MDISC REPORT

Group 11 - PROS

DIJKSTRA ALGORITHM

Dijkstra's algorithm is a popular algorithm for finding the shortest path between nodes in a graph and works as follows:

The algorithm keeps track of a set of visited vertices and a set of unvisited vertices. It begins at the source vertex and iteratively chooses the unvisited vertex with the smallest tentative distance from the source. Then, it explores the neighbors of this vertex and adjusts their tentative distances if a shorter path is discovered. This cycle persists until the destination vertex is reached, or all reachable vertices have been explored.

Algorithm implementation:

1. **Initialization**: In the dijkstra method, an array shortestDistances is initialized with maximum integer values and an array parents is initialized to keep track of the parent node for each node in the shortest path tree. The shortestDistances for the source node is set to 0 and its parent is set to -1.

```
int n = graph.length;
int[] shortestDistances = new int[n];
boolean[] added = new boolean[n];
for (int vertexIndex = 0; vertexIndex < n; vertexIndex++) {
    shortestDistances[vertexIndex] = Integer.MAX_VALUE;
    added[vertexIndex] = false;
}
shortestDistances[source] = 0;
int[] parents = new int[n];
parents[source] = -1;
```

2. **Visit the Neighbors**: The algorithm then enters a loop that runs for n-1 times where n is the number of nodes. In each iteration, it finds the node with the minimum distance value, from the set of nodes not yet included in the shortest path tree. It calculates the tentative distance to each unvisited neighbor of this node and updates the shortestDistances and parents arrays if a shorter path is found.

```
for (int i = 1; i < n; i++) {
    int nearestVertex = -1;
    int shortestDistance = Integer.MAX_VALUE;
    for (int vertexIndex = 0; vertexIndex < n; vertexIndex++)
{
        if (!added[vertexIndex] &&
shortestDistances[vertexIndex] < shortestDistance) {
            nearestVertex = vertexIndex;
            shortestDistance = shortestDistances[vertexIndex];
        }
```

```
    }
    added[nearestVertex] = true;
    for (int vertexIndex = 0; vertexIndex < n; vertexIndex++)
{
        int edgeDistance = graph[nearestVertex][vertexIndex];
        if (edgeDistance > 0 && ((shortestDistance +
edgeDistance) < shortestDistances[vertexIndex])) {
            parents[vertexIndex] = nearestVertex;
            shortestDistances[vertexIndex] = shortestDistance
+ edgeDistance;
        }
    }
}
```

3. **Update the Current Node**: After considering all unvisited neighbors of the current node, the node is marked as visited. A visited node will not be checked again; its distance recorded now is final and minimal.

```
added[nearestVertex] = true;
```

4. **Select the Next Node**: The next node to be visited is the one with the smallest distance value that hasn't been visited yet. The process is repeated until all nodes have been visited.

```
int nearestVertex = -1;
int shortestDistance = Integer.MAX_VALUE;
for (int vertexIndex = 0; vertexIndex < n; vertexIndex++) {
    if (!added[vertexIndex] && shortestDistances[vertexIndex]
< shortestDistance) {
        nearestVertex = vertexIndex;
        shortestDistance = shortestDistances[vertexIndex];
    }
}
```

5. **Repeat**: The algorithm continues until it has found the shortest path to all nodes or, in a more optimized version, until it has found the shortest path to a specific node.The algorithm continues until it has found the shortest path to all nodes or, in a more optimized version, until it has found the shortest path to a specific node.

In the findShortestPathToAP method of the EvacuationSigns class, the Dijkstra's algorithm is used to find the shortest path from each node to the nearest assembly point. The dijkstra method is called for each node and the shortest path and its cost are printed out. The printShortestPath method is used to recursively print the shortest path from a given node to the source node by following the parent nodes in the parents array.

```
public void findShortestPathToAP(String matrixFileName) {
    if (assemblyPoints.isEmpty()) {
        System.out.println("No assembly points found");
```

```
            return;
        }
        for (int i = 0; i < locationNames.length; i++) {
            int nearestAPIndex = -1;
            int shortestDistance = Integer.MAX_VALUE;
            for (String ap : assemblyPoints) {
                int apIndex = -1;
                for (int j = 0; j < locationNames.length; j++) {
                    if (locationNames[j].equals(ap)) {
                        apIndex = j;
                        break;
                    }
                }
                if (apIndex == -1) {
                    System.out.println("AP " + ap + " not found");
                    continue;
                }
                DijkstraResult result = dijkstra(i);
                int[] shortestDistances = result.getShortestDistances();
                if (shortestDistances[apIndex] < shortestDistance) {
                    shortestDistance = shortestDistances[apIndex];
                    nearestAPIndex = apIndex;
                }
            }
            if (nearestAPIndex != -1) {
                DijkstraResult result = dijkstra(i);
                int[] parents = result.getParents();
                System.out.print("Path from " + locationNames[i] + " to
" + locationNames[nearestAPIndex] + ": ");
                printShortestPath(i, nearestAPIndex, parents);
                System.out.println(" with cost " + shortestDistance);

writePathsToCSV("src/main/java/pt/ipp/isep/dei/mdisc/database/input/
paths.csv", matrixFileName);
            }
        }
}
```

CLASSES

1. The code for the application consists in two classes: EvacuationSigns and DijkstraResult.

The EvacuationSigns class is the main class that handles the logic of the application. It reads data from CSV files calculates shortest path between locations using Dijkstra's algorithm, and visulizes the graph of locations.

- printShortestPath method

This method is used to print the shortest path from the start vertex to the end vertex. It uses the parents array to backtrack from the end vertex to the start vertex.

```
public void printShortestPath(int startVertex, int endVertex, int[]
parents) {
    if (startVertex == endVertex) {
        System.out.print(locationNames[startVertex]);
    } else {
        printShortestPath(startVertex, parents[endVertex], parents);
        System.out.print(" -> " + locationNames[endVertex]);
    }
}
```

- findShortestPathToAP method

This method finds the shortest path from each node to the nearest assembly point using the Dijkstra's algorithm. It calls the dijkstra method for each node and prints the shortest path and its cost.

```
public void findShortestPathToAP(String matrixFileName) {
    if (assemblyPoints.isEmpty()) {
        System.out.println("No assembly points found");
        return;
    }
    for (int i = 0; i < locationNames.length; i++) {
        int nearestAPIndex = -1;
        int shortestDistance = Integer.MAX_VALUE;
        for (String ap : assemblyPoints) {
            int apIndex = -1;
            for (int j = 0; j < locationNames.length; j++) {
                if (locationNames[j].equals(ap)) {
                    apIndex = j;
                    break;
                }
            }
            if (apIndex == -1) {
                System.out.println("AP " + ap + " not found");
                continue;
            }
            DijkstraResult result = dijkstra(i);
            int[] shortestDistances = result.getShortestDistances();
            if (shortestDistances[apIndex] < shortestDistance) {
                shortestDistance = shortestDistances[apIndex];
                nearestAPIndex = apIndex;
            }
        }
        if (nearestAPIndex != -1) {
            DijkstraResult result = dijkstra(i);
            int[] parents = result.getParents();
            System.out.print("Path from " + locationNames[i] + " to
" + locationNames[nearestAPIndex] + ": ");
            printShortestPath(i, nearestAPIndex, parents);
            System.out.println(" with cost " + shortestDistance);

writePathsToCSV("src/main/java/pt/ipp/isep/dei/mdisc/database/input/
paths.csv", matrixFileName);
        }
    }
}
```

- buildPath method

This method is used to build the path from the start vertex to the end vertex. It uses the parents array to backtrack from the end vertex to the start vertex and appends each node to the path.

```
private void buildPath(int startVertex, int endVertex, int[]
parents, StringBuilder path) {
    if (startVertex == endVertex) {
        path.append(locationNames[startVertex]);
    } else {
        buildPath(startVertex, parents[endVertex], parents, path);
        path.append(",").append(locationNames[endVertex]);
    }
}
```

- calculatePathToAP method

This method calculates the shortest path from the start point to the nearest assembly point using the Dijkstra's algorithm. It calls the dijkstra method and uses the buildPath method to construct the path.

```
public String[] calculatePathToAP(String startPoint) {
    int startIndex = -1;
    for (int i = 0; i < locationNames.length; i++) {
        if (locationNames[i].equals(startPoint)) {
            startIndex = i;
            break;
        }
    }
    if (startIndex == -1) {
        System.out.println("Start point not found");
        return null;
    }

    int nearestAPIndex = -1;
    int shortestDistance = Integer.MAX_VALUE;
    for (String ap : assemblyPoints) {
        int apIndex = -1;
        for (int i = 0; i < locationNames.length; i++) {
            if (locationNames[i].equals(ap)) {
                apIndex = i;
                break;
            }
        }
        if (apIndex == -1) {
            System.out.println("AP " + ap + " not found");
            continue;
        }

        DijkstraResult result = dijkstra(startIndex);
        int[] shortestDistances = result.getShortestDistances();

        if (shortestDistances[apIndex] < shortestDistance) {
            shortestDistance = shortestDistances[apIndex];
            nearestAPIndex = apIndex;
        }
```

```
    }

    if (nearestAPIndex == -1) {
        System.out.println("No AP found");
        return null;
    }

    DijkstraResult result = dijkstra(startIndex);
    int[] parents = result.getParents();
    StringBuilder path = new StringBuilder();
    buildPath(startIndex, nearestAPIndex, parents, path);

    String[] pathNodes = path.toString().split(",");
    return pathNodes;
}
```

- getCostBetweenPlaces method

This method returns the cost between two places. It is used to get the cost of the edge between two nodes in the graph.

```
public int getCostBetweenPlaces(String place1, String place2) {
    int index1 = -1;
    int index2 = -1;
    for (int i = 0; i < locationNames.length; i++) {
        if (locationNames[i].equals(place1)) {
            index1 = i;
        }
        if (locationNames[i].equals(place2)) {
            index2 = i;
        }
    }
    if (index1 == -1 || index2 == -1) {
        System.out.println("One or both places not found");
        return -1;
    }
    return graph[index1][index2];
}
```

2. The DijkstraResult class is a simple data class that stores the result of Dijkstra's algorithm. It contains two arrays: parents[] and shortestDistance[].

```
public class DijkstraResult {
    private int[] parents;
    private int[] shortestDistances;

    public DijkstraResult(int[] parents, int[] shortestDistances) {
        this.parents = parents;
        this.shortestDistances = shortestDistances;
    }

    public int[] getParents() {
        return parents;
    }
```

```
    public int[] getShortestDistances() {
        return shortestDistances;
    }
}
```

- The parents[] array stores the parent of each location in the shortest path tree
- The shortestDistance[] array stores the shortest distance from the source to each location.

Conclusion

This report elucidates the algorithm's iterative process, efficiently traversing vertices and continually adjusting distances and parent nodes until optimal paths are established. Through the included code snippets, the report illustrates how Dijkstra's algorithm is applied within the context of emergency evacuation route planning.

Specifically, it demonstrates the algorithm's utility in identifying the shortest paths from various locations to the nearest assembly points. The EvacuationSigns class serves as the central orchestrator, employing key methods such as printShortestPath, findShortestPathToAP, and calculatePathToAP to streamline pathfinding and visualization tasks. Additionally, the DijkstraResult class serves to encapsulate the algorithm's outcomes, storing vital data such as parent nodes and shortest distances for future analysis.

In essence, through this report, we understand why Dijkstra's algorithm is such a powerful and versatile tool for resolving graph-related challenges, particularly in critical scenarios like emergency evacuation planning.

GROUP 011 - PROS
Afonso Marques – 1221018
Ana Filipa Alves – 1230929
Gonçalo Fernandes – 1231170
Marisa Afonso – 1231151
Ricardo Morim – 1230481