按下 Shift 键或者 Ctrl 键，看看选择顶点的时候又有什么效果！

一般说来，Picking 状态下系统内置的交互操作包括：

- 1)直接左键选择并拖动：移动顶点。
- 2)Shift+左键选择：选择多个顶点。
- 3)Ctrl+左键选择：使被选择的顶点居中。
- 4)滚轮：缩放视图。

而在 Transforming 模式下内置的交互操作包括：

- 1)直接左键选择并拖动：移动视图。
- 2)Shift+左键拖动：旋转视图。
- 3)Ctrl+左键拖动：切变（扭曲）视图。
- 4)滚轮：缩放视图。

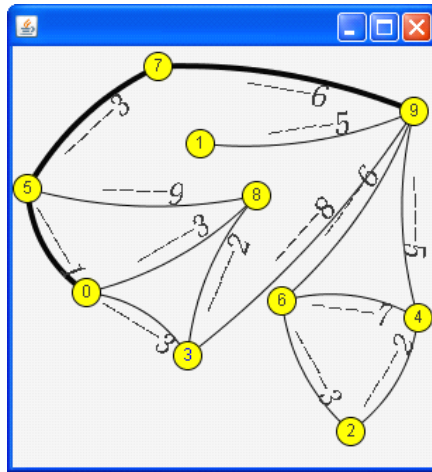


图 7 使用 Picking 模式选择并拖动顶点改变图的布局

简单交互的完整程序请参见 `Interact.java`，该程序基本上是从 `ReadMatrixFile2.java` 修改而来的。如果还希望在运行时切换选择（Picking）和移动（Transforming）模式，可以将 `DefaultModalGraphMouse` 内置的键盘监听器添加到系统中即可：

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

该代码支持在运行时点击“p”键切换到选择模式，点击“t”键切换到移动模式。

但是如果出于某种目的，你不希望使用 `DefaultModalGraphMouse` 内置的全部交互功能，而只希望使用其中一部分，例如平移和缩放，则可以使用如下的方式构造一个“可插入图鼠标（`PluggableGraphMouse`）”，并插入需要的“鼠标插件”即可：

```
PluggableGraphMouse gm = new PluggableGraphMouse();
gm.add(new TranslatingGraphMousePlugin(MouseEvent.BUTTON1_MASK));
gm.add(new ScalingGraphMousePlugin(new CrossoverScalingControl(), 0, 1.1f, 0.9f));
vv.setGraphMouse(gm); // vv is the VisualizationViewer Object
```

`PluggableGraphMouse` 可以容纳各种鼠标插件（mouse plugin）以实现特定的交互任务。所谓鼠标插件，就是一个实现了定义在 `edu.uci.ics.jung.visualization.control` 中的 `GraphMousePlugin` 接口的类。JUNG 提供了多种现成的鼠标插件，例如 `TranslatingGraphMousePlugin`、`ScalingGraphMousePlugin` 等等，我们只需要构造若干合适的鼠标插件并将其添加到 `PluggableGraphMouse` 中即可；如上例所示，则应用程序只接收平移和缩放操作，而忽略其他交互请求。

11. 图的可视化编辑

图的编辑实际上是鼠标交互的一种特殊情况，也就是要使用 `EditingModalGraphMouse` 鼠标插件。因此，我们将 `ReadMatrixFile.java` 稍作修改，保留边和顶点的工厂函数，去掉从文件中读入图的部分，仅构造一个空的 `SparseGraph`，并添加下述代码：

```
// !!!this is how we can edit the graph!!!
EditingModalGraphMouse<Integer, String> gm
    = new EditingModalGraphMouse<Integer, String>(vv.getRenderContext(), vertexFactory, edgeFactory);
vv.setGraphMouse(gm);
```

程序 `EditGraph.java` 的运行截图参见图 8。

之所以 `EditingModalGraphMouse` 要使用创建顶点和边的工厂类，是因为该鼠标插件的功能就是要动态地添加顶点和边。（附注：原来的 `CircleLayout` 布局也改成了简单的静态布局 `StaticLayout`，JUNG 文档中说明这种布局中顶点的位置由一个 `Map` 对象初始化，并且也可以在以后从该映射中获取；但是直到本文所用版本，该静态布局仅仅只是上级类 `AbstractLayout` 的一个简单继承）。图 8 所示截图中已经添加了 5 个顶点（鼠标左键单击即可添加节点），正在顶点 4 和 3 之间添加边。

`EditingModalGraphMouse` 实际上是由多个鼠标插件组合而成的，包括 `PickingGraphMousePlugin`、`TranslatingGraphMousePlugin`、`ScalingGraphMousePlugin`、`EditingGraphMousePlugin`、`AnnotatingGraphMousePlugin`、`EditingPopupGraphMousePlugin` 等，这些鼠标插件的不同组合形成三种不同的模式：Transforming、Picking、Editing；这三种模式可以通过 `EditingModalGraphMouse.setMode()` 方法来设置，也可以通过一个内置的菜单（称为

ModeMenu) 来设置。将 ModeMenu 添加到窗体菜单条上的代码是:

```
// *****this is how we can toggle mouse mode
// 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);
```

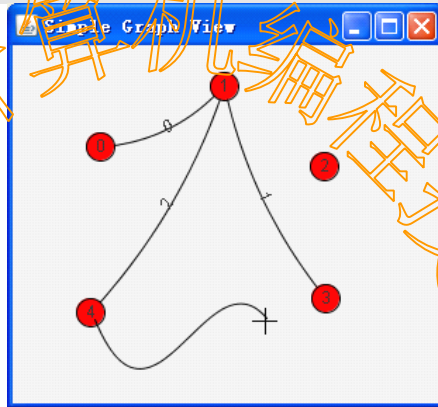


图 8 正在顶点 4 和 3 之间添加边的操作的截图

需要说明的是, `EditingModalGraphMouse` 内置支持鼠标右键菜单, 但是直到版本 2.0.1, 这个菜单**仍有 Bug**, 多次点击时会生成重复的菜单项。而且在很多时候, 我们需要根据自定义的数据类型来决定弹出菜单的项目, 需要自定义属性编辑菜单。下面以一个更为复杂和全面的例子来介绍自定义的显示和编辑模式。

12. 自定义顶点形状

下面以一个“脑图 (mind map)”应用为例, 来看看图的显示和编辑中各种自定义模式的实现。

所谓脑图, 即各种概念或想法之间的关系的网络图, 是一种常用的思考问题和辅助决策的工具; 一个简单的英文单词脑图可以象图 9 那样: 围绕单词 `mind`, 有多个相关的单词节点; 显然这样的脑图对于单词的理解和记忆会有帮助的。下面我们先来构造这样一个脑图的数据结构。

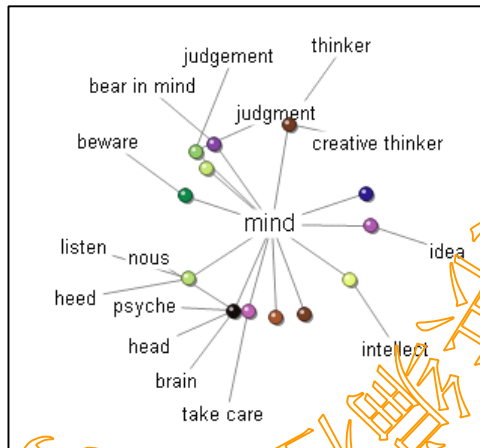


图 9 一个关于单词的脑图构造样例

假设在我们的脑图中, 首先, 顶点类命名为 `Idea`, 边类命名为 `Link`, 其定义分别在源码 `Idea.java` 和 `Link.java` 中。在 `Idea` 中, 除了最基本的 `id` (数值标识), `title` (文本标识) 等字段外, 我们还引入一个 `Vector` 类型 `links` 字段 (用于存储该顶点到其他相关顶点的链接), 以及相应的添加联接的函数 `addLink()`。除此之外, 还提供一个工厂类用于自动创建一个空的 `Idea` 对象, 其中的静态方法用于方便的获得该工厂的实例。主要代码如下:

```
public class Idea {
    static long _id = 0; // seed for generating id
```

```

long id;// unique id of this idea
String title, brief;// essential data
Vector<Link> links;// pointer to other nodes

public Idea(String title, String brief, Vector<String> keywords) {
    this.id = _id++;
    this.title = title;
    this.brief = brief;
    if (keywords == null)
        this.keywords = keywords;
    else
        this.keywords = new Vector<String>();
    this.links = new Vector<Link>();
}

public void addLink(Link link){
    this.links.add(link);
}

public String toString(){//good habit!
    return title;
}
}

class IdeaFactory implements Factory<Idea> {
    static IdeaFactory instance = new IdeaFactory();
    public Idea create() {
        return new Idea("", "", new Vector<String>());
    }
    public static IdeaFactory getInstance() {
        return instance;
    }
}

```

类似的，Link 的定义如下，其中的 LinkFactory 与 Idea 中的类似，从略：

```

public class Link {
    static long _id = 0;// seed for generating id
    long id;// unique id
    Idea toIdea;
    String title, brief;

    public Link(Idea toIdea, String title, String brief) {
        this.id = _id++;
        this.toIdea = toIdea;
        this.title = title;
        this.brief = brief;
    }

    public String toString(){
        return title;
    }
}

class LinkFactory implements Factory<Link> { //略
}

```

使用上述结构的顶点和边的定义，可以按下面方式进行：

```
Idea id1 = new Idea("mind", "", null);
```



```
Idea id2 = new Idea("idea", "", null);
```

```
Link l12 = new Link(id2, "create", "");  
id1.addLink(l12);
```

```
g.addVertex(id1);  
g.addEdge(l12, id1, id2);
```

上面代码只初始化了两个顶点和一条边；现在我们采用自定义的模式来显示这样结构的图。一种简单的方法是使用 JUNG 系统内置的“顶点形状工厂”（定义在 `edu.uci.ics.jung.visualization.util` 包中）获得需要的形状，例如方形、多边形等。该工厂的构造形式为：

```
VertexShapeFactory(Transformer<V,Integer> vst, Transformer<V,Float> varf)
```

也就是说，我们需要两个变换器参数来构造的该工厂，其中 `vst` 负责从顶点获取顶点显示尺寸（size），`varf` 负责从顶点获取高宽比（aspect ratio）。由于我们希望在顶点中使用 16 点的字体显示单词（英文字符大约宽 8 像素），因此我们设置顶点形状的显示尺寸为：字符数*8+16，其中 16 是两端的留空值。据此公式我们可以设计出顶点到显示尺寸的变换器：

```
// 1. transformer from vertex to its size (width)  
Transformer<Idea, Integer> vst = new Transformer<Idea, Integer>() {  
    public Integer transform(Idea i) {  
        int len = i.toString().length();  
        if (len < 3)  
            len = 3;  
        return new Integer(len * 8 + 16);  
    }  
};
```

根据顶点的宽度 `len`，我们容易推导出，如果希望顶点的高度是 20 像素的话，则需要的高宽比应该是 $2/len$ ，因此，我们设计出高宽比变换器为：

```
// 2. transformer from vertex to its shape's "aspect ratio"  
Transformer<Idea, Float> varf = new Transformer<Idea, Float>() {  
    public Float transform(Idea i) {  
        int len = i.toString().length();  
        if (len < 3)  
            len = 3;  
        return new Float(2.0 / len);  
    }  
};
```

有了上面两个参数，就可以构造出需要“顶点形状工厂”了：

```
//3. create the shape factory  
final VertexShapeFactory<Idea> vsf = new VertexShapeFactory<Idea>(vst, varf);
```

我们真正需要的，是一个“顶点形状变换器”，因此，使用上述工厂，可以构造系统支持度圆形、方形、星型等的变换器；以圆角方形为例，我们的 vertex shape transformer 为：

```
// 4. EASY way to have a "vertex shape transformer"  
Transformer<Idea, Shape> vstr = new Transformer<Idea, Shape>() {  
    public Shape transform(Idea i) {  
        return vsf.getRoundRectangle(i);  
    }  
};
```

最后，将该形状变换器置入渲染语境中即可：

```
//5. put the shape transformer to render context, done!  
vv.getRenderContext().setVertexShapeTransformer(vstr);
```

程序的运行结果如图 10 所示。

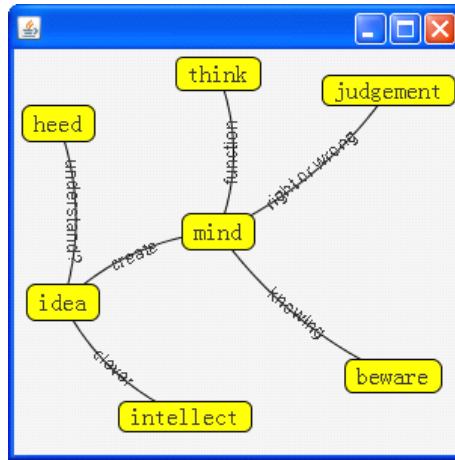


图 10 内置的圆角矩形顶点形状

由于使用 `setVertexShapeTransformer` 可以接受到任意形状的变换器，所以可以实现任意形状的顶点显示；例如，如果你对 Java2D 比较熟悉，完全可以使用自己编码来实现上述效果。因此图 10 的第二种实现方式为：

```

// Manually create a custom vertex shape
vstr = new Transformer<Idea, Shape>() {
    public Shape transform(Idea i) {
        int len = i.toString().length();
        if (len < 4)
            len = 4;
        //Arc2D.Double r = new Arc2D.Double();
        //r.setArc(-len * 5, -len * 3, len * 10, len * 6, 60, 240, Arc2D.PIE);
        RoundRectangle2D.Double r =
            new RoundRectangle2D.Double(-len * 5, -10, len * 10, 20, 10, 10);
        return r;
    }
};
vv.getRenderContext().setVertexShapeTransformer(vstr);

```

你可以试试使用上面代码中带下划线的两行来替代后面的关于 `r` 的定义，看看有何效果——著名的吃豆子的小怪物出现了！

`RenderContext` 中可以设置的 `Transformer` 有 7 项之多，例如：

```

Transformer<V, Icon> getVertexIconTransformer();
用于设置顶点图标。因此，我们可从图像文件构造 ImageIcon 对象，并通过一个图标变换器传递给渲染器：
final ImageIcon ii=new ImageIcon("shy.jpg");
Transformer<Idea, Icon> vit=new Transformer<Idea,Icon>(){
    public Icon transform(Idea arg0) {
        return ii;
    }
};
vv.getRenderContext().setVertexIconTransformer(vit);
vv.getRenderer().getVertexLabelRenderer().setPosition(Position.E);

```

上文中最后一行的作用是将顶点文本显示在图标的右方（东方）。图片文件 `shy.jpg` 放在工程文件夹中；效果见图 11。这里我们使用了统一的图标，您当然可以为不同的顶点提供不同的图标。本节程序的源码在 `Ex.java`，请注意其中包含了上面所说的 3 种方式的代码，您需要注释掉其中的一些语句才能分别显示其中一种。

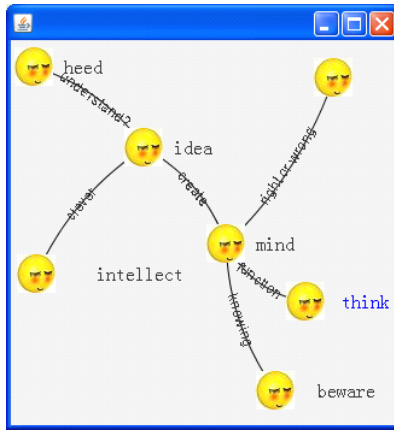


图 11 使用图标表示顶点

除了使用 Transformer 来设置渲染语境外，还可以直接构造自己的渲染器，例如 DefaultVertexLabelRenderer 是一个以 JLabel 为基类的渲染器，它可以使用 JLabel 的丰富功能来顶点的文本标签，例如多行文本，HTML 支持，图标支持等等。例如下面的渲染器可以产生如图 12 的效果：

```
//vertex label renderer with icon & html
DefaultVertexLabelRenderer vlr = new DefaultVertexLabelRenderer(Color.BLUE){
    public <Idea> Component getVertexLabelRendererComponent(JComponent vv, Object value,
        Font font, boolean isSelected, Idea vertex) {
        super.getVertexLabelRendererComponent(vv, value, font, isSelected, vertex);
        setIcon(new ImageIcon("shy.jpg"));
        this.setBorder(BorderFactory.createEtchedBorder());
        setText("<html>title:<br><b>" + vertex.toString() + "</b></html>");
        return this;
    }
};
vv.getRenderContext().setVertexLabelRenderer(vlr);
```

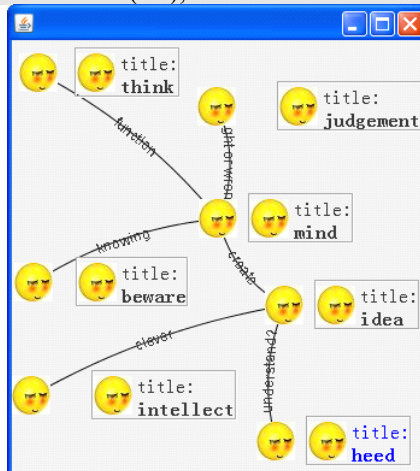


图 12 使用自定义的顶点标签渲染器

注意我们需要继承 DefaultVertexLabelRenderer 并覆盖 getVertexLabelRendererComponent() 方法；该方法也是 VertexLabelRenderer 接口中定义的唯一一个需要实现的方法。

图 12 看上去有些乱，但只要注意到每个顶点都有两个图标，第一个是顶点本身的图标，由前文所讲的顶点图标变换器 VertexIconTransformer 生成；第二个图标是顶点标签的一部分（你可以看到我故意加上的 JLabel 边框），由自定义的渲染器生成。需要注意的是，顶点标签的图标不能替代顶点本身，因为如果顶点如果不显示的话，则该图无法响应鼠标的“选择顶点”的操作；另外，如果有顶点图标的话，JUNG 就不再绘制顶点的形状 (Shape)，也就是说，如果你同时提供了顶点图标和顶点形状两个变换器，则 JUNG 优先使用图标变换器。

13. 自定义弹出式编辑菜单

JUNG 的鼠标支持可以通过“鼠标插件”实现。实际上，我们已经用过的支持编辑功能的“图鼠标” `EditingModalGraphMouse` 就是由多个 `MousePlugin` 组合而成，并根据需要添加和移除某个 `plugin`；例如，“选择”（picking）模式的源代码为：

```
protected void setPickingMode() {
    remove(translatingPlugin);
    remove(rotatingPlugin);
    remove(shearingPlugin);
    remove(editingPlugin);
    remove(annotatingPlugin);
    add(pickingPlugin);
    add(animatedPickingPlugin);
    add(labelEditingPlugin);
    add(popupEditingPlugin);
}
```

从中可以看到，选择“picking”模式，实际上就是移除 `translating`（平移）等 5 个 `plugin`，激活 `picking` 等 4 个 `plugin`。所有的这些 `plugins` 都是继承 `AbstractPopupGraphMousePlugin` 而来的，而后者是 `GraphMousePlugin` 接口的一个最简单的实现。通过将各种 `mouse plugins` 安装到 `PluggableGraphMouse` 对象中，可以构造自己的鼠标支持。因此，构造一个自定义的鼠标插件，也需要按这样的思路来进行：

- 1) 继承 `AbstractPopupGraphMousePlugin`；
- 2) 重载其中的抽象函数 `handlePopup(MouseEvent e)` 方法以处理弹出菜单。

好在 Greg Bernstein（JUNG 项目组成员之一）已经给我们实现了一个通用的处理顶点和边的弹出菜单的插件：

`PopupVertexEdgeMenuMousePlugin.java`

使用该插件的最简单过程是：

- 1) 先构造一个内置的图鼠标（例如 `EditingModalGraphMouse`），并移除其中的弹出菜单支持；
- 2) 构造一个 `PopupVertexEdgeMenuMousePlugin` 对象，并设置其中的顶点和边弹出菜单；
- 3) 将上一步中的构造好的插件添加到第(1)步构造的图鼠标中。

样例代码如下（见 `Ex1.java`），其中的核心语句加了下划线：

```
// mouse plugin demo
EditingModalGraphMouse<Idea, Link> gm =
    new EditingModalGraphMouse<Idea, Link>(vv.getRenderContext(),
        IdeaFactory.getInstance(), LinkFactory.getInstance());
PopupVertexEdgeMenuMousePlugin<Idea, Link> myPlugin =
    new PopupVertexEdgeMenuMousePlugin<Idea, Link>();
JPopupMenu edgeMenu = new JPopupMenu("Vertex Menu");
JPopupMenu vertexMenu = new JPopupMenu("Edge Menu");
edgeMenu.add(new JMenuItem("edge!"));
vertexMenu.add(new JMenuItem("vertex!"));
myPlugin.setEdgePopup(edgeMenu);
myPlugin.setVertexPopup(vertexMenu);
// Removes the existing popup editing plugin!
gm.remove(gm.getPopupEditingPlugin());
// Add our new plugin to the mouse
gm.add(myPlugin);
vv.setGraphMouse(gm);
```

试试分别在顶点和边上点击鼠标右键，会弹出不同的菜单。很显然，上面例子中的弹出菜单仅仅区分了顶点和边，没有体现具体的被点击对象。要想构造上下文敏感的弹出菜单，需要先研究一下

`PopupVertexEdgeMenuMousePlugin.java` 中重载的弹出菜单处理方法：

```
protected void handlePopup(MouseEvent e) {
    final VisualizationViewer<V,E> vv =
        (VisualizationViewer<V,E>)e.getSource();
    Point2D p = e.getPoint();

    GraphElementAccessor<V,E> pickSupport = vv.getPickSupport();
    if(pickSupport != null) {
```



```

final V v = pickSupport.getVertex(vv.getGraphLayout(), p.getX(), p.getY());
if(v != null) {
    // System.out.println("Vertex " + v + " was right clicked");
    updateVertexMenu(v, vv, p);
    vertexPopup.show(vv, e.getX(), e.getY());
} else {
    final E edge = pickSupport.getEdge(vv.getGraphLayout(), p.getX(), p.getY());
    if(edge != null) {
        // System.out.println("Edge " + edge + " was right clicked");
        updateEdgeMenu(edge, vv, p);
        edgePopup.show(vv, e.getX(), e.getY());
    }
}
}
}
}

```

该代码的主要职责，是在鼠标点击时，确定事件发生的位置、被点击的对象（是边还是顶点），并根据此信息构造菜单内容并弹出菜单（见上文中带下划线的代码）；其中最为重要的是 updateVertexMenu() 或 updateEdgeMenu() 方法，这个方法用于在菜单弹出前重新构造菜单内容。正如上述代码的作者所说，我们不需要修改 handlePopup() 方法，而只需要修改 updateVertexMenu() 或 updateEdgeMenu() 这两个函数即可。

以顶点菜单的更新为例，其函数形式为：

```
private void updateVertexMenu(V v, VisualizationViewer vv, Point2D point)
```

该函数有三个参数，分别是当前被选中的顶点 *v*，当前图的显示部件 *vv*，以及鼠标点击的位置 *point*。根据这些参数，我们可以编写自己的 updateVertexMenu() 函数，以初始化其中的 *vertexPopup* 和 *edgePopup* 两个变量；这个变量都是 *JPopupMenu* 类型，其类型声明为：

```
private JPopupMenu edgePopup, vertexPopup;
```

作为示范，我们在顶点菜单中设置两个菜单项：删除当前节点和切换“是否计划项目 (*isSchedule*)”。菜单的构造和事件响应都放在 *VertexPopupMenu.java* 中，其核心函数为：

```

public static JPopupMenu update(final Idea v,
    final VisualizationViewer<Idea, Link> vv, Point2D point) {
    //1. Clear the menu
    m.removeAll();

    // 2. "delete" menu
    String title = v.title;
    JMenuItem mi = new JMenuItem("Delete [" + title + "]");
    mi.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            vv.getGraphLayout().getGraph().removeVertex(v);
            vv.repaint();
        }
    });
    m.add(mi);

    // 3. "schedule" checkbox menu
    final JCheckBoxMenuItem mic = new JCheckBoxMenuItem("Schedule");
    mic.setSelected(v.isSchedule);
    mic.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            v.isSchedule = mic.isSelected();
        }
    });
    m.add(mic);

    return m;
}

```

上述代码可分为三个部分：

第一步从菜单中移除所有菜单项，以便重新构造菜单；

第二步构造“删除当前顶点”菜单：顶点的名称从参数 *v* 的 *title* 字段中得到；

第三步构造复选框式的“是否属于计划项目”菜单：如果该顶点是计划项目（字段 *isSchedule* 为真），则复选框初始状态为“选中”。

这样，在 *PopupVertexEdgeMenuMousePlugin.java* 的 *updateVertexMenu()* 方法中，只需要加入对上面方法的调用即可：

```
private void updateVertexMenu(V v, VisualizationViewer vv, Point2D point) {  
    vertexPopup = VertexPopupMenu.update((Idea) v, vv, point);  
}
```

Ex2.java 的执行结果如图 13 所示。正如预料的那样，第一次点击 *mind* 顶点的时候，弹出菜单中 *Schedule* 复选框未被选中；点击该菜单项后，则第二次弹出菜单时该复选框是出于选中状态。另外，由于我们没有为边编写代码，因此在边上点击鼠标右键将抛出异常。

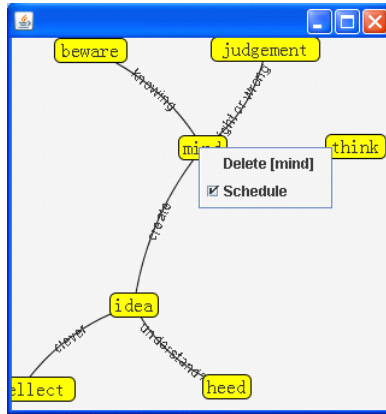


图 13 上下文敏感的弹出菜单

14. 树的实现及计算

树 (Tree) 和森林 (Forrest) 是图的特例。JUNG 为这两种数据结构提供了专用的接口以实现其相关的算法。以 *DelegateTree* 的树实现为例，这种树直接以有向图为基础实现 *Tree* 接口，其常用的函数包括：

```
public boolean addVertex(V vertex): 添加根节点;  
public boolean addChild(E edge, V parent, V child): 添加子节点;  
public boolean addEdge(E e, V v1, V v2): 同上。
```

下述代码将构造如图 14 所示的树，其中的布局为 *TreeLayout*：

```
DelegateTree<String, Integer> g = new DelegateTree<String, Integer>();  
g.addVertex("V0");  
g.addEdge(edgeFactory.create(), "V0", "V1");  
g.addEdge(edgeFactory.create(), "V0", "V2");  
g.addEdge(edgeFactory.create(), "V1", "V4");  
g.addEdge(edgeFactory.create(), "V2", "V3");  
g.addEdge(edgeFactory.create(), "V2", "V5");  
g.addEdge(edgeFactory.create(), "V4", "V6");  
g.addEdge(edgeFactory.create(), "V4", "V7");  
g.addEdge(edgeFactory.create(), "V3", "V8");  
g.addEdge(edgeFactory.create(), "V6", "V9");  
g.addEdge(edgeFactory.create(), "V4", "V10");  
TreeLayout layout = new TreeLayout<String, Integer>(g);  
(其它代码略)
```

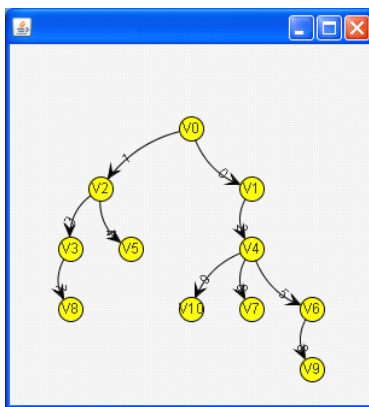


图 14 树及 TreeLayout 布局

15. 结语

JUNG 对图、树、森林等数据结构的计算和可视化提供了完整的基础支持，使程序员可以从繁杂的底层编程中解放出来，从而大大的提高相关应用软件的实现进度。同时 JUNG 在图树算法、交互支持、可视化布局等方面的强大功能，更可以使程序员如虎添翼；不足的地方是，其 API 的文档不够齐全，有些内容甚至严重过期，因此需要参考其源码（可以在 [Source Forge](http://sourceforge.net) 下载）进行工作。本文对 JUNG 的应用编程做了较为全面的讲解，并对如何参考源码来实现特定的功能也做了示范。<http://jung.sourceforge.net/applet/index.html> 中更有多个内容丰富的样例可供参考；源码包中同时也包含这些样例的源码，可供直接学习套用。
