GTU DEPARTMENT of COMPUTER ENGINEERING

CSE222/505 – SPRING 2023

HOMEWORK 8 REPORT

ÖMER FARUK ÇOLAKEL

200104004043

1. SYSTEM REQUIREMENTS

```
public static void Test(String input) {
```

Test method in main class requires name of the input file. It creates output directories, creates the map, graph, BFS and Dijkstra objects, draws the paths on the map, prints path planning time in nanoseconds and writes the outputs to files.

Example usage:

```
Test( input: "map01");
```

```
public CSE222BFS(CSE222Graph graph) {
```

Constructor with graph parameter for BFS. Graph is needed to get start and end points at first. Throws exception if any of them is invalid. Then, I used it to get width and height of the map. Also, I got neighbours (adjacent vertices) of the current vertex with it.

```
private void findPath(String[][] parent) {
```

Method to get the path from star to end. Requires parents of the end vertex. Adds them to an array and reverses it to store the path in the path ArrayList.

```
public BufferedImage drawPath(BufferedImage image, String filename) {
```

Method to draw path on a PNG file with the name of "filename". Requires the map that was created by CSE222Map. CSE222BFS and CSE222Dijkstra has this method.

```
public CSE222Dijkstra(CSE222Graph graph)
```

Constructor with graph parameter for BFS. Graph is needed to get start and end points at first. Throws exception if any of them is invalid. Then, I used it to get width and height of the map. Also, I got neighbours (adjacent vertices) of the current vertex with it.

```
public ArrayList<int[]> getNeighbors(int[] current)
```

This method is used to get adjacent nodes to the current node. Current should be an int array with 2 elements.

```
public Edge(int x, int y) {
```

This constructor is used to set x and y coordinates of an edge.

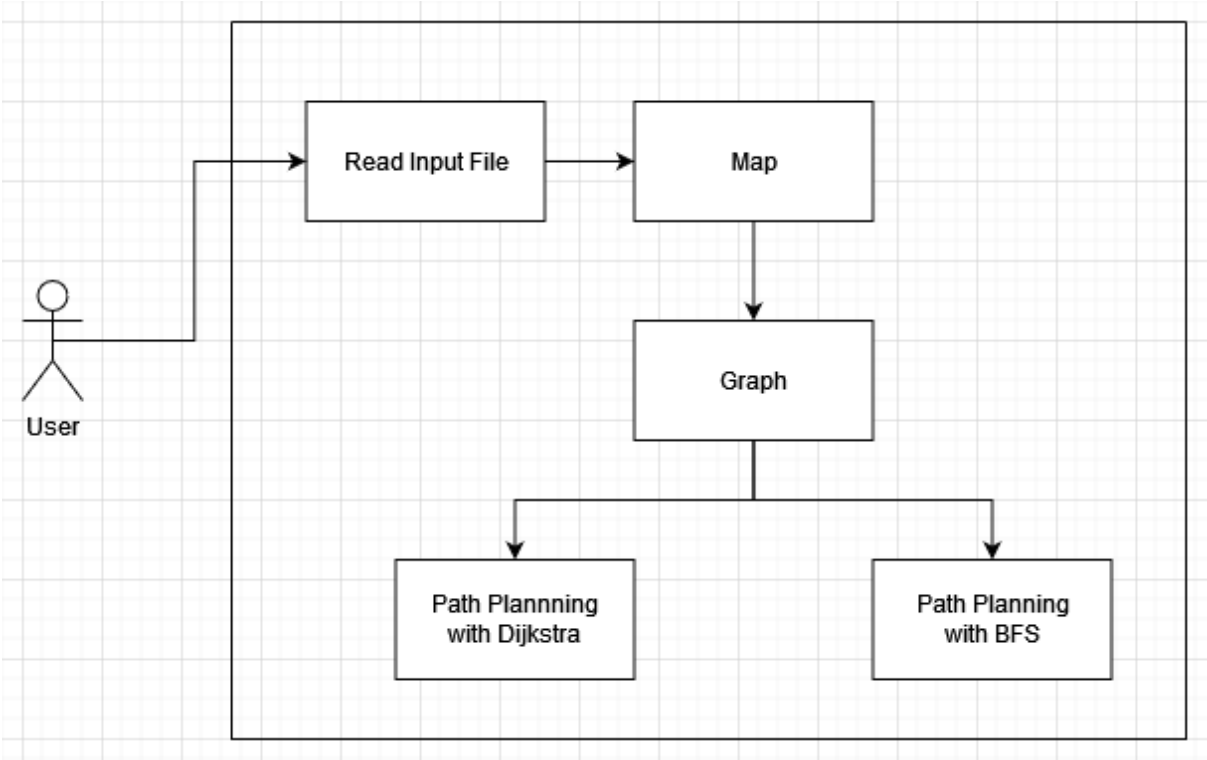
```
public CSE222Graph(CSE222Map map)
```

This constructor requires a map. Takes every element in the map and checks if it is one or zero. If it's then checks if adjacent nodes are 0 or not. If current node is not in the graph adds it to graph as key and every adjacent node as value.

```
public void addEdge(String source, ArrayList<Edge> edges)
```

Previous constructor calls this method to add a new node and adjacent nodes to graph.

2. USE CASE DIAGRAM



3. PROBLEM SOLUTION APPROACH

Purpose of the Test method in main class is create directories for outputs, read input files, instantiate map, graph, BFS and Dijkstra objects, calculate their runtimes and convert the output as PNG files/

Map objects holds start and end points of the map and coordinates of every element's position as string and their value. It also holds height and width of the map. "convertPNG" method creates the map as PNG. It checks if the start or end point is 1.

Graph objects reads maps. Keeps the coordinates of every 0 in the map as key. It also checks adjacent 0 nodes and adds their coordinates as value. It also holds the start and end coordinates; height and width of the map. "getNeighbors" method takes coordinates and checks if it exists. If it is, then gets all adjacent nodes coordinates and returns it in ArrayList. This class has an inner class called Edge. It holds the x and y coordinates of an adjacent node. To add a node as adjacent, it should be next to current node or cross to it. I added the current node as string because it's easier to search for map. Using an integer would be problematic since they are harder t compare. I override toString method to print it to a txt file or to the terminal.

BFS is a traversal algorithm to find the shortest path between two coordinates. The first path that was found is always the shortest. We start with pushing the start node to a queue. Then we pick that node, get all the adjacent nodes and push them to queue. Poll a node from queue, take its adjacent nodes to queue and repeat until it finds the end node. But there is possibility of no path from start to end. Also, we shouldn't revisit the nodes or else it might get stuck in a loop. For that reason, I initialized a 2D boolean array. At the start every element is false, meaning they are not visited. After we get all the adjacent nodes, we check if we already visited it. If not adds them to queue and sets the array true according to it's coordinates. In addition to these, we need to hold the path and for that I used a String array that holds the data of visited nodes. Then I found the path from end to start, reversed it and set the path arraylist.

Dijkstra's algorithm is also a graph traversal algorithm. It is very close to BFS in terms of how it works. Every element in distances array has maximum integer at start. It changes after visiting the array and equalizing it with +1 of the current node. After that, starting from end, I added every node with -1 distance compared to current distance. Then I reversed and added them to path.

4. TEST CASES AND RESULTS

Format of Inputs

```
Test( input: "map01");
Test( input: "map02");
Test( input: "map03");
Test( input: "map04");
Test( input: "map05");
Test( input: "map06");
Test( input: "map07");
Test( input: "map08");
Test( input: "map09");
Test( input: "map10");
Test( input: "map11");
Test( input: "pisa");
Test( input: "tokyo");
Test( input: "triumph");
Test( input: "vatican");
```

Terminal Outputs

```
Testing map01.txt
Graph created in 341531000ns
BFS Algorithm
Path found!
Distance: 991
BFS created in 188346200ns
Djikstra's Algorithm
Path found
Distance: 991
Dijkstra created in 49860400ns
```

```
Testing map02.txt
Graph created in 132777100ns
BFS Algorithm
Path found!
Distance: 666
BFS created in 88098800ns
Djikstra's Algorithm
Path found
Distance: 666
Dijkstra created in 30303300ns
```

```
Testing map03.txt
Graph created in 145879100ns
BFS Algorithm
Path found!
Distance: 760
BFS created in 43691900ns
Djikstra's Algorithm
Path found
Distance: 760
Dijkstra created in 65220600ns
```

```
Testing map04.txt
Graph created in 100417900ns
BFS Algorithm
Path found!
Distance: 673
BFS created in 94590300ns
Djikstra's Algorithm
Path found
Distance: 673
Dijkstra created in 32758100ns
```

```
Testing map05.txt
Graph created in 71215000ns
BFS Algorithm
Path found!
Distance: 599
BFS created in 67889400ns
Djikstra's Algorithm
Path found
Distance: 599
Dijkstra created in 38558000ns
```

```
Testing map06.txt
Graph created in 77020900ns
BFS Algorithm
Path found!
Distance: 506
BFS created in 31347500ns
Djikstra's Algorithm
Path found
Distance: 506
Dijkstra created in 30653900ns
```

```
Testing map07.txt
Graph created in 132954700ns
BFS Algorithm
Path found!
Distance: 709
BFS created in 60486100ns
Djikstra's Algorithm
Path found
Distance: 709
Dijkstra created in 53809300ns
```

```
Testing map08.txt
Graph created in 175643900ns
BFS Algorithm
Path found!
Distance: 640
BFS created in 42592700ns
Djikstra's Algorithm
Path found
Distance: 640
Dijkstra created in 36720900ns
```

```
Testing map09.txt
Graph created in 147941500ns
BFS Algorithm
Path found!
Distance: 957
BFS created in 34976700ns
Djikstra's Algorithm
Path found
Distance: 957
Dijkstra created in 30528200ns
```

```
Testing map11.txt
Error: TextFiles\map11.txt (The system cannot find the file specified)
```

```
Testing map10.txt
Graph created in 92139100ns
BFS Algorithm
Path found!
Distance: 478
BFS created in 21326200ns
Djikstra's Algorithm
Path found
Distance: 478
Dijkstra created in 30101600ns
```

```
Testing triumph.txt
Graph created in 406956500ns
BFS Algorithm
Path found!
Distance: 1059
BFS created in 87768200ns
Djikstra's Algorithm
Path found
Distance: 1059
Dijkstra created in 73145200ns
```

```
Testing vatican.txt
Graph created in 461562800ns
BFS Algorithm
Path found!
Distance: 1412
BFS created in 121964400ns
Djikstra's Algorithm
Path found
Distance: 1412
Dijkstra created in 115559500ns
```

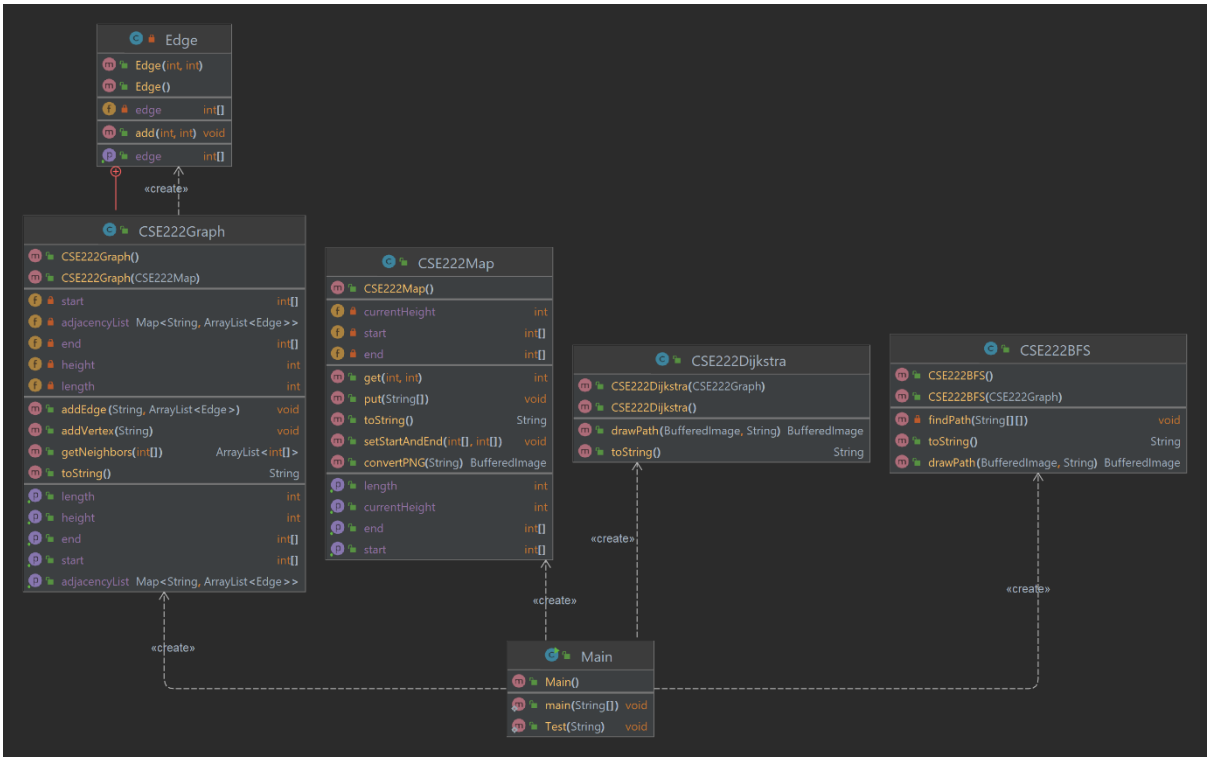
```
Testing pisa.txt
Graph created in 308943300ns
BFS Algorithm
Path found!
Distance: 1642
BFS created in 80270700ns
Djikstra's Algorithm
Path found
Distance: 1642
Dijkstra created in 71447600ns
```

```
Testing tokyo.txt
Graph created in 493630800ns
BFS Algorithm
Path found!
Distance: 890
BFS created in 85265800ns
Djikstra's Algorithm
Path found
Distance: 890
Dijkstra created in 104114100ns
```

File Outputs

You can find PNG files in the src/PNGs as “Map_INPUTNAME.png”, “BFS_INPUTNAME.png” and “Dijkstra_INPUTNAME.png”. Txt files can be found in the folder Outputs/INPUTNAME folders. Outputs folder is next to src folder. “map_INPUTNAME.txt” has start point (x,y) in the first line, end point (x,y) in the second line and the whole map under them. “graph_INPUTNAME.txt” has all vertices and edges. “bfs_INPUTNAME.txt” has a feasible route between start and end. First line is start and last line is end. “Dijkstra_INPUTNAME.txt” is same with BFS file.

5. CLASS DIAGRAM



6. COMLEXITY ANALYSIS AND RUNTIMES

Runtimes (in nanoseconds):

Input Name/Class	Graph	Dijkstra	BFS
Map01	341530000	49860400	188346200
Map02	132777100	30303300	31347500
Map03	145879100	65220600	43691900
Map04	100417900	32758100	94590300
Map05	71215000	38558000	67889400
Map06	77020900	30653900	31347500
Map07	132954700	53809300	60486100
Map08	175643900	36720900	42592700
Map09	147941500	30528200	34976700
Map10	92139100	30101600	21326200
Map11	-	-	-
Triumph	406956500	73145200	87768200
Vatican	461562800	11559500	121964400
Pisa	308943300	30101600	21326200
Tokyo	493630800	104114100	85265800

Time Complexities:

In BFS algorithm, at worst case we have to traverse through all nodes. So, time complexity changes according to number of vertices and edges. This gives us the result of O(V + E).

In Dijkstra's algorithm at worst case, we have to go through every node to calculate their distances. That gives us the time complexity of O(V+E).