CSE 246
Analysis of Algorithms
Homework 1 Part 1
Report

Ertuğrul Sağdıç – 150116061

Objective

The purpose of this project is, to determine whether the laptops on the given location can communicate with each other wireless, and more importantly, calculate how many intermediate router laptops that they need, to communicate from laptop i, to laptop j, by using their wireless transmission ranges.

To calculate intermediate router laptops, we need to have BFS based algorithm, in order to find out reachable laptops from first laptop to other laptops.

a) Code and Details of the algorithm

First of all, we need to take the graph, source laptop, and destination laptop as parameters of the function. Also, the function needs to return the path as array list of laptops.

```
public static ArrayList<Laptop> pathFinder(LocationGraph locationGraph, Laptop
sourceLaptop, Laptop destinationLaptop)
```

Than, we need to create array list to hold the path, from source laptop to destination laptop. Array list type must be Laptop as we store laptops in it.

```
//array to store current path
ArrayList<Laptop> path = new ArrayList<>();
```

There can be more than one path from source laptop to destination laptop. In order to store these paths we need to create a Queue, which holds the array list of the laptops.

```
//queue which stores paths
Queue<ArrayList<Laptop>> pathQueue = new LinkedList<>();
```

We also need to have an boolean array to keep track of the data which shows us, if the laptops that we are going to search for, are visited or not.

```
//visit array to keep if the laptop visited or not
boolean[] visited = new boolean[locationGraph.getMaxNumberOfLaptops()];
```

These are all the things that we need to create. In order to start, we need to add source laptop into path array list. Than, we need to add this path to the

queue. Also, we need to change the value of the visited array as true with the corresponding value of the laptop.

```
//add souceLaptop into the path
path.add(sourceLaptop);
pathQueue.add(path);
visited[sourceLaptop.getValue()] = true;
```

After that, we need to create a while loop which will loop forever as long as the queue is not empty.

```
//loop until queue become empty
while (pathQueue.size() != 0)
```

Inside of the loop, we need to assign path with the element in the queue. Also, we need to assign the last element in the path as current laptop that we will work with.

```
path = pathQueue.remove();
Laptop currentLaptop = path.get(path.size() - 1);
```

If the current laptop is equal to destination laptop, we need to return the path as we reached to destination.

```
//if the current laptop equals to destination laptop return the path
if (currentLaptop == destinationLaptop) {
    return path;
}
```

If not, we need to traverse all the laptops in the graph that can be reachead from the current laptop. If one of the laptops can be reachable, we need to update the path with adding the reached laptop into the new array list of laptops. Also,, we need to add the new path into the queue that we hold paths. Also, we need to update the reached laptops visited data with true as we reached that laptop.

```
else {
    //traverse to all the laptops that can be reached from the current laptop
    for (int i = 0; i < locationGraph.getMaxNumberOfLaptops(); i++) {
        if
((locationGraph.getDistances()[currentLaptop.getValue()][i].getDistance() > 0) &&
(!visited[i])) {
            ArrayList<Laptop> newPath = new ArrayList<>(path);
            newPath.add(locationGraph.getLaptops()[i]);
            //push new path to queue
            pathQueue.add(newPath);
            visited[i] = true;
```

```
        }
    }
}
```

The travesing laptops and the checking if we reach the destination laptop should be looped until we get all the paths from source to destination.

After we finish the loop, we check the path again in order to catch the possibility of source laptop would not reach to any destination laptop from source. If we can not reach the destination laptop, than we should remove existing path until source laptop because the path will be unvalid.

```
//if the node is unreachable from source to destination remmove the laptos from
path
if(path.get(path.size() - 1) != destinationLaptop){
    while(path.get(path.size() - 1) != sourceLaptop){
        path.remove(path.size() - 1);
    }
}
```

Last of all, we return the path.

```
return path;
```

The hop distance will be equal to the path that we return minus one.

```
//writes the outputs to the output file
outputFile.write(Integer.toString(path.size() - 1) + "\n");
```

Outputs:

test1output.txt

```
0
1
2
```

test2output.txt

```
0
2
1
```

test3output.txt

```
0
1
3
2
2
```

test4output.txt

```
0
0
0
0
```

As seen, the outputs are correct with the given inputs.

test5output.txt

```
0
1
2
3
3
4
0
0
4
```

test6output.txt

```
0
1
2
3
4
2
3
3
5
4
5
6
7
0
8
```

b)Time and Space complexity of the algorithm

Time Complexity:

Time complexity estimates the time to run an algorithm. In order to calculate time complexity, we need to check the core of the algorithm, and determine the input size and basic operation of the algorithm.

```
//traverse to all the laptops that can be reached from the current laptop
for (int i = 0; i < locationGraph.getMaxNumberOfLaptops(); i++) {
    if ((locationGraph.getDistances()[currentLaptop.getValue()][i].getDistance() >
0) && (!visited[i])) {
        ArrayList<Laptop> newPath = new ArrayList<>(path);
        newPath.add(locationGraph.getLaptops()[i]);
        //push new path to queue
        pathQueue.add(newPath);
        visited[i] = true;
    }
}
```

Here, we traverse all the laptops inside of the graph. The input size is number of laptops. Lets take the number of laptops as n. So, input size is equal to n.

The basic operation, which the algorithm performs repeatedly is comparison. We compare all the nodes with the current laptop if there are any connection between them. Also, we compare the number of laptops is visited or not. So, we have two comparison as a basic operatiob

So, we can say that;

$$T(n) = \sum_{i=0}^{n-1}(2) = 2n \ \in \text{O(n)}$$

Space Complexity:

Space complexity is the amount of memory used by the algorithm to execute and produce the result. In order to calculate space complexity of our algorithm, lets have a look at the variables and objects that we use in the algorihm.

We have an array list of laptops:

```
        ArrayList<Laptop> path = new ArrayList<>();
```

Queue which holds array list of laptops:

```
Queue<ArrayList<Laptop>> pathQueue = new LinkedList<>();
```

Also, we have a boolean array:

```
boolean[] visited = new boolean[locationGraph.getMaxNumberOfLaptops()];
```

For n laptos, we need 1*n memory.

In Laptop class, we have for variables:

```
//Value of the laptop
private int value;
// x coordinate of the laptop
private double x;
// y coordinate of the laptop
private double y;
//wireless range of the laptop
private double wirelessTransmissionRange;
```

For integer value size is 4 bytes, for double values size is 8 bytes. So, total is 4+3*8 = 28 bytes for creating Laptop object.

Fort he worst case, which is visiting all the laptops. So, we need to hold all vertices in the array list. Assuming that we have n laptops, the total memory that the path array need to hold is 28*n.

We can say that the total memory requirement will be (n + 28n) = 29n. So, the space complexity will be O(n).

References:

https://books.google.com.tr/books?id=DuQxX8GVtVgC&printsec=frontcover&hl=tr&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

https://stackoverflow.com/questions/57534074/how-to-represent-an-undirected-weighted-graph-in-java

https://yourbasic.org/algorithms/time-complexity-arrays/

https://www.studytonight.com/data-structures/space-complexity-of-algorithms

https://stackoverflow.com/questions/9844193/what-is-the-time-and-space-complexity-of-a-breadth-first-and-depth-first-tree-tr