NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

# ADA
# Project 3 Report
Lost

Jacinta Sousa 55075
João Bordalo 55697

May 2021

## Introduction

N: Number of Test Cases ($1 \leq N \leq 28$)
R: Number of Rows ($3 \leq R \leq 60$)
C: Number of Columns ($3 \leq C \leq 60$)
M: Number of Magic Wheels ($0 \leq M \leq 9$)
V: Number of Vertices ($V = R * C$)
A: Number of Edges

## Temporal Complexity

| | |
|---|---|
| Initializing graph: | $\Theta(V)$ |
| Adding all the magic wheels: | $\Theta(M)$ |
| bellmanFord function: | $O(V * A)$ |
| dijkstra function: | $O((V + A) * log(V))$ |
| Attaining the solution: | $= O(V + M + V * A + (V + A) * log(V)) \equiv O(V * A)$ |

## Spacial Complexity

$*$ **edgesKate**
An array of size V, where each position i has a list of edges which go in the node i.
$= O(V + A)$

$*$ **edgesJohn**
A list of all the edges, except those which connect to Water.
$= O(A)$

$*$ **magicWheels**
An integer array of size M. For each position i, it saves the tail, i.e. position on the grid, where the $i^{th}$ magical wheel is found.
$= \Theta(M)$

$*$ **grid**
A char matrix representing each cell, used during the creation of the graphs.
$= \Theta(V^2)$

$*$ **dijkstra function**
*length* - An array which saves the minimum path cost found from the origin to the $i^{th}$ node.
$= \Theta(V)$
*selected* - An array of V booleans indicating which nodes have been processed.
$= \Theta(V)$
*connected* - A priority queue that will contain the nodes never selected to which there is already a path from the origin to.
$= O(V)$

$*$ **bellman-ford function**
*length* - An array that will save the minimum path cost found from the origin to the $i^{th}$ node.
$= \Theta(V)$

$$= O((V + A) + A + M + V^2 + V + V + V + V) \equiv O(V * 2)^1$$

$^1$ The $V^2$ grid's memory is freed after the graphs are fully created so the space complexity ends up being $O(V + A)$ during execution.

## Conclusion

∗ **Strong Aspects**:
Our graphs don't keep any edges related to obstacles, saving memory. The edges for water are also not saved in John's graph (list of edges), since they would take up memory unnecessarily. Kate's solution is calculated with Dijkstra's algorithm, which is more efficient than Bellman-Ford, since she cannot take edges of negative weight.

∗ **Weak Aspects**:
Our solution keeps nodes for Obstacles even though there are no edges connecting them to any other node. The same thing happens to Water nodes in John's case. These nodes do not influence the algorithms, for they are disconnected, but take up some memory, in Kate's case and during initialization. By having two graphs we're also duplicating some of the information. However, we believe it's a positive trade-off between memory and time, allowing us to use a more efficient algorithm for Kate's case.

∗ **Alternatives Studied**:
A Bellman-Ford algorithm for both Kate and John, the edges had a type parameter to define if the person could take them or not. That way, there was only one graph. However, Bellman-Ford was not optimal for Kate's case, which had no negative weights.

∗ **Improvements**:
We could free up some memory by not keeping the nodes which aren't connected to any other nodes in the graph.

# Code Annex

## Main.java

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

public class Main {

    public final static String UNREACHABLE = "Unreachable";
    public final static String LOST_IN_TIME = "Lost in Time";

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

        int testCases = Integer.parseInt(input.readLine());

        for (int i = 0; i < testCases; i++)
            solve(input, i + 1);

        input.close();
    }

    public static void solve(BufferedReader input, int test_case) throws
        IOException {
        String[] tokens = input.readLine().split(" ");
        int rows = Integer.parseInt(tokens[0]);
        int columns = Integer.parseInt(tokens[1]);
        int magicWheels = Integer.parseInt(tokens[2]);

        Lost lost = new Lost(rows, columns, magicWheels);

        char[][] rowsInput = new char[rows][columns];

        for (int r = 0; r < rows; r++) {
            char[] row = input.readLine().toCharArray();
            rowsInput[r] = row;
        }

        lost.addRows(rowsInput);
        for (int m = 0; m < magicWheels; m++) {
            tokens = input.readLine().split(" ");
            // Magic wheel's destination cell
            int r_i = Integer.parseInt(tokens[0]);
            int c_i = Integer.parseInt(tokens[1]);
            // Magic wheel's time travelled
            int t_i = Integer.parseInt(tokens[2]);

            lost.addMagicWheel(r_i, c_i, t_i, m);
        }

        System.out.printf("Case #%d%n", test_case);

        tokens = input.readLine().split(" ");
        // John's initial position
        int r_j = Integer.parseInt(tokens[0]);
        int c_j = Integer.parseInt(tokens[1]);

        try {
            int john = lost.solveJohn(r_j, c_j);
            printResult("John", john);
        } catch (NegativeWeightCycleException e) {
            System.out.printf("John %s%n", LOST_IN_TIME);
        }

        int r_k = Integer.parseInt(tokens[2]);
        int c_k = Integer.parseInt(tokens[3]);
        int kate = lost.solveKate(r_k, c_k);
        printResult("Kate", kate);
    }

    public static void printResult(String character, int result) {
        if (result < Lost.INF) {
            System.out.printf("%s %d%n", character, result);
        } else {
            System.out.printf("%s %s%n", character, UNREACHABLE);
        }
    }
}
```

## Lost.java

```java
import java.util.AbstractMap.SimpleEntry;
import java.util.LinkedList;
import java.util.List;
import java.util.PriorityQueue;

public class Lost {

    public static final int INF = Integer.MAX_VALUE;
    private static final char GRASS = 'G';
    private static final char WATER = 'W';
    private static final char OBSTACLE = 'O';
    private static final char EXIT = 'X';

    private final int numNodes;
    private final List<Edge> edgesJohn;
    private final List<Edge>[] edgesKate;
    private int exitNode;
    private final int columns;
    private final int rows;
    private final int[] magicWheels;

    @SuppressWarnings("unchecked")
    public Lost(int rows, int columns, int magicWheels) {
        this.numNodes = rows * columns;
        this.edgesJohn = new LinkedList<>();
        this.edgesKate = new LinkedList[this.numNodes];
        this.exitNode = -1;
        this.rows = rows;
        this.columns = columns;
        this.magicWheels = new int[magicWheels];

        for (int i = 0; i < this.numNodes; i++) {
            this.edgesKate[i] = new LinkedList<>();
        }
    }

    /**
     * Checks if the character given is a magic wheel,
     * if so, it saves the tail in the magicWheels
     * (for later use in the creation of the edges related)
     * Returns the type of cell (GRASS if it is a magic wheel).
     */
    private char isMagicWheel(char cell, int tail) {
        int index = Character.getNumericValue(cell);
        if (index > 0 && index <= 9) {
            this.magicWheels[index - 1] = tail;
            return GRASS;
        }
        return cell;
    }
```

```
/**
 * Creates the edge and adds it to the graphs
 * (may not add to John's graph if it is not useful - WATER).
 */
private void createEdge(char cell, int head, int tail, char neighbour) {
    int label = cell == WATER ? 2 : 1;
    // Check for magic wheel
    cell = isMagicWheel(cell, tail);
    if (neighbour == OBSTACLE) return;
    Edge e = new Edge(head, tail, label);
    if (cell != WATER && neighbour != WATER)
        this.edgesJohn.add(e);
    this.edgesKate[tail].add(e);
}

/**
 * Given the char grid, this function will connect a node
 * to the nodes directly above, below and to his sides.
 */
public void addRows(char[][] grid) {
    char cell;
    int tail;
    // Connect the grid
    for (int i = 0; i < this.rows; i++) {
        for (int j = 0; j < this.columns; j++) {
            cell = grid[i][j];
            tail = i * this.columns + j;
            if (cell == OBSTACLE) continue;
            if (cell == EXIT) {
                this.exitNode = tail;
                continue;
            }
            // top
            if (i - 1 >= 0)
                this.createEdge(cell, (i - 1) * this.columns + j, tail, grid[i
                    - 1][j]);
            // right
            if (j + 1 < this.columns)
                this.createEdge(cell, i * this.columns + j + 1, tail,
                    grid[i][j + 1]);
            // bottom
            if (i + 1 < this.rows)
                this.createEdge(cell, (i + 1) * this.columns + j, tail, grid[i
                    + 1][j]);
            // left
            if (j - 1 >= 0)
                this.createEdge(cell, i * this.columns + j - 1, tail,
                    grid[i][j - 1]);
        }
    }
}
```

```java
/**
 * Creates the magic wheel edges and adds them to Johns graph.
 */
public void addMagicWheel(int r, int c, int label, int i) {
    this.edgesJohn.add(new Edge(r * this.columns + c, this.magicWheels[i],
        label));
}


private boolean updateLengths(int[] len) {
    boolean changes = false;

    for (Edge e : this.edgesJohn) {

        int tail = e.getTail();
        int head = e.getHead();

        if (len[tail] < INF) {
            int newLen = len[tail] + e.getLabel();
            if (newLen < len[head]) {
                len[head] = newLen;
                changes = true;
            }
        }
    }
    return changes;
}

private int bellmanFord(int origin) throws NegativeWeightCycleException {
    int[] length = new int[this.numNodes];

    for (int i = 0; i < this.numNodes; i++)
        length[i] = INF;

    length[origin] = 0;

    boolean changes = false;

    for (int i = 0; i < this.numNodes; i++) {
        changes = this.updateLengths(length);
        if (!changes) break;
    }

    // Negative-weight cycles detection
    if (changes && this.updateLengths(length))
        throw new NegativeWeightCycleException();

    return length[this.exitNode];
}
```

```
    private void exploreNode(int source, boolean[] selected, int[] length,
        PriorityQueue<SimpleEntry<Integer, Integer>> connected) {
        for (Edge e : this.edgesKate[source]) {
            int node = e.getHead();
            if (!selected[node]) {
                int newLength = length[source] + e.getLabel();
                if (newLength < length[node]) {
                    boolean nodeIsInQueue = length[node] < INF;
                    SimpleEntry<Integer, Integer> oldPair = new
                        SimpleEntry<>(length[node], node);
                    length[node] = newLength;
                    if (nodeIsInQueue) {
                        // This will emulate a decreaseKey
                        connected.remove(oldPair);
                    }
                    connected.add(new SimpleEntry<>(newLength, node));
                }
            }
        }
    }

    private int dijkstra(int origin) {
        boolean[] selected = new boolean[this.numNodes];
        int[] length = new int[this.numNodes];
        PriorityQueue<SimpleEntry<Integer, Integer>> connected = new
            PriorityQueue<>(this.numNodes, new EntryComparator());
        for (int v = 0; v < this.numNodes; v++) {
            selected[v] = false;
            length[v] = INF;
        }
        length[origin] = 0;
        connected.add(new SimpleEntry<>(0, origin));
        do {
            int node = connected.remove().getValue();
            selected[node] = true;
            exploreNode(node, selected, length, connected);
        }
        while (!connected.isEmpty());
        return length[this.exitNode];
    }

    // Solves the problem for John using Bellman Ford's algorithm.
    public int solveJohn(int r, int c) throws NegativeWeightCycleException {
        return this.bellmanFord(r * this.columns + c);
    }

    // Solves the problem for Kate using Dijkstra's algorithm.
    public int solveKate(int r, int c) {
        return this.dijkstra(r * this.columns + c);
    }
}
```

## Edge.java

```
public class Edge {

    private final int head;
    private final int tail;
    private final int label;

    public Edge(int head, int tail, int label) {
        this.head = head;
        this.tail = tail;
        this.label = label;
    }

    public int getHead() { return this.head; }

    public int getTail() { return this.tail; }

    public int getLabel() { return this.label; }
}
```

## EntryComparator.java

```
import java.util.AbstractMap.SimpleEntry;
import java.util.Comparator;
class EntryComparator implements Comparator<SimpleEntry<Integer, Integer>> {

    public int compare(SimpleEntry<Integer, Integer> p1, SimpleEntry<Integer,
        Integer> p2) {
        return p1.getKey() - p2.getKey();
    }
}
```

## NegativeWeightCycleException.java

```
public class NegativeWeightCycleException extends Exception {

    public NegativeWeightCycleException() {
        super();
    }

    public NegativeWeightCycleException(String msg) {
        super(msg);
    }
}
```