ADA
Project 2 Report
Legionellosis

Jacinta Sousa 55075
João Bordalo 55697

May 2021

Introduction

S: Number of sick people ($1 \leq S \leq 20$)
L: Number of Locations ($S \leq L \leq 10000$)
C: Number of Connections ($L - 1 \leq C \leq 17500$)

L will correspond to the number of vertices and C the number of edges of the graph.

Temporal Complexity

Initializing nodeAdjacencies array of lists: $\Theta(L)$
bfsExplore function: $O(C + L)$
Adding all the interviews: $O(S * (C + L))$
Attaining the solution: $O(L)$
$= O(L + S * (C + L) + L) \equiv O(S * (C + L))$

Spacial Complexity

∗ **perilousLocations**
An array of size L, saving integers that will represent the number of sick people that have passed by the i^{th} location.
$= \Theta(L)$

∗ **nodeAdjacencies**
An array of size L, saving lists that will contain the adjacencies of said location, nodeAdjacencies[i] will have a list of locations reachable by paths of length 1 from the i^{th} location.
$= O(L + C)$

∗ **bfsExplore function**
waiting - A queue that will save the locations (Integers) to be processed later.
$= \Theta(L)$
found - An array of booleans that will save which locations have been processed thus far.
$= \Theta(L)$

∗ **solution function**
sol - The list of perilous locations (represented as integers).
$= O(L)$

$= O(L + (L + C) + L + L) \equiv O(L + C)$

Conclusion

∗ **Weak Aspects**:
In solution, one has to run through the whole array to find the perilous locations.

∗ **Alternatives Studied**:
Depth control was made using dummy nodes in the first version. That solution led to a bigger queue size, increasing the memory used by the data structure.

∗ **Improvements**:
We thought about having two HashMaps, one would always have the previous intersection while the other would receive the interception of the new locations by the sick person and the first map, meaning the locations the sick person visited which were already on the first map. After concluding the addition of said sick person, one would set the first map to be equal to the second one. With these structures the used array of potential perilous locations (containing all the locations) wouldn't be necessary so one wouldn't have to iterate through it to arrive at a solution, only iterating though the actual perilous locations to return them at the end. When the number of perilous locations is close to the number of total locations, L, this alternative would take nearly double the memory.

Code Annex

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));

        String[] tokens = input.readLine().split(" ");
        int locations = Integer.parseInt(tokens[0]);
        int connections = Integer.parseInt(tokens[1]);

        Legionellosis legionellosis = new Legionellosis(locations);

        for (int i = 0; i < connections; i++) {
            tokens = input.readLine().split(" ");
            int l1 = Integer.parseInt(tokens[0]);
            int l2 = Integer.parseInt(tokens[1]);
            legionellosis.addConnection(l1, l2);
        }

        int sickPeople = Integer.parseInt(input.readLine());
        legionellosis.setSickPeople(sickPeople);

        for (int i = 0; i < sickPeople; i++) {
            tokens = input.readLine().split(" ");
            int h = Integer.parseInt(tokens[0]);
            int d = Integer.parseInt(tokens[1]);
            legionellosis.addInterview(h, d);
        }

        input.close();

        System.out.println(legionellosis.solution());
    }
}
```

Legionellosis.java

```java
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class Legionellosis {

    private final int numNodes;
    private final List<Integer>[] nodeAdjacencies;
    private final int[] perilousLocations;
    private int sickPeople;

    @SuppressWarnings("unchecked")
    public Legionellosis(int locations) {
        this.numNodes = locations;
        this.nodeAdjacencies = new List[this.numNodes];
        this.perilousLocations = new int[this.numNodes];
        this.sickPeople = 0;

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

    public void addConnection(int l1, int l2) {
        // l1 -> l2
        this.nodeAdjacencies[l1 - 1].add(l2 - 1);
        // l2 -> l1
        this.nodeAdjacencies[l2 - 1].add(l1 - 1);
    }

    public void addInterview(int home, int distance) {
        this.bfsExplore(home - 1, distance);
    }

    public void setSickPeople(int sickPeople) {
        this.sickPeople = sickPeople;
    }
```

```
    private void bfsExplore(int root, int maxDepth) {
        Queue<Integer> waiting = new LinkedList<>();
        boolean[] found = new boolean[this.numNodes];
        // Depth control
        int depth = 0;
        int currentLevelNodes = 1;
        int nextLevelNodes = 0;
        // Treat the root
        waiting.add(root);
        found[root] = true;
        this.perilousLocations[root]++;
        do {
            // Pop a node
            int node = waiting.remove();
            currentLevelNodes--;

            // Explore its adjacencies
            for (Integer neighbour : this.nodeAdjacencies[node]) {
                if (!found[neighbour]) {
                    this.perilousLocations[neighbour]++;
                    nextLevelNodes++;
                    waiting.add(neighbour);
                    found[neighbour] = true;
                }
            }
            // Depth control
            if (currentLevelNodes == 0) {
                depth += 1;
                currentLevelNodes = nextLevelNodes;
                nextLevelNodes = 0;
            }
        }
        while (!waiting.isEmpty() && depth < maxDepth);
    }

    public String solution() {
        StringBuilder sol = new StringBuilder();
        for (int i = 0; i < this.numNodes; i++) {
            if (this.perilousLocations[i] == this.sickPeople) {
                sol.append(i + 1);
                sol.append(" ");
            }
        }
        if (sol.length() == 0) {
            sol.append(0);
        } else {
            // Removes trailing white space
            sol.setLength(sol.length() - 1);
        }
        return sol.toString();
    }
}
```