UNIVERSIDADE DE ÉVORA

Relatório - 2º Trabalho Prático

Estrutura de Dados e Algoritmos II

Engenharia Informática

2022 - 2023

Problema: Was it a Dream?

Trabalho Realizado por:

- Daniel Barreiros nº48452
- Tomás Antunes nº48511
- Grupo do Mooshak: g203

Índice

I – Introdução

O presente relatório apresenta todos os detalhes referentes à resolução do problema "Was it a Dream?". 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 um mapa composto por obstáculos no qual irá existir uma esfera que se movimenta pelos espaços disponíveis. Pela leitura do enunciado compreendemos que a esfera movimenta-se apenas nas direções paralelas aos limites do mapa, sendo que, quando inicia este movimento, apenas pode mudar a sua direção ao atingir um obstáculo e nunca pode inverter o seu sentido. Deste modo, a esfera poderá em algumas situações sair do mapa, não concretizando o objetivo final que consiste em atingir o buraco existente.

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.

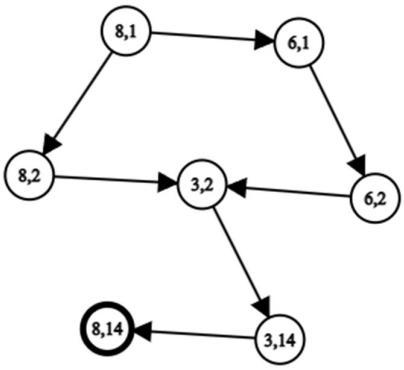
II – Resolução do Problema

2.1 – Descrição/Construção dos Grafos

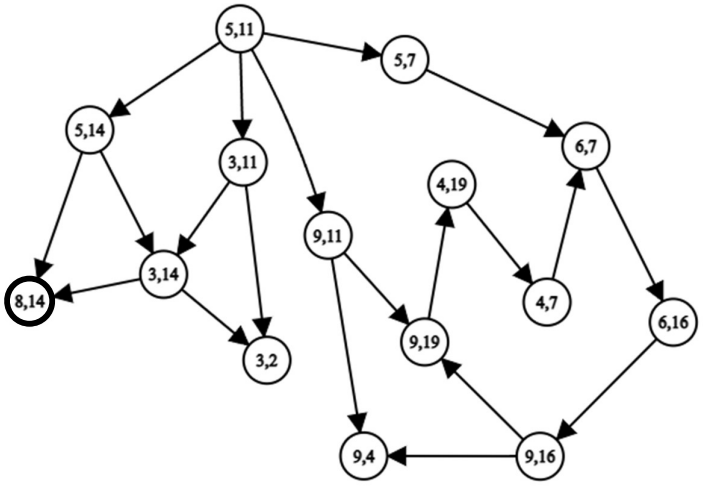
Através da observação do mapa exemplo representado no enunciado conseguimos compreender que teríamos de realizar a construção de um grafo no qual fosse possível representar todos os caminhos possíveis, desde a posição inicial até que terminassem o caminho, fosse por sair do mapa, atingir o buraco ou atingir uma posição na qual não fosse possível realizar mais nenhuma movimentação.

Deste modo o grafo irá representar os caminhos possíveis nas quatro direções para cada ponto até encontrar um obstáculo ou o buraco. Cada nó do grafo representa uma posição sem obstáculo do mapa e os arcos representam as mudanças de direção.

Apresentamos assim parte do grafo que vai representar todos os pontos ligados e os seus caminhos através de um ponto inicial. O primeiro grafo tem origem no ponto (8,1):



Como exemplo, apresentamos mais um grafo, um pouco mais complexo, com origem no ponto (5,11):



III – Explicação do Código

3.1 – Classe Main

```java
class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

        String[] tmp = input.readLine().split(" ");

        int R = Integer.parseInt(tmp[0]); // rows
        int C = Integer.parseInt(tmp[1]); // columns
        int T = Integer.parseInt(tmp[2]); // test cases

        char[][] map = new char[R][C];

        for(int i = 0; i < R; i++) {
            map[i] = input.readLine().toCharArray();
        }

        Dream dream = new Dream(R, C, map);

        for(; T > 0; T--) {
            tmp = input.readLine().split(" ");

            int steps = dream.escape(Integer.parseInt(tmp[0]) - 1, Integer.parseInt(tmp[1]) - 1);

            if(steps == -1)
                System.out.println("Stuck");
            else
                System.out.println(steps);
        }
    }
}
```

Inicialmente declaramos a classe main. Este método realiza o throw de duas exceções, sendo estas:

NumberFormatException – Esta exceção irá ocorrer se tentarmos converter uma String para um tipo numérico, mas a string não está no formato apropriado.

IOException – Esta exceção irá ocorrer no caso de o input introduzido não ser o esperado pelo programa. Um possível exemplo seria o programa estar à espera de um input do tipo inteiro e acaba por receber um input do tipo string. Desta forma, a exceção será levantada.

Para realizarmos a leitura do input declaramos um novo objeto da classe BufferedReader que irá utilizar uma nova instância da classe InputStreamReader que vai ler o input em System.in.

Após a leitura de alguns dados de input realizamos a criação de uma nova instância da classe "Dream".

Posteriormente realizamos um loop no qual irá ser lido cada ponto do mapa por forma a calcular o menor número de movimentos necessários para chegar ao buraco. No caso de ser possível chegar ao buraco a partir desse ponto irá ser mostrado o número de movimentos necessários. Caso contrário irá ser mostrada a string "Stuck".

3.2 – Classe Dream

```
39  class Dream {
40      int rows;
41      int columns;
42      int nodes;
43      char[][] map;
44      List<Point>[] adj;
45
46      int[][] pointMap;
47      int c = 1;
48
49      Point hole;
50
51      @SuppressWarnings("unchecked")
52      public Dream(int rows, int columns, char[][] map) {
53          this.rows = rows;
54          this.columns = columns;
55          this.nodes = rows * columns;
56          this.map = map;
57
58          pointMap = new int[rows][columns];
59
60          adj = new List[nodes];
61          for(int i = 0; i < nodes; i++) {
62              adj[i] = new LinkedList<>();
63          }
64
65          buildGraph();
66      }
67
```

3.2.1 – Variáveis de Classe

Na classe Dream declaramos as seguintes variáveis de classe:

int rows – Número de linhas do mapa.

int columns – Número de colunas do mapa.

int nodes – Número máximo de nós do grafo.

char[][] map – Mapa fornecido no input do problema.

List<Point>[] adj – Lista de adjacências de cada ponto do grafo.

int[][] pointMap - Mapa que contém o inteiro respetivo para cada ponto.

int c – Contador de pontos.

Point hole – Coordenadas do buraco.

3.2.2 – 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 (*adj*).

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

Este construtor recebe como argumentos os inteiros que representam o número de linhas e o número de colunas e recebe uma matriz do tipo char que contem o mapa.

Dentro deste construtor são inicializadas as variáveis declaradas acima e é chamada a função *buildGraph*.

3.2.3 – Método getPoint

```
68      public int getPoint(Point u) {
69          if(pointMap[u.x][u.y] == 0) {
70              pointMap[u.x][u.y] = c;
71              c++;
72          }
73
74          return pointMap[u.x][u.y] - 1;
75      }
```

O método *getPoint* recebe como argumento um Point. Com o ponto recebido em argumento esta função irá verificar se este ponto já foi declarado. No caso de não ter sido declarado este irá sê-lo, retornando o inteiro correspondente a este ponto na matriz *pointMap*. Se este já tiver sido declarado, apenas será retornado o inteiro correspondente.

3.2.4 – Método addEdge

```
77      public void addEdge(Point u, Point v) {
78          if(map[v.x][v.y] == 'H') {
79              hole = v;
80          }
81
82          adj[getPoint(u)].add(v);
83      }
```

O método *addEdge* recebe como argumentos dois Points.

Primeiramente verifica se o segundo ponto representa o buraco (H) no mapa, inicializando, caso se verifique, a variável *hole*.

Após esta verificação é adicionado o segundo ponto à lista de adjacências do primeiro ponto.

3.2.4 – Método buildArch

```
87      public void buildArch(int x, int y, int v) {
88          for(int i = x; i >= 0 && i < rows; i += v) {
89              char tmp = map[i][y];
90
91              if(tmp == 'O') {
92                  if(i-v == x) break;
93                  addEdge(new Point(x,y), new Point(i-v, y));
94                  break;
95              } else if(tmp == 'H') {
96                  addEdge(new Point(x, y), new Point(i, y));
97                  break;
98              }
99          }
100
101         for(int j = y; j >= 0 && j < columns; j += v) {
102             char tmp = map[x][j];
103
104             if(tmp == 'O') {
105                 if(j-v == y) break;
106                 addEdge(new Point(x,y), new Point(x, j-v));
107                 break;
108             } else if(tmp == 'H') {
109                 addEdge(new Point(x, y), new Point(x, j));
110                 break;
111             }
112         }
113     }
```

O método *buildArch* recebe como argumento três inteiros (x, y, v) que irão representar a coordenada x e y de um ponto e a direção para a qual a esfera se está a dirigir. Este último inteiro representa através de -1 ou 1 a direção dos eixos para o qual a esfera se está a dirigir.

Se o valor for -1 representa o deslocamento para cima e para a esquerda. Já se o valor for 1 representa o deslocamento para a direita e para baixo.

Dentro desta função são executados dois ciclos for. Um que irá percorrer o mapa na vertical e outro que irá percorrer o mapa na horizontal.

Dentro destes ciclos será verificada cada posição percorrida até que seja encontrado um obstáculo, o buraco ou saia dos limites do mapa. Se for verificado que uma posição corresponde a:

- Obstáculo: irá ser adicionada a posição anterior à lista de adjacências do ponto inicial (x, y).
- Buraco (H): é adicionada essa mesma posição à lista de adjacências do ponto inicial.

Se sair dos limites do mapa: não é adicionado nenhum ponto à lista de adjacências do ponto inicial.

3.2.5 – Método buildGraph

```
115         public void buildGraph() {
116             for(int i = 0; i < rows; i++) {
117                 for(int j = 0; j < columns; j++) {
118                     if(map[i][j] == '.') {
119                         buildArch(i, j, -1);
120                         buildArch(i, j, 1);
121                     }
122                 }
123             }
124         }
```

Este método irá percorrer o mapa por forma a que para cada posição em que seja possível a realização do movimento da esfera (Representado por *“.”*) seja chamada a função *buildArch*. Será realizada a chamada da função duas vezes por forma a que sejam tratados todos os casos de possível movimentação (cima e esquerda com v = -1; direita e baixo com v = 1).

3.2.6 – Método escape

```
126         public static final int INFINITY = Integer.MAX_VALUE;
127         public static final int NONE = -1;
128 
129         public int escape(int sx, int sy) {
130             int s = getPoint(new Point(sx, sy));
131             int[] color  = new int[nodes];
132             int[] d = new int[nodes];
133 
134             for(int u = 0; u < nodes; u++) {
135                 color[u] = 0;
136                 d[u] = INFINITY;
137             }
138 
139             color[s] = 1;
140             d[s] = 0;
141 
142             Queue<Integer> Q = new LinkedList<>();
143             Q.add(s);
144 
145             while(!Q.isEmpty()) {
146                 int u = Q.remove();
147 
148                 for(Point v : adj[u]) {
149                     int vi = getPoint(v);
150                     if(color[vi] == 0) {
151                         color[vi] = 1;
152                         d[vi] = d[u] + 1;
153 
154                         if(!v.equals(hole)) Q.add(vi);
155                     }
156                 }
157             }
158 
159             int dHole = d[getPoint(hole)];
160             if(dHole == Integer.MAX_VALUE)
161                 return -1;
162             return dHole;
163         }
164     }
```

Este método irá receber como argumento as coordenadas x e y do ponto do mapa a ser estudado. Irá utilizar o algoritmo BFS (pesquisa em largura) para encontrar o caminho mais curto desde o ponto inicial dado até ao buraco.

No caso de encontrar o caminho mais curto irá retornar o número de movimentos necessários para atingir o buraco desde a posição inicial.

No caso de não encontrar um caminho da posição inicial até ao buraco irá retornar o valor -1 que irá representar que a mesma está “Stuck”.

3.3 – Classe Point

```
166   class Point {
167       public int x;
168       public int y;
169
170       public Point(int x, int y) {
171           this.x = x;
172           this.y = y;
173       }
174   }
```

A classe *Point* serve apenas para representar as coordenadas de um ponto do mapa.

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 linhas.

C – Número de colunas.

N – Número de nós (R*C).

E – Número de arcos do grafo.

T – Número de test cases.

As linhas 14, 15 e 16 tem uma complexidade *O(1).*

A linha 18 apresenta uma complexidade de *O(R*C)* pois é a criação de uma matriz com R linhas por C colunas.

As linhas 20 - 22 apresentam complexidade *O(R)* visto que se trata de um ciclo que vai ser executado R vezes.

A linha 24 tem complexidade *O(R*C)* pois é chamado o construtor da classe *Dream* que apresenta esta complexidade.

As linhas 26 – 34 apresentam complexidade *O(T*N + E)* pois trata-se de um ciclo que irá ser executado T vezes e dentro deste ciclo é chamada a função *escape* da classe *Dream* que consiste no algoritmo BFS que apresenta uma complexidade de *O(N+E).* Ao adicionarmos estas complexidades obtemos *O(T*N+E).*

A linhas 40 – 49 têm uma complexidade *O(1).*

As linhas 52 – 66 apresentam uma complexidade de *O(R*C*max(R, C))* pois irá inicializar as variáveis de classe, incluindo a matriz *pointMap* o que apresenta uma complexidade de *O(R*C)* pois tem R linhas e C colunas. De seguida é inicializado o array de listas adj que irá ter tamanho N, ou seja, terá uma complexidade de *O(N)* seguido de um ciclo com complexidade *O(N)* pois é executado N vezes. Por último é chamada a função buildGraph desta mesma classe que tem uma complexidade de *O(R*C*max(R, C)).*

As linhas 68 – 85 são caracterizadas por operações de complexidade constante, logo, têm uma complexidade de *O(1).*

As linhas 87 – 113 apresentam uma complexidade de *O(max(R, C))* pois apresenta dois ciclos. Um ciclo é executado no máximo R vezes enquanto o outro é executado no máximo C vezes.

As linhas 115 – 124 apresentam uma complexidade de *O(R*C*max(R, C))* pois apresenta dois ciclos contidos um no outro sendo que o ciclo de fora vai ser executado R vezes e o de dentro C vezes. É chamada a função *buildArch* duas vezes por cada ponto navegável pela esfera, o que apresenta uma complexidade de *O(max(R, C)),* ficando assim no final uma complexidade de *O(R*C*max(R,C))*.

As linhas 126 e 127 têm uma complexidade *O(1)*.

As linhas 129 – 164 apresentam uma complexidade de *O(N+E)* pois trata-se do algoritmo BFS.

As linhas 166 – 174 são compostas por operações de complexidade constante logo apresentam uma complexidade *O(1)*.

Tendo em conta as complexidades temporais obtidas a complexidade final do programa será *O(T*(N+E)+R*C*max(R,C))*.

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 linhas.

C – Número de colunas.

N – Número de nós (R*C).

As linhas 14, 15 e 16 têm uma complexidade de *O(1)* pois cada uma destas linhas aloca memória para armazenar apenas um inteiro.

A linha 18 tem uma complexidade de *O(R*C)* pois aloca memória para uma matriz de R linhas por C colunas.

A linha 24 tem complexidade de *O(R*C*max(R, C))* pois invoca o construtor da classe *Dream* que apresenta esta complexidade.

A linha 27 tem complexidade de *O(k)* em que k representa o número de elementos da String.

A linha 29 tem complexidade de *O(N)* pois embora aloque espaço apenas para um inteiro, esta irá invocar a função *escape* da classe *Dream* que é composta pelo algoritmo BFS que apresenta complexidade espacial *O(N)*.

As linhas 40, 41, 42 e 47 têm complexidade *O(1)* pois alocam memória para apenas um inteiro.

A linha 49 tem complexidade *O(1*) pois aloca memória para uma instância da classe *Point*.

A linha 52 – 56 tem complexidade de *O(R*C*max(R, C))* pois aloca memória para uma matriz com R linhas e C colunas o que apresenta uma complexidade *O(R*C),* um array do tipo List composto por N posições o que apresenta uma complexidade *O(N),* em que cada posição vai ser alocada memória para uma LinkedList, o que apresenta uma complexidade de O(k) sendo que k é o número de elementos presentes na lista.

As linhas 68 – 85 apresentam uma complexidade constante pois trata-se de um número constante de operações com complexidade espacial *O(1)*.

As linha 87 – 113 e 115 – 124 apresentam uma complexidade *O(1)* pois não é alocado espaço para nenhuma nova estrutura de dados.

As linhas 126 e 127 apresentam uma complexidade *O(1)* pois alocam memória apenas para um inteiro (cada).

As linhas 129 – 164 apresentam uma complexidade *O(N)* pois trata-se do algoritmo BFS que apresenta esta complexidade.

Tendo em conta as complexidades espaciais obtidas a complexidade final do programa será *O(R*C*max(R, C)+N)*.

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 em encontrar a forma mais eficiente de percorrer o mapa.

Após algumas tentativas e esclarecimentos de dúvidas com o professor, conseguimos tornar o nosso código mais eficiente alterando operações de elevado custo por operações com um menor custo.

Uma das alterações realizadas consistiu na troca de uma matriz do tipo *string* que armazenava apenas caracteres por uma matriz do tipo *char*.

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.