Instituto Superior de Engenharia do Porto

Licenciatura em Engenharia Informática

MDISC – 2023/2024

Report Summary

Analysing the algorithm and results

Authors:

1191330 Luigy Lima

1170499 Daniel Silva

1191377 Tomás Pereira

1200356 Diogo Almeida

Class: 1DB **Group:** 22

Date: 08/06/2024

Lecturer: Alexandra Antunes Gavina

Dijkstra's algorithm(Method):

methodToReplaceSize:

The method methodToReplaceSize counts non-null elements in a list of SignalPoint objects. It initializes a counter and iterates through the list, incrementing the counter for each non-null element. The loop continues until it encounters a null or goes out of the list bounds, which triggers an IndexOutOfBoundsException. When this exception occurs, the method returns the count of non-null elements. If the loop exits normally, it returns -1, although this scenario is unlikely due to the infinite loop structure.

```
3 usages  1191330
private int methodToReplaceSize(List<SignalPoint> vertices) {
    int count = 0;

    try {
        while (true) {
            if (vertices.get(count) != null) {
                count++;
                continue;
            }
            break;
        }
    } catch (IndexOutOfBoundsException io) {
        return count;

    }
    return -1;
}
```

methodToReplaceSizeRoute:

This method does the same as methodToReplaceSize but uses a list of Rout instead of SignalPoint.

```
private int methodToReplaceSizeRoute(List<Route> route) {
    int count = 0;

    try {
        while (true) {
            if (route.get(count) != null) {
                count++;
                continue;
            }
            break;
        }
    } catch (IndexOutOfBoundsException io) {
        return count;

    }
    return -1;
}
```

methodToReplaceIndexOf:

The methodToReplaceIndexOf function finds the index of a given SignalPoint object within a list of SignalPoint objects by comparing their names. It first determines the list size, iterates through the list, and returns the index of the matching object if found, otherwise, it returns -1.

```
private int methodToReplaceIndexOf(List<SignalPoint> vertices, SignalPoint signalPoint) {
    int size = methodToReplaceSize(vertices);

    for (int i = 0; i < size; i++) {
        if (vertices.get(i).getName().equals(signalPoint.getName())) {
            return i;
        }
    }
    return -1;
}
```

importNamesFromCSV:

The importNamesFromCSV function reads a CSV file specified by filePath, converts each line into a SignalPoint object, and stores them in a list. It then returns this list of SignalPoint objects.

```
public List<SignalPoint> importNamesFromCSV(String filePath) {

    List<SignalPoint> listaNomes = new ArrayList<>();

    try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
        String linha;

        while ((linha = br.readLine()) != null) {
            String[] partes = linha.split( regex: ";");
            for (String parte : partes) {
                listaNomes.add(new SignalPoint(parte));
            }
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
    return listaNomes;
}
```

importRouteFromCSV:

The importRouteFromCSV function reads data from a CSV file and creates Route objects based on the information. It uses a list of SignalPoint objects to determine the route connections. Finally, it returns the list of created Route objects.

```
public List<Route> importRouteFromCSV(List<SignalPoint> signalPoints, String filePath) {
    List<Route> routeList = new ArrayList<>();

    try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
        String line;
        int l = 0;

        while ((line = br.readLine()) != null) {

            String[] parts = line.split( regex: ";");
            int c = 0;
            for (String part : parts) {
                try {
                    int number = Integer.parseInt(part);
                    if (c != l && number != 0) {
                        routeList.add(new Route(number, signalPoints.get(l), signalPoints.get(c)));
                    }
                } catch (NumberFormatException e) {
                }
                ++c;
            }

            ++l;
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
    return routeList;
}
```

findShortestPath:

The findShortestPath function calculates the shortest path between a source and a target SignalPoint. It uses Dijkstra's algorithm to efficiently find the path. It iterates through points, updating distances and predecessors until the shortest path to all points is determined. Then, it reconstructs the path from the source to the target and returns it as a list of Route objects representing the shortest route. If no path is found, it returns an empty list.

```java
public List<Route> findShortestPath(SignalPoint source, SignalPoint target, List<SignalPoint> signalPoints, List<Route> routes) {
    // Número total de pontos de sinalização
    int numPoints = methodToReplaceSize(signalPoints);

    // Arrays para armazenar distâncias, pontos visitados e predecessores
    int[] distances = new int[numPoints];
    boolean[] visited = new boolean[numPoints];
    int[] previous = new int[numPoints];

    // Inicializar arrays de distâncias e visitados
    for (int i = 0; i < numPoints; i++) {
        distances[i] = Integer.MAX_VALUE; // Inicialmente, todas as distâncias são definidas como infinito
        visited[i] = false; // Nenhum ponto foi visitado inicialmente
        previous[i] = -1; // Inicialmente, não há predecessores para nenhum ponto
    }

    // Índice do ponto de origem
    int sourceIndex = methodToReplaceIndexOf(signalPoints, source);
    distances[sourceIndex] = 0; // A distância do ponto de origem para si mesmo é zero

    // Iterar até encontrar o caminho mais curto para todos os pontos ou até que todos os pontos sejam visitados
    for (int i = 0; i < numPoints - 1; i++) {
        // Encontrar o ponto não visitado mais próximo
        int closest = -1; // Índice do ponto mais próximo
        int closestDistance = Integer.MAX_VALUE; // Distância do ponto mais próximo
        for (int j = 0; j < numPoints; j++) {
            // Verificar se o ponto não foi visitado e se sua distância é menor que a distância mais próxima atual
            if (!visited[j] && distances[j] < closestDistance) {
                closest = j;
                closestDistance = distances[j];
            }
        }

        // Se não houver ponto acessível a partir do ponto de origem, sair do loop
        if (closest == -1) {
            break;
        }
```

```
    // Marcar o ponto escolhido como visitado
    visited[closest] = true;
    int sizeRoute = methodToReplaceSizeRoute(routes);

    // Atualizar as distâncias para os pontos vizinhos do ponto escolhido
    for (int j = 0; j < sizeRoute; j++) {

        int fromIndex = methodToReplaceIndexOf(signalPoints, routes.get(j).getS1());
        int toIndex = methodToReplaceIndexOf(signalPoints, routes.get(j).getS2());
        // Verificar se o ponto atual é o ponto de origem da rota e se o destino não foi visitado ainda
        if (fromIndex == closest && !visited[toIndex]) {
            // Calcular a nova distância
            int newDist = distances[closest] + routes.get(j).getDistance();
            // Atualizar a distância se a nova distância for menor
            if (newDist < distances[toIndex]) {
                distances[toIndex] = newDist;
                previous[toIndex] = closest; // Atualizar o predecessor
            }
            // Verificar se o ponto atual é o destino da rota e se a origem não foi visitada ainda
        } else if (toIndex == closest && !visited[fromIndex]) {
            // Calcular a nova distância
            int newDist = distances[closest] + routes.get(j).getDistance();
            // Atualizar a distância se a nova distância for menor
            if (newDist < distances[fromIndex]) {
                distances[fromIndex] = newDist;
                previous[fromIndex] = closest; // Atualizar o predecessor
            }
        }
    }

}

// Reconstruir o caminho mais curto
List<SignalPoint> path = new ArrayList<>();
int atIndex = methodToReplaceIndexOf(signalPoints, target);
for (int at = atIndex; at != -1; at = previous[at]) {
    path.add( index: 0, signalPoints.get(at));
}
```

```
    // Se o ponto de origem não estiver no inicio, retornar um caminho vazio (nenhum caminho encontrado)
    if (path.isEmpty() || path.get(0) != source) {
        return new ArrayList<>();
    }

    // Construir a rota com base no caminho encontrado
    return constructRoute(path, routes);
}
```

constructRoute:

The constructRoute function creates a new route based on a list of SignalPoint objects and existing routes. It iterates through each pair of adjacent signal points and searches for a corresponding route in the list of routes. If a matching route is found, it creates a new route object using the distance and signal points of the found route and adds it to the list of new routes. Finally, it returns the list of new routes.

```
private List<Route> constructRoute(List<SignalPoint> signalPoints, List<Route> routes) {
    List<Route> newRoute = new ArrayList<>();

    int sizeSignalPoint = methodToReplaceSize(signalPoints);
    int sizeRoute = methodToReplaceSizeRoute(routes);
    for (int i = 0; i < sizeSignalPoint - 1; i++) {

        for (int j = 0; j < sizeRoute; j++) {
            if (routes.get(j).equals(new Route(signalPoints.get(i), signalPoints.get(i + 1)))) {
                newRoute.add(new Route(routes.get(j).getDistance(), signalPoints.get(i), signalPoints.get(i + 1)));
            }
        }
    }

    return newRoute;
}
```

totalDistance:

The totalDistance function calculates the total distance of a list of routes. It iterates through each route in the list and adds its distance to a running total. Finally, it returns the total distance as an integer value.

```
public int totalDistance(List<Route> routes) {
    int total = 0;
    for (Route route : routes) {
        total += route.getDistance();
    }

    return total;
}
```

Input and Output Methods

generateSubgraphCSV:

This method generates CSV content representing a subgraph.
It constructs CSV content by appending vertices, edges, and their costs to a StringBuilder object.

```
public String generateSubgraphCSV(List<Route> shortestPath) {
    StringBuilder csvContent = new StringBuilder();

    // Append edges and calculate total cost
    for (int i = 0; i < shortestPath.size(); i++) {
        if (i == shortestPath.size() - 1) {
            csvContent.append(shortestPath.get(i).getS1().getName()).append(",")
                    .append(shortestPath.get(i).getS2().getName()).append(";");
        } else {
            csvContent.append(shortestPath.get(i).getS1().getName()).append(",");
        }
    }

    // Append total cost
    csvContent.append("Total Cost:").append(totalDistance(shortestPath));

    return csvContent.toString(); // Return the CSV content
}
```

writeCSVToFile:

This method writes CSV content to a file.
It takes the CSV content and writes it to the specified file path.

```
1 usage  new *
public void writeCSVToFile(String csvContent, String filePath) {
    String csvFilePath = filePath + File.separator + "output_subgraph.csv";

    try (BufferedWriter writer = new BufferedWriter(new FileWriter(csvFilePath))) {
        writer.write(csvContent);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

generateAllSubgraphCSV:

The generateAllSubgraphCSV function creates CSV content representing a subgraph based on the provided shortest path. It iterates through each route in the shortest path, appending the names of the signal points to the CSV content. Finally, it returns the generated CSV content as a string.

```
public String generateAllSubgraphCSV(String content, List<Route> shortestPath) {

    StringBuilder csvContent = new StringBuilder();
    StringBuilder contentBuilder = new StringBuilder(content);

    // Anexar arestas
    for (int i = 0; i < shortestPath.size(); i++) {

        if (i == shortestPath.size() - 1) {
            contentBuilder.append(csvContent.append(shortestPath.get(i).getS1().getName()).append(",")
                    .append(shortestPath.get(i).getS2().getName()).append(";"));
            csvContent.delete(0, csvContent.length());
        } else {
            contentBuilder.append(csvContent.append(shortestPath.get(i).getS1().getName()).append(","));
            csvContent.delete(0, csvContent.length());

        }
    }

    // Anexar custo total e calcular o custo total
    content = contentBuilder.append("Total Cost: ").append(totalDistance(shortestPath)).append("\n").toString();

    return content;
}
```

displayAllPath:

The displayAllPath method orchestrates the display of all paths. It first imports signal point names and routes from CSV files. Then, it iterates through each signal point, excluding the access point ("AP"). For each signal point, it finds the shortest path to the access point, generates CSV content representing the subgraph, and attempts to visualize the graph. Finally, it writes the CSV content to an output file.

```
private void displayAllPath() {
    List<SignalPoint> listPointNames = getController().importNamesFromCSV(FILE_PATH_NAMES);
    List<Route> listRoutes = getController().importRouteFromCSV(listPointNames, FILE_PATH_MATRIX);

    String content = "";
    for (SignalPoint ls : listPointNames) {
        if (!ls.equals(new SignalPoint( name: "AP"))) {
            List<Route> paths = getController().findShortestPath(ls, listPointNames.get(listPointNames.indexOf(new SignalPoint( name: "AP"))), listPointNames, listRoutes);
            content = getController().generateAllSubgraphCSV(content, paths);
            try {
                getController().visualizeGraph(paths, title: "Output_Graph_" + ls.getName(), FILE_PATH_OUTPUT_IMG);
            } catch (IOException ignored) {

            }
        }
    }


    getController().writeCSVToFile(content, FILE_PATH_OUTPUT);
}
```

displayOnePath:

The displayOnePath method finds and displays the shortest path from a specified signal point to the access point. It retrieves input data, imports signal point names and routes, finds the shortest path, prints the path details and total distance, generates a CSV file representing the subgraph, and visualizes both the input graph and the output subgraph. Any exceptions encountered are rethrown as a RuntimeException.

```
private void displayOnePath() {
    try {
        //requestData();
        requestDataInput();
        List<SignalPoint> listPointNames = getController().importNamesFromCSV(FILE_PATH_NAMES);
        List<Route> listRoutes = getController().importRouteFromCSV(listPointNames, FILE_PATH_MATRIX);
        List<Route> paths = getController().findShortestPath(listPointNames.get(listPointNames.indexOf(signalPoint)),
                listPointNames.get(listPointNames.indexOf(new SignalPoint( name: "AP"))), listPointNames, listRoutes);

        System.out.println("\n\n--- Path -------------");
        for (Route routes : paths) {
            System.out.println(routes.toString());
        }
        System.out.printf("\nTotal Distance: %d%n", getController().totalDistance(paths));


        getController().writeCSVToFile(getController().generateSubgraphCSV(paths), FILE_PATH_OUTPUT);


        getController().visualizeGraph(listRoutes, title: "Input_Graph", FILE_PATH);

        // Visualize the output subgraph
        getController().visualizeGraph(paths, title: "Output_Subgraph", FILE_PATH);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Results (Display One Graph)

Console Result:

- Route, this is route of points that leads to an Assembly point
- Total Distance, this is the total distance to the Assembly point from a Point

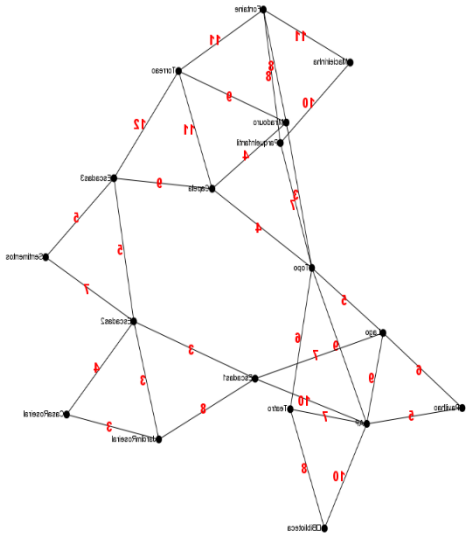
The print below is the result for Escadas1:

```
--- Path -------------
Route{distance=10, s1=SignalPoint{name='Escadas1'}, s2=SignalPoint{name='AP'}}

Total Distance: 10
```

Normal graph and minimum spanning three

Input Graph



output Graph

CSV File Infomation export:

Escadas1,AP	Total Cost:10

Results (Display All Graphs)

Console Result:

For this choice there isn't any information displayed in the console.

CSV File Information Export

Biblioteca,AP	Total Cost: 10
Teatro,AP	Total Cost: 7
Pavilhao,AP	Total Cost: 5
Topo,AP	Total Cost: 9
ParqueInfantil,Topo,AP	Total Cost: 16
Macieirinha,ParqueInfantil,Topo,AP	Total Cost: 26
Fontaine,Miradouro,Topo,AP	Total Cost: 20
Miradouro,Topo,AP	Total Cost: 12
Capela,Topo,AP	Total Cost: 13
Torreao,Miradouro,Topo,AP	Total Cost: 21
Lago,AP	Total Cost: 9
Escadas3,Escadas2,Escadas1,AP	Total Cost: 18
Sentimentos,Escadas2,Escadas1,AP	Total Cost: 20
CasaRoseiral,Escadas2,Escadas1,AP	Total Cost: 17
JardimRoseiral,Escadas2,Escadas1,AP	Total Cost: 16
Escadas2,Escadas1,AP	Total Cost: 13
Escadas1,AP	Total Cost: 10

One of the Graphs: