GTU Department of Computer Engineering

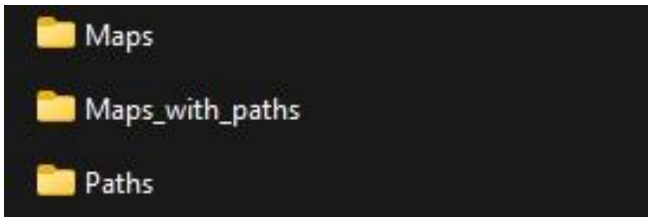
CSE 222/505 – Spring 2023

Homework 7 Report

Barış Batuhan Bolat

210104004029

- When program finished these three folder was created. “Maps” contains maps as PNG , “Maps_with_paths” contains maps with path lines and “Paths” contains paths as txt file.



1. Time Complexity Analysis

A. CSE222Map Class

a. Constructor

- The constructor starts by reading the input file. The time complexity of reading a file is typically considered as O(n), where n is the size of the file.
- After that reads lines list and parses the map data. It iterates over each line and splits the values, then converts them to integers and stores them in the map array. This step has a time complexity of $O(n^2)$ since it involves iterating over each element of the map array.
- Overall time complexity of this method is $O(n^2)$.

```
public CSE222Map(String filename,int sizeX,int sizeY) {
    try {
        Scanner reader = new Scanner(new File(filename));
        //Reading first two rows that holds starting and ending points of map
        String[] startCoords = reader.nextLine().split(regex:",");
        startY = Integer.parseInt(startCoords[0]);
        startX = Integer.parseInt(startCoords[1]);

        String[] endCoords = reader.nextLine().split(regex:",");
        endY = Integer.parseInt(endCoords[0]);
        endX = Integer.parseInt(endCoords[1]);

        //And rest of the txt file holds map
        size = sizeX;
        map = new int[size][size];

        List<String> lines = new ArrayList<>();
        String line;
        while (reader.hasNextLine() && (line = reader.nextLine()) != null) {
            lines.add(line);
        }

        for (int i = 0; i < size; i++) {
            String[] values = lines.get(i).split(regex:",");
            for (int j = 0; j < size; j++) {
                map[i][j] = Integer.parseInt(values[j].replaceAll(regex:"-1", replacement:"1"));
            }
        }

        reader.close();
    }catch (FileNotFoundException e) {
        System.out.println(x:"File not found");
        System.exit(status:0);
    }
}
```

b. convertPNG(String filename)

- The time complexity of creating a folder using folder.mkdirs() is O(1).
- The nested for loop iterates over each pixel in the image, which has a time complexity of $O(n^2)$ where n is size.setRGB and getRGB method takes O(1) time
- So the overall time complexity of this method is $O(n^2)$.

```java
public void convertPNG(String filename) {
    try {
        // Create the folder if it doesn't exist
        File folder = new File(pathname:"Maps");
        if (!folder.exists()) {
            boolean created = folder.mkdirs();
            if (!created) {
                return;
            }
        }

        BufferedImage image = new BufferedImage(size, size, BufferedImage.TYPE_INT_RGB);
        // Roads colored by white and walls colored by gray
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (map[i][j] == 1) {
                    image.setRGB(j, i, Color.GRAY.getRGB());
                }
                else {
                    image.setRGB(j, i, Color.WHITE.getRGB());
                }
            }
        }

        // Save the image to the folder
        ImageIO.write(image, formatName:"PNG", new File("Maps" + "/" + filename + ".png"));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

c. drawLine(List<String> path,String filename)

- The time complexity of reading the original image using ImageIO.read() , ImageIO.write() and creating the folder using folder.mkdirs() has constant time complexity.
- The time complexity of marking the coordinates on the image depends on the number of coordinates in the path, denoted as n. The loop that iterates over each coordinate has a time complexity of O(n).
- So the overall time complexity of this method is O(n).

```
public void drawLine(List<String> path,String filename) {
    if (path != null) {
        try {
            BufferedImage image = null;
            // Detecting maps kind for naming
            if (filename.contains(s:"BFS")) {
                image = ImageIO.read(new File("Maps/" + filename.replaceAll(regex:"BFS", replacement:"") + ".png"));
            }
            if (filename.contains(s:"Dijkstra")) {
                image = ImageIO.read(new File("Maps/" + filename.replaceAll(regex:"Dijkstra", replacement:"") + ".png"));
            }

            // Create the folder if it doesn't exist
            File folder = new File(pathname:"Maps_with_paths");
            if (!folder.exists()) {
                boolean created = folder.mkdirs();
                if (!created) {
                    return;
                }
            }

            // Marking the coordinates given in the path on the PNG file
            Graphics2D g2d = image.createGraphics();
            g2d.setColor(Color.RED);
            for (String coordinate : path) {
                String[] coordinates = coordinate.split(regex:",");
                int y = Integer.parseInt(coordinates[0]);
                int x = Integer.parseInt(coordinates[1]);
                g2d.drawLine(x, y, x, y);
            }
            g2d.dispose();

            // Save the image to the folder
            ImageIO.write(image, formatName:"PNG", new File("Maps_with_paths" + "/" + filename + "_with_path.png"));
        } catch (IOException e) {
            System.out.println(x:"File not found.");
        }
    } else {
        System.out.println(x:"Path is null");
    }
}
```

d. writePath(List<String> path,String filename)

- This method iterates over each coordinate in the path list and writes it to the file. The time complexity of this operation is O(n), where n is the number of coordinates in the path list. write method has constant time complexity.
- So the overall time complexity is O(n).

```
public void writePath(List<String> path,String filename) {
    if(path != null){
        File folder = new File(pathname:"Paths");
        if (!folder.exists()) {
            boolean created = folder.mkdirs();
            if (!created) {
                return;
            }
        }
        File file = new File(parent:"Paths", filename + "_path.txt");
        try {
            boolean created = file.createNewFile();
            if (!created) {
                System.out.println(x:"Failed to create the file.");
            }
            else {
                FileWriter out = new FileWriter(file);
                for (String coordinate : path) {
                    out.write(coordinate + "\n");
                }
                out.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    else{
        System.out.println(x:"Path is null");
    }
}
```

B. CSE222Graph Class

a. Constructor

- The constructor uses nested loops to iterate over each cell in the mapArr. The outer loop iterates size times, and the inner loop also iterates size times. Therefore, the nested loops contribute $O(n^2)$ where time complexity. Second nested does not effect time complexity. So overall time complexity is $O(n^2)$.

```
public CSE222Graph(CSE222Map cse222Map) {
    this.map = cse222Map;
    this.adjList = new ArrayList<>();

    int[][] mapArr = map.getMap();
    int size = map.getSize();

    for (int i = 0; i < size * size; i++) {
        adjList.add(new ArrayList<>());
    }

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (mapArr[i][j] == 1) {
                continue;
            }

            int vertex = i * size + j;

            for (int dy = -1; dy <= 1; dy++) {
                for (int dx = -1; dx <= 1; dx++) {
                    if (dy == 0 && dx == 0) {
                        continue;
                    }

                    int nx = j + dx;
                    int ny = i + dy;

                    if (nx >= 0 && nx < size && ny >= 0 && ny < size && mapArr[ny][nx] != 1) {
                        int neighbor = ny * size + nx;
                        adjList.get(vertex).add(new Node(neighbor));
                    }
                }
            }
        }
    }
}
```

- This class has get methods that has constant time complexity

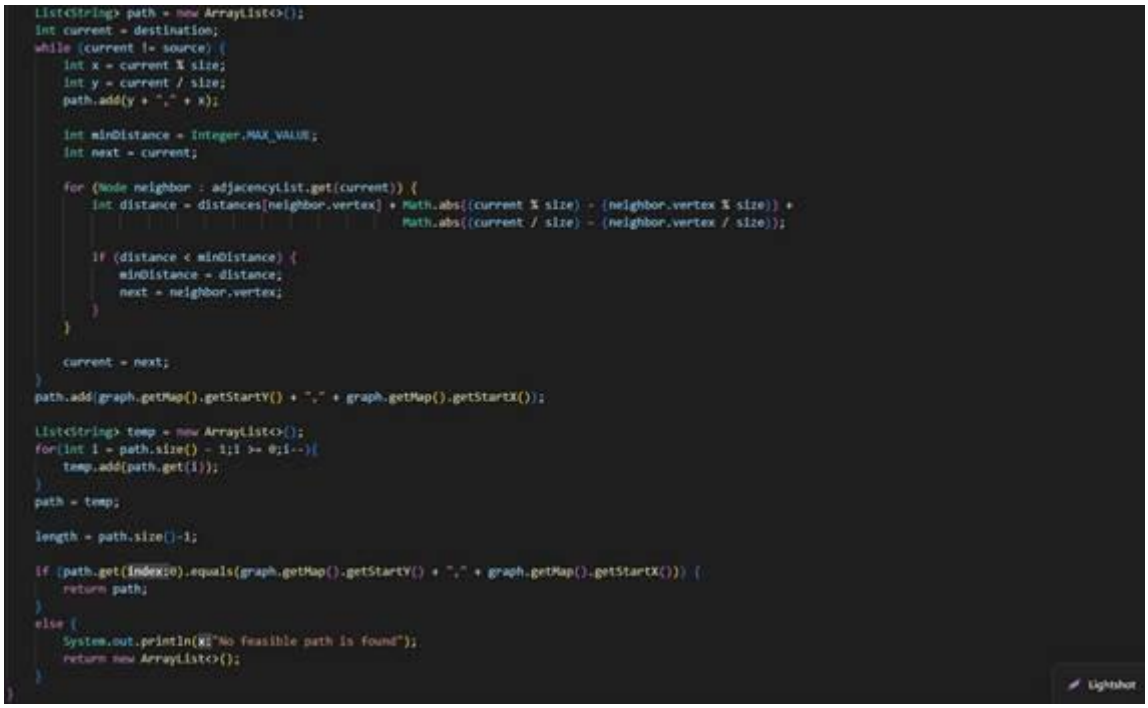
C. CSE222Dijkstra Class

- I skip the consturctor of this class because it copies parameter CSE222Graph variable to its instance variable.

a. findPath()

- The first loop that initializes the distances array has a time complexity of O(size^2), where size is the size of the map.
- The while loop that implements Dijkstra's algorithm in the worst case, each vertex is visited once, and for each vertex, its neighbors are processed. So the time complexity of the while loop is O(V + E), where V is the number of vertices and E is the number of edges in the graph. In this case, V is equal to size^2, and E is at most 4* size^2, as each vertex can have at most 4 neighbors in a 2D map. So this part of method has O(size^2 + 4* size^2) time complexity and this can be minimalized as O(size^2).
- After the Dijkstra's algorithm finishes, there is a loop that constructs the path by traversing the previous nodes. This loop's complexity at most O(size^2). Reversinghas the same complexity.
- So the overall time complexity of this method is O(n^2) where n is size;

```
public List<String> findPath() {
    List<List<Node>> adjacencyList = graph.getAdjList();
    int size = graph.getMap().getSize();
    int[] distances = new int[size * size];

    for (int i = 0; i < size * size; i++) {
        distances[i] = Integer.MAX_VALUE;
    }

    int source = graph.getMap().getStartY() * size + graph.getMap().getStartX();
    int destination = graph.getMap().getEndY() * size + graph.getMap().getEndX();

    distances[source] = 0;

    Queue<Node> queue = new LinkedList<>();
    queue.offer(new Node(source));

    while (!queue.isEmpty()) {
        Node current = queue.poll();

        if (current.isVisited) {
            continue;
        }

        current.isVisited = true;

        if (current.vertex == destination) {
            break;
        }

        for (Node neighbor : adjacencyList.get(current.vertex)) {
            int distance = distances[current.vertex] + Math.abs((current.vertex % graph.getMap().getSize()) - (neighbor.vertex % graph.getMap().getSize())) +
                                                       Math.abs((current.vertex / graph.getMap().getSize()) - (neighbor.vertex / graph.getMap().getSize()));

            if (distance < distances[neighbor.vertex]) {
                distances[neighbor.vertex] = distance;
                queue.offer(neighbor);
            }
        }
    }
```



```
List<String> path = new ArrayList<>();
int current = destination;
while (current != source) {
    int x = current % size;
    int y = current / size;
    path.add(y + "," + x);

    int minDistance = Integer.MAX_VALUE;
    int next = current;

    for (Node neighbor : adjacencyList.get(current)) {
        int distance = distances[neighbor.vertex] + Math.abs((current % size) - (neighbor.vertex % size)) +
                                                    Math.abs((current / size) - (neighbor.vertex / size));

        if (distance < minDistance) {
            minDistance = distance;
            next = neighbor.vertex;
        }
    }

    current = next;
}
path.add(graph.getMap().getStartY() + "," + graph.getMap().getStartX());

List<String> temp = new ArrayList<>();
for(int i = path.size() - 1;i >= 0;i--){
    temp.add(path.get(i));
}
path = temp;

length = path.size()-1;

if (path.get(index:0).equals(graph.getMap().getStartY() + "," + graph.getMap().getStartX())) {
    return path;
}
else {
    System.out.println(x:"No feasible path is found");
    return new ArrayList<>();
}
}
```

- This class has get methods that has constant time complexity

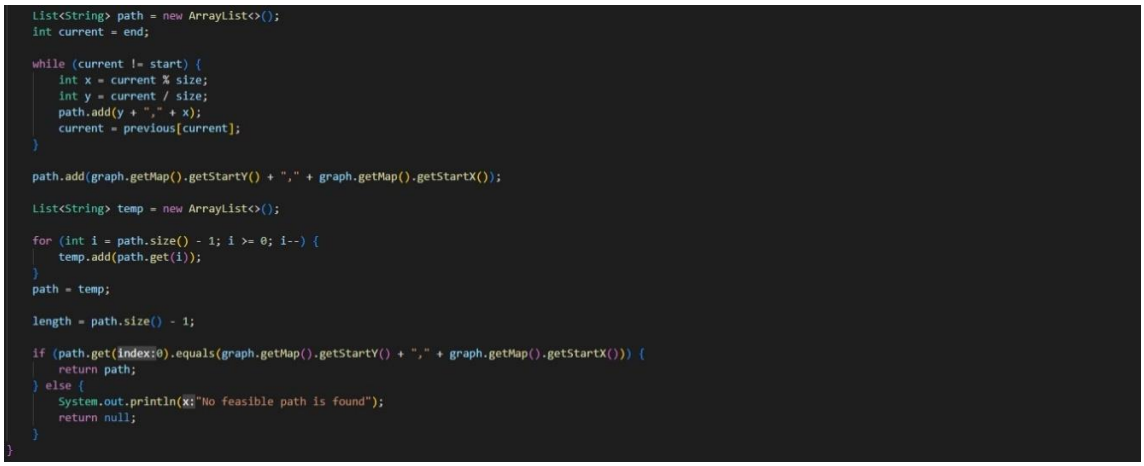
A. CSE222BFS Class

- I skip the consturctor of this class because it copies parameter CSE222Graph variable to its instance variable.

a. findPath()

- The first loop that initializes the previous array has a time complexity of O(size^2), where size is the size of the map.
- The while loop that implements BFS search algorithm in the worst case, each vertex is visited once, and for each vertex, its neighbors are processed. So the time complexity of the while loop is O(V + E), where V is the number of vertices and E is the number of edges in the graph. In this case, V is equal to size^2, and E is at most 4* size^2, as each vertex can have at most 4 neighbors in a 2D map. So this part of method has O(size^2 + 4* size^2) time complexity and this can be minimalized as O(size^2).
- After the BFS Search finishes, there is a loop that constructs the path by traversing the previous nodes. This loop's complexity at most O(size^2). Reversinghas the same complexity.
- So the overall time complexity of this method is O(n^2) where n is size;

```
public List<String> findPath() {
    int size = graph.getMap().getSize();

    int start = graph.getMap().getStartY() * size + graph.getMap().getStartX();
    int end = graph.getMap().getEndY() * size + graph.getMap().getEndX();

    int[] previous = new int[size * size];

    for (int i = 0; i < previous.length; i++) {
        previous[i] = -1;
    }

    Queue<Node> queue = new LinkedList<>();
    queue.offer(new Node(start));

    previous[start] = start;

    while (!queue.isEmpty()) {
        Node current = queue.poll();

        if (current.vertex == end) {
            break;
        }

        int currentX = current.vertex % size;
        int currentY = current.vertex / size;

        int[] newX = {currentX - 1, currentX + 1, currentX, currentX};
        int[] newY = {currentY, currentY, currentY - 1, currentY + 1};

        for (int i = 0; i < 4; i++) {
            if (newX[i] >= 0 && newX[i] < graph.getMap().getSize() && newY[i] >= 0 && newY[i] < graph.getMap().getSize() && graph.getMap().getMap()[newY[i]][newX[i]] != 1) {
                int neighborVertex = newY[i] * size + newX[i];

                if (previous[neighborVertex] == -1) {
                    queue.offer(new Node(neighborVertex));
                    previous[neighborVertex] = current.vertex;
                }
            }
        }
    }
```



```
    List<String> path = new ArrayList<>();
    int current = end;

    while (current != start) {
        int x = current % size;
        int y = current / size;
        path.add(y + "," + x);
        current = previous[current];
    }

    path.add(graph.getMap().getStartY() + "," + graph.getMap().getStartX());

    List<String> temp = new ArrayList<>();

    for (int i = path.size() - 1; i >= 0; i--) {
        temp.add(path.get(i));
    }
    path = temp;

    length = path.size() - 1;

    if (path.get(index:0).equals(graph.getMap().getStartY() + "," + graph.getMap().getStartX())) {
        return path;
    } else {
        System.out.println(x:"No feasible path is found");
        return null;
    }
}
```

- This class has get methods that has constant time complexity

2. Running Time Performance(miliseconds)

Alg/Map	map01	map02	map03	map04	map05	map06	map07	map08	map09	map10	AVG
Dijkstra	345	762	67	261	137	93	264	103	50	60	214
BFS	20	41	118	174	134	330	103	126	71	27	114

Dijkstra's Algorithm:

- Worst-case time complexity: $O(n^2)$, where n is size of map(Map is square and size is length one side).
- Dijkstra's algorithm guarantees finding the shortest path from a source node to all other nodes in the graph.
- It explores nodes in a priority order based on their distances from the source, prioritizing nodes with shorter distances.

Breadth-First Search (BFS) Algorithm:

- Worst-case time complexity: $O(n^2)$, where n is size of map(Map is square and size is length one side).
- BFS explores nodes in a breadth-first manner, visiting all nodes at a given distance level before moving to the next level.

- According my implementation of these two algorithm it looks like their time complexities are same. But for running time comparisons in the table above most of times BFS algorithm is faster than Dijkstra. The reason of this Dijkstra algorithm needs much more effort than BFS when finding the shortest paths.