GEBZE TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE 222 - SPRING 2023

HOMEWORK 8 REPORT

ETHEM MENGAASLAN

200104004016

Class Structures:

Coordinate:

Represents a coordinate with x and y values.

The hashCode() method is overridden to generate a unique hash code for each Coordinate object. This is necessary when using the Coordinate class in data structures such as hash maps or hash sets, which rely on hash codes to efficiently store and retrieve elements.
The equals() method is overridden to define the equality comparison between Coordinate objects. By comparing the x and y values of two Coordinate objects, this method determines if they represent the same coordinate or not. This is useful when comparing coordinates for equality checks or when using Coordinate objects as keys in data structures.

CSE222Map:

The CSE222Map class represents a map with coordinates and provides methods for map operations, specifically for turning text files into maps and generating graphs for path-finding operations.

The class utilizes various helper methods to perform tasks such as filling the coordinates matrix with values from the file, counting the number of rows and columns in the file, and setting the start and end points from the file.

CSE222Graph:

The CSE222Graph class represents a graph used for finding paths in the context of CSE222 Homework 8. It contains an adjacency map that represents the graph structure, where each node is associated with a list of neighboring nodes.

The class can be constructed either as an empty graph or based on a given CSE222Map object. When constructed from a map, it extracts the start and end points from the map and builds the graph by iterating over the elements of the coordinatesMatrix. If a coordinate value is 0, it creates a new node and adds it to the graph. It also checks the surrounding elements to determine adjacency relations and adds corresponding edges between nodes.

The CSE222Graph class encapsulates the graph structure used for path finding in the CSE222 homework assignment. It allows adding nodes, creating edges between nodes, retrieving neighboring nodes, and inspecting the graph's contents.

CSE222BFS:

The CSE222BFS class implements the Breadth First Search (BFS) algorithm for finding the shortest path in a given graph represented by the CSE222Graph class.

The class can be constructed with a CSE222Graph object, which represents the graph on which the BFS algorithm will be applied.

The findPath method applies the BFS algorithm to find the shortest path in the graph. It returns a list of Coordinate objects representing the shortest path from the source node to the destination node**.**

The BFS Path Finding algorithm works as follows:

It initializes the source and destination nodes based on the start and end points of the graph.

It maintains a map called previousNodes to store node-to-previous-node pairs. This map is used to reconstruct the path after the BFS traversal.

It uses a queue to keep track of the nodes to be visited and a set to mark visited nodes. The algorithm starts with the source node. It adds it to the queue and marks it as visited.

While the queue is not empty, it dequeues the next node and checks if it is the destination node. If so, it breaks the loop.

For each neighbor of the current node that has not been visited, it adds the neighbor to the queue, marks it as visited, and stores the previous node information in the previousNodes map.

After the BFS traversal completes, it reconstructs the shortest path by iterating from the destination node to the source node using the previousNodes map. It adds each node to the shortestPath list.

Finally, it reverses the shortestPath list to obtain the path from the source to the destination and returns the reversed list.

CSE222Dijkstra:

The CSE222Dijkstra class implements Dijkstra's algorithm for finding the shortest path in a CSE222Graph object.

The class has a constructor that takes a CSE222Graph object as a parameter. This represents the graph on which the Dijkstra's algorithm will be applied.

The findPath method applies the Dijkstra’s algorithm to find the shortest path in the graph. It returns a list of Coordinate objects representing the shortest path from the source node to the destination node.

The Dijkstra Path Finding algorithm works as follows:

The algorithm starts by setting the source and destination nodes based on the start and end points of the graph.

The algorithm initializes the necessary data structures, including distances, previousNodes, a priorityQueue, and a visitedNodes set.

distances stores the distances from the source node to each node in the graph. Initially, all distances are set to a large value, except the start node which is set to 0.

previousNodes keeps track of the previous node in the shortest path for each node in the graph.

The priorityQueue is used to prioritize the nodes based on their distances, a Comparator is used for this purpose.

The visitedNodes set is used to mark the nodes that have been visited.

The algorithm starts by adding the start node to the priorityQueue.

While the priorityQueue is not empty, the algorithm dequeues the node with the smallest distance.

If the dequeued node is the destination node, the algorithm breaks the loop as it has found the shortest path.

If not, the algorithm marks the dequeued node as visited and explores its neighboring nodes.

For each neighboring node, the algorithm calculates a new distance by adding the edge weight (1 in this case) to the distance of the current node.

If the new distance is smaller than the previous distance to the neighboring node, the algorithm updates the distance, sets the previous node, and enqueues the neighboring node into the priorityQueue.

Once the algorithm completes, it reconstructs the shortest path by iterating from the destination node to the source node using the previousNodes map. It adds each node to the shortestPath list.

Finally, the algorithm reverses the shortestPath list to obtain the path from the source to the destination and returns the reversed list.

Time Complexity Analysis for BFS:

Starting the breadth-first search (BFS) logic with a while loop: The time complexity of the while loop depends on the number of nodes in the graph and the number of edges. In the worst case, where every node is connected to every other node, the time complexity would be O(N^2), where N is the number of nodes. However since we have at max 8 adjacent/neighbouring nodes, it can be ignored as a constant value. So the time complexity is O(N + E), where N is number of nodes and E is the total number of edges.

Time Complexity Analysis of Dijkstra:

The algorithm iterates over the nodes in the graph and performs some operations based on the number of edges between the nodes. In the worst case where every node is connected to every other node, the time complexity would be O(N^2), (N is the number of nodes) . But, since the number of adjacent/neighboring nodes is limited (e.g., at most 8), this can be considered a constant value and be ignored. Because of that, the overall time complexity can be simplified to O(N + E), where N is the number of nodes and E is the total number of edges in the graph.

OUTPUT:

```
For Map01.txt:
Dijkstra length: 991
BFS length: 991
```

```
For Map02.txt:
Dijkstra length: 666
BFS length: 666
```

```
For Map03.txt:
Dijkstra length: 760
BFS length: 760
```

```
For Map04.txt:
Dijkstra length: 673
BFS length: 673
```

```
For Map05.txt:
Dijkstra length: 599
BFS length: 599
```

```
For Map06.txt:
Dijkstra length: 506
BFS length: 506
```

```
For Map07.txt:
Dijkstra length: 709
BFS length: 709
```

```
For Map08.txt:
Dijkstra length: 640
BFS length: 640
```

```
For Map09.txt:
Dijkstra length: 957
BFS length: 957
```

```
For Map10.txt:
Dijkstra length: 478
BFS length: 478
```

Run Times:

Map1:

```
Map01.txtDijkstra findPath() Elapsed Time: 1775158
```

```
Map01.txtBFS findPath() Elapsed Time: 418108
```

Map2:

```
Map02.txtDijkstra findPath() Elapsed Time: 2606353
```

```
Map02.txtBFS findPath() Elapsed Time: 404537
```

Map3:

```
Map03.txtDijkstra findPath() Elapsed Time: 1604976
```

```
Map03.txtBFS findPath() Elapsed Time: 663963
```

Map4:

```
Map04.txtDijkstra findPath() Elapsed Time: 1826740
```

```
Map04.txtBFS findPath() Elapsed Time: 404576
```

Map5:

```
Map05.txtDijkstra findPath() Elapsed Time: 1213491
```

```
Map05.txtBFS findPath() Elapsed Time: 472038
```

Map6:

```
Map06.txtDijkstra findPath() Elapsed Time: 1534519
```

```
Map06.txtBFS findPath() Elapsed Time: 623547
```

Map7:

```
Map07.txtDijkstra findPath() Elapsed Time: 2227903
```

```
Map07.txtBFS findPath() Elapsed Time: 565858
```

Map8:

```
Map08.txtDijkstra findPath() Elapsed Time: 1198833
```

```
Map08.txtBFS findPath() Elapsed Time: 354865
```

Map9:

```
Map09.txtDijkstra findPath() Elapsed Time: 1342556
```

```
Map09.txtBFS findPath() Elapsed Time: 364155
```

Map10:

```
Map10.txtDijkstra findPath() Elapsed Time: 889359
```

```
Map10.txtBFS findPath() Elapsed Time: 180540
```

As seen, BFS works much faster than Dijkstra's algorithm.