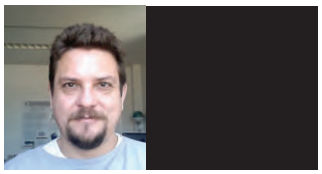
INTRODUÇÃO À REPRESENTAÇÃO E ANÁLISE DE GRAFOS COM A API JUNG

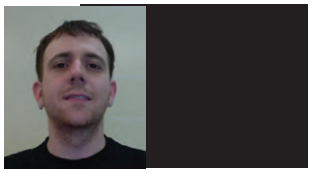
Técnicas e exemplos de código para representar e analisar estruturas como redes sociais, redes de computadores, conjuntos de dados relacionais e muitos outros que podem ser representados como grafos.

Grafos são estruturas que servem para representar muitos artefatos 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 (como, por exemplo, diretórios em um computador) e muitos outros exemplos podem ser representados como grafos. Neste artigo veremos alguns conceitos básicos de grafos e como representá-los e executar análises simples em Java usando a API JUNG (Java Universal Network/Graph Framework).



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.

Grafos são estruturas de dados que representam objetos e as relações entre eles. Embora esta definição seja simples, podemos encontrar no dia-a-dia inúmeros exemplos de objetos-relações que podem ser representadas como grafos: redes sociais, redes de computadores, malhas rodoviárias ou aeroviárias, estruturas de páginas na Web e de forma mais geral a própria Web. Árvores (como organogramas e estruturas hierárquicas de arquivos e diretórios) são casos específicos de grafos.

Grafos são compostos de **vértices** (que correspondem aos objetos que queremos representar) e **arestas**, que ligam pares de vértices. Com estes elementos relativamente simples podemos representar conjuntos de dados e relações complexos, e efetuar alguns tipos de análises que resultam em várias medidas sobre o grafo, seus elementos e subconjuntos dos mesmos. Grafos podem ser categorizados de acordo com suas características, alguns exemplos serão mostrados a seguir.

A figura 1 mostra um exemplo relativamente simples de grafo, simulando relações entre alguns parti-

cipantes de uma rede social. Para as finalidades deste artigo consideraremos redes como um conjunto de relações entre objetos, que considera como exemplos não somente as amplas redes sociais humanas já conhecidas como Facebook e Orkut, como também as inferidas (como relações entre funcionários trabalhando em projetos, por exemplo) e até redes que contêm elementos não-humanos, como dependências de classes em um projeto de desenvolvimento de software de grande porte.

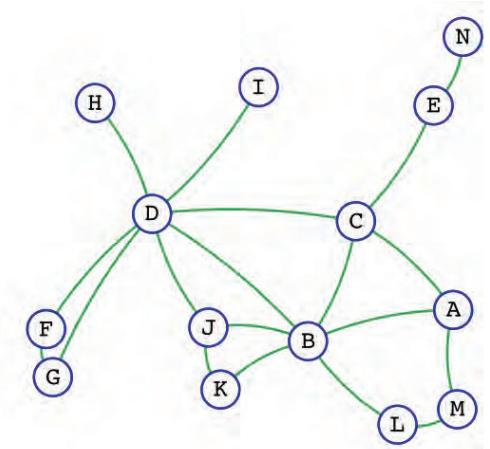


Figura 1. *Grafo que representa uma rede social simples, com poucos membros.*

No grafo mostrado na figura 1 temos 14 vértices, representados pelas letras entre A e M, e 19 arestas, que unem alguns vértices a outros. Neste grafo os vértices correspondem a pessoas em uma rede social, e as arestas indicam que existe alguma relação entre estas pessoas. Este grafo é considerado não-dirigido ou não-direcionado, pois não existe orientação nas arestas: elas somente denotam a ligação entre um vértice e outro. O grafo também não é completo: grafos completos são aqueles nos quais todos os vértices são ligados uns aos outros. O grafo também não tem pesos associados às arestas, o que pode ser útil quando queremos associar valores ou custos para as ligações entre vértices (como, por exemplo, distâncias entre cidades em um grafo que representa uma malha rodoviária).

Outro exemplo de grafo é mostrado na figura 2, que representa, de forma bastante simplificada, um conjunto de roteadores responsáveis por conectar o tráfego entre um usuário e um site na Web (este exemplo foi inspirado no exemplo em *http://computer.howstuffworks.com/internet/basics/internet-infrastructure.htm*). Neste grafo os oito vértices são roteadores (o usuário e site podem ser considerados vértices, dependendo da aplicação que será feita deste grafo) e são representados pelas letras A até H. As 12 arestas correspondem às ligações entre os roteadores. No exemplo da figura 2 temos um grafo não-dirigido, que não é completo, mas que tem pesos associados às arestas, correspondentes, no exemplo, ao tempo médio de conexão entre os roteadores.

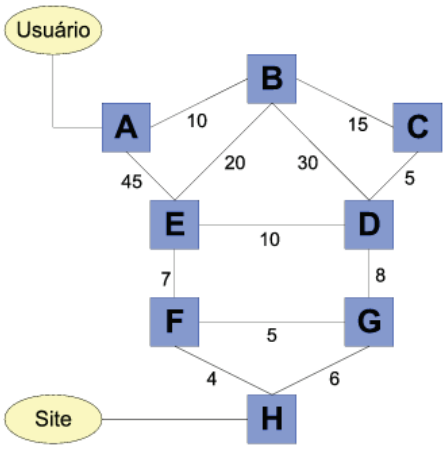


Figura 2. *Rede de computadores representada como grafo com pesos.*

Ainda outro exemplo de grafo é mostrado na figura 3. Este grafo corresponde a um trecho de uma malha de ruas locais em um bairro, e pode ser usado para estudos sobre condições de transporte neste bairro. O grafo contém 19 vértices que são pontos relevantes na malha viária, e que são representados pelas letras de A até S. As arestas têm pesos associados que correspondem às distâncias aproximadas entre os vértices. As arestas são direcionadas de um ponto para outro: na maioria dos casos levam de um ponto a outro e vice-versa, mas em alguns (como a aresta que liga B a E) só apresentam uma direção. Arestas bidirecionais podem ser representadas como um par de arestas que liga os vértices, portanto no exemplo teremos uma aresta que liga A a B e uma que liga B a A (o peso associado poderia até ser diferente): temos então um total de 44 arestas. O grafo na figura 3 é direcionado (é um digrafo), não é completo e tem pesos associados às arestas.

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 agosto 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.

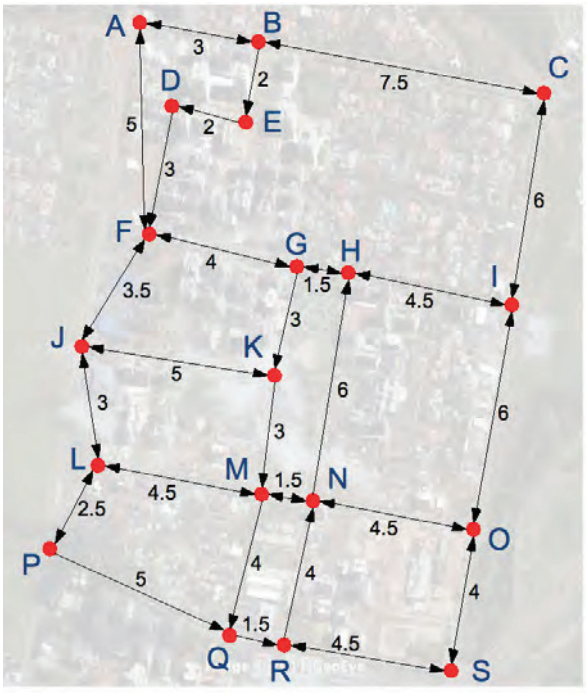


Figura 3. *Exemplo de rede viária representada como grafo.*

O restante deste artigo descreve como a API JUNG pode ser usada para representar grafos e que operações podem ser realizadas com os mesmos, buscando mostrar exemplos reais.

Representando Grafos na API JUNG

Grafos podem ser representados como coleções de vértices e arestas, e podem ser facilmente criados usando as coleções de Java, sem precisar usar outras APIs. Uma desvantagem desta abordagem simplista é que algoritmos que operam sobre grafos devem também ser implementados pelo programador.

A API JUNG (Java Universal Network/Graph Framework) possibilita a representação de grafos de diversos tipos e para diversas aplicações. A API também contém vários algoritmos para cálculo de atributos das arestas e vértices e de subconjuntos de um grafo (como, por exemplo, algoritmos de análise estatística, cálculo de distâncias, centralidade etc.), assim como algoritmos para visualização dos grafos e para leitura e armazenamento de grafos em arquivos.

Grafos em JUNG podem ter várias arestas conectando os vértices. Arestas podem ser direcionais ou não, e podem ou não ter peso associado a elas. Grafos, no entanto, não podem ter vértices repetidos nem arestas repetidas – se for necessário criar várias arestas entre dois vértices, cada uma deverá ser um objeto diferente.

Um grafo em JUNG é representado como uma instância de classes que implementam a interface Graph. A instância deve ser declarada como tendo dois parâmetros de tipo: o primeiro define que classe será usada para representar os vértices, e o segundo tipo define a classe que será usada para representar as arestas. Esta característica da API possibilita a criação de grafos onde vértices e arestas são representados por instâncias de praticamente qualquer classe, o que garante enorme flexibilidade de representação e aplicações. De forma análoga às coleções de Java, a API provê classes que implementam diversos tipos de grafos (todas implementando a interface Graph), cada uma com características específicas que as fazem mais adequadas para determinadas aplicações.

Depois da criação de uma instância de classe que implementa Graph podemos adicionar vértices e arestas à instância, o que pode ser feito através dos métodos addVertex e addEdge, respectivamente. O método addVertex aceita como parâmetro uma instância da classe escolhida para representar vértices, enquanto o método addEdge recebe como argumentos uma instância da classe que representa arestas e duas instâncias da classe que representa vértices. Alternativamente podemos usar somente o método addEdge, que cuida de adicionar ao grafo as arestas e os vértices conectados por elas.

Para exemplificar a criação de um grafo com a API JUNG, vamos criar o grafo mostrado na figura 1. Para simplificar vamos considerar que tanto as arestas quanto os vértices são instâncias da classe String (lembrando que não podemos ter identificadores repetidos, nem para as arestas nem para os vértices). Usaremos para representar o grafo a classe UndirectedSparseGraph, que é adequada para representar grafos esparsos (com número de arestas relativamente pequeno) não-dirigidos.

A Listagem 1 mostra como este grafo pode ser criado (o código neste artigo não segue algumas regras e boas práticas de orientação a objetos, isto foi feito para mantê-lo simples e legível). O grafo é criado no método criaGrafo, e algumas estatísticas básicas sobre este grafo são calculadas e mostradas no método main, que tam-

bém armazena o grafo em um arquivo. Usamos somente o método addEdge para adicionar ao grafo, simultaneamente, vértices e arestas (a inclusão de arestas repetidas será ignorada pela classe que representa o grafo, mas a inclusão de um vértice com o mesmo identificador causará uma exceção).

Listagem 1. Classe CriaGrafo1, que cria uma representação do grafo mostrado na figura 1.

```
import edu.uci.ics.jung.graph.Graph;
import edu.uci.ics.jung.graph.UndirectedSparseGraph;

// Classe que cria um grafo usando a API JUNG.
public class CriaGrafo1 {
// 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;
   }

   public static void main(String[] args) {
      // Criamos o grafo através do método auxiliar.
      Graph<String,String> grafo = criaGrafo();
      // Algumas métricas do nosso grafo:
      System.out.println("Número de vértices: "
            +grafo.getVertexCount());
      System.out.println("Número de arestas: "
            +grafo.getEdgeCount());
      System.out.println("Vértices ligados a A: "
            +grafo.getNeighbors("A"));
      System.out.println("A tem "
            +grafo.getNeighborCount("A")+" vizinhos.");
   }
}
```

O método main da classe mostrada na Listagem 1 cria o grafo e mostra algumas estatísticas básicas sobre o mesmo: número de arestas e vértices, e vértices que estão ligados ao vértice A.

Como segundo exemplo vamos criar o grafo mostrado na figura 2. A diferença principal deste grafo em comparação com o mostrado na figura 1 é o peso dado às arestas. Faremos a implementação de uma forma diferente: para facilitar alguns exemplos mostrados neste artigo, tanto as arestas quanto os vértices do grafo serão representadas por classes específicas (Roteador e Conexao, respectivamente). A Listagem 2 mostra a classe CriaGrafo2, que cria instâncias de Roteador e Conexão e as usa para montar o grafo visto na figura 2.

Listagem 2. Classe CriaGrafo2, que cria uma representação.

```
import edu.uci.ics.jung.graph.Graph;
import edu.uci.ics.jung.graph.UndirectedSparseGraph;
```

// Classe que cria um grafo usando a API JUNG.

```
public class CriaGrafo2 {
   // Para facilitar criamos logo os roteadores.
   public static Roteador rA =
      new Roteador("A","Zisco");
   public static Roteador rB =
      new Roteador("B","Z-Link");
   public static Roteador rC =
      new Roteador("C","Zisco");
   public static Roteador rD =
      new Roteador("D","Z-Link");
   public static Roteador rE =
     new Roteador("E","Zetgear");
   public static Roteador rF =
     new Roteador("F","Zisco");
   public static Roteador rG =
     new Roteador("G","Zisco");
   public static Roteador rH =
     new Roteador("H","Zetgear");

   // Este método cria um grafo simples para uso
   // em alguns exemplos deste artigo.
   public static Graph<Roteador,Conexao>
   criaGrafo() {
      // Criamos um grafo onde vértices são
      //instâncias de Roteador e arestas são
      // instâncias de Conexao.
      Graph<Roteador,Conexao> grafo =
         new UndirectedSparseGraph
         <Roteador,Conexao>();
      grafo.addEdge(new Conexao("A-B",10),rA,rB);
      grafo.addEdge(new Conexao("A-E",45),rA,rE);
      grafo.addEdge(new Conexao("B-C",15),rB,rC);
      grafo.addEdge(new Conexao("B-D",30),rB,rD);
      grafo.addEdge(new Conexao("B-E",20),rB,rE);
```

```
        grafo.addEdge(new Conexao("C-D", 5),rC,rD);
        grafo.addEdge(new Conexao("D-E",10),rD,rE);
        grafo.addEdge(new Conexao("D-G", 8),rD,rG);
        grafo.addEdge(new Conexao("E-F", 7),rE,rF);
        grafo.addEdge(new Conexao("F-G", 5),rF,rG);
        grafo.addEdge(new Conexao("F-H", 4),rF,rH);
        grafo.addEdge(new Conexao("G-H", 6),rG,rH);
        return grafo;
    }

    public static void main(String[] args) {
        // Criamos o grafo através do método auxiliar.
        Graph<Roteador,Conexao> grafo = criaGrafo();
        // Algumas métricas do nosso grafo:
        System.out.println("Número de vértices: "
            +grafo.getVertexCount());
        System.out.println("Número de arestas: "
            +grafo.getEdgeCount());
        System.out.println("Conexões entre A e B:"
            +grafo.findEdgeSet(rA,rB));
        System.out.println("Conexões entre B e A:"
            +grafo.findEdgeSet(rB,rA));
        System.out.println("Conexões entre A e H:"
            +grafo.findEdgeSet(rA,rH));
    }
}
```

O grafo criado na classe mostrada na Listagem 2 tem oito vértices e 12 arestas. Como o grafo foi declarado como não-dirigido (instância de UndirectedSparseGraph) cada aresta que conecta dois vértices faz a conexão nos dois sentidos: por exemplo, a conexão entre os vértices A e B é a mesma entre os vértices B e A, como mostrado na execução dos métodos findEdgeSet, que retorna a lista de arestas entre dois vértices. A terceira execução do método findEdgeSet retorna um conjunto vazio indicando que não existem arestas entre os vértices A e H.

As Listagens 3 e 4 mostram as classes Roteador e Conexão, respectivamente. As classes simplesmente encapsulam os atributos necessários para representar roteadores e conexões neste exemplo, sem conter métodos especiais, mas com os métodos get para alguns atributos e toString para retornar uma representação textual da instância. Instâncias da classe Roteador serão usadas como vértices no grafo, portanto devem implementar métodos que facilitem a sua comparação (para evitar que duas instâncias com valores de atributos iguais sejam contidas pelo grafo, entre outras coisas) – para isto basta implementar os métodos hashCode e equals. A implementação destes métodos foi feita automaticamente pela IDE Eclipse, considerando como critério de igualdade somente o atributo id.

Listagem 3. Classe Roteador, usada como vértice para a criação do grafo na Listagem 2.

```
// Classe que representa um roteador como vértice em
um grafo.
// Cada roteador terá uma identificação que deve ser
única e um nome de
// fabricante.

public class Roteador {
    private String id;
    private String fabricante;

    public Roteador(String i,String f) {
        id = i;
        fabricante = f;
    }

    public String toString() {
        return id;
    }

    public String getId() {
        return id;
    }

    public String getFabricante() {
        return fabricante;
    }

    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;
        Roteador other = (Roteador) obj;
        if (id == null) {
            if (other.id != null) return false;
        } else if (!id.equals(other.id)) return false;
        return true;
    }

}
```

Listagem 4. Classe Conexão, usada como aresta para a criação do grafo na Listagem 2.

```
// Classe que representa uma conexão entre roteadores.
Cada conexão terá uma
// identificação única e uma velocidade.

public class Conexao {
    private String id;
    private double velocidade;
```

```
    public Conexao(String i,double v) {
        id = i; velocidade = v;
    }

    public String getId() {
        return id;
    }

    public double getVelocidade() {
        return velocidade;
    }

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

Como terceiro exemplo de representação de grafos com a API JUNG vamos representar o grafo mostrado na figura 3. Este grafo é diferente dos exemplos mostrados anteriormente, pois contém arestas direcionadas, que em alguns casos são unidirecionais: ligam um vértice a outro, mas não o segundo ao primeiro (correspondendo a trechos de mão única em uma malha viária).

A representação em código do grafo da figura 3 também usará classes específicas para seus vértices e arestas. Para representar os vértices, usaremos instâncias da classe Ponto, que representará um ponto marcado no grafo da figura 3. Para representar as arestas usaremos instâncias da classe Trecho, que correspondem a um trecho entre pontos na figura 3 e que tem, além de um identificador, dois valores associados às instâncias: a distância do trecho (em uma unidade arbitrária) e um valor booleano que indica se no trecho existem pontos de ônibus. Embora nem todos estes atributos sejam usados nos exemplos deste artigo, vale a pena criar estas classes de forma mais elaborada do que o estritamente necessário para mostrar que praticamente qualquer classe pode ser usada para representar vetores e/ou arestas em grafos com a API JUNG.

A Listagem 5 mostra a classe CriaGrafo3, que cria de forma programática uma representação do grafo mostrado na figura 3. Como este grafo contém arestas direcionadas, usaremos uma instância da classe DirectedSparseGraph para representar o grafo.

Listagem 5. Classe CriaGrafo3, que cria uma representação do grafo mostrado na figura 3.

```
import edu.uci.ics.jung.graph.DirectedSparseGraph;
import edu.uci.ics.jung.graph.Graph;

// Classe que cria um grafo usando a API JUNG.
public class CriaGrafo3 {
// Para facilitar criamos logo os pontos da rede viária.
    public static Ponto pA = new Ponto("A");
public static Ponto pB = new Ponto("B");
    public static Ponto pC = new Ponto("C");
    public static Ponto pD = new Ponto("D");
public static Ponto pE = new Ponto("E");
public static Ponto pF = new Ponto("F");
public static Ponto pG = new Ponto("G");
public static Ponto pH = new Ponto("H");
public static Ponto pI = new Ponto("I");
public static Ponto pJ = new Ponto("J");
public static Ponto pK = new Ponto("K");
public static Ponto pL = new Ponto("L");
public static Ponto pM = new Ponto("M");
public static Ponto pN = new Ponto("N");
public static Ponto pO = new Ponto("O");
public static Ponto pP = new Ponto("P");
public static Ponto pQ = new Ponto("Q");
public static Ponto pR = new Ponto("R");
public static Ponto pS = new Ponto("S");

// 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;
    }

    public static void main(String[] args) {
        // Criamos o grafo através do método auxiliar.
        Graph<Ponto,Trecho> grafo = criaGrafo();
        // Algumas métricas do nosso grafo:
        System.out.println("Número de vértices: "
            +grafo.getVertexCount());
        System.out.println("Número de arestas: "
            +grafo.getEdgeCount());
        System.out.println("Vizinhos ao ponto N: "
            +grafo.getNeighbors(pN));
        System.out.println("Se conectam ao N: "
            +grafo.getPredecessors(pN));
        System.out.println("N se conecta a: "
            +grafo.getSuccessors(pN));
    }

}
```

O método main na classe da Listagem 5 demonstra mais alguns métodos que podem ser aplicados a grafos: getNeighbors retorna um Set com os vértices que se conectam (independentemente da direção) ao vértice passado como argumento. O método getPredecessors retorna o conjunto de vértices que se conectam ou levam para o passado como argumento, e o método getSuccessors faz o mesmo para os vértices que recebem conexão do passado como argumento. No exemplo mostrado na Listagem 5 os conjuntos resultantes da execução destes três métodos [H, R, O, M], [R, O, M] e [H, O, M].

As Listagens 6 e 7 mostram, respectivamente, as classes Ponto e Trecho, que representam os vértices e arestas no grafo criado na Listagem 5. A classe Ponto encapsula somente um identificador, e contém métodos para comparar instâncias da classe, garantindo que não teremos identificadores repetidos no conjunto (Set) de vértices do grafo (veja também a Listagem 3). A classe Trecho encapsula um identificador para o trecho, um valor do tipo Double para representar a distância do trecho e um booleano que indica se existem pontos de ônibus no trecho, além de métodos para recuperar estes valores.

Listagem 6. Classe Ponto, que representa um ponto no grafo criado na Listagem 5.

```
// Esta classe representa um ponto no grafo mostrado na Figura 3.
public class Ponto {
    private String id;

    public Ponto(String i) {
        id = i;
    }

    public String getId() {
        return id;
    }

    public String toString() {
        return id;
    }

    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;
    }

}
```

Listagem 7. Classe Trecho, que representa um trecho entre pontos no grafo criado na Listagem 5.

```
// Esta classe representa um trecho entre dois pontos no grafo mostrado na
// Figura 3.
public class Trecho {
    private String id;
    private double distância;
    private boolean passaÔnibus;
        public Trecho(String i,double d,boolean p) {
        id = i; distância = d; passaÔnibus = p;
    }
        public Trecho(String i,double d,boolean p) {
        id = i; distância = d; passaÔnibus = p;
    }
```

```
    public String getId() {
        return id;
    }

    public double getDistância() {
        return distância;
    }

    public boolean isPassaÔnibus() {
        return passaÔnibus;
    }

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

}
```

Alguns algoritmos em grafos

Nos exemplos de código mostrados até agora somente vimos como representar grafos de diversos tipos e como obter informações básicas destes grafos (número de vértices e arestas, informações sobre conexão). A API JUNG provê classes e métodos que permitem fazer análises mais complexas nos grafos, inclusive implementando muitos algoritmos tradicionais de teoria dos grafos.

Alguns algoritmos interessantes em grafos tratam da centralidade de um vértice em um grafo, que pode ser definida como uma medida da importância de um vértice naquele grafo (por exemplo, como uma medida de popularidade em redes sociais ou de densidade de tráfego em redes viárias). Existem várias medidas de centralidade, baseadas em diferentes critérios, que estão implementadas na API JUNG e que podem ser calculadas para cada vértice. Algumas delas são:

- Centralidade por grau (implementada pela classe DegreeScorer) retorna o número de arestas que liga aquele vértice a outros no mesmo grafo.
- Centralidade por participação em caminhos ("betweenness centrality" em inglês): vértices que participam de muitos caminhos entre todos os outros vértices de um grafo tem um grau maior de centralidade. O algoritmo para calcular esta métrica é implementado na classe BetweennessCentrality. A métrica não tem unidade associada.
- Centralidade por proximidade (implementada pela classe ClosenessCentrality) é calculada como a média dos caminhos mais curtos que unem um vértice a todos os outros do grafo. A métrica não tem unidade associada.

Instâncias das classes que implementam algoritmos para calcular centralidade (e que implementam a interface VertexScorer) usam instâncias da classe que representa o grafo como argumento para seus construtores. As instâncias de classes que implementam algoritmos devem ser declaradas com parâmetros de tipo compatíveis com os usados para os vértices (e às vezes arestas) do grafo que irá ser usado para o cálculo. Estas instâncias podem então ser usadas para obter o valor da centralidade de cada vértice executando o método getVertexScore. Isto é demonstrado na classe da Listagem 8, que calcula três métricas de centralidade para os vértices do grafo mostrado na figura 1.

Listagem 8. Classe CentralidadeGrafo1, que calcula centralidade para os vértices do grafo na figura 1.

```
import edu.uci.ics.jung.algorithms.scoring.Between-
nessCentrality;
import edu.uci.ics.jung.algorithms.scoring.Closeness-
Centrality;
import edu.uci.ics.jung.algorithms.scoring.Degree-
Scorer;
import edu.uci.ics.jung.graph.Graph;

public class CentralidadeGrafo1 {
    public static void main(String[] args) {
   // Criamos o grafo (usando um método de outra clas-
se).
        Graph<String, String> grafo =
          CriaGrafo1.criaGrafo();
  // Criamos três instâncias de classes que calculam
  // centralidade.
```

Armazenando grafos com a API JUNG

A API JUNG permite o armazenamento de instâncias de classes que representam grafos em três formatos: texto ASCII, GraphML (padrão baseado em XML) e Pajek NET (software para processamento de redes e grafos). Embora o armazenamento de grafos em arquivos seja relativamente simples, a recuperação de grafos de arquivos para a representação em memória é significativamente mais complexa, requerendo a codificação de fábricas de instâncias que representam vértices e arestas; e codificação de classes especiais para armazenar os metadados dos grafos. A documentação da API e tutoriais encontrados na WWW não mostram exemplos claros e simples; neste momento a solução mais simples para armazenar grafos criados com a API é através da serialização simples.

```
        DegreeScorer<String> ds =
            new DegreeScorer<String>(grafo);
        BetweennessCentrality<String,String> bs =
                new BetweennessCentrality<String,String>
                    (grafo);
        ClosenessCentrality<String,String> cs =
                new ClosenessCentrality<String,String>
                    (grafo);
        // Mostramos as três métricas de centralidade
para cada
        // vértice no grafo.
        for(String s:grafo.getVertices()) {
            System.out.printf("%s  %2d  %7.3f  %7.3f\n",s,
                    ds.getVertexScore(s),
                    bs.getVertexScore(s),cs.getVertexScore(s));
        }
    }
}
```

A classe mostrada na Listagem 8 imprime como resultado as três métricas de centralidade do grafo mostrado na figura 1. Para o vértice I, que está isolado no grafo, as métricas são 1, zero e 0.394; para o vértice D, que é bem central ao grafo, as métricas são 7, 42.5 e 0.619; e para o vértice C, que é medianamente central, as métricas são 4, 26.167 e 0.591.

Outro tipo de algoritmo importante que pode ser aplicado a grafos envolve a criação de um conjunto de subgrafos a partir do grafo original usando um critério específico. Um destes algoritmos faz a partição do grafo em subgrafos eliminando as arestas que têm maior participação em caminhos entre todos os grafos (veja a definição de centralidade por participação em caminhos). Este algoritmo pode ser usado para identificar comunidades em redes sociais ou regiões altamente conectadas em outros tipos de redes, e é implementado pela classe EdgeBetweennessClusterer. Para usar este algoritmo, criamos primeiro uma instância da classe, declarando como parâmetros de tipo as classes usadas para representar os vértices e arestas do grafo e passando para seu construtor o número de vértices que deve ser removido, e para recuperar os subgrafos formados usamos o método transform que recebe o grafo como argumento e retorna um conjunto de conjuntos da classe usada para representar vértices. Outro método interessante desta classe é o getEdgesRemoved, que retorna uma lista das arestas removidas. O uso desta classe é demonstrado na Listagem 9.

Listagem 9. Classe ClusterGrafo1, que separa o grafo da figura 1 em subgrafos.

```
import java.util.Set;

import edu.uci.ics.jung.algorithms.cluster.EdgeBe-
tweennessClusterer;
import edu.uci.ics.jung.graph.Graph;

public class ClusterGrafo1 {
    public static void main(String[] args) {
        // Criamos o grafo (usando um método de outra
        // classe).
        Graph<String, String> grafo =
            CriaGrafo1.criaGrafo();
        EdgeBetweennessClusterer<String,String> ec =
                new EdgeBetweennessClusterer<String,
                    String>(4);
        Set<Set<String>> clusters = ec.transform(grafo);
        // Mostramos os subgrafos resultantes.
        for(Set<String> conjunto:clusters) {
            System.out.println(conjunto);
        }
        // Mostramos as arestas que foram removidas.
        for(String vérticesRemovidos:ec.
            getEdgesRemoved()) {
                System.out.println(vérticesRemovidos);
        }
    }
}
```

O resultado da execução da classe mostrada na Listagem 9 mostra que o grafo da figura 1 foi separado em três subgrafos: [D, F, G, H, I], [A, B, C, L, M, J, K] e [E, N], e que os quatro vértices removidos foram [C-E, B-D, C-D, D-J]. O resultado pode ser apreciado comparando-o com o grafo mostrado na figura 1.

Outros algoritmos importantes que podem ser aplicados a grafos envolvem o cálculo de caminhos entre os vértices, considerando as características das arestas que conectam os vértices. Um destes algoritmos é o Algoritmo de Dijkstra, que retorna o caminho mais curto ou de menor custo entre dois vértices de um grafo. A API JUNG implementa este algoritmo através da classe DijkstraShortestPath. Para calcular o caminho mais curto entre dois vértices de um grafo, devemos criar uma instância de DijkstraShortestPath, novamente com parâmetros de tipo iguais às classes usadas para representar vértices e arestas no grafo e passando o grafo como argumento para o construtor.

Para que o algoritmo seja executado corretamente, ele precisa ter uma forma de recuperar o peso associado às arestas do grafo. Para isto precisamos de uma classe auxiliar cujo objetivo é transformar instâncias da classe usada como aresta para valores numéricos. Esta classe deve implementar a interface Transformer, e deve ter dois parâmetros de tipo: um para a classe que implementa a aresta e uma que encapsula um valor numérico, frequentemente usaremos Double.

A Listagem 10 mostra um exemplo de uso da classe DijkstraShortestPath, que calcula o caminho mais curto e seu comprimento total (considerando os pesos associados às arestas) entre vértices do grafo mostrado na figura 2.

Listagem 10. Classe CaminhosGrafo2, que mostra como calcular o caminho mais curto entre os roteadores da figura 2.

```
import edu.uci.ics.jung.algorithms.shortestpath.Dijks-
traShortestPath;
import edu.uci.ics.jung.graph.Graph;

// Classe que calcula o caminho mais curto entre dois
// roteadores.
public class CaminhosGrafo2 {
   public static void main(String[] args) {
      // Criamos o grafo através do método auxiliar.
      Graph<Roteador, Conexao> grafo =
         CriaGrafo2.criaGrafo();
      // Criamos a implementação do algoritmo de
      // Dijkstra.
      DijkstraShortestPath<Roteador,Conexao> sp =
            new DijkstraShortestPath
               <Roteador,Conexao>(grafo,
                  new VelocidadeDaConexao());
      // Qual é o caminho mais curto entre os
      // roteadores A e H?
      System.out.println(
         sp.getPath(CriaGrafo2.rA,CriaGrafo2.rH));
      // Qual é a distância entre os roteadores A e H?
      System.out.println(
         sp.getDistance(CriaGrafo2.rA,CriaGrafo2.rH));
   }
}
```

A classe mostrada na Listagem 10 usa o grafo criado no método criaGrafo da Listagem 2, e calcula o caminho mais curto (método getPath, que retorna uma lista de instâncias de Conexão) e distância mínima (método getDistance) entre dois roteadores do grafo ilustrado pela figura 2. No exemplo mostrado, o caminho mais curto é A-B, B-E, E-F e F-H e a distância correspondente a este caminho é 41.

Conforme mencionado, para que o algoritmo calcule corretamente o caminho mais curto é preciso informar à instância da classe DijkstraShortestPath uma classe que converte instâncias que representam arestas para valores numéricos. Esta classe deve implementar a interface Transformer (da API Commons Collections, que está inclusa na API JUNG) e deve conter o método transform, que recebe uma instância de Conexão e retorna um valor do tipo Double (a velocidade associada a esta conexão). Esta classe é mostrada na Listagem 11.

Listagem 11. Classe VelocidadeDaConexao, que recupera o valor da velocidade de uma instância de Conexão.

```
import org.apache.commons.collections15.Trans-
former;

// Esta classe possibilita a recuperação da
// velocidade de uma instância de
// Conexao.
public class VelocidadeDaConexao implements
   Transformer<Conexao,Double>
{
   public Double transform(Conexao c) {
      return c.getVelocidade();
   }
}
```

No exemplo mostrado na Listagem 10, o caminho mais curto entre os roteadores H e A é o reverso exato do caminho entre os roteadores A e H, pois o grafo foi criado como não direcionado, e neste caso as arestas são consideradas bidirecionais. A implementação do algoritmo de Dijkstra considera também grafos com arestas direcionais no cálculo do caminho mais curto. Para exemplificar, vamos usar o grafo criado na Listagem 5, e que corresponde ao grafo mostrado na figura 3, e calcular o caminho mais curto entre alguns de seus pontos. Este exemplo é mostrado na Listagem 12, que segue basicamente o mesmo roteiro da Listagem 10.

Listagem 12. Classe CaminhosGrafo3, que mostra como calcular o caminho mais curto entre os pontos da figura 3.

```
import edu.uci.ics.jung.algorithms.shortestpath.Dijks-
traShortestPath;
import edu.uci.ics.jung.graph.Graph;

// Classe que calcula o caminho mais curto entre dois
// pontos de  uma malha viária.
public class CaminhosGrafo3 {
   public static void main(String[] args) {
      // Criamos o grafo através do método auxiliar.
      Graph<Ponto,Trecho> grafo =
         CriaGrafo3.criaGrafo();
      // Calculamos a distância mais próxima entre
      // alguns pontos do grafo.
      DijkstraShortestPath<Ponto,Trecho> sp =
            new DijkstraShortestPath<Ponto, Trecho>(
            grafo,new DistanciaDoTrecho());
      // Mostramos alguns caminhos mais curtos e
      // suas distâncias.
      System.out.println(
         sp.getPath(CriaGrafo3.pA,CriaGrafo3.pS));
      System.out.println(
         sp.getDistance(CriaGrafo3.pA,CriaGrafo3.pS));
      System.out.println(
         sp.getPath(CriaGrafo3.pS,CriaGrafo3.pA));
      System.out.println(
         sp.getDistance(CriaGrafo3.pS,CriaGrafo3.pA));
      System.out.println(
         sp.getPath(CriaGrafo3.pM,CriaGrafo3.pK));
      System.out.println(
         sp.getDistance(CriaGrafo3.pM,CriaGrafo3.pK));
      System.out.println(
```

```
            sp.getPath(CriaGrafo3.pH,CriaGrafo3.pN));
        System.out.println(
            sp.getDistance(CriaGrafo3.pH,CriaGrafo3.pN));
    }
}
```

Para criar a instância da classe DijkstraShortestPath na classe mostrada na Listagem 12 precisamos de uma classe que recupere os valores das distâncias encapsuladas na classe Trecho. Esta classe é mostrada na Listagem 13.

Listagem 13. Classe DistanciaDoTrecho, que recupera o valor da distância encapsulado em uma instância de Trecho.

```
import org.apache.commons.collections15.Trans-
former;

public class DistanciaDoTrecho implements
Transformer<Trecho, Double>
{
    public Double transform(Trecho arg) {
        return arg.getDistância();
    }
}
```

O resultado da execução da classe CaminhosGrafo3, mostrada na Listagem 12, mostra os seguintes caminhos mais curtos e distâncias entre pontos:

Entre os pontos A e S: [AF(5.0), FJ(3.5), JL(3.0), LP(2.5), PQ(5.0), QR(1.5), RS(4.5)], distância total: 25.0

Entre os pontos S e A: [SO(4.0), ON(4.5), NH(6.0), HG(1.5), GF(4.0), FA(5.0)], distância total: 25.0

Entre os pontos M e N: [MN(1.5), NH(6.0), HG(1.5), GK(3.0)], distância total: 12.0

Entre os pontos H e N: [HG(1.5), GK(3.0), KM(3.0), MN(1.5)], distância total: 9.0

Podemos ver pelos resultados que o caminho mais curto entre A e S é diferente do entre S e A, mesmo que a distância seja coincidentemente a mesma.

Considerações finais

Grafos são estruturas de dados flexíveis e que podem ser usadas para representar diversos objetos e relações do mundo real. Sua representação e análise são muito mais simples quando usamos a API JUNG. Este artigo demonstrou algumas classes e métodos desta API e a implementação de alguns algoritmos clássicos de grafos na mesma.

Além dos algoritmos demonstrados, a API contém também outros algoritmos para agrupamento de grafos, extração de subgrafos a partir de um grafo (usando filtros ou critérios de vizinhança entre vértices, por exemplo), geração de grafos, análises estatísticas, resolução do problema do fluxo máximo (algoritmo de Edmonds-Karp), e várias métricas de importância de vértices e arestas em um grafo, como HITS e PageRank. A API também apresenta vários algoritmos que facilitam a representação visual dos grafos.

/para saber mais

> 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 www.grotto-networking.com/JUNG/JUNG2-Tutorial.pdf.

> Dois bons livros de estruturas de dados em Java que cobrem aspectos teóricos e práticos de grafos são Data Structures and Algorithms in Java de Michael T. Goodrich e Roberto Tamassia; e Object-Oriented Data Structures Using Java de Nell Dale, Daniel T. Joyce e Chip Weems.

> 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. A descrição de alguns algoritmos de centralidade pode ser vista (em inglês) em http://en.wikipedia.org/wiki/Centrality, e a descrição em português (com simulação) do algoritmo de Dijkstra pode ser vista em http://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra.