Breadth-First Search

28th March 2020

OVERVIEW

Computing the hop distance in a graph that consists of laptops with wireless transmitters using BFS.

GOALS

1. Creating a network of laptops
2. Implementing BFS algorithm to compute the hope distance from one laptop to another with 100% accuracy.

All goals have been reached.

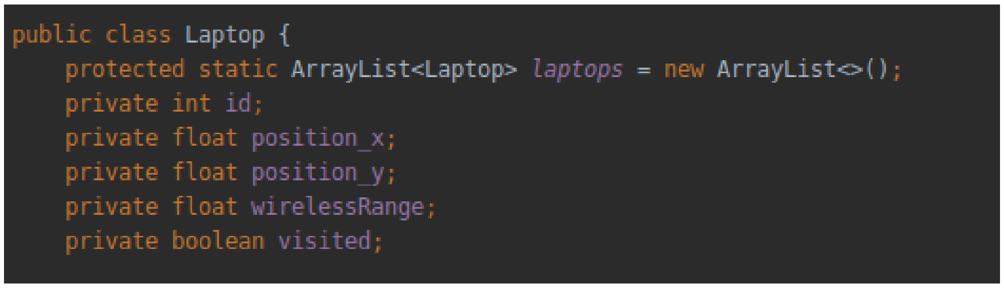
MILESTONES

Algorithm and Implementation

Main Class has been used to parse command-line arguments, creating an instance of the input handler class to create Laptop objects and add them to the static 'laptops' array list to make them easy to use.

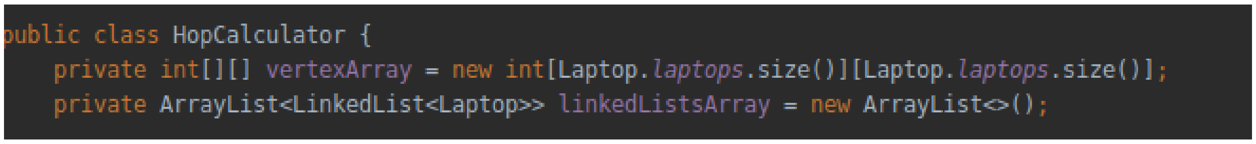
InputHandler Class simply reads the text file with a scanner object and adds all the rows to an array list to create laptops.

Laptop Class is the node class for the graph. It contains the important fields to be used in BFS and an array list of the laptops which comes in handy while BFS.



```java
public class Laptop {
    protected static ArrayList<Laptop> laptops = new ArrayList<>();
    private int id;
    private float position_x;
    private float position_y;
    private float wirelessRange;
    private boolean visited;
```

HopCalculator Class has the implementation of a breadth-first search algorithm. It uses the vertex array, an adjacency matrix and an ArrayList linked lists named linkedListArray.



```
public class HopCalculator {
    private int[][] vertexArray = new int[Laptop.laptops.size()][Laptop.laptops.size()];
    private ArrayList<LinkedList<Laptop>> linkedListsArray = new ArrayList<>();
```

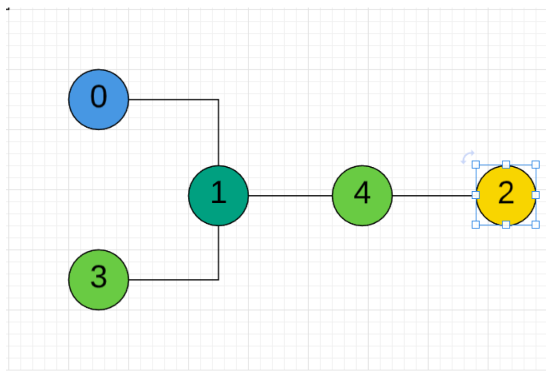
Purpose of the linkedListArray is to hold the laptops in this manner if hop distance from laptop 0 to 4 is being calculated:

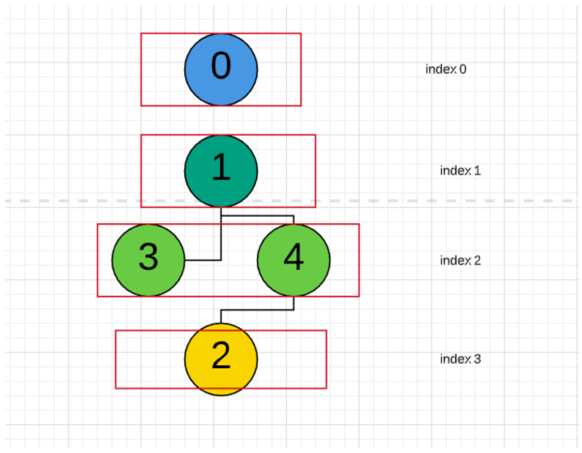
linkedListArray.get(0) -------------> will give the laptop with id 0

linkedListArray.get(1) -------------> will give the laptop with id 1

linkedListArray.get(2) -------------> will give the laptop with id 3 and 4

linkedListArray.get(0) -------------> will give the laptop with id 2





If we are trying to compute the hope distance from 0 to 2, what we need is to sort the nodes in the order below. Red boxes have been named as stages in the code so to speak, we have to divide the graph into stages. So the code will search as:

1-) Does 2 exist in the neighbours of the 0 which is stage 1 (only number 1).

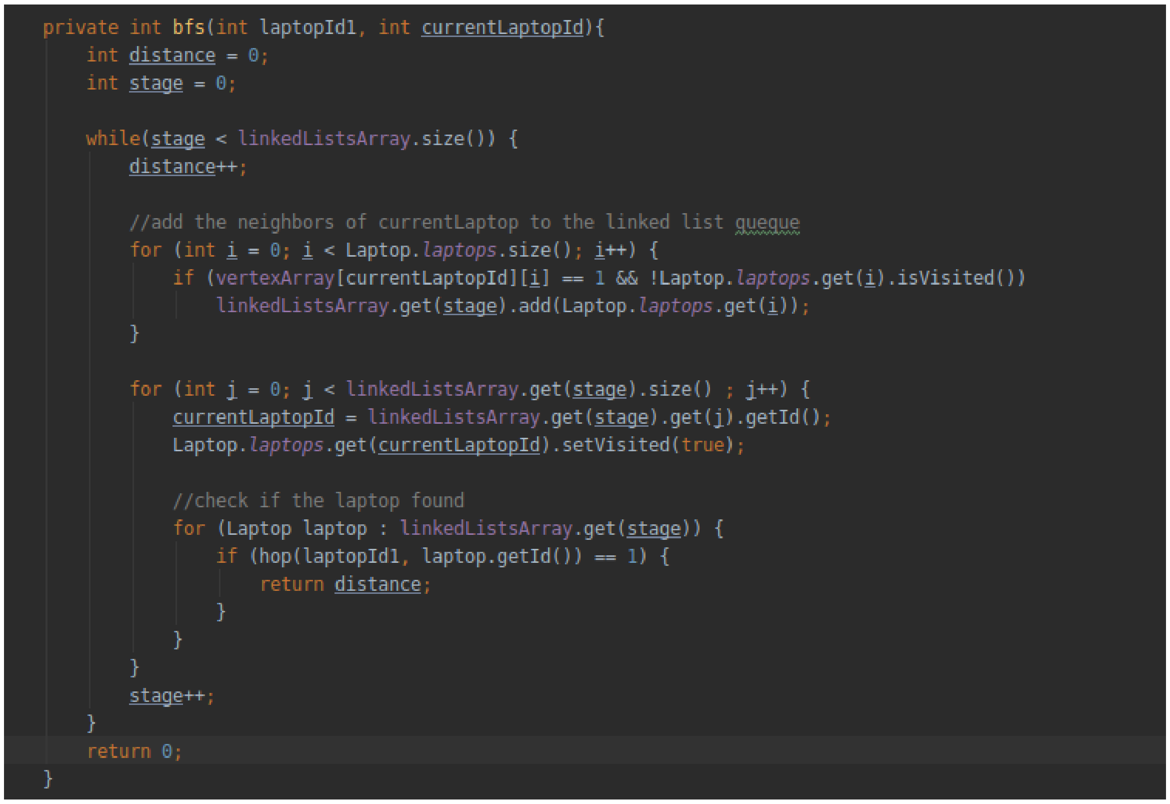
Stage 1 will be generated.

2-) Does 2 exists in the neighbours of stage 1 (only number 1) which is stage 2 (3 and 4)

Stage 2 will be generated

3-) Does 2 exists in the neighbours of stage 2 (3 and 4) which is stage 2 (only number 2)

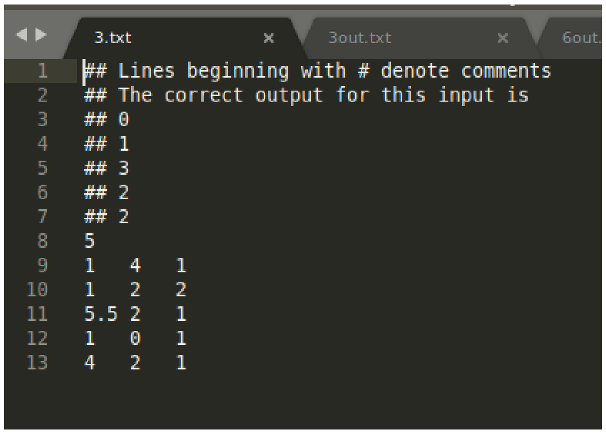
The algorithm mentioned above has been implemented as the following function.



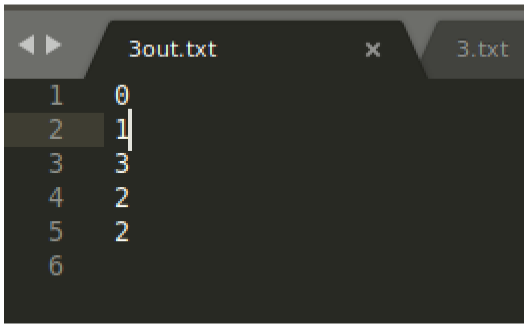
```java
private int bfs(int laptopId1, int currentLaptopId){
    int distance = 0;
    int stage = 0;

    while(stage < linkedListsArray.size()) {
        distance++;

        //add the neighbors of currentLaptop to the linked list queque
        for (int i = 0; i < Laptop.laptops.size(); i++) {
            if (vertexArray[currentLaptopId][i] == 1 && !Laptop.laptops.get(i).isVisited())
                linkedListsArray.get(stage).add(Laptop.laptops.get(i));
        }

        for (int j = 0; j < linkedListsArray.get(stage).size() ; j++) {
            currentLaptopId = linkedListsArray.get(stage).get(j).getId();
            Laptop.laptops.get(currentLaptopId).setVisited(true);

            //check if the laptop found
            for (Laptop laptop : linkedListsArray.get(stage)) {
                if (hop(laptopId1, laptop.getId()) == 1) {
                    return distance;
                }
            }
        }
        stage++;
    }
    return 0;
}
```

Accuracy

The algorithm worked perfectly on all four of the given inputs in the classroom. When the argument “X.txt” is given, Its output will be “Xout.txt”.



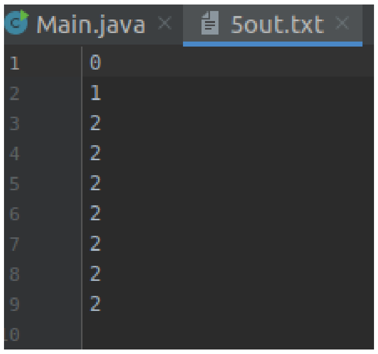
```
## Lines beginning with # denote comments
## The correct output for this input is
## 0
## 1
## 3
## 2
## 2
5
1   4   1
1   2   2
5.5 2   1
1   0   1
4   2   1
```



```
0
1
3
2
2
```

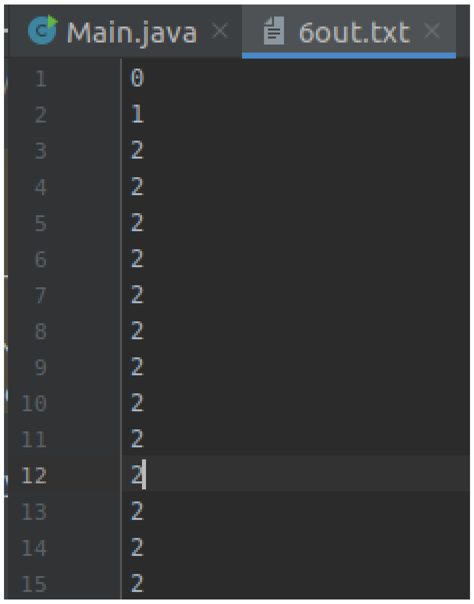
Testing with New Inputs

```
9
1      1      1
2      2      1
4      1.5    1.5
4      3      1
6      3      2
8      4      2
2      7      2
4      9      1
6.5    5.5    1.5
```



```
0
1
2
2
2
2
2
2
2

```

```
15
1      1      1
2      2      1
3      3      1
4      4      1
5.2    5.2    1
3.2    1      1
4.5    1.2    1
4.5    3      1
7      4.5    1
6      1      1
7      2      1
8      3      1
9.5    3      1
9      1      1
11     3      1
```



```
0
1
2
2
2
2
2
2
2
2
2
2
2
2
2
```