ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Estruturas de Informação

SPRINT 2

26/11/2023

GRUPO 85

- Luna Silva – 1221184
- Diogo Moutinho – 1221014
- Pedro Coelho – 1220688
- José Mendes - 1220858

Índice

Índice de Figuras

Introdução

O presente relatório aborda a implementação e otimização de uma rede de distribuição de cabazes de produtos agrícolas para a empresa GFH, um operador logístico especializado nessa atividade. A GFH atua como intermediária entre produtores de diversos tipos de produtos agrícolas e clientes finais, que encomendam cabazes contendo uma variedade de produtos de diferentes produtores. Este sistema envolve a interação entre produtores, clientes e hubs, sendo estes últimos as entidades responsáveis por receber e disponibilizar os cabazes aos clientes.

A empresa GFH não realiza a composição dos cabazes, mas é encarregue da eficiente distribuição, utilizando veículos elétricos explorados pela própria empresa. A logística de distribuição é condicionada por vários parâmetros, como autonomia dos veículos, horário de funcionamento dos hubs e tempo de descarga dos cabazes nos mesmos.

Este relatório dedica-se à análise detalhada do Sprint 2 do projeto, com foco específico na cadeira Estruturas de Informação (ESINF) e relacionadas às classes que implementam a interface Graph. O objetivo principal é criar e testar um conjunto de classes que permitam gerir de forma eficiente a rede de distribuição de cabazes de produtos agrícolas. Este sistema é composto por vértices representativos de localidades onde hubs de distribuição podem ser estabelecidos, com cabazes sendo transportados entre eles por veículos elétricos.

Os principais desafios a serem abordados neste Sprint incluem a construção da rede de distribuição, a determinação dos locais ideais para a localização de hubs, a otimização da rede de distribuição com base em critérios específicos (influência, proximidade e centralidade), a rota mínima entre os locais mais afastados da rede, a determinação da rede de ligação mínima entre todas as localidades e a divisão da rede em clusters conexos e coesos. Estes objetivos serão alcançados através da implementação de classes e testes que consideram as características particulares da distribuição de cabazes no contexto da empresa GFH.

Além disso, será realizada uma avaliação da complexidade do código desenvolvido em resposta aos requisitos específicos apresentados neste sprint. Essa análise contribuirá para a compreensão da eficiência das soluções implementadas, fornecendo informações valiosos para possíveis otimizações e melhorias futuras no sistema de distribuição de cabazes da empresa GFH.

Apresentação das classes desenvolvidas

USEI01- Construção Eficiente da Rede de Distribuição de Cabazes

A classe US_EI01_GraphBuilder é responsável por construir e manipular um grafo representando a distribuição de hubs.

Método *US_EI01_GraphBuilder(), getDistribuicao()* e *getInstance()*

O método construtor, *US_EI01_GraphBuilder()*, inicializa a instância única da classe e cria um novo grafo não direcionado, o qual é acessível através do método *getDistribuicao()*.

```
public class US_EI01_GraphBuilder {
    1 usage
    private static final US_EI01_GraphBuilder instance = new US_EI01_GraphBuilder();

    5 usages
    final private MapGraph<Hub, Integer> distribuicao;

    1 usage   ± 35193
    private US_EI01_GraphBuilder() { this.distribuicao = new MapGraph<>( directed: false); }

    /**
     * Gets distribuicao.
     *
     * @return the distribuicao
     */
    7 usages   ± Diogo
    public MapGraph<Hub, Integer> getDistribuicao() { return distribuicao; }

    /**
     * Gets instance.
     *
     * @return the instance
     */
    ± 35193
    public static US_EI01_GraphBuilder getInstance() { return instance; }
```

Figure 1 - Método getDistribuição() & getInstance()

O método construtor *US_EI01_GraphBuilder()* desempenha um papel crucial na inicialização da classe. Ao ser chamado, cria automaticamente uma instância única da classe US_EI01_GraphBuilder devido à aplicação do padrão de design *Singleton*. Esse padrão assegura que apenas uma instância dessa classe seja mantida durante a execução do programa, promovendo um ponto global de acesso.

No método construtor, a variável estática *instance* é declarada e inicializada para armazenar a instância única da classe US_EI01_GraphBuilder. Esta inicialização envolve a criação de uma nova instância da classe usando a palavra-chave *new*. A palavra-chave *static* indica que essa variável pertence à classe, não a instâncias específicas, garantindo a consistência da instância única.

O método estático *getInstance()* é fornecido para acessar essa instância única. Ele verifica se a instância já foi criada, se não, cria uma nova antes de retorná-la. Isso assegura que, em todo o programa, apenas uma instância de US_EI01_GraphBuilder exista quando necessário, seguindo o padrão *Singleton*.

Dentro do construtor, um novo grafo não direcionado (*MapGraph*) é inicializado e atribuído à variável de instância *distribuicao*. O argumento *false* especifica que o grafo é não direcionado, adequado para representar a distribuição de *hubs*, onde as rotas entre eles não têm uma direção específica.

Método *addHub(String numId, Double lat, Double lon)*

O método *addHub(String numId, Double lat, Double lon)* desempenha o papel crucial de adicionar hubs ao grafo.

```
public boolean addHub(String numId, Double lat, Double lon) {
    Hub vert = new Hub(numId, lat, lon);
    return distribuicao.addVertex(vert);
}
```

Figure 2 - Método addHub

Primeiramente, ele cria uma nova instância da classe Hub utilizando os parâmetros fornecidos, que representam o identificador único (*numId*), a latitude (*lat*), e a longitude (*lon*). Em seguida, o método utiliza o grafo associado à instância da classe US_EI01_GraphBuilder, mais especificamente o método *addVertex* desse grafo. Isso resulta na adição do hub como um vértice no grafo de distribuição.

O método retorna um valor booleano, sendo *true* indicando que a adição foi bem-sucedida e *false* se o hub já existir no grafo, evitando duplicados com base no identificador único. Em suma, o método simplifica o processo de adição de hubs ao grafo, garantindo a consistência da representação do grafo de distribuição de hubs.

Método addRoute(Hub orig, Hub dest, Integer distance)

O método addRoute(Hub orig, Hub dest, Integer distance) é responsável por adicionar uma rota (aresta) entre dois hubs no grafo, especificando a distância como um peso para a aresta.

```
public boolean addRoute(Hub orig, Hub dest, Integer distance) {
    return distribuicao.addEdge(orig, dest, distance);
}
```

Figure 3 - Método addRout

Ao receber como parâmetros os hubs de origem (*orig*) e destino (*dest*), juntamente com a distância entre eles (*distance*), o método cria uma aresta ponderada no grafo. A ponderação da aresta é representada pela distância fornecida.

Internamente, esse método utiliza o grafo associado à instância da classe US_EI01_GraphBuilder e, mais especificamente, o método *addEdge* desse grafo. Isso resulta na criação de uma aresta direcionada entre os *hubs* de origem e destino, representando a rota da distribuição. Vale ressaltar que o grafo utilizado é não direcionado, refletindo a natureza bidirecional das rotas entre *hubs*.

O método retorna um valor booleano, sendo *true* indicando que a adição da rota foi bem-sucedida e *false* se a rota já existir no grafo, evitando duplicados com base nos *hubs* de origem e destino. Em resumo, o método simplifica o processo de estabelecimento de rotas entre *hubs*, contribuindo para a construção coerente do grafo de distribuição de *hubs*.

Método toString()

O método toString() desempenha o papel de fornecer uma representação em formato de string do grafo de distribuição mantido pela classe US_EI01_GraphBuilder. Ao ser invocado, este método utiliza o método *toString* do grafo subjacente (*MapGraph*), proporcionando uma visualização clara e compreensível do estado atual do grafo.

```
@Override
public String toString() { return distribuicao.toString(); }
```

Figure 4 - Método toString()

Internamente, a chamada ao método *toString* do grafo subjacente possibilita obter uma representação textual detalhada do grafo, incluindo informações sobre seus vértices, arestas e outras propriedades. Essa representação é então retornada pelo método *toString* da classe US_EI01_GraphBuilder, tornando-se uma ferramenta útil para a depuração e entendimento visual do grafo de distribuição de hubs.

Análise da Complexidade da USEI01

A eficiência de uma classe depende de vários fatores, incluindo a complexidade de suas operações. No caso da classe US_EI01_GraphBuilder, a sua eficiência é influenciada principalmente pela escolha do tipo de grafo subjacente e pela implementação das operações de adição de *hubs*, rotas e pela construção do grafo.

A escolha de um grafo não direcionado (*MapGraph*) é apropriada para representar a distribuição de *hubs*, onde as rotas entre eles não têm uma direção específica. No entanto, é importante considerar que a eficiência das operações pode variar dependendo da quantidade de *hubs* e rotas no sistema.

A complexidade das operações principais, como adição de *hubs* (*addHub*) e adição de rotas (*addRoute*), depende em grande parte da implementação do grafo subjacente. Se essas operações são implementadas de forma eficiente, por exemplo, usando estruturas de dados adequadas, a classe tem o potencial de ser eficiente em termos de desempenho.

É crucial considerar o contexto de uso da classe e as operações predominantes. Se o número de *hubs* e rotas é relativamente pequeno, a eficiência pode ser satisfatória. No entanto, se o sistema envolve uma grande quantidade de *hubs* e rotas, seria benéfico realizar análises de desempenho mais detalhadas para garantir a escalabilidade da classe.

Em resumo, a eficiência da classe US_EI01_GraphBuilder depende da implementação e do contexto de uso, sendo importante avaliar a complexidade das operações em relação ao tamanho do sistema.

USEI02 – Determinar os vértices ideais para a localização de N hubs de modo a otimizar a rede de distribuição

A classe **US_EI02_IdealVerticesForNHubs** disponibiliza métodos para calcular e obter os N vértices principais com base em vários critérios num grafo de hubs. Estes critérios incluem influência, centralidade e proximidade, oferecendo igualmente uma visão combinada dos N vértices principais ao considerar todos esses critérios. A classe implementa métodos para calcular a influência, proximidade e centralidade dos hubs no grafo. Adicionalmente, permite obter os N hubs principais com base em critérios separados (influência, centralidade ou proximidade) e numa visão combinada que considera centralidade, influência e proximidade. Estes métodos são úteis para análises de redes que procuram identificar hubs significativos num grafo por vários critérios.

Método calculateInfluence()

```
2 usages   pedrom2004_ *
public Map<Hub, Integer> calculateInfluence(MapGraph<Hub, Integer> graph) {
    Map<Hub, Integer> influence = new HashMap<>();
    for (Hub vertex : graph.vertices()) {
        influence.put(vertex, graph.outDegree(vertex));
    }

    return influence;
}
```

Figure 5 - Método calculateInfluence

O método **calculateInfluence** recebe um grafo representado pela classe **MapGraph** com vértices do tipo **Hub** e arestas ponderadas do tipo **Integer**. O objetivo do método é calcular a influência de cada vértice no grafo, definindo a influência como o grau de saída (**out-degree**) de cada vértice, ou seja, o número de arestas que saem desse vértice. Para isso, o método percorre todos os vértices do grafo, calcula o grau de saída de cada vértice usando o método **outDegree** do grafo e armazena essas informações num novo mapa chamado **influence**. Posteriormente, o método devolve esse mapa, que contém a influência de cada vértice no grafo.

Método calculateCentrality()

```
2 usages   new *
public Map<Hub, Integer> calculateCentrality(MapGraph<Hub, Integer> graph) {
    Map<Hub, Integer> centrality = Algorithms.betweennessCentrality(graph);

    return centrality;
}
```

Figure 6 - Método calculateCentrality

O método **calculateCentrality** simplesmente utiliza um método da classe **Algorithms** para calcular a centralidade de betweenness em um grafo. Essa métrica indica o quão importante cada vértice é na comunicação entre outros vértices na rede. O resultado é um mapa que associa cada vértice (**Hub**) a um valor numérico representando sua centralidade de betweenness.

Método calculateProximity()

```
2 usages   pedrom2004_ *
public Map<Hub, Integer> calculateProximity(MapGraph<Hub, Integer> graph) {
    Map<Hub, Integer> proximity = new HashMap<>();

    for (Hub vertex : graph.vertices()) {
        Integer proximityValue = calculateVertexProximity(graph, vertex);
        proximity.put(vertex, proximityValue);
    }

    return proximity;
}
```

Figure 7 - Método calculateProximity

O método **calculateProximity**, recebe um grafo representado por um objeto **MapGraph<Hub, Integer>**, onde **Hub** é o tipo de vértice no grafo e **Integer** é o tipo de peso nas arestas. O método retorna um **Map<Hub, Integer>** que representa a proximidade de cada vértice (ou **hub**) em relação aos outros vértices no grafo.

A lógica do método é iterar sobre todos os vértices do grafo, e para cada vértice, chama o método **calculateVertexProximity(graph, vertex)**, que aparentemente calcula a proximidade desse vértice específico em relação aos outros vértices no grafo. O valor resultante desse cálculo é então associado ao vértice correspondente no mapa **proximity**.

Método calculateVertexProximity()

```
1 usage   pedrom2004_
private Integer calculateVertexProximity(MapGraph<Hub, Integer> graph, Hub vertex) {
    ArrayList<Integer> dists = new ArrayList<>();
    Algorithms.shortestPaths(graph, vertex, Comparator.naturalOrder(), Integer::sum,  zero: 0, new ArrayList<>(), dists);

    int proximitySum = 0;
    for (Integer dist : dists) {
        if (dist != null) {
            proximitySum += dist;
        }
    }

    return proximitySum;
}
```

Figure 8 - Método calculateVertexProximity

O método **calculateVertexProximity** recebe como entrada um grafo representado por **MapGraph<Hub, Integer>** e um vértice específico (**vertex**). A sua lógica é calcular a proximidade desse vértice em relação aos outros vértices do grafo, através da soma das distâncias dos caminhos mais curtos.

Inicialmente, é criada uma lista (**dists**) para armazenar as distâncias calculadas. Em seguida, é chamado o método **Algorithms.shortestPaths** para encontrar os caminhos mais curtos a partir do vértice fornecido. O resultado desses caminhos mais curtos é guardado na lista **dists**.

Após o cálculo das distâncias, o método percorre a lista e soma as distâncias dos caminhos mais curtos, ignorando as distâncias nulas. O resultado final, que representa a medida de proximidade do vértice, é então retornado.

Método getTopNHubsSeparate()

```
1 usage  new *
public Map<Hub, Integer> getTopNHubsSeparate(Map<Hub, Integer> map, Integer n, boolean isProximity) {
    Map<Hub, Integer> topNHubsMap = new LinkedHashMap<>();

    // Sort the map entries based on values
    List<Map.Entry<Hub, Integer>> sortedEntries = new ArrayList<>(map.entrySet());

    if (isProximity) {
        sortedEntries.sort(Map.Entry.comparingByValue());
    } else {
        sortedEntries.sort(Map.Entry.comparingByValue(Comparator.reverseOrder()));
    }

    int count = 0;
    for (Map.Entry<Hub, Integer> entry : sortedEntries) {
        if (count < n) {
            topNHubsMap.put(entry.getKey(), entry.getValue());
            count++;
        } else {
            break;
        }
    }

    return topNHubsMap;
}
```

Figure 9 - Método getTopNHubsSeparate

O método **getTopNHubsSeparate** tem como objetivo selecionar os N principais **hubs** (vértices) de um grafo, considerando os valores associados a esses **hubs** como métricas relevantes. Os parâmetros incluem um mapa que associa instâncias de **Hub** a valores inteiros, o número desejado de hubs principais (N), e um indicador booleano (**isProximity**) que determina se a ordenação deve ser feita com base na proximidade (se verdadeiro) ou em influência e centralidade (se falso).

O método utiliza uma estrutura de dados **LinkedHashMap** para armazenar os **N** principais hubs e seus valores associados. Inicialmente, converte as entradas do mapa original numa lista de pares key-value, e depois ordena essa lista com base nos valores associados aos hubs. A direção da ordenação (crescente ou decrescente) é determinada pelo parâmetro **isProximity**.

Seguidamente, o método percorre a lista ordenada e adiciona os **N principais hubs** e seus valores associados ao novo mapa. O **loop** é interrompido assim que o número desejado de hubs é atingido.

O resultado final é o mapa contendo os N principais hubs e as métricas associadas, ordenados de acordo com a lógica estabelecida.

Método getTopNMap()

```
1 usage   new *
public Map<Hub, List<Integer>> getTopNMap(Map<Hub, List<Integer>> map, Integer n){
    List<Map.Entry<Hub, List<Integer>>> entries = new ArrayList<>(map.entrySet());
    entries.sort((entry1, entry2) -> {
        List<Integer> values1 = entry1.getValue();
        List<Integer> values2 = entry2.getValue();

        int compareCentrality = Integer.compare(values2.get(0), values1.get(0));
        if (compareCentrality != 0) {
            return compareCentrality;
        }

        int compareInfluence = Integer.compare(values2.get(1), values1.get(1));
        if (compareInfluence != 0) {
            return compareInfluence;
        }

        return Integer.compare(values1.get(2), values2.get(2));
    });

    Map<Hub, List<Integer>> sortedFinalMap = new LinkedHashMap<>();
    for (Map.Entry<Hub, List<Integer>> entry : entries) {
        sortedFinalMap.put(entry.getKey(), entry.getValue());
    }

    return getTopNHubs(sortedFinalMap, n);
}
```

Figure 10 - Método getTopNMap

O método **getTopNMap** tem como objetivo ordenar um mapa de hubs, onde cada hub está associado a uma lista de valores inteiros, seguindo uma lógica de comparação específica. Em seguida, o método retorna os N principais hubs e as suas listas associadas.

O código começa por converter as entradas do mapa original numa lista de pares key-value. Essa lista é então ordenada com base nos valores associados aos hubs, utilizando uma lógica de comparação personalizada. A ordenação ocorre em três fases, considerando três elementos na lista associada a cada hub: centralidade, influência e proximidade valor.

Após a ordenação, é criado um novo mapa para armazenar os hubs ordenados e as respetivas listas associadas. O método utiliza uma estrutura de dados **LinkedHashMap** para manter a ordem de inserção. Em seguida, o resultado é limitado aos N principais hubs, utilizando o método **getTopNHubs**.

Método getTopNHubs()

```
1 usage   new *
public Map<Hub, List<Integer>> getTopNHubs(Map<Hub,List<Integer>> map, Integer n){
    Map<Hub, List<Integer>> topNHubsMap = new LinkedHashMap<>();

    int count = 0;
    for (Map.Entry<Hub, List<Integer>> entry : map.entrySet()) {
        if (count < n) {
            topNHubsMap.put(entry.getKey(), entry.getValue());
            count++;
        } else {
            break;
        }
    }

    return topNHubsMap;
}
```

Figure 11 - Método getTopNHubs

O método **getTopNHubs** tem como objetivo devolver os N principais hubs e as suas listas associadas, com base num mapa onde cada hub está ligado a uma lista de valores inteiros. Neste processo, o código percorre as entradas do mapa original e adiciona os hubs e respetivas listas associadas a um novo mapa chamado **topNHubsMap**. Este processo continua até que o número desejado de hubs (**n**) seja atingido. O resultado final é, então, limitado aos N principais hubs, e o método devolve este mapa.

Esta abordagem é útil quando se pretende extrair uma seleção específica de hubs com base em algum critério, neste caso, limitando o número para **n**. A escolha de utilizar um `**LinkedHashMap**` garante a preservação da ordem de inserção, se isso for relevante para a aplicação.

Análise da Complexidade da USEI02

calculateInfluence:

- Este método itera sobre todos os vértices no grafo uma vez (complexidade O(n)), onde n é o número de vértices. Para cada vértice, é feita uma operação constante (adicionar a influência ao mapa). Assim, a complexidade total é **O(n)**.

calculateProximity:

- Esse método chama o método **calculateVertexProximity** para cada vértice no grafo. Internamente, **calculateVertexProximity** utiliza o algoritmo de caminho mais curto (**Algorithms.shortestPaths**). A complexidade desse algoritmo é O((V + E) * log(V)), onde V é o número de vértices e E é o número de arestas. Portanto, a complexidade total de **calculateProximity** é **O(n * (V + E) * log(V))**, onde n é o número de vértices no grafo.

calculateVertexProximity:

- Este método utiliza o algoritmo de caminho mais curto (**Algorithms.shortestPaths**). A complexidade desse algoritmo é **O((V + E) * log(V))**, onde V é o número de vértices e E é o número de arestas.

calculateCentrality:

- Utiliza o algoritmo de centralidade de betweenness (**Algorithms.betweennessCentrality**). A complexidade deste algoritmo é **O(V * (V + E))**, onde V é o número de vértices e E é o número de arestas.

getTopNHubsSeparate:

- Este método envolve a ordenação da lista de entradas do mapa com base nos valores. A ordenação em Java utiliza o algoritmo de ordenação QuickSort, que tem uma complexidade média **de O(n * log(n))**, onde n é o número de entradas no mapa.

getTopNMap:

- Similar ao método anterior, também envolve ordenação e tem complexidade **O(n * log(n))**.

getTopNHubs:

- Itera sobre as entradas ordenadas, resultando em uma complexidade **O(n)**, onde n é o número de entradas no mapa.

Resumidamente, a complexidade de cada método é determinada principalmente pelo número de vértices (V) e arestas (E) no grafo. Métodos que envolvem algoritmos de grafos tendem a ter uma complexidade relacionada a esses parâmetros, enquanto métodos de ordenação têm complexidade associada ao número de elementos a serem ordenados.

USEI03- US_EI03_VehicleRouteAutonomy

Esta classe Java, denominada **US_EI03_VehicleRouteAutonomy**, é parte de uma aplicação que lida com a otimização de rotas para veículos em uma rede de hubs logísticos.

```
private static Hub findHubByNumId(MapGraph<Hub, Integer> g, String numId) {
    for (Hub hub : g.vertices()) {
        if (hub.getNumId().equals(numId)) {
            return hub;
        }
    }
    return null;
}
```

Figure 12 - findHubByNumId

O método **findHubByNumId** é uma função utilitária que busca e retorna um objeto `Hub` com base em seu identificador numérico (**numId**) em um grafo (**MapGraph**). O método é declarado como **private** para limitar seu acesso a outras classes, e aceita um grafo **MapGraph** que contém hubs e um identificador numérico **numId** que será usado para localizar o hub desejado. O método utiliza um loop **for-each** para iterar sobre todos os vértices (hubs) no grafo representado por **g.**

Para cada hub na iteração, o método verifica se o identificador numérico do hub é igual ao identificador numérico fornecido como parâmetro. Se uma correspondência é encontrada, o hub é retornado. Se nenhum hub corresponde ao identificador numérico fornecido, o método retorna **null**, indicando que nenhum hub foi encontrado.

Em resumo, o método **findHubByNumId** é responsável por buscar e retornar um objeto `Hub` com base em seu identificador numérico em um grafo.

```java
public static Integer shortestPath(MapGraph<Hub, Integer> g, String vOrig, String vDest,
                                   Comparator<Integer> ce, BinaryOperator<Integer> sum, Integer zero,
                                   LinkedList<Hub> shortPath) {

    Hub hubOrig = findHubByNumId(g,vOrig) /* Obtain the Hub object for vOrig */;
    Hub hubDest = findHubByNumId(g,vDest)/* Obtain the Hub object for vDest */;

    if (hubOrig == null || hubDest == null) {
        return null;
    }

    LinkedList<Hub> hubShortPath = new LinkedList<>();
    Integer totalDistance = Algorithms.shortestPath(g, hubOrig, hubDest, ce, sum, zero, hubShortPath);

    // Convert Hub objects to String representation for the result
    for (Hub hub : hubShortPath) {
        shortPath.add(hub);
    }

    return totalDistance;
}
```

Figure 13 - shortestPath

O método **shortestPath** desempenha um papel essencial no cálculo do caminho mais curto entre dois hubs em um grafo ponderado. O método aceita um grafo (**g**) representando a rede de hubs, os identificadores numéricos dos hubs de origem (**vOrig**) e destino (**vDest**), comparador (**ce**) para comparar pesos de arestas, operador binário (**sum**) para combinar distâncias, um valor inicial (**zero**), e uma lista (**shortPath**) que será preenchida com o caminho mais curto.

Os objetos **Hub** correspondentes aos identificadores numéricos de origem e destino são obtidos utilizando o método **findHubByNumId**. Se os hubs de origem ou destino não existirem no grafo, o método retorna `null`, indicando uma condição inválida. Utilizando o método **shortestPath** da classe **Algorithms**, o código calcula o caminho mais curto entre os hubs de origem e destino, bem como a distância total desse caminho. O caminho é mantido na lista **hubShortPath**. Os objetos **Hub** no caminho mais curto são convertidos para representações de string e adicionados à lista **shortPath**. Isso é feito para fornecer uma forma mais legível dos resultados.

Em resumo, o método **shortestPath** encapsula a lógica para calcular o caminho mais curto entre dois hubs em um grafo ponderado, fornecendo informações essenciais sobre a distância total desse caminho. A utilização de objetos **Hub** e a conversão para representações de string contribuem para a clareza e utilidade dos resultados obtidos

```java
public static Pair<Hub, Hub> findTwoFarthestHubs(MapGraph<Hub, Integer> g,LinkedList<Hub> shortestPath) {

    // Inicializa a variável para armazenar a maior distância encontrada
    Integer maxDistance = 0;

    // Inicializa as variáveis para armazenar os dois hubs mais afastados
    Hub farthestHub1 = null;
    Hub farthestHub2 = null;

    // Percorre todos os pares de hubs e calcula as distâncias
    for (Hub hub1 : g.vertices()) {
        for (Hub hub2 : g.vertices()) {
            if (!hub1.equals(hub2)) {
                LinkedList<Hub> currentShortestPath = new LinkedList<>();
                Integer distance = Algorithms.shortestPath(g, hub1, hub2, Comparator.naturalOrder(), Integer::sum,  zero: 0, currentShortestPath);

                // Atualiza os hubs mais afastados se a distância for maior que a atual
                if (distance > maxDistance) {
                    maxDistance = distance;
                    farthestHub1 = hub1;
                    farthestHub2 = hub2;
                    shortestPath.clear();
                    shortestPath.addAll(currentShortestPath);
                }
            }
        }
    }

    return new Pair<>(farthestHub1, farthestHub2);
}
```

Figure 14 - findTwoFarthestHubs

O método **findTwoFarthestHubs** tem como objetivo identificar os dois hubs mais distantes em um grafo ponderado, representando uma rede de locais. O método aceita um grafo (**g**) representando a rede de locais (hubs) e uma lista (**shortestPath**) que será preenchida com o caminho mais curto entre os dois hubs mais distantes. São inicializadas variáveis para armazenar a maior distância encontrada (**maxDistance**) e os dois hubs mais afastados (**farthestHub1 e farthestHub2**). O método utiliza dois loops aninhados para iterar sobre todos os pares de hubs no grafo, excluindo pares iguais.

Para cada par de hubs distintos, o método calcula a distância entre eles utilizando o método **shortestPath** da classe **Algorithms**. A distância é armazenada na variável **distance,** e o caminho mais curto é mantido em **currentShortestPath**. Os hubs mais afastados são atualizados se a distância entre o par atual for maior que a distância máxima registrada. A lista **shortestPath** é então limpa e preenchida com o caminho mais curto entre esses hubs. O método retorna um par de hubs representando os dois hubs mais distantes no grafo.

Em resumo, o método **findTwoFarthestHubs** é responsável por percorrer todos os pares de hubs em um grafo, calcular as distâncias entre eles, e identificar os dois hubs mais distantes, mantendo o caminho mais curto entre eles na lista **shortestPath**. Este processo é fundamental para determinar os hubs entre os quais a autonomia dos veículos pode ser mais desafiadora, influenciando a estratégia de carregamento ao longo do trajeto.

```java
public Map<String, Object> shortestPathBetweenTheTwoLocals(MapGraph<Hub, Integer> distancesGraph,Integer autonomia) {

    // Run Dijkstra's algorithm to find the shortest path
    LinkedList<Hub> shortestPath = new LinkedList<>();
    Pair<Hub, Hub> pair = findTwoFarthestHubs(distancesGraph,shortestPath);


    String sourceVertex =pair.getFirst().getNumId();
    String destinationVertex = pair.getSecond().getNumId();
    // Calculate the distances between locations in the path
    List<Pair<String, String>> pathDistances = calculatePathDistances(shortestPath, distancesGraph);

    // Calculate the number of loadings (assuming it's the number of vertices in the path - 2)
    List<Hub> numberOfLoadings = calculateLoadingHubs(shortestPath, distancesGraph, autonomia);


    // Construct the result map
    Map<String, Object> result = new HashMap<>();
    result.put("sourceVertex", sourceVertex);
    result.put("loadingLocations",numberOfLoadings); // Locais de passagem excluindo origem e destino
    result.put("pathDistances", pathDistances);
    result.put("destinationVertex", destinationVertex);
    result.put("numberOfLoadings", numberOfLoadings.size());

    return result;
}
```

Figure 15 - shortestPathBetweenTheTwoLocals

Este método é o método principal desta classe que recebe um grafo ponderado (**distancesGraph**) representando as distâncias entre hubs, e um valor de autonomia para veículos e tem como objetivo encontrar a rota mais curta entre dois locais em um grafo ponderado, representando distâncias entre diferentes hubs.

O método inicia executando o **algoritmo de Dijkstra** que se localiza na pasta **graph** que foi nos fornecida para completarmos e serve para encontrar o caminho mais curto entre dois hubs mais distantes no grafo. O resultado é um par de hubs (**pair**) que representa a origem e o destino do caminho mais longo, e a lista **shortestPath** é preenchida com os hubs que compõem esse caminho. Para encontrar os dois hubs mais distantes este método recorre a um método auxiliar presente na classe **findTwoFarthestHubs.**

Os identificadores numéricos dos hubs de origem e destino são extraídos do par de hubs obtido anteriormente. Utilizando o método **calculatePathDistances**, o código calcula as distâncias entre hubs consecutivos no caminho mais curto. O resultado é uma lista de pares de strings representando as distâncias. O método **calculateLoadingHubs** é utilizado para determinar os hubs de carregamento necessários ao longo do caminho e por sua vez o número de vezes que o carro carrega, levando em consideração a autonomia do veículo.

Um mapa chamado **result** é criado para armazenar informações cruciais sobre o caminho mais curto.

- **sourceVertex**, que é o identificador numérico do hub de origem;
- **loadingLocations**, uma lista de hubs representando os locais de carregamento, excluindo origem e destino;
- **pathDistances**, uma lista de pares de strings representando as distâncias entre hubs consecutivos no caminho;
- **destinationVertex**, o identificador numérico do hub de destino;
- **numberOfLoadings**, o número total de hubs de carregamento ao longo do caminho.

Essas informações proporcionam uma visão abrangente do trajeto mais curto, abordando pontos de partida e chegada, locais de carregamento necessários e distâncias entre hubs consecutivos. O mapa **result** é utilizado para organizar e apresentar esses dados de maneira coerente.

Analise da Complexidade da USEI03

O método **findHubByNumId** apresenta uma complexidade temporal linear em relação ao número de vértices no grafo.

Relativamente ao método **calculatePathDistances,** o loop for percorre todos os elementos da lista path uma vez. Assumindo que o tamanho da lista é n, a complexidade dessa parte é O(n). Dentro do loop, as operações realizadas que são operações de tempo constante para cada iteração. Isso significa que a complexidade dentro do loop é O(1) para cada iteração. Portanto, a complexidade total do método é dominada pelo loop, e é O(n).

Relativamente ao método **calculateLoadingHubs** o loop é executado um número de vezes igual a path.size() - 1, onde path.size() representa o número de vértices no caminho. Portanto, a complexidade do loop é proporcional ao tamanho da lista path, e é expressa como O(n), onde n é o tamanho da lista. O acesso ao grafo é realizado através do método distancesGraph.edge(vertexA, vertexB).O acesso a uma aresta específica deve ser em O(1), ou seja, uma operação de tempo constante em média. A adição à lista loadingHubs não depende do tamanho da lista e é considerada uma operação eficiente.

Relativamente ao método **shortestPath**, duas operações de busca são realizadas para encontrar os objetos Hub correspondentes aos identificadores vOrig e vDest. A complexidade dessas operaçõespode ser considerada O(n), onde n é o número de vértices no grafo. A complexidade temporal desta parte é dominada pela complexidade do algoritmo de Dijkstra. Portanto, é **O((V + E) * log(V)).** Iteração sobre a lista hubShortPath e adição dos objetos Hub à lista shortPath. A complexidade desta operação é linear em relação ao tamanho da lista hubShortPath, ou seja, O(n), onde n é o número de vértices no caminho mais curto.

Relativamente ao método **findTwoFarthestHubs**, a complexidade total do método é então determinada pela multiplicação da complexidade do loop duplo e da complexidade da chamada ao método Algorithms.shortestPath. Portanto, a complexidade temporal é aproximadamente:

O(n^2 *((V + E) * log(V)))

Onde n é o número de hubs no grafo e **O((V + E) * log(V))** é a complexidade do algoritmo de caminho mais curto utilizado Dijkstra.

Relativamente ao método **shortestPathBetweenTheTwoLocals**, é a soma de todos as outras complexidade já calculadas. Dessa forma, a complexidade dominante seria a do primeiro termo, e a complexidade total pode ser aproximadamente expressa como **O(n^2 *((V + E) * log(V))).**

USEI04 – US_EI04_MinimumSpanningTree

A classe *US_EI04_MinimumSpanningTree* é a responsável por obter a arvore de custo mínimo e o seu custo total. No problema em questão, as arestas representam o percurso entre dois Hubs, e o peso a respetiva distancia. Este é um grafo não direcionado, visto que é permitida a deslocação quer de X a Y quer de Y a X.

Método getMinimumSpanningTree()

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

O método **getMinimumSpanningTree** é o responsável por receber o grafo criado na US 01 e chamar o método **minimumSpanningTree** da classe *Algorithms*.

Método minimumSpanningTree()

```
public static <V, E> MapGraph<V, E> minimumSpanningTree(MapGraph<V, E> g, Comparator<E> ce, BinaryOperator<E> sum, E zero) {
    // Create a set to keep track of visited vertices
    Set<V> visitedVertices = new HashSet<>();

    // Create a priority queue to store edges based on their weights
    PriorityQueue<Edge<V, E>> edgeQueue = new PriorityQueue<>(Comparator.comparing(Edge::getWeight, ce));

    // Create a graph to represent the minimum spanning tree
    MapGraph<V, E> minimumSpanningTree = new MapGraph<>(true);

    // Add an arbitrary vertex to start the process
    V startVertex = g.vertices().iterator().next();
    visitedVertices.add(startVertex);

    // Add all edges connected to the start vertex to the priority queue
    edgeQueue.addAll(g.outgoingEdges(startVertex));

    // Continue adding edges until all vertices are visited
    while (visitedVertices.size() < g.numVertices()) {
        // Get the minimum weight edge from the priority queue
        Edge<V, E> minEdge = edgeQueue.poll();
        // Check if the priority queue is empty
        if (minEdge == null) {
            // The graph is not connected
            break;
        }

        // Get the destination vertex of the minimum weight edge
        V destVertex = minEdge.getVDest();
```

```
            // Check if adding this edge creates a cycle
            if (!visitedVertices.contains(destVertex)) {
                // Add the destination vertex to the set of visited vertices
                visitedVertices.add(destVertex);

                // Add the edge to the minimum spanning tree
                minimumSpanningTree.addEdge(minEdge.getVOrig(), destVertex, minEdge.getWeight());

                // Add all edges connected to the destination vertex to the priority queue
                edgeQueue.addAll(g.outgoingEdges(destVertex));
            }
        }
        return minimumSpanningTree;
    }
```

O método **minimumSpanningTree** é um algoritmo que encontra e constrói a árvore de custo mínimo de um grafo ponderado não direcionado. Para isso, recorre ao algoritmo de Kruskal, presente na classe Algorithms. Este método utiliza um algoritmo conhecido como "Prim's Algorithm" para criar uma árvore de custo mínimo em um grafo ponderado. A árvore de custo mínimo é uma subestrutura do grafo original, que conecta todos os vértices de forma que a soma dos pesos das arestas seja minimizada. No contexto do nosso problema, simboliza a distancia mínima necessária para visitar todos os Hubs.

O algoritmo inicia escolhendo um vértice arbitrário como ponto de partida. Em seguida, ele adiciona à árvore de custo mínimo a aresta de peso mínimo conectada ao vértice atual. Este processo é repetido até que todos os vértices estejam incluídos na árvore.

Para realizar isso, o método mantém uma fila de prioridade (representada por **PriorityQueue**) para ordenar as arestas com base em seus pesos. Ele também utiliza um conjunto (**Set**) para rastrear os vértices já foram contabilizados.

O algoritmo continua a adicionar as arestas de menor peso até que todos os vértices sejam contabilizados. O processo verifica se adicionar uma aresta cria um ciclo no grafo e, se não, a aresta é adicionada à árvore e os vértices conectados são marcados como contabilizados.

O método retorna a árvore de custo mínimo representada por um objeto do tipo **MapGraph**, que contém apenas as arestas necessárias para conectar todos os vértices de forma eficiente. Este algoritmo é eficaz para encontrar soluções ótimas em termos de peso em problemas práticos, como uma rede de transporte neste caso.

Método calculateTotalHeight()

Por fim, na classe *US_EI04_MinimumSpanningTree* está presente o método **calculateTotalHeight**, que recebe um MapGraph<Hub, Integer> e retorna a soma do peso de todas as suas arestas. Na implementação desta User Storie, é o responsável por retornar o peso/distancia total do grafo de custo mínimo.

```
public static int calculateTotalHeight(MapGraph<Hub, Integer> graph) {
    int totalHeight = 0;
    for (Edge<Hub, Integer> edge : graph.edges()) {
        totalHeight += edge.getWeight();
    }
    return totalHeight;
}
```

Análise da Complexidade da USEI04

Ao examinarmos a temporalidade deste algoritmo, destacamos a operação fundamental de preenchimento da fila de prioridade. Esta etapa, impulsionada pelo algoritmo de Prim, demonstra uma complexidade temporal de $O(E \log E)$, onde E representa o número de arestas no grafo. A relação logarítmica surge da ordenação eficiente das arestas pela fila de prioridade(PriorityQueue), refletindo a habilidade do algoritmo de Kruskal em selecionar as arestas de peso mínimo de maneira o mais otimizada possível.

A estrutura da Árvore de Custo Mínimo, núcleo do nosso algoritmo, começa com uma complexidade temporal que reflete a densidade de vértices (V) e arestas (E). No entanto, a inserção eficiente de vértices e arestas, aliada à verificação de ciclos, mantém a complexidade sob controle, resultando num desempenho que resiste ao teste do tempo, não se tornando demasiado lento nem ineficiente.