CSE222

HomeWork #8

Report

1901042676

Mustafa Mercan

Subject:

In this assignment, we are given a '.txt' file representing a map consisting of 0s, 1s, and -1s. We are required to perform certain operations based on this dataset. Our task is to read the data and store it in an NxN matrix, while converting -1 values to 1. The 1 values in the data represent obstacles, while the 0 values represent usable paths.

This project consists of multiple classes, each with a different functionality. These classes are: "CSE222Map, CSE222Graph, CSE222Dijkstra, CSEE222BFS".

The CSE222Map class should read the ".txt" file and store the data it contains in an NxN matrix.
The CSE222Graph class takes the Map class we created as input and based on the data, it should determine the usable paths and create a graph structure.
The CSE222Dijkstra class takes the created map object as input and applies the Dijkstra algorithm to this graph.
The CSE222BFS class, again taking the created map object as input, should apply the BFS algorithm to this graph.

In addition to these, I have created a class named "Node" to make the code more readable and understandable. This class represents the nodes in the graph. The Node class holds the y and x coordinates specific to that node.

SOLUTIONS APPROACH

CSE222Map Class

Firstly, I organized the file structure of my project. The 'Maps(inputs)' folder contains the necessary input files with the .txt extension. The 'Text(path)' file stores the generated paths after the operations. Finally, the 'Maps(outputs)' folder contains the generated map visuals.

I created the Map class. The Map object takes a string parameter in the format of "./Maps(inputs)/<map_name>.txt", representing the file path, and an integer representing the size N of the NxN matrix to be created.

"Here is the content of the Map class that I have created."

```
String path_name;// it holds input path name
String file_name;// it holds input file name

public int [][] map; // it holds NxN matrix
public int [] start_points; // it holds y-x values
public int [] end_points; // it holds y-x values
//The parameters flag1 and flag2 are used in the draw_line method to respectively draw the paths of the Dijkstra and BFS algorithms.
boolean flag1;
boolean flag2;
```

The Map class consists of the following methods: constructor, read_input, map_to_png, draw_line, and write_path.

The read_input method reads the input.txt file located at the given path and stores the 0 and 1 values in an N x N matrix.

The map_to_png method generates a visual representation based on the 0 and 1 values in the N x N matrix and saves it to the specified path.

The draw_line method takes a Node path as a parameter and draws the generated path on the map based on the y and x values in the path.

The write_path method takes a Node path as a parameter and saves the y and x values in the path to a ".txt" file created in the "Text(path)" folder.

Node Class

After successfully creating the Map, I proceeded to create the Node class which I will use in my Graph class. Here is the content of the Node class:

```java
public class Node{

    public int x; // it holds x value
    public int y;// it holds y value

    public Node(int y, int x)
    {
        this.y = y;
        this.x = x;
    }
    public void print()// it prints y and x value
    {
        System.out.println("y -> " +this.y + " x -> "+this.x);
    }
}
```

CSE222Graph Class

This class requires the use of two different map structures. The 'Nodes' map structure holds a node in the format of 'y-x' as the key and a Node object as the value. The 'y-x' format represents the coordinates where the Node is located, allowing easy access to that node.

The other structure, 'node_list_with_edge', holds a Node as the key and the value is a list structure containing the neighboring Nodes associated with that Node.

The content of CSE222Graph is as follows:

```java
public class CSE222Graph{


    //holds Node as the key and stores a list of edge nodes belonging to the key node as the value
    public Map<Node, List<Node>> node_list_with_edge;

    //holds data in the format of "y-x" as the key, and holds the node located at that coordinate as the value
    public Map<String,Node> nodes;

    public Node start_node; //holds the start node
    public Node end_node; // golds the target node
    public int count; // holds the number of created nodes
```

This class takes a pre-created Map as a parameter and creates a graph with the 0 values found in the map. The class consists of the following methods: constructor, add_node, add_edge, get_node, create_graph, and print_graph.

Constructor method: It takes the Map object as a parameter. After allocating space for the necessary data, it creates a graph.
It also finds the start_node and end_node values. Finally, it displays an informative message.

Add_node method: It adds the Node passed as a parameter to the graph.
Add_edge method: It takes src and dest nodes as parameters. It then adds the dest node to the edge of the src node.

Get_node method: It finds and returns the Node with the given y and x values passed as parameters.

Create_graph method: It reads the values in the input map and creates a new node and adds it to the graph if the value is equal to 0. After successfully completing this step, it traverses the map again and whenever it encounters a value of 0, it finds the node corresponding to those coordinates and checks if there are neighboring nodes within that node. If there is another neighboring node, it adds it as an edge.

Print_graph method: This method prints the nodes and edges of the created graph to verify if the graph is correctly constructed.

CSE222Dijkstra Class

The 'CSE222Dijkstra' class takes the pre-created 'graph' object as input parameter. It then processes the data as desired. The 'CSE222Dijkstra' class holds three different data: 'graph', 'start_node', and 'end_node'. The 'graph' variable stores the graph object of type 'CSE222Graph' that was previously created. The 'start_node' and 'end_node' variables store the 'start_node' and 'end_node' Nodes within the graph object.

The content of the CSE222Dijkstra class is as follows:

```
public CSE222Dijkstra(CSE222Graph input)
{
    this.graph = input; // it holds the graph of type CSE222Graph
    this.start_node = input.start_node;// it holds the start node of type Node
    this.end_node = input.end_node; // it holds the target node of type Node
}
```

The CSE222Dijkstra class consists of two different methods: constructor and findPath.

Constructor method: Processes the data held within the class.

findPath method: Takes a Graph as input and finds the shortest path between the start_node and end_node values using the Dijkstra algorithm. It then saves this path into a list and returns it.

findPath Method Analysis

1→ Firstly, a HashMap named 'distances' is created, which stores Integer
values corresponding to Node keys. This map will hold the distances of each node from the start node.

2→ Next, a HashMap structure named 'previous' is created, where both the key and value are of type 'Node'. This map is used to store the previous node on the shortest path for each node. It keeps track of the node that comes before a particular node on the shortest path.

3→ Then, a HashSet named 'visited' is created. This set will store the visited nodes.

4→An instance of a priority queue, named 'priorityQueue', is created. This queue will prioritize the nodes based on their distances. (Initially, I attempted to use a regular list structure, but the program took 10-15 minutes to run. Therefore, I decided to use the priority queue data structure to maintain the nodes in a sorted order.)

5→The 'distances' map is populated with initial values. For all nodes, distances are set to infinity (Integer.MAX_VALUE), except for the start node, which has a distance of 0.

6→All nodes are added to the priority queue

7→The following steps are repeated as long as the visited set does not contain the target node:
*The closest node (current_node) is removed from the priority queue and added to the visited set.
*The neighbors of the current_node are obtained.
*For each neighbor, a new distance is calculated by increasing the distance from current_node by 1 unit. If this new distance is smaller than the current distance recorded in the distances map for the neighbor, the new distance is stored in the distances map, the current_node is stored in the previous map as the previous node for the neighbor, and the neighbor is removed from and re-added to the priority queue

8→An ArrayList is created for the shortest path list (shortest_path).

9→The shortest_path list is updated using the previous map and then returned.

CSE222BFS Class

The 'CSE222BFS' class takes the pre-created 'graph' object as input parameter. It then processes the data as desired. The 'CSE222BFS' class holds three different data: 'graph', 'start_node', and 'end_node'. The 'graph' variable stores the graph object of type 'CSE222Graph' that was previously created. The 'start_node' and 'end_node' variables store the 'start_node' and 'end_node' Nodes within the graph object.

The content of the CSE222BFS class is as follows:

```
public class CSE222BFS{

    CSE222Graph graph;// it holds the graph of type CSE222Graph
    Node start_node;// it holds the start node of type Node
    Node end_node; // it holds the target node of type Node

    public CSE222BFS(CSE222Graph input)
    {
        this.graph = input;
        this.start_node = input.start_node;
        this.end_node = input.end_node;
    }
```

The CSE222BFS class consists of two different methods: constructor and findPath.

Constructor method: Processes the data held within the class.

findPath method: Takes a Graph as input and finds the shortest path between the start_node and end_node values using the BFS algorithm. It then saves this path into a list and returns it.

findPath Method Analysis

This methods aims to find the shortest path from the start node to the end node using the Breadth-First Search (BFS) algorithm.

1→Firstly, a function named findPath() is defined, and it is expected to return a result of type List<Node>.

2→Within the function, three data structures named previous, queue, and visited are defined. previous is a Map (HashMap) that maps a Node to its corresponding Node, queue is a Queue that stores Nodes, and visited is a set of Nodes.

3→The start_node is added to the queue and also added to the visited set. This signifies that the start_node has been visited.

4→The loop is entered without emptying the queue, and the following steps are repeated:
*A node (current_node) is taken from the queue.
*If the current_node is the same as the target node (end_node), the loop is exited because it means the target has been reached.
*Otherwise, the neighbors of the current_node are retrieved, and the following steps are repeated for each neighbor node within a loop:
**If the neighbor has not been visited (i.e., it is not present in the visited set)
***The neighbor is added to the queue.
***The neighbor is added to the visited set.
***A key-value pair (neighbor, current_node) is added to the previous map. This indicates that the neighbor node can be reached through the current_node.

5→After the completion of the queue loop, an empty list named shortest_path is defined to store the nodes of the shortest path, and the current_node variable is assigned the value of end_node.

6→After the operations are completed, the shortest_path list is updated and contains the shortest path, and this list is returned using the return statement.

TestCases Class

The TestCases class is created to instantiate and test the necessary objects in the order specified in the PDF. Its contents are as follows:

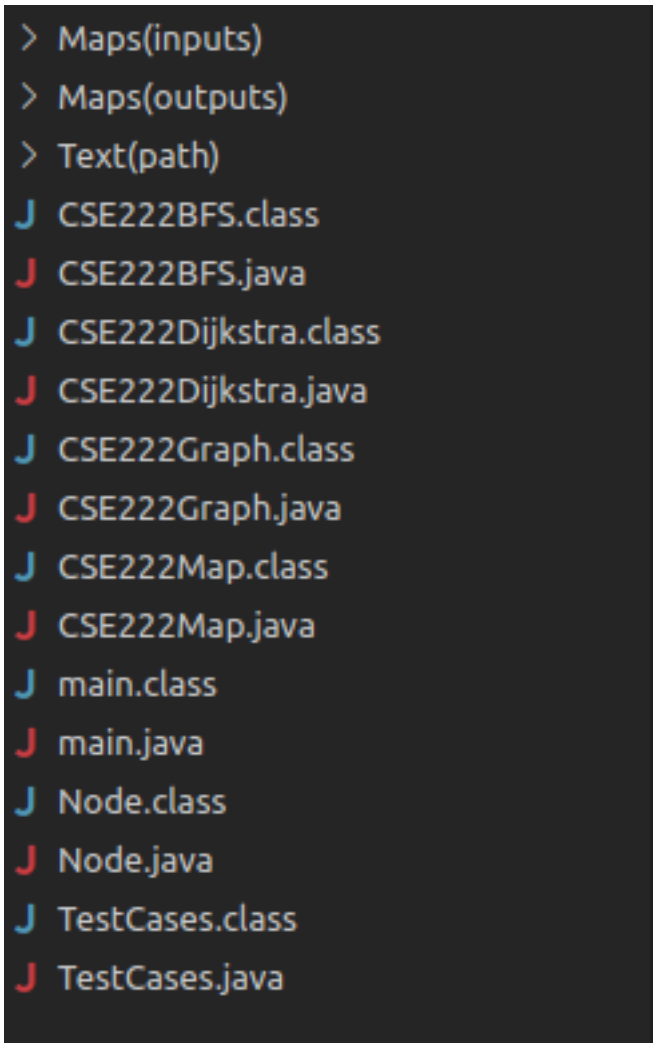
```java
public TestCases(String map_path, int y, int x)
{
    this.map_path = map_path;
    this.y = y;
    this.x = x;
}
public void run()
{
    CSE222Map map = new CSE222Map(map_path,y,x); // +

    String[] parts = map_path.split("/");


    String fileNameWithExtension = parts[parts.length - 1];

    String fileName = fileNameWithExtension.replace(".txt", "");


    String map_png_path = "./Maps(outputs)/"+fileName+".png";

    map.map_to_png(map_png_path);


    CSE222Graph graph = new CSE222Graph(map);

    CSE222Dijkstra dijkstra = new CSE222Dijkstra(graph);

    List <Node> dijkstraPath = dijkstra.findPath();

    CSE222BFS bfs = new CSE222BFS(graph);

    List <Node> bfsPath = bfs.findPath();

    map.draw_line(dijkstraPath);
    map.draw_line(bfsPath);

    System.out.println("Dijkstra path size = " + dijkstraPath.size());
    System.out.println("BFS path size = " + bfsPath.size());

    map.write_path(dijkstraPath);

    map.write_path(bfsPath);


}
```

File Structure

"Maps(inputs)" = It contains the ".txt" files to be given as input.

"Maps(outputs)" = It contains the generated maps as output.

"Text(path)" = It contains the ".txt" outputs that include the coordinates of the generated paths.



Test Case

If you want to test with a different map other than the .txt files shared with us, you should add the .txt file to the Maps(inputs) folder. Then, you need to create the TestCases class inside the main.java file and provide the correct parameters.

```
new Thread (new TestCases("./Maps(inputs)/map01.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map02.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map03.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map04.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map05.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map06.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map07.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map08.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map09.txt",500,500)).start();
new Thread (new TestCases("./Maps(inputs)/map10.txt",500,500)).start();


new Thread (new TestCases("./Maps(inputs)/tokyo.txt",1000,1000)).start();
new Thread (new TestCases("./Maps(inputs)/pisa.txt",1000,1000)).start();
new Thread (new TestCases("./Maps(inputs)/triumph.txt",1000,1000)).start();
new Thread (new TestCases("./Maps(inputs)/vatican.txt",1000,1000)).start();
```

Run Times

```
PNG was created: /home/mustafa/Desktop/data-/hw08/hw08-update-2/./Maps(outputs)/map01.png
==============Graph was created==========
start cord. -> 0 114
end cord. -> 499 392
=========================================
==================================
Runtime Dijkstra: 1776.56266 ms
Runtime BFS: 146.35335 ms
==================================
==================================
Dijkstra path size = 992
BFS path size = 992
==================================
```

```
PNG was created: /home/mustafa/Desktop/data-/hw08/hw08-update-2/./Maps(outputs)/map02.png
==============Graph was created==========
start cord. -> 0 382
end cord. -> 499 62
=========================================
==================================
Runtime Dijkstra: 1417.721527 ms
Runtime BFS: 114.966324 ms
==================================
==================================
Dijkstra path size = 667
BFS path size = 667
==================================
```

```
PNG was created: /home/mustafa/Desktop/data-/hw08/hw08-update-2/./Maps(outputs)/map03.png
==============Graph was created==========
start cord. -> 171 0
end cord. -> 460 499
=========================================
==================================
Runtime Dijkstra: 1381.561703 ms
Runtime BFS: 121.561883 ms
==================================
==================================
Dijkstra path size = 761
BFS path size = 761
==================================
```

```
PNG was created: /home/mustafa/Desktop/data-/hw08/hw08-update-2/./Maps(outputs)/map04.png
==============Graph was created==========
start cord. -> 0 406
end cord. -> 499 55
=========================================
==================================
Runtime Dijkstra: 2104.948046 ms
Runtime BFS: 122.241845 ms
==================================
==================================
Dijkstra path size = 674
BFS path size = 674
==================================
```

Dijkstra Algorithm Time Comlexity:

The time complexity of the Dijkstra Algorithm:

→ Initially, all nodes are set to infinity and added to the priority queue. This step takes O(V log V) time.

→ The loop continues until all nodes are visited or the target is reached. In the worst case, the loop runs V times.

→ In each loop iteration, the neighboring nodes of a node are scanned, and the priority queue is updated. This step takes O(E log V) time.

→ Finally, the nodes of the shortest path are added to the shortest_path list. This step takes O(V) time.
In total, the time complexity of the code is O((V + E) log V), where V represents the number of nodes and E represents the number of edges.

BFS Algorithm Time Comlexity:

→First, the start_node is added to the queue and marked as visited. This step has a time complexity of O(1).

→The loop continues as long as the queue is not empty. In each iteration, a node is dequeued and its neighboring nodes are checked. This step has a complexity of O(E), depending on the number of nodes and edges.

→If a neighboring node has not been visited before, it is added to the queue, marked as visited, and added to the previous map. These operations have a time complexity of O(1).

→The loop continues until the target node is reached or the queue is emptied. In the worst case, all nodes are visited and all edges are traversed, so the loop has a complexity of O(V + E).

→To retrieve the nodes of the shortest path by traversing the previous map in reverse, another loop is used. This step has a complexity of O(V) in the case of a path with a maximum of V nodes.

→Finally, the duration of the process is calculated, which does not affect the cumulative complexity.
Therefore, the overall time complexity of the BFS algorithm in this code can be summarized as O(V + E).

Dijkstra Algorithm VS BFS Algorithm Running Time

When comparing the speed of BFS and Dijkstra algorithms, BFS is generally faster because it visits each node at the same depth and expands downward. Dijkstra algorithm, on the other hand, serves a more specific purpose by using a priority queue based on costs. Dijkstra algorithm may be slower because it requires more operations in each step to update the priority queue and select the node with the minimum cost.