UNIVERSIDADE DE ÉVORA

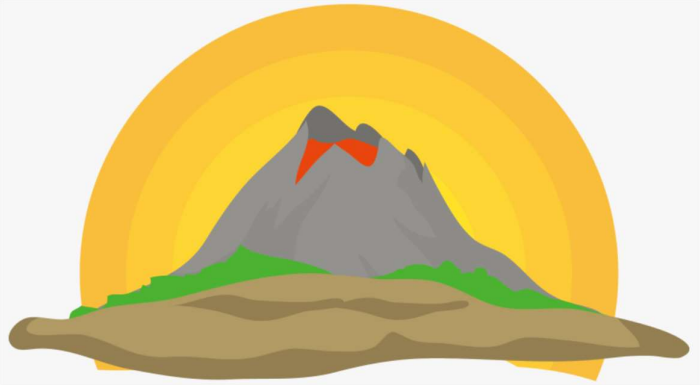
Relatório - 3º Trabalho Prático

Estrutura de Dados e Algoritmos II

Engenharia Informática

2022 - 2023

Problema: Rescue by Rail



Trabalho Realizado por:

- Daniel Barreiros nº48452
- Tomás Antunes nº48511
- Grupo do Mooshak: g311

Índice

I – Introdução

O presente relatório apresenta todos os detalhes referentes à resolução do problema "Rescue by Rail". Para a resolução deste problema procurámos obter uma solução que fosse suficientemente eficiente por forma a cumprir os parâmetros impostos pelo professor através da plataforma Mooshak bem como através da qual nos fosse possível aplicar alguma matéria lecionada nesta disciplina.

No problema apresentado existe uma ilha, composto por regiões, na qual cada região detém uma certa população. Por forma a movimentar as pessoas de lugar, cada região tem um comboio com uma capacidade limitada. As regiões estão ligadas entre si, o que não implica que obrigatoriamente cada região tenha apenas um caminho para o seu comboio. O comboio não necessita de utilizar a sua capacidade como um conjunto, pode enviar x número de pessoas por cada caminho, desde que a soma das pessoas enviadas por todos os caminhos disponíveis não exceda a capacidade máxima atribuída para o "comboio".

Assim o objetivo do problema consiste em calcular o número máximo de pessoas que é possível levar até ao ponto seguro.

Assim o objetivo do problema consiste na computação do menor número de movimentos necessários para a esfera atingir, a partir da posição inicial fornecida, o buraco definido no mapa.

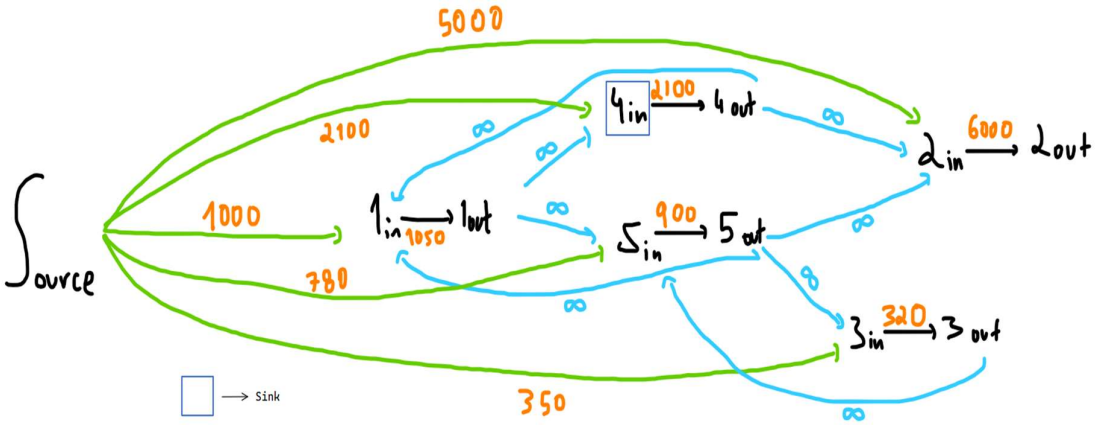
II – Resolução do Problema

2.1 – Descrição/Construção da Rede Fluxo

Após a leitura e interpretação do problema conseguimos compreender que iríamos necessitar de construir uma rede de fluxos que representasse as regiões e as linhas de comboio da ilha *Alertland*.

A rede de fluxos irá representar cada região da ilha com dois nós, um nó de entrada e um nó de saída. O nó de entrada servirá como ponto de chegada para todas as pessoas da região e todas as pessoas que chegam de comboio de outras regiões. O nó de saída servirá como ponto de partida do comboio para cada região. A rede também irá ter um nó (*source*) que irá ser responsável por distribuir o número de habitantes a cada região respetiva, através de arcos ligados desde este nó, até ao nó de entrada de cada região. As linhas de comboio para cada região serão representadas por arcos, desde os nós de saída das regiões, até aos nós de entrada das respetivas regiões de destino.

A imagem abaixo ilustra a rede de fluxos para o "Sample Input 2" do enunciado do problema:



III – Explicação do Código

3.1 – Classe Alertland

```
33  class Alertland {
34      int R;
35      int nodes;
36      int source;
37      List<Rail>[] rails;
38
39      private static final int INFINITY = Integer.MAX_VALUE;
40      private static final int NONE = -1;
41
42      @SuppressWarnings("unchecked")
43      public Alertland(int R) {
44          this.R = R;
45          this.nodes = (R*2) + 1;
46          this.source = 0;
47
48          this.rails = new List[nodes];
49          for(int i = 0; i < nodes; i++) {
50              this.rails[i] = new LinkedList<>();
51          }
52      }
```

3.1.1 – Variáveis de Classe

Na classe *Alertland* declaramos as seguintes variáveis de classe:

- int R -> número de regiões
- int nodes -> número de nós da rede de fluxos
- int source -> fonte da rede de fluxos
- List<Rail>[] rails -> Lista de adjacências de cada nó da rede de fluxos

3.1.2 – Construtor da Classe Dream

Antes de inicializarmos o construtor usamos a anotação *SuppressWarnings* para suprimir os warnings que seriam gerados pelo Java ao inicializarmos o array de listas (*rails*).

Após criarmos estas variáveis de classe seguimos para a criação do construtor da classe *Alertland*.

Este construtor recebe como argumento o número de regiões existentes na ilha *Alerland*.

Dentro deste construtor são inicializadas as variáveis declaradas acima.

3.1.3 – Método addRegion

```
54          public void addRegion(int i, int pop, int dep) {
55              rails[source].add(new Rail(i, pop));
56              rails[i].add(new Rail(i+R, dep));
57          }
```

O método *addRegion* recebe como argumento o número representante da região a ser tratada (*i*), a população respetiva a esta região, e o total de pessoas que pode viajar a partir desta.

Esta função irá criar um arco do nó *source* ao nó de entrada da respetiva região, com capacidade igual à população da região. De seguida irá criar um arco do nó de entrada da região ao nó de saída da mesma, com capacidade igual ao total de pessoas que pode viajar a partir desta.

3.1.4 – Método addRail

```
60          public void addRail(int r1, int r2) {
61              rails[r1+R].add(new Rail(r2, INFINITY));
62              rails[r2+R].add(new Rail(r1, INFINITY));
63          }
64
65          public void addRail(int r1, int r2, int c) {
66              rails[r1].add(new Rail(r2, c));
67          }
```

O método *addRail* recebe como argumentos dois inteiros representantes de duas regiões (*r1, r2*) a ser ligadas por linhas de comboio, e poderá receber também um inteiro referente à capacidade do arco (*c*).

Esta função irá criar um arco do nó de saída da região *r1* para o nó de entrada da região *r2*, com capacidade infinita. Se a capacidade for fornecida, a função irá criar um nó de *r1* a *r2* com a capacidade *c*.

3.1.5 – Método buildResidualNetwork

```
69      private Alertland buildResidualNetwork() {
70          Alertland r = new Alertland(R);
71
72          for(int i = 0; i < nodes; i++) {
73              for(Rail e : rails[i]) {
74                  int destination = e.d;
75                  int capacity = e.c;
76                  int flow = e.f;
77
78                  r.addRail(i, destination, capacity - flow);
79                  r.addRail(destination, i, flow);
80              }
81          }
82
83          return r;
84      }
```

O método *buildResidualNetwork* irá ser usado para criar uma rede residual, que corresponda ao fluxo atual na rede de fluxos.

3.1.6 – Método updateResidualCapacity

```
86      private void updateResidualCapacity(int from, int to, int capacity, int flow) {
87          for(Rail e : rails[from]) {
88              if(e.d == to) {
89                  e.c = capacity - flow;
90                  break;
91              }
92          }
93
94          for(Rail e : rails[to]) {
95              if(e.d == from) {
96                  e.c = flow;
97                  break;
98              }
99          }
100     }
```

O método *updateResidualCapacity* recebe como argumentos um inteiro *from* que representa o nó inicial de um arco, um inteiro *to* que representa o nó final do arco (*from*, *to*). Também irá receber a capacidade (*capacity*) e o fluxo (*flow*) do arco (*from, to*).

Esta função irá ser utilizada para atualizar as capacidades residuais dos arcos (*from*, *to*) e (*to*, *from*) de uma rede residual.

3.1.7 – Método incrementFlow

```
102     private void incrementFlow(int[] p, int increment, Alertland r, int sink) {
103         int v = sink;
104         int u = p[v];
105 
106         while(u != NONE) {
107             int uv = 0;
108 
109             for(Rail e : rails[u]) {
110                 if(e.d == v) {
111                     e.f += increment;
112                     r.updateResidualCapacity(u, v, e.c, e.f);
113                     uv = 1;
114                     break;
115                 }
116             }
117 
118             if(uv == 0) {
119                 for(Rail e : rails[v]) {
120                     if(e.d == u) {
121                         e.f -= increment;
122                         r.updateResidualCapacity(v, u, e.c, e.f);
123                         break;
124                     }
125                 }
126             }
127 
128             v = u;
129             u = p[v];
130         }
131     }
```

O método *incrementeFlow* irá receber como argumentos um array de inteiros que irá representar os precedentes do caminho (*p*), um inteiro que representa o incremento (*incremente*) a fazer, uma rede residual *r* e o inteiro que representa o dreno (*sink*).

Esta função irá ser utilizada para aumentar o fluxo (*flow*) em *incremente* ao longo do caminho, a começar no dreno até à fonte da rede residual *r*.

3.1.8 – Método findPath

```
133     private int findPath(int[] p, int sink) {
134         int[] cf = new int[nodes];
135 
136         Queue<Integer> Q = new LinkedList<>();
137 
138         for(int u = 0; u < nodes; u++) {
139             p[u] = NONE;
140         }
141 
142         cf[source] = INFINITY;
143         Q.add(source);
144 
145         while(!Q.isEmpty()) {
146             int u = Q.remove();
147 
148             if(u == sink) break;
149 
150             for(Rail e : rails[u]) {
151                 int v = e.d;
152 
153                 if(e.c > 0 && cf[v] == 0) {
154                     cf[v] = Math.min(cf[u], e.c);
155 
156                     p[v] = u;
157                     Q.add(v);
158                 }
159             }
160         }
161 
162         return cf[sink];
163     }
```

O método *findPath* irá receber como argumentos um array de inteiros correspondente aos precedentes do caminho (*p*), e um inteiro referente ao dreno (*sink*).

Esta função irá ser utilizada para encontrar um caminho de menor comprimento, da fonte (*source*) até ao dreno (*sink*), numa rede residual, utilizando o algoritmo de pesquisa em largura. Irá também guardar o predecessor de cada nó em *p* e devolve a capacidade residual do caminho. No caso de não haver caminho, irá retornar 0.

3.1.9 – Método edmondsKarp

```
165       public int edmondsKarp(int sink) {
166           Alertland r = buildResidualNetwork();
167           int flowValue = 0;
168           int[] p = new int[nodes];
169           int increment;
170
171           while((increment = r.findPath(p, sink)) > 0) {
172               incrementFlow(p, increment, r, sink);
173               flowValue += increment;
174           }
175
176           return flowValue;
177       }
178   }
```

O método *edmondsKarp* irá receber como argumento um inteiro referente ao dreno da rede (*sink*). Este método irá ser usado para executar o algoritmo de Edmonds-Karp, e irá devolver o valor do fluxo máximo da rede.

3.2 – Classe Rail

```
180   class Rail {
181       public int d;
182       public int c;
183       public int f;
184
185       public Rail(int destination, int capacity) {
186           this.d = destination;
187           this.c = capacity;
188
189           f = 0;
190       }
191   }
```

A classe *Point* serve apenas para representar os arcos da rede de fluxos, contendo o nó de destino, a capacidade do arco e fluxo correspondente.

IV – Cálculo de Complexidade

4.1 – Complexidade Temporal

A complexidade temporal consiste no tempo que um determinado programa necessita para completar uma tarefa da forma correta.

R – Número de regiões.

N – Número de nós (R*2) + 1.

E – Número de arcos do grafo.

As linhas 34 – 40 têm complexidade *O(1)*.

O construtor da classe *Alertland* presente nas linhas 43 – 52, tem uma complexidade *O(N)*, pois tem um loop que irá ser percorrido N vezes.

Os métodos *addRegion* e *addRail* presentes nas linhas 54 – 57 e 60 – 67, têm uma complexidade *O(1)*, pois trata-se de funções que apenas irão criar arcos da rede de fluxos e adicioná-los à respetiva lista *rails*.

O método *buildResidualNetwork* presente nas linhas 69 – 84, tem uma complexidade de $O(N + E)$.

O método *updateResidualCapacity* presente nas linhas 86 – 100, tem uma complexidade de *O(N)*.

O método *incrementFlow* presente nas linhas 102 – 131, tem uma complexidade de $O(N + E)$.

O método *findPath* presente nas linhas 133-163, tem uma complexidade de $O(N + E)$.

O método *edmondsKarp* presente nas linhas 165 – 178, tem uma complexidade $O(NE^2)$.

As linhas 180 – 191 referentes à classe *Rail* terão uma complexidade $O(1)$.

Tendo em conta as complexidades temporais calculadas, temos que:

$$O(N) + O(N + E) + O(N) + O(N + E) + O(N + E) + O(NE^2) = O(NE^2)$$

Logo, a complexidade temporal do programa será $O(NE^2)$.

4.2 – Complexidade Espacial

A complexidade espacial consiste na quantidade de memoria necessária para realizar a execução de um programa ou algoritmo.

R – Número de regiões.

N – Número de nós (R*2) + 1.

E – Número de arcos do grafo.

As linhas 34 – 40 terão uma complexidade de O(1).

O construtor da classe *Alertland* presente nas linhas 43 – 52 tem complexidade $O(N)$.

Os métodos *addRegion* e *addRails* presentes nas linhas 54 – 57 e 60 – 67, têm complexidade $O(1)$.

O método *buildResidualNetwork* presente nas linhas 69 – 84, têm complexidade $O(N + E)$.

O método *updateResidualCapacity* presente nas linhas 86 – 100, tem complexidade $O(N)$.

O método *incrementFlow* presente nas linhas 102 – 131, tem complexidade $O(N + E)$.

O método *findPath* presente nas linhas 133 – 160, tem complexidade $O(N)$.

O método *edmondsKarp* presente nas linhas 165 – 178, tem complexidade $O(N + E)$.

As linhas 180 – 191 referentes à classe Rail têm complexidade $O(1)$.

Tendo em conta as complexidades calculadas, temos que:

$$O(N) + O(N + E) + O(N) + O(N + E) + O(N) + O(N + E) = O(N + E)$$

Logo, a complexidade espacial do programa é $O(N + E)$.

V – Conclusão

Através da resolução deste problema foi-nos possível aplicar parte da matéria lecionada na disciplina, bem como desenvolver as nossas capacidades na resolução de problemas e aplicação de algoritmos.

Pelo nosso ponto de vista, consideramos que a dificuldade do problema residia na construção da rede de fluxos. Inicialmente não implementámos a rede residual, o que nos levou a não conseguir obter a resposta correta em apenas alguns casos, tornando o debugging do problema mais difícil.

Após algumas tentativas e esclarecimentos de dúvidas com o professor conseguimos perceber que o problema residia fundamentalmente no facto de não termos implementado a rede residual e no impacto que a não implementação da mesma tinha na obtenção do resultado.

Consideramos que, apesar das dificuldades encontradas, a resolução deste tipo de problemas é algo fundamental para o nosso desenvolvimento a níveis académicos e profissionais pois dá-nos uma melhor perceção e fornece-nos ferramentas importantes para desafios futuros.