

## **Relatório Sprint 2**

**Feito por:**

Rodrigo Pontes, 1221083

Miguel Oliveira, 1211281

Mário Ribeiro, 1221019

Rodrigo Castro, 1220636

**Professores:**

Alberto Sampaio (ACS)

05/11/2023

## Índice

1. Introdução .....	1
2. Diagrama de Classes .....	2
3. Algoritmos Implementados.....	3
3.1. USEI01 .....	3
3.2. USEI02 .....	5
3.3. USEI03 .....	7
3.4. USEI04 .....	10
4. Melhoramentos Possíveis.....	12
5. Conclusão.....	13

## Índice de figuras

Figure 1 - Diagrama de classes .....	2
--------------------------------------	---

# 1. Introdução

A empresa GFH atua como operador logístico, realizando a distribuição de cabazes contendo produtos agrícolas em uma rede. No sistema, os produtores são responsáveis por produzir e vender produtos agrícolas em cabazes, podendo se especializar em frutas, hortaliças ou derivados da produção animal. Cada cabaz corresponde a uma encomenda que contém uma lista de produtos de um produtor específico, e os clientes são as entidades que realizam essas encomendas.

Os hubs, que podem ser instituições como universidades, hospitais, ginásios e empresas, são os locais de entrega e retirada dos cabazes pelos clientes. A GFH é responsável pela distribuição, não se envolvendo na composição dos cabazes. A procura agregada é resultado de todas as encomendas, enquanto a oferta agregada é o conjunto de cabazes disponibilizados pelos produtores.

O projeto propõe a criação de classes e testes utilizando a interface Graph para gerenciar a rede de distribuição de cabazes de produtos agrícolas. A rede é composta por vértices representativos de localidades onde hubs de distribuição podem existir. Os cabazes são transportados em veículos elétricos explorados pela GFH, com autonomia limitada, disponíveis nos hubs para a retirada pelos clientes. A distribuição é condicionada ao horário de funcionamento do hub.

Para simplificação, considera-se que os veículos têm uma autonomia máxima de  $A$  km, se deslocam a uma velocidade média de  $V$  km/h, os carregamentos ocorrem apenas nas localidades, cada carregamento demora  $T_c$  minutos, e o tempo médio de descarga dos cabazes nos hubs é de  $T_d$  minutos.

## 2. Diagrama de Classes

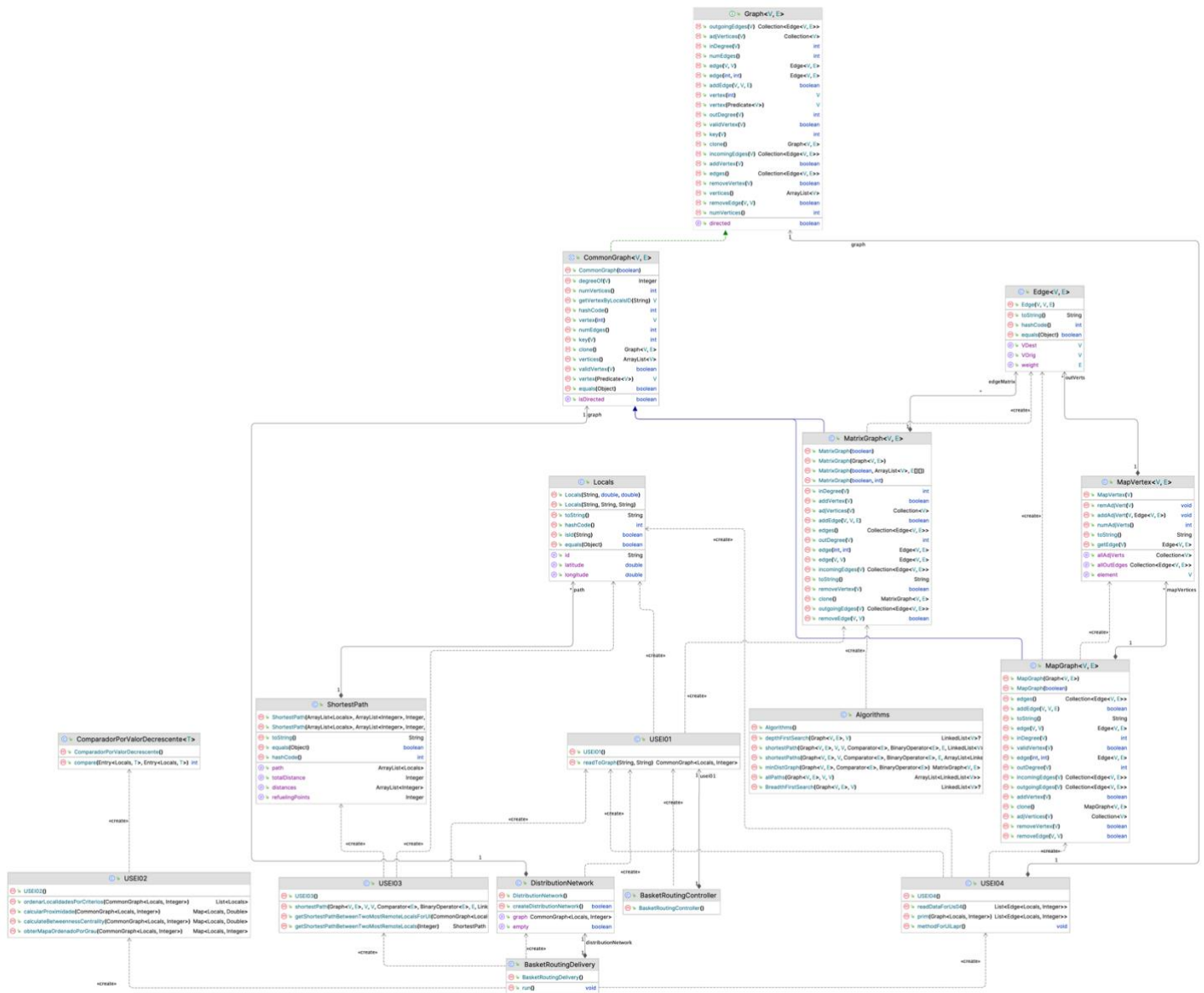


Figure 1 - Diagrama de classes

## 3. Algoritmos Implementados

### 3.1. USEI01

#### Descrição

Dada uma série de locais e as suas distâncias, o objetivo é estabelecer a rede de distribuição de cabazes entre estes pontos, usando um grafo criado a partir de ficheiros de entrada.

#### Como funciona

1. método principal recebe os ficheiros com informações sobre os locais e as distâncias entre eles. Utilizando esta informação, é construído um grafo representativo da rede de distribuição.
2. A construção do grafo é feita pela inclusão de vértices que representam os locais e arestas que representam as conexões entre eles, utilizando os dados fornecidos nos ficheiros.
3. A estrutura de dados escolhida para representar o grafo (por exemplo, uma matriz de adjacência) permite uma manipulação eficiente das informações, viabilizando operações como busca de vértices e arestas adjacentes.
4. No final do método, o grafo é devolvido para que outras operações possam ser realizadas sobre esta rede de distribuição de cabazes.

#### Análise de complexidade

O algoritmo está dividido em duas partes principais: a leitura e adição de vértices, representando os locais, e a leitura e adição de arestas, representando as distâncias entre esses locais no grafo.

A leitura e adição de vértices são realizadas no método `readVerticesGraph`. Este método percorre o ficheiro de locais, onde cada linha contém informações sobre um local. Para cada local encontrado no ficheiro, um vértice correspondente é criado e adicionado ao grafo, desde que esse vértice ainda não exista. Assim, a complexidade dessa operação é linear em relação ao número de locais no ficheiro, resultando em uma complexidade de  $O(V)$ , onde  $V$  é o número de locais.

```
while ((linha = br.readLine()) != null) {
    String[] partes = linha.split(",");
    Locals local = new Locals(partes[0],partes[1],partes[2]);

    if (!rede.validVertex(local)) {
        rede.addVertex(local);
    }
}
```

Por outro lado, a leitura e adição de arestas são realizadas no método `readEdgesGraph`. Este método percorre o ficheiro que contém informações sobre as distâncias entre os locais. Cada linha deste ficheiro representa uma aresta entre dois locais, com a distância entre eles. A complexidade

dessa operação depende do número de arestas no ficheiro. Se houver  $E$  arestas, a complexidade seria  $O(E)$ .

```
while ((linha = br.readLine()) != null) {
    String[] partes = linha.split(",");

    Locals local1 = rede.getVertexByLocalsID(partes[0]);
    Locals local2 = rede.getVertexByLocalsID(partes[1]);
    Integer distance = Integer.parseInt(partes[2]);

    rede.addEdge(local1, local2, distance);
}
```

Portanto, a complexidade total do algoritmo é determinada pela soma das complexidades das duas operações: a leitura e adição de vértices e a leitura e adição de arestas. Assumindo que  $V$  é o número de locais (vértices) e  $E$  é o número de arestas (distâncias) nos ficheiros de entrada, a complexidade total do algoritmo seria aproximadamente  $O(V + E)$ . Isso considera o processo completo de construção do grafo a partir das informações fornecidas nos ficheiros, sendo linear em relação ao número de locais e ao número de distâncias entre esses locais.

## Testes

Foram realizados testes para todos os dados existentes nos ficheiros, sendo que para todas as linhas do ficheiro, foi verificada a sua existência no grafo, após a chamada da função.

Para a fiabilidade dos resultados nos testes, foram validadas as informações dos locais e das caminhos.

## 3.2. USEI02

### Descrição

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

- Influência: Vértices com maior grau;
- Proximidade: Vértices mais próximos dos vértices restantes;
- Centralidade: Vértices com maior número de caminhos mínimos que passam por eles.

### Como funciona

1. Foram criados métodos para cada um dos critérios pedidos.
2. Em cada um dos métodos era retornado um mapa com os valores dos critérios pedidos e as respectivas localidades associadas.
3. Por fim foi criado um método que iria ordenar primeiramente o mapa por de forma decrescente pelo critério da centralidade, em caso de igualdade entraria em ação o desempate por ordem decrescente pelo critério da Influência
4. As entradas do mapa resultante ordenado são armazenadas numa lista que é finalmente retornada.

### Análise de complexidade

Neste caso em específico iremos analisar a complexidade do método `optimiseLocations`, visto que neste método são utilizados os métodos que contribuem para o resultado final.

A função `optimiseLocations` realiza a otimização de localizações de um grafo. Primeiramente, calcula a centralidade e influência para cada vértice no grafo. Para o cálculo da centralidade, para cada vértice, cria uma lista de todos os vértices no gráfico e, para cada vértice na lista, calcula o caminho mais curto entre o vértice atual e todos os outros vértices. Isto resulta numa complexidade total de  $O(V^4)$ , onde V representa o número de vértices no gráfico.

Para o cálculo da influência, para cada vértice, calcula-se o grau de entrada e o grau de saída do vértice, que resulta numa complexidade total de  $O(V^2)$ .

De seguida, a função cria uma lista de critérios para cada localização, onde cada sorteio inclui informações sobre a localização, a sua influência e centralidade. Posteriormente, ordena essa lista por ordem decrescente de centralidade e influência, utilizando uma complexidade de  $O(V * \log(V))$ .

A função seleciona então os N primeiros elementos dessa lista ordenada, onde N representa o número desejado de hubs. Isto implica uma complexidade de  $O(N * \log(V))$ .

Por fim, a função cria listas separadas de centralidade e influência para os hubs selecionados, que resultam numa complexidade adicional de  $V$  operações.

A complexidade é influenciada principalmente pelos métodos `calcularCentralidade` e `calcularInfluência`, que, por sua vez, dependem da quantidade de vértices no gráfico ( $O(V^4)$  e  $O(V^2)$  respetivamente).

```
// Calcular centralidade para cada vértice
HashMap<Locals, Integer> centralityMap = calculateCentrality(graph);

// Calcular influência para cada vértice
HashMap<Locals, Integer> influenceMap = calculateInfluence(graph);
```

A operação de ordenação também contribui para a complexidade ( $O(V * \log(V))$ ).

```
// Criar lista de critérios para cada localização
for (Locals locals : graph.vertices()) {
    int influence = influenceMap.get(locals);
    int centrality = centralityMap.get(locals);
    locationCriteriaList.add(new LocationCriteria(locals, influence, centrality));
}
```

Deste modo, podemos concluir que a complexidade total da função `optimizeLocations` é dada por  $O(V^4) + O(V^2) + O(V * \log(V)) + O(N * \log(V))$ .

## Testes

Foram realizados testes ao método `optimizeLocations` USEI02, que comprovam, através dos dados importados que os critérios aplicados estão a funcionar de maneira correta e que as listas retornadas têm os tamanhos corretos.

Foi também verificado que a lista que é retornada está devidamente ordenada.



### 3.3. USEI03

#### Descrição

Dado a autonomia de um veículo elétrico e sabendo que todos os vértices do grafo são pontos de carregamento, é pedido a determinação do caminho mais curto possível entre os pontos mais afastados da rede de distribuição.

#### Como funciona

O algoritmo funciona da seguinte forma, em passos:

1. O método principal irá receber como parâmetro principal a autonomia dada. Irá obter-se o grafo já implementado na USEI01 para efetuar o restante do algoritmo.
2. A partir do grafo obtido irá ser descoberto os locais mais afastados, com base na fórmula da distância euclidiana  $\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$ , e com base na latitude e longitude dada por cada local. Irá ser guardada num array de 2 espaços, que corresponde aos dois locais mais afastados da rede.
3. O algoritmo de Dijkstra, que calcula o caminho mínimo entre dois pontos, irá ser adaptado de forma para levar em consideração a autonomia de um veículo elétrico. Na medida que o caminho ultrapasse a autonomia, irá ser encontrado um “desvio” para poupar bateria do veículo e chegar a um ponto de carregamento antes que ficasse sem energia a meio do percurso.
4. Após descobrir o caminho, irá ser exportado esse mesmo caminho para uma melhor estrutura de informação. Também serão obtidas as distâncias entre cada ponto e o número total de carregamentos (o número de pontos de carregamento onde teve que parar para efetuar o carregamento do veículo).
6. No final, o método irá guardar a informação numa classe chamada **ShortestPath**. Nessa classe irá ter instanciado o percurso feito, as distâncias entre cada pontos, a distância total do percurso e o número total de carregamentos.

#### Análise de complexidade

A complexidade do algoritmo irá ser analisada em passos:

1. O método de descoberta dos locais mais remotos irá comparar todas as distâncias euclidianas entre todos os pares de pontos do grafo.

```
for (int i = 0; i < localsList.size(); i++) { // Primeiro ciclo iterativo
    Locals local1 = localsList.get(i);
```

```

        for (int j = i + 1; j < localsList.size(); j++) { // Segundo ciclo
iterativo

```

Nesse método irá ser envolvido dois ciclos iterativos alinhados, tendo complexidade  $O(n^2)$ .

2. O cálculo do caminho mais curto usa uma adaptação do Algoritmo de Dijkstra, tendo uma complexidade  $O((V + E) * \log(V))$ .

```

for (V vertex: g.vertices()) { // Percorre todos os vértices
    int vertexKey = g.key(vertex);
    visited[vertexKey] = false;
    dist[vertexKey] = infinity;
    pathKeys[vertexKey] = null;
}

```

Aqui irá percorrer todos os vértices e preenche-os com valores, tendo complexidade  $O(V)$ , sendo V os vértices.

```

while (!priorityQueue.isEmpty()) { // Enquanto os vértices não tem adjacências
    V currentVertex = priorityQueue.poll();
    int currentVertexKey = g.key(currentVertex);

    if (!visited[currentVertexKey]) {
        visited[currentVertexKey] = true;

        for (V neighbor : g.adjVertices(currentVertex)) { // Percorre todos os
adjacentes

```

Aqui irá percorrer todos os vértices adjacentes e preencher o menor caminho entre os adjacentes, tendo complexidade  $O(E * \log(V))$ , sendo E os caminhos e V os vértices

3. Irá proceder a reconstrução do caminho e ao preenchimento das distâncias entre cada ponto.

```

while (current != null) { // Enquanto os vertices do caminho não forem nulos
    path.addFirst(current);

```

Irá percorrer todos os vértices do grafo, obtendo complexidade  $O(V)$ , sendo V os vértices.

Sendo assim, o algoritmo apresenta a complexidade máximas das demonstradas, logo, terá complexidade  $O(n^2)$ , sendo um algoritmo determinístico.

## Testes

Foram feitos testes para o algoritmo implementado, em 3 casos diferentes:

- Quando a autonomia é menor que o menor caminho mínimo do grafo:
  - Apresenta um caminho mínimo diferente do maior, por causa dos desvios feitos no caminho.
- Quando a autonomia é menor que o primeiro nó do caminho:
  - Não apresenta um caminho.
- Quando a autonomia é maior que o menor caminho mínimo do grafo:
  - Apresenta o menor caminho mínimo do grafo.

Para a fiabilidade dos resultados nos testes, foram demonstrados o local de origem, o caminho percorrido, as distâncias de cada nó, o local de destino, a distância percorrida e os pontos onde foi efetuado o carregamento.

## 3.4. USEI04

### Descrição

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

### Como funciona

O Código funciona da seguinte forma:

1. Realizamos a leitura dos dados
2. Executamos algoritmo de Prim para encontrar a árvore geradora mínima
3. Calculamos a distância total através da soma do peso das arestas da árvore geradora mínima.
4. O resultado é encapsulado na classe `MaximumSpanningTreeResult` para facilitar o acesso às informações.

Ao chamar o método `findMinimumSpanningTree()`, é obtido um objeto que contém a árvore geradora mínima e a distância total, conforme especificado nos critérios de limitação do exercício.

### Análise da complexidade

O método `findMinimumSpanningTree` inicia a leitura dos dados das localidades e distâncias a partir de arquivos CSV, utilizando o método `readToGraph` da classe `USEI01`.

Em termos de espaço, a complexidade está relacionada ao armazenamento do gráfico e às estruturas temporárias utilizadas pelo algoritmo de Prim. O espaço necessário para o gráfico é proporcional a  $O(V + E)$ , e o espaço adicional para estruturas temporárias durante a execução do algoritmo de Prim é geralmente  $O(V + E)$ .

O método `calculaTotalDistance` percorre todas as arestas da árvore geradora mínima, que resulta numa complexidade de tempo adicional de  $O(E)$ . Não há um uso significativo de espaço adicional neste método.

O método `prim` é responsável pela execução do algoritmo de Prim, que possui uma complexidade de tempo de  $O((V + E) * \log(V))$  e uma complexidade de espaço associada a  $O(V + E)$ .

```
if (!visited.contains(nextVertex)) {
    visited.add(nextVertex);
    minimumSpanningTree.add(minEdge);

    Collection<Edge<Locals, Integer>> neighbors = graph.outgoingEdges(nextVertex);
    if (neighbors != null) {
        priorityQueue.addAll(neighbors);
    }
}
```

- `visited.contains(nextVertex)` verifica se o vértice de destino já foi visitado.
- `visitado.add(nextVertex)` adiciona o vértice de destino ao conjunto de vértices visitados.
- `mínimaSpanningTree.add(minEdge)` adiciona uma aresta à árvore geradora mínima.
- `graph.outgoingEdges(nextVertex)` obtém as arestas que saem do vértice de destino.
- `PriorityQueue.addAll(neighbors)` adiciona todas as arestas ao conjunto de prioridade. A complexidade depende do número de arestas no conjunto, resultando em  $O(E * \log(E))$ .

Podemos então concluir que a complexidade de tempo total do método `prim` é dominada pela operação `prioridadeQueue.addAll(neighbors)`, que é  $O((V + E) * \log(E))$ , onde  $V$  é o número de vértices e  $E$  é o número de arestas no grafo. A complexidade de espaço está relacionada principalmente ao armazenamento de estruturas temporárias, como o conjunto visitado e a fila de prioridade `prioridadeQueue`, ambas com complexidade  $O(V + E)$ .

## Testes

Foram realizados testes que verificam se a implementação do método `readDataForUs04` na classe `USEI04` está correta ao encontrar a Árvore Geradora Mínima no grafo fornecido e se a distância total da rede calculada é a esperada para o exemplo específico.

## 4. Melhoramentos Possíveis

O desenvolvimento deste projeto correu bem e todas as funcionalidades estão a operar conforme o esperado. No entanto, reconhecemos que existem áreas que poderiam ter sido otimizadas para acelerar a conclusão do projeto.

Apesar de todas as funcionalidades estarem a funcionar conforme previsto, identificámos oportunidades para otimizar a eficiência dos algoritmos implementados e reestruturar partes do código para torná-lo mais conciso e legível. Identificamos também que, com alguns ajustes antecipados, o projeto poderia ter sido concluído num período mais curto.

O êxito alcançado não diminui a possibilidade de melhorias. Reconhecemos que, apesar de tudo estar a funcionar como planeado, sempre existe espaço para refinamentos e ajustes que poderiam ter permitido concluir o projeto de forma mais eficiente.

## 5. Conclusão

Neste projeto, o grupo implementou com sucesso uma série de algoritmos fundamentais, divididos em User Stories, para gerir eficientemente a rede de distribuição de cabazes de produtos agrícolas da empresa GFH. Através da utilização de classes que implementam a interface Graph, conseguimos criar uma estrutura robusta que representa as localidades e hubs na rede.

Os algoritmos desenvolvidos possibilitam otimizar a distribuição dos cabazes, levando em consideração as restrições de autonomia dos veículos elétricos, horários de funcionamento dos hubs e tempos de carregamento e descarga. Essas considerações são essenciais para garantir uma operação logística eficaz, maximizando a satisfação dos clientes e minimizando custos operacionais.

Trabalhar em grupo foi fundamental para o sucesso do projeto. A colaboração permitiu a troca de ideias, a resolução eficiente de desafios e a combinação de habilidades individuais para alcançar um resultado coeso. A diversidade de perspetivas enriqueceu a abordagem adotada, resultando em soluções mais abrangentes e inovadoras.

O grupo concluiu que a implementação desses algoritmos proporciona uma base sólida para a gestão eficiente da distribuição de cabazes, contribuindo para a missão da GFH como operador logístico. O projeto do SPRINT 2 não apenas atendeu aos requisitos propostos, mas também serviu como uma valiosa experiência de aprendizado e aplicação prática dos conceitos discutidos durante o desenvolvimento do sistema.