Uso da Biblioteca JUNG e a Classe Mapa

Mateus Costa

Ifes - Campus Serra

January 21, 2025

O que é a Biblioteca JUNG?

- JUNG (Java Universal Network/Graph Framework) é uma biblioteca Java para modelagem, análise e visualização de grafos e redes.
- Permite trabalhar com diversos tipos de grafos e realizar operações como:
  - Adicionar/remover vértices e arestas;
  - Consultar propriedades de grafos;
  - Visualizar grafos.

Tipos de Grafos na Biblioteca JUNG

- Grafos não direcionados (ex.: `SparseMultigraph`).
- Grafos direcionados (ex.: `DirectedSparseMultigraph`).
- Grafos ponderados (usando transformadores para atribuir pesos a arestas).

Criação do Grafo

```
1  Graph<String, String> grafo = new
       SparseMultigraph<>();
```

- Instancia um grafo não direcionado.
- `SparseMultigraph` é eficiente em termos de memória.

Inserção de Vértices e Criação de Arestas

Inserir Vértice

```
1     grafo.addVertex("CidadeA");
```

Criar Aresta

```
1     grafo.addEdge("Aresta1", "CidadeA", "CidadeB
          ");
```

Impressão do Grafo

Iterar sobre vértices e arestas

```
1      for (String vertice : grafo.getVertices()) {
2          System.out.println("V rtice: " +
               vertice);
3      }
4      for (String aresta : grafo.getEdges()) {
5          System.out.println("Aresta: " + aresta);
6      }
```

Alterar Dados de um Vértice ou Aresta

- Não é possível alterar diretamente os dados de um vértice ou aresta na JUNG.
- Solução:
  - Remover o vértice ou aresta existente.
  - Adicionar um novo com os dados atualizados.

Visualização do Grafo

Exemplo de Visualização

```
1    CircleLayout<String, String> layout = new
         CircleLayout<>(grafo);
2    VisualizationImageServer<String, String> viz
         =
3        new VisualizationImageServer<>(layout,
             new Dimension(600, 600));
4    viz.getRenderContext().
         setVertexLabelTransformer(v -> v);
```

Listar Vértices Adjacentes

Iterar sobre vizinhos

```
1     for (String vizinho : grafo.getNeighbors("
          CidadeA")) {
2          System.out.println("Vizinho: " + vizinho
               );
3     }
```

O Problema do Caixeiro Viajante

- O Problema do Caixeiro Viajante (TSP) consiste em encontrar o circuito hamiltoniano de menor custo em um grafo.
- Aplicação em logística, roteirização e otimização de trajetos.
- O algoritmo implementado utiliza **DFS com backtracking** para explorar todas as rotas possíveis.

Estruturas de Dados Utilizadas

- **visitados**: *Set¡String¿*
  - Conjunto que mantém as cidades visitadas durante a execução.
- **rotaAtual**: *List¡String¿*
  - Lista que armazena a sequência de cidades visitadas no trajeto atual.
- **menorRota**: *List¡String¿*
  - Lista que guarda a sequência do menor trajeto encontrado.
- **custoAtual**: *double[]*
  - Armazena o custo acumulado durante a execução do trajeto atual.
- **menorCusto**: *double[]*
  - Mantém o menor custo encontrado entre todos os trajetos possíveis.

Visão Geral do Algoritmo

- Utilizamos **DFS com backtracking** para explorar todas as rotas possíveis.
- Garantimos que:
  - Todas as cidades sejam visitadas exatamente uma vez.
  - O circuito seja fechado retornando à cidade inicial.
- Atualizamos o menor custo e a rota correspondente ao encontrar uma solução válida.

Implementação do Algoritmo

```
1  private void visita(String cidadeAtual, String cidadeInicial, Set<String>
       visitados,
2                      List<String> rotaAtual, List<String> menorRota, double[]
                           custoAtual, double[] menorCusto) {
3      visitados.add(cidadeAtual);
4      rotaAtual.add(cidadeAtual);
5
6      if (visitados.size() == grafo.getVertexCount()) {
7          if (grafo.findEdge(cidadeAtual, cidadeInicial) != null) {
8              double custoFinal = custoAtual[0] + getPesoAresta(cidadeAtual,
                   cidadeInicial);
9              if (custoFinal < menorCusto[0]) {
10                 menorCusto[0] = custoFinal;
11                 menorRota.clear();
12                 menorRota.addAll(rotaAtual);
13                 menorRota.add(cidadeInicial);
14             }
15         }
16     } else {
17         for (String cidadeAdjacente : grafo.getNeighbors(cidadeAtual)) {
18             if (!visitados.contains(cidadeAdjacente)) {
19                 double pesoAresta = getPesoAresta(cidadeAtual, cidadeAdjacente);
20                 custoAtual[0] += pesoAresta;
21                 visita(cidadeAdjacente, cidadeInicial, visitados, rotaAtual,
                       menorRota, custoAtual, menorCusto);
22                 custoAtual[0] -= pesoAresta;
23             }
24         }
25     }
26     rotaAtual.remove(rotaAtual.size() - 1);
27     visitados.remove(cidadeAtual);
28 }
```

Exemplo de Teste de Mesa

Mapa de Entrada:

- Cidades: **A, B, C, D**
- Matriz de Adjacências:

	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

Execução do Algoritmo (Tabela de Progresso)

Progresso da Execução:

Passo	Cidade Atual	Rota Atual	Custo Atual	Menor Custo
1	A	{A}	0	∞
2	B	{A, B}	10	∞
3	C	{A, B, C}	45	∞
4	D	{A, B, C, D}	75	∞
5	A	{A, B, C, D, A}	95	95

Exercício 1: Verificação de Conectividade

Tarefa:

- Implemente um método que verifique se o grafo é conectado antes de executar o TSP.
- Utilize BFS ou DFS para determinar se todas as cidades são alcançáveis a partir de qualquer cidade.

Exercício 2: Contagem de Caminhos Hamiltonianos

Tarefa:

- Modifique o código do TSP para contar o número total de circuitos hamiltonianos possíveis no grafo.

Exercício 3: Exibição em Matriz

Tarefa:

- Crie um método que exiba o grafo no formato de matriz de adjacências.
- Cada posição (i, j) da matriz deve representar o peso da aresta entre as cidades i e j.

Exercício 4: Impacto de Novas Cidades

Tarefa:

- Adicione uma nova cidade ao grafo, conectando-a a outras cidades com pesos aleatórios.
- Execute o TSP antes e depois da adição e compare os resultados.

Exercício 5: Visualização do Progresso

Tarefa:

- Modifique o algoritmo do TSP para exibir o progresso durante a execução.
- Mostre a cidade atual, o custo acumulado e a rota parcial sendo explorada.

Exercício 6: Limitação de Tempo

Tarefa:

- Adicione uma limitação de tempo ao TSP.
- Caso o tempo limite seja atingido (por exemplo, 5 segundos), interrompa a execução e imprima o melhor caminho encontrado até o momento.

Exercício 7: Heurística para o TSP

Tarefa:

- Implemente uma solução heurística para o TSP.
- Escolha sempre a aresta de menor peso disponível ao construir o caminho, em vez de explorar todas as rotas possíveis.

Exercício 8: Variações no Grafo

Tarefa:

- Multiplique todos os pesos das arestas por um fator fixo (por exemplo, 2).
- Execute o TSP antes e depois da modificação e compare os resultados.

Exercício 9: Rota de Maior Custo

Tarefa:

- Modifique o algoritmo para encontrar o circuito hamiltoniano de maior custo em vez do menor.

Exercício 10: Comparação com Força Bruta

Tarefa:

- Implemente uma solução baseada em permutações para resolver o TSP.
- Compare os resultados e o tempo de execução com o algoritmo recursivo implementado.