GTU Department of Computer Engineering

CSE 222/505 - Spring 2023

Homework 8 Report

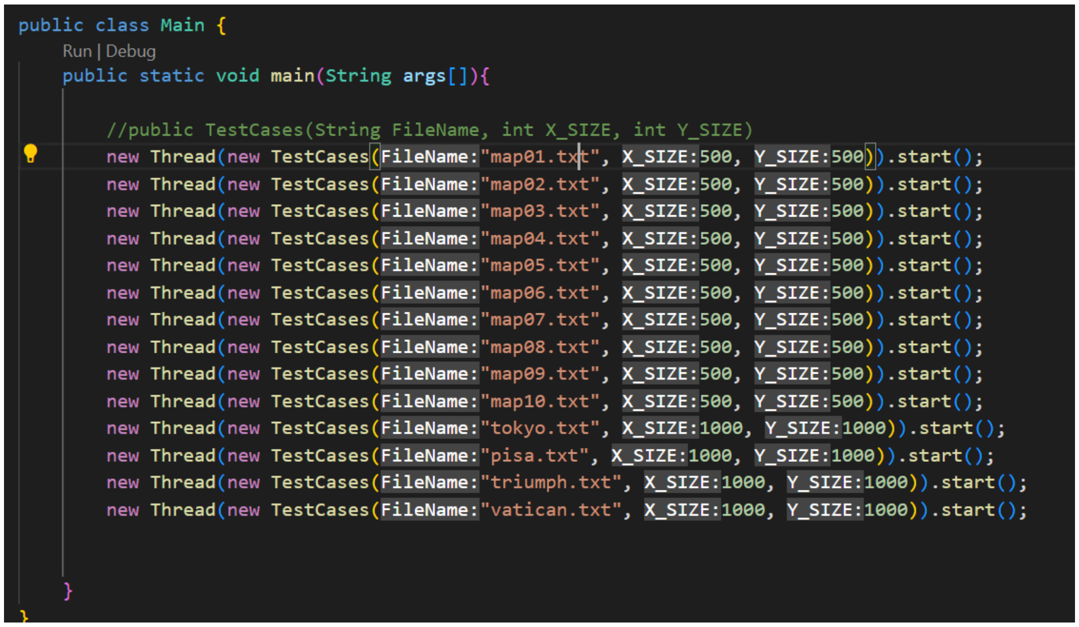
BERRU LAFCI

1901042681

PROBLEM SOLUTION:

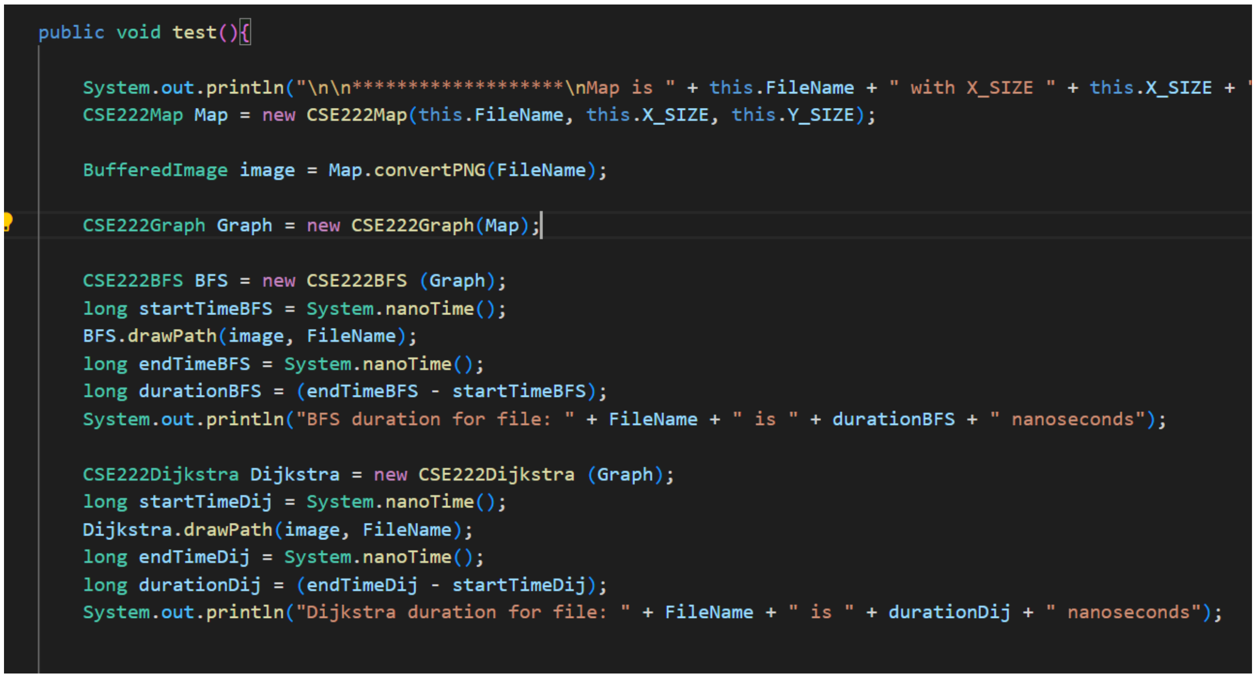
I used your Main.java and TestCases.java and rearranged my main part a little bit. So, my code runs correctly with these main and testcases classes.

(Because of thread, the output may be mixed but the titles which I write can help to understand easily. Also, I explained the output clearly in my report.)



```java
public class Main {
    Run | Debug
    public static void main(String args[]){

        //public TestCases(String FileName, int X_SIZE, int Y_SIZE)
        new Thread(new TestCases(FileName:"map01.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map02.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map03.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map04.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map05.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map06.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map07.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map08.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map09.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"map10.txt", X_SIZE:500, Y_SIZE:500)).start();
        new Thread(new TestCases(FileName:"tokyo.txt", X_SIZE:1000, Y_SIZE:1000)).start();
        new Thread(new TestCases(FileName:"pisa.txt", X_SIZE:1000, Y_SIZE:1000)).start();
        new Thread(new TestCases(FileName:"triumph.txt", X_SIZE:1000, Y_SIZE:1000)).start();
        new Thread(new TestCases(FileName:"vatican.txt", X_SIZE:1000, Y_SIZE:1000)).start();


    }
}
```

I calculated the duration between finding and drawing path.



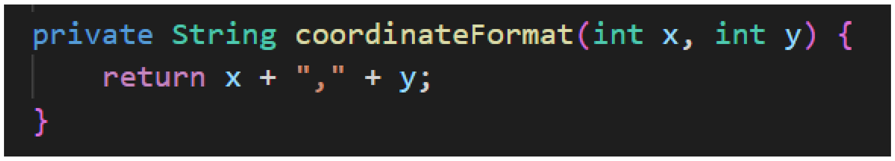
```java
public void test(){

    System.out.println("\n\n******************\nMap is " + this.FileName + " with X_SIZE " + this.X_SIZE + '
    CSE222Map Map = new CSE222Map(this.FileName, this.X_SIZE, this.Y_SIZE);

    BufferedImage image = Map.convertPNG(FileName);

    CSE222Graph Graph = new CSE222Graph(Map);

    CSE222BFS BFS = new CSE222BFS (Graph);
    long startTimeBFS = System.nanoTime();
    BFS.drawPath(image, FileName);
    long endTimeBFS = System.nanoTime();
    long durationBFS = (endTimeBFS - startTimeBFS);
    System.out.println("BFS duration for file: " + FileName + " is " + durationBFS + " nanoseconds");

    CSE222Dijkstra Dijkstra = new CSE222Dijkstra (Graph);
    long startTimeDij = System.nanoTime();
    Dijkstra.drawPath(image, FileName);
    long endTimeDij = System.nanoTime();
    long durationDij = (endTimeDij - startTimeDij);
    System.out.println("Dijkstra duration for file: " + FileName + " is " + durationDij + " nanoseconds");
```

CSE222Map:

For mapping, I used this variable: private Map<String, Integer> map;

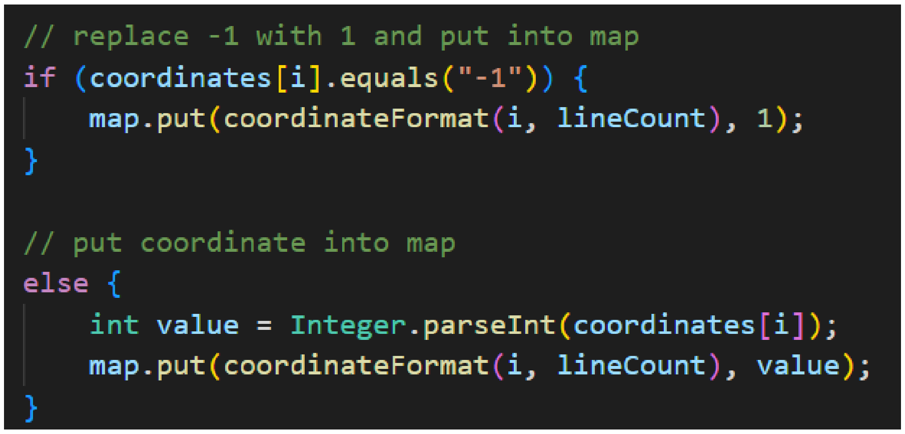
I preferred to hold the coordinates as string in this homework.

I used it to store the map matrix read from the file, where each coordinate is mapped to its corresponding value (0 or 1). The format of the coordinates is "x,y". I used a method for formatting coordinate strings.



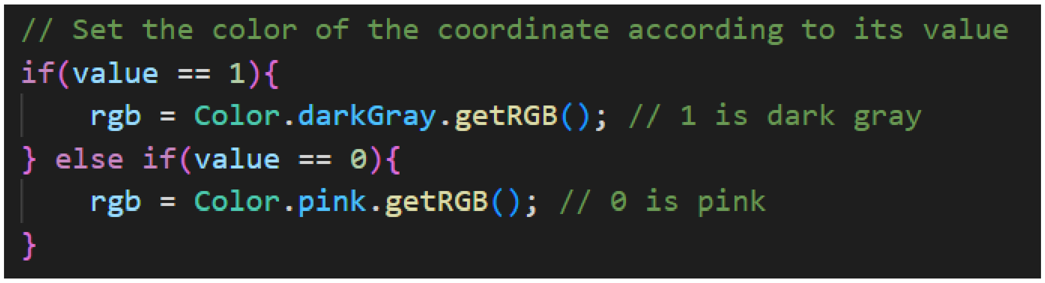
```java
private String coordinateFormat(int x, int y) {
    return x + "," + y;
}
```

For each coordinate in the map matrix, it checks if the value is "-1". If it is, it puts a value of 1 in the map at that coordinate as a wall. Otherwise, it parses the value as an integer and puts it in the map.



```java
// replace -1 with 1 and put into map
if (coordinates[i].equals("-1")) {
    map.put(coordinateFormat(i, lineCount), 1);
}

// put coordinate into map
else {
    int value = Integer.parseInt(coordinates[i]);
    map.put(coordinateFormat(i, lineCount), value);
}
```

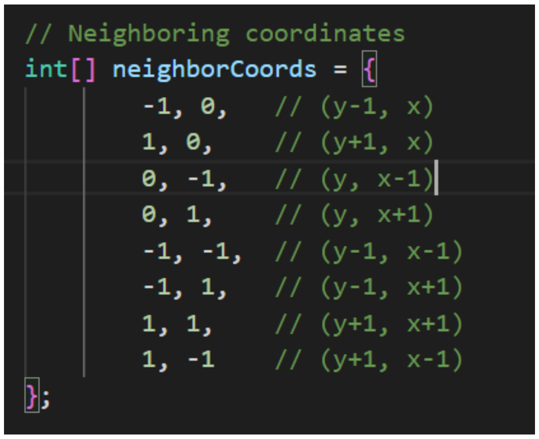
Then I made the PNG according to 0 and 1 values in the map.



```java
// Set the color of the coordinate according to its value
if(value == 1){
    rgb = Color.darkGray.getRGB(); // 1 is dark gray
} else if(value == 0){
    rgb = Color.pink.getRGB(); // 0 is pink
}
```

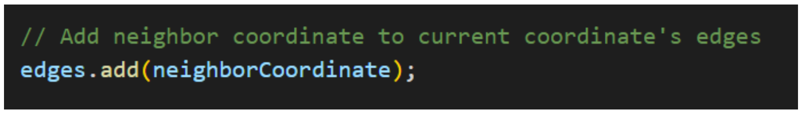
CSE222Graph:

I iterate over each coordinate in the map, I check if the value at that coordinate is zero. If it's zero, I add it as a node to the graph and determine its neighboring coordinates.



```java
// Neighboring coordinates
int[] neighborCoords = {
        -1, 0,   // (y-1, x)
        1, 0,    // (y+1, x)
        0, -1,   // (y, x-1)
        0, 1,    // (y, x+1)
        -1, -1,  // (y-1, x-1)
        -1, 1,   // (y-1, x+1)
        1, 1,    // (y+1, x+1)
        1, -1    // (y+1, x-1)
};
```

I then check if the neighboring coordinates are valid within the map's boundaries. If a neighboring coordinate is valid and has a zero value, I add it as an edge to the current coordinate's list of edges in the graph.

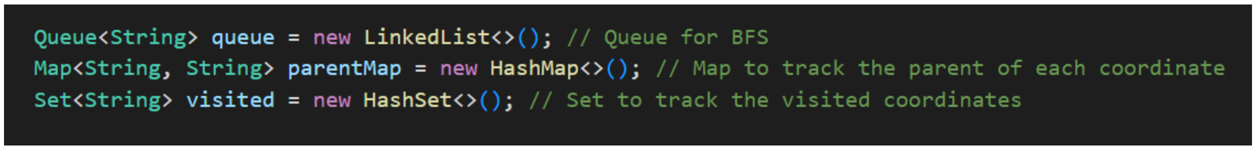


```java
// Add neighbor coordinate to current coordinate's edges
edges.add(neighborCoordinate);
```

Finally, I return the completed graph.

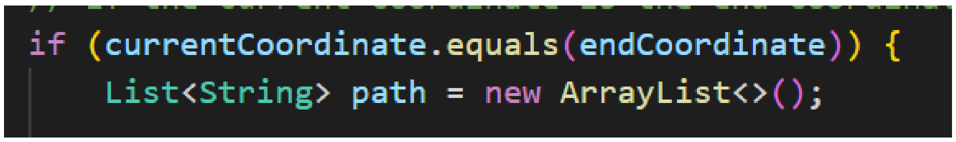
CSE222BFS:

I set up a queue, which will help me perform the BFS traversal. I also created a parent map to keep track of the parent of each coordinate in the path and a set to mark visited coordinates.



```java
Queue<String> queue = new LinkedList<>(); // Queue for BFS
Map<String, String> parentMap = new HashMap<>(); // Map to track the parent of each coordinate
Set<String> visited = new HashSet<>(); // Set to track the visited coordinates
```

I enter a loop that continues until the queue becomes empty. Within each iteration of the loop, I dequeue the next coordinate from the queue. If this coordinate happens to be the end coordinate, it means I have found the path from the start to the end.

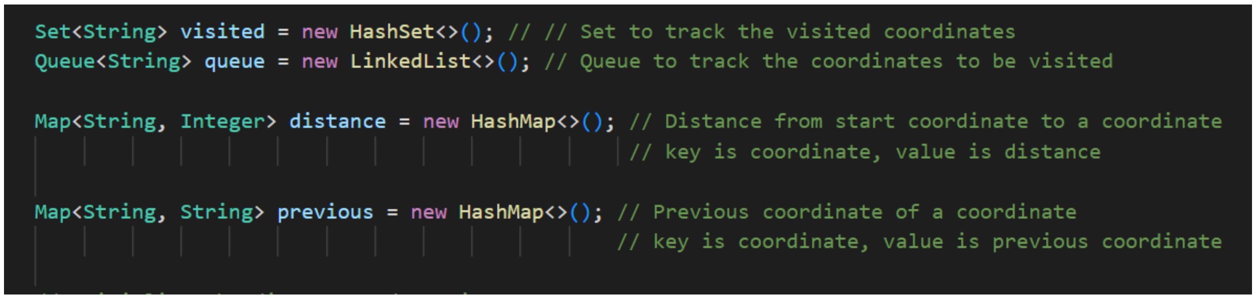


```java
if (currentCoordinate.equals(endCoordinate)) {
    List<String> path = new ArrayList<>();
```

I construct the path by backtracking from the end coordinate to the start coordinate. I use the parent map to trace the shortest path and add each coordinate to the beginning of a list called "path." Once I have added all the coordinates to the path, I include the start coordinate at the beginning and then return the path.

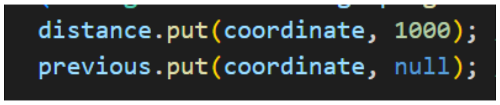
CSE222Dijkstra:

I have a set called "visited" to keep track of the coordinates I have visited, a queue called "queue" to store the coordinates to be visited in a particular order, and two maps called "distance" and "previous" to store the distance from the start coordinate to each coordinate and the previous coordinate in the shortest path, respectively.



```java
Set<String> visited = new HashSet<>(); // // Set to track the visited coordinates
Queue<String> queue = new LinkedList<>(); // Queue to track the coordinates to be visited

Map<String, Integer> distance = new HashMap<>(); // Distance from start coordinate to a coordinate
                                                  // key is coordinate, value is distance

Map<String, String> previous = new HashMap<>(); // Previous coordinate of a coordinate
                                                 // key is coordinate, value is previous coordinate
```

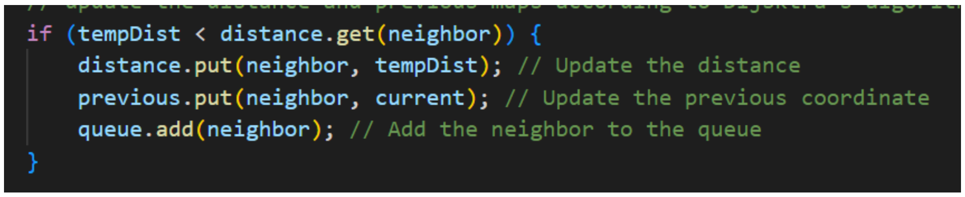
I iterate over all the coordinates in the graph. For each coordinate, I set the initial distance to a largest map value (1000 in this case). I also set the previous coordinate as null because I haven't visited any coordinates yet.



```java
distance.put(coordinate, 1000);
previous.put(coordinate, null);
```

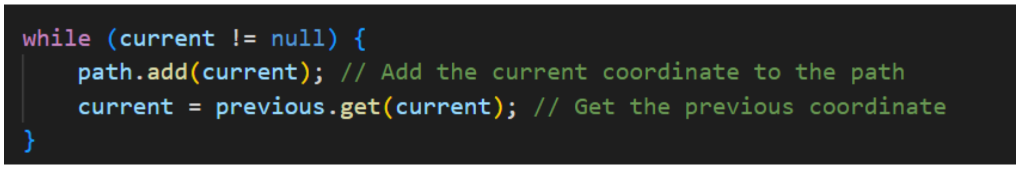
I enter a while loop that continues until the queue is empty. In each iteration, I retrieve the first coordinate from the queue and mark it as visited by adding it to the "visited" set.

Then, I explore the neighbors of the current coordinate. For each neighbor, I check if it has already been visited. If not, I calculate the temporary distance from the start coordinate to the neighbor by adding the distance from the current coordinate to the neighbor. If this temporary distance is smaller than the current distance stored in the "distance" map, it means I have found a shorter path to the neighbor. So, I update the distance map with the new distance, update the previous map to store the current coordinate as the previous coordinate of the neighbor, and add the neighbor to the queue for further exploration.



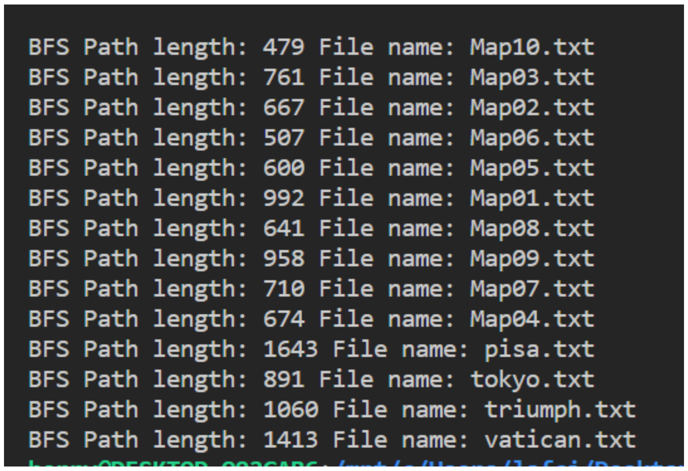
```java
if (tempDist < distance.get(neighbor)) {
    distance.put(neighbor, tempDist); // Update the distance
    previous.put(neighbor, current); // Update the previous coordinate
    queue.add(neighbor); // Add the neighbor to the queue
}
```

Starting from the end coordinate, I iterate through the previous coordinates by following the path backwards until I reach the start coordinate (where the previous coordinate is null). During each iteration, I add the current coordinate to the "path" list. This way, I build the shortest path in reverse order. (I didn't swap because it would look the same either way.)

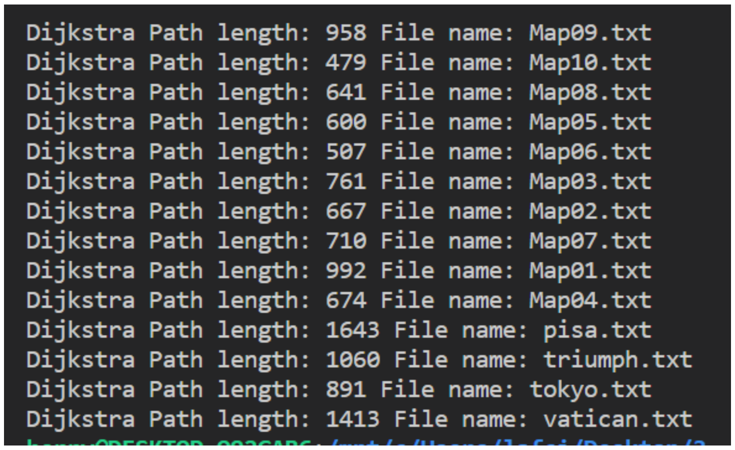


```
while (current != null) {
    path.add(current); // Add the current coordinate to the path
    current = previous.get(current); // Get the previous coordinate
}
```

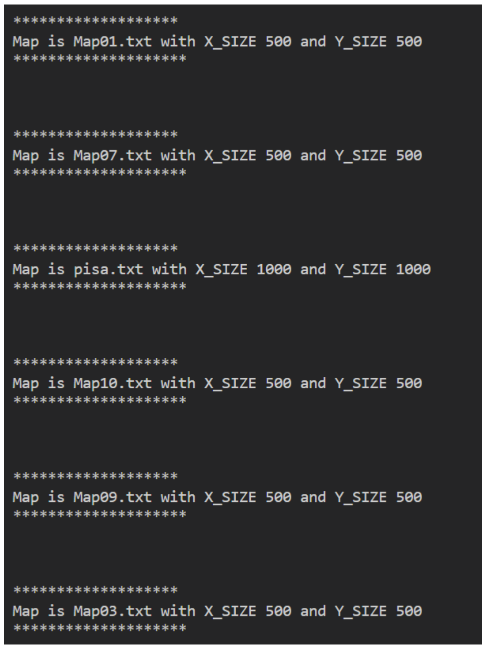
TEST CASES:



```
BFS Path length: 479 File name: Map10.txt
BFS Path length: 761 File name: Map03.txt
BFS Path length: 667 File name: Map02.txt
BFS Path length: 507 File name: Map06.txt
BFS Path length: 600 File name: Map05.txt
BFS Path length: 992 File name: Map01.txt
BFS Path length: 641 File name: Map08.txt
BFS Path length: 958 File name: Map09.txt
BFS Path length: 710 File name: Map07.txt
BFS Path length: 674 File name: Map04.txt
BFS Path length: 1643 File name: pisa.txt
BFS Path length: 891 File name: tokyo.txt
BFS Path length: 1060 File name: triumph.txt
BFS Path length: 1413 File name: vatican.txt
```

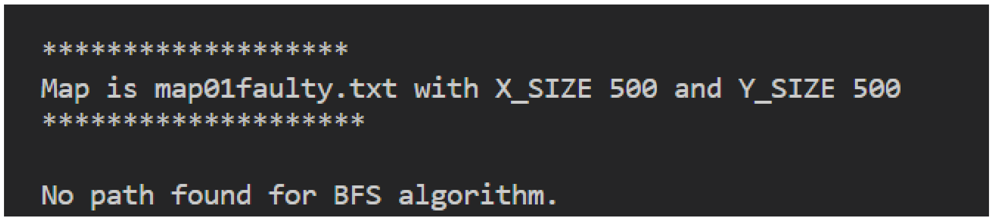


```
Dijkstra Path length: 958 File name: Map09.txt
Dijkstra Path length: 479 File name: Map10.txt
Dijkstra Path length: 641 File name: Map08.txt
Dijkstra Path length: 600 File name: Map05.txt
Dijkstra Path length: 507 File name: Map06.txt
Dijkstra Path length: 761 File name: Map03.txt
Dijkstra Path length: 667 File name: Map02.txt
Dijkstra Path length: 710 File name: Map07.txt
Dijkstra Path length: 992 File name: Map01.txt
Dijkstra Path length: 674 File name: Map04.txt
Dijkstra Path length: 1643 File name: pisa.txt
Dijkstra Path length: 1060 File name: triumph.txt
Dijkstra Path length: 891 File name: tokyo.txt
Dijkstra Path length: 1413 File name: vatican.txt
```



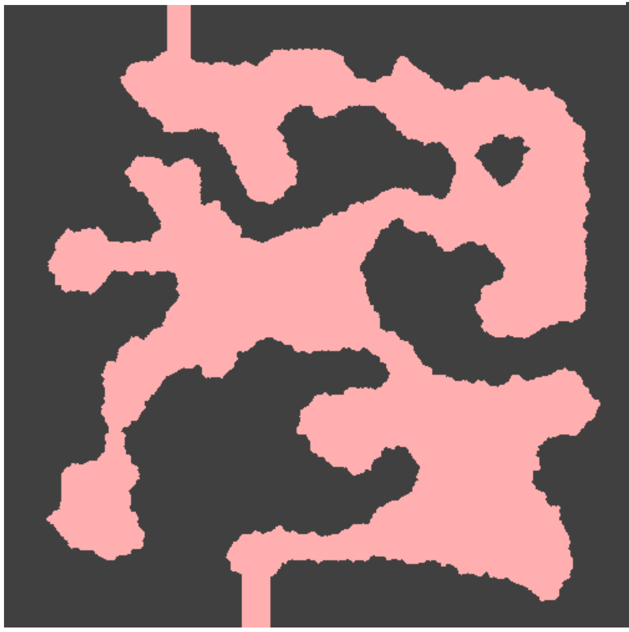
```
*******************
Map is Map01.txt with X_SIZE 500 and Y_SIZE 500
********************



*******************
Map is Map07.txt with X_SIZE 500 and Y_SIZE 500
********************



*******************
Map is pisa.txt with X_SIZE 1000 and Y_SIZE 1000
********************



*******************
Map is Map10.txt with X_SIZE 500 and Y_SIZE 500
********************



*******************
Map is Map09.txt with X_SIZE 500 and Y_SIZE 500
********************



*******************
Map is Map03.txt with X_SIZE 500 and Y_SIZE 500
********************
```

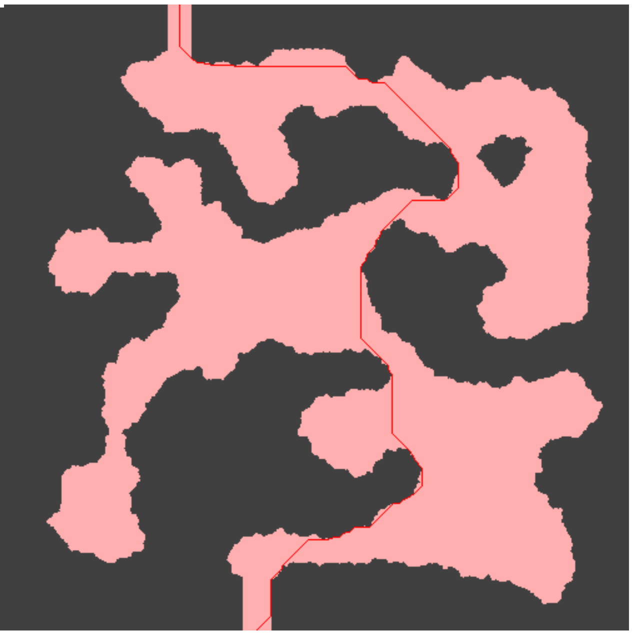
Here is a faulty path test:

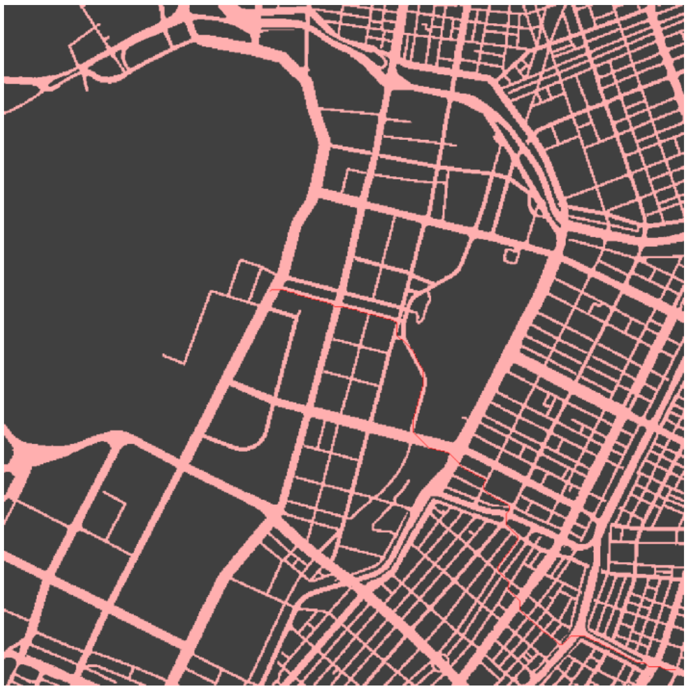


```
*******************
Map is map01faulty.txt with X_SIZE 500 and Y_SIZE 500
********************

No path found for BFS algorithm.
```

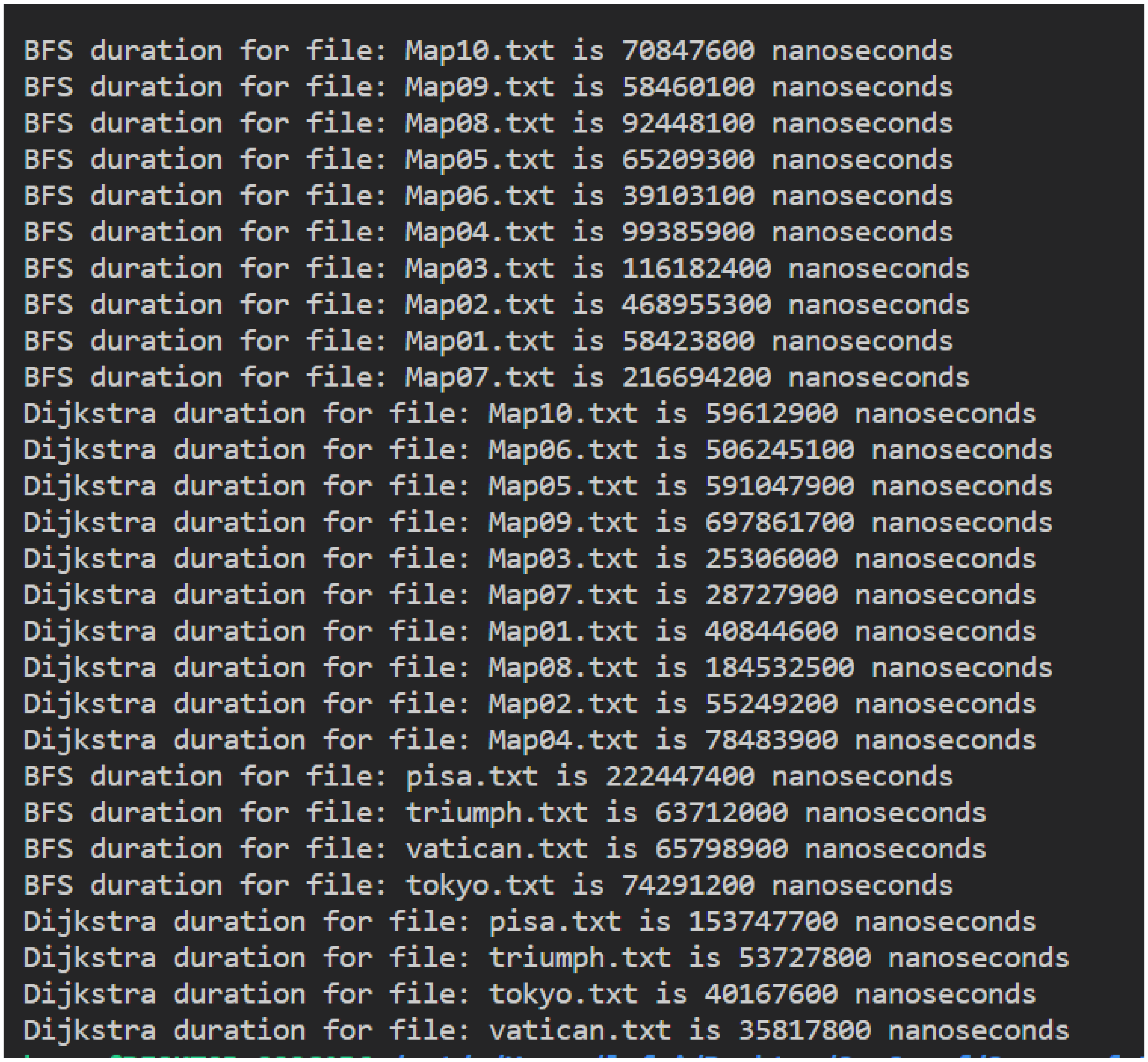
Here are examples of map and map with path:







Durations:



```
BFS duration for file: Map10.txt is 70847600 nanoseconds
BFS duration for file: Map09.txt is 58460100 nanoseconds
BFS duration for file: Map08.txt is 92448100 nanoseconds
BFS duration for file: Map05.txt is 65209300 nanoseconds
BFS duration for file: Map06.txt is 39103100 nanoseconds
BFS duration for file: Map04.txt is 99385900 nanoseconds
BFS duration for file: Map03.txt is 116182400 nanoseconds
BFS duration for file: Map02.txt is 468955300 nanoseconds
BFS duration for file: Map01.txt is 58423800 nanoseconds
BFS duration for file: Map07.txt is 216694200 nanoseconds
Dijkstra duration for file: Map10.txt is 59612900 nanoseconds
Dijkstra duration for file: Map06.txt is 506245100 nanoseconds
Dijkstra duration for file: Map05.txt is 591047900 nanoseconds
Dijkstra duration for file: Map09.txt is 697861700 nanoseconds
Dijkstra duration for file: Map03.txt is 25306000 nanoseconds
Dijkstra duration for file: Map07.txt is 28727900 nanoseconds
Dijkstra duration for file: Map01.txt is 40844600 nanoseconds
Dijkstra duration for file: Map08.txt is 184532500 nanoseconds
Dijkstra duration for file: Map02.txt is 55249200 nanoseconds
Dijkstra duration for file: Map04.txt is 78483900 nanoseconds
BFS duration for file: pisa.txt is 222447400 nanoseconds
BFS duration for file: triumph.txt is 63712000 nanoseconds
BFS duration for file: vatican.txt is 65798900 nanoseconds
BFS duration for file: tokyo.txt is 74291200 nanoseconds
Dijkstra duration for file: pisa.txt is 153747700 nanoseconds
Dijkstra duration for file: triumph.txt is 53727800 nanoseconds
Dijkstra duration for file: tokyo.txt is 40167600 nanoseconds
Dijkstra duration for file: vatican.txt is 35817800 nanoseconds
```

Time complexities:

BFS: O(V + E)

The BFS algorithm visits each vertex and each edge once, resulting in a linear time complexity relative to the size of the graph (V for vertices and E for edges).

Dijkstra: O((V + E) log V)

The time complexity of Dijkstra's algorithm depends on the chosen implementation. With a min-heap data structure, the time complexity is O((V + E) log V), where V is the number of vertices and E is the number of edges.

But for my code the time complexity is O(V + E). Because I used a **regular queue** instead of a priority queue to store the coordinates to be visited. As a result, the time complexity of the queue operations is linear, not logarithmic.

As we see in durations, the nanoseconds are almost similar in both algorithms.

If we compare time complexities, BFS has a linear time complexity and is suitable for unweighted graphs, while Dijkstra's algorithm has a higher time complexity but is designed to handle weighted graphs and find the shortest path. The choice between the algorithms depends on the specific characteristics of the graph and the problem requirements.