ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

26 DE NOVEMBRO DE 2023

RELATÓRIO ESINF

ANÁLISE DE COMPLEXIDADE DAS USER STORIES

1210701 - Miguel Ferreira

1220607 - Gonçalo Pombo

1221223 - Diogo Martins

1221349 - Gustavo Lima

Turma: 2DC

Índice

USEI 01

Construir a rede de distribuição de cabazes a partir dos ficheiros com o formato disponibilizado.

A base de todo a rede de distribuição é o *FileReadingSystem*, que foi criado com o intuito de controlar a leitura de qualquer ficheiro, seja este um ficheiro *.csv*, *.txt*, ou uma folha de cálculo.

Como estamos a tratar de um ficheiro *.csv*, que foi o que nos foi fornecido pelos docentes da cadeira de Estruturas de Informação, vamos apenas analisar a complexidade do ficheiro *CSVReadingSystem*.

```java
public short countFileLines() throws FileNotFoundException {
    /**
     * We want to verify if the file was really
     * initializes and is ready to be used
     */
    if(this.file.exists() && !this.file.isFile())
        throw new IllegalArgumentException("You must first provide a valid file.");

    Scanner read = new Scanner(this.file);
    short counter = 0;

    while (read.hasNextLine())
        if(!read.nextLine().isEmpty()) counter++;

    read.close();

    return counter;
}
```

Figura 1 - Função countFileLines

Analisando este método de contagem de linhas (figura 1), que é usado logo no construtor, temos uma complexidade O(n), já que estamos a fazer um simples contador para todas as linhas do ficheiro.

De seguida, temos o método principal da função, o *readData* (figura 2), que vai analisar todas as linhas do ficheiro e separá-las numa *ArrayList<ArrayList<String>>*. Todas as linhas vão estar divididas na *ArrayList* que, por sua vez, vão ser divididas em *Strings* mais pequenas.

```java
@Override
public ArrayList<ArrayList<String>> readData() throws FileNotFoundException {
    String dataChunk;
    String processedData;

    int currentLine = 0;

    if(this.file.exists() && !this.file.isFile())
        throw new IllegalArgumentException("You must first provide a valid file.");

    /** If the file is not initialized correctly, then ...*/
    if(lines == -1)
        throw new IllegalArgumentException("You must first provide a valid file.");

    ArrayList<ArrayList<String>> data = new ArrayList<>();

    for (int i = this.startingLine; i <= this.lines; i++) {
        dataChunk = this.findLine(i);

        /** Sometimes the file may present some empty ...*/
        if(this.isEmpty(dataChunk))
            continue;

        processedData = this.processData(dataChunk);

        while (data.size() <= currentLine)
            data.add(new ArrayList<>());

        /** We're going to loop through each string chunk in a line ...*/
        for (String parameter : processedData.split( regex: ",")) {
            data.get(currentLine).add(parameter.trim());
        }

        currentLine++;
    }

    return data;
}
```

Figura 2 - Função readData

Aqui nesta função temos dois ciclos *for* principais, um para percorrer todas as linhas do ficheiro, e outro para percorrer todos os elementos da linha que foi obtida.

Posto isto, visto que temos dois ciclos com tamanhos finais diferentes, temos uma complexidade de O (n * m), onde n corresponde ao número de linhas do ficheiro e m o número de colunas de uma linha.

Depois de lidas todas as linhas, e inseridos todos os dados, vamos apenas retornar o *ArrayList* que criamos dentro desta função.

Agora que já analisamos a função de *readData*, vamos passar para a classe que trata da conversão de dados para as estruturas necessárias: *DistributionDataToModel*.

Esta classe, no seu construtor, começa por chamar a função *readData* para os ficheiros necessários e, de seguida, chama as funções que vão construir as estruturas (*insertLocalities, insertDistances*).

```java
public DistributionDataToModel(String locationsFile, String distancesFile) throws IOException {
    FileReadingSystem locationsReadingSystem = new FileReadingSystem(locationsFile,  startingLine: 2);
    FileReadingSystem distancesReadingSystem = new FileReadingSystem(distancesFile,  startingLine: 2);

    this.locationsData = locationsReadingSystem.readData();
    this.distancesData = distancesReadingSystem.readData();

    this.insertLocalities();
    this.insertDistances();
}
```

Figura 3 - Função construtora da classe DistributionDataToModel

Como é possível ver pela imagem abaixo, a função *insertLocalities* (figura 4) vai ter uma complexidade O (n), sendo n o número de locais presentes no ficheiro. Até agora o pior caso é O (n * m).

```java
private void insertLocalities() {
    this.locaties = new ArrayList<>();

    for(ArrayList<String> data : locationsData)
        this.locaties.add(
            new Locality(
                data.get(0),
                Double.parseDouble(data.get(1)),
                Double.parseDouble(data.get(2))
            )
        );
}
```

Figura 4 - Função insertLocalities

Depois de inserir os locais numa *ArrayList<Locality>*, vamos passar para o passo onde vamos inserir os nossos *Hubs* e as suas distâncias.

```java
private void insertDistances() {
    this.distances = new MapGraph<>( directed: false);

    Locality origin;
    Locality destiny;

    for(ArrayList<String> data : this.distancesData) {
        origin = localityByName(data.get(0));
        destiny = localityByName(data.get(1));

        /**
         * Since there's the chance that the vertexes
         * already exist, then we're going to
         * verify it
         */
        if(!vertexExists(origin))
            this.distances.addVertex(origin);

        if(!vertexExists(destiny))
            this.distances.addVertex(destiny);

        this.distances.addEdge(
            origin,
            destiny,
            Double.parseDouble( data.get(2))
        );
    }
}
```

Figura 5 - Função insertDistances

Na função *insertDistances* (figura 5), começamos por fazer um ciclo que percorre todas as distâncias, n, e de seguida, vamos obter o objeto *Locality* através do seu nome, que gera outro ciclo for por m elementos.

Ou seja, após esta análise, podemos concluir que a complexidade deste algoritmo de inserção nas estruturas de dados corretas, tem complexidade O (n * m).

USEI 02

Determinar os vértices ideais para a localização de N *hubs* de modo a otimizar a rede de distribuição segundo diferentes critérios.

Nesta *user story* foi-nos pedido para através de três critérios determinar os vértices ideais de uma grafo, as localidades, para a localização de, um número determinado pelo utilizador, de *hubs* de modo a otimizar a rede de distribuição.

```java
public List<Locality> orderByAllCriteria(MapGraph<Locality, Double> mapGraph, int numberOfHubs){
    List<Locality> localities = new ArrayList<>(mapGraph.vertices());

    Comparator<Locality> influenceComparator = new Comparator<Locality>() {
        @Override
        public int compare(Locality locality1, Locality locality2) {
            int comparision = Integer.compare(mapGraph.inDegree(locality1), mapGraph.inDegree(locality2));

            if (comparision == 0)
                comparision = Double.compare(calculateAverageDistance(mapGraph, locality1), calculateAverageDistance(mapGraph, locality2));

            if (comparision == 0)
                comparision = Integer.compare(calculateCentrality(mapGraph, locality1), calculateCentrality(mapGraph, locality2));

            return comparision;
        }
    };

    localities.sort(influenceComparator);
    Collections.reverse(localities);

    List<Locality> selectedHubs = localities.subList(0, Math.min(numberOfHubs, localities.size()));
    assignCollaborators(selectedHubs);
    setHubOperatingHours(selectedHubs);

    return selectedHubs;
}
```

Figura 6 - Função orderByAllCriteria

Em primeiro lugar, começamos por organizar todos os vértices segundo o critério de influência. No caso de a comparação vir a ‘0’, ou seja, têm o mesmo número de vértices comparamos segundo o critério de proximidade que chama o método *calculateAverageDistance* (figura 7) que vai ser explicado já a seguir, no caso da comparação deste também ser ‘0’, comparamos segundo o último critério, o de centralidade, que que com ajuda do método *calculateCentrality* (figura 8) faz a última ordenação deste algoritmo.

Para fazer estas comparações isso foi criado um *Comparator* que compara todos os vértices e através do método *sort* do java ordenamos os vértices por ordem crescente sendo depois necessário reverter a ordem para que assim estejam por ordem decrescente (*Collections.reverse*). Depois de organizados bastou colocar em uma lista os n vértices primeiros vértices.

Quanto à complexidade, este método, tem complexidade O (V log(V) (V ((V + E) * log(V)))), sendo 'V' o número de vértices e 'E' o número de ramos, visto que as únicas coisas que contribuem para a sua complexidade é o método *sort* do java que tem complexidade O (V log(V)) e os método que são chamados no cálculo dos critérios, visto que, os métodos que têm maior complexidade são os métodos auxiliares no cálculo dos critérios de proximidade e centralidade, que têm de complexidade O (V ((V + E) * log(V))), como vai ser explicado já a seguir.

```java
private static double calculateAverageDistance(MapGraph<Locality, Double> mapGraph, Locality locality) {
    double totalDistance = 0;
    int numVertices = mapGraph.numVertices() - 1;

    for(Locality targetLocality : mapGraph.vertices()) {
        if(!locality.equals(targetLocality)){
            LinkedList<Locality> shortestPath = new LinkedList<>();

            Double distance = Algorithms.shortestPath(mapGraph, locality, targetLocality, Comparator.naturalOrder(), Double::sum, zero: 0.0, shortestPath);

            if (distance != null)
                totalDistance += distance;
        }
    }

    double averageDistance = totalDistance / numVertices;

    return averageDistance;
}
```

Figura 7 - Função calculateAverageDistance

O método, *calculateAverageDistance* (figura 7) auxiliar no cálculo do critério de proximidade, através de um *loop* calcula a média de custo que um vértice tem até chegar a todos os outros, utilizando assim o método *shortestPath*, é feito o mesmo para todos os outros vértices.

Este método tem complexidade O (V ((V + E) * log(V))), visto que dentro de um *loop* O(V) é chamado o método *shortestPath* com complexidade O ((V + E)*log(V)), pois este por outro lado chama o método *shortestPathDijkstra* que tem essa mesma complexidade.

```java
private static int calculateCentrality(MapGraph<Locality, Double> mapGraph, Locality locality) {
    int centrality = 0;
    for (Locality targetLocality : mapGraph.vertices()) {

        if (!locality.equals(targetLocality) && Algorithms.shortestPath(mapGraph, locality, targetLocality, Comparator.naturalOrder(),
                Double::sum,  zero: 0.0, new LinkedList<>()) != null) {
            centrality++;

        }
    }


    return centrality;
}
```

Figura 8 - Função calculateCentrality

O método, *calculateCentrality* (figura 8) auxiliará no cálculo do critério de centralidade, calcula para cada vértice o número de caminhos mínimos que passam pelo mesmo.

A complexidade deste método será a mesma que o método *calculateAverageDistance* e pelas mesmas razões visto que dentro de um *loop* O(V) é chamado o método *shortestPat'* com complexidade O ((V + E)*log(V)), pois este por outro lado chama o método *shortestPathDijkstra* que tem essa mesma complexidade.

```java
private static void assignCollaborators(List<Locality> hubs) {
    for (Locality hub : hubs)
        hub.setCollaborators(Integer.parseInt(hub.getName().substring( beginIndex: 2)));

}
```

Figura 9 - Método assignCollaborators

```java
private static void setHubOperatingHours(List<Locality> hubs) {
    for (Locality hub : hubs) {
        int hubNumber = Integer.parseInt(hub.getName().substring( beginIndex: 2));
        if (hubNumber <= 105) {
            hub.setOperatingHours("9h:00 - 14h:00");
        } else if (hubNumber <= 215) {
            hub.setOperatingHours("11h:00 - 16h:00");
        } else {
            hub.setOperatingHours("12h:00 - 17h:00");
        }
    }
}
```

Figura 10 - Método setHubOperatingHours

Estes dois penúltimos métodos, *assignCollaborators* (figura 9) e *setHubOperatingHours* (figura 10), contribuem para fazer a atribuição do número de colaboradores e definem o horário de funcionamento do *hub* com base no número da localidade.

A complexidade destes dois método é a mesma visto que ambos só percorrem um *loop*, tendo assim complexidade O (V (log(V))).

Por fim, a complexidade final da classe *HubOptimization* será O (V ((V + E) * log(V))), visto que esta é a maior complexidade que se encontra em todo o código.

USEI 03

Determinar o percurso mínimo possível entre os dois locais mais afastados da rede de distribuição.

O foco desta *user story* é determinar o percurso mínimo possível entre os dois locais mais afastados da rede de distribuição tendo em conta a autonomia (distancia máxima sem precisar de carregar) de um dado veículo, contudo também é pedido os locais de passagem do percurso, os locais onde o veículo carregou, a distância entre os locais do percurso, a distância do percurso e por fim, o número total de carregamento.

Comecei por criar um método para obter os dois locais mais afastados da rede de distribuição (figura 11).

```java
public static <V, E> Map.Entry<V, V> furthestVertices(Graph<V, E> g, Comparator<E> ce, BinaryOperator<E> sum) {
    int numVerts = g.numVertices();
    if (numVerts == 0) {
        return null;
    }

    E[][] dist = (E[][]) new Object[g.numVertices()][g.numVertices()];

    for (int i = 0; i < numVerts; i++) {
        for (int j = 0; j < numVerts; j++) {
            Edge<V, E> e = g.edge(g.vertices().get(i), g.vertices().get(j));
            if (e != null) {
                dist[i][j] = e.getWeight();
            }
        }
    }

    for (int k = 0; k < numVerts; k++) {
        for (int i = 0; i < numVerts; i++) {
            if (i != k && dist[i][k] != null) {
                for (int j = 0; j < numVerts; j++) {
                    if (j != i && j != k && dist[k][j] != null) {
                        E s = sum.apply(dist[i][k], dist[k][j]);
                        if ((dist[i][j] == null) || ce.compare(dist[i][j], s) > 0) {
                            dist[i][j] = s;
                        }
                    }
                }
            }
        }
    }

    E maxDistance = null;
    Map.Entry<V, V> furthestVertices = null;

    for (int i = 0; i < numVerts; i++) {
        for (int j = 0; j < numVerts; j++) {
            if (dist[i][j] != null && (maxDistance == null || ce.compare(dist[i][j], maxDistance) > 0)) {
                maxDistance = dist[i][j];
                furthestVertices = new AbstractMap.SimpleEntry<>(g.vertices().get(i), g.vertices().get(j));
            }
        }
    }

    return furthestVertices;
}
```

Figura 11 - Função furthestVertices

A função começa a criar uma matriz *dist* que representa as distâncias entre todos os pares de vértices no grafo. Para cada par de vértices *(i, j)*, é verificado se há uma aresta entre eles. Se houver uma aresta, o peso da aresta é atribuído à posição correspondente na matriz de distâncias. De maneira a inicializar a matriz de distâncias é usado um for dentro de um for, resultando numa complexidade de tempo O(V^2), onde V é o número de vértices.

Seguidamente é implementada uma versão simplificada do algoritmo de *Floyd-Warshall*, que itera sobre todos os trios de vértices *(i, j, k)* para encontrar distâncias mais curtas. Para cada par de vértices *(i, j)*, verifica se o caminho através de *k* é mais curto do que o caminho direto. Se for, atualiza a matriz de distâncias com a nova distância. Este algoritmo tem complexidade de tempo O(V^3), onde V é o número de vértices, visto que possui um for dentro de um for dentro de um for, ou seja, 3 *for's*.

Após a execução do algoritmo de *Floyd-Warshall*, a função procura pelos vértices mais distantes com base nas distâncias calculadas, iterando sobre todos os pares de vértices, atualizando *maxDistance* e *furthestVertices* sempre que uma distância maior é encontrada. Esta etapa tem complexidade de tempo O(V^2), pois, assim como a inicialização da matriz, é utilizado um for dentro de um for.

O fator dominante na complexidade é o algoritmo de *Floyd-Warshall*, que é O(V^3). Portanto, a complexidade do método *furthestVertices* é O(V^3).

Tendo as duas localidades mais distantes, vou obter o percurso com o auxílio da função *shortestPathElementsWithCharging* (figura 12).

```java
public static <V, E extends Comparable<E>> LinkedList<V> shortestPathElementsWithCharging(Graph<V, E> g, V vOrig, V vDest, Comparator<E> ce, BinaryOperator<E> sum, E zero, double vehicleAutonomy) {

    if (!g.validVertex(vOrig) || !g.validVertex(vDest))
        return null;

    LinkedList<V> shortPath = new LinkedList<>();
    int numVerts = g.numVertices();
    boolean[] visited = new boolean[numVerts];

    @SuppressWarnings("unchecked")
    V[] pathKeys = (V[]) Array.newInstance(vOrig.getClass(), numVerts);

    @SuppressWarnings("unchecked")
    E[] dist = (E[]) Array.newInstance(zero.getClass(), numVerts);

    for (int i = 0; i < numVerts; i++) {
        dist[i] = null;
        pathKeys[i] = null;
    }

    shortestPathDijkstraWithCharging(g, vOrig, ce, sum, zero, visited, pathKeys, dist, vehicleAutonomy);
    E lengthPath = dist[g.key(vDest)];

    if (lengthPath == null)
        return null;

    getPath(g, vOrig, vDest, pathKeys, shortPath);

    return shortPath;
}
```

Figura 12 - Função shortestPathElementsWithCharging

Primeiramente verifica se os vértices de origem e destino são válidos no grafo. Além disso, inicializa a lista que conterá o caminho mais curto, um *array* para guardar os vértices visitados, um para armazenar os predecessores durante o caminho mais curto e outro para armazenar as distâncias dos vértices à origem.

A seguir, chama a função que implementa uma versão do algoritmo de *Dijkstra* em que é considerada a autonomia da viatura. Esta função preenche os *arrays*, *pathKeys* e *dist* com os predecessores e distâncias mínimas.

Depois, é chamada a função *getPath* para reconstruir o caminho mais curto a partir dos predecessores armazenados, preenchendo a lista *shortPath* com os vértices do caminho.

Finalmente, é retornada a lista *shortPath* que contém o caminho mais curto de *vOrig* para *vDest* considerando a autonomia do veículo.

A criação de estruturas de dados, como *arrays* para distâncias, chaves de caminho e a lista de vértices a complexidade é O (1). Inicializar o *array visited* e outros *arrays* também é feito em tempo constante, pois o tamanho é determinado pelo número de vértices. Portanto, a complexidade é O(V).

A complexidade do algoritmo de *Dijkstra* (figura 13), é dominada pelo número de iterações no *loop* principal. Se considerarmos o número de vértices como V e o número de arestas como E, a complexidade é aproximadamente O ((V + E) * log(V)), onde o log(V) vem da operação de adição e remoção na fila de prioridade. Esta análise é uma estimativa, pois a complexidade exata depende do comportamento específico do grafo.

```java
private static <V, E extends Comparable<E>> void shortestPathDijkstraWithCharging(Graph<V, E> g, V vOrig, Comparator<E> ce, BinaryOperator<E> sum, E zero, boolean[] visited, V[] pathKeys, E[] dist, double vehicleAutonomy) {

    int numVerts = g.numVertices();

    PriorityQueue<AbstractMap.SimpleEntry<V, E>> queue = new PriorityQueue<>(Comparator.comparing(AbstractMap.SimpleEntry::getValue));
    queue.add(new AbstractMap.SimpleEntry<>(vOrig, zero));

    while (!queue.isEmpty()) {
        AbstractMap.SimpleEntry<V, E> current = queue.poll();
        V currentNode = current.getKey();
        E currentDist = current.getValue();

        if (visited[g.key(currentNode)])
            continue;

        visited[g.key(currentNode)] = true;

        for (V neighbor : g.adjVertices(currentNode)) {
            if (!visited[g.key(neighbor)]) {
                E newDist = sum.apply(currentDist, g.edge(currentNode, neighbor).getWeight());

                // If the neighbor is a charging station, reset the remaining autonomy to the vehicle's autonomy
                int neighborKey = g.key(neighbor);
                if (neighborKey >= 0 && neighborKey < dist.length && dist[neighborKey] != null && ce.compare(newDist, dist[neighborKey]) > 0) {
                    newDist.equals(vehicleAutonomy);
                }

                if (dist[g.key(neighbor)] == null || ce.compare(newDist, dist[g.key(neighbor)]) < 0) {
                    dist[g.key(neighbor)] = newDist;
                    pathKeys[g.key(neighbor)] = currentNode;
                    queue.add(new AbstractMap.SimpleEntry<>(neighbor, newDist));
                }
            }
        }
    }
}
```

Figura 13 - Função shortestPathDijkstraWithCharging

A função *getPath* percorre o caminho a partir dos resultados já computados, por isso a sua complexidade é linear, O(V).

Portanto, a complexidade total pode ser aproximadamente expressa como O((V + E) * log(V)), considerando que a parte principal é o algoritmo de *Dijkstra*.

Agora, tendo o percurso, vou calcular a distância entre as localidades do mesmo com o auxilia da função *shortestPath* (figura 14).

```java
public static <V, E> E shortestPath(Graph<V, E> g, V vOrig, V vDest,
                                    Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                    LinkedList<V> shortPath) {

    if (!g.validVertex(vOrig) || !g.validVertex(vDest))
        return null;

    shortPath.clear();
    int numVerts = g.numVertices();
    boolean[] visited = new boolean[numVerts];
    /unchecked/
    V[] pathKeys = (V[]) new Object[numVerts];
    /unchecked/
    E[] dist = (E[]) new Object[numVerts];

    for (int i = 0; i < numVerts; i++) {
        dist[i] = null;
        pathKeys[i] = null;
    }

    shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, pathKeys, dist);
    E lengthPath = dist[g.key(vDest)];

    if (lengthPath == null)
        return null;

    getPath(g, vOrig, vDest, pathKeys, shortPath);

    return lengthPath;
}
```

Figura 14 - Função shortestPath

O método começa por validar os vértices de entrada. De seguida, cria matrizes para monitorizar os vértices visitados, os vértices do percurso e as distâncias. O algoritmo de *Dijkstra* é então utilizado para calcular o caminho mais curto e as distâncias e comprimento do caminho calculado é obtido. Se não existir um caminho válido (quando *lengthPath* é nulo), o método devolve *null*. Caso contrário, os vértices do caminho mais curto são adicionados à lista ligada *shortPath*. Por fim, o comprimento do caminho mais curto é devolvido.

Uma vez que esta função tambem utiliza uma variação do algoritmo de *Dijkstra*, é a complexidade deste que prevalece tendo sido já analisada, portanto a complexidade de *shortestPath* é O ((V + E) * log(V)).

Com isto, podemos concluir que o funcionamento desta *user story* (figura 15) baseia-se nas diferentes versões do algoritmo de *Dijkstra*, e como tal, a complexidade global é O ((V + E) * log(V)), uma vez que é a maior complexidade encontrada ao longo do código.

```java
private void determineVehicleMinimumCourse() {
    Map.Entry<Locality, Locality> furthestPair = Algorithms.furthestVertices(mapGraph, Comparator.naturalOrder(), Double::sum);
    LinkedList<Locality> shortPath = new LinkedList<>();
    List<Double> distances = new LinkedList<>();
    List<Locality> chargings = new ArrayList<>();

    Vehicle vehicle = new Vehicle( autonomy: 300000, averageSpeed: 1, chargingTime: 1);
    double autonomy = vehicle.getAutonomy();
    double currentAutonomy = autonomy;


    Locality origin = furthestPair.getKey();
    Locality destination = furthestPair.getValue();

    System.out.println("\n");
    System.out.printf("Furthest locations are %s and %s \n", origin, destination);
    System.out.printf("Origin is %s \n", origin);
    System.out.printf("Destination is %s \n", destination);
    System.out.println("\n");
    System.out.println("Path between the two furthest locations in the distribution network:");


    LinkedList<Locality> localities = Algorithms.shortestPathElementsWithCharging(this.mapGraph, origin, destination, Comparator.naturalOrder(), Double::sum,  zero: 0.0, autonomy);
    double totalDistance = 0;
    double distanceBetweenLocalities = 0;

    if (localities != null) {
        for (int i = 0; i < localities.size() - 1; i++) {
            distanceBetweenLocalities = Algorithms.shortestPath(this.mapGraph, localities.get(i), localities.get(i + 1), Comparator.naturalOrder(), Double::sum,  zero: 0.0, shortPath);
            System.out.printf("%s -> %s : %.2f Km\n", localities.get(i).toString(), localities.get(i + 1).toString(), distanceBetweenLocalities/1000);
            totalDistance += distanceBetweenLocalities;
            distances.add(distanceBetweenLocalities);

            if (distances.get(i) > currentAutonomy) {
                    chargings.add(localities.get(i));
                    currentAutonomy = autonomy;
            } else {
                    currentAutonomy -= distances.get(i);
            }
        }
    }

    System.out.println("\n");
    System.out.println("Locations where the car was charged:");
    for (int i = 0; i < chargings.size(); i++) {
        System.out.println(chargings.get(i).toString());
    }

    System.out.println("\n");
    int numCharges = chargings.size();
    System.out.printf("Number of charges needed: %d\n", numCharges);
    System.out.printf("Total distance of the path is %.02f km\n",totalDistance/1000);
}
```

Figura 15 - Método determineVehicleMininumCourse

USEI 04

Determinar a rede que liga todas as localidades com uma distância total mínima.

Primeiramente, é importante esclarecer o propósito do exercício e as técnicas que devem ser usadas para o alcançar.

O objetivo é calcular um percurso que ligue todas as localidades no menor custo possível, ou seja, na mínima distância possível e para tal, existem dois algoritmos, o algoritmo de Kruskal e o algoritmo de Prim. Optei pelo algoritmo de Prim devido a este ser mais eficiente e apresentar melhor perfomances para grafos mais densos do que o de Kruskal e ainda ser mais fácil de implementar.

```java
public static MapGraph<Locality, Double> findMinimumConnectionNetwork(MapGraph<Locality, Double> network) {
    Set<Locality> minimumSpanningTree = new LinkedHashSet<>();
    PriorityQueue<Edge<Locality, Double>> priorityQueue = new PriorityQueue<>(Comparator.comparing(Edge::getWeight));

    // Create a copy of the graph received by parameter to build the MST
    MapGraph<Locality, Double> minimumConnectionNetwork = new MapGraph<>(network.isDirected());
```

Figura 16 - Método findMinimumConnectioNetwork

O nosso método *findMinimumConnectionNetwork* recebe um grafo e retorna um *mapgraph* com uma *locality* e o respetivo “peso”.

É criado um *linkedhashset*, para guardarmos os locais que fazem parte da nossa *minimum spanning tree*, uma *priority queue*, que implementa um *comparator* que ordena as arestas em função do valor de cada aresta e, por fim, um *mapgraph* que guarda cada localidade e peso associado.

Quanto à complexidade, a *priority queue* tem alguma importância. Isto porque, a mesma apresenta, como pior cenário, uma complexidade temporal de O (E*log E). Sendo que o E, deve-se ao tempo necessário para a inicializar caso ela tenha E arestas e o log E, o tempo que demora a colocar as arestas por ordem.

```java
// Get whatever vertex to start the algorithm
Locality start = network.vertices().iterator().next();

// Add the vertex we just got to the MST
minimumSpanningTree.add(start);
```

Figura 17 – Escolha do vértice inicial da priority queue

Escolhemos um qualquer vértice dos existentes no grafo para começar e adicionamo-lo à nossa *MST*.

```java
// Add the vertex of starting vertex to the priority queue
for (Edge<Locality, Double> edge : network.outgoingEdges(start)) {
    priorityQueue.add(edge);
}
```

Figura 18 – Initialização da priority queue com o vértice inicial

De seguida, adicionamos todas as arestas associadas a esse mesmo vértice à *priority queue* (figura 18).

De forma a analisar a complexidade temporal, este *for loop* apresenta, tal como a *priority queue*, uma complexidade de O (E * log E). Este valor do E deve-se ao possível número de arestas que sai do vértice adicionado inicialmente, e o log E deve-se ao método add da *priority queue*, que adiciona, mas também ordena os valores, pesando log E.

```java
// Start using Prim's algorithm
while (minimumSpanningTree.size() < network.vertices().size()) {
    Edge<Locality, Double> minEdge = priorityQueue.poll();

    Locality vOrig = minEdge.getVOrig();
    Locality vDest = minEdge.getVDest();

    if (!minimumSpanningTree.contains(vDest)) {
        minimumSpanningTree.add(vDest);
        minimumConnectionNetwork.addVertex(vDest);
        minimumConnectionNetwork.addEdge(vOrig, vDest, minEdge.getWeight());

        for (Edge<Locality, Double> edge : network.outgoingEdges(vDest)) {
            if (!minimumSpanningTree.contains(edge.getVDest())) {
                priorityQueue.add(edge);
            }
        }
    }
}

return minimumConnectionNetwork;
```

Figura 19 – Algoritmo Prim

Agora, é o momento de implementar o algoritmo de Prim (figura 19). Começa com um *while loop*, onde tiramos da nossa *priority queue*, que está ordenada de forma crescente, a primeira aresta e guardamos o respetivo vértice de origem e destino em duas variáveis, *vOrig* e *vDest*, respetivamente.

Depois temos uma condição *if*, que é um dos segredos do algoritmo que, caso o vértice de destino não pertença à *MST*, este é adicionado à mesma e assim, o nosso grafo é construído.

Com isto, continuamos a acrescentar os vértices associados ao novo vértice inserido de forma a continuar o algoritmo.

Para este algoritmo temos uma complexidade temporal considerável. O *while loop* corresponde a O(V-1), sendo V o número de vértices. Já o *for loop* apresenta uma complexidade de O(E), dependendo do número de vértices e dentro ainda tem o método *add* da *priority queue*, que multiplica fazendo com que este *for loop* tenha uma complexidade de O (E * log E). Já o *while loop* acaba por ficar com O((V-1) *E*log E).

Concluindo, este algoritmo de Prim tem uma complexidade de O((V-1) *E*log E), sendo composto por um *while loop*, um *for loop* e um método da *priority queue* que é relevante para esta análise.