

# Relatório Trabalho de ADA – Lost

Ano Letivo	2020/2021	Semestre	2	Cadeira	ADA
Alunos	Gonalo Martins Loureno n55780				
	Joana Soares Faria n 55754				

## Complexidade Temporal

Na anlise da complexidade temporal consideramos as seguintes variveis:

$p$  - nmero de caminhos possveis (arcos do grafo)

$i$  - nmero de localizaes existentes (clulas da ilhas)

Para computar a soluo foi escolhido o algoritmo de Bellman-Ford. Para este algoritmo comeamos com a inicializao de um array de tamanho igual ao total de nmero de posies na ilha ( $i$ ). Este array tem que ser inicializado em todas as posies com  $+\infty$ , pelo que temos um custo associado a este passo de  $\theta(i)$ .

Depois, temos um ciclo executado para todos os vrtices do grafo, mas que pode parar antes, pelo que  executado, no mximo,  $i$  vezes. Dentro deste ciclo temos uma srie de passos constantes, como testes de condies e depois todos os arcos a considerar so iterados. Os arcos a iterar esto divididos por tipos (arcos associados a clulas de gua, a clulas de erva e a rodas mgicas) e apenas so iterados os arcos possveis de serem utilizados pelo jogador em questo. Sendo assim este ciclo  executado, no mximo,  $p$  vezes. Conclumos ento que este passo tm uma complexidade de  $O(p \times i)$ .

O algoritmo  executado duas vezes, uma vez para cada jogador, uma vez que o grafo que representa os movimentos possveis de cada jogador difere ligeiramente. Obtemos assim uma complexidade final de  $O(2 \times (i + p \times i))$ , que simplifica para uma complexidade final de  $O(p \times i)$ .

## Complexidade Espacial

Na anlise da complexidade espacial consideramos as seguintes variveis:

$p$  - nmero de caminhos possveis (arcos do grafo)

$i$  - nmero de localizaes existentes (clulas da ilhas) (que corresponde a  $r \times c$ )

$r$  - nmero de linhas da ilha

$c$  - nmero de colunas da ilha

$w$  - nmero de rodas mgicas

Para a complexidade espacial identificamos os seguintes elementos:

- Varivel *island* - uma matriz de dimenso  $r \times c$ , para guardar o tipo de cada clula/posio da ilha. Estas variveis  apenas utilizada na construo do grafo, para que os arcos do grafo possam ser adicionados corretamente ao conjunto a que

pertencem (os arcos encontram-se divididos por tipos), ou não serem adicionados arcos para células correspondentes a obstáculos. Temos assim  $\theta(i)$ .

- Seguidamente temos três variáveis: *normalPaths*, *waterPaths*, *magicWheelPaths*, que totalizam o número de caminhos possíveis na ilha, dividido pelo tipo de caminho que são. Temos assim  $\theta(p)$ .
- A variável *magicWheels*, também utilizada para a construção do grafo, que guarda a informação da posição das rodas mágicas, dá-nos  $\theta(w)$ .
- A variáveis *places* guarda a informação sobre a codificação de cada posição da ilha, usando como chave a posição  $(x, y)$  da célula e como valor a codificação correspondente, que varia de 0 a  $r \times c$ . Dado é que necessário uma entrada por posição da ilha temos uma complexidade espacial de  $\theta(i)$ .
- Por fim necessitamos, para a computação do algoritmo de um vetor com tamanho igual ao total de número de vértices do grafo, *lengths*, que acresce uma complexidade temporal de  $\theta(i)$ .

Sendo assim temos uma complexidade espacial de  $\theta(i + p + w + i + i)$ , dado que  $w < i < p$ , temos uma complexidade espacial simplificada de  $\theta(p)$ . Dado que cada posição da ilha pode ter 4 ligações, uma a cada célula adjacente, mais as ligações provenientes das rodas mágicas, sabemos que o número total de arcos do grafo é superior ao número total de vértices.

## Conclusões

A nossa solução responde ao problema de forma bastante eficiente e foi implementada de uma forma bastante legível na nossa opinião. No entanto, temos consciência de que não escolhemos a implementação mais eficiente e por isso sentimos a obrigação de justificar a nossa decisão.

Escolhemos resolver o problema implementado o algoritmo de Bellman-Ford, por conseguir suportar todas as especificações do problema: arcos pesados e arcos de pesos negativos, com uma complexidade temporal de  $|V| \times |A|$ . Em comparação com o algoritmo de Floyd-Warshall, cuja complexidade temporal seria algo como  $|V|^3$ , é vantajoso. Podemos limitar superiormente o número de arcos como sendo  $4 \times |V|$ , pois mesmo existindo alguns vértices com 5 arcos (rodas), os da fronteira da ilha têm menos de 4 e compensam o número de arcos a mais das rodas, assim sendo:  $|V| \times |A| < 4 \times |V|^2 < |V|^3$  (nas condições do problema).

Primeiramente, a outra solução (ou soluções) em que pensamos consiste(m) em fazer uma avaliação *à priori* da situação específica daquele problema, e, a partir dessa avaliação identificar a existência de alternativas mais eficientes do que o algoritmo de Bellman-Ford para a procura do caminho.

Assim, com a solução geral solucionada, pensámos que: uma vez que o Jonh não nada, se não existirem rodas mágicas, a procura feita para este caso pode ser uma procura em largura, pois todos os arcos têm o mesmo custo. Podemos ainda verificar, se no caso de existirem rodas mágicas, se estas têm custo 1, o que as tornaria esta solução igualmente válida.

Por outro lado, e agora uma solução um tanto quanto mais abrangente, pelo algoritmo de Dijkstra podemos procurar o caminho em todos os casos que não envolvam pesos negativos, o que nos permite descobrir o caminho da Kate, e, se não existirem pesos negativos nos arcos das rodas, do Jonh.

É de referir ainda que a implementação dos diferentes algoritmos só traria vantagens se as estruturas usadas para o grafo fossem as mais adequadas para eficiência de cada algoritmo. Como nem todos os algoritmos referidos usam a mesma representação do grafo, este teria que ser guardado de diferentes maneiras o que levaria a um aumento da complexidade espacial.

Dito isto, por interpretação do enunciado, compreendemos que seriam poucos os casos em que estas condições se verificariam, e, assim sendo, deduzimos não ser objetivo deste trabalho a implementação de diversos algoritmos, que tornariam sim a complexidade temporal mais baixa, embora apenas em circunstâncias específicas a diferença se mantivesse após simplificação, a custo de muito mais tempo de implementação, testes e cálculos explicativos do nosso raciocínio. Por estes motivos decidimo-nos pela implementação usando o algoritmo Bellman-Ford para evitar maior complexidade espacial e menor legibilidade do código.

## Anexo – Código Class Lost

---

```

1 import java.util.*;
2 /*
3  * Ada Trabalho 3 - Lost
4  *
5  * @author Joana Soares Faria n55754
6  * @author Goncalo Martins Lourenco n55780
7  */
8 public class Lost {
9     //Types of places in the island
10    public final static int GRASS = 0;
11    public final static int OBSTACLE = 1;
12    public final static int WATER = 2;
13    public final static int MAGIC_WHEEL = 3;
14    public final static int EXIT = 4;
15    //No paths constants
16    public static final int UNREACHABLE = Integer.MAX_VALUE;
17    public static final int LOST_IN_TIME = Integer.MIN_VALUE;
18    //Type of players
19    public static final int CAN_SWIM = 0;
20    public static final int CAN_USE_WHEEL = 1;
21    //Cost of paths from cells in island
22    private final static int WATER_COST = 2;
23    private final static int GRASS_COST = 1;
24    //Paths structure - way the edge is represented
25    private static final int START_PLACE = 0;
26    private static final int END_PLACE = 1;
27    private static final int PLACE_COST = 2;
28    /**
29     * Types of cells in which island's positions
30     */
31    private final int[][] island;
32    /**
33     * Paths from and to a Grass cell
34     */
35    private final List<int[]> normalPaths;
36    /**
37     * Paths to or from a water cell
38     */
39    private final List<int[]> waterPaths;
40    /**
41     * Paths from the magic wheel
42     */
43    private final List<int[]> magicWheelPaths;
44
45    /**
46     * All magic wheels. Key is the number presented and value is the magic wheel codification
47     * position (from 0 to numRows*numCols-1, the total number of cells)
48     */
49    private final Map<Integer, Integer> magicWheels;
50    /**
51     * All cells of the island. key is the (x,y) value, in integer form, and the value is the
52     * codification position (from 0 to numRows*numCols-1, the total number of cells)
53     */
54    private final Map<Integer, Integer> places;
55    private final int numRows;
56    private final int numCols;
57    private int numPlaces;
58    /**
59     * Codification of the exit cell
60     */
61    private int EXIT_POS;
62
63    public Lost(int numRows, int numCols, int numMagicWheels) {
64        this.numRows = numRows;
65        this.numCols = numCols;
66
67        normalPaths = new LinkedList<>();
68        waterPaths = new LinkedList<>();
69        magicWheelPaths = new LinkedList<>();
70
71        island = new int[numRows][numCols];
72        magicWheels = new HashMap<>(numMagicWheels);

```

```

73     places = new HashMap<>(numRows * numCols);
74     numPlaces = 0;
75
76 }
77
78 /**
79  * Adds the edges associated with a given position. The graph edges are separated by type,
80  * the edges will be added to the correspondent Type List.
81  *
82  * @param x      x position to the island's cell to evaluate
83  * @param y      y position to the island's cell to evaluate
84  * @param type    type of island's cell to evaluate
85  * @param magicWheel if the cell is a magic island this is the number presented in the grid
86  */
87 public void addIslandPosition(int x, int y, int type, int magicWheel) {
88     island[y][x] = type;
89     int start = posToInt(x, y);
90     int pos = numPlaces++;
91     places.put(start, pos);
92     if (y > 0 && type != OBSTACLE) { //has up
93         //if the current cell has an upper cell, add the 2 edges to and from the current cell
94         int upType = island[y - 1][x];
95         int upPos = places.get(posToInt(x, y - 1));
96         addPaths(type, pos, upType, upPos);
97     }
98     if (x > 0 && type != OBSTACLE) { //has left
99         //if the current cell has a left cell, add the 2 edges to and from the current cell
100        int leftType = island[y][x - 1];
101        int leftPos = places.get(posToInt(x - 1, y));
102        addPaths(type, pos, leftType, leftPos);
103    }
104
105    if (type == MAGIC_WHEEL) {
106        //if it is a magic wheel store the number presented in the grid and its codification
107        // position
108        magicWheels.put(magicWheel, pos);
109    }
110    if (type == EXIT) {
111        //store the codification position of the exit
112        EXIT_POS = pos;
113    }
114 }
115
116 /**
117  * Converts an (x,y) position to an int
118  *
119  * @param x x position
120  * @param y y position
121  * @return transformed (x,y) position to an int
122  */
123 private int posToInt(int x, int y){
124     return x * 100 + y;
125 }
126
127 /**
128  * Adds the edges between two vertices to the correspondent types list
129  *
130  * @param startType type of the start position
131  * @param startPos  codified start position
132  * @param endType   type of the end position
133  * @param endPos    codified end position
134  */
135 private void addPaths(int startType, int startPos, int endType, int endPos) {
136     int[] edgeFrom = new int[]{startPos, endPos, cost(startType)};
137     int[] edgeTo = new int[]{endPos, startPos, cost(endType)};
138
139     if (startType == WATER || endType == WATER) {
140         //if is a path connected to a water cell and is not from the exit
141         if (startType != EXIT) {
142             waterPaths.add(edgeFrom);
143         }
144         if (endType != EXIT) {

```

```

145         waterPaths.add(edgeTo);
146     }
147 } else if (endType != OBSTACLE) {
148     //if is not an obstacle add path to normal paths, bidirectional
149     if (startType != EXIT) {
150         normalPaths.add(edgeFrom);
151     }
152     if (endType != EXIT) {
153         normalPaths.add(edgeTo);
154     }
155 }
156 }
157
158 /**
159  * Computes the cost of exiting a cell
160  *
161  * @param type type of the cell to exit
162  * @return the cost
163  */
164 private int cost(int type) {
165     return type == GRASS || type == MAGIC_WHEEL ? GRASS_COST : WATER_COST;
166 }
167
168 /**
169  * Adds the edges from a magic wheel
170  *
171  * @param i magic wheel that is the start of the edge
172  * @param x x position of the end of the edge
173  * @param y y position of the end of the edge
174  * @param cost cost of the edge of the magic wheel
175  */
176 public void addMagicWheel(int i, int x, int y, int cost) {
177     int start = magicWheels.get(i);
178     int end = places.get(posToInt(x, y));
179     int[] edge = new int[]{start, end, cost};
180     magicWheelPaths.add(edge);
181 }
182 }
183
184 /**
185  * Computes the length of the path between a player's position to the exit.
186  * Considers the type of player, if he can swim or use the wheel
187  *
188  * @param originX x start position of the player
189  * @param originY y start position of the player
190  * @param playerType boolean array with the type player. If he can swim in the first position
191  * and if he can use the magic wheel in the second position
192  * @return the length of the path from the player's initial position to the exit. If the exit
193  * is unreachable returns INTEGER.MAX_VALUE. If the graph has a negative weight cycle
194  * reachable by the player returns INTEGER.MIN_VALUE
195  */
196 public int solution(int originX, int originY, boolean[] playerType) {
197     int[] lengths = new int[numRows * numCols];
198
199     Arrays.fill(lengths, UNREACHABLE);
200
201     int origin = places.get(posToInt(originX, originY));
202     lengths[origin] = 0;
203     boolean changes = false;
204     for (int i = 1; i < lengths.length; i++) {
205         changes = updateLength(lengths, playerType);
206         if (!changes) {
207             // length vector stabilized, end cycle
208             break;
209         }
210     }
211
212     //Detect negative-weight cycles
213     if (changes && updateLength(lengths, playerType)) {
214         lengths[EXIT_POS] = LOST_IN_TIME;
215     }
216

```

```

217         return lengths[EXIT_POS];
218     }
219
220     private boolean updateLength(int[] lengths, boolean[] playerType) {
221         //Iterates all edges in the graph by types. The normal edges are always considered
222         boolean changes = updateLengthsInSubPaths(lengths, normalPaths);
223         if (playerType[CAN_SWIM]) {
224             //iterated only if the player can swim
225             changes = updateLengthsInSubPaths(lengths, waterPaths) || changes;
226         }
227         if (playerType[CAN_USE_WHEEL] && magicWheels.size() > 0) {
228             //iterated only if the player can use the magic wheels
229             changes = updateLengthsInSubPaths(lengths, magicWheelPaths) || changes;
230         }
231         return changes;
232     }
233
234     /**
235     * Performs the update length of the algorithm
236     * @param lengths array of lengths used by Bellman-Ford algorithm
237     * @param paths list of the paths to consider
238     * @return if there are any changes in the vector lengths
239     */
240     private boolean updateLengthsInSubPaths(int[] lengths, List<int[]> paths) {
241         boolean changes = false;
242         for (int[] path : paths) {
243             int tail = path[START_PLACE];
244             int head = path[END_PLACE];
245             int cost = path[PLACE_COST];
246             if (lengths[tail] < Integer.MAX_VALUE) {
247                 int newCost = lengths[tail] + cost;
248                 if (newCost < lengths[head]) {
249                     lengths[head] = newCost;
250                     //continue cycle because there are changes in the length vector
251                     changes = true;
252                 }
253             }
254         }
255         return changes;
256     }
257 }
258

```



## Anexo – Código Main

---

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 /*
5  * Ada Trabalho 3 - Lost
6  *
7  * @author Joana Soares Faria n55754
8  * @author Goncalo Martins Lourenco n55780
9  */
10 public class Main {
11
12     public static void main(String[] args) throws IOException {
13
14         BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
15         int numTestCases = Integer.parseInt(input.readLine());
16         for (int i = 0; i < numTestCases; i++) {
17             Lost problem = processProblem(input);
18             String[] playersPositions = input.readLine().split(" ");
19             int yJ = Integer.parseInt(playersPositions[0]);
20             int xJ = Integer.parseInt(playersPositions[1]);
21             int yK = Integer.parseInt(playersPositions[2]);
22             int xK = Integer.parseInt(playersPositions[3]);
23             int solutionJohn = problem.solution(xJ, yJ, new boolean[]{false, true});
24             int solutionKate = problem.solution(xK, yK, new boolean[]{true, false});
25             System.out.println("Case #" + (i+1));
26             String john = solutionString(solutionJohn);
27             String kate = solutionString(solutionKate);
28             System.out.println("John " + john);
29             System.out.println("Kate " + kate);
30         }
31
32     }
33
34     private static Lost processProblem(BufferedReader input) throws IOException {
35         String[] problemInfo = input.readLine().split(" ");
36         int numRows = Integer.parseInt(problemInfo[0]);
37         int numCols = Integer.parseInt(problemInfo[1]);
38         int numMagicWheels = Integer.parseInt(problemInfo[2]);
39         Lost problem = new Lost(numRows, numCols, numMagicWheels);
40         for (int y = 0; y < numRows; y++) {
41             String row = input.readLine();
42             for (int x = 0; x < numCols; x++) {
43                 char charAt = row.charAt(x);
44                 int type = positionType(charAt);
45                 int w = -1;
46                 if (type == Lost.MAGIC_WHEEL) {
47                     w = Integer.parseInt(String.valueOf(charAt));
48                 }
49                 problem.addIslandPosition(x, y, type, w);
50             }
51         }
52         for (int i = 1; i <= numMagicWheels; i++) {
53             String[] magicWheel = input.readLine().split(" ");
54             int y = Integer.parseInt(magicWheel[0]);
55             int x = Integer.parseInt(magicWheel[1]);
56             int cost = Integer.parseInt(magicWheel[2]);
57             problem.addMagicWheel(i, x, y, cost);
58         }
59         return problem;
60     }
61
62     private static String solutionString(int solutionJohn) {
63         String string;
64         if (solutionJohn == Lost.UNREACHABLE) {
65             string = "Unreachable";
66         } else if (solutionJohn == Lost.LOST_IN_TIME) {
67             string = "Lost in Time";
68         } else {
69             string = String.valueOf(solutionJohn);
70         }
71     }
72

```

```
73     }
74     return string;
75 }
76
77 private static int positionType(char charAt) {
78     int type;
79     if (charAt == 'G') {
80         type = Lost.GRASS;
81     } else if (charAt == 'O') {
82         type = Lost.OBSTACLE;
83     } else if (charAt == 'W') {
84         type = Lost.WATER;
85     } else if (charAt == 'X') {
86         type = Lost.EXIT;
87     } else {
88         type = Lost.MAGIC_WHEEL;
89     }
90     return type;
91 }
92 }
93
```