Aula 05 – Notas de Aula – JGraphT

Notas de Aula de Teoria dos Grafos

Profª: Patrícia D. L. Machado

UFCG – Unidade Acadêmica de Sistemas e Computação

Sumário

Introdução

Neste laboratório, exploramos recursos da **JGraphT** e também de classes do pacote **util** de **GraphTheory-JGraphT** para analisar propriedades de um grafo, iterar sobre vértices e arestas de um grafo e conhecer classes que dão suporte a geração de grafos.

Acessando Vértices e Arestas sem Atributos

Em um programa, é comum precisarmos acessar o objeto vértice ou aresta de um grafo através de seu identificador. Quando estamos utilizando a classe de arestas **DefaultEdge**, basta invocar o método *getEdge*, da Interface ***Graph***, com os identificadores dos vértices terminais da aresta que queremos recuperar. Caso a aresta exista, um objeto é retornado.

```
DefaultEdge e = graph.getEdge(0, 1);
```

Para uma certa aresta, os vértices terminais são retornados através dos métodos *getEdgeSource* e *getEdgeTarget*. Lembrando que para um grafo não-direcionado, não faz diferença se um vértice é *source* ou *target*.

```
Integer v1 = graph.getEdgeSource(e);

Integer v2 = graph.getEdgeTarget(e);
```

A classe Aula04GetVertexEdgeWithOutAttributes do pacote **classexamples** apresenta um exemplo simples de uso destes métodos em um grafo cujos vértices são instâncias da classe **Integer** e as arestas são da classe **DefaultEdge**. Observe que o grau de um vértice é retornado pelo método *degreeOf*, passando o valor inteiro correspondente ao vértice.

Acessando Vértices e Arestas com Atributos através do *Label*

Para acessar objetos vértice e aresta das classes **DefaultVertex** e **RelationshipEdge** através do *label*, utilizamos métodos auxiliares da classe **VertexEdgeUtil** do pacote **util**.

Para vértices, utilizamos o método *getVertexfromLabel*. Este método recebe como parâmetros o conjunto de vértices do grafo e o *label* do vértice que queremos retornar.

```
DefaultVertex a = VertexEdgeUtil.getVertexfromLabel(graph.vertexSet(),"a");
```

Para arestas, utilizamos o método *getEdgefromLabel* que recebe como parâmetro a lista de arestas do grafo e o *label* da aresta que queremos retornar.

```
RelationshipEdge e1 = VertexEdgeUtil.getEdgefromLabel(graph.edgeSet(), "x");
```

Para obtermos os vértices terminais de um objeto aresta, podemos utilizar os métodos *getV1* e *getV2* da classe **RelationshipEdge**, ou o método *getNeighbor* caso possua a referência de um dos terminais. Podemos também utilizar os métodos *getEdgeSource* e *getEdgeTarget* da interface ***Graph***.

```
DefaultVertex v1 = e.getV1();

DefaultVertex v2 = e.getV2();

DefaultVertex v3 = e.getNeighbor(v);

DefaultVertex v4 = graph.getEdgeSource(e);
```

A classe Aula04GetVertexEdgeWithAttributes do pacote **classexamples** apresenta um exemplo de uso destes métodos.

Testando Propriedades em um Grafo

A classe **GraphTests** apresenta uma coleção de métodos utilitários para computar várias propriedades sobre um grafo. Os métodos desta classe são estáticos. Dentre eles, podemos destacar o método *hasMultipleEdges* que determina se um grafo passado como parâmetro possui arestas paralelas. O método *hasSelfLoops* determina se um grafo possui arestas *loop*. O método *isConnected* determina se um grafo é conectado. O método *isBipartite* determina se um grafo é bipartido. O método *isComplete* determina se um grafo é completo.

```
GraphTests.hasMultipleEdges(graph);
GraphTests.hasSelfLoops(graph);
GraphTests.isConnected(graph);
GraphTests.isBipartite(graph);
GraphTests.isComplete(graph);
```

A classe Aula04GraphTests do pacote **classexamples** exemplifica o uso destes métodos.

Iterando sobre o Grafo

A maneira mais simples de iterar sobre os vértices e arestas de um grafo é utilizando Java *Iterators*. A JGraphT disponibiliza tipos especializados de *Iterators* que utilizam estratégias elaboradas de busca as quais veremos mais adiante. Considere o exemplo abaixo, onde queremos imprimir no console os graus de cada vértice de um grafo. Neste exemplo, utilizamos o *Iterator* padrão da classe **Set** de Java. Definimos um Iterator *it* para um **Set** de **DefaultVertex** e inicializamos ele com o *Iterator* do conjunto de vértices do grafo. Depois criamos uma repetição com o comando *while* que executará enquanto houver um vértice ainda não explorado no conjunto, ou seja, enquanto o método *hasNext*() retornar *true*. No corpo do *while*, obtemos um vértice do *Iterator* invocando o método *next*() e guardamos sua referência na variável v. Isto é necessário se vamos utilizar o vértice mais de uma vez porque a cada invocação de *next*, um novo vértice é retornado. Depois imprimimos o grau do vértice através do método *degreeOf* .

```
Iterator<DefaultVertex> it = graphgml.vertexSet().iterator();

while (it.hasNext()) {
	DefaultVertex v = it.next();
	System.out.println("d_G(" + v.toString() + ") = " + graph.degreeOf(v));
}
```

Alternativamente, podemos iterar sobre os vértices do grafo usando expressões *lambda* do **Java 8**. Para aqueles que possuem conhecimento avançado em Java, esta é uma alternativa que permite deixar o código mais simples e elegante.

```
graph.vertexSet().stream().forEach(v -> {
	System.out.println("d_G(" + v.toString() + ") = " + graph.degreeOf(v));
});
```

A classe Aula04GraphIteration do pacote **classexamples** apresenta um exemplo com as duas formas de iterar sobre um grafo.

Nesta classe exemplo, consideramos também a impressão no console da lista de vizinhos de um vértice. Para tal usamos a classe **NeighborCache** que mantém um cache dos vizinhos de cada vértice do grafo. Embora as listas de vizinhos possam ser obtidas a partir do grafo, elas são recalculadas em cada chamada, percorrendo as arestas incidentes de um vértice, o que se torna

excessivamente caro quando executado com frequência, especialmente em grafos grandes. O cache também controla a lista dos sucessores e predecessores de cada vértice, o que é útil para grafos direcionados.

Geradores de Grafos

Por fim, vamos conhecer um pouco sobre o pacote **generate** da JGraphT que possui geradores automáticos de grafos. A interface *GraphGenerator* é implementada por todos os geradores deste pacote. A seguir, veremos um exemplo de utilização das classes **CompleteGraphGenerator** e **CompleteBipartiteGraphGenerator.**

Vamos criar um grafo completo com uma quantidade de vértices fornecida como entrada. Para tal, criamos uma instância de um grafo simples passando como parâmetro um *Supplier* para vértices e outro para arestas, tal como fizemos para importar grafos. Qualquer *Supplier* da classe **SupplierUtil** pode ser utilizado. Aqui estamos usando um especial, fornecido pela classe **VertexEdgeUti**l, onde os vértices serão nomeados v0, v1, ...

```
Graph<String, DefaultEdge> completeGraph = new SimpleGraph<>
      (VertexEdgeUtil.createStringVVertexSupplier(),
       SupplierUtil.createDefaultEdgeSupplier(), false);
```

Agora, criamos uma instância da classe **CompleteGraphGenerator** com os mesmos tipos para vértices e arestas definidos para a instancia de grafo que criamos e passamos como parâmetro para este gerador a quantidade de vértices que o grafo deverá conter. O método *generateGraph* é invocado para o objeto gerador a fim de que o grafo seja criado.

```
CompleteGraphGenerator<String, DefaultEdge> completeGenerator =
                  new CompleteGraphGenerator<>(n);
completeGenerator.generateGraph(completeGraph);
```

Para criar um grafo bipartido completo usando a classe **CompleteBipartiteGraph**, a estratégia é semelhante. A diferença é que na criação do gerador, precisamos passar como parâmetro os tamanhos das partições.

```
CompleteBipartiteGraphGenerator<String, DefaultEdge> completeGenerator =
                  new CompleteBipartiteGraphGenerator<>(p1,p2);
```

As classes Aula04CompleteGraph e Aula04BipartiteCompleteGraph do pacote **classexamples** ilustram o uso destes geradores.

O pacote shortestpath e a interface GraphPath

O pacote **org.jgrapht.alg.shortestpath** da **JGraphT** possui classes que implementa diferentes algoritmos para calcular um caminho (*path*) entre dois vértices ou de um vértice para todos os outros. Estudaremos

alguns destes algoritmos em aulas posteriores. Nesta aula, veremos apenas a interface **GraphPath** e uma classe básica para calcular o menor caminhos em grafos não-direcionados.

Todos os algoritmos de **shortestpath** retornam paths como instâncias de classe que implementam a interface **GraphPath**. Esta interface apresenta métodos que nos permitem obter:

- uma lista com as arestas do path, *getEdgeList()*;
- uma lista com os vértices de um path, *getVertexList()*;
- o peso associado a um path, *getWeight()*; dentre outros.

A classe **GraphWalk** implementa a interface **GraphPath**. Raramente será necessário construir uma instância desta classe diretamente, visto que os métodos da JGraphT sempre fazem referência a **GraphPath**. No entanto, caso necessite representar um *path* vazio a ser retornado por seus métodos, por exemplo, quando nenhuma opção de *path* for encontrada, crie o seguinte objeto:

```
GraphPath <DefaultVertex, RelationshipWeightedEdge> emptyPath;
emptypath = new GraphWalk <> (graph, new ArrayList <DefaultVertex> (), 0.0);
```

Classe YenKShortestPath e YenShortestPathIterator

A classe **YenKShortestPath** implementa o algoritmo de *Yen* que encontra os *K* menores caminhos sem loop entre 2 vértices, tomando como base os pesos das arestas, no caso do grafo ser ponderado (estudaremos grafos ponderados em aulas posteriores). Caminhos sem loop são aqueles sem vértices repetidos, ou seja, são **caminhos simples**. Para utilizar esta classe, basta criar uma instância passando como parâmetro um grafo.

```
Graph<DefaultVertex, DefaultEdge> graph =
                    new SimpleGraph <>(VertexEdgeUtil.createDefaultVertexSupplier(),
                        SupplierUtil.createDefaultEdgeSupplier(), false);

YenKShortestPath <DefaultVertex, DefaultEdge> yenk =
                                                new YenKShortestPath <> (graph);
```

No exemplo acima, criamos um grafo simples utilizando a classe **SimpleGraph** da JGraphT e a classe **DefaultEdge** para representar as arestas.

A classe **YenShortestPathIterator** implementa um *iterator* sobre os menores caminhos sem loop entre dois vértices de um grafo, seguindo o algoritmo de Yen. O *iterator* retorna os caminhos ordenados do menor para o maior. Para utilizar esta classe, basta criar uma instância passando como parâmetros o grafo e os vértices de origem e destino. Em seguida, como qualquer outro *iterator*, usa-se os métodos *hasNext*() e *next*() para testar se ainda existem caminhos e para obter o próximo caminho respectivamente.

```
YenShortestPathIterator <DefaultVertex, DefaultEdge> yenI =
                    new YenShortestPathIterator <> (graph,source,sink);
```

Dependendo do grafo, a quantidade de caminhos pode ser muito grande. Então é recomendável limitar as iterações a um certo número ***n*** de caminhos, como no exemplo abaixo.

```
int count = 0;
int limit = 10;
System.out.println(limit + " shortest paths:");
while ((yenI.hasNext()) && (count < limit)) {
        GraphPath <DefaultVertex, DefaultEdge> yenIpath = yenI.next();
        System.out.println(yenIpath.getVertexList());
        count++;
}
```

A classe Aula04ShortestPaths do pacote **classexamples** apresenta um exemplo de uso destas classes.

Identificando Ciclos Simples em um Grafo Não-Direcionado

A classe **PatonCycleBase** computa todos os ciclos básicos de um grafo não direcionado. Para tal, basta criar uma instância desta classe passando o grafo sobre o qual os ciclos serão computados.

```
PatonCycleBase <String, DefaultEdge> patoncycles = new PatonCycleBase(graph);
```

E invocar o método *getCycleBasis* sobre esta instância.

A classe Aula04SimpleCycles do pacote classexamples apresenta um exemplo de uso desta classe.