Relatório Trabalho de ADA – Lost

Ano Letivo	2020/2021	Semestre	2	Cadeira	ADA
Alunos	Gonçalo Martins Lourenço nº55780				
	Joana Soares Faria nº 55754				

Complexidade Temporal

Na análise da complexidade temporal consideramos as seguintes variáveis:

p - número de caminhos possíveis (arcos do grafo)

i - número de localizações existentes (células da ilhas)

Para computar a solução foi escolhido o algoritmo de Bellman-Ford. Para este algoritmo começamos com a inicialização de um array de tamanho igual ao total de número de posições na ilha (i). Este array tem que ser inicializado em todas as posições com $+\infty$, pelo que temos um custo associado a este passo de $\theta(i)$.

Depois, temos um ciclo executado para todos os vértices do grafo, mas que pode parar antes, pelo que é executado, no máximo, i vezes. Dentro deste ciclo temos uma série de passos constantes, como testes de condições e depois todos os arcos a considerar serão iterados. Os arcos a iterar estão divididos por tipos (arcos associados a células de água, a células de erva e a rodas mágicas) e apenas serão iterados os arcos possíveis de serem utilizados pelo jogador em questão. Sendo assim este ciclo é executado, no máximo, p vezes. Concluímos então que este passo têm uma complexidade de $O(p \times i)$.

O algoritmo é executado duas vezes, uma vez para cada jogador, uma vez que o grafo que representa os movimentos possíveis 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 análise da complexidade espacial consideramos as seguintes variáveis:

p - número de caminhos possíveis (arcos do grafo)

i - número de localizações existentes (células da ilhas) (que corresponde a $r \times c$)

r – número de linhas da ilha

c – número de colunas da ilha

w – número de rodas mágicas

Para a complexidade espacial identificamos os seguintes elementos:

- Variável $island$ – uma matriz de dimensão $r \times c$, para guardar o tipo de cada célula/posição da ilha. Estas variáveis é apenas utilizada na construção 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 $0\ a\ r \times c$. Dados 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 Main

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main {

    public static void main(String[] args) throws IOException {

        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        int numTestCases = Integer.parseInt(input.readLine());
        for (int i = 0; i < numTestCases; i++) {
            Lost problem = processProblem(input);
            String[] playersPositions = input.readLine().split(" ");
            int yJ = Integer.parseInt(playersPositions[0]);
            int xJ = Integer.parseInt(playersPositions[1]);
            int yK = Integer.parseInt(playersPositions[2]);
            int xK = Integer.parseInt(playersPositions[3]);
            int solutionJohn = problem.solution(xJ, yJ, new boolean[]{false, true});
            int solutionKate = problem.solution(xK, yK, new boolean[]{true, false});
            System.out.println("Case #" + (i+1));
            String john = solutionString(solutionJohn);
            String kate = solutionString(solutionKate);
            System.out.println("John " + john);
            System.out.println("Kate " + kate);
        }


    }

    private static Lost processProblem(BufferedReader input) throws IOException {
        String[] problemInfo = input.readLine().split(" ");
        int numRows = Integer.parseInt(problemInfo[0]);
        int numCols = Integer.parseInt(problemInfo[1]);
        int numMagicWheels = Integer.parseInt(problemInfo[2]);
        Lost problem = new Lost(numRows, numCols, numMagicWheels);
        for (int y = 0; y < numRows; y++) {
            String row = input.readLine();
            for (int x = 0; x < numCols; x++) {
                char charAt = row.charAt(x);
                int type = positionType(charAt);
                int w = -1;
                if(type == Lost.MAGIC_WHEEL){
                    w = Integer.parseInt(String.valueOf(charAt));
                }
                problem.addIslandPosition(x, y, type, w);
            }
        }
        for (int i = 1; i <= numMagicWheels; i++) {
            String[] magicWheel = input.readLine().split(" ");
            int y = Integer.parseInt(magicWheel[0]);
            int x = Integer.parseInt(magicWheel[1]);
            int cost = Integer.parseInt(magicWheel[2]);
            problem.addMagicWheel(i, x, y, cost);
        }

        return problem;
    }

    private static String solutionString(int solutionJohn) {
        String string;
        if (solutionJohn == Lost.UNREACHABLE) {
            string = "Unreachable";

        } else if (solutionJohn == Lost.LOST_IN_TIME) {
            string = "Lost in Time";
        } else {
            string = String.valueOf(solutionJohn);
        }
        return string;
    }

    private static int positionType(char charAt) {
```

```
        int type;
        if (charAt == 'G') {
            type = Lost.GRASS;
        } else if (charAt == 'O') {
            type = Lost.OBSTACLE;
        } else if (charAt == 'W') {
            type = Lost.WATER;
        } else if (charAt == 'X') {
            type = Lost.EXIT;
        } else {
            type = Lost.MAGIC_WHEEL;
        }
        return type;
    }
}

```

Anexo – Código Class Lost

```java
import java.util.*;

public class Lost {
    //Types of places in the island
    public final static int GRASS = 0;
    public final static int OBSTACLE = 1;
    public final static int WATER = 2;
    public final static int MAGIC_WHEEL = 3;
    public final static int EXIT = 4;
    //No paths constants
    public static final int UNREACHABLE = Integer.MAX_VALUE;
    public static final int LOST_IN_TIME = Integer.MIN_VALUE;
    //Type of players
    public static final int CAN_SWIM = 0;
    public static final int CAN_USE_WHEEL = 1;
    //Cost of paths from cells in island
    private final static int WATER_COST = 2;
    private final static int GRASS_COST = 1;
    //Paths structure - way the edge is represented
    private static final int START_PLACE = 0;
    private static final int END_PLACE = 1;
    private static final int PLACE_COST = 2;
    /**
     * Types of cells in which island's positions
     */
    private final int[][] island;
    /**
     * Paths from and to a Grass cell
     */
    private final List<int[]> normalPaths;
    /**
     * Paths to or from a water cell
     */
    private final List<int[]> waterPaths;
    /**
     * Paths from the magic wheel
     */
    private final List<int[]> magicWheelPaths;

    /**
     * All magic wheels. Key is the number presented and value is the magic wheel codification
     * position (from 0 to numRows*numCols-1, the total number of cells)
     */
    private final Map<Integer, Integer> magicWheels;
    /**
     * All cells of the island. key is the (x,y) value, in integer form, and the value is the
     * codification position (from 0 to numRows*numCols-1, the total number of cells)
     */
    private final Map<Integer, Integer> places;
    private final int numRows;
    private final int numCols;
    private int numPlaces;
    /**
     * Codification of the exit cell
     */
    private int EXIT_POS;

    public Lost(int numRows, int numCols, int numMagicWheels) {
        this.numRows = numRows;
        this.numCols = numCols;

        normalPaths = new LinkedList<>();
        waterPaths = new LinkedList<>();
        magicWheelPaths = new LinkedList<>();

        island = new int[numRows][numCols];
        magicWheels = new HashMap<>(numMagicWheels);
        places = new HashMap<>(numRows * numCols);
        numPlaces = 0;

    }

```

```java
    /**
     * Adds the edges associated with a given position. The graph edges are separated by type,
     * the edges will the added to the correspondent Type List.
     *
     * @param x          x position to the island's cell to evaluate
     * @param y          y position to the island's cell to evaluate
     * @param type       type of island's cell to evaluate
     * @param magicWheel if the cell is a magic island this is the number presented in the grid
     */
    public void addIslandPosition(int x, int y, int type, int magicWheel) {
        island[y][x] = type;
        int start = posToInt(x, y);
        int pos = numPlaces++;
        places.put(start, pos);
        if (y > 0 && type != OBSTACLE) { //has up
            //if the current cell has an upper cell, add the 2 edges to and from the current cell
            int upType = island[y - 1][x];
            int upPos = places.get(posToInt(x, y - 1));
            addPaths(type, pos, upType, upPos);
        }
        if (x > 0 && type != OBSTACLE) { //has left
            //if the current cell has a left cell, add the 2 edges to and from the current cell
            int leftType = island[y][x - 1];
            int leftPos = places.get(posToInt(x - 1, y));
            addPaths(type, pos, leftType, leftPos);
        }

        if (type == MAGIC_WHEEL) {
            //if it is a magic wheel store the number presented in the grid and its codification
            // position
            magicWheels.put(magicWheel, pos);
        }
        if (type == EXIT) {
            //store the codification position of the exit
            EXIT_POS = pos;
        }
    }

    /**
     * Converts an (x,y) position to an int
     *
     * @param x x position
     * @param y y  position
     * @return transformed (x,y)  position to an int
     */
    private int posToInt(int x, int y) {
        return x * 100 + y;
    }

    /**
     * Adds the edges between two vertices to the correspondent types list
     *
     * @param startType type of the start position
     * @param startPos  codified start position
     * @param endType   type of the end position
     * @param endPos    codified end position
     */
    private void addPaths(int startType, int startPos, int endType, int endPos) {
        int[] edgeFrom = new int[]{startPos, endPos, cost(startType)};
        int[] edgeTo = new int[]{endPos, startPos, cost(endType)};

        if (startType == WATER || endType == WATER) {
            //if is a path connected to a water cell and is not from the exit
            if (startType != EXIT) {
                waterPaths.add(edgeFrom);
            }
            if (endType != EXIT) {
                waterPaths.add(edgeTo);
            }
        } else if (endType != OBSTACLE) {
            //if is not an obstacle add path to normal paths, bidirectional
            if (startType != EXIT) {
```

```
                normalPaths.add(edgeFrom);
            }
            if (endType != EXIT) {
                normalPaths.add(edgeTo);
            }
        }
    }

    /**
     * Computes the cost of exiting a cell
     *
     * @param type type of the cell to exit
     * @return the cost
     */
    private int cost(int type) {
        return type == GRASS || type == MAGIC_WHEEL ? GRASS_COST : WATER_COST;
    }

    /**
     * Adds the edges from a magic wheel
     *
     * @param i    magic wheel that is the start of the edge
     * @param x    x position of the end of the edge
     * @param y    y position of the end of the edge
     * @param cost cost of the edge of the magic wheel
     */
    public void addMagicWheel(int i, int x, int y, int cost) {
        int start = magicWheels.get(i);
        int end = places.get(posToInt(x, y));
        int[] edge = new int[]{start, end, cost};
        magicWheelPaths.add(edge);

    }

    /**
     * Computes the length of the path between a player's position to the exit.
     * Considers the type of player, if he can swim or use the wheel
     *
     * @param originX    x start position of the player
     * @param originY    y start position of the player
     * @param playerType boolean array with the type player. If he can swim in the first position
     *                   and if he can use the magic wheel in the second position
     * @return the length of the path from the player's initial position to the exit. If the exit
     * is unreachable returns INTEGER.MAX_VALUE. If the graph has a negative weight cycle
     * reachable by the player returns INTEGER.MIN_VALUE
     */
    public int solution(int originX, int originY, boolean[] playerType) {
        int[] lengths = new int[numRows * numCols];

        Arrays.fill(lengths, UNREACHABLE);

        int origin = places.get(posToInt(originX, originY));
        lengths[origin] = 0;
        boolean changes = false;
        for (int i = 1; i < lengths.length; i++) {
            changes = updateLength(lengths, playerType);
            if (!changes) {
                // length vector stabilized, end cycle
                break;
            }
        }

        //Detect negative-weight cycles
        if (changes && updateLength(lengths, playerType)) {
            lengths[EXIT_POS] = LOST_IN_TIME;
        }

        return lengths[EXIT_POS];
    }

    private boolean updateLength(int[] lengths, boolean[] playerType) {
        //Iterates all edges in the graph by types. The normal edges are always considered
```

```java
        boolean changes = updateLengthsInSubPaths(lengths, normalPaths);
        if (playerType[CAN_SWIM]) {
            //iterated only if the player can swim
            changes = updateLengthsInSubPaths(lengths, waterPaths) || changes;
        }
        if (playerType[CAN_USE_WHEEL] && magicWheels.size() > 0) {
            //iterated only if the player can use the magic wheels
            changes = updateLengthsInSubPaths(lengths, magicWheelPaths) || changes;
        }
        return changes;
    }

    /**
     * Performs the update length of the algorithm
     * @param lengths array of lengths used by Bellman-Ford algorithm
     * @param paths list of the paths to consider
     * @return if there are any changes in the vector lengths
     */
    private boolean updateLengthsInSubPaths(int[] lengths, List<int[]> paths) {
        boolean changes = false;
        for (int[] path : paths) {
            int tail = path[START_PLACE];
            int head = path[END_PLACE];
            int cost = path[PLACE_COST];
            if (lengths[tail] < Integer.MAX_VALUE) {
                int newCost = lengths[tail] + cost;
                if (newCost < lengths[head]) {
                    lengths[head] = newCost;
                    //continue cycle because there are changes in the length vector
                    changes = true;
                }
            }
        }
        return changes;
    }
}

```