

# RELATÓRIO DE ESINF

## *ANÁLISE DE COMPLEXIDADE DAS USER STORIES*

1210701 - Miguel Ferreira

1220607 - Gonçalo Silva

1221223 - Diogo Martins

1221349 - Gustavo Lima

**Turma:** 2DC

## Índice

USEI 06 .....	2
USEI 07 .....	7
USEI 08 .....	10
USEI 09 .....	16


## USEI 06

---

Encontrar para um produtor os diferentes percursos que consegue fazer entre um local de origem e um hub (ou localidade) limitados pelos Kms de autonomia do seu veículo elétrico, ou seja, não considerando carregamentos no percurso.

Para além disso, é pedido a distância e tempo totais de cada percurso. Para obter esses dados, fiz o método `calculateTimeTrip`, que tem uma complexidade constante,  $O(1)$ , uma vez que se trata de uma simples divisão de duas variáveis.

Quanto à distância total, esta é calculada à medida que obtenho as distâncias entre as localidades do percurso, como irei mostrar posteriormente.



```
public static double calculateTimeTrip (Double averageSpeed, Double totalDistance) {  
    return totalDistance/averageSpeed;  
}
```

*Figura 1 - Método `calculateTimeTrip`*

Uma vez que o produtor dá-nos o local de origem e o *hub*, que também é uma localidade, temos que verificar se existem, e para efetuar essa verificação criei este método que percorre os vértices, neste caso as localidades, do grafo e verifica se o nome da localidade dado existe nesse grafo.

Como os vértices do grafo são percorridos com o auxílio de um *loop* for, este método detêm uma complexidade  $O(n)$ , sendo  $n$  o número de vértices.



```
public static Locality localityExists(MapGraph<Locality, Double> mapGraph, String localityName) {  
    for (Locality locality : mapGraph.vertices()) {  
        if (locality.getName().equals(localityName)) {  
            return locality;  
        }  
    }  
    return null;  
}
```

Figura 2 - Método *localityExists*

Uma vez verificada a existência da origem e do destino, podemos obter os diferentes percursos com o seguinte algoritmo.

A complexidade do algoritmo que encontra todos os caminhos de um vértice de origem para um vértice de destino em um grafo, levando em consideração uma autonomia de veículo, pode ser bastante variável.

O *loop* for percorre os vizinhos do vértice atual. No pior caso, para cada vértice, há uma chamada recursiva. A complexidade desse *loop* depende da estrutura do grafo: se o grafo for denso, a complexidade total pode ser maior, se o grafo for esparsamente conectado, a complexidade total pode ser menor.

Portanto, podemos representar a complexidade total como uma função do número de vértices e arestas no grafo, mas a análise exata pode depender da estrutura específica do grafo.

Assumindo que  $V$  é o número de vértices e  $E$  é o número de arestas no grafo, a complexidade do algoritmo pode ser aproximadamente expressa como  $O(V \cdot E)$  no pior caso, devido à exploração de todas as arestas no grafo. Essa é uma estimativa grosseira, e o desempenho real dependerá das características específicas do grafo em questão.

```

public static <V, E extends Comparable<E>> ArrayList<LinkedList<V>> allPathsWithAutonomy(Graph<V, E>
g, V vOrig, V vDest, double vehicleAutonomy) {
    boolean[] visited = new boolean[g.numVertices()];
    ArrayList<LinkedList<V>> paths = new ArrayList<>();

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

    allPathsWithAutonomy(g, vOrig, vDest, visited, new LinkedList<>(), paths, vehicleAutonomy);
    return paths;
}

private static <V, E extends Comparable<E>> void allPathsWithAutonomy(Graph<V, E> g, V current, V
destination, boolean[] visited, LinkedList<V> currentPath, ArrayList<LinkedList<V>> allPaths, double
remainingAutonomy) {
    visited[g.key(current)] = true;
    currentPath.add(current);

    if (current.equals(destination)) {
        allPaths.add(new LinkedList<>(currentPath));
    } else {
        for (V neighbor : g.adjVertices(current)) {
            if (!visited[g.key(neighbor)]) {
                E edgeWeight = g.edge(current, neighbor).getWeight();

                double edgeWeightValue = Double.parseDouble(edgeWeight.toString());
                double newRemainingAutonomy = remainingAutonomy - edgeWeightValue;

                if (newRemainingAutonomy >= 0) {
                    LinkedList<V> newPath = new LinkedList<>(currentPath);
                    allPathsWithAutonomy(g, neighbor, destination, visited, newPath, allPaths,
newRemainingAutonomy);
                }
            }
        }
        visited[g.key(current)] = false;
    }
}

```

Figura 3 - Método *allPathsWithAutonomy*

Tendo os percursos, agora só falta calcular a distância entre cada localidade do percurso e para tal é utilizado o método *shortestPath* que implementa o algoritmo de Dijkstra.

As inicializações das estruturas de dados têm complexidade  $O(V)$ , onde  $V$  é o número de vértices no grafo.

A complexidade do algoritmo de *Dijkstra* é geralmente dominada pelo *heap* binário ou fila de prioridade usada para atualizar as distâncias. Numa implementação padrão usando um *heap* binário, a complexidade de *Dijkstra* é  $O((V + E) \log V)$ , onde  $V$  é o número de vértices e  $E$  é o número de arestas. Já a construção do caminho mais curto envolve percorrer o *array pathKeys* e pode ser feita em tempo  $O(V)$ .

A complexidade total do método *shortestPath* é então dominada pela complexidade de *Dijkstra*, resultando em  $O((V + E) \log V)$ , onde  $V$  é o número de vértices e  $E$  é o número de arestas.

```

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];
    @SuppressWarnings("unchecked")
    V[] pathKeys = (V[]) new Object[numVerts];
    @SuppressWarnings("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 4 - Método *shortestPath*

Aqui temos o *main*, onde é pedido a localidade, o hub, a autonomia e a velocidade do veículo. Depois vai se buscar os percursos e em cada percurso é calculado a distância entre cada localidade, a distância total e o tempo.

Portanto, a complexidade da funcionalidade é a complexidade do método *allPathsWithAutonomy*  $O(V \cdot E)$ , visto esta ser a maior complexidade encontrada no *main*.

```

public static void main(String[] args) throws IOException {
    DistributionDataToModel dataToModel = new DistributionDataToModel("loais_small.csv",
"distancias_small.csv");
    MapGraph<Locality, Double> mapGraph = dataToModel.getDistances();
    Locality origin = null;
    Locality hub = null;
    Scanner reader = new Scanner(System.in);

    while (origin == null) {
        System.out.print("Enter a locality name >> ");
        String localityName = reader.next();

        origin = HubPath.localityExists(mapGraph, localityName);

        if (origin == null) {
            System.out.println("Localidade não encontrada. Tente novamente.");
        }
    }

    while (hub == null) {
        System.out.print("Enter a hub name >> ");
        String localityName = reader.next();

        hub = HubPath.localityExists(mapGraph, localityName);

        if (hub == null) {
            System.out.println("Hub não encontrado. Tente novamente.");
        }
    }

    System.out.print("Choose the vehicle autonomy (in km) >> ");
    double autonomy = reader.nextDouble();

    System.out.print("Choose the vehicle average speed (in km/h) >> ");
    double averageSpeed = reader.nextDouble();

    System.out.println();

    LinkedList<Locality> shortPath = new LinkedList<>();

    ArrayList<LinkedList<Locality>> paths = Algorithms.allPathsWithAutonomy(mapGraph, origin, hub,
autonomy * 1000);

    double totalDistance = 0;

    if (paths != null) {
        for (LinkedList<Locality> path : paths) {
            double totalDistanceWithinPath = 0;

            System.out.println("Point of Origin: " + origin);
            System.out.print("Path: ");
            for (Locality vertex : path) {
                System.out.print(vertex + " ");
            }
            System.out.println();

            for (int i = 0; i < path.size() - 1; i++) {
                Locality currentVertex = path.get(i);
                Locality nextVertex = path.get(i + 1);

                double distanceBetweenVertices = Algorithms.shortestPath(mapGraph, currentVertex,
nextVertex, Comparator.naturalOrder(), Double::sum, 0.0, shortPath);
                totalDistanceWithinPath += distanceBetweenVertices;

                System.out.printf("%s -> %s : Distance %.2f Km\n", currentVertex, nextVertex,
distanceBetweenVertices / 1000);
            }

            System.out.printf("Total Distance for Path: %.2f Km\n", totalDistanceWithinPath / 1000);
            System.out.println("Time Spent: " +
convertToHoursAndMinutes(calculateTimeTrip(averageSpeed * 1000, totalDistanceWithinPath)));
            System.out.println();
        }
    }
}

```

Figura 5 - Função main da US06

## USEI 07

---

Encontrar para um produtor, que parte de um local, o percurso de entrega que maximiza o número de hubs pelo qual passa, tendo em consideração o horário de funcionamento de cada hub, o tempo de descarga dos cestos em cada hub, as distâncias a percorrer, a velocidade média do veículo e os tempos descarregamento do veículo.

```
~ ~ ~ ~ ~
MapGraph<Locality, Double> hubMapGraph;
1 usage
MapGraph<Locality, Double> hubMapGraphClone;
4 usages
MapGraph<Locality, Double> trueHubMapGraph;
2 usages
HashMap<Pair<Integer,Integer>, Path> trueHubMapGraphEdgesDetails = new HashMap<>();
5 usages
Locality startingLocation;
5 usages
MutablePair<Integer, Integer> currentTime = new MutablePair<>();
4 usages
int autonomy;
3 usages
double velocity;
3 usages
int chargingTime;
3 usages
int unloadingTime;
3 usages
Set<Locality> trueHubs = new HashSet<>();
3 usages
double currentAutonomy;
```

Figura 6 – Definição de variáveis

Temos um construtor que recebe os dados dos nossos ficheiros csv e guarda o respetivo grafo na variável *hubMapGraph*.

Também temos o método *getPathWithMostHubs()* que vai inicializar a variável local *currentAutonomy* e fazer um clone do grafo.



```

4 usages  - Gustavo
public MaximizeHubsController() throws Exception {
    DistributionDataToModel dataToModel = new DistributionDataToModel( locationsFile: "loais_small.csv", distancesFile: "distancias_small.csv", scheduleFile: "schedules_small.csv");
    hubMapGraph = dataToModel.getDistances();
}

3 usages  - Gustavo
public List<HubTrip> getPathWithMostHubs() {
    currentAutonomy = autonomy;
    hubMapGraphClone = hubMapGraph.clone();
    if (trueHubs.contains(startingLocation))
        System.out.printf("Delivery denied on %s due to being the starting point.\n ", startingLocation.getName());
    return nearestNeighbour();
}

```

Figura 7 – Construtor e Método *getPathWithMostHubs*

De seguida, criamos o método *generateTrueMapGraph*, que recebe o grafo e o número de *hubs* pedidos pelo utilizador.

Em função do número de *hubs*, vamos buscar uma lista composta pelos N *hubs*, através de um método desenvolvido anteriormente para a USEI02.

Escolhemos um vértice para começar a iterar e iniciamos dois *for loops* sobre os vértices do grafo.

```

2 usages  - Gustavo *
public void generateTrueMapGraph(MapGraph<Locality, Double> trueHubMapGraph, int numberOfHubs) {
    List<Locality> hubs = HubOptimization.orderByAllCriteria(trueHubMapGraph, numberOfHubs);

    for (Locality hub : hubs) {
        trueHubMapGraph.addVertex(hub);
    }

    Locality start = trueHubMapGraph.vertices().iterator().next();
    trueHubMapGraph.addVertex(start);

    for (Locality hubI : trueHubMapGraph.vertices()) {
        for (Locality hubJ : trueHubMapGraph.vertices()) {

```

Figura 8 – Parte do método *generateTrueMapGraph*

Após fazer algumas verificações, chamamos o método *shortestPath*, que se encontra na nossa classe dos algoritmos.

```

if (hubI != hubJ) {
    if (trueHubMapGraph.edge(hubI, hubJ) == null) {
        while(true) {
            Path hubPath = new Path();
            Double path = Algorithms.shortestPath(trueHubMapGraph, hubI, hubJ, Comparator.naturalOrder(), Double::sum, zero: 0.0, hubPath.getShortestPath());
            if (path == null) {
                if (getId(hubI.getName()) < getId(hubJ.getName()))
                    System.out.printf("There is no path available between %s and %s.\n", hubI.getName(), hubJ.getName());
                break;
            }
            Optional<Pair<Locality, Locality>> problemEdge = hubPath.updateValues(trueHubMapGraph, autonomy);
            if (problemEdge.isEmpty()) {
                trueHubMapGraphEdgesDetails.put(new ImmutablePair<>(getId(hubI.getName()), getId(hubJ.getName())), hubPath);
                trueHubMapGraph.addEdge(hubI, hubJ, hubPath.getWeight());
                break;
            } else {
                trueHubMapGraph.removeEdge(problemEdge.get().getLeft(), problemEdge.get().getRight());
            }
        }
    }
}

```

Figura 9 - Outra parte do método generateTrueMapGraph

Este algoritmo possui uma complexidade de  $O(V * \log V + E)$ , sendo  $V$  o número de vértices do grafo e  $E$  o número de arestas.

```

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 10 - Método shortestPath

Em jeito de conclusão e fazendo os cálculos finais, a complexidade final desta classe é de  $O(N^3 * \log N)$ .

## USEI 08

Encontrar para um produtor o circuito de entrega que parte de um local de origem, passa por N hubs com maior número de colaboradores uma só vez e volta ao local de origem minimizando a distância total percorrida.

Nesta *user story* foi-nos pedido para encontrar um circuito de entrega que parte de um local de origem, passa por N *hubs* com maior número de colaboradores uma só vez e volta ao local de origem, através do caminho mais curto possível.

```
public static List<Locality> findOptimalDeliveryCircuit(Map<Graph<Locality, Double> graph, Locality origin, int numHubs, Vehicle vehicle) {
    HubOptimization hubOptimization = new HubOptimization();
    List<Locality> hubs = hubOptimization.orderByAllCriteria(graph, numHubs);

    // Diogo Martins
    Comparator<Locality> collaboratorsComparator = new Comparator<Locality>() {
        // Diogo Martins
        @Override
        public int compare(Locality locality1, Locality locality2) {
            return Integer.compare(locality2.getCollaborators(), locality1.getCollaborators());
        }
    };

    // Going path calculation
    hubs.sort(collaboratorsComparator);
    hubs.remove(origin);
    List<Locality> goingPath = findShortestPathThroughVertices(graph, origin, hubs, vehicle.getAutonomy());

    if (goingPath.isEmpty())
        System.exit( status: 1);

    // Coming path calculation
    Locality newOrigin = goingPath.get(goingPath.size()-1);
    LinkedList<Locality> comingPath = new LinkedList<>();
    Algorithms.shortestPath(graph, newOrigin, origin, Comparator.naturalOrder(), Double::sum, zero: 0.0, comingPath);
    comingPath.remove(newOrigin);

    List<Locality> result = new ArrayList<>(goingPath);
    result.addAll(comingPath);

    List<Locality> finalResult = new ArrayList<>();
    Locality previousLocality = null;

    for (Locality currentLocality : result) {
        if (previousLocality == null || !previousLocality.equals(currentLocality)) {
            finalResult.add(currentLocality);
        }
        previousLocality = currentLocality;
    }

    return finalResult;
}
```

Figura 11 - Método *findOptimalDeliveryCircuit*

Neste primeiro método começamos por ordenar os *hubs* usando o método '*orderByAllCriteria*', que tem uma complexidade de  $O(V * \log(V) * (V * ((V + E) * \log(V))))$ , como já foi explicado no relatório do sprint anterior (*USEI02*). Em seguida, os *hubs* são reordenados com base no número de colaboradores em ordem decrescente. A complexidade desta ordenação é  $O(N * \log(N))$ , onde  $N$  é o número de *hubs* selecionados, esta complexidade deve-se ao método de ordenação do java '*sort*' que tem essa complexidade.

Em seguida utilizamos o método que será explicado mais à frente '*findShortestPathThroughVertices*' que calcula o caminho de ida mais curto para os *hubs* selecionados. A complexidade deste método é de  $O(V^2 * (E + V * \log(V)))$ , que vai ser explicada mais à frente.

Depois para calcular o caminho de volta para o vértice inicial, utilizamos o método '*shortestPath*' que através da utilização do algoritmo de *Dijkstra* calcula o caminho mais curto, quanto à complexidade deste método, como foi dito no relatório anterior, é  $O(V * ((V + E) * \log(V)))$ .

Por fim os caminhos de ida e volta são combinados para formar o circuito, e então a complexidade deste primeiro método é  $O(V * \log(V) * (V * ((V + E) * \log(V))))$ , visto que é a maior complexidade deste método.

```

public static List<Locality> findShortestPathThroughVertices(Map<Graph<Locality, Double> graph, Locality origin, List<Locality> targetVertices, double autonomy) {
    try {
        List<Locality> optimalPath = new ArrayList<>();
        Set<Locality> remainingVertices = new HashSet<>(targetVertices);
        Locality currentVertex = origin;

        while (!remainingVertices.isEmpty()) {
            List<List<Locality>> paths = new ArrayList<>();

            for (Locality targetVertex : remainingVertices) {
                List<List<Locality>> currentPaths = bfsShortestPath(graph, currentVertex, targetVertex, autonomy);

                if (currentPaths != null && !currentPaths.isEmpty()) {
                    paths.addAll(currentPaths);
                }
            }

            if (paths.isEmpty()) {
                throw new LowAutonomyException("Error. Didn't find all the hubs. Low autonomy");
            }

            List<Locality> shortestPath = findShortestPath(paths, graph);
            optimalPath.addAll(shortestPath);
            remainingVertices.remove(optimalPath.get(optimalPath.size() - 1));
            currentVertex = optimalPath.get(optimalPath.size() - 1);
        }

        return optimalPath;
    } catch (LowAutonomyException e) {
        // Handle the exception (print a message, log, etc.)
        System.out.println("Low autonomy exception: " + e.getMessage());
        return Collections.emptyList(); // Return an empty list or handle it as needed
    }
}

```

Figura 12 - Método *findShortestPathThroughVertices*

Este segundo método, começa por inicializar uma lista que armazenará o caminho mais curto de ida. Em seguida entra-se em um loop principal que continua até todos os vértices de destino serem alcançados. Para cada um destes vértices é chamado o método '*bfsShortestPath*' que calcula todos os caminhos mais curtos do '*currentVertex*' para o vértice de destino, que tem complexidade  $O((N * (E + V)) * \log(V))$ .

Os caminhos mais curtos encontrados são adicionados à lista de caminhos ('*paths*') e é chamado então o método '*findShortestPath*' que encontra o caminho mais curto entre os caminhos disponíveis nessa lista, que tem complexidade  $O(V * (N * (E + V * \log(V))))$ .

Por fim é retornado o caminho mais curtos para todos os vértices de destino alcançados.

A complexidade deste método será aproximadamente  $O(\text{Número de vértices de destino} * (E + V * \log(V)))$ .

```

public static List<Locality> findShortestPath(List<List<Locality>> paths, MapGraph<Locality, Double> graph) {
    List<Locality> shortestPath = null;
    double minCost = Double.MAX_VALUE;

    for (List<Locality> path : paths) {
        double totalCost = calculateCost(path, graph);
        if (shortestPath == null || totalCost < minCost) {
            minCost = totalCost;
            shortestPath = new ArrayList<>(path);
        }
    }

    return shortestPath;
}

```

Figura 13 - Método *findShortestPath*

Este terceiro método, itera sobre cada caminho na lista de caminhos fornecida, e utilizando o método ‘*calculateCost*’, de complexidade  $O(P * (E + V * \log(V)))$ , onde  $P$  é o número de caminhos na lista, calcula o custo total do caminho atual, e se o caminho atual tiver um menor custo mínimo do que o custo encontrado até agora, utiliza esse como caminho atual.

Quanto a complexidade deste método, ‘*findShortestPath*’, é  $O(V * P * (E + V * \log(V)))$ , onde  $V$  é o número de vértices do grafo e  $P$  o número de caminhos na lista.

```

public static List<List<Locality>> bfsShortestPath(MapGraph<Locality, Double> graph, Locality start, Locality end, double autonomy) {
    List<List<Locality>> allPaths = new ArrayList<>();
    PriorityQueue<List<Locality>> queue = new PriorityQueue<>(Comparator.comparingDouble(path -> calculateCost(path, graph)));
    Set<Locality> visited = new HashSet<>();

    queue.add(Arrays.asList(start));
    visited.add(start);

    while (!queue.isEmpty()) {
        List<Locality> currentPath = queue.poll();
        Locality currentVertex = currentPath.get(currentPath.size() - 1);

        if (currentVertex.equals(end)) {
            allPaths.add(new ArrayList<>(currentPath));
        }

        for (Locality neighbor : graph.adjVertices(currentVertex)) {
            if (!visited.contains(neighbor)) {
                double distance = calculateDistanceBetweenLocalities(currentVertex, neighbor, graph);

                if (autonomy >= distance) {
                    List<Locality> newPath = new ArrayList<>(currentPath);
                    newPath.add(neighbor);
                    queue.add(newPath);
                    visited.add(neighbor);
                }
            }
        }
    }

    return allPaths.isEmpty() ? null : allPaths;
}

```

Figura 14 - Método *bfsShortestPath*

Neste penúltimo método é inicializada uma lista de prioridade que armazenará caminhos, começando com o caminho que contém apenas vértices de início. Também inicializa um conjunto *'visited'* para rastrear vértices visitados.

Em seguida, enquanto a lista criada anteriormente não estiver vazia, removemos o caminho de menor custo da fila e obtemos o último vértice desse caminho, para cada vizinho não visitado do vértice atual, cria-se um caminho estendendo o caminho atual até esse vizinho e esse novo caminho é adicionado à lista e o vizinho é marcado como visitado.

Por fim é retornada a lista de todos os caminhos mais curtos encontrados. A complexidade deste método é  $O((N * (E + V)) * \log(V))$ , onde  $E$  é o número de arestas e  $V$  é o número de vértices. A fila de prioridade ao ser atualizada em cada interação, também influencia na complexidade deste algoritmo.

```
private static double calculateCost(List<Locality> path, MapGraph<Locality, Double> graph) {  
    double totalCost = 0.0;  
  
    for (int i = 0; i < path.size() - 1; i++) {  
        Locality currentVertex = path.get(i);  
        Locality nextVertex = path.get(i + 1);  
        Edge<Locality, Double> edge = graph.edge(currentVertex, nextVertex);  
        if (edge != null) {  
            totalCost += edge.getWeight();  
        } else {  
            return Double.MAX_VALUE;  
        }  
    }  
  
    return totalCost;  
}
```

Figura 15 - Método calculateCost

Por fim, o método *'calculateCost'*, é utilizado para iterar sobre cada par de vértices consecutivos de um caminho, para cada par de vértices obtém-se a aresta correspondente no grafo e adiciona-se o peso da aresta ao custo total. E no final é retornado o custo total do caminho.

Quanto à complexidade, esta é influenciada principalmente pelo loop que itera sobre cada aresta no caminho, considerando como  $P$  o número de vértices no caminho. A complexidade é então  $O(P * (E + V * \log(V)))$ .

Por fim a complexidade deste algoritmo é de  $O(V * \log(V) * (V * ((V + E) * \log(V))))$ , visto que esta é a maior complexidade presente em todo o algoritmo.



## USEI 09

---

Organizar as localidades do grafo em N clusters que garantam apenas 1 hub por cluster de localidades. Os clusters devem ser obtidos iterativamente através da remoção das ligações com o maior número de caminhos mais curtos entre localidades até fiquem clusters isolados. Não deverá fornecer soluções de clusters de localidades sem o respetivo hub.

Para organizar as localidades do grafo em N clusters, foi criado o método *organizeGraphIntoClusters*, que recebe o respetivo grafo e o valor N, que corresponde ao número de clusters pretendidos pelo utilizador.

Existem algumas estruturas de dados para guardar as nossas informações, nomeadamente, o *map* dos clusters, a lista dos *hubs* e mais algumas variáveis locais para nos ajudar da distribuição das localidades pelos diversos *hubs*.

É importante salientar, em particular, a forma como obtemos os *hubs*, sendo que chamamos um método que foi desenvolvido no sprint anterior, para a USEI02, que nos permite obter uma lista de N *hubs* em função de certos critérios.

```
public static Map<Locality, Set<Locality>> organizeGraphIntoClusters(MapGraph<Locality, Double> mapGraph, int numberOfClusters) {  
    Map<Locality, Set<Locality>> clusters = new HashMap<>();  
    List<Locality> hubs = orderByAllCriteria(mapGraph, numberOfClusters);  
  
    int totalVertices = mapGraph.numVertices();  
    int maxVerticesPerCluster = totalVertices / numberOfClusters;  
    int remainingVertices = totalVertices % numberOfClusters;  
  
    if (remainingVertices > 0) {  
        maxVerticesPerCluster++;  
    }  
  
    Set<Locality> assignedLocalities = new HashSet<>();  
}
```

Figura 16 - Parte da US09

Até ao momento a nossa complexidade, é de  $O(K \log K)$  devido ao método *orderByAllCriteria*.

Sabendo que cada cluster é composto por apenas um hub, vamos iterar sobre essa lista de *hubs* e começar a construir o respetivo cluster. Para isso, vamos calcular o *betweenness* de cada aresta do grafo e guardar tudo num *Map<Edge, Integer>* e ordenamos este *map*.

```

for (Locality hub : hubs) {
    Set<Locality> cluster = new HashSet<>();
    cluster.add(hub);

    int verticesInCluster = 1;

    while (verticesInCluster < maxVerticesPerCluster && !hubs.isEmpty()) {
        Map<Edge, Double> shortestPathsMap = new HashMap<>();

        for (Edge edge : mapGraph.edges()) {
            Locality startLocality = (Locality) edge.getVOrig();
            Locality endLocality = (Locality) edge.getVDest();

            if (!assignedLocalities.contains(startLocality) && !assignedLocalities.contains(endLocality)) {
                boolean[] visitedLocality = new boolean[mapGraph.numVertices()];
                Locality[] paths = new Locality[mapGraph.numVertices()];
                Double[] numberOfDistances = new Double[mapGraph.numVertices()];
                Map<Edge, Integer> edgeUsageCount = new HashMap<>();
            }
        }
    }
}

```

Figura 17 - Outra parte da US09

A complexidade começa a aumentar e estamos perante dois *for loops* e um *while loop*. Sendo que o primeiro *for loop*, tem uma complexidade  $O(N)$ , sendo  $N$  o número de *hubs* e o segundo *for loop* uma complexidade  $O(E)$ , sendo  $E$  o número de *edges*. Já o *while loop*, tem complexidade  $O(V)$ , sendo  $V$  o número máximo de vértices do grafo.

Por fim, retiramos cada aresta, por ordem decrescente de *betweenness* e adicionamo-las aos clusters existentes até que não exista mais nenhuma aresta disponível.

```

List<Map.Entry<Edge, Double>> sortedShortestPaths = new ArrayList<>(shortestPathsMap.entrySet());
sortedShortestPaths.sort(Map.Entry.<comparingByValue().reversed());

Edge edgeToAdd = sortedShortestPaths.get(0).getKey();
Locality startLocality = (Locality) edgeToAdd.getVOrig();
Locality endLocality = (Locality) edgeToAdd.getVDest();

cluster.add(startLocality);
cluster.add(endLocality);

assignedLocalities.add(startLocality);
assignedLocalities.add(endLocality);

verticesInCluster += 2;

mapGraph.removeEdge((Locality) edgeToAdd.getVOrig(), (Locality) edgeToAdd.getVDest());
}

clusters.put(hub, cluster);
assignedLocalities.addAll(cluster);

```

Figura 18 - Outra parte da US09

É importante realçar que para efetuar o cálculo do *betweenness*, foi usado o algoritmo de *Dijkstra*. Sendo que o *betweenness* corresponde à frequência com que uma aresta, em particular, aparece nos vários *shortest paths* do grafo.

Este método de *Dijkstra* tem uma complexidade de  $O(V^2)$ .

```

shortestPathDijkstraWithEdgeCount(mapGraph, startLocality, Comparator.naturalOrder(),
    Double::sum, zero: 0.0, visitedLocality, paths, numberOfDistances, edgeUsageCount);

int targetIndex = mapGraph.key(endLocality);
if (numberOfDistances[targetIndex] != null) {
    double currentShortestPaths = numberOfDistances[targetIndex];
    shortestPathsMap.put(edge, currentShortestPaths);
}

```

*Figura 19 - Outra parte da US09*

Concluindo, este método conclui a tarefa de organizar o grafo em N clusters com uma taxa de eficiência elevada e tem uma complexidade de  $O(N * E * V^3 + K \log K)$ .