1 Problema

Nesta atividade prática a sua tarefa vai ser ajudar Natal a combater a Dengue. Devido as problemas técnicos:

E outros problemas não tão técnicos assim...

A prefeitura da cidade do Natal está com poucos veículos fumacê disponíveis para aplicar o inseticida usado contra o mosquito da Dengue. Dessa forma, vamos ajuda-los fornecendo um planejamento próximo do ótimo de como as ruas devem ser percorridas. Com esse planejamento, a prefeitura poderá gastar menos inseticida, menos combustível e menos tempo para tratar uma área com foco do mosquito. Logo, mais regiões da cidade poderão ser tratadas com os veículos disponíveis.

2 Introdução

Na pesquisa operacional(`https://pt.wikipedia.org/wiki/Investigaç~ao_operacional`), o problema abordado aqui nesta prática computacional é conhecido como o Problema do Carteiro Chinês também conhecido como Problema de Inspeção de Rotas(`http://pt.wikipedia.org/wiki/Problema_da_inspe%C3%A7%C3%A3o_de_rotas`). Nesse problema um carteiro deve visitar todas as ruas de um bairro de forma que o seu trajeto começe e termine no mesmo lugar(ciclo). No entanto, ele deve fazer o menor percurso possível, logo, ele deve tentar visitar uma rua uma única vez. Então vamos usar esse problema para determinar o percurso que o veículo deve fazer.

Para resolver este problema iremos recorrer a uma das mais poderosas ferramentas de modelagem existente: Grafos!

Se o grafo que representa a área a ser percorrida for do tipo Euleriano, então, existirá com certeza um ciclo que irá visitar todas as arestas do grafo somente uma vez(Ciclo Euleriano). Caso contrário, isso não é possivel. Logo, toda vez que grafo não for euleriano vamos ter que percorrer algumas arestas mais de uma vez. Um grafo é dito euleriano se TODOS OS SEUS VÉRTICES POSSUIREM GRAU PAR, em outras palavras, todo vértice sempre terá um número par de vétices adjacentes.

Dessa forma, a heurística a ser implementada aqui irá seguir, basicamente, os seguintes passos:

1. Inicializar o grafo;
2. Verificar se ele é euleriano;
3. Se não for desse tipo, aplicar algum método para torna-lo euleriano adicionando novas arestas nos vértices de grau impar do grafo;
4. buscar um ciclo euleriano(algoritmo de Hierholzer);

Não se preocupe em como iremos fazer isso tudo agora, pois tudo será explicado passo-a-passo.

2.1 Modelagem do Problema

Como pode ser observado na figura do grafo acima, cada vértice do grafo representará um encontro de ruas, logo, cada aresta(u, v) irá representar uma rua no nosso grafo que liga o vértice u ao vértice v. Cada aresta do grafo será modelada pela classe **Edge.java**. Esta classe possui os seguintes atributos e funções:

- ***int u*** - indice do primeiro vértice;

- ***int v*** - indice do segundo vértice;
- ***double weight*** - peso da aresta;
- ***int getU()*** - função para obter o valor de u
- ***int getV()*** - função para obter o valor de v
- ***double getWeight()*** - função para obter o valor de *weight*
- ***String toString()*** - função para obter a representação em String de Edge;
- ***Edge Edge(int u, int v, double weight)*** - construtor da classe Edge;

Um objeto da classe **Graph.java** vai ser usado para representar um grafo. Esta classe possui os seguintes atributos e funções:

- ***nVertices*** - número de vértices existentes no grafo;
- ***int nEdges*** - número de arestas existentes no grafo ;
- ***ArrayList<LinkedList<Edge>>*** adjacentList - lista de adjacência do grafo. Cada indice de *adjacentList* corresponde a um LinkedList<Edge> das arestas que ligam ele aos seus vizinhos. Por exemplo, na posição 0 estarão as arestas que ligam 0 aos seus vizinhos;
- ***boolean existEdges()*** - retorna *true* se existem arestas no grafos;
- ***void turnEulerian()*** - método que transforma um grafo não-euleriano em euleriano;
- ***void markNode(int vertice, Color color)*** - método usado para marcar um vértice no mapa;
- ***String toString()*** - função para obter a representação em String de Graph;

Uma vez que o problema já foi devidamente modelado, mãos a obra!!

3 Passo 1 - Adicionando Arestas

Na classe **Graph.java**, complete o método **public void addEdge(int u, int v, double weight)**. Esse método deve popular adjacentList com os objetos *Edge* necessários de acordo com os parâmetros informados. LEMBRE-SE de atualizar o número de arestas existentes no grafo.

Para testar este passo use o *test1()* presente no arquivo Hierholzer.java. Você deve obter como saída:

```
Lista de adjacencia do grafo:
(0,1|5.0) (0,2|5.0) (0,3|5.0) (0,4|5.0)
(1,0|5.0) (1,4|5.0)
(2,0|5.0) (2,4|5.0)
(3,0|5.0) (3,4|5.0)
(4,0|5.0) (4,1|5.0) (4,2|5.0) (4,3|5.0)
```

4 Passo 2 - Removendo Arestas

Na classe **Graph.java**, complete o método **public void removeEdge(Edge edge)**. Esse método deve remover de adjacentList os objetos *Edge* necessários de aresta passada como parâmetro da função. LEMBRE-SE de atualizar o número de arestas existentes no grafo.

Para testar este passo use o *test2()* presente no arquivo Hierholzer.java. Você deve obter como saída:

```
Lista de adjacencia do grafo ANTES da remocao:
(0,1|5.0) (0,2|5.0) (0,3|5.0) (0,4|5.0)
(1,0|5.0) (1,4|5.0)
(2,0|5.0) (2,4|5.0)
(3,0|5.0) (3,4|5.0)
(4,0|5.0) (4,1|5.0) (4,2|5.0) (4,3|5.0)

Lista de adjacencia do grafo DEPOIS da remocao:
(0,1|5.0) (0,3|5.0)
(1,0|5.0)
(2,4|5.0)
(3,0|5.0) (3,4|5.0)
(4,2|5.0) (4,3|5.0)
```

5 Passo 3 - Obtendo os Vizinhos de um Vértice

Agora que já podemos adicionar e remover arestas de um grafo devemos ser capazes de obter de alguma forma os vizinho de um vértice específico. Para isso, complete o método **public LinkedList<Edge> getNeighbours(int index)**. Ele irá receber um inteiro indicando qual vértice que os vizinhos serão retornados. Ele deve retornar um *LinkedList<Edge>* contendo uma lita com as arestas que ligam o vértice do *index* com os seus adjacentes.

Para testar este passo use o *test3()* presente no arquivo Hierholzer.java. Você deve obter como saída:

```
Vertices vizinhos do vertice 0:
1 2 3 4
Vertices vizinhos do vertice 2:
0 4
```

6 Passo 4 - Verificando se o Grafo é Euleriano

Nesta etapa devemos ser capazes de saber se um grafo é do tipo euleriano observando somente a sua estrutura de adjacencia. Complete o método **public boolean isEulerian()**. Ele deve retornar *true* quando o grafo é euleriano e *false* caso contrário.

Para testar este passo use o *test4()* presente no arquivo Hierholzer.java. Você deve obter como saída:

```
Lista de adjacencia do grafo 1:
(0,1|5.0) (0,2|5.0) (0,3|5.0) (0,4|5.0)
(1,0|5.0) (1,4|5.0)
(2,0|5.0) (2,4|5.0)
(3,0|5.0) (3,4|5.0)
(4,0|5.0) (4,1|5.0) (4,2|5.0) (4,3|5.0)
Euleriano:true

Lista de adjacencia do grafo 2:
```

```
(0,1|5.0) (0,2|5.0) (0,3|5.0)
(1,0|5.0) (1,4|5.0) (1,2|5.0)
(2,0|5.0) (2,1|5.0) (2,3|5.0) (2,5|5.0)
(3,0|5.0) (3,2|5.0) (3,5|5.0)
(4,1|5.0) (4,5|5.0)
(5,2|5.0) (5,3|5.0) (5,4|5.0)
Euleriano: false
```

7 Passo 5 - Procurando um Ciclo Simples

Agora devemos implementar um método que seja capaz de encontrar um ciclo simples no grafo, ou seja, ele deve começar e terminar em um mesmo vértice. Não se preocupe se o grafo é euleriano ou não nesta etapa. Simplemente procure por um ciclo simples no grafo. Complete o método **LinkedList<Integer> SearchSimpleCycle(int startVertice)** contido na classe Hierholzer. Ele receberá como parâmetro o vértice de inicio/fim do ciclo e retornará um *LinkedList<Integer>* com os vértices do ciclo encontrado. OBS: Toda vez que uma aresta é incluida no ciclo ela será removida do grafo.

Para testar este passo use o *test5(int startVertice)* presente no arquivo Hierholzer.java. Um exemplo de saida para um ciclo começando do vértice 0 será(dependendo da sua implementação a saida pode ser diferente):

```
Ciclo Simples Encontrado: comecando de 0
0 -> 1 -> 4 -> 0
```

8 Passo 6 - Procurando um Ciclo Euleriano

Antes de iniciar a busca por um ciclo euleriano vamos atualizar o método **LinkedList<Integer> SearchSimpleCycle(int startVertice)** para que seja possível usar a classe **GraphicsUtils** para visualizar graficamente o quê o algoritmo está fazendo.

Vamos precisar marcar os vértices que fazem parte do ciclo e vamos usar a função **public void markNode(int vertice, Color color)** de **Graph.java** para isso. Nela você deve informar como parâmetros o vértice a ser marcado e a cor que será usada. ATENÇÃO: cada vértice pertencente a um mesmo ciclo simples deve ser pintado pela mesma cor. Para obter um objeto do tipo *Color* de uma cor aleatória use a função **public static Color getRadomColor()** da classe **Hierholzer.java**. Também será necessário usar o método **slow()** para desacelerar a busca por um ciclo simples. Finalmente, com o fim de saber qual foi o ciclo simples encontrado, o imprima usando o método **public static String CycleToString(LinkedList<Integer> cycle)** antes de retorna-lo com o *return*.

Para encontar um ciclo euleriano iremos implementar um algoritmo com base no desenvolvido por Hierholzer. Ele têm como metodologia encontrar, primeiramente, um ciclo simples no grafo. Em seguida, ele busca por ciclos simples partindo dos vértices pertencentes a esse primeiro ciclo encontrado. Nessa ultima etapa todo novo ciclo encontrado é inserido no final do ciclo inicial. Dessa forma, ao final teremos somente um único ciclo que passa por todas as arestas do grafo somente uma única vez. Ele deve seguir os seguintes passos:

1. verifique se o grafo é euleriano e tome, se necessário, as medidas adequadas(método **public void turnEulerian()**);
2. busque um ciclo simples e armazene-o na estrutura **eulerianCycle**;
3. enquanto existir arestas no grafo procure por novos ciclos simples que partam dos nós encontrados no ciclo do passo acima;

4. insira no primeiro ciclo encontrado(passo 2) todos os ciclos que forem achados no passo 3 com o método (**private void InsertCycle(LinkedList<Integer> newCycle, int index)**) ;

5. retorne o ciclo final;

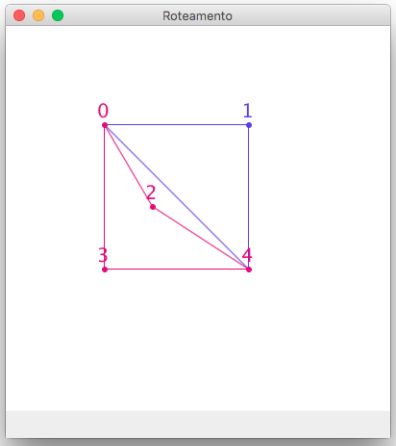
Para testar este passo use o *test6()* presente no arquivo Hierholzer.java. Para este teste uma possível saída, dependendo de como o algoritmo foi implementado, pode ser:

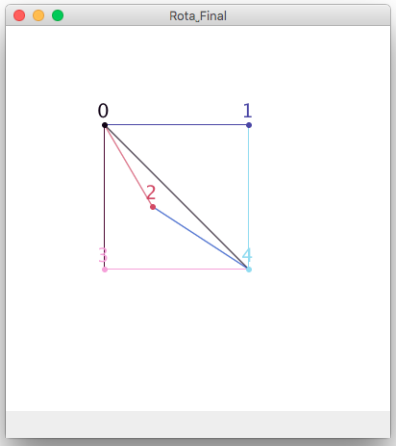
```
Ciclo Simples partindo de 0 :
0 -> 1 -> 4 -> 0

Ciclo Simples partindo de 0 :
0 -> 2 -> 4 -> 3 -> 0

Ciclo_Final:
0 -> 2 -> 4 -> 3 -> 0 -> 1 -> 4 -> 0
```

As imagens obtidas podem ser coparadas com as seguinte:





9 Aplique nos Mapas!

Uma vez que tudo está pronto teste nos mapas!!! Use o *test7()* para isso. Para este teste uma possível saída, dependendo de como o algoritmo foi implementado, pode ser:

```
Ciclo Simples partindo de 0 :
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 8 -> 7 -> 4 -> 11 -> 10 -> 3 -> 2 -> 9 -> 1 ->
15 -> 14 -> 0

Ciclo Simples partindo de 5 :
5 -> 7 -> 12 -> 11 -> 18 -> 17 -> 10 -> 9 -> 16 -> 15 -> 22 -> 21 -> 14 -> 22 ->
23 -> 16 -> 17 -> 24 -> 23 -> 24 -> 25 -> 18 -> 19 -> 12 -> 13 -> 8 -> 5

Ciclo Simples partindo de 25 :
25 -> 26 -> 19 -> 20 -> 13 -> 20 -> 27 -> 26 -> 25
```

```
Ciclo Final:
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 7 -> 12 -> 11 -> 18 -> 17 -> 10 -> 9 -> 16 -> 15 ->
22 -> 21 -> 14 -> 22 -> 23 -> 16 -> 17 -> 24 -> 23 -> 24 -> 25 -> 26 -> 19 ->
20 -> 13 -> 20 -> 27 -> 26 -> 25 -> 18 -> 19 -> 12 -> 13 -> 8 -> 5 -> 6 -> 8 ->
7 -> 4 -> 11 -> 10 -> 3 -> 2 -> 9 -> 1 -> 15 -> 14 -> 0
```

*Este trabalho prático computacional é de autoria de Leandro Rochink(UFRN, Brasil).