Visualização Gráfica de Grafos com a API JUNG

Grafos podem ser usados para representar relacionamentos em diversos tipos de redes: sociais, de computadores, de tráfego e outros tipos de dados relacionais. Frequentemente é necessário representar grafos de forma visual para ilustrações ou análises. Neste artigo, veremos como usar a API JUNG para visualizar conjuntos de dados representados como grafos.

Este artigo complementa o artigo "Introdução à Representação e Análise de Grafos com a API JUNG", publicado na Edição 49 da revista MundoJ.

Grafos são estruturas que servem para representar muitos objetos reais de forma natural: redes sociais e relações entre objetos em geral, redes de computadores e estradas, hierarquias de dados em vários tipos de sistema e muitos outros conceitos podem ser representados como grafos. Em um artigo na última edição vimos como representar grafos e executar análises simples em Java usando a API JUNG (Java Universal Network/Graph Framework). Neste artigo, veremos como usar a mesma API para visualizar estes grafos.

Grafos são estruturas de dados que representam objetos e relações entre eles. Grafos podem ser usados para representar vários tipos de conceitos e objetos do mundo real, como, por exemplo, relações em redes sociais reais ou virtuais, links em documentos na Web, rotas de tráfego de veículos ou de redes de computadores e muitos outros. Grafos são conjuntos de vértices (que correspondem aos objetos que queremos representar) e arestas, que ligam pares de vértices e correspondem às relações entre os objetos.

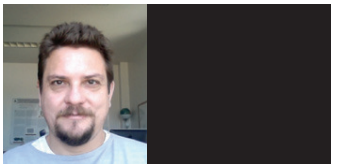
No artigo "Introdução à Representação e Análise de Grafos com a API JUNG", publicado na Edição 49 da revista MundoJ, apresentamos a API JUNG (Java Universal Network/Graph Framework), que permite a representação de grafos em Java como coleções de vértices e arestas. Além de classes para representar grafos de diversos tipos, a API contém implementações de vários algoritmos para análise dos grafos e operações como criação de subconjuntos dos grafos. Neste artigo veremos outras classes da API que permitem a visualização dos grafos usando também diversos algoritmos para determinação das posições e aparência de seus vértices e arestas.

Visualização de grafos pode ser usada para ilustração simples: é mais fácil entender os objetos e suas relações em um grafo quando este é representado graficamente do que pela lista de vértices e arestas. Visualização também pode ser usada para análises visuais mais complexas: alguns fenômenos podem ser mais bem compreendidos ou identificados através da visualização dos grafos, em especial quando associamos outras informações sobre vértices e arestas à representação visual dos grafos.

Visualização de grafos é uma tarefa mais complexa do que aparenta ser: grafos com grande número de arestas e vértices podem ter uma representação visual muito densa e até incompreensível. Adicionalmente não existe uma única forma de visualizar um grafo – os vértices podem ser posicionados de muitas formas diferentes, o que afeta também a aparência das arestas que conectam estes vértices. Embora alguns tipos específicos de grafos possam ser visualizados de forma natural (árvores, por exemplo), muitos tipos de grafos podem ser representados graficamente de forma diferente, mesmo tendo a

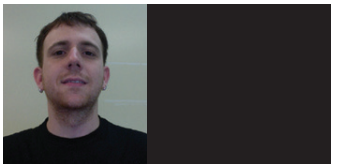
Rafael Santos | rafael.santos@lac.inpe.br

é doutor em Inteligência Artificial pelo Instituto Tecnológico de Kyushu, Japão. É tecnologista do Instituto Nacional de Pesquisas Espaciais, atuando em pesquisa e desenvolvimento de aplicações e técnicas de mineração de dados, processamento de imagens e inteligência artificial aplicada, e presentemente é pesquisador visitante da Universidade Johns Hopkins, nos Estados Unidos. É autor do livro "Introdução à Programação Orientada a Objetos usando Java" e de várias palestras e tutoriais sobre Java.



André Grégio | argregio@cti.gov.br

é mestre em Computação Aplicada pelo Instituto Nacional de Pesquisas Espaciais. É tecnologista do Centro de Tecnologia da Informação Renato Archer (CTI), atuando em pesquisa e desenvolvimento na área de segurança de sistemas de informação.



mesma representação como conjuntos de vértices e arestas. Isto é exemplificado na figura 1, que mostra quatro layouts ou representações gráficas diferentes para o mesmo grafo (mesmo conjunto de vértices com o mesmo conjunto de arestas conectando estes vértices).

Este artigo apresenta seções mostrando como criar aplicações para visualização básica de grafos com a API JUNG, comenta sobre os layouts disponíveis na API (que determinam como os vértices de um grafo serão posicionados na visualização), e apresenta diversos exemplos de como a aparência gráfica de vértices e arestas podem ser modificados para visualização do grafo.

Visualização básica de grafos com a API JUNG

A API JUNG contém vários métodos que facilitam a visualização de grafos de diversos tipos, permitindo também a definição da aparência gráfica de vértices e arestas de forma bastante flexível, o layout (distribuição) dos vértices e arestas usando vários algoritmos diferentes ou mesmo de forma manual e algumas formas de interação do usuário com a representação visual do grafo.

A API JUNG foi criada de forma a facilitar bastante a criação de aplicações para visualização de grafos, usando algumas classes básicas para as tarefas mais comuns e permitindo a customização de vários aspectos através da criação de instâncias de classes específicas com tarefas bem definidas.

Para visualizar um grafo representado na API JUNG precisamos primeiro de uma instância de classe que implementa a interface Graph. Os passos para a criação de uma representação visual do grafo são:

1. Usando a instância que representa o grafo, criamos um layout para a representação gráfica deste. Layouts em JUNG são implementados por instâncias de classes que herdam de AbstractLayout (que por sua vez implementa a interface Layout) e que devem ser declaradas com os mesmos parâmetros de tipo usa-

Instalando a API JUNG

A API JUNG permite a representação e processamento de grafos em Java de forma eficiente e flexível. Os arquivos da API podem ser copiados de http://sourceforge.net/projects/jung/files/. A versão mais recente em setembro de 2011 é a 2.0.1. A API é distribuída como um arquivo .zip, que contém 17 arquivos .jar, que devem ser adicionados ao classpath da máquina virtual Java ou do projeto que usa a API. Para adicionar estes arquivos .jar em um projeto desenvolvido com a IDE Eclipse, clique no nome do projeto com o botão direito do mouse, selecionando o menu Build Path / Add External Archives e escolhendo os nomes dos arquivo .jar no diálogo de seleção. Estas instruções são as mesmas apresentadas no artigo complementar na Edição 49.

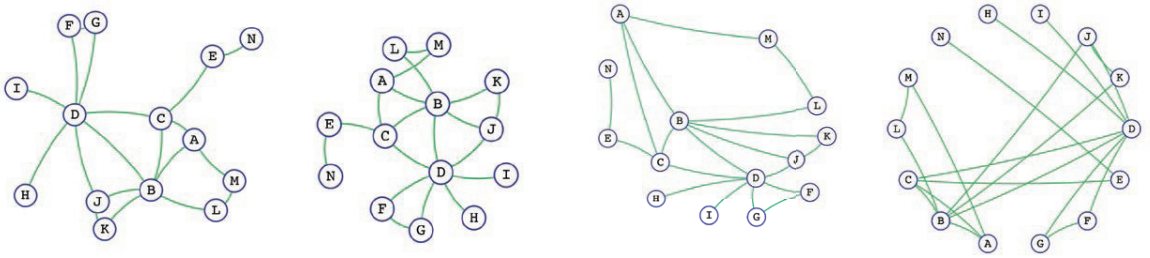


Figura 1. *Quatro representações gráficas do mesmo grafo.*

dos para declarar a instância que representa o grafo.
2. Usando a instância que representa o layout, criamos uma instância de VisualizationViewer, classe que herda indiretamente de JComponent e que será usada como componente gráfico na criação da interface gráfica para visualização.
3. Opcionalmente recuperamos o método getRenderContext() da instância de VisualizationViewer para acessar o objeto correspondente ao contexto de renderização. Este objeto possui métodos para definir, através de classes transformadoras, a aparência dos vértices e arestas no grafo.
4. Opcionalmente associamos comportamentos ao componente de visualização para responder a gestos do mouse – isto é usado para permitir transformações como pan e zooming no grafo e seleção e modificação das arestas, entre outras funções.
5. Usamos a instância de VisualizationViewer para compor a interface gráfica da aplicação.

Para ilustrar estes passos vejamos um exemplo simples baseado em um grafo apresentado no artigo "Introdução à Representação e Análise de Grafos com a API JUNG", publicado na Edição 49. Para facilitar a leitura, o trecho de código que cria este grafo é mostrado na Listagem 1. O grafo criado tem vértices e arestas do tipo String, e é o mesmo grafo usado para criar as visualizações mostradas na figura 1.

Listagem 1. Trecho de código que cria um grafo no qual vértices e arestas são Strings.

```
// Este método cria um grafo simples para uso em alguns
exemplos deste artigo.
public static Graph<String,String> criaGrafo() {
   //Criamos um grafo onde vértices e arestas são Strings
   Graph<String,String> grafo =
      new UndirectedSparseGraph<String,String>();
   // Criamos as arestas e vértices simultaneamente.
   grafo.addEdge("A-B","A","B"); grafo.addEdge("A-C","A","C");
   grafo.addEdge("A-M","A","M"); grafo.addEdge(
      "B-C","B","C");
   grafo.addEdge("B-D","B","D"); grafo.addEdge(
      "B-J","B","J");
   grafo.addEdge("B-K","B","K"); grafo.addEdge(
      "B-L","B","L");
   grafo.addEdge("C-D","C","D"); grafo.addEdge(
      "C-E","C","E");
   grafo.addEdge("D-F","D","F"); grafo.addEdge(
      "D-G","D","G");
   grafo.addEdge("D-H","D","H"); grafo.addEdge(
      "D-I","D","I");
   grafo.addEdge("D-J","D","J"); grafo.addEdge(
      "E-N","E","N");
   grafo.addEdge("F-G","F","G"); grafo.addEdge("J-K","J","K");
   grafo.addEdge("L-M","L","M");
   return grafo;
}
```

Os passos para a criação da visualização do grafo são mostrados na Listagem 2 (para manter as listagens concisas, os passos opcionais, usados para definir diversos aspectos visuais do grafo, não são implementados nesta versão do código. A visualização criada pela Listagem 2 pode ser vista na figura 2).

Listagem 2. Primeira versão da classe para visualização do grafo criado na Listagem 1.

```
// Classe para criar uma visualização básica de um grafo.
public class VisBasica {
   public VisBasica() {
      // O grafo é criado em outra classe, para organizar
      // melhor o código.
      Graph<String,String> grafo = CriaGrafo1.criaGrafo();
      // Passo 1: criamos uma instância de classe para
      // controlar o layout.
      AbstractLayout<String,String> layout =
         new KKLayout<String,String>(grafo);
      // Passo 2: criamos o componente para visualização.
      VisualizationViewer<String,String> componente =
         new VisualizationViewer<String,String>(layout);
      //Modificamos características do componente
      componente.setPreferredSize(
         new Dimension(750,750));
      componente.setBackground(Color.WHITE);
      //Passo 5: usamos o componente para montar a GUI
      JFrame f = new JFrame("Visualização de Grafos");
      f.getContentPane().add(componente);
      f.pack();
      f.setVisible(true);
      f.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE);
   }

   // Executa a classe como uma aplicação.
   public static void main(String[] args) {
      new VisBasica(); // Monta a GUI.
   }

}
```

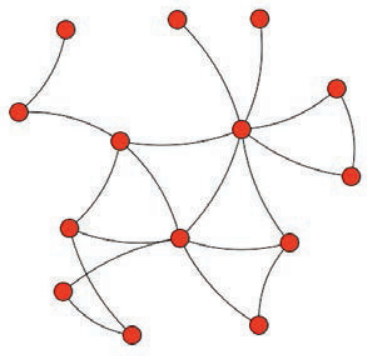


Figura 2. *Visualização criada pelo código na Listagem 2.*

A figura 2 mostra uma visualização bastante básica do grafo criado na Listagem 1 – alguns passos

que permitem a customização da aparência dos vértices e arestas não foram executados, para manter a primeira versão do código simples.

Podemos observar que as arestas e vértices da figura 2 estão em posições diferentes das arestas e vértices mostradas nas diversas visualizações da figura 1, embora o grafo usado nas figuras seja o mesmo. Existem duas razões para estas diferenças: a primeira é porque escolhemos classes diferentes para representar e definir o layout que será usado para visualização do grafo; a segunda é que alguns dos layouts implementados na API JUNG dependem de inicialização das posições das arestas, que pode ser aleatória em alguns casos. Em outras palavras, mesmo usando a mesma classe para implementar o layout podemos ter diferenças na aparência final da visualização (basta executar a classe mostrada na Listagem 2 algumas vezes para ver a variação dos resultados). Existem soluções na própria API para resolver este problema, que serão vistas em outras seções neste artigo.

Alguns layouts para visualização de grafos

Ao criar visualizações de grafos é importante considerar que tipo de layout deve ser usado, e que alterações nas aparências das arestas e vértices devem ser feitas para que o resultado seja o mais próximo do esperado. A definição do tipo de layout é simples para alguns tipos de grafos, como árvores e florestas ou organogramas e fluxogramas, mas não é trivial para outros tipos de grafos, em especial os genéricos. Nestes casos, sugere-se tentar diferentes layouts, verificando que implementação da API produz os resultados mais agradáveis ou interpretáveis.

Alguns layouts implementados na API JUNG e suas respectivas classes são listados a seguir. Detalhes específicos das classes não serão apresentados: o leitor pode obter mais informações (inclusive referências bibliográficas sobre os algoritmos de layout correspondentes às classes) na documentação da API.

- **CircleLayout:** faz com que os vértices sejam distribuídos em um círculo, com distância igual entre eles. A quarta visualização mostrada na figura 1 foi criada usando uma instância desta classe. Não realiza cálculos interativos para determinar a posição final do vértice, e não usa posicionamento aleatório inicial. O raio do círculo é determinado automaticamente, mas pode ser definido pelo usuário através do método setRadius.
- **FRLayout:** usa o algoritmo iterativo Fruchterman-Reingold para determinar as posições finais dos vértices através de cálculos de atração e repulsão entre vértices. O usuário pode mudar o comportamento do algoritmo modificando as constantes de atração e repulsão e o número máximo de iterações. A API JUNG possui também a classe FRLayout2, que contém uma versão melhorada, mas ainda experimental do algoritmo.
- **KKLayout:** usa o algoritmo iterativo Kamada-Kawai para determinar as posições finais dos vértices através de uma heurística que considera os comprimentos dos vértices que conectam as arestas do grafo.
- **SpringLayout**: usa um método iterativo para posicionar os vértices, que tenta aproximar e afastar grupos de arestas considerando os vértices que as unem. A execução do algoritmo que calcula o layout das arestas é consideravelmente lenta, sendo possível ver a sua modificação durante a exibição do componente de visualização.
- **ISOMLayout:** inicializa as coordenadas dos vértices de forma aleatória e usa uma variante do Mapa Auto-Organizável de Kohonen (um tipo de rede neural) para calcular as posições finais dos vértices. Embora sua execução seja iterativa e lenta este layout pode ser adequado para visualização de grafos esparsos.
- **TreeLayout:** usado para visualizar florestas ou conjuntos de árvores (não pode ser usado diretamente para visualizar grafos como os usados nos exemplos deste artigo). Modifica o posicionamento inicial dos vértices para espalhar as árvores da floresta na área do componente gráfico.
- **RadialTreeLayout:** similar ao TreeLayout, tenta organizar as árvores da floresta em círculos concêntricos, com as raízes das árvores no centro do componente.
- **StaticLayout:** permite ao programador determinar posições específicas para cada vértice. Este layout garante consistência entre execuções de uma aplicação: os vértices estarão sempre na posição definida pelo programador. É possível usar uma função específica para determinar a posição de cada vértice, ou explicitar as coordenadas de cada vértice, mas isto pode ser impraticável para grandes grafos. Um exemplo de uso desta classe será apresentado neste artigo.

A classe AbstractLayout, ancestral de todas as classes que implementam algoritmos de layout de grafos (mas não de árvores e floresta) provê dois métodos interessantes que podem ser usados para forçar o posicionamento de vértices: o método setLocation recebe como parâmetros uma instância da classe usada para representar o vértice e uma instância da classe Point2D e usa as coordenadas deste ponto para inicializar a posição para aquele vértice. Como alguns dos algoritmos de layout são iterativos, ao especificar a posição de um vértice devemos também executar o método lock, passando como argumentos o nome do vértice e a constante booleana true, para garantir que durante a modificação iterativa das coordenadas as coordenadas daquele vértice não serão modificadas.

Modificando a aparência de vértices e arestas

Para uma visualização eficiente de grafos é essencial ter a capacidade de identificar os diferentes vértices e arestas e de possivelmente representar vértices e arestas de forma diferente dependendo de seu tipo ou valor. Para que vértices e arestas sejam desenhados de forma diferenciada é preciso fornecer classes transformadoras para a instância que representa o componente, através de seu contexto de renderização. Estas classes transformadoras retornarão instâncias de outras classes que determinam a aparência do vértice ou aresta.

Para fazer isto é preciso seguir estes passos:

1. Obter o contexto de renderização do componente através da chamada ao método getRenderContext da classe VisualizationViewer, que retorna uma instância que implementa a interface RenderContext (com mesmos parâmetros de tipo do grafo).
2. Usar o contexto de renderização para associar as classes transformadoras desejadas através dos métodos setXXXTransform (em que XXX é o nome de uma característica de renderização do grafo – alguns destes métodos serão apresentados em exemplos).
3. Criar classes transformadoras cujas instâncias serão usadas como argumentos para os métodos setXXXTransform.

Como regra básica, uma classe transformadora deve implementar a interface Transformer (parte da API Apache Commons-Collections, distribuída como parte da API JUNG) que é declarada com dois parâmetros de tipo, o primeiro correspondendo a classe que faz parte do grafo (vértice ou aresta) e o segundo correspondendo à classe ou interface que deve ser retornada como resultado da transformação. As classes transformadoras devem implementar o método transform, que deve criar e retornar instâncias das classes resultantes da transformação. Como um exemplo simples, uma classe para retornar a cor que será usada para desenhar a aresta de um grafo onde arestas são representadas por inteiros deverá implementar a interface Transformer<Integer,Paint> e o método transform, que recebe como argumento uma instância de Integer e retorna uma instância de classe que implementa Paint.

Alguns dos métodos da interface RenderContext que podem ser usados para modificar a aparência visual do grafo renderizado são (nestes exemplos usaremos as classes fictícias Vértice e Aresta para substituir as classes correspondentes aos vértices e arestas do grafo):

- **setArrowDrawPaintTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Aresta,Paint>. O método transform desta classe recebe como argumento uma instância de Aresta e retorna uma instância de classe que implementa Paint, que determina como as setas em arestas em grafos dirigidos serão desenhadas.
- **setArrowFillPaintTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Aresta,Paint>. O método transform desta classe recebe como argumento uma instância de Aresta e retorna uma instância de classe que implementa Paint, que determina como as setas em arestas em grafos dirigidos serão preenchidas.
- **setEdgeArrowStrokeTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Aresta,Stroke>. O método transform desta classe recebe como argumento uma instância de Aresta e retorna uma instância de classe que implementa Stroke, que determina o traço que será usado para desenhar setas em arestas em grafos dirigidos.
- **setEdgeFillPaintTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Aresta,Paint>. O método transform desta classe recebe como argumento uma instância de Aresta e retorna uma instância de classe que implementa Paint, que determina o preenchimento que será usado para desenhar arestas nos grafos.
- **setEdgeStrokeTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Aresta,Stroke>. O método transform desta classe recebe como argumento uma instância de Aresta e retorna uma instância de classe que implementa Stroke, que determina o traço que será usado para desenhar arestas nos grafos.
- **setEdgeFontTransformer**, que recebe como argumento uma instância de classe que imple-

menta Transformer<Aresta,Font>. O método transform desta classe recebe como argumento uma instância de Aresta e retorna uma instância da classe Font, que determina a fonte que será usada para desenhar rótulos associados às arestas do grafo.

» **setEdgeLabelTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Aresta,String>. O método transform desta classe recebe como argumento uma instância de Aresta e retorna uma instância de String correspondente ao rótulo que será associado à aresta na visualização do grafo.

» **setEdgeDrawPaintTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Aresta,Paint>. O método transform desta classe recebe como argumento uma instância de Vértice e retorna uma instância de classe que implementa Paint, que corresponde ao tipo e parâmetros de preenchimento que serão usados para desenhar as arestas na visualização. Um exemplo de classe que pode ser usada como argumento para este método é mostrada na Listagem 10.

» **setVertexShapeTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Vertice,Shape>. O método transform desta classe recebe como argumento uma instância de Vértice e retorna uma instância de classe que implementa Shape, que determina como os vértices serão representados graficamente. Praticamente qualquer forma pode ser usada para a representação gráfica do vértice, mas para melhor aparência visual os objetos retornados devem ser centralizados na coordenada (0,0) (veja exemplo na Listagem 4).

» **setVertexLabelTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Vertice,String>. O método transform desta classe recebe como argumento uma instância de Vértice e retorna uma instância de String, que será o rótulo usado para identificar os vértices na visualização. Qualquer classe pode ser usada para representar vértices na API JUNG, mas é adequado que esta classe tenha um método toString para facilitar a representação do vértice como rótulo para visualização. Veja exemplo de classe usada como argumento para este método na Listagem 5.

» **setVertexFontTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Vertice,Font>. O método transform desta classe recebe como argumento uma instância de Vértice e retorna uma instância de Font, que corresponde à fonte usada para desenhar o rótulo do vértice na visualização. Veja exemplo de classe usada como argumento para este método na Listagem 6.

» **setVertexStrokeTransformer,** que recebe como argumento uma instância de classe que implementa Transformer<Vertice,Stroke>. O método transform desta classe recebe como argumento uma instância de Vértice e retorna uma instância de classe que implementa Stroke, que corresponde ao traço usado para desenhar o símbolo do vértice na visualização. Um exemplo de classe usada como argumento para este método é mostrado na Listagem 7.

» **setVertexFillPaintTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Vertice,Paint>. O método transform desta classe recebe como argumento uma instância de Vértice e retorna uma instância de classe que implementa Paint, que corresponde ao tipo e parâmetros de preenchimento que serão usados para desenhar o símbolo do vértice na visualização. Um exemplo de classe usada como argumento para este método pode ser visto na Listagem 8.

» **setVertexDrawPaintTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Vertice,Paint>. O método transform desta classe recebe como argumento uma instância de Vértice e retorna uma instância de classe que implementa Paint, que corresponde ao tipo e parâmetros de desenho que serão usados para desenhar o símbolo do vértice na visualização. Um exemplo de classe usada como argumento para este método é mostrado na Listagem 9.

» **setVertexIconTransformer**, que recebe como argumento uma instância de classe que implementa Transformer<Vertice,Icon>. O método transform desta classe recebe como argumento uma instância de Vértice e retorna uma instância de classe que implementa Icon, que corresponde a um ícone que pode ser usado para representar o vértice graficamente.

Um exemplo simples de visualização de grafos com alguns dos parâmetros visuais definidos pelo programador pode ser visto nas listagens a seguir: neste exemplo modificaremos vários atributos visuais do grafo criado na Listagem 1. O exemplo é mostrado na Listagem 3.

Listagem 3. Segunda versão da classe para visualização do grafo criado na Listagem 1.

```
public class VisDetalhada {
   public VisDetalhada() {
      // O grafo é criado em outra classe, para organizar
      // melhor o código.
      Graph<String,String> grafo = CriaGrafo1.criaGrafo();
```

```
        // Passo 1: criamos uma instância de classe para
        // controlar o layout.
        ISOMLayout<String,String> layout =
            new ISOMLayout<String,String>(grafo);
        // Passo 2: criamos o componente para visualização.
        VisualizationViewer<String,String> componente =
            new VisualizationViewer<String,String>(layout);
        // Passo 3: Modificamos o processo de renderização
        // através de classes transformadoras.
        RenderContext<String, String> ctx =
            componente.getRenderContext();
        ctx.setVertexShapeTransformer(
            new TransformaFormaDosVertices());
        ctx.setVertexLabelTransformer(
            new TransformaRotulosDosVertices());
        ctx.setVertexFontTransformer(
            new TransformaFontesDosVertices());
        ctx.setVertexStrokeTransformer(
            new TransformaLinhasDosVertices());
        ctx.setVertexFillPaintTransformer(
            new TransformaPreenchimentoDosVertices());
        ctx.setVertexDrawPaintTransformer(
            new TransformaCorDasLinhasDosVertices());
        ctx.setEdgeDrawPaintTransformer(
            new TransformaCorDasLinhasDasArestas());
        // Modificamos também a posição do rótulo dos
        // vértices para aparecer no centro dos ícones dos
        // vértices.
        VertexLabel<String, String> vl =
            componente.getRenderer().
            getVertexLabelRenderer();
        vl.setPosition(Renderer.VertexLabel.Position.CNTR);
        // Modificamos algumas caract. do componente.
        componente.setPreferredSize(
            new Dimension(640,640));
        componente.setBackground(Color.WHITE);
        //Passo 5:usamos o componente para montar a GUI.
        JFrame f = new JFrame("Visualização de Grafos");
        f.getContentPane().add(componente);
        f.pack();
        f.setVisible(true);
        f.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
    }

    // Executa a classe como uma aplicação.
    public static void main(String[] args) {
        new VisDetalhada(); // Monta a GUI.
    }

}
```

A classe mostrada na Listagem 3 é semelhante à da Listagem 2, com algumas diferenças significativas: através dos métodos setXXXTransformer do contexto de renderização modificamos a forma dos vértices, os seus rótulos e a fonte usada para desenhá-los, as linhas (traços e cores) e o preenchimento usados nos vértices, e a cor das linhas que representam as arestas. Cada chamada a método setXXXTransformer recebeu como parâmetro uma instância de classe criada especificamente para isto, e que implementa a interface Transformer com os parâmetros de tipos necessários. Estas classes são mostradas nas Listagens 4 a 10.

Listagem 4. Classe transformadora que determina que forma gráfica será usada para representar vértices na visualização.

```
// Esta classe transforma uma String em uma instância
// de Shape que será
// usada como ícone/símbolo para o vértice.
public class TransformaFormaDosVertices implements
Transformer<String, Shape>
{
    // Retorna uma elipse de tamanho (50,30) centrada
    // em (0,0).
    public Shape transform(String vértice) {
        return new Ellipse2D.Float(-25,-15,50,30);
    }
}
```

Listagem 5. Classe transformadora que determina que rótulo será usado para representar vértices na visualização.

```
// Esta classe transforma uma String no rótulo que será
// usado para o vértice.
public class TransformaRotulosDosVertices implements
Transformer<String, String>
{
    // Retorna o próprio nome do vértice.
    public String transform(String vértice) {
        return vértice;
    }
}
```

Listagem 6. Classe transformadora que determina que fonte será usada para desenhar os rótulos dos vértices na visualização.

```
// Esta classe transforma uma String em uma instância
// de Font que será
// usada para desenhar o rótulo do vértice.
public class TransformaFontesDosVertices implements
Transformer<String, Font>
{
    // Retorna Serif 18.
    public Font transform(String vértice) {
        return new Font("Serif",Font.PLAIN,18);
    }
}
```

Listagem 7. Classe transformadora que determina que traço será usado para desenhar os símbolos dos vértices na visualização.

```
// Esta classe transforma uma String em uma instância
// de Stroke que será
// usada para desenhar o símbolo/ícone para o vértice.
public class TransformaLinhasDosVertices implements
Transformer<String, Stroke>
{
```

```
    // Retorna uma linha mais grossa.
    public Stroke transform(String vértice) {
        return new BasicStroke(3f);
    }
}
```

Listagem 8. Classe transformadora que determina que preenchimento será usado para representar vértices na visualização.

```
// Esta classe transforma uma String em uma instância
// de Paint que será
// usada para preencher o desenho do vértice.
public class TransformaPreenchimentoDosVertices
implements Transformer<String, Paint>
{
    // Retorna verde claro.
    public Paint transform(String vértice) {
        return new Color(230,255,240);
    }
}
```

Listagem 9. Classe transformadora que determina que tipo de "tinta" (cor, padrão) será usada para desenhar os vértices na visualização.

```
// Esta classe transforma uma String em uma instância
// de Paint que será usada para desenhar a linha usada
// no desenho do vértice.
public class TransformaCorDasLinhasDosVertices
implements Transformer<String, Paint>
{
    // Retorna verde escuro.
    public Paint transform(String arg0) {
        return new Color(0,100,0);
    }
}
```

Listagem 10. Classe transformadora que determina que tipo de "tinta" (cor, padrão) será usada para desenhar as arestas na visualização do grafo.

```
// Esta classe transforma uma String em uma instância
// de Paint que será usada para desenhar a linha usada
// nas arestas.
public class TransformaCorDasLinhasDasArestas
implements Transformer<String, Paint>
{
    // Retorna azul.
    public Paint transform(String aresta) {
        return Color.BLUE;
    }
}
```

O resultado da execução da classe na Listagem 3 (e que usa instâncias das classes nas Listagens 4 a 10 para determinar a aparência visual do grafo) é mostrado na figura 3.

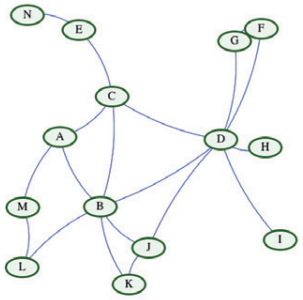


Figura 3. *Visualização criada pelo código na Listagem 3.*

Outro exemplo de modificação de aparência de vértices e arestas

Para consolidar os conceitos apresentados vejamos outro exemplo mais complexo: visualização de um grafo dirigido de uma malha viária simplificada (este exemplo é baseado no exemplo de malha viária como grafo mostrado no artigo "Introdução à Representação e Análise de Grafos com a API JUNG", publicado na Edição 49 da revista MundoJ). Neste exemplo, representaremos uma malha viária simples usando a classe Ponto como vértice e Trecho como aresta. A figura 4 mostra a malha viária que será representada.

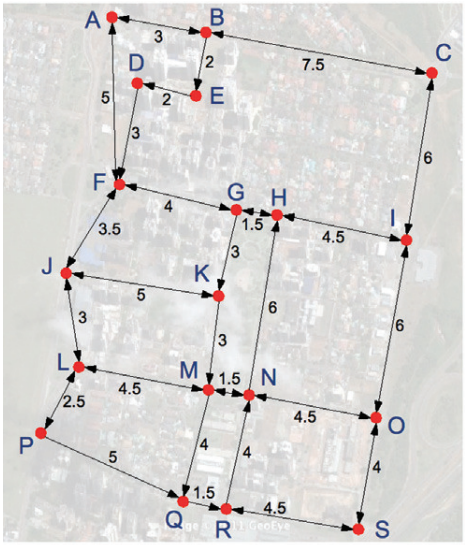


Figura 4. Malha viária simples que será representada como grafo e visualizada.

A classe Ponto representa uma coordenada em duas dimensões (instância de Point2D.Float) com um rótulo associado (instância de String). A classe contém um construtor para inicialização dos atributos, getters para os atributos e os métodos hashCode e equals que são necessários para que instâncias de Ponto sejam manipuladas corretamente pela API JUNG. Esta classe é um pouco diferente da classe

Ponto mostrada no artigo da edição 49, sua versão nova é mostrada na Listagem 11.

Listagem 11. Classe que representa um ponto no grafo de malha viária mostrado na figura 4.

```
// Esta classe representa um ponto no grafo mostrado na
// Figura 4.
public class Ponto {
   private String id;
   private Point2D.Float coordenadas;

   public Ponto(String i,Point2D.Float coords) {
      id = i;
      coordenadas = coords;
   }

   public String getId() {
      return id;
   }

   public Point2D.Float getCoordenadas() {
      return coordenadas;
   }

   public String toString() {
      return id;
   }

   // Os métodos hashCode e equals foram criados
   // automaticamente pela IDE Eclipse; outras IDEs tem
   // mecanismos semelhantes para a criação destes
   // métodos usando os atributos da classe para
   // determinar igualdade.
   public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + ((id == null) ? 0 :
         id.hashCode());
      return result;
   }

   public boolean equals(Object obj) {
      if (this == obj) return true;
      if (obj == null) return false;
      if (getClass() != obj.getClass()) return false;
      Ponto other = (Ponto) obj;
      if (id == null) {
         if (other.id != null) return false;
      } else if (!id.equals(other.id)) return false;
      return true;
   }

}
```

A classe Trecho representa um trecho entre dois pontos, contendo somente um identificador (String) para o trecho, um valor do tipo double para representar sua distância e um valor booleano que receberá true se houver alguma linha de ônibus que passa naquele trecho. Esta classe também tem um construtor para inicializar dados e os getters necessários, e não precisa implementar os métodos hashCode ou equals. A classe Trecho é omitida por ser exatamente a mesma mostrada na Listagem do artigo "Introdução à Representação e Análise de Grafos com a API JUNG", publicado na edição 49 da revista MundoJ.

Com as classes Ponto e Trecho podemos criar o grafo que representará a malha viária. A classe mostrada na Listagem 12 contém instâncias da classe Ponto que podem ser referenciadas em outras classes, e um método estático que cria o grafo e o retorna. Em aplicações reais o grafo poderia ser criado com dados obtidos de um arquivo ou banco de dados.

Listagem 12. Classe que cria um grafo que representa uma malha viária simples.

```
// Classe que cria um grafo usando a API JUNG.
public class CriaGrafoMV {
   // Para facilitar criamos logo os pontos da malha
   // viária. Cada ponto recebe um rótulo e um par de
   // coordenadas.
   public static Ponto pA = new Ponto("A",
      new Point2D.Float(162,23));
   public static Ponto pB = new Ponto("B",
      new Point2D.Float(302,46));
   public static Ponto pC = new Ponto("C",
      new Point2D.Float(638,107));
   public static Ponto pD = new Ponto("D",
      new Point2D.Float(200,122));
   public static Ponto pE = new Ponto("E",
      new Point2D.Float(286,141));
   public static Ponto pF = new Ponto("F",
      new Point2D.Float(173,272));
   public static Ponto pG = new Ponto("G",
      new Point2D.Float(347,311));
   public static Ponto pH = new Ponto("H",
      new Point2D.Float(407,318));
   public static Ponto pI = new Ponto("I",
      new Point2D.Float(600,355));
   public static Ponto pJ = new Ponto("J",
      new Point2D.Float(94,405));
   public static Ponto pK = new Ponto("K",
      new Point2D.Float(320,439));
   public static Ponto pL = new Ponto("L",
      new Point2D.Float(113,545));
   public static Ponto pM = new Ponto("M",
      new Point2D.Float(305,579));
   public static Ponto pN = new Ponto("N",
      new Point2D.Float(366,587));
   public static Ponto pO = new Ponto("O",
      new Point2D.Float(555,620));
   public static Ponto pP = new Ponto("P",
      new Point2D.Float(56,644));
   public static Ponto pQ = new Ponto("Q",
      new Point2D.Float(268,746));
   public static Ponto pR = new Ponto("R",
      new Point2D.Float(332,757));
   public static Ponto pS = new Ponto("S",
      new Point2D.Float(528,787));

   // Este método cria um grafo simples para uso em
   // alguns exemplos deste artigo.
   public static Graph<Ponto,Trecho> criaGrafo() {
```

```
        // Criamos um grafo onde vértices são instâncias de
        // Ponto e arestas são instâncias de Trecho. O
        // gráfico é dirigido, então usamos uma instância
// de DirectedSparseGraph.
        Graph<Ponto,Trecho> grafo =
          new DirectedSparseGraph<Ponto,Trecho>();
        // Criamos as arestas e vértices simultaneamente.
        grafo.addEdge(new Trecho("AB",3,false),pA,pB);
        grafo.addEdge(new Trecho("BA",3,false),pB,pA);
        grafo.addEdge(new Trecho("BC",7.5,false),pB,pC);
        grafo.addEdge(new Trecho("CB",7.5,false),pC,pB);
        grafo.addEdge(new Trecho("BE",2,false),pB,pE);
        grafo.addEdge(new Trecho("ED",2,false),pE,pD);
        grafo.addEdge(new Trecho("DF",3,false),pD,pF);
        grafo.addEdge(new Trecho("AF",5,true),pA,pF);
        grafo.addEdge(new Trecho("FA",5,true),pF,pA);
        grafo.addEdge(new Trecho("CI",6,false),pC,pI);
        grafo.addEdge(new Trecho("IC",6,false),pI,pC);
        grafo.addEdge(new Trecho("FG",4,false),pF,pG);
        grafo.addEdge(new Trecho("GF",4,false),pG,pF);
        grafo.addEdge(new Trecho("GH",1.5,false),pG,pH);
        grafo.addEdge(new Trecho("HG",1.5,false),pH,pG);
        grafo.addEdge(new Trecho("HI",4.5,false),pH,pI);
        grafo.addEdge(new Trecho("IH",4.5,false),pI,pH);
        grafo.addEdge(new Trecho("FJ",3.5,true),pF,pJ);
        grafo.addEdge(new Trecho("JF",3.5,true),pJ,pF);
        grafo.addEdge(new Trecho("GK",3,false),pG,pK);
        grafo.addEdge(new Trecho("JK",5,false),pJ,pK);
        grafo.addEdge(new Trecho("KJ",5,false),pK,pJ);
        grafo.addEdge(new Trecho("JL",3,true),pJ,pL);
        grafo.addEdge(new Trecho("LJ",3,true),pL,pJ);
        grafo.addEdge(new Trecho("KM",3,false),pK,pM);
        grafo.addEdge(new Trecho("NH",6,false),pN,pH);
        grafo.addEdge(new Trecho("IO",6,false),pI,pO);
        grafo.addEdge(new Trecho("OI",6,false),pO,pI);
        grafo.addEdge(new Trecho("LM",4.5,false),pL,pM);
        grafo.addEdge(new Trecho("ML",4.5,true),pM,pL);
        grafo.addEdge(new Trecho("MN",1.5,false),pM,pN);
        grafo.addEdge(new Trecho("NM",1.5,true),pN,pM);
        grafo.addEdge(new Trecho("NO",4.5,false),pN,pO);
        grafo.addEdge(new Trecho("ON",4.5,false),pO,pN);
        grafo.addEdge(new Trecho("LP",2.5,true),pL,pP);
        grafo.addEdge(new Trecho("PL",2.5,false),pP,pL);
        grafo.addEdge(new Trecho("PQ",5,true),pP,pQ);
        grafo.addEdge(new Trecho("MQ",4,false),pM,pQ);
        grafo.addEdge(new Trecho("QR",1.5,true),pQ,pR);
        grafo.addEdge(new Trecho("RN",4,true),pR,pN);
        grafo.addEdge(new Trecho("RS",4.5,true),pR,pS);
        grafo.addEdge(new Trecho("SR",4.5,true),pS,pR);
        grafo.addEdge(new Trecho("OS",4,false),pO,pS);
        grafo.addEdge(new Trecho("SO",4,false),pS,pO);
        return grafo;
    }
}
```

Para a visualização do grafo criado na classe mostrada na Listagem 12 usaremos as seguintes condições:

- O layout do grafo será baseado nas coordenadas dos pontos. Para isto, usaremos a classe StaticLayout e inicializaremos as coordenadas com uma classe transformadora auxiliar.
- Queremos visualizar o grafo com vértices e arestas representadas de forma diferenciada dependendo dos dados representados. Para isto iremos recuperar o contexto de visualização e usar seus métodos setXXXTransformer para indicar classes transformadoras para desenhar vértices e arestas.
- Para tornar a visualização mais interessante, usaremos o algoritmo de Dijkstra, que calcula o caminho mais curto entre dois pontos em um grafo, para calcular este caminho entre dois pontos. O algoritmo é implementado na API JUNG. Vértices e arestas que pertencem a este ponto serão mostrados de forma diferenciada.
- Usaremos um componente de visualização que permite pan, zoom e outras modificações na aparência do grafo. Este componente é implementado na API JUNG pela classe GraphZoomScrollPane.

A visualização do grafo é realizada pela classe VisMV, mostrada na Listagem 13.

Listagem 13. Classe VisMV, que cria uma GUI para visualização interativa do grafo da malha viária.

```
// Classe para criar uma visualização mais complexa de
um grafo.
public class VisMV {
   public VisMV() {
      // O grafo é criado em outra classe, para organizar melhor o
      // código.
      Graph<Ponto,Trecho> grafo = CriaGrafoMV.criaGrafo();
      // Passo 1: criamos uma instância de StaticLayout para
      // controlar o layout. Inicializamos as posições com um
      // transformador de instâncias de Pontos para
coordenadas.
      StaticLayout<Ponto,Trecho> layout =
        new StaticLayout<Ponto,Trecho>(grafo);
      layout.setInitializer(new PosicionaPontos());
      // Passo 2: criamos o componente para visualização.
      VisualizationViewer<Ponto,Trecho> componente =
          new VisualizationViewer<Ponto,Trecho>(layout);
      // Passo 3: Modificamos o processo de renderização através
      // de classes transformadoras. Precisamos do contexto de
      // renderização.
      RenderContext<Ponto,Trecho> ctx =
        componente.getRenderContext();
      // Para algumas transformações vamos precisar de
      // informações auxiliares: o caminho mais curto entre dois
      // pontos ('E' e 'R').
      DijkstraShortestPath<Ponto,Trecho> sp =
          new DijkstraShortestPath<Ponto, Trecho>(grafo,
          new DistanciaDoTrecho());
      List<Trecho> caminho =
        sp.getPath(CriaGrafoMV.pE,CriaGrafoMV.pR);
     // Queremos que os rótulos dos vértices e arestas apareçam
      // na visualização do grafo.
      ctx.setVertexLabelTransformer(new VMV_
VertexLabelT());
      ctx.setEdgeLabelTransformer(new VMV_
EdgeLabelT());
```

```
        // Vértices serão colorizados de acordo com o número de
        // conexões.
        ctx.setVertexFillPaintTransformer(
            new VMV_VertexFillT(grafo,caminho));
        // A forma, cor e o traço do ícone dos vértices será diferente
        // para vértices que fazem parte do caminho mais curto.
        ctx.setVertexShapeTransformer(
            new VMV_VertexShapeT(grafo,caminho));
        ctx.setVertexStrokeTransformer(
            new VMV_VertexStrokeT(grafo,caminho));
        ctx.setVertexDrawPaintTransformer(
            new VMV_VertexDrawT(grafo,caminho));
        // A cor e traço das linhas das arestas será diferente para
        // vértices que fazem parte do caminho mais curto.
        ctx.setEdgeStrokeTransformer(
            new VMV_EdgeStrokeT(grafo,caminho));
        ctx.setEdgeDrawPaintTransformer(
            new VMV_EdgeDrawT(grafo,caminho));
        // Precisamos mudar também a cor das setas das arestas,
        // podemos usar a mesma classe que determina a cor das
        // arestas.
        ctx.setArrowDrawPaintTransformer(
            new VMV_EdgeDrawT(grafo,caminho));
        ctx.setArrowFillPaintTransformer(
            new VMV_EdgeDrawT(grafo,caminho));
        // Modificamos também a posição do rótulo dos vértices.
        VertexLabel<Ponto,Trecho> vl =
                componente.getRenderer().
getVertexLabelRenderer();
        vl.setPosition(Renderer.VertexLabel.Position.CNTR);
        // Modificamos algumas características do componente.
        componente.setPreferredSize(new
Dimension(689,820));
        componente.setBackground(Color.WHITE);
        // Melhoramos a aparência do desenho deste componente
        // modificando seus RenderingHints.
        HashMap<Key, Object> renderingHints =
            new HashMap<Key,Object>();
        renderingHints.put(
            RenderingHints.KEY_ALPHA_INTERPOLATION,
                RenderingHints.VALUE_
                ALPHA_INTERPOLATION_QUALITY);
        renderingHints.put(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        componente.setRenderingHints(renderingHints);
        // Queremos que este componente esteja em um
        // scrollpane especial, com gestos básicos do mouse
        // associados.
        componente.setGraphMouse(
            new DefaultModalGraphMouse<Ponto,Trecho>());
        GraphZoomScrollPane pane =
            new GraphZoomScrollPane(componente);
        //Passo 5:usamos o componente para montar a GUI
        JFrame f = new JFrame("Visualização de Grafos –
            Malha Viária");
        f.getContentPane().add(pane);
        f.pack();
        f.setVisible(true);
        f.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
    }
```

```
    // Executa a classe como uma aplicação.
    public static void main(String[] args) {
        new VisMV(); // Monta a GUI.
    }
}
```

As classes auxiliares necessárias para o funcionamento da aplicação na classe VisMV (Listagem 13) são apresentadas a seguir, na ordem em que aparecem referenciadas no código. A primeira classe auxiliar é a classe PosicionaPontos, que será usada para inicializar as coordenadas dos vértices do grafo (já que o layout implementado pela classe StaticLayout não calcula posições para os vértices). Esta classe é mostrada na Listagem 14.

Listagem 14. Classe PosicionaPontos, que mapeia instâncias de Ponto para instâncias de Point2D.

```
// Esta classe transforma uma instância de Ponto em
// uma instância de Point2D,
// que será usada como coordenada em um layout
// estático. A transformação é  trivial pois instâncias de
// Ponto já contém as coordenadas em seus atributos.
public class PosicionaPontos implements
Transformer<Ponto,Point2D>
{
    public Point2D transform(Ponto p) {
        return p.getCoordenadas();
    }
}
```

Precisamos calcular o menor caminho entre dois pontos ('E' e 'R') do grafo, e usar este caminho para desenhar de forma diferenciada os vértices e arestas que pertencem a ele. O cálculo do menor caminho é feito pelo algoritmo de Dijkstra, implementado pela classe DijkstraShortestPath, que usa o grafo e uma instância de uma classe transformadora para obter as distâncias a partir de instâncias das classes que representam arestas. Esta classe é mostrada na Listagem 15.

Listagem 15. Classe DistanciaDoTrecho, que mapeia instâncias de Trecho para instâncias de Double.

```
// Esta classe transforma uma instância de Trecho em
// uma instância de Double, que será usada por
// algoritmos que calculam distâncias e caminhos. A
// transformação é trivial pois instâncias de Trecho já
// contêm os valores necessários.
public class DistanciaDoTrecho implements
Transformer<Trecho,Double>
{
    public Double transform(Trecho t) {
        return t.getDistância();
    }
}
```

Para a visualização deste grafo queremos usar rótulos associados aos vértices e arestas. Para os vértices, usamos o atributo id da classe Ponto; e para as arestas usaremos o valor da distância associada às instâncias de Trecho. A transformação de instâncias de Ponto e Trecho em Strings usadas como rótulos é feita pelas classes transformadoras mostradas nas Listagens 16 e 17, respectivamente.

Listagem 16. Classe VMV_VertexLabelT, que mapeia instâncias de Ponto para String.

```
// Esta classe transforma uma instância de Ponto no
// rótulo que será usado para visualização do vértice.
public class VMV_VertexLabelT implements
Transformer<Ponto,String>
{
   // Retorna o identificador do ponto.
   public String transform(Ponto p) {
      return p.getId();
   }
}
```

Listagem 17. Classe VMV_EdgeLabelT, que mapeia instâncias de Trecho para String.

```
// Esta classe transforma uma instância de Trecho no
// rótulo que será usado para visualização da aresta.
public class VMV_EdgeLabelT implements
Transformer<Trecho,String>
{
   // Retorna a distância encapsulada na classe Trecho.
   public String transform(Trecho t) {
      return ""+t.getDistância();
   }
}
```

As classes transformadoras usadas na classe VisDetalhada (Listagem 3) só precisam ter acesso a referências das instâncias das classes que representam os vértices e arestas para mapear os atributos destas instâncias em atributos gráficos para visualização das mesmas. No exemplo da classe VisMV precisaremos de mais informações para este mapeamento, especificamente, precisaremos do grafo em si e do caminho calculado pelo algoritmo de Dijkstra – a implementação do algoritmo retorna uma lista de instâncias de Trecho, e será necessário converter esta lista em um conjunto de Arestas para algumas transformações. O desenvolvimento das classes transformadoras para este exemplo será simplificado se criarmos uma classe que faz a conversão e que pode ser usada como ancestral para as outras classes transformadoras neste exemplo e que também armazena referências ao próprio grafo e caminho mínimo. Esta classe-base é mostrada na Listagem 18.

Listagem 18. Classe VMV_TransBase, que pré-processa o caminho mínimo entre dois pontos para uso posterior.

```
// Esta classe preprocessa a lista de trechos que compõe
// um caminho mínimo, transformando-a em um
// conjunto de vértices por onde o caminho passa.
public class VMV_TransBase {
   protected Graph<Ponto,Trecho> grafo;
   protected List<Trecho> caminho;
   protected HashSet<Ponto> noCaminho;
   public VMV_TransBase(Graph<Ponto,Trecho>
      g,List<Trecho> c) {
      grafo = g;
      caminho = c;
      noCaminho = new HashSet<Ponto>();
      for(Trecho t:caminho) {
         Pair<Ponto> pontos = grafo.getEndpoints(t);
         noCaminho.addAll(pontos);
      }
   }
}
```

Com a classe-base mostrada na Listagem 18 podemos implementar as classes que transformam instâncias de vértices e arestas (usando também o grafo em si e o caminho mais curto) em atributos gráficos, usando para isto os atributos dos vértices, arestas, do grafo em si e do caminho mínimo calculado na classe VisMV (Listagem 13). Estas classes herdam de VMV_TransBase e são mostradas nas Listagens 19 a 24.

A classe VMV_VertexFillT, mostrada na Listagem 19, determina uma cor para os vértices dependendo do grau (número de arestas) que conecta este vértice a outros no grafo.

Listagem 19. Classe VMV_VertexFillT, que determina a cor a ser usada para desenhar os vértices.

```
// Esta classe transforma uma instância de Ponto em
// uma instância de Paint que será usada para preencher
// o desenho do vértice. A cor de preenchimento será
// determinada pelo número de trechos que contém
// aquele ponto.
public class VMV_VertexFillT extends VMV_TransBase
implements Transformer<Ponto,Paint>
{
   public VMV_VertexFillT(Graph<Ponto,Trecho> grafo,
      List<Trecho> caminho) {
      super(grafo,caminho);
   }
   // Determina a cor de preenchimento pelo número de
   // vértices que incluem esta aresta.
   public Paint transform(Ponto p) {
      int cont = grafo.degree(p);
      Color cor;
      switch(cont) {
         case 0: cor = Color.BLACK; break;
         case 1: cor = Color.DARK_GRAY; break;
```

```
            case 2: cor = Color.BLUE; break;
            case 3: cor = Color.GREEN; break;
            case 4: cor = Color.YELLOW; break;
            case 5: cor = Color.RED; break;
            default: cor = Color.MAGENTA; break;
        }
        return cor;
    }
}
```

A classe VMV_VertexShapeT, mostrada na Listagem 20, determina uma forma geométrica para os vértices: quadrados para pontos que pertencem ao caminho mínimo e círculos para os outros.

Listagem 20. Classe VMV_VertexShapeT, que determina a forma geométrica a ser usada para desenhar os vértices.

```
// Esta classe transforma uma instância de Ponto em
// uma instância de Shape que será usada para desenhar
// o vértice. O tipo de desenho (forma geométrica)
// será diferente para pontos dentro e fora do caminho
// mínimo.
public class VMV_VertexShapeT extends VMV_TransBase
        implements Transformer<Ponto,Shape>
{
    public VMV_VertexShapeT(Graph<Ponto, Trecho>
        grafo, List<Trecho> caminho) {
        super(grafo,caminho);
    }
    // Retorna quadrados para pontos que pertencem ao
    // caminho mínimo e círculos para pontos fora do
    // caminho mínimo.
    public Shape transform(Ponto p) {
        if (noCaminho.contains(p))
            return new Rectangle2D.Float(-10,-10,20,20);
        else
            return new Ellipse2D.Float(-10,-10,20,20);
    }
}
```

A classe VMV_VertexStrokeT, mostrada na Listagem 21, determina diferentes tipos de traço para o desenho dos vértices.

Listagem 21. Classe VMV_VertexStrokeT, que determina o traço a ser usado para desenhar os vértices.

```
// Esta classe transforma uma instância de Ponto em
// uma instância de Stroke que será usada para desenhar
// o vértice. Pontos dentro do caminho mínimo serão
// desenhados com linha mais grossa.
public class VMV_VertexStrokeT extends VMV_TransBase
        implements Transformer<Ponto,Stroke>
{
    public VMV_VertexStrokeT(Graph<Ponto,Trecho>
        grafo, List<Trecho> caminho) {
        super(grafo,caminho);
    }
    // Linhas mais grossas para pontos dentro do caminho
    // mínimo.
    public Stroke transform(Ponto p) {
        if (noCaminho.contains(p)) return
            new BasicStroke(3f);
        else return new BasicStroke(1f);
    }
}
```

A classe VMV_VertexDrawT, mostrada na Listagem 22, determina diferentes cores para o traço usado para desenhar os vértices.

Listagem 22. Classe VMV_VertexDrawT, que determina a cor do traço a ser usado para desenhar os vértices.

```
// Esta classe transforma uma instância de Ponto em
// uma instância de Paint que será usada para desenhar
// o vértice. A cor da borda do desenho será diferente
// para pontos dentro e fora do caminho mínimo.
public class VMV_VertexDrawT extends VMV_TransBase
implements Transformer<Ponto,Paint>
{
    public VMV_VertexDrawT(Graph<Ponto,Trecho> grafo,
        List<Trecho> caminho) {
        super(grafo,caminho);
    }
    // Azul claro para pontos no caminho, preto para os
    // outros.
    public Paint transform(Ponto p) {
        if (noCaminho.contains(p)) return
            new Color(150,150,255);
        else return Color.BLACK;
    }
}
```

A classe VMV_EdgeStrokeT, mostrada na Listagem 23, determina a aparência (espessura e tracejados) para o traço usado para desenhar as arestas.

Listagem 23. Classe VMV_EdgeStrokeT, que determina a aparência do traço usado para desenhar os vértices.

```
// Esta classe transforma uma instância de Trecho em uma
// instância de Stroke que será usada para desenhar a aresta.
Dois
// fatores são considerados: trechos no caminho mínimo terão
// linhas grossas, e trechos por onde passa ônibus serão
tracejados.
public class VMV_EdgeStrokeT extends VMV_TransBase
        implements Transformer<Trecho,Stroke>
{
    public VMV_EdgeStrokeT(Graph<Ponto,Trecho> grafo,
        List<Trecho> caminho) {
        super(grafo,caminho);
    }
```

```
    public Stroke transform(Trecho t) {
        float largura = 1f;
        if (caminho.contains(t)) largura = 3f;
        if (t.isPassaÔnibus())
            return new BasicStroke(largura,
              BasicStroke.CAP_BUTT,BasicStroke.JOIN_BEVEL,
               0,new float[]{5},0);
        else
            return new BasicStroke(largura);
    }
}
```

A classe VMV_EdgeDrawT, mostrada na Listagem 24, determina a cor que será usada para desenhar as arestas. A mesma classe será usada para determinar a cor das arestas, e a cor e preenchimento das setas usadas nas arestas (veja a classe VisMV na Listagem 13).

Listagem 24. Classe VMV_EdgeDrawT, que determina a aparência do traço usado para desenhar os vértices.

```
// Esta classe transforma uma instância de Trecho em
// uma instância de Paint que será usada para desenhar
// a aresta. Arestas no caminho mínimo serão
// desenhadas em azul, as outras em preto.
public class VMV_EdgeDrawT extends VMV_TransBase
implements Transformer<Trecho,Paint>
{
    public VMV_EdgeDrawT(Graph<Ponto,Trecho> grafo,
        List<Trecho> caminho) {
        super(grafo,caminho);
    }
    public Paint transform(Trecho t) {
        if (caminho.contains(t)) return Color.BLUE;
        else return Color.GRAY;
    }
}
```

As figuras 5, 6 e 7 mostram a interface gráfica criada pela classe VisMV (Listagem 13). A figura 5 mostra a aparência inicial, a figura 6 mostra o grafo reduzido para poder ser visualizado por inteiro e a figura 7 mostra um trecho do grafo ampliado – é possível observar que os desenhos têm características vetoriais: a ampliação não degrada a qualidade das linhas e elementos gráficos.

A classe GraphZoomScrollPane, parte da API JUNG, facilita bastante a visualização de grandes grafos: ela funciona como uma versão de JScrollPane, e contém métodos para processar a interação do mouse com o componente.

Para usar esta classe é preciso primeiro definir o comportamento do mouse para o componente de visualização (neste exemplo, instância de VisualizationViewer) através da chamada ao método setGraphMouse, passando como argumento a classe que determina o comportamento, neste caso, DefaultModalGraphMouse (com parâmetros de tipo iguais ao do grafo). Depois basta criar uma instância de GraphZoomScrollPane recebendo a instância de VisualizationViewer como parâmetro para seu construtor. O componente automaticamente processará os seguintes gestos do mouse:

- Clicar e arrastar o mouse: modifica a área de visualização do grafo no componente (pan).
- Usar o mouse wheel: amplia ou reduz a área visualizada (zoom).
- Clicar e arrastar o mouse pressionando shift: gira o grafo no componente.
- Clicar e arrastar o mouse pressionando control ou command: deforma o grafo no componente na vertical ou horizontal, dependendo da direção de arrasto (skew).
-

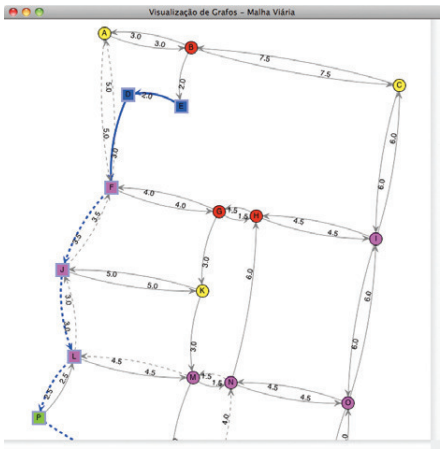


Figura 5. *Interface gráfica criada pela classe VisVM (Listagem 12).*

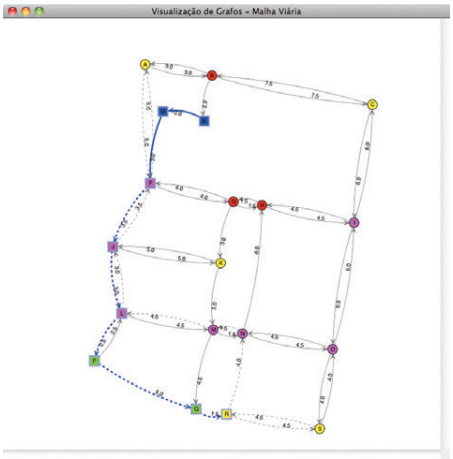


Figura 6. *Interface gráfica criada pela classe VisVM (mostrando todo o grafo).*

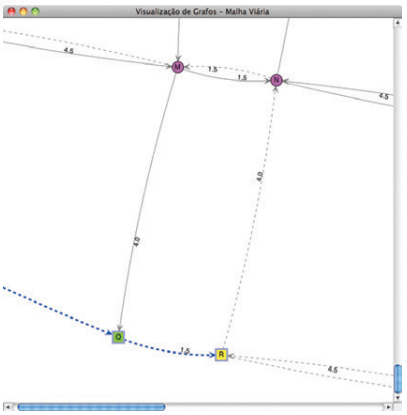


Figura 7. *Interface gráfica criada pela classe VisVM (mostrando região ampliada).*

A instância da classe DefaultModalGraphMouse que é usada para controlar o comportamento do mouse para o componente de visualização também permite operações de seleção de vértices: basta criar a instância desta classe e executar o método setMode(ModalGraphMouse.Mode.PICKING) antes de associar o comportamento com a classe de visualização. Vértices podem ser selecionados com o mouse e arrastados para novas posições. O trecho de código apresentado na Listagem 25 mostra como isto pode ser feito.

Listagem 25. Trecho de código que demonstra como configurar o componente de visualização para permitir seleção de vértices no grafo.

/para saber mais

> O artigo "Introdução à Representação e Análise de Grafos com a API JUNG", publicado na Edição 49 da revista MundoJ, mostra como representar e processar grafos de diversos tipos, inclusive como executar algoritmos de análise sobre os grafos.

> O site da API JUNG contém vários exemplos e acesso à documentação da própria API gerada pela ferramenta javadoc, e pode ser acessado em http://jung.sourceforge.net/doc/index.html. Um tutorial curto, mas simples da API JUNG pode ser encontrado em http://www.grotto-networking.com/JUNG/JUNG2-Tutorial.pdf (ambos em inglês).

> O documento http://jung.sourceforge.net/doc/JUNGVisualizationGuide.html (também em inglês) explica os conceitos usados no design das classes da API para visualização, e foi usado como base para parte deste artigo.

> Conceitos básicos de grafos podem ser encontrados em diversas páginas na Wikipédia, em especial em http://pt.wikipedia.org/wiki/Teoria_dos_grafos e http://pt.wikipedia.org/wiki/Grafo. Alguns conceitos e referências sobre visualização de grafos, particularmente sobre os layouts, pode ser vista em http://en.wikipedia.org/wiki/Graph_drawing (em inglês).

```
VisualizationViewer<Ponto,Trecho> componente =
      new VisualizationViewer<Ponto,Trecho>(layout);
DefaultModalGraphMouse<Ponto,Trecho> graphMouse =
      new DefaultModalGraphMouse<Ponto,Trecho>();
graphMouse.setMode(ModalGraphMouse.Mode.PICKING);
componente.setGraphMouse(graphMouse);
```

Considerações finais

A API JUNG contém classes para facilitar a visualização de grafos de diversos tipos, com várias formas de alterar a aparência visual dos elementos do grafo. A API contém também algumas classes que implementam algoritmos de layout dos vértices e arestas de um grafo, mas também permite a determinação manual do layout.

A criação de exemplos de visualização requer a criação de várias classes que determinam a aparência de vários elementos gráficos do grafo, o que é complexo e trabalhoso. Se o objetivo é criar uma ilustração simples de um grafo pode ser mais simples usar de um editor gráfico; por outro lado a abordagem programática permite a automação da tarefa de desenho com grande flexibilidade, como demonstrado – no exemplo do grafo de malha viária, para visualizar o caminho mais curto entre outros pontos basta mudar uma linha de código.