Instituto Superior de Engenharia do Porto

# Relatório Estruturas de Informação

## Sprint 2

Trabalho Realizado por:

- 1221194 – Diana Neves
- 1220913 – Marco Ferreira
- 1220917 – Tomás Gonçalves
- 1221330 – Francisco Monteiro

# Diagrama de Classes

# Análise dos Métodos Implementados nas User Stories :

## 1. USEI01

### 1.1 importData(Graph<Hub, Integer> graph)

O método importData é responsável por importar informações de arquivos CSV para preencher um grafo. Aqui está um resumo do que ele faz:

1.Cria um mapa (Map<String, Hub> map) para armazenar informações dos hubs.

2. Lê dados de um arquivo CSV locais_small.csv que contém informações sobre hubs, onde cada linha representa um hub.
3. Para cada linha lida, cria um objeto Hub com base nos dados lidos e tenta adicionar esse hub ao grafo. Se o hub for adicionado com sucesso, é inserido no mapa.
4. Fecha o scanner do primeiro arquivo.
5. Inicia a leitura de um segundo arquivo CSV distancias_small.csv que contém informações sobre as distâncias entre os hubs.
6. Para cada linha lida, recupera os hubs correspondentes usando o mapa criado anteriormente e adiciona uma aresta com o peso (distância) entre esses hubs no grafo.

A complexidade desse método depende principalmente do número de hubs e das distâncias presentes nos arquivos CSV.

A leitura de cada linha e a criação dos hubs tem complexidade O(n), onde n é o número de linhas nos arquivos.

Para adicionar vértices ao grafo, a complexidade depende da implementação subjacente da estrutura de dados do grafo, mas geralmente tem complexidade O(1) ou O(log n), onde n é o número de vértices no grafo.

A adição de arestas também depende da implementação do grafo e da estrutura de dados subjacente, geralmente com complexidade O(1) ou O(log n), onde n é o número de arestas no grafo.

Assim, a complexidade total do método pode ser aproximadamente O(n), considerando o número total de linhas nos arquivos CSV e assumindo operações de adição de vértices e arestas com complexidade constante ou logarítmica em relação ao número de vértices ou arestas no grafo.

```java
public static void importData(Graph<Hub, Integer> graph) throws FileNotFoundException {
    Map<String, Hub> map = new HashMap<>();
    File file = new File( pathname: "src/resources/ESINF/locais_small.csv");
    Scanner input = new Scanner(file);
    String header = input.nextLine();
    while (input.hasNextLine()){
        String info = input.nextLine();
        String[] line = info.split( regex: ",");
        Hub h = new Hub(line[0], new Coordinates(Double.parseDouble(line[1]), Double.parseDouble(line[2])));
        boolean f = graph.addVertex(h);
        if (f) {
            map.put(h.getLocalId(), h);
        }
    }
    input.close();
    input = new Scanner(new File( pathname: "src/resources/ESINF/distancias_small.csv"));
    header = input.nextLine();
    while (input.hasNextLine()){
        String info = input.nextLine();
        String[] line = info.split( regex: ",");
        Hub v1 = map.get(line[0]);
        Hub v2 = map.get(line[1]);
        int weight = Integer.parseInt(line[2]);
        graph.addEdge(v1, v2, weight);
    }
    input.close();
}
```

## 1.2 MapGraph<Hub, Integer> importMapGraph()

O método importMapGraph cria um grafo (MapGraph) vazio, preenche-o utilizando o método importData que lê informações de arquivos CSV e, por fim, retorna o grafo preenchido.

1. Inicializa um novo grafo vazio.
2. Utiliza o método importData para preencher o grafo com informações lidas de arquivos CSV.
3. Retorna o grafo preenchido.

A complexidade desse método depende principalmente da complexidade do método importData.

Supondo que a complexidade de importData seja O(n), onde n é o número total de linhas nos arquivos CSV, a complexidade de importMapGraph seria a mesma, pois basicamente delega o trabalho para o método importData.

```
public static MapGraph<Hub, Integer> importMapGraph() throws FileNotFoundException {
    MapGraph<Hub, Integer> result = new MapGraph<>( directed: false);
    importData(result);
    return result;
}
```

## 1.3 USES01()

O método USES01 é responsável por realizar a importação de um grafo preenchido com informações de arquivos CSV e exibir uma representação desse grafo na saída.

1. Chama o método importMapGraph() para obter um grafo preenchido com informações lidas de arquivos CSV.
2. Imprime na saída a representação em formato de texto desse grafo.

A complexidade desse método é essencialmente a mesma que a complexidade do método importMapGraph() utilizado internamente.

Supondo que a complexidade de importMapGraph() seja O(n), onde n é o número total de linhas nos arquivos CSV, a complexidade de USES01 seria, em sua maior parte, também O(n) devido ao trabalho realizado dentro do método importMapGraph().

```
public static void USES01() throws FileNotFoundException {
    MapGraph<Hub, Integer> mapGraph = importMapGraph();
    System.out.println(mapGraph.toString());
}
```

# 2. USEI02

## 2.1 List<HubOptimizationCriteria> optimizeHubs( Graph<Hub, Integer> graph)

O método optimizeHubs recebe um grafo como entrada e realiza a otimização dos hubs dentro desse grafo. Ele itera por cada hub no grafo e calcula diferentes critérios de otimização para cada um deles. Para cada hub:

- Calcula a distância mínima para outros hubs no grafo usando o algoritmo de caminhos mais curtos.
- Calcula a proximidade somando as distâncias mínimas encontradas.
- Calcula a centralidade do hub.

- Configura esses critérios calculados em um objeto HubOptimizationCriteria e o adiciona a uma lista de critérios de otimização.

Ao final, ordena a lista de critérios de otimização com base na centralidade e influência de cada hub, em ordem decrescente, e retorna essa lista ordenada.

A complexidade do método depende principalmente da quantidade de hubs no grafo (n) e das operações realizadas para calcular os critérios de otimização, como o cálculo de caminhos mínimos e centralidade, cujas complexidades variam dependendo das implementações específicas dos métodos utilizados. No geral, a complexidade principal do método é de O(n), onde n é o número de hubs no grafo.

```
public static List<HubOptimizationCriteria> optimizeHubs(Graph<Hub, Integer> graph) {

    List<HubOptimizationCriteria> optimizationCriteria = new ArrayList<>();

    for (Hub hub : graph.vertices()) {
        double prox = 0;
        double centrality = 0;
        int distance = 0;
        ArrayList<LinkedList<Hub>> paths = new ArrayList<>();
        ArrayList<Integer> dists = new ArrayList<>();

        if (Algorithms.shortestPaths(graph, hub, Integer::compare, Integer::sum, zero: 0, paths, dists)) {

            for (int d : dists) {
                if (d > 0) {
                    distance += d;
                }
            }

            prox = distance;

            centrality = calculateCentrality(graph, hub);

        }

        HubOptimizationCriteria hubOptimizationCriteria = new HubOptimizationCriteria(hub);
        hubOptimizationCriteria.setInfluence(graph.outDegree(hub));
        hubOptimizationCriteria.setProximity(prox);
        hubOptimizationCriteria.setCentrality(centrality);
        optimizationCriteria.add(hubOptimizationCriteria);

    }

    return  orderListByDecreasinglyOrderOfCentralityAndInfluence(optimizationCriteria);
}
```

## 2.2 calculateCentrality(Graph<Hub, Integer> graph, Hub hub)

O método calculateCentrality calcula a centralidade de um determinado hub em um grafo. Para isso:

1. Itera sobre todos os vértices no grafo.
2. Para cada vértice, calcula os caminhos mais curtos para todos os outros vértices.
3. Verifica se o hub específico está presente nos caminhos encontrados e incrementa a centralidade se estiver.
4. Retorna o valor da centralidade calculada.

O método possui um loop externo que itera sobre todos os vértices do grafo, resultando em O(V), onde V é o número de vértices no grafo.

Dentro do loop externo, utiliza o algoritmo de caminho mais curto para calcular os caminhos entre o vértice atual e todos os outros, resultando em O(V * (V + E)), onde E é o número de arestas no grafo.

Em resumo, a complexidade total do método é O(V^2 + V*E), considerando a iteração sobre todos os vértices e o cálculo dos caminhos mais curtos para cada vértice.

## 2.3 List<HubOptimizationCriteria> orderListByDecreasinglyOrderOfCentrality AndInfluence(List<HubOptimizationCriteria > list)

O método orderListByDecreasinglyOrderOfCentralityAndInfluence recebe uma lista de objetos HubOptimizationCriteria e os ordena com base na centralidade e influência de cada elemento, em ordem decrescente.

1. Utiliza o método Collections.sort para ordenar a lista de acordo com critérios de comparação definidos por um comparador anônimo.
2. O comparador verifica primeiro a centralidade dos elementos e, se forem iguais, compara a influência para desempate.
3. Retorna a lista ordenada.

A complexidade do método Collections.sort depende do algoritmo de ordenação utilizado. Geralmente, o algoritmo de ordenação padrão do Java é o algoritmo Quicksort ou Timsort.

Em média, a complexidade de tempo do Quicksort é O(n log n), onde n é o número de elementos na lista.

Portanto, a complexidade do método orderListByDecreasinglyOrderOfCentralityAndInfluence é O(n log n) em média, considerando a ordenação da lista de elementos usando o algoritmo Quicksort.

```
public static List<HubOptimizationCriteria> orderListByDecreasinglyOrderOfCentralityAndInfluence(List<HubOptimizationCriteria> list){
    Marco
    Collections.sort(list, new Comparator<HubOptimizationCriteria>() {
        Marco
        @Override
        public int compare(HubOptimizationCriteria o1, HubOptimizationCriteria o2) {
            if (o1.getCentrality() > o2.getCentrality()) {
                return -1;
            } else if (o1.getCentrality() < o2.getCentrality()) {
                return 1;
            } else {
                if (o1.getInfluence() > o2.getInfluence()) {
                    return -1;
                } else if (o1.getInfluence() < o2.getInfluence()) {
                    return 1;
                } else {
                    return 0;
                }
            }
        }
    });
    return list;
}
```

## 2.4 USES02()

O método USES02 é responsável por realizar uma série de operações sobre a otimização de hubs em um grafo e imprimir informações relevantes na saída.

1. Chama o método InputInfo.importMapGraph() para obter um grafo.

3. Utiliza o método optimizeHubs para otimizar os hubs no grafo.
4. Itera sobre a lista de HubOptimizationCriteria resultante da otimização.
5. Imprime na saída informações específicas de cada hub, como o identificador local, a influência, a proximidade e a centralidade.

O método importMapGraph() para obtenção do grafo pode ter uma complexidade dependente da quantidade de dados lidos e do método utilizado para importação.

O método optimizeHubs possui uma complexidade considerável, que depende do tamanho do grafo e dos cálculos realizados para otimização.

O loop for que itera sobre a lista de HubOptimizationCriteria tem complexidade O(n), onde n é o número de hubs na lista.

Cada operação de impressão na saída é de complexidade constante O(1), não afetando significativamente a complexidade total.

A complexidade geral do método depende, em grande parte, das complexidades dos métodos importMapGraph e optimizeHubs, sendo dominada pelo que tiver maior complexidade, ou seja , complexidade O(n).

# 3. USEI03

## 3.1 US03()

No geral, o construtor parece destinado a inicializar um objeto US03 e carregar uma rede de hubs a partir de algum arquivo, mas o trecho new HubNetwork(); lê a rede de hubs , que a partir do método getHubs(), são adicionadas a um grafo graph, que as representa.

```
public US03() throws FileNotFoundException {
    System.out.println("Imported from File");
    new HubNetwork();
    graph = HubNetwork.getHubs();

}
```

## 3.2 print()

O método print tem como objetivo calcular e exibir informações sobre a rota mais longa (em termos de distância) que um veículo com uma determinada autonomia pode percorrer na rede de hubs.

- São inicializadas variáveis necessárias para rastrear a distância mínima (min), o hub de destino (hDest), o hub de origem correspondente à rota mais longa (hOrigMax), e o caminho correspondente à rota mais longa (pathMax).
- Há um loop externo que itera sobre todos os hubs na rede (graph). Para cada hub de origem (hOrig), é executado o algoritmo de Dijkstra para calcular as distâncias mais curtas até todos os outros hubs na rede.
- Dentro do loop externo, há um loop interno que compara as distâncias calculadas para os hubs de destino. Se a distância até um hub de destino (graph.vertex(i)) for maior que a distância mínima (min), então atualizamos as variáveis min, hDest, hOrigMax, e pathMax.
- Após a conclusão dos loops, o método imprime no console informações sobre a rota mais longa encontrada. Isso inclui o hub de destino e a distância mínima até esse hub.
- Utilizando o método buildShortestPath, é criada uma lista (pathInOrder) que representa o caminho mais longo em ordem. Em seguida, o método shortestPathCarVei é chamado para calcular os pesos das arestas ao longo do

caminho, levando em consideração a autonomia do veículo. Essas informações são então impressas no console.

- Há um assert hDest != null; para garantir que o hub de destino (hDest) não seja nulo antes de criar a lista do caminho.

  A complexidade final do método print é dominada pelos loops aninhados e pelas chamadas de métodos dijkstra e shortestPathCarVei. Portanto, a complexidade total é aproximadamente O(V^2).

```
public void print() throws FileNotFoundException {
    Scanner scanner = new Scanner(System.in);
    double autonomia;

    System.out.println("Insira a autonomia do veículo: (m)");
    autonomia = scanner.nextInt();

    int min = 0;
    Hub hDest = null;
    Hub hOrigMax = null;
    int[] pathMax = null;

    for (int j = 0; j < graph.numVertices(); j++) {
        Hub hOrig = graph.vertex(j);
        boolean[] visited = new boolean[graph.numVertices()];
        int[] path = new int[graph.numVertices()];
        int[] distances = new int[graph.numVertices()];
        dijkstra(graph, hOrig, visited, path, distances);

        for (int i = j + 1; i < graph.numVertices(); i++) {

            if (!(graph.vertex(i) == hOrig)) {
                if (distances[i] > min) {
                    min = distances[i];
                    hDest = graph.vertex(i);
                    hOrigMax = hOrig;
                    pathMax = path;

                }
            }
        }
    }
```

```
        System.out.println("Vertex: " + hDest + " Distance: " + min);
        assert hDest != null;
        LinkedList<Hub> pathInOrder = buildShortestPath(graph, hOrigMax, hDest, pathMax);

        System.out.println("Shortest Path in Order with Edge Weights:");
        Map<Edge, Double> edgeWeightMap = shortestPathCarVei(pathInOrder, autonomia, hOrigMax);
        System.out.println(edgeWeightMap.toString());

    }
```

## 3.3 dijkstra(MapGraph<Hub, Integer> graph, Hub hOrig, boolean[] visited, int[] path, int[] distances) {

O método dijkstra implementa o algoritmo de Dijkstra para encontrar os caminhos mais curtos a partir de um hub de origem (hOrig) para todos os outros hubs em um grafo ponderado (graph).

- Inicializa os arrays visited, distances, e path para rastrear os hubs visitados, as distâncias mínimas e os caminhos mais curtos.
- Define a distância do hub de origem para ele mesmo como zero (distances[graph.key(hOrig)] = 0).
- Inicia um loop que continua até que todos os hubs tenham sido visitados. No loop, seleciona-se o hub atual (hOrig), marca-o como visitado e itera sobre seus hubs adjacentes não visitados, ajustando as distâncias mínimas se encontrar um caminho mais curto.
- Utiliza a função getVertMinDist para obter o índice do próximo hub (nextVertex) com a menor distância entre os hubs não visitados.
- Atualiza o hub de origem (hOrig) para o próximo hub a ser processado. Se não houver mais hubs a serem processados, o loop é encerrado.

A complexidade final do método dijkstra é dominada pelo loop principal, resultando em uma complexidade aproximada de O(V^2).

```java
public void dijkstra(MapGraph<Hub, Integer> graph, Hub hOrig, boolean[] visited, int[] path, int[] distances) {

    for (int i = 0; i < graph.numVertices(); i++) {
        visited[i] = false;
        distances[i] = MAX_VALUE;
        path[i] = -1;
    }
    int i, j;
    double weight;
    distances[graph.key(hOrig)] = 0;

    while (hOrig != null) {
        i = graph.key(hOrig);
        visited[i] = true;
        for (Hub hAdj : graph.adjVertices(hOrig)) {
            weight = graph.edge(hOrig, hAdj).getWeight();
            j = graph.key(hAdj);

            if (!visited[j] && distances[j] > distances[i] + weight) {
                distances[j] = (int) (distances[i] + weight);
                path[j] = i;
            }
        }
        int nextVertex = getVertMinDist(distances, visited);
        if (nextVertex != -1) {
            hOrig = graph.vertex(nextVertex);
        } else {
            hOrig = null;
        }
    }
}
```

## 3.4 getVertMinDist(int[] distances, boolean[] visited) {

Este método é utilizado para encontrar o índice do vértice não visitado com a menor distância a partir da origem.

A complexidade final do método getVertMinDist é dominada pelo loop de busca, resultando em uma complexidade linear em relação ao número de vértices no grafo. Portanto, a complexidade é aproximadamente O(V), onde V é o número de vértices. Este método é geralmente eficiente, especialmente para grafos com um número moderado de vértices.

```java
private int getVertMinDist(int[] distances, boolean[] visited) {
    int min = MAX_VALUE;
    int index = -1;
    for (int i = 0; i < distances.length; i++) {
        if (!visited[i] && min > distances[i]) {
            min = distances[i];
            index = i;
        }
    }
    return index;
}
```

## 3.5 buildShortestPath(MapGraph<Hub, Integer> graph, Hub hOrig, Hub hDest, int[] path) {

```java
public LinkedList<Hub> buildShortestPath(MapGraph<Hub, Integer> graph, Hub hOrig, Hub hDest, int[] path) {
    int destIndex = graph.key(hDest);
    LinkedList<Hub> pathInOrder = new LinkedList<>();

    while (destIndex != -1 && !hOrig.equals(hDest)) {
        Hub vertex = graph.vertex(destIndex);

        int prevIndex = path[destIndex];
        if (prevIndex != -1) {

            pathInOrder.addFirst(vertex);

        }

        destIndex = prevIndex;
    }
    return pathInOrder;

}
```

Este método implementa a reconstrução do caminho mais curto de um vértice de origem (hOrig) para um vértice de destino (hDest) em um grafo ponderado representado por um MapGraph. O caminho mais curto é determinado usando um array de índices (path) que armazena o índice do vértice predecessor para cada vértice no caminho.

A análise de complexidade deste método é O(n), onde n é o número de vértices no grafo. Vamos analisar o código para entender por quê:

Inicialização:

A inicialização de destIndex é feita uma vez, e a operação graph.key(hDest) para obter o índice do destino é uma operação de tempo constante.

A criação da lista vinculada (pathInOrder) e do loop while inicializa as variáveis e é executada uma vez.

Portanto, a inicialização do método é O(1).

Loop While:

O loop while executa até que o índice de destino seja -1 (indicando que chegamos à origem) ou que a origem seja igual ao destino.

Dentro do loop, a obtenção do vértice (vertex) a partir do índice de destino é uma operação de tempo constante (graph.vertex(destIndex)).

A condição if (prevIndex != -1) verifica se o índice do vértice predecessor é válido antes de adicionar o vértice à lista vinculada. Isso também é uma operação de tempo constante.

A linha pathInOrder.addFirst(vertex) adiciona o vértice ao início da lista vinculada, o que é uma operação de tempo constante.

A atualização do índice de destino (destIndex = prevIndex) é uma operação de tempo constante.

O loop é executado no máximo n vezes (onde n é o número de vértices no caminho mais curto).

Retorno:

O método retorna a lista vinculada que contém o caminho mais curto reconstruído.

Em resumo, a complexidade total é dominada pelo loop while, que é O(n), onde n é o número de vértices no caminho mais curto. Portanto, a análise de complexidade é O(n).

```
public LinkedList<Hub> buildShortestPath(MapGraph<Hub, Integer> graph, Hub hOrig, Hub hDest, int[] path) {
    int destIndex = graph.key(hDest);
    LinkedList<Hub> pathInOrder = new LinkedList<>();

    while (destIndex != -1 && !hOrig.equals(hDest)) {
        Hub vertex = graph.vertex(destIndex);

        int prevIndex = path[destIndex];
        if (prevIndex != -1) {

            pathInOrder.addFirst(vertex);

        }

        destIndex = prevIndex;
    }
    return pathInOrder;

}
```

## 3.6 shortestPathCarVei(LinkedList<Hub> pathInOrder, double autonomia, Hub hOrigMax)

O método shortestPathCarVei calcula o caminho mais curto entre os hubs (nós) de uma rede, considerando restrições de autonomia de um veículo. Ele utiliza um grafo representado por uma estrutura de dados graph e uma lista encadeada de hubs pathInOrder que define a ordem dos hubs a percorrer.

Resumindo o código:

Inicialização de variáveis e estruturas de dados: O método começa inicializando variáveis como sum, sumAutonomia, e count, e utiliza estruturas de dados como TreeMap, HashMap, e ArrayList.

Cálculo do caminho mais curto: Itera sobre os hubs em pathInOrder e, para cada par de hubs consecutivos, calcula o peso do caminho percorrido. Isso é feito somando os pesos das arestas entre os hubs.

Consideração da autonomia do veículo: Enquanto calcula o caminho, verifica se a autonomia do veículo é suficiente para percorrer cada aresta. Se não for, contabiliza isso no contador count e exibe mensagens indicando a necessidade de recarregar o veículo em certos hubs da rede.

Ordenação dos resultados: Ao final, a estrutura edgeWeightMap é ordenada com base nos pesos das arestas, utilizando uma lista de entradas do mapa. A ordenação é realizada pelo valor (peso do caminho) em ordem crescente.

Retorno do resultado: Retorna um mapa (sortedEdgeWeightMap) contendo as arestas e seus pesos, classificados em ordem crescente com base no peso do caminho percorrido.

Complexidade do código:

Complexidade temporal: O laço principal percorre os hubs em pathInOrder, resultando em uma complexidade de tempo de aproximadamente O(n), onde n é o número de hubs na lista. As operações realizadas dentro desse laço, como acessar arestas do grafo e realizar operações de soma e comparação, têm uma complexidade que depende da implementação subjacente das estruturas de dados, mas geralmente são operações de tempo constante ou logarítmico, considerando que graph é uma estrutura de grafo otimizada.

Complexidade espacial: O método utiliza estruturas de dados adicionais, como mapas e listas, para armazenar e manipular informações. A complexidade espacial está relacionada ao número de arestas no grafo e ao tamanho da lista pathInOrder, resultando em uma complexidade de espaço que pode ser aproximadamente O(e + n), onde e é o número de arestas e n é o número de hubs na lista.

Em resumo, o código calcula o caminho mais curto em um grafo, considerando a autonomia do veículo, e retorna um mapa ordenado das arestas percorridas com seus respetivos pesos. A complexidade do código é principalmente influenciada pelo número de hubs na lista e pelo tamanho do grafo subjacente.

O método shortestPathCarVei calcula o caminho mais curto entre os hubs (nós) de uma rede, considerando restrições de autonomia de um veículo. Ele utiliza um grafo representado por uma estrutura de dados graph e uma lista encadeada de hubs pathInOrder que define a ordem dos hubs a percorrer.

```java
public Map<Edge, Double> shortestPathCarVei(LinkedList<Hub> pathInOrder, double autonomia, Hub hOrigMax) {
    Hub prevVertex = null;
    Map<Edge, Double> edgeWeightMap = new TreeMap<>(new EdgeComparator());

    double sum = 0;
    double sumAutonomia = 0;
    int count = 0;
    if (!pathInOrder.isEmpty()) {
        Edge<Hub,Integer> firstEdge = graph.edge(hOrigMax, pathInOrder.getFirst());
        int firstEdgeWeight =  firstEdge.getWeight();
        sum = sum + firstEdgeWeight;
        Map<Edge, Double> map = new HashMap<>();
        map.put(firstEdge, sum);
        edgeWeightMap.put(firstEdge, sum);
        System.out.println("Edge: " + hOrigMax.getLocalId() + " to " + pathInOrder.getFirst().getLocalId() + " Weight: " + sum);
    }

    for (Hub vertex : pathInOrder) {
        if (prevVertex != null) {
            Edge<Hub, Integer> edge = graph.edge(prevVertex, vertex);
            int edgeWeight =  edge.getWeight();
            sum = sum + edgeWeight;
            sumAutonomia = sumAutonomia + edgeWeight;
            if (autonomia < edgeWeight) {
                count++;
            }
            if (sumAutonomia > autonomia && autonomia > edgeWeight) {
                System.out.println("Carregue o veículo na rede de distribuição: " + prevVertex.getLocalId());
                sumAutonomia = edgeWeight;
            }
            edgeWeightMap.put(edge, sum);
            System.out.println("Edge: " + prevVertex.getLocalId() + " to " + vertex.getLocalId() + " Weight: " + sum);
        }
        }
        prevVertex = vertex;
    }
    if (count > 0) {
        System.out.println(CoresOutput_RED + "Não é possível fazer esta viagem com um veículo com esta autonomia." + CoresOutput_RESET);
    }
    List<Map.Entry<Edge, Double>> list = new ArrayList<>(edgeWeightMap.entrySet());
    list.sort(Map.Entry.comparingByValue());

    // Criar um LinkedHashMap para manter a ordem de inserção após a classificação
    Map<Edge, Double> sortedEdgeWeightMap = new LinkedHashMap<>();
    for (Map.Entry<Edge, Double> entry : list) {
        sortedEdgeWeightMap.put(entry.getKey(), entry.getValue());
    }

    return sortedEdgeWeightMap;
}
```

# 4. USEI04

## 4.1 US04()

O método US04() é um construtor de uma classe não explicitamente fornecida, mas parece instanciar uma variável networkBuilder com base em uma instância de HubNetwork e a chamada do método getHubs().

1. O construtor US04() cria uma nova instância da classe.
2. Atribui à variável networkBuilder o resultado de new HubNetwork().getHubs().

A complexidade deste construtor é principalmente determinada pelo método getHubs() da classe HubNetwork,, ou seja, tem complexidade O(n).

O construtor em si é de complexidade constante O(1), pois não envolve iteração, laços ou operações que dependam do tamanho dos dados. Ele apenas realiza a criação de um objeto e a atribuição de um valor a uma variável.

```
public US04() throws FileNotFoundException {
    this.networkBuilder = new HubNetwork().getHubs();
}
```

## 4.2 MapGraph<Hub, Integer> getMiniumSpanningTree(MapGraph<Hub, Integer> graph)

O método getMinimumSpanningTree() é uma função estática que recebe um grafo como parâmetro e retorna a Árvore Geradora Mínima (Minimum Spanning Tree - MST) desse grafo, usando o algoritmo de Kruskal ou Prim. Ele verifica se o grafo é nulo e, se não for, chama o método Algorithms.minSpanTree() para calcular a Árvore Geradora Mínima.

Verifica se o grafo é nulo. Se for, retorna nulo; caso contrário, calcula a Árvore Geradora Mínima usando o método Algorithms.minSpanTree().

Algorithms.minSpanTree() utiliza o algoritmo de Kruskal, sua complexidade é O(E log V), onde E é o número de arestas e V é o número de vértices no grafo.

```
public static MapGraph<Hub, Integer> getMinimumSpanningTree(MapGraph<Hub, Integer> graph) {
    if (graph == null) {
        return null;
    } else {
        return Algorithms.minSpanTree(graph, Integer::compare, Integer::sum, zero: 0);
    }
}
```

## 4.3 execute()

O método execute() realiza as seguintes operações:

Obter a Árvore Geradora Mínima (MST):

Chama o método getMinimumSpanningTree(graph) para encontrar a MST do grafo. A complexidade depende do algoritmo utilizado para encontrar a MST, podendo ser algo em torno de O(E log V), onde E é o número de arestas e V é o número de vértices.

Iteração sobre os vértices e arestas da MST:

Utiliza dois loops aninhados para percorrer os vértices e arestas da MST. A complexidade desse trecho é de O(V + E), onde V é o número de vértices e E é o número de arestas na MST.

Verificação e soma das distâncias das arestas:

Dentro dos loops, verifica se a aresta já foi visitada e, se não foi, adiciona o peso da aresta à variável totalDistance. As verificações e adições têm complexidade O(1).

Impressão de informações:

Realiza operações de impressão que têm uma complexidade de tempo constante O(1), que são relativamente insignificantes em comparação com o restante do código.

A complexidade total do método execute() é dominada pela obtenção da MST e iteração sobre seus vértices e arestas. Em termos de pior caso, a complexidade geral seria aproximadamente O(E log V + V + E), considerando a obtenção da MST e a iteração sobre ela.

```java
public void execute() throws FileNotFoundException {
    MapGraph<Hub, Integer> graph = networkBuilder;
    MapGraph<Hub, Integer> minSpanTree = getMinimumSpanningTree(graph);
    Set<Edge<Hub, Integer>> visitedEdges = new HashSet<>();
    double totalDistance = 0;
    for (Hub hub : minSpanTree.vertices()) {
        for (Hub hub1 : minSpanTree.adjVertices(hub)) {
            if (hub != hub1) {
                Edge<Hub, Integer> edge = graph.edge(hub, hub1);
                if (!visitedEdges.contains(edge)) {
                    totalDistance += edge.getWeight();
                    visitedEdges.add(edge);
                    System.out.printf(edge.getVOrig().getLocalId() +" → " + edge.getVDest().getLocalId() +" Peso: "+ edge.getWeight() +  " metros\n");
                }
            }
        }
    }
    System.out.printf("Distância total: " + (int) totalDistance + " metros\n");
}
```

O método execute() realiza as seguintes operações: