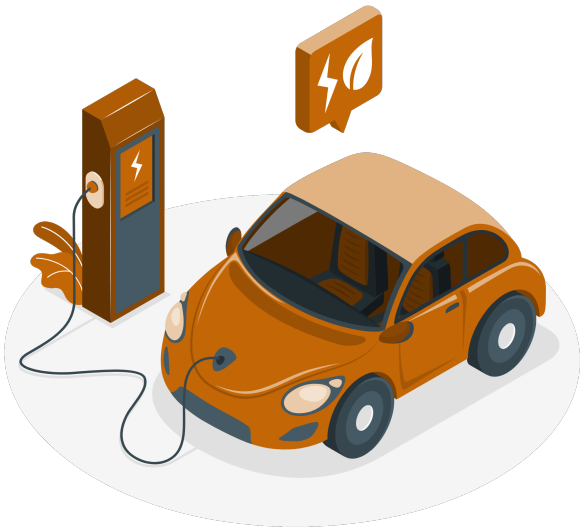
ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO



ESTRUTURAS DE INFORMAÇÃO

Projeto Integrador

Relatório

Sprint 3

GRUPO

1221184 – Luna Silva

1221014 – Diogo Moutinho

1220858 – José Mendes

1220688 – Pedro Coelho

PROFESSOR RESPONSÁVEL

Alberto Sampaio (ACS)

03/01/2024

Índice

Índice de Imagens

Introdução

No âmbito da disciplina Estruturas de Informação, juntamente com o projeto integrador, foi-nos proposto criar uma aplicação para a empresa GFH.

A empresa GFH desempenha um papel crucial na distribuição eficiente de cabazes contendo uma variedade de produtos agrícolas numa rede interconectada. Neste contexto, a Estrutura de Informação assume um papel fundamental na implementação de um sistema robusto para gerenciar a rede de distribuição.

A GFH atua como intermediária entre os produtores, que fornecem uma diversidade de produtos agrícolas, e os clientes, que solicitam cabazes destes produtos. Cada cabaz representa uma encomenda específica, composta por uma lista de produtos provenientes de um produtor específico. Os hubs desempenham um papel crucial como locais de entrega e recolha, sendo associados a instituições como universidades, hospitais, ginásios e empresas.

O projeto, fazendo uso de classes que implementam a interface Graph, tem como objetivo desenvolver um conjunto de classes e testes para gerir de forma eficiente a rede de distribuição de cabazes. A rede é representada por vértices que correspondem a localidades onde hubs de distribuição podem estar presentes.

A implementação é orientada pelos seguintes requisitos:

- USEI06: Identificar para um produtor os diferentes percursos possíveis entre um ponto de origem e um hub, limitados pela autonomia do veículo elétrico. Este tópico será implementado pelo José.
- USEI07: Determinar para um produtor o percurso de entrega que maximiza o número de hubs pelos quais passa, considerando horários de funcionamento, tempo de descarga, distâncias e velocidade média. Este tópico será implementado pela Luna.
- USEI08: Encontrar para um produtor o circuito de entrega que parte de um ponto de origem, passa por um número específico de hubs uma só vez e retorna ao ponto de origem, minimizando a distância total percorrida. Este tópico será implementado pelo Diogo.
- USEI09: Organizar as localidades do grafo em clusters, garantindo apenas um hub por cluster. Os clusters devem ser obtidos iterativamente pela remoção de ligações com o maior número de caminhos mais curtos até que clusters isolados sejam formados. Este tópico será implementado pelo Pedro.

O desenvolvimento do código em Java, utilizando os grafos, será essencial para atender a esses requisitos, proporcionando uma solução eficiente e escalável para a distribuição de produtos agrícolas pela GFH.

Domain Model

Disclaimer: Dada a extensão do Modelo de Domínio, caracterizada por um número significativo de classes e métodos, decidimos não o apresentar neste contexto, pois sua exibição seria limitada em tamanho, comprometendo a legibilidade e visibilidade. No entanto, para uma análise mais detalhada, o Modelo de Domínio completo pode ser encontrado no ambiente de desenvolvimento IntelliJ IDEA.

Para aceder o Modelo de Domínio no IntelliJ, siga os seguintes passos:

1. Abra a documentação do projeto, que está disponível no diretório "docs" na raiz do repositório.
2. Navegue até a subpasta "ESINF", que contém informações específicas do projeto.
3. Dentro de "ESINF", localize o diretório referente ao sprint atual, neste caso, o "Sprint 3".
4. Dentro do "Sprint 3", procure pelo arquivo chamado "Domain.png".

Este arquivo, "Domain.png", é o Modelo de Domínio visual que proporciona uma representação gráfica clara e concisa das entidades e relações fundamentais no sistema. A sua visualização no IntelliJ IDEA simplifica a compreensão da arquitetura do projeto e facilita o processo de desenvolvimento, permitindo uma abordagem mais eficiente e organizada para as tarefas relacionadas ao domínio da aplicação.

USEI06

Descrição

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

Implementação

getPathsBetweenTwoPoints()

```
2 usages   José +1 *
public static TreeMap<LinkedList<Localidades>, PathInfo> getPathsBetweenTwoPoints(MapGraph<Localidades, Integer> graph,
                                                                                   Localidades startingLocalidades,
                                                                                   Localidades destinationLocalidades,
                                                                                   Integer vehicleAutonomy,
                                                                                   double tripAverageSpeed) {

    TreeMap<LinkedList<Localidades>, PathInfo> pathsWithCostAndTime = new TreeMap<>(Comparator.comparingInt(o -> calculatePathCost(graph, o)));

    ArrayList<LinkedList<Localidades>> allPaths = Algorithms.dfsAlgorithm(graph, startingLocalidades, destinationLocalidades, vehicleAutonomy);

    for (LinkedList<Localidades> fullPath : allPaths) {
        int cost = calculatePathCost(graph, fullPath);

            double totalTime = calculateTotalTime(graph, fullPath, tripAverageSpeed);
            Map<LocalityPair, Integer> individualDistances = calculateIndividualDistances(graph, fullPath);
            PathInfo pathInfo = new PathInfo(cost, totalTime, individualDistances);
            pathsWithCostAndTime.put(fullPath, pathInfo);
    }
    return pathsWithCostAndTime;
}
```

O método getPathsBetweenTwoPoints é responsável por encontrar todos os caminhos possíveis entre dois pontos em um grafo ponderado, considerando a autonomia do veículo e a velocidade média da viagem. O resultado é um TreeMap que mapeia cada caminho encontrado para informações detalhadas sobre o custo, tempo total e distâncias individuais.

O método utiliza um algoritmo de busca em profundidade (DFS) para encontrar todos os caminhos possíveis entre a origem e o destino. Para cada caminho encontrado, calcula o custo total do caminho, o tempo total de viagem e as distâncias individuais entre as localidades ao longo do caminho. As informações são então armazenadas em um TreeMap, onde a chave é o caminho e o valor é um objeto PathInfo contendo as informações relevantes.

calculatePathCost()

```
2 usages   35193 +1
private static int calculatePathCost(MapGraph<Localidades, Integer> graph, LinkedList<Localidades> path) {
    int cost = 0;
    Iterator<Localidades> iterator = path.iterator();
    Localidades currentLocalidades = iterator.next();

    while (iterator.hasNext()) {
        Localidades nextLocalidades = iterator.next();
        cost += graph.edge(currentLocalidades, nextLocalidades).getWeight();
        currentLocalidades = nextLocalidades;
    }
    return cost;
}
```

O método calculatePathCost calcula o custo total de um caminho em um grafo ponderado, somando os pesos das arestas ao longo do caminho.

O método utiliza um iterador para percorrer o caminho e soma os pesos das arestas consecutivas. O resultado é o custo total do caminho.

calculateIndividualDistances()

```
1 usage   José +1
private static Map<LocalityPair, Integer> calculateIndividualDistances(MapGraph<Localidades, Integer> graph, LinkedList<Localidades> path) {
    Map<LocalityPair, Integer> individualDistances = new HashMap<>();
    Iterator<Localidades> iterator = path.iterator();
    Localidades currentLocalidades = iterator.next();

    while (iterator.hasNext()) {
        Localidades nextLocalidades = iterator.next();
        int distance = graph.edge(currentLocalidades, nextLocalidades).getWeight();
        individualDistances.put(new LocalityPair(currentLocalidades, nextLocalidades), distance);
        currentLocalidades = nextLocalidades;
    }

    return individualDistances;
}
```

O método calculateIndividualDistances calcula as distâncias individuais entre localidades ao longo de um caminho em um grafo ponderado.

O método utiliza um iterador para percorrer o caminho, obtém as distâncias entre localidades consecutivas e armazena essas distâncias em um mapa, onde as chaves são pares de localidades e os valores são as distâncias correspondentes.

calculateTotalTime()

```
1 usage   35193 +1
private static double calculateTotalTime(MapGraph<Localidades, Integer> graph, LinkedList<Localidades> path, double averageSpeed) {
    double totalTime = 0.0;
    Iterator<Localidades> iterator = path.iterator();
    Localidades currentLocalidades = iterator.next();

    while (iterator.hasNext()) {
        Localidades nextLocalidades = iterator.next();
        double distance = graph.edge(currentLocalidades, nextLocalidades).getWeight();
        double time = distance / averageSpeed;      // Tempo = Distância / Velocidade Média
        totalTime += time;
        currentLocalidades = nextLocalidades;
    }
    return totalTime;
}
```

O método calculateTotalTime calcula o tempo total de uma viagem ao longo de um caminho em um grafo ponderado, considerando a velocidade média.

O método utiliza um iterador para percorrer o caminho, obtém as distâncias entre localidades consecutivas e calcula o tempo correspondente com base na fórmula Tempo = Distância / Velocidade Média. O resultado é o tempo total de viagem.

dfsAlgorithm()

```
public static <V, E extends Comparable<E>> ArrayList<LinkedList<V>> dfsAlgorithm(Graph<V, E> g, V vOrig, V vDest, E maxWeight) {
    ArrayList<LinkedList<V>> paths = new ArrayList<>();
    Set<V> visitedSet = new HashSet<>();
    LinkedList<V> currentPath = new LinkedList<>();
    Integer totalHeight = 0;

    if (!g.validVertex(vOrig) || !g.validVertex(vDest)) {
        System.out.println("Invalid vertices");
        return paths;
    }

    dfsAlgorithm(g, vOrig, vDest, maxWeight, visitedSet, currentPath, paths, totalHeight);

    return paths;
}

private static <V, E extends Comparable<E>> void dfsAlgorithm(Graph<V, E> g, V vOrig, V vDest, E maxWeight, Set<V> visitedSet,
                                                               LinkedList<V> path, ArrayList<LinkedList<V>> paths, Integer totalWeight) {

    visitedSet.add(vOrig);
    path.add(vOrig);

    if (vOrig.equals(vDest)) {
        paths.add(new LinkedList<>(path));
        return;
    }else {
        for (V neighbor : g.adjVertices(vOrig)) {
            Integer weight = (Integer)( g.edge(vOrig, neighbor)).getWeight();

            if (!visitedSet.contains(neighbor) && (totalWeight + weight) < (Integer)maxWeight) {

                totalWeight += weight;
                dfsAlgorithm(g, neighbor, vDest, maxWeight, visitedSet, path, paths, totalWeight);
            }
        }
    }
    visitedSet.remove(vOrig);
    path.removeLast();
}
```

O código em questão implementa um algoritmo de Busca em Profundidade (DFS) num grafo, fornecendo funcionalidades essenciais para explorar caminhos entre vértices, considerando os pesos máximos, que no contexto desta US são a autonomia máxima do veículo. O algoritmo é encapsulado em dois métodos dfsAlgorithm.

- Método Público dfsAlgorithm:

O método dfsAlgorithm público serve como ponto de entrada para a execução do algoritmo DFS. Este método inicializa as estruturas de dados necessárias, como conjuntos para controlar os vértices visitados, uma lista para armazenar os caminhos encontrados, e uma lista encadeada para representar o caminho atual.

Este método realiza verificações preliminares para garantir que os vértices fornecidos são válidos. Caso contrário, exibe uma mensagem de erro e retorna uma lista vazia. Em seguida, invoca o método privado dfsAlgorithm para a execução real do algoritmo.

- Método Privado dfsAlgorithm:

O método privado dfsAlgorithm é responsável pela execução real do algoritmo DFS. Este método utiliza recursividade para explorar os caminhos entre os vértices do grafo, considerando os pesos máximos definidos, ou seja, a autonomia máxima do automóvel que vai representar a distância máxima que vai conseguir percorrer sem carregar.

O método inicia marcando o vértice de origem como visitado e adiciona ao caminho(Path). Em seguida, verifica se o vértice atual é o destino. Se sim, adiciona o caminho à lista de caminhos encontrados. Caso contrário, explora os vizinhos não visitados, garantindo que o peso total do caminho não ultrapasse o máximo permitido. A recursividade continua até que todos os caminhos possíveis sejam explorados.

Após a conclusão da exploração de um caminho, o vértice atual é removido do conjunto de visitados e do caminho, permitindo a exploração de outros caminhos a partir do vértice anterior.

USEI06 - Análise da complexidade

A busca em profundidade (DFS) é como uma "exploração" em um labirinto, onde se vai o mais fundo possível antes de retornar e explorar outras opções. Esse algoritmo é usado para navegar em grafos, onde os "caminhos" são os links entre pontos. Analisando a sua compelxidade, podemos reparar que:

- Cada vértice e cada aresta são visitados uma vez.
- loop principal do DFS é executado uma vez para cada vértice não visitado.
- Em cada execução do loop, cada aresta conectada ao vértice atual é verificada.

Assim, a complexidade de tempo no pior caso é O(|V| + |E|), onde |V| é o número de vértices e |E| é o número de arestas.

Nos restantes metodos de calcular o custo total, custo de vertice a vertice e tempo total, a complexidade temporal é O (N), onde N é o número de vértices no caminho, visto que é usado um iterador que percorre todos e realiza operações constantes como a obtenção do peso da aresta e a sua soma.

USEI07

Descrição

Encontrar para um produtor que parte de um local origem 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 de carregamento do veículo.

Explicação das mudanças feitas na US_EI02

Para implementar esta US, tiveram que ser feitas algumas modificações ao nível da USEI02. Sendo assim foram acrescentados 3 métodos à US previamente implementada.

A incorporação desses novos métodos permitiu a atualização do MaphGraph, destacando n localidades com características mais propícias para se tornarem hubs. Como resultado, essas localidades foram marcadas como hubs, com o parâmetro "isHub" sendo configurado como verdadeiro.

```
public void setHubs(Map<Localidades, List<Integer>> topHubs, int n) {
    US_EI01_GraphBuilder rede = US_EI01_GraphBuilder.getInstance();
    MapGraph<Localidades, Integer> graph = rede.getDistribuicao();

    getTopNHubs(topHubs, n);
    for (Localidades local: graph.vertices()) {
        if (topHubs.containsKey(local)){
            graph.vertex(p -> p.equals(local)).setHub(true);
        }
    }
}
```

Figura 1 – setHubs

Este método recebe um mapa de localidades (topHubs) associado a uma lista de inteiros e um valor inteiro n de forma a conseguir calcular as melhores localidades para Hubs. Ele utiliza a classe US_EI01_GraphBuilder para obter uma instância do grafo (MapGraph<Localidades, Integer> graph) que representa a distribuição original de cabazes.

O método então chama getTopNHubs(topHubs, n) para obter as n melhores localidades (hubs) com base nas características fornecidas no mapa topHubs. Posteriormente, percorre todas as

localidades no grafo e, se uma localidade estiver presente no mapa topHubs, configura o atributo "isHub" do vértice correspondente no grafo como verdadeiro.

```
public MapGraph<Localidades, Integer> getMapGraph() {
    Map<Localidades, Integer> influence = calculateInfluence(graph);
    Map<Localidades, Integer> proximity = calculateProximity(graph);
    Map<Localidades, Integer> centrality = calculateCentrality(graph);

    Map<Localidades, List<Integer>> combinedMap = new HashMap<>();
    for (Localidades localidades : graph.vertices()) {
        List<Integer> values = new ArrayList<>();
        values.add(centrality.get(localidades));
        values.add(influence.get(localidades));
        values.add(proximity.get(localidades));
        combinedMap.put(localidades, values);
    }

    Map<Localidades, List<Integer>> topNMap =getTopNMap(combinedMap, n);
    setHubs(topNMap, n);

    return graph;
}
```

Figura 2 – getMapGraph

Este método calcula e retorna um grafo (MapGraph<Localidades, Integer> graph) que representa a distribuição de cabazes após terem sido escolhidas as melhores localidades para Hubs. Para isso, calcula três métricas distintas (influence, proximity e centrality) para cada localidade no grafo. Em seguida, combina essas métricas num mapa (combinedMap) que associa cada localidade a uma lista de inteiros representando suas características.

Utilizando getTopNMap(combinedMap, n), obtém as n localidades principais e, em seguida, chama setHubs(topNMap, n) para marcá-las como hubs no grafo. Finalmente, retorna o grafo atualizado.

```
public Map<Localidades, Integer> getMapHubs(){
    Map<Localidades, Integer> influence = calculateInfluence(graph);
    Map<Localidades, Integer> proximity = calculateProximity(graph);
    Map<Localidades, Integer> centrality = calculateCentrality(graph);

    Map<Localidades, List<Integer>> combinedMap = new HashMap<>();
    for (Localidades localidades : graph.vertices()) {
        List<Integer> values = new ArrayList<>();
        values.add(centrality.get(localidades));
        values.add(influence.get(localidades));
        values.add(proximity.get(localidades));
        combinedMap.put(localidades, values);
    }

    Map<Localidades, Integer> hubs = new HashMap<>();
    getTopNMap(combinedMap, n).forEach((key, value) -> hubs.put(key, value.get(0)));

    return hubs;
}
```

Figura 3 - getMapHubs

Este método realiza um processo semelhante ao anterior, calculando as métricas de influence, proximity e centrality para cada localidade no grafo. Em seguida, combina essas métricas num mapa (combinedMap). A partir desse mapa, obtém as n localidades principais utilizando getTopNMap(combinedMap, n) e cria um novo mapa (hubs) associando cada localidade à sua métrica de centralidade.

O método retorna o mapa hubs, que contém as n localidades mais propensas a serem e as suas métricas de centralidade.

```
private void escolherHorario(MapGraph<Localidades, Integer> graph) {
    for (Localidades localidades : graph.vertices()) {
        String numId = localidades.getNumId();

        // Extrair o número da localidade (ignorando o prefixo "CT")
        int numeroLocalidade = Integer.parseInt(numId.substring( beginIndex: 2));

        // Lógica para definir horários apenas se for um hub
        if (localidades.isHub()) {
            LocalTime horarioAbertura;
            LocalTime horarioFecho;

            // Aplicar lógica para definir os horários com base no número da localidade
            if (numeroLocalidade >= 1 && numeroLocalidade <= 105) {
                horarioAbertura = LocalTime.of( hour: 9,  minute: 0);
                horarioFecho = LocalTime.of( hour: 14,  minute: 0);
            } else if (numeroLocalidade >= 106 && numeroLocalidade <= 215) {
                horarioAbertura = LocalTime.of( hour: 11,  minute: 0);
                horarioFecho = LocalTime.of( hour: 16,  minute: 0);
            } else if (numeroLocalidade >= 216 && numeroLocalidade <= 324) {
                horarioAbertura = LocalTime.of( hour: 12,  minute: 0);
                horarioFecho = LocalTime.of( hour: 17,  minute: 0);
            } else {
                // Adicione lógica adicional conforme necessário para outros casos
                horarioAbertura = LocalTime.of( hour: 0,  minute: 0);
                horarioFecho = LocalTime.of( hour: 0,  minute: 0);
            }

            // Criar Horario com os parâmetros fornecidos
            Horario horario = new Horario(horarioAbertura, horarioFecho);
            localidades.setHorario(horario);
        }
    }
}
```

Figura 4 - escolherHorario

O método recebe como parâmetro um grafo (MapGraph) contendo as localidades e as informações relacionadas a elas, sendo que as localidades já estão atualizadas de acordo com a escolha de hubs. Para cada localidade no grafo, o método verifica se a localidade é um hub usando o método isHub(). Se a localidade for um hub, o método extrai o número da localidade a partir do ID. O ID da localidade é uma sequência de caracteres, e o número da localidade é obtido removendo o prefixo "CT" e convertendo o restante para um número inteiro. Com base no número da localidade, o método aplica uma lógica para determinar os horários de abertura e fecho. A lógica envolve faixas específicas de números de localidades e associa horários correspondentes a essas faixas. Se o número

da localidade não se encaixar em nenhuma das faixas especificadas, o método define horários padrão (nesse caso, ambos os horários são definidos como meia-noite).

Com os horários de abertura e fecho determinados, o método cria um objeto Horario usando a classe Horario, que parece ser uma classe definida em outro lugar no código (não fornecida neste trecho). A classe Horario provavelmente contém informações sobre os horários de abertura e fechamento. O método então associa o objeto Horario recém-criado à localidade utilizando o método setHorario().

Explicação da implementação da US_EI07

getPontoPartida()

```
public String getPontoPartida(Localidades pontoPartida) {
    StringBuilder info = new StringBuilder();

    for (Localidades localidades : graphMod.vertices()) {
        if (localidades.getNumId().equals(pontoPartida.getNumId())) {
            info.append("NumId: ").append(localidades.getNumId()).append(" ");
            info.append("Coordenadas: ").append(localidades.getCoordenadas());
            pontoPartida.setCoordenadas(localidades.getCoordenadas());
        }
    }
    return info.toString();
}
```

Figura 5 - getPontoPartida

Este método realiza a busca de informações sobre o ponto de partida de uma rota de entrega. Ele recebe como parâmetro uma localidade que representa o ponto inicial da rota. O método percorre as localidades no grafo e compara os identificadores (NumId). Quando encontra a correspondência, extrai informações como o NumId e coordenadas dessa localidade.

calculateMelhorPercurso()

```java
public static StructurePath calculaMelhorPercurso(Localidades localInicio, LocalTime hora, int autonomia, double averageVelocity, int tempoRecarga, int tempoDescarga) {
    int hubsAindaAbertosNumero = 0;
    Localidades tempLocal = localInicio;
    LocalTime tempHora = hora, tempoRestante = LocalTime.of( hour: 0,  minute: 0), horaInicial = hora;
    List<Localidades> locaisVisitados = new ArrayList<>();
    LinkedList<Localidades> tempCaminho = new LinkedList<>(), melhorCaminho = new LinkedList<>();

    while (tempoRestante != null) {
        tempoRestante = null;
        for (Localidades hub : getHubs()) {
            if (!locaisVisitados.contains(hub)) {
                LinkedList<Localidades> caminho = new LinkedList<>();
                Algorithms.shortestPathWithAutonomy(graphMod, autonomia, tempLocal, hub, Comparator.naturalOrder(), Integer::sum,  zero: 0, caminho);
                if (getTempoFinalCompleto(caminho, hora, autonomia, averageVelocity, tempoRecarga, tempoDescarga, getAllHubsInCourse(caminho, locaisVisitados).size()).isBefore(hub.getHorario().getCloseTim
                    if (tempoRestante == null || getStillOpenHubs(getTempoFinalCompleto(caminho, hora, autonomia, averageVelocity, tempoRecarga, tempoDescarga, getAllHubsInCourse(caminho, locaisVisitados)
                        tempCaminho = caminho;
                        tempoRestante = minusTime(hub.getHorario().getCloseTime(), getTempoFinalCompleto(caminho, hora, autonomia, averageVelocity, tempoRecarga, tempoDescarga, getAllHubsInCourse(caminho,
                        tempHora = getTempoFinalCompleto(caminho, hora, autonomia, averageVelocity, tempoRecarga, tempoDescarga, getAllHubsInCourse(caminho, locaisVisitados).size());
                        hubsAindaAbertosNumero = getStillOpenHubs(tempHora).size();
                    }
                }
            }
        }
        if (tempoRestante != null) {
            if (melhorCaminho.isEmpty()) {
                melhorCaminho.addAll(tempCaminho);
            } else {
                melhorCaminho.removeLast();
                melhorCaminho.addAll(tempCaminho);
            }

            tempLocal = melhorCaminho.get(melhorCaminho.size() - 1);
            for (int i = 0; i < getAllHubsInCourse(tempCaminho, locaisVisitados).size(); i++) {
                locaisVisitados.add(getAllHubsInCourse(tempCaminho, locaisVisitados).get(i));
            }
            hora = tempHora;
        }
    }
```

Figura 6 – calculateMelhorPercurso

Este método é responsável por determinar o percurso ideal para uma rota de entrega. Recebe como parâmetros o ponto inicial, o horário inicial, a autonomia do veículo, a velocidade média, o tempo de carregamento e o tempo de descarregamento de mercadorias.

O método utiliza um loop while que continua a executar até não haver mais hubs viáveis a serem visitados. Dentro do loop, realiza a procura dos melhores horários disponíveis para os hubs não visitados. O próximo hub a ser visitado é escolhido com base no horário mais vantajoso.

A lógica do método envolve a iteração pelos hubs, cálculo do caminho mais curto levando em consideração autonomia e horários, e atualização das variáveis que controlam o percurso. O resultado final é encapsulado em uma estrutura StructurePath que contém informações detalhadas sobre o percurso otimizado.

Este método utiliza o algoritmo shortestPathWithAutonomy que é uma adaptação do algoritmo de Dijkstra que leva em consideração a autonomia de um veículo ao calcular o caminho mais curto em um grafo ponderado.

```java
public static <V, E> E shortestPathWithAutonomy(Graph<V, E> g,E autonomy, 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];
    V[] pathKeys = (V[]) new Object[numVerts];
    E[] dist = (E[]) new Object[numVerts];
    initializePathDist(numVerts, pathKeys, dist);

    shortestPathDijkstraWithAutonomy(g, autonomy, vOrig, ce, sum, zero, visited, pathKeys, dist);

    E lengthPath = dist[g.key(vDest)];

    if (lengthPath != null) {
        getPath(g, vOrig, vDest, pathKeys, shortPath);
        return lengthPath;
    }

    return null;
}
```

Figura 7- shortestPathWithAutonomy

O método inicia inicializando arrays e variáveis necessárias para rastrear os vértices visitados, os predecessores no caminho mais curto e as distâncias acumuladas até cada vértice.

O algoritmo central é invocado através do método shortestPathDijkstraWithAutonomy, que implementa a lógica específica do Dijkstra modificado para levar em consideração a autonomia do veículo.

```java
public static <V, E> void shortestPathDijkstraWithAutonomy(Graph<V, E> g,E autonomia, V vOrig,
                                                           Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                                           boolean[] visited, V[] pathKeys, E[] dist) {

    int vkey = g.key(vOrig);
    dist[vkey] = zero;
    pathKeys[vkey] = vOrig;

    while (vOrig != null) {
        vkey = g.key(vOrig);
        visited[vkey] = true;

        for (Edge<V, E> edge : g.outgoingEdges(vOrig)) {
            int vkeyAdj = g.key(edge.getVDest());
            if (!visited[vkeyAdj]) {
                E remainingAutonomy = sum.apply(autonomia, dist[vkey]);
                if (ce.compare(edge.getWeight(), remainingAutonomy) <= 0) {
                    E s = sum.apply(dist[vkey], edge.getWeight());
                    if (dist[vkeyAdj] == null || ce.compare(dist[vkeyAdj], s) < 0) {
                        dist[vkeyAdj] = s;
                        pathKeys[vkeyAdj] = vOrig;
                    }
                }
            }
        }

        E minDist = null;
        vOrig = null;

        for (V vert : g.vertices()) {
            int i = g.key(vert);
            if (!visited[i] && (dist[i] != null) && ((minDist == null) || ce.compare(dist[i], minDist) < 0)) {
                minDist = dist[i];
                vOrig = vert;
            }
        }
    }
}
```

Figura 8 – shortestPathDijkstraWithAutonomy

O método principal usa um loop para explorar os vértices do grafo. Em cada iteração, seleciona o vértice não visitado com a menor distância acumulada. Marca esse vértice como visitado e atualiza as distâncias acumuladas dos vértices adjacentes, considerando a autonomia do veículo.

Dentro do loop, para cada vértice adjacente não visitado, o algoritmo verifica se o caminho até esse vértice, passando pelo vértice atual, é mais curto do que o caminho conhecido anteriormente. Essa comparação leva em conta a autonomia restante do veículo, garantindo que o veículo possa percorrer a distância entre os vértices.

getTempoFinalCompleto()

```java
public static LocalTime getTempoFinalCompleto(LinkedList<Localidades> caminhoPercorrido, LocalTime horaComeco, int autonomia, double averageVelocity, int tempoRecarga, int tempoDescarga, int numeroDescargas) {
    LocalTime horaFim = LocalTime.of( hour: 0,  minute: 0);
    StructurePath structurePath = analyzeData(autonomia, caminhoPercorrido);
    horaFim = addTime(horaFim, getFinishingTimeRoute(structurePath.getDistanciaTotal(), averageVelocity, horaComeco));
    for (int i = 0; i < structurePath.getCarregamentos().size(); i++) {
        horaFim = addTime(horaFim, intToLocalTime(tempoRecarga));
    }
    for (int i = 0; i < numeroDescargas; i++) {
        horaFim = addTime(horaFim, intToLocalTime(tempoDescarga));
    }
    return horaFim;
}
```

Figura 9 – getTempoFinalCompleto

Este método desempenha um papel fundamental ao calcular o tempo total necessário para percorrer uma rota de entrega. Primeiramente, o método recebe informações essenciais, como o caminho percorrido representado pela lista de localidades caminhoPercorrido, o horário inicial da rota (horaComeco), a autonomia do veículo, a velocidade média, o tempo de recarga e o tempo de descarga de mercadorias.

O processo começa inicializando a variável horaFim como meia-noite (00:00), que será gradualmente atualizada com os tempos necessários ao longo da rota.

O método calcula o tempo necessário para percorrer a distância total do caminho, considerando a velocidade média e o horário de início. De seguida, itera sobre os pontos de recarga no caminho, adicionando o tempo de recarga correspondente a cada parada. Além disso, considera o tempo de descarga de mercadorias ao iterar sobre o número de descargas no caminho.

O resultado final é o tempo total necessário para completar a rota, incorporando aspetos críticos como distância, velocidade, horário de início, tempo de recarga e tempo de descarga de mercadorias.

intToLocalTime()

```
public static LocalTime intToLocalTime(int tempo) {
    double tempoDouble = (double) tempo / 60;
    LocalTime tempoPercurso = LocalTime.of((int) tempoDouble, (int) ((tempoDouble - (int) tempoDouble) * 60));
    return tempoPercurso;
}
```

Figura 10 – intToLocalTime

Este método é responsável por converter um valor de tempo expresso em minutos para o formato de tempo utilizado pela classe LocalTime no Java, que representa horas, minutos e segundos.

Ao receber como parâmetro um valor de tempo em minutos (um número inteiro), o método realiza a conversão desse valor para horas e minutos, considerando a parte decimal correspondente aos minutos. Em seguida, utiliza a classe LocalTime para criar um objeto que representa o tempo equivalente, tratando a parte inteira como horas e a parte decimal como minutos.

getFinishingTimeRoute()

```
public static LocalTime getFinishingTimeRoute(int distanciaTotal, double averageVelocity, LocalTime horaComeco) {
    LocalTime horaFim;
    double tempoPercursoDouble = ((double) (distanciaTotal) / 1000) / averageVelocity;
    LocalTime tempoPercurso = LocalTime.of((int) tempoPercursoDouble, (int) ((tempoPercursoDouble - (int) tempoPercursoDouble) * 60));
    horaFim = addTime(horaComeco, tempoPercurso);
    return horaFim;
}
```

Figura 11 – getFinishingTimeRoute

Este método desempenha a função de calcular o horário de término de uma rota de entrega. Ele recebe como parâmetros a distância total da rota, a velocidade média do veículo e o horário de início da rota. Utilizando a relação básica entre velocidade, distância e tempo, o método calcula o tempo necessário para percorrer toda a distância com a velocidade média fornecida.

Em seguida, converte esse tempo de percurso para o formato LocalTime do Java, que representa horas, minutos e segundos. O horário de término é obtido somando-se esse tempo de

percurso ao horário de início da rota. O resultado final é o horário de término da rota, expresso como um objeto LocalTime.

getTimeTable ()

```
public static Map<Localidades, List<LocalTime>> getTimeTable(StructurePath structurePath, LocalTime horaComeco, int autonomia, double averageVelocity, int tempoRecar
    Map<Localidades, List<LocalTime>> timeTable = new LinkedHashMap<>();
    for (int i = 1; i < structurePath.getPercurso().size(); i++) {
        LocalTime afterEverything = getFinishingTimeRoute(graphMod.edge(structurePath.getPercurso().get(i - 1), structurePath.getPercurso().get(i)).getWeight(), aver

        List<LocalTime> listOfTimes = new ArrayList<>();
        if (structurePath.getCarregamentos().contains(i)) {
            if (structurePath.getPercurso().get(i).isHub()) {
                listOfTimes.add(afterEverything.plusMinutes(tempoDescarga));
                listOfTimes.add(afterEverything.plusMinutes(tempoRecarga));
                timeTable.put(structurePath.getPercurso().get(i), listOfTimes);
                horaComeco = afterEverything.plusMinutes(tempoRecarga).plusMinutes(tempoDescarga);
            } else {
                listOfTimes.add(afterEverything);
                listOfTimes.add(afterEverything.plusMinutes(tempoRecarga));
                timeTable.put(structurePath.getPercurso().get(i), listOfTimes);
                horaComeco = afterEverything.plusMinutes(tempoRecarga);
            }
        } else {
            if (structurePath.getPercurso().get(i).isHub()) {
                listOfTimes.add(afterEverything);
                listOfTimes.add(afterEverything.plusMinutes(tempoDescarga));
                timeTable.put(structurePath.getPercurso().get(i), listOfTimes);
                horaComeco = afterEverything.plusMinutes(tempoDescarga);
            } else {
                listOfTimes.add(afterEverything);
                listOfTimes.add(afterEverything);
                timeTable.put(structurePath.getPercurso().get(i), listOfTimes);
                horaComeco = afterEverything;
            }
        }
    }
    return timeTable;
}
```

Figura 12 – getTimeTable

Este método recebe como entrada a estrutura StructurePath, que representa a rota de entrega, juntamente com informações como o horário de início da rota (horaComeco), autonomia do veículo, velocidade média, tempo de recarga e tempo de descarga, o método realiza uma iteração sobre os locais da rota.

Para cada local, calcula os horários associados, considerando se o local é um hub, o que pode impactar nos tempos de descarga e recarga. Esses horários são então utilizados para construir uma tabela detalhada, onde cada localidade é mapeada para uma lista de horários relevantes.

Além disso, o método atualiza dinamicamente o horário de início da rota, levando em conta os momentos de recarga e descarga ao longo do percurso. O resultado final é uma tabela que fornece uma visão abrangente dos horários estimados de chegada e dos momentos críticos de recarga e descarga em cada ponto da rota.

getAllHubsInCourse()

```
public static List<Localidades> getAllHubsInCourse(LinkedList<Localidades> caminho, List<Localidades> locaisVisitados) {
    List<Localidades> lista = new ArrayList<>();
    for (int i = 1; i < caminho.size(); i++) {
        if (caminho.get(i).isHub()) {
            lista.add(caminho.get(i));
        }
    }
    return lista;
}
```

Figura 13 – getAllHubsInCourse

Este método vai identificar todos os hubs ao longo de um determinado percurso de entrega. Ele recebe como entrada uma lista encadeada de localidades caminho, que representa o percurso de entrega, e uma lista de localidades locaisVisitados contendo os locais já visitados até o momento, o método itera sobre o percurso.

Durante a iteração, verifica se cada localidade no percurso é um hub. Se a localidade for identificada como um hub, ela é adicionada a uma lista denominada lista de hubs.

O resultado final do método é a lista lista, que contém todos os hubs encontrados ao longo do percurso de entrega.

addTime()

```
public static LocalTime addTime(LocalTime time1, LocalTime time2) {
    return time1.plusHours(time2.getHour()).plusMinutes(time2.getMinute()).plusSeconds(time2.getSecond());
}
```

Figura 14 – addTime

Ao receber como entrada dois objetos LocalTime, time1 e time2, representando intervalos de tempo, o método utiliza os métodos plusHours, plusMinutes e plusSeconds da classe LocalTime para realizar a adição de horas, minutos e segundos, respectivamente.

O resultado final é um novo objeto LocalTime que representa a soma dos tempos time1 e time2.

minusTime()

```
public static LocalTime minusTime(LocalTime time1, LocalTime time2) {
    return time1.minusHours(time2.getHour()).minusMinutes(time2.getMinute()).minusSeconds(time2.getSecond());
}
```

Figura 15 – minusTime

Este método recebe dois objetos LocalTime, time1 e time2, o método utiliza os métodos minusHours, minusMinutes e minusSeconds da classe LocalTime para calcular a diferença em horas, minutos e segundos entre esses tempos.

O resultado final é um novo objeto LocalTime representando a diferença entre time1 e time2.

getStillOpenHubs()

```
public static List<Localidades> getStillOpenHubs(LocalTime hora) {
    List<Localidades> result = new ArrayList<>();
    for (Localidades local : getHubs()) {
        if (local.isHub() && local.getHorario().getOpenTime().isBefore(hora) && local.getHorario().getCloseTime().isAfter(hora)) {
            result.add(local);
        }
    }
    return result;
}
```

Figura 16 – getStillOpenHubs

Este método desempenha a função de identificar e retornar uma lista de hubs que permanecem abertos num determinado horário.

Ao receber como entrada um objeto LocalTime, representando o horário específico para análise, o método itera sobre todos os hubs no grafo. Durante essa iteração, verifica-se se cada hub está aberto nesse horário, comparando o horário de abertura (OpenTime) e o horário de fecho (CloseTime) do hub com o horário fornecido.

A condição para inclusão de um hub na lista de hubs abertos é que o horário fornecido esteja posterior ao horário de abertura e anterior ao horário de fecho do hub.

getHubs()

```
public static List<Localidades> getHubs() {
    List<Localidades> result = new ArrayList<>();
    for (Localidades local : graphMod.vertices()) {
        if (local.isHub()) {
            result.add(local);
        }
    }
    return result;
}
```

Figura 17 - getHubs

Desempenha a função de recuperar e retornar uma lista contendo todos os hubs presentes no grafo logístico. Ele realiza isso iterando sobre os vértices (locais) no grafo e identificando quais deles são considerados hubs, com base na propriedade isHub da classe Localidades. Os locais que atendem a essa condição são adicionados a uma lista, que é então retornada como o resultado final.

analyzeData()

```
public static StructurePath analyzeData(int autonomia, LinkedList<Localidades> caminho) {
    ArrayList<Integer> indexDeCarregamentos = new ArrayList<>();
    LinkedList<Localidades> percurso = caminho;
    int distanciaPercorrida = 0, bateria = autonomia;
    boolean flag = true;
    if (percurso != null) {
        for (int i = 0; i < percurso.size() - 1; i++) {
            int distanciaEntrePontos = graphMod.edge(percurso.get(i), percurso.get(i + 1)).getWeight();
            distanciaPercorrida += distanciaEntrePontos;
            if (distanciaEntrePontos / 1000 > bateria) {
                if (distanciaEntrePontos / 1000 <= autonomia) {
                    indexDeCarregamentos.add(i);
                    bateria = autonomia;
                } else {
                    flag = false;
                }
            } else {
                bateria -= distanciaEntrePontos / 1000;
            }
        }
    } else {
        flag = false;
        return new StructurePath(distanciaPercorrida, percurso, indexDeCarregamentos, flag);
    }
    return new StructurePath(distanciaPercorrida, percurso, indexDeCarregamentos, flag);
}
```

Figura 18 – analyzeData

Recebe como entrada a autonomia do veículo e a lista encadeada de locais que compõem o percurso, o método inicializa variáveis importantes.

Durante a iteração pelo percurso, calcula a distância entre locais consecutivos, ajustando a autonomia do veículo de acordo. Se a distância entre dois locais ultrapassar a autonomia disponível, verifica a possibilidade de recarga, adicionando o índice do local à lista de carregamentos. O método atualiza a distância total percorrida e o nível da bateria em cada etapa.

Ao final do percurso, o método retorna um objeto StructurePath encapsulando informações como a distância total percorrida, a lista de locais no percurso, os índices de carregamentos e a viabilidade do percurso. Esses dados são fundamentais para avaliar se o percurso pode ser realizado com sucesso, levando em consideração a autonomia do veículo.

USEI07 - Análise da complexidade

getPontoPartida: Este método itera sobre todos os vértices no gráfico. Portanto, a complexidade de tempo é O(V), onde V é o número de vértices no gráfico.

calculaMelhorPercurso: Este método contém um loop while que, no pior caso, pode iterar sobre todos os hubs no gráfico. Dentro desse loop, ele chama o método shortestPathWithAutonomy, que tem uma complexidade de tempo de O(V^2) ou O(V log V). Portanto, a complexidade de tempo total seria O(H * V^2) ou O(H * V log V), onde H é o número de hubs.

A complexidade de tempo do algoritmo de Dijkstra é O((V+E)logV), onde V é o número de vértices e E é o número de arestas no grafo. Isso ocorre porque, para cada vértice, podemos ter que visitar todos os seus vértices adjacentes, e essa operação é realizada usando uma fila de prioridade (estrutura de dados heap), daí o fator logV. A complexidade de espaço é O(V), porque, no pior caso, a fila de prioridade armazenará todos os vértices do grafo.

getTempoFinalCompleto(LinkedList<Localidades> caminhoPercorrido, LocalTime horaComeco, int autonomia, double averageVelocity, int tempoRecarga, int tempoDescarga, int numeroDescargas): Este método tem uma complexidade de tempo de O(V), onde V é o número de vértices no caminho percorrido. A complexidade do espaço é O(1), pois não usa estruturas de dados adicionais que crescem com o tamanho da entrada.

intToLocalTime(int tempo): Este método tem uma complexidade de tempo e espaço de O(1), pois realiza uma operação constante.

getFinishingTimeRoute(int distanciaTotal, double averageVelocity, LocalTime horaComeco): Este método tem uma complexidade de tempo e espaço de O(1), pois realiza uma operação constante.

getTimeTable(StructurePath structurePath, LocalTime horaComeco, int autonomia, double averageVelocity, int tempoRecarga, int tempoDescarga): Este método tem uma complexidade de tempo de O(V), onde V é o número de vértices no percurso. A complexidade do espaço é O(V), pois cria uma tabela de horários que tem um tamanho proporcional ao número de vértices no percurso.

getAllHubsInCourse(LinkedList<Localidades> caminho, List<Localidades> locaisVisitados): Este método tem uma complexidade de tempo de O(V), onde V é o número de vértices no caminho. A complexidade do espaço é O(V), pois cria uma lista de hubs que tem um tamanho proporcional ao número de vértices no caminho.

addTime(LocalTime time1, LocalTime time2): Este método tem uma complexidade de tempo e espaço de O(1), pois realiza uma operação constante.

minusTime(LocalTime time1, LocalTime time2): Este método tem uma complexidade de tempo e espaço de O(1), pois realiza uma operação constante.

getStillOpenHubs(LocalTime hora): Este método tem uma complexidade de tempo de O(V), onde V é o número de vértices no gráfico. Isso ocorre porque ele itera sobre todos os vértices do gráfico. A complexidade do espaço é O(V), pois cria uma lista de hubs que tem um tamanho proporcional ao número de vértices no gráfico.

getHubs(): Este método tem uma complexidade de tempo de O(V), onde V é o número de vértices no gráfico. Isso ocorre porque ele itera sobre todos os vértices do gráfico. A complexidade do

espaço é O(V), pois cria uma lista de hubs que tem um tamanho proporcional ao número de vértices no gráfico.

analyzeData(int autonomia, LinkedList<Localidades> caminho): Este método tem uma complexidade de tempo de O(V), onde V é o número de vértices no caminho. A complexidade do espaço é O(V), pois cria uma lista de índices de carregamentos que tem um tamanho proporcional ao número de vértices no caminho.

Para a classe como um todo, a complexidade de tempo é dominada pelo método calculaMelhorPercurso, que tem uma complexidade de tempo de O(H * V^2) ou O(H * V log V).

USEI08

A classe US_EI08_FindDeliveryCircuit é uma implementação que visa encontrar um circuito de entrega otimizado, considerando várias restrições e parâmetros específicos, sendo eles os N hubs que obrigatoriamente o circuito tem que conter e o facto de não poder repetir nenhum vértice, salvo a exceção no caminho de volta fornecendo informações detalhadas sobre o percurso, tempos e distâncias.

extractNumber()

```
private int extractNumber(String s) {
    StringBuilder number = new StringBuilder();

    for (int i = s.length() - 1; i >= 0; i--) {
        char ch = s.charAt(i);

        if (Character.isDigit(ch)) {
            number.insert( offset: 0, ch);
        } else {
            break;
        }
    }

    return number.length() > 0 ? Integer.parseInt(number.toString()) : 0;
}
```

Figura 19-Método extractNumber()

O método `extractNumber` é responsável por extrair um número inteiro de uma sequência de caracteres. A implementação utiliza um `StringBuilder` para construir dinamicamente a representação do número enquanto percorre a string de trás para frente. Durante esse processo, o método verifica cada caractere da string, inserindo os dígitos no início do `StringBuilder` sempre que encontra um caractere numérico.

O loop é encerrado assim que um caractere não numérico é encontrado, pois a função está projetada para extrair apenas números consecutivos no final da string. Ao finalizar o loop, o método verifica se há algum dígito extraído no `StringBuilder`. Se houver, converte a representação do número para um inteiro usando `Integer.parseInt(number.toString())`. Caso contrário, retorna 0.

getTopHubs()

```java
public List<Localidades> getTopHubs(MapGraph<Localidades, Integer> distancesGraph, int nHubs) {

    List<Localidades> hubs = new ArrayList<>();
    List<Localidades> topNHubs = new ArrayList<>();
    int qtd = 0;

    for (Localidades localidade : distancesGraph.vertices()) {

        if (localidade.isHub()) {
            hubs.add(localidade);
        }

    }

    hubs.sort(Comparator.comparingInt(o -> -extractNumber(o.getNumId())));

    for (Localidades localidades : hubs) {
        if (qtd != nHubs) {
            topNHubs.add(localidades);
            qtd++;
        }

        if (qtd == 5) {
            break;
        }
    }


    return topNHubs;
}
```

Figura 20-getTopHubsMétodo

O método `getTopHubs` é responsável por identificar e retornar os principais centros de distribuição (hubs) com base em um número especificado (`nHubs`). Inicialmente, duas listas são criadas: `hubs` e `topNHubs`.

A primeira é destinada a armazenar todas as localidades que são hubs, enquanto a segunda será utilizada para armazenar os principais hubs identificados. O método itera sobre todas as localidades presentes no grafo `distancesGraph` e para cada localidade, verifica-se se é um hub utilizando o método `isHub()`.

Se for, a localidade é adicionada à lista `hubs`. A lista `hubs` é então ordenada com base em um critério de comparação personalizado esta ordenação é realizada pelo número extraído do identificador (`getNumId()`) de cada hub, utilizando o método `extractNumber` explicado anteriormente.

Em seguida, o método percorre os hubs ordenados e adiciona os primeiros `nHubs` à lista `topNHubs`,o loop que adiciona hubs é interrompido quando `qtd` atinge o valor especificado por `nHubs` isso garante que apenas a quantidade desejada de hubs seja considerada. Finalmente, a lista `topNHubs`, contendo os hubs mais relevantes, é retornada pelo método.

calcularCaminhoMaisCurto()

```java
public LinkedList<Localidades> calcularCaminhoMaisCurto(MapGraph<Localidades, Integer> distancesGraph,Localidades localOrigemVolta, Localidades localOrigem) {

    LinkedList<Localidades> shortestPath = new LinkedList<>();

    Algorithms.shortestPath(distancesGraph, localOrigemVolta, localOrigem, Comparator.naturalOrder(), Integer::sum, zero: 0, shortestPath);

    return shortestPath;
}
```

Figura 21-calcularCaminhoMaisCurto método

O método `calcularCaminhoMaisCurto` é responsável por calcular o caminho mais curto entre duas localidades em um grafo ponderado. O método inicia criando uma lista encadeada (`shortestPath`) para armazenar o caminho mais curto entre duas localidades.

Utiliza o método `Algorithms.shortestPath`, que implementa um algoritmo de caminho mais curto sobre o grafo (`distancesGraph`). Este algoritmo é então projetado para encontrar o caminho mais curto entre dois vértices específicos (`localOrigemVolta` e `localOrigem`), levando em consideração os pesos das arestas do grafo fornecendo al algoritmo o`distancesGraph` que representa o grafo com as distâncias entre as localidades, a localidade de origem para o caminho de volta, a localidade de origem para o caminho de ida ,a lista encadeada que será preenchida com o caminho mais curto entre outros.

Após a execução do algoritmo, a lista encadeada `shortestPath` contém o caminho mais curto entre as localidades de origem e destino.

calculateLoadingHubs ()

```
public List<Localidades> calculateLoadingHubs(List<Localidades> shortestPath, MapGraph<Localidades, Integer> distancesGraph, Integer autonomia, List<Integer> distance
    List<Localidades> loadingLocalidades = new ArrayList<>();
    int currentAutonomy = autonomia;
    int tempoDescargaVoltaVar = 0;
    int totalDistanceVar = 0;

    for (int i = 0; i < shortestPath.size() - 1; i++) {
        Localidades vertexA = shortestPath.get(i);
        Localidades vertexB = shortestPath.get(i + 1);
        int distance = distancesGraph.edge(vertexA, vertexB).getWeight();
        distanceBeetweenLocals.add(distance);
        totalDistanceVar += distance;

        if (vertexB.isHub()){
            tempoDescargaVoltaVar += tempoDescarga;
        }

        int distanceAux = 0;
        Edge<Localidades, Integer> edge = distancesGraph.edge(vertexA, vertexB);
        if (edge != null) {
            distanceAux = edge.getWeight();
        }

        // Verifica se a autonomia é suficiente para a distância entre os vértices
        if (distanceAux > currentAutonomy) {
            // Se não for, é necessário um carregamento
            loadingLocalidades.add(vertexA);
            // Resetamos a autonomia para a autonomia máxima após o carregamento
            currentAutonomy = autonomia;
        }

        // Reduzimos a autonomia com a distância percorrida
        currentAutonomy -= distanceAux;
    }

    tempoDescargaVolta.set(0,tempoDescargaVoltaVar);
    totalDistance.set(0,totalDistance.get(0) + totalDistanceVar);


    return loadingLocalidades;
```


Figura 22-calculateLoadingHubs Método

O método `calculateLoadingHubs` é responsável por calcular os hubs de carregamento ao longo do caminho mais curto identificado. O método inicia criando variáveis necessárias para rastrear a autonomia atual do veículo (`currentAutonomy`), o tempo total de descarga durante o percurso de volta (`tempoDescargaVoltaVar`), e a distância total percorrida (`totalDistanceVar`).

O método itera sobre o caminho mais curto (`shortestPath`) entre as localidades, considerando a ordem das conexões no grafo.Para cada aresta no caminho, são calculados a distância entre as localidades e o tempo de descarga associado que serão adicionadas às listas `distanceBeetweenLocals` e `totalDistanceVar`, respectivamente.

Se a localidade de destino (`vertexB`) for um hub, o tempo de descarga (`tempoDescargaVoltaVar`) é acumulado, refletindo o tempo necessário para descarregar no hub durante o percurso de volta. A autonomia do veículo é verificada em relação à distância entre os vértices (`distanceAux`), se a autonomia não for suficiente, é considerado necessário um carregamento, e a localidade de origem (`vertexA`) é adicionada à lista `loadingLocalidades`.

A autonomia é resetada para o valor máximo após cada carregamento e de seguida é reduzida com base na distância percorrida entre os vértices (`distanceAux`). As variáveis `tempoDescargaVolta` e `totalDistance` são atualizadas com os valores acumulados durante a iteração.

Finalmente a lista `loadingLocalidades` contendo as localidades onde é necessário realizar carregamento é retornada pelo método.

findDeliveryCircuitMethod()

```
public Map<String, Object> findDeliveryCircuit(MapGraph<Localidades, Integer> distancesGraph, int nHubs, int velMedia, int autonomia, Localidades localOrigem, int te

    List<Localidades> topNHubs = getTopHubs(distancesGraph, nHubs);

    List<Localidades> loadingLocations = new ArrayList<>();

    List<Integer> distanceBeetweenLocals = new ArrayList<>();

    List<Integer> totalDistance = new ArrayList<>();

    List<Integer> tempoTotalDescargaIda = new ArrayList<>();
    tempoTotalDescargaIda.add(tempoDeDescarga);

    LinkedList<Localidades> path = Algorithms.nearestNeighbor(distancesGraph, localOrigem, topNHubs, Comparator.naturalOrder(), autonomia, loadingLocations, distanceB

    Localidades localOrigemVolta = path.getLast();
    path.removeLast();

    LinkedList<Localidades> shortestPath = calcularCaminhoMaisCurto(distancesGraph,localOrigemVolta,localOrigem);;

    path.addAll(shortestPath);

    List<Integer> tempoTotalDescargaVolta = new ArrayList<>();
    tempoTotalDescargaVolta.add(tempoDeDescarga);

    loadingLocations.addAll(calculateLoadingHubs(shortestPath, distancesGraph, autonomia, distanceBeetweenLocals, totalDistance,tempoDeDescarga,tempoTotalDescargaVolt

    int tempoDescargaTotal = tempoTotalDescargaVolta.get(0) + tempoTotalDescargaIda.get(0);

    int tempoPercurso =  totalDistance.get(0) / velMedia;

    int tempoTotalCarregamento = loadingLocations.size() * tempoDeCarregamento;

    int tempoTotal = tempoPercurso + tempoTotalCarregamento + tempoDescargaTotal;

    loadingLocations.remove( index: loadingLocations.size() - 1);
```

Figura 23-findDeliveryCircuit Metodo Principal

```
Map<String, Object> result = new HashMap<>();

result.put("localOrigem", localOrigem);
result.put("locaisPassagem", path);
result.put("distanciaEntreLocais", distanceBeetweenLocals);
result.put("distanciaTotal", totalDistance.get(0));
result.put("numeroCarregamentos", loadingLocations.size());
result.put("tempoPercurso", tempoPercurso);
result.put("loadingLocations", loadingLocations);
result.put("tempoTotalDescarga", tempoDescargaTotal);
result.put("tempoTotalCarregamento", tempoTotalCarregamento);
result.put("tempoTotal", tempoTotal);
return result;
```

Figura 24-Resultado Método findDeliveryCircuit

Este método é o núcleo da funcionalidade e desempenha um papel fundamental na otimização do circuito de entrega.

O método `findDeliveryCircuit` representa a essência do sistema de otimização de circuitos de entrega, integrando diversos algoritmos e lógicas para determinar a rota mais eficiente para entrega de mercadorias. Inicia-se pela obtenção dos principais hubs a serem considerados no circuito de entrega, utilizando o método `getTopHubs`.

De seguida são inicializadas listas e estruturas necessárias para armazenar informações relevantes, como locais de carregamento (`loadingLocations`), distâncias entre localidades (`distanceBetweenLocals`), distância total (`totalDistance`), e tempos de descarga. Utiliza o algoritmo `nearestNeighbor` da classe `Algorithms` para determinar um caminho aproximadamente ótimo, começando na `localOrigem`, visitando os hubs principais e retornando à origem, depois calcula o caminho mais curto entre a última localidade do caminho (`localOrigemVolta`) e a localidade de origem (`localOrigem`), utilizando o método `calcularCaminhoMaisCurto`.

Após isso, combina os caminhos obtidos, removendo o último ponto do caminho de ida e anexando o caminho mais curto de volta, utiliza o método `calculateLoadingHubs` para identificar os hubs onde é necessário carregar durante o percurso e calcula os tempos totais de descarga, percurso e carregamento, bem como a distância total percorrida.

Seguidamente, prepara um mapa (`result`) contendo informações detalhadas sobre o circuito otimizado, incluindo locais de origem, locais de passagem, distâncias, tempos, número de carregamentos, entre outros. Finalmente, retorna o mapa contendo todas as informações relevantes sobre o circuito de entrega otimizado.

Algoritmos utilizados - USEI08

nearestNeighbor()

```
public static <V, E> LinkedList<V> nearestNeighbor(Graph<V, E> g, V start, List<V> targetVertices,
                                                   Comparator<E> ce, int autonomia , List<V> locaisDeCarregamento,List<E> distanceBeetweenLocals,List<Integer> totalDistance,List<Integer> tempoTotalDescarga) {
    LinkedList<V> path = new LinkedList<>();
    Set<V> visited = new HashSet<>();
    Set<V> visitedHubs = new HashSet<>();
    path.add(start);
    visited.add(start);

    int currentAutonomy = autonomia;
    int total = 0;


    while (!targetVertices.isEmpty()) {
        V currentVertex = path.getLast();
        E minWeight = null;
        V nextVertex = null;

        for (V neighbor : g.adjVertices(currentVertex)) {
            if ((!visited.contains(neighbor) && targetVertices.contains(neighbor)) && !visitedHubs.contains(neighbor)) {
                Edge<V, E> edge = g.edge(currentVertex, neighbor);
                if (edge != null) {
                    if (minWeight == null || ce.compare(edge.getWeight(), minWeight) <= 0) {
                        if (minWeight == null || ce.compare(edge.getWeight(), minWeight) < 0) {
                            minWeight = edge.getWeight();
                            nextVertex = neighbor;
                        }
                    }
                }
            }
        }

        if (nextVertex != null) {


            int distance = (Integer) minWeight;

            // Verifica se a autonomia é suficiente para a distância entre os vértices
            if (distance > currentAutonomy) {
                // Se não for, é necessário um carregamento
                locaisDeCarregamento.add(currentVertex);
                // Resetamos a autonomia para a autonomia máxima após o carregamento
                currentAutonomy = autonomia;
            }

            // Reduzimos a autonomia com a distância percorrida
            currentAutonomy -= distance;

            path.add(nextVertex);
            // Remove o vértice da lista de destinos apenas se não for um hub ou se algum de seus vizinhos ainda não foi visitado
            if (!targetVertices.contains(nextVertex) || !allHubNeighborsVisited(g, nextVertex, visited, targetVertices)) {
                visited.add(nextVertex);
                targetVertices.remove(nextVertex);
            }else {
```


Figura 25-nearestNeighbor Método

```
if (!targetVertices.contains(nextVertex) || !allHubNeighborsVisited(g, nextVertex, visited, targetVertices)) {
    visited.add(nextVertex);
    targetVertices.remove(nextVertex);
}else {
    visitedHubs.add(nextVertex);
}

else {
  // Escolhe o vizinho mais próximo, mesmo que não esteja na lista de destinos
  minWeight = null;
  for (V neighbor : g.adjVertices(currentVertex)) {
      if (!visited.contains(neighbor)) {
          Edge<V, E> edge = g.edge(currentVertex, neighbor);
          if (edge != null) {
              if (minWeight == null || ce.compare(edge.getWeight(), minWeight) < 0) {
                  minWeight = edge.getWeight();
                  nextVertex = neighbor;
              }
          }
      }
  }

  if (nextVertex != null) {

      int distance = (Integer) minWeight;

      // Verifica se a autonomia é suficiente para a distância entre os vértices
      if (distance > currentAutonomy) {
          // Se não for, é necessário um carregamento
          locaisDeCarregamento.add(currentVertex);
          // Resetamos a autonomia para a autonomia máxima após o carregamento
          currentAutonomy = autonomia;
      }

      // Reduzimos a autonomia com a distância percorrida
      currentAutonomy -= distance;

      path.add(nextVertex);
      visited.add(nextVertex);
      visitedHubs.clear();
  } else if (!visited.isEmpty()) {
      // Revisit the last vertex if there are no unvisited neighbors
      V lastVertex = path.size() >= 2 ? path.get(path.size() - 2) : null;
      if (lastVertex != null ) {
          // Revisit the last vertex

          visited.add(lastVertex);
          path.removeLast(); // Remove the last vertex
          currentAutonomy = autonomia; // Reset autonomy after revisiting
          continue; // Continue to the next iteration
      } else {
          // No more vertices to revisit, break the loop
          break;
      }
```

Figura 26-nearestNeighbor Método(continuação)

```
                }
            } else {
                // No more neighbors or unvisited vertices, break the loop
                break;
            }
        }
    }
    int totalTempoDescargaVar = tempoTotalDescarga.get(0);
    // Calculate distances only between consecutive locations in the path
    for (int i = 0; i < path.size() - 1; i++) {
        V currentLocation = path.get(i);
        V nextLocation = path.get(i + 1);

        Edge<V, E> edge = g.edge(currentLocation, nextLocation);
        if (edge != null) {
            distanceBeetweenLocals.add((E) edge.getWeight());
            total += (Integer) edge.getWeight();
            Localidades nextLocation1 = (Localidades) nextLocation;

            if ( nextLocation1.isHub()){
                totalTempoDescargaVar += tempoTotalDescarga.get(0);
            }

        }
    }

    totalDistance.add(total);
    tempoTotalDescarga.set(0,totalTempoDescargaVar);

    return path;
```

Figura 27-nearestNeighbor Método(continuação)

Este método, chamado nearestNeighbor, implementa o algoritmo do Vizinho Mais Próximo para encontrar um caminho aproximadamente otimizado em um grafo ponderado. Este método, chamado `nearestNeighbor`, implementa o algoritmo do Vizinho Mais Próximo para encontrar um caminho aproximadamente otimizado em um grafo ponderado. Aqui está um resumo das principais funcionalidades:

Enquanto houver destinos a visitar, o método continua a escolher o próximo vértice com base no vizinho mais próximo e vai iterando sobre os vizinhos do vértice atual e escolhe o vizinho mais próximo que ainda não foi visitado e é um destino válido. Verifica se a autonomia do veículo é suficiente para a distância até o próximo vértice. Se não for, adiciona o local atual à lista de locais de carregamento e reinicia a autonomia e depois então atualiza conjuntos e listas conforme necessário, removendo o vértice atual dos destinos se necessário.

Se não houver vizinhos válidos para visitar, o método escolhe o vizinho mais próximo, mesmo que não esteja na lista de destinos. Se não houver vizinhos não visitados, o método pode revisitar o último vértice, reiniciando a autonomia.

Por fim, calcula distâncias entre locais consecutivos no caminho, atualiza a distância total, e considera tempos de descarga adicionais para hubs e retorna o caminho percorrido, informações sobre distâncias, e atualiza a lista de locais de carregamento e o tempo total de descarga.

shortestPath ()

```java
/**
 * Shortest-path between two vertices
 *
 * @param <V>       the type parameter
 * @param <E>       the type parameter
 * @param g         graph
 * @param vOrig     origin vertex
 * @param vDest     destination vertex
 * @param ce        comparator between elements of type E
 * @param sum       sum two elements of type E
 * @param zero      neutral element of the sum in elements of type E
 * @param shortPath returns the vertices which make the shortest path
 * @return if vertices exist in the graph and are connected, true, false otherwise
 */
3 usages  35193 +1
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];
    V[] pathKeys = (V[]) new Object [numVerts];
    E[] dist = (E[]) new Object [numVerts];
    initializePathDist(numVerts, pathKeys, dist);

    shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, pathKeys, dist);

    E lengthPath = dist[g.key(vDest)];

    if(lengthPath != null){
        getPath(g, vOrig, vDest, pathKeys, shortPath);
        return lengthPath;
    }

    return null;
}
```

Figura 28-shortestPath Método

Este método, chamado `shortestPath`, utiliza o algoritmo de Dijkstra para encontrar o caminho mais curto em um grafo ponderado, representado por um objeto `Graph`.

Inicializa arrays e variáveis necessárias para rastrear os vértices visitados, chaves de caminho e distâncias e de seguida, chama o método `shortestPathDijkstra`, que implementa o algoritmo de Dijkstra para encontrar os caminhos mais curtos, obtém a distância do caminho mais curto entre os vértices de origem e destino. Se o caminho existe, recupera o caminho real e retorna a distância. Se o caminho não existir (distância é `null`), retorna `null`.

USEI08 - Análise da complexidade

extractNumber()

A complexidade temporal total deste algoritmo é dominada pela construção da StringBuilder no loop. Dado que a inserção em uma posição específica de uma StringBuilder tem complexidade O(m) e a operação é realizada em cada iteração do loop, a complexidade temporal total é O(n^2), onde n é o comprimento da string de entrada.

getTopHubs()

A complexidade temporal do loop inicial é O(V). Dentro deste loop, há uma operação de verificação de um atributo booleano (`localidade.isHub()`). Essa verificação é realizada em O(1) para cada vértice. O segundo loop `for` itera sobre os vértices marcados como "hubs" e realiza operações de ordenação e seleção.

`hubs.sort(Comparator.comparingInt(o -> -extractNumber(o.getNumId())));`: A complexidade de ordenação é geralmente O(N log N), onde N é o número de elementos a serem ordenados. A complexidade de extrair o número é O(K), onde K é o comprimento da string. Portanto, a complexidade total desta operação de ordenação é O(V K log V), onde K é o comprimento máximo dos identificadores dos hubs e o segundo loop `for` tem uma complexidade O(V) porque itera sobre os "hubs".

Portanto, a complexidade temporal total deste algoritmo é dominada pela operação de ordenação, resultando em O(V * K * log V), onde V é o número de vértices e K é o comprimento máximo dos identificadores dos "hubs".

calcularCaminhoMaisCurto()

Este método calcularCaminhoMaisCurto depende do shortestPath. 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)).**

calculateLoadingHubs ()

O loop `for` percorre a lista `shortestPath`. A complexidade deste loop é O(n), onde n é o tamanho da lista `shortestPath`, sendo que as operações dentro do loop é O(1) e a adição de elementos à lista `distanceBetweenLocals`,`loadingLocalidades`, e modificação dos elementos em `tempoDescargaVolta` e `totalDistance`: O(1) para cada operação, pois são operações de lista.

No geral, a complexidade temporal é dominada pelo loop principal. Portanto, a complexidade total do método `calculateLoadingHubs` é O(n), onde n é o tamanho da lista `shortestPath`. As operações dentro do loop, como verificações de hub e acesso a arestas, são realizadas em tempo constante e não alteram a complexidade global do algoritmo.

findDeliveryCircuitMethod()

Como mencionado anteriormente, a complexidade temporal deste método é dominada pela operação de ordenação dos hubs. A ordenação ocorre em O(V * K * log V), onde (V) é o número de vértices e K é o comprimento máximo dos identificadores dos hubs. A criação de listas, como `loadingLocations`, `distanceBetweenLocals`, `totalDistance`, `tempoTotalDescargaIda`, e `tempoTotalDescargaVolta`, é O(1) pois são operações de tempo constante.

A complexidade temporal é O(V^2), onde V é o número de vértices. A complexidade deste método também foi discutida anteriormente e é O((V + E) log V), onde V é o número de vértices e E é o número de arestas no grafo. A complexidade deste método é O(n), onde n é o tamanho da lista `shortestPath`. As operações finais, como cálculos de tempo e manipulação de listas, são O(1) porque envolvem operações de tempo constante.

A complexidade total do método `findDeliveryCircuit` é dominada principalmente pelas chamadas aos métodos `getTopHubs`, `nearestNeighbor`, e `calcularCaminhoMaisCurto`. A complexidade temporal total do método depende principalmente do algoritmo específico usado em `nearestNeighbor`. Se a complexidade do algoritmo de vizinho mais próximo for otimizada para O(V log V) ou algo semelhante, então a complexidade global seria O(V^2).

USEI09

Esta classe organiza vértices de um grafo em clusters baseado na proximidade aos hubs. Usa dois algoritmos: Floyd-Warshall e Dijkstra. O método **getClustersFW()** usa Floyd-Warshall, removendo a aresta com mais caminhos mais curtos até isolar os clusters. O método **getClustersD()** usa Dijkstra, atribuindo cada localidade ao hub com o caminho mais curto. Ambos retornam um mapa com hubs como chaves e listas de localidades como valores.

getHubs()

```
public List<Localidades> getHubs() {
    List<Localidades> result = new ArrayList<>();
    for (Localidades local : graphMod.vertices()) {
        if (local.isHub()) {
            result.add(local);
        }
    }
    return result;
}
```

Este método retorna uma lista de Localidades que são hubs. Itera sobre todos os vértices no grafo e adiciona aqueles que são hubs à lista de resultados.

getClustersFW()

```
public Map<Localidades, List<Localidades>> getClusters() {
    Map<Localidades, List<Localidades>> clusters = new HashMap<>();
    List<Localidades> hubs = getHubs();
    MapGraph<Localidades, Integer> graphCopy = new MapGraph<>(graphMod);

    // Initialize clusters with empty lists for each hub
    for (Localidades hub : hubs) {
        clusters.put(hub, new ArrayList<>());
    }

    // Calculate the number of shortest paths for each edge
    Map<Edge<Localidades, Integer>, Integer> edgePathCounts = calculateEdgePathCounts(graphCopy);

    // Sort the edges in descending order based on the number of shortest paths
    List<Edge<Localidades, Integer>> sortedEdges = new ArrayList<>(edgePathCounts.keySet());
    sortedEdges.sort(Comparator.comparing(edgePathCounts::get).reversed());

    // Iteratively remove the edge with the highest number of shortest paths
    for (Edge<Localidades, Integer> edge : sortedEdges) {
        graphCopy.removeEdge(edge.getVOrig(), edge.getVDest());

        // Check if the clusters are isolated
        if (areClustersIsolated(graphCopy, clusters)) {
            break;
        }
    }
```

```
        for (Localidades localidade : graphCopy.vertices()) {
            if (!localidade.isHub()) {
                for (Localidades hub : hubs) {
                    if (graphCopy.getEdge(hub, localidade) != null) {
                        clusters.get(hub).add(localidade);
                        break;
                    }
                }
            }
        }

        return clusters;
```

Este método cria clusters de vértices com base na sua proximidade aos hubs. Inicialmente, cria-se um mapa onde as chaves são os hubs e os valores são listas vazias que irão armazenar os vértices que pertencem ao cluster desse hub. Em seguida, calcula-se o número de caminhos mais curtos para cada aresta do grafo e ordena-se as arestas em ordem decrescente com base nesse número. A aresta com o maior número de caminhos mais curtos é removida iterativamente até que cada cluster esteja isolado. Finalmente, cada Localidades é atribuída ao hub ao qual está atualmente conectada. Este método não recebe nenhum argumento e retorna um mapa onde as chaves são hubs e os valores são listas de Localidades que pertencem ao cluster desse hub.

calculateEdgePathCounts()

```
private Map<Edge<Localidades, Integer>, Integer> calculateEdgePathCounts(MapGraph<Localidades, Integer> graph) {
    int n = graph.numVertices();
    int[][] pathCount = new int[n][n];

    // Initialize pathCount
    for (int i = 0; i < n; i++) {
        pathCount[i][i] = 1;
        for (Edge<Localidades, Integer> edge : graph.outgoingEdges(graph.vertices().get(i))) {
            int j = graph.vertices().indexOf(edge.getVDest());
            pathCount[i][j] = 1;
        }
    }

    // Calculate the number of shortest paths
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                pathCount[i][j] += pathCount[i][k] * pathCount[k][j];
            }
        }
    }

    // Count the number of shortest paths for each edge
    Map<Edge<Localidades, Integer>, Integer> edgePathCounts = new HashMap<>();
    for (Edge<Localidades, Integer> edge : graph.edges()) {
        int u = graph.vertices().indexOf(edge.getVOrig());
        int v = graph.vertices().indexOf(edge.getVDest());
        edgePathCounts.put(edge, pathCount[u][v]);
    }

    return edgePathCounts;
}
```

Este método calcula o número de caminhos mais curtos que passam por cada aresta no grafo. Utiliza uma variante do algoritmo **Floyd-Warshall** para calcular o número de caminhos mais curtos entre cada par de vértices. O resultado é um mapa onde as chaves são as arestas e os valores são o número de caminhos mais curtos que passam por cada aresta. Este método recebe um **MapGraph<Localidades,**

Integer> como argumento e retorna um mapa onde as chaves são as arestas e os valores são o número de caminhos mais curtos que passam por cada aresta.

areClustersIsolated()

```
private boolean areClustersIsolated(MapGraph<Localidades, Integer> graph, Map<Localidades, List<Localidades>> clusters)
    for (Edge<Localidades, Integer> edge : graph.edges()) {
        Localidades u = edge.getVOrig();
        Localidades v = edge.getVDest();

        // Find the clusters of u and v
        List<Localidades> clusterU = null;
        List<Localidades> clusterV = null;
        for (Map.Entry<Localidades, List<Localidades>> entry : clusters.entrySet()) {
            if (entry.getValue().contains(u)) {
                clusterU = entry.getValue();
            }
            if (entry.getValue().contains(v)) {
                clusterV = entry.getValue();
            }
        }

        // If u and v belong to different clusters and there is an edge between them, return false
        if (clusterU != clusterV && graph.getEdge(u, v) != null) {
            return false;
        }
    }

    // If no such pair of vertices is found, return true
    return true;
}
```

Este método verifica se os clusters estão isolados. Itera sobre todas as arestas no grafo e verifica se existe uma aresta entre vértices que pertencem a diferentes clusters. Se tal aresta for encontrada, retorna falso. Se não for encontrada tal aresta, retorna verdadeiro. Isso garante que os clusters estão efetivamente isolados uns dos outros no grafo. Este método recebe um **MapGraph<Localidades, Integer>** e um mapa de clusters como argumentos e retorna um booleano indicando se os clusters estão isolados ou não.

getClustersD()

```
public Map<Localidades, List<Localidades>> getClusters() {
    Map<Localidades, List<Localidades>> clusters = new HashMap<>();
    List<Localidades> hubs = getHubs();
    MapGraph<Localidades, Integer> graphCopy = new MapGraph<>(graphMod);

    // Initialize clusters with empty lists
    for (Localidades hub : hubs) {
        clusters.put(hub, new ArrayList<>());
    }

    // Assign each localidade to the closest hub
    for (Localidades localidade : graphCopy.vertices()) {
        if (!localidade.isHub()) {
            Localidades closestHub = null;
            int shortestDistance = Integer.MAX_VALUE;

            for (Localidades hub : hubs) {
                LinkedList<Localidades> path = new LinkedList<>();
                int distance = Algorithms.shortestPath(graphCopy, hub, localidade, Comparator.naturalOrder(), Integer::sum,  zero: 0, path);
                if (distance < shortestDistance) {
                    closestHub = hub;
                    shortestDistance = distance;
                }
            }

            clusters.get(closestHub).add(localidade);
        }
    }

    return clusters;
}
```

Este método usa o algoritmo de Dijkstra para organizar os vértices (localidades) de um grafo em clusters com base em sua proximidade com os hubs. Ele encontra o caminho mais curto de cada localidade para todos os hubs e atribui cada localidade ao hub que fornece o caminho mais curto. O resultado é um mapa onde cada chave é um hub e cada valor é uma lista de localidades que pertencem ao cluster desse hub.

USEI09 - Análise da complexidade

getHubs(): Este método percorre todos os vértices do grafo uma vez, portanto, a complexidade de tempo é O(V), onde V é o número de vértices no grafo.

getClustersFW(): Este método usa o algoritmo de Floyd-Warshall, que tem uma complexidade de tempo de O(V^3), onde V é o número de vértices no grafo. Além disso, ele percorre todas as arestas do grafo, adicionando uma complexidade de tempo de O(E), onde E é o número de arestas no grafo. Portanto, a complexidade de tempo total é O(V^3 + E).

calculateEdgePathCounts(): Este método também usa o algoritmo de Floyd-Warshall, portanto, a complexidade de tempo é O(V^3), onde V é o número de vértices no grafo.

areClustersIsolated(): Este método percorre todas as arestas do grafo uma vez, portanto, a complexidade de tempo é O(E), onde E é o número de arestas no grafo.

getClustersD(): Este método usa o algoritmo de Dijkstra, que tem uma complexidade de tempo de O(V^2), onde V é o número de vértices no grafo. Além disso, ele percorre todos os vértices do grafo, adicionando uma complexidade de tempo de O(V). Portanto, a complexidade de tempo total é O(V^2 + V).

A complexidade da classe como um todo depende de como seus métodos são usados. Se todos os métodos forem usados uma vez, a complexidade de tempo será a soma das complexidades dos métodos, que é O(V^3 + E) para getClustersFW(), O(V^3) para calculateEdgePathCounts(), O(E) para areClustersIsolated(), e O(V^2 + V) para getClustersD(). Portanto, a complexidade total seria O(V^3 + E + V^2 + V).

Conclusão

Ao finalizar a implementação do sistema de distribuição de cabazes pela GFH em Java, utilizando estruturas de grafos, é evidente a complexidade adicional enfrentada em comparação com trabalhos anteriores. Cada user story apresentou desafios significativos, uma vez que foi necessário levar em consideração uma variedade de parâmetros intrínsecos ao contexto da distribuição de produtos agrícolas.

A gestão eficiente da autonomia limitada dos veículos elétricos, o horário de funcionamento dos hubs, o tempo de descarga dos cabazes, a velocidade média dos veículos, os tempos de carregamento e a distância total percorrida são fatores interdependentes que exigiram uma abordagem minuciosa em cada US. Essa complexidade acrescida tornou o processo de resolução mais desafiador, demandando uma análise detalhada de cada elemento para garantir a precisão e a funcionalidade do sistema.

A pesquisa nos grafos, embora ofereça uma precisão valiosa na representação e manipulação da rede de distribuição, também introduziu uma dificuldade adicional. Manter um nível de complexidade baixo tornou-se um desafio, pois era crucial equilibrar a atenção aos detalhes necessários para a resolução eficaz das US com a necessidade de evitar uma sobrecarga desnecessária no sistema.

Em última análise, a implementação bem-sucedida deste sistema destacou não apenas a importância da compreensão profunda dos conceitos de grafos, mas também a habilidade de integrar esses conceitos de maneira eficiente para lidar com uma série de requisitos específicos do domínio. A abordagem meticulosa adotada na implementação visa proporcionar uma solução robusta e adaptável à complexidade inerente ao desafio da distribuição de produtos agrícolas em uma rede interconectada.