2023

RELATÓRIO DE ESINF - TRABALHO SPRINT 2

José Sá, 1220612

Mariana Correia, 1211883

Rafael Araújo, 1201804

João Pinto, 1221694

Vasco Sousa, 1221700

Índice

Objetivo 3
Algoritmo 4
Estrutura de Dados Inicial 4
Diagrama de Classes 5
Exercício 1 6
Exercício 2 10
Exercício 3 15
Exercício 4 17
Exercício 5 20
Melhoramentos Possíveis 23
Conclusão 23

Objetivo

Para este trabalho, é pedido que se desenvolva de forma mais adequada um conjunto de funcionalidades que permitam através da interface Graph gerir uma rede de distribuição de cabazes de produtos agrícolas.

A rede deve ser constituída por vários vértices que representam localidades onde poderão existir Hubs de distribuição e cada localidade está a X distancia de qualquer outra localidade. Os cabazes são transportados em veículos elétricos, com autonomia limitada (em km) e a distribuição dos produtos é condicionada ao horário de funcionamento do hub.

Algoritmo

Estrutura de Dados Inicial

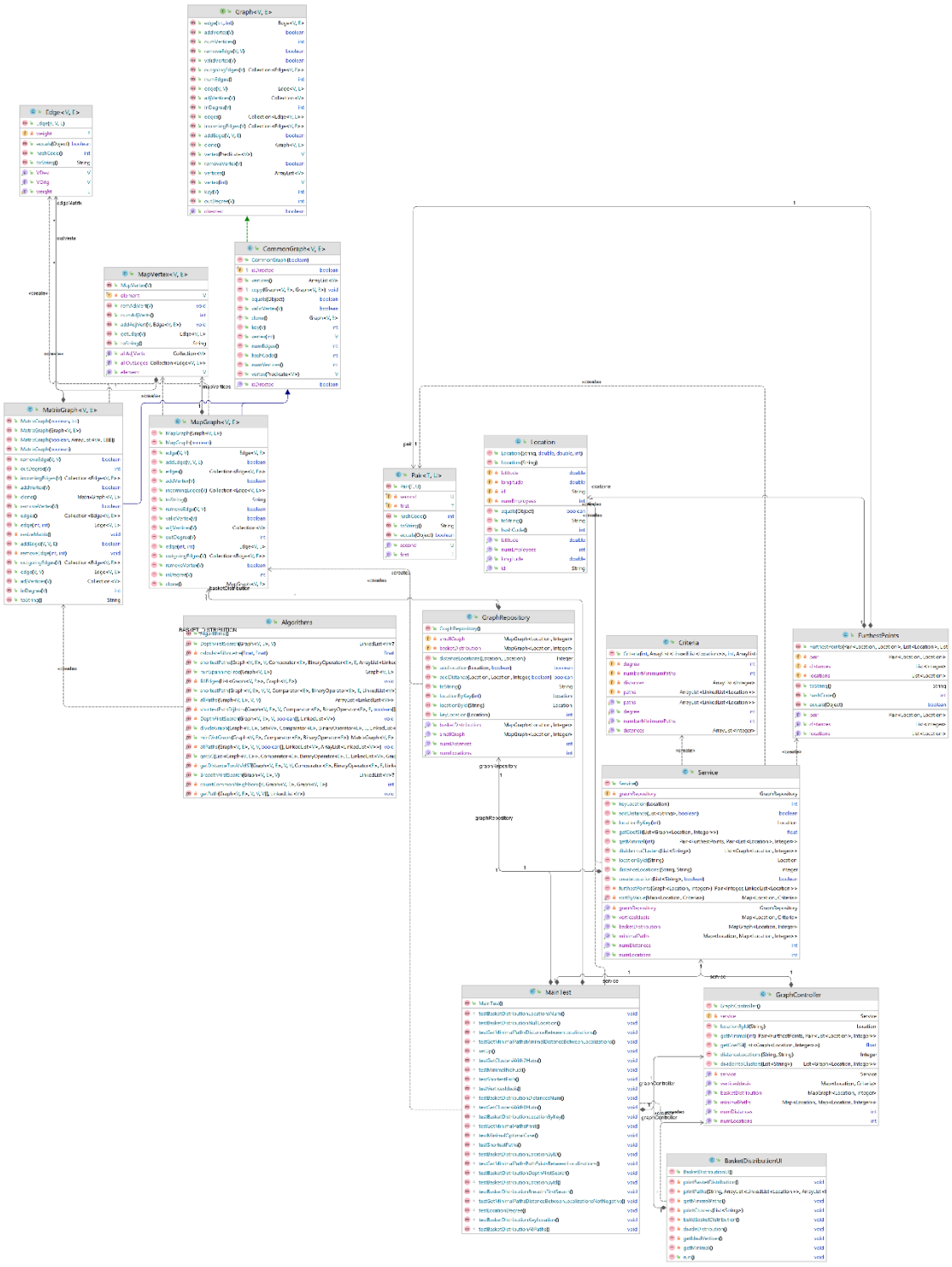
De maneira a inicializarmos a implementação dos Grafos e respetivos algoritmos, primeiramente foi necessário extrair os dados existentes nos ficheiros fornecidos. Tal como nos foi pedido, para executar os testes deve-se utilizar os ficheiros grandes, sendo assim optamos por extrair apenas esses dados. Após executada a extração, os dados ficam armazenados em duas Lists, uma de localizações e uma de distâncias, tal como é possível verificar nas imagens que se seguem:

```
public List<String> importTxtFile(String filePath, boolean skip) {

    InputStream stream = getClass().getClassLoader().getResourceAsStream(filePath);
    List<String> output = new ArrayList<>();
    Scanner sc = new Scanner(stream);

    if (skip)
        sc.nextLine();
    while (sc.hasNext()) {
        output.add(sc.nextLine());
    }
    sc.close();

    return output;
}
```

```
List<String> locations = importClass.importTxtFile( filePath: "esinf/locais_big.csv",  skip: true);
List<String> distances = importClass.importTxtFile( filePath: "esinf/distancias_big.csv",  skip: true);
```

Diagrama de Classes

Consoante o que fomos desenvolvendo e que iremos explicar mais à frente, o diagrama de classes final é o que se apresenta na seguinte imagem:



Exercício 1

Problema: Construir a rede de distribuição de cabazes a partir dos ficheiros (distancias xxx.csv e locais xxx.csv) com o formato disponibilizado. O grafo deve ser implementado usando a representação mais adequada para realizar de forma eficiente as funcionalidades pretendidas.

Resolução: Neste exercício optamos por representar o grafo através de um Map.

A escolha de um mapa para representar o grafo, caracterizado por ter poucas ligações em comparação com o número máximo possível de arestas entre os vértices, destaca-se pela eficiência espacial neste contexto específico. A disparidade entre o número real de ligações, que é de 1568 com 324 vértices, e o total possível, é clara.

Vale destacar que o grafo é não orientado, possuindo o dobro de arestas que constam no ficheiro, devido à natureza de uma rede de distribuição de cabazes, e, portanto, as 784 ligações passam a ser duplas, totalizando 1568.

Se tivéssemos optado por uma matriz nesta situação, teríamos de reservar espaço para todas as combinações possíveis de vértices (324 * 324), mesmo que a maioria desses espaços permanecesse vazia. Isso resultaria em uma utilização de memória muito maior do que a necessária para representar as 1568 ligações existentes. Em contrapartida, a estrutura de mapa aloca espaço apenas para as ligações efetivas, otimizando assim a eficiência espacial e garantindo uma utilização mais económica da memória disponível.

Com 324 vértices e 1568 ligações, a estrutura de mapa também oferece vantagens na procura de informações específicas. Comparado a uma matriz, onde a busca exigiria percorrer todas as combinações possíveis de vértices, o mapa permite um acesso direto e eficiente às ligações existentes. O tempo de busca num mapa é proporcional ao número de ligações efetivas, tornando-a uma escolha eficiente para localizar informações sobre conexões entre vértices.

Assim, a escolha do mapa como estrutura de dados economiza espaço e facilita a manipulação e recuperação de dados dentro do grafo. Esses são aspetos fundamentais, variando conforme as operações específicas exigidas pelo contexto do problema em questão.

Para a resolução deste exercício, recorreu-se à interface Graph e às classes CommonGraph, Edge, MapGraph e MapVertex. Foram também adotadas boas práticas de programação orientada a objetos, organizando o exercício em quatro camadas distintas: a UI, o controlador, o serviço e o repositório.

Através da BasketDistributionUI podemos dar início à criação do grafo:

```
private void buildBasketDistribution() {
    if (importController.importToGraph()) {
        System.out.println("Grafo criado com sucesso");
    } else
        System.out.println("Erro ao criar o grafo");
}
```

Primeiramente e tal como já foi mencionada anteriormente os dados são importados por meio do “ImportController” e armazenados em listas, sendo uma para as localizações e outra para as distâncias. Em caso de sucesso na operação de importação, essas listas são encaminhadas para o “Service”.

```
public boolean importToGraph() {
    boolean check = true;
    List<String> locations = importClass.importTxtFile(filePath:"esinf/locais_big.csv", skip:true);
    List<String> distances = importClass.importTxtFile(filePath:"esinf/distancias_big.csv", skip:true);
    if (locations.isEmpty() || distances.isEmpty())
        return false;
    if (!service.createLocation(locations) || !service.addDistance(distances))
        check = false;
    return check;
}
```

A função do "Service" é criar objetos do tipo "Localização" com base nos dados fornecidos e enviá-los para o "GraphRepository".

```
public boolean createLocation(List<String> locations) {
    boolean allAdded = true;
    for (String location : locations) {
        String[] line = location.split(",");
        if (!graphRepository.addLocation(new Location(line[0], Double.parseDouble(line[1].replace(',', '.')),
                Double.parseDouble(line[2].replace(',', '.')), Integer.parseInt(line[0].substring(2))))) {
            allAdded = false;
        }
    }
    return allAdded;
}
```

No "GraphRepository", utilizando o "GraphMap", os vértices necessários são criados e armazenados de forma a garantir uma construção eficiente do grafo a partir dos dados importados.

```
    public boolean addLocation(Location location) {
        return this.basketDistribution.addVertex(location);
    }
```

```
@Override
public boolean addVertex(V vert) {

    if (vert == null)
        throw new RuntimeException("Vértices não podem ser null!");
    if (validVertex(vert))
        return false;

    MapVertex<V, E> mv = new MapVertex<>(vert);
    vertices.add(vert);
    mapVertices.put(vert, mv);
    numVerts++;

    return true;
}
```

Posteriormente, o "Service" envia as distâncias para o "GraphRepository". Neste ponto, o "GraphRepository", novamente utilizando o "GraphMap", cria os edges necessários para conectar os vértices previamente criados, atribuindo os pesos correspondentes às distâncias. Este processo assegura uma representação completa e eficaz das relações entre as localizações no grafo.

```
public boolean addDistance(Location location1, Location location2, Integer distance) {
    return this.basketDistribution.addEdge(location1, location2, distance);
}
```

```java
@Override
public boolean addEdge(V vOrig, V vDest, E weight) {

    if (vOrig == null || vDest == null)
        throw new RuntimeException("Vértices não podem ser null!");
    if (edge(vOrig, vDest) != null)
        return false;

    if (!validVertex(vOrig))
        addVertex(vOrig);

    if (!validVertex(vDest))
        addVertex(vDest);

    MapVertex<V, E> mvo = mapVertices.get(vOrig);
    MapVertex<V, E> mvd = mapVertices.get(vDest);

    Edge<V, E> newEdge = new Edge<>(mvo.getElement(), mvd.getElement(), weight);
    mvo.addAdjVert(mvd.getElement(), newEdge);
    numEdges++;

    if (!isDirected)
        if (edge(vDest, vOrig) == null) {
            Edge<V, E> otherEdge = new Edge<>(mvd.getElement(), mvo.getElement(), weight);
            mvd.addAdjVert(mvo.getElement(), otherEdge);
            numEdges++;
        }

    return true;
}
```

Complexidade: A complexidade deste código está principalmente relacionada à construção do mapa de adjacência, sendo O(V + E) para a inserção de vértices e inserção de arestas.

Exercício 2

Problema: 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 restantes vértices**
- **centralidade: vértices com maior número de caminhos mínimos que passam por eles**

Para cada hub o número de colaboradores do hub é igual ao número do identificador da localidade em que o hub reside. Por exemplo um hub na localidade CT149 terá 149 colaboradores. O horário de funcionamento dos hubs deve ser definido de acordo com a tabela:

Localidade	Horário
CT1 . . . CT105	9h:00 – 14h:00
CT106 . . . CT215	11h:00 – 16h:00
CT216 . . . CT323	12h:00 – 17h:00

Resolução: Este exercício está distribuído em 4 camadas, a UI, o controller, o service e os repositórios. Na primeira camada são enviados os pedidos para o "GraphController". É também na UI que irão ser apresentados todos os resultados, neste caso as localizações, os critérios, o horário de funcionamento e o numero de colaboradores.

```
private void getIdealVertices() {

    Map<Location, Criteria> idealVertices = graphController.getVerticesIdeais();
    // Calculate the maximum length of the IDs
    int maxIdLength = Math.max(idealVertices.keySet().stream() Stream<Location>
            .mapToInt(Location -> Location.getId().length()) IntStream
            .max() OptionalInt
            .orElse( other: 0), "ID".length());
    // Calculate the maximum length of the degrees
    int maxDegreeLength = Math.max(idealVertices.values().stream() Stream<Criteria>
            .mapToInt(criteria -> Integer.toString(criteria.getDegree()).length()) IntStream
            .max() OptionalInt
            .orElse( other: 0), "Grau".length());
    // Calculate the maximum length of the number of minimum paths
    int maxNumPathsLength = Math.max(idealVertices.values().stream() Stream<Criteria>
            .mapToInt(criteria -> Integer.toString(criteria.getNumberMinimumPaths()).length()) IntStream
            .max() OptionalInt
            .orElse( other: 0), "N° Caminhos minimos".length());

    int maxLength = Math.max(maxIdLength, Math.max(maxDegreeLength, maxNumPathsLength));
    for (Map.Entry<Location, Criteria> entry : idealVertices.entrySet()) {
        System.out.println("\n-----------------------------------------------------------------------");
        String formatString = "| ID: %" + maxLength / 3 + "s | Degree: %" + maxLength / 3
                + "d | Número de Caminhos Mínimos: %" + maxLength / 3 + "d |\n";
        System.out.printf(formatString, entry.getKey().getId(), entry.getValue().getDegree(),
                entry.getValue().getNumberMinimumPaths());
        printPaths(entry.getKey().getId(), entry.getValue().getPaths(), entry.getValue().getDistances(), maxLength);
        formatString = "| Número de Colaboradores: %" + ((maxLength / 2) - 4) + "d | Horário: %" + (maxLength + 3)
                + "s |\n";
        int numEmployees = entry.getKey().getNumEmployees();
        String schedule = numEmployees > 105 ? (numEmployees < 216 ? "11h:00 - 16h:00" : "12h:00 - 17h:00")
                : "9h:00 - 14h:00";
        System.out.printf(formatString, numEmployees, schedule);
        System.out.println("-----------------------------------------------------------------------");
    }
}
```

```
private void printPaths(String id, ArrayList<LinkedList<Location>> arrayList, ArrayList<Integer> distances,
        int maxLength) {
    if (arrayList.size() != (distances.size())) {
        throw new IllegalArgumentException("As duas listas têm de ter o mesmo tamanho");
    }

    for (int i = 0; i < arrayList.size(); i++) {
        LinkedList<Location> path = arrayList.get(i);
        Integer distance = distances.get(i);
        String formatString = "| ID Destino: %" + (maxLength - 1) + "s | Distância: %" + (maxLength - 1)
                + "d m |\n";
        System.out.println("-----------------------------------------------------------------------");
        System.out.printf(formatString, path.getLast().getId(), distance);
    }
    System.out.println("-----------------------------------------------------------------------");
}
```

Após ser chamado o “GraphController”, este irá distribuir os diferentes pedidos para o respetivo serviço que será encarregue de pedir os dados aos repositórios e os processar.

```
public Map<Location, Criteria> getVerticesIdeais() {
    return service.getVerticesIdeais();
}
```

Na camada do service, este irá pedir os dados referentes a cada localização chamando o “GraphRepository” e o “Algorithms” e após os receber irá fazer um tratamento dos mesmos começando por os armazenar num Map<Location, Criteria> onde Criteria é correspondente a todos os critérios de cada localização. Este tratamento irá servir para determinar o último critério, ou seja, o número de caminhos mínimos que passam por um determinado ponto e também para ordenar as localizações por ordem decrescente de influência e centralidade.

```
public Map<Location, Criteria> getVerticesIdeais() {
    Map<Location, Criteria> map = new HashMap<>();
    int numberMinimumPaths = 0;
    int i;
    for (Location location : getBasketDistribution().vertices()) {
        int degree = graphRepository.getBasketDistribution().inDegree(location);
        ArrayList<LinkedList<Location>> locationPaths = new ArrayList<>();
        ArrayList<Integer> locationDistance = new ArrayList<>();
        Algorithms.shortestPaths(graphRepository.getBasketDistribution(), location, Integer::compare, Integer::sum,
                zero: 0, locationPaths, locationDistance);

        List<Integer> indices = IntStream.range(0, locationDistance.size()).mapToObj(Integer::valueOf)
                .collect(Collectors.toList());
        indices.sort(Comparator.comparing(locationDistance::get));
        List<LinkedList<Location>> sortedLocationPaths = indices.stream() Stream<Integer>
                .map(locationPaths::get) Stream<LinkedList<...>>
                .collect(Collectors.toList());
        locationDistance.sort(Integer::compareTo);
        locationPaths.clear();
        locationPaths.addAll(sortedLocationPaths);
        locationPaths.remove( index: 0);
        locationDistance.remove( index: 0);
        Criteria criteria = new Criteria(degree, locationPaths, numberMinimumPaths, locationDistance);
        map.put(location, criteria);
    }
    for (Map.Entry<Location, Criteria> entry : map.entrySet()) {
        numberMinimumPaths = 0;
        Location location = entry.getKey();
        for (Map.Entry<Location, Criteria> entry2 : map.entrySet()) {
            if (!entry.getKey().equals(entry2.getKey())) {
                for (i = 0; i < entry2.getValue().getPaths().size(); i++) {
                    if (entry2.getValue().getPaths().get(i).contains(location)) {
                        numberMinimumPaths++;
                    }
                }
            }
        }
        entry.getValue().setNumberMinimumPaths(numberMinimumPaths);
    }
    map = sortByValue(map);
    return map;
```

De forma a ordenar o Map pelas ordens desejadas foi utilizado o seguinte método:

```
private static Map<Location, Criteria> sortByValue(Map<Location, Criteria> map) {
    // Convert the map to a list of entries
    List<Map.Entry<Location, Criteria>> list = new ArrayList<>(map.entrySet());

    // Sort the list using a custom comparator
    list.sort(Comparator.comparing((Map.Entry<Location, Criteria> entry) -> entry.getValue().getDegree())
            .thenComparing(entry -> entry.getValue().getNumberMinimumPaths()).reversed());

    // Convert the sorted list back to a map
    Map<Location, Criteria> sortedMap = new LinkedHashMap<>();
    for (Map.Entry<Location, Criteria> entry : list) {
        sortedMap.put(entry.getKey(), entry.getValue());
    }

    return sortedMap;
}
```

Este método irá ordenar, através do método sort, por ordem decrescente de influência e, em caso de empate, por ordem decrescente de centralidade. Após a ordenação, os dados ordenados serão inseridos de volta no Map e este Map será transferido pelas diferentes camadas até chegar de volta à UI onde os dados poderão ser apresentados ao utilizador.

Por fim, na camada dos repositórios, são chamados dois métodos, um no "GraphRepository" e outro nos "Algorithms". O método chamado no "GraphRepository" é o "getBasketDistribution" que posteriormente irá chamar o "inDegree" que tem como função retornar o grau do vértice enviado por parâmetro, sendo assim executamos este método x vezes, ou seja, executamos uma vez para cada localização.

```
@Override
public int inDegree(V vert) {

    if (!validVertex(vert))
        return -1;

    int degree = 0;
    for (V otherVert : mapVertices.keySet())
        if (edge(otherVert, vert) != null)
            degree++;

    return degree;
}
```

Já o método utilizado existente na classe “Algorithms” é o shortestPaths que retorna todos os caminhos para um determinado vértice e as respetivas distâncias através do uso do algoritmo de Dijkstra.

```
public static <V, E> boolean shortestPaths(Graph<V, E> g, V vOrig,
                                           Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                           ArrayList<LinkedList<V>> paths, ArrayList<E> dists) {

    if (!g.validVertex(vOrig)) {
        return false;
    }

    int vertices = g.numVertices();
    boolean[] visited = new boolean[vertices];
    V[] pathKeys = (V[]) new Object[vertices];
    E[] dist = (E[]) new Object[vertices];

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

    dists.clear();
    paths.clear();
    for (V v : g.vertices()) {
        dists.add(dist[g.key(v)]);
        LinkedList<V> shortPath = new LinkedList<>();
        getPath(g, vOrig, v, pathKeys, shortPath);
        paths.add(shortPath);
    }

    return true;
}
```

Complexidade: A complexidade deste método é $O(3V^3+V^2+N^3+N^2+N)$

Exercício 3

Problema: Dado um veículo, a sua autonomia e atendendo a que os carregamentos só podem ser feitos nas localidades, determinar o percurso mínimo possível entre os dois locais mais afastados da rede de distribuição, indicando o número de paragens necessárias para carregamentos do veículo.

Resolução

Após o utilizador escolher a opção 1 no MainMenuUI, este irá levar ao BasketDistributionUI onde irá deparar-se com a opção 3 que corresponde à USEI03, onde se pretende realizar o caminho mais longo, pelo trajeto mais curto, de um determinado veículo, com uma dada autonomia. Além disso, temos de devolver o percurso entre os dois locais mais afastados da rede de distribuição: local de origem, os locais de passagem (indicando os locais onde foi feito carregamento da viatura), distância entre os locais do percurso, local destino, distância total do percurso e o número total de carregamentos.

```
public Pair<FurthestPoints, Pair<List<Location>, Integer>> getMinimal(int autonomy) {
    // this graph takes a long time to run
    Graph<Location, Integer> distributionGraph = graphRepository.getBasketDistribution();
    // get graph pequeno
    //Graph<Location, Integer> distributionGraph = graphRepository.getSmallGraph();

    // furthest points
    Pair<Integer, LinkedList<Location>> furthestPoints = furthestPoints(distributionGraph);
    // create shortest path
    LinkedList<Location> shortPath = furthestPoints.getSecond();
```

Primeiramente, vamos buscar o grafo, previamente criado, ao seu repositório. Em seguida, invocamos a função furthestPoint(), a qual proporciona o caminho mais longo dentro dos mais pequenos e a sua respetiva distância.

```
private Pair<Integer, LinkedList<Location>> furthestPoints(Graph<Location, Integer> g) {

    Integer maxDist = 0;
    Pair<Integer, LinkedList<Location>> furthestPoints = null;

    for (Location location : g.vertices()) {
        ArrayList<LinkedList<Location>> locationPaths = new ArrayList<>();
        ArrayList<Integer> locationDistance = new ArrayList<>();
        Algorithms.shortestPaths(g, location, Integer::compare, Integer::sum, zero:0, locationPaths,
                locationDistance);

        for (int i = 0; i < locationDistance.size(); i++) {

            if (locationDistance.get(i) > maxDist) {
                maxDist = locationDistance.get(i);
                furthestPoints = new Pair<>(maxDist, locationPaths.get(i));
            }
        }

    }
    return furthestPoints;
}
```

Dentro do método furthestPoint() é chamado o algoritmo shortestPaths() para todos os vértices e é calculado o caminho mais longo.

```
int distance = furthestPoints.getFirst();
int distanceAutonomy = autonomy;
List<Location> rechargeLocations = new ArrayList<>();
List<Integer> distances = new ArrayList<>();
rechargeLocations.add(shortPath.getFirst());
// get recharge locations
for (int i = 0; i < shortPath.size() - 1; i++) {
    Location location1 = shortPath.get(i);
    Location location2 = shortPath.get(i + 1);
    int distanceBetweenPoints = distributionGraph.edge(location1, location2).getWeight();
    distances.add(distanceBetweenPoints);
    try {
        // if distance between points is greater than autonomy
        if (distanceBetweenPoints > autonomy) {
            throw new RuntimeException("\nO veiculo ficou sem bateria na viagem entre local: "
                    + location1.getId() + " e " + location2.getId() + "\n");
        }

        distanceAutonomy -= distanceBetweenPoints;
        // if distance autonomy is greater than autonomy
        if (distanceAutonomy < distanceBetweenPoints && rechargeLocations.get(0) != location1) {
            rechargeLocations.add(location1);
            distanceAutonomy = autonomy;
        }
    } catch (RuntimeException e) {
        // Declaring ANSI_RESET so that we can reset the color
        System.err.println(e.getMessage());
        break;
    }
}
```

Por fim, adicionamos a primeira localização á lista de carregamento para o veículo começar a viagem com a bateria cheia, de seguida entramos num loop que para cada par de localizações calcula a sua distância e verifica se o veículo tem bateria suficiente para se deslocar para a próxima localização ou precisa de ser carregado. O método retorna um Pair contendo dois conjuntos de informações. O primeiro conjunto inclui os pontos mais distantes, o caminho mais curto, e as distâncias entre as localizações consecutivas. O segundo conjunto inclui os locais de recarga e a distância total do caminho mais curto.

Complexidade:

A complexidade deste método é principalmente determinada pelo algoritmo shortestPaths(), de complexidade O(V^3+V^2), que usa o algoritmo shortestPathDijkstra() que por sua vez tem de complexidade O(V+V^2). Em suma a complexidade total deste método:

$$O\left((V^3 + V^2) \cdot V\right)$$

Exercício 4

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

Resolução:

Após o utilizador escolher a opção 1 no MainMenuUI, este irá levar ao BasketDistributionUI onde irá deparar-se com a opção 4 que corresponde à USEI04, onde se pretende determinar a distância total mínima que liga todas as localidades. Além disso, temos de devolver a ligação entre dois locais com a distância correspondente.

```
private void getMinimalPaths() {
    Map<Location, Map<Location, Integer>> map = graphController.getMinimalPaths();
    Integer sumDistance = 0;
    for (Map.Entry<Location, Map<Location, Integer>> entry : map.entrySet()) {
        String locationId = entry.getKey().getId();
        for (Map.Entry<Location, Integer> entry2 : entry.getValue().entrySet()) {
            String location1Id = entry2.getKey().getId();
            int distance = entry2.getValue();
            System.out.println(locationId + " -> " + location1Id + "; Distância: " + distance);
            sumDistance = sumDistance + distance;
        }
    }
    System.out.print("\n");
    System.out.println("Distância total: " + sumDistance);
}
```

Primeiramente, definimos a variável “map”, que irá apresentar o resultado do método “getMinimalPaths” do objeto “graphController”, ou seja, irá retornar um mapa que contém os locais e a distância entre eles, e a variável “sumDistance”, que representada a soma total das distâncias dos caminhos mínimos.

Para imprimirmos a ligação entre as localidades e a distância correspondente, tal como determinar o total das distâncias, iniciamos um ciclo for onde obtermos a primeira localidade, pois cada entrada representa um local e os caminhos mínimos associados a ele mesmo. De seguida, realizamos um segundo ciclo for, no qual as entradas representam um local adjacente à primeira localidade e a distância associada entre esses locais, e atualizamos a soma total das distâncias para, posteriormente, apresentarmos o valor.

```
public Map<Location, Map<Location, Integer>> getMinimalPaths() {
    return service.getMinimalPaths();
}
```

```
public Map<Location, Map<Location, Integer>> getMinimalPaths() {
    Map<Location, Map<Location, Integer>> map = new HashMap<>();
    MapGraph<Location, Integer> graph = graphRepository.getBasketDistribution();
    Graph<Location, Integer> minDistGraph = Algorithms.minSpanningTree(graph);
    for (Edge<Location, Integer> edge : minDistGraph.edges()) {
        Location location = edge.getVDest();
        Location location1 = edge.getVOrig();
        String locationId = location.getId();
        String location1Id = location1.getId();
        int distance = edge.getWeight();
        if (locationId.compareTo(location1Id) < 0) {
            if (map.containsKey(location)) {
                map.get(location).put(location1, distance);
            } else {
                Map<Location, Integer> map1 = new HashMap<>();
                map1.put(location1, distance);
                map.put(location, map1);
            }
        }
    }
    return map;
}
```

O método "getMinimalPaths()" retorna um mapa contendo os caminhos mínimos entre os locais, em que as keys são do tipo Location e os values é outro mapa que apresenta keys do tipo Location e values do tipo Integer.

Neste método, para se obter as informações das localidades e a distância entre as mesmas sobre onde se encontra uma rede de distribuição de cabazes de produtos agrícolas, chamamos o método "getBasketDistribution()", que fornece um grafo do tipo "MapGraph" que contém as informações sobre a rede de distribuição de cabazes de produtos agrícolas. A partir deste grafo, iremos determinar o caminho mínimo entre locais através do método "minSpanningTree(graph)", que retorna o grafo "minDistGraph" com os caminhos mínimos.

```
public MapGraph<Location, Integer> getBasketDistribution() {
    return basketDistribution;
}
```

```java
public static <V, E extends Comparable<E>> Graph<V, E> minSpanningTree(Graph<V, E> g) {
    PriorityQueue<Edge<V, E>> pq = new PriorityQueue<>(Comparator.comparing(Edge::getWeight));
    Set<V> visited = new HashSet<>();
    Graph<V, E> mst = new MatrixGraph<>( directed: false, g.numVertices());

    V start = g.vertices().get(0);
    visited.add(start);

    for (Edge<V, E> edge : g.outgoingEdges(start)) {
        pq.add(edge);
    }

    while (!pq.isEmpty()) {
        Edge<V, E> edge = pq.poll();
        V v = edge.getVDest();
        if (!visited.contains(v)) {
            visited.add(v);
            mst.addEdge(edge.getVOrig(), v, edge.getWeight());
            for (Edge<V, E> e : g.outgoingEdges(v)) {
                if (!visited.contains(e.getVDest())) {
                    pq.add(e);
                }
            }
        }
    }

    return mst;
}
```

O método “minSpanningTree(graph)” implementa o algoritmo de Prim, que usa uma estrutura de dados de fila de prioridade (PriorityQueue) para manter as arestas ordenadas pelo peso, e constrói a árvore mínima do grafo “graph”, começando por um vértice inicial, que é o primeiro vértice do grafo, adicionando esse vértice ao conjunto de vértices visitados. Depois, enquanto a PriorityQueue “pq” não estiver vazia, removemos a aresta de menor peso de “pq” e obtemos o vértice de destino “v” da aresta, em que caso esse vértice não foi visitado, adicionamos ao conjunto de vértices visitados e a aresta ao grafo e, de seguida, adicionamos todas as arestas de “v” à “pq”. Por fim, retornamos o grafo que representa a árvore mínima do grafo “graph”.

De seguida, iteramos cada edge do grafo com esses caminhos e obtemos os vértices de origem (“location”) e de destino (“location1”) e a distância entre os mesmos. Posteriormente, realizamos a condição if entre os IDs das localizações, em que comparamos o locationId com o location1Id para determinar a ordem dos IDs dos locais.

Finalmente, verificamos se o mapa, que iremos retornar, já contém a key “location”, em que caso se verifique, adicionamos a key “location” ao mapa com a associação da key “location1” e o valor da distância entre ambas, e se não se verificar, criamos um mapa “map1”, onde colocamos a key “location1” e o valor da distância, e adicionamos ao mapa “map” a key “location” e o mapa criado. Esta lógica assegura que cada par de localidades e a distância associada só é registada uma vez, evitando a duplicação de relações no mapa “map”.

Complexidade: A complexidade deste código está predominantemente associada à construção da árvore mínima do grafo, que é $O(E^2 + E + E \log V)$.

Exercício 5

Problema: Considerando a rede de ligação mínima de distribuição de cabazes e dada uma lista de N hubs, dividir a rede em N clusters conexos e o mais coesos/separados possível.

Resolução:

A resolução consiste de dois algoritmos, o algoritmo divideGraph que divide o Grafo dado em clusters, e getSC que calcula o coeficiente de silhueta para os clusters.

Primeiro, veremos o divideGraph():

```
2 usages  ± José Sá
public static <V, E> List<Graph<V, E>> divideGraph(Graph<V, E> g, Set<V> verticeList, Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                                   LinkedList<V> shortPath) {
    int numVertices = g.numVertices();
    List<Graph<V, E>> clusterList = new ArrayList<>();
    for (V vertice : verticeList) {
        Graph<V, E> graph = new MatrixGraph<>( directed: false, numVertices);
        graph.addVertex(vertice);
        clusterList.add(graph);
    }
    for (V vertice : g.vertices()) {
        int maxClusterIndex = -1;
        int minDist = Integer.MAX_VALUE;
        int currentCluster = 0;
        for (V hub : verticeList) {
            int dist = getDistanceTwoVsMST(g, vertice, hub, ce, sum, zero, shortPath);
            if (minDist > dist) {
                minDist = dist;
                maxClusterIndex = currentCluster;
            }
            currentCluster++;
        }
        clusterList.get(maxClusterIndex).addVertex(vertice);
    }
    fillEdges(clusterList, g);
    return clusterList;
}
```

A função primeiro usa a lista de vértices que servirão de centros para os clusters para criar os subgrafos vazios onde os clusters serão guardados, e de seguida a função vai para cada vértice do grafo MST dado, e vê qual é o hub mais próximo desse vértice. O vértice é então guardado no cluster do hub mais próximo.

Após ir por todos os vértices do grafo, a função preenche os clusters com as arestas do grafo original.

```
1 usage  ± José Sá
private static <V, E> void fillEdges(List<Graph<V, E>> clusterList, Graph<V, E> graph) {
    for (Graph<V, E> cluster : clusterList) {
        for (V vertice : cluster.vertices()) {
            for (V dest : graph.adjVertices(vertice)) {
                if (cluster.validVertex(dest)) {
                    if (cluster.edge(dest, vertice) == null && cluster.edge(vertice, dest) == null) {
                        E weight = graph.edge(vertice, dest).getWeight();
                        cluster.addEdge(vertice, dest, weight);
                    }
                }
            }
        }
    }
}
```

```
2 usages   ± José Sá
public static <V, E> float getSC(List<Graph<V, E>> clusterList, Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                 LinkedList<V> shortPath, Graph<V, E> originalGraph) {
    List<Float> sillouetteAverages = new ArrayList<>();
    for (Graph<V, E> cluster : clusterList) {
        float sillouetteSum = 0;
        float numSillouettes = 0;
        float lowAvgDistOut = Float.MAX_VALUE;
        float sumDistIn = 0;
        for (V vertice : cluster.vertices()) {
            for (V verticeIn : cluster.vertices()) {
                if (!vertice.equals(verticeIn)) {
                    shortestPath(cluster, vertice, verticeIn, ce, sum, zero, shortPath);
                    int dist = 0;
                    for (int i = 0, u = 1; u < shortPath.size(); i++, u++) {
                        dist += (int) cluster.edge(shortPath.get(i), shortPath.get(u)).getWeight();
                    }
                    sumDistIn += dist;
                }
            }
            float avgDistIn = sumDistIn / (cluster.numVertices() - 1);
            for (Graph<V, E> clusterout : clusterList) {
                float sumDistOut = 0;
                if (!cluster.equals(clusterout)) {
                    for (V outVert : clusterout.vertices()) {
                        shortestPath(originalGraph, vertice, outVert, ce, sum, zero, shortPath);
                        int dist = 0;
                        for (int i = 0, u = 1; u < shortPath.size(); i++, u++) {
                            dist += (int) originalGraph.edge(shortPath.get(i), shortPath.get(u)).getWeight();
                        }
                        sumDistOut += dist;
                    }
                    float avgDistOut = sumDistOut / clusterout.numVertices();
                    if (avgDistOut < lowAvgDistOut) {
                        lowAvgDistOut = avgDistOut;
                    }
                }
            }
            sillouetteSum += calculateSillouette(lowAvgDistOut, avgDistIn);
            numSillouettes++;

        }
        sillouetteAverages.add(sillouetteSum / numSillouettes);
    }
    if (sillouetteAverages.stream().max(Float::compareTo).isPresent()) {
        return sillouetteAverages.stream().max(Float::compareTo).get();
    } else {
        return 0;
    }
}
```

A função getSC() vai por cada vértice de cada cluster, e calcula a(i), que é a distância média entre esse vértice e todos os outros vértices do mesmo cluster, e b(i), que é a menor distancia média do vértice para todos os vértices dos outros clusters. Após obter estes resultados, calculateSillouette calcula a silhueta para esse vértice, e adiciona a silhueta para a soma de todas elas. Após obter todas as silhuetas para um dado cluster, é calculada e guardada a média e a função passa para o próximo cluster. Após ir por todos os clusters, a função retorna a média maior.

```
1 usage   José Sá
private static float calculateSillouette(float lowAvgDistOut, float avgDistIn) {
    if (avgDistIn < lowAvgDistOut) {
        return (1 - (avgDistIn / lowAvgDistOut));
    } else if (avgDistIn > lowAvgDistOut) {
        return ((lowAvgDistOut / avgDistIn) - 1);
    } else {
        return 0;
    }
}
```

Complexidade:

A complexidade de divideGraph é determinada principalmente pelos loops que são usados para ver a distância de cada vértice para cada um dos hubs. Para além disso, há a criação dos clusters iniciais, o uso do algoritmo dijkstra para obter as distâncias, e a função que adquire as arestas para preencher os clusters.

Assim a complexidade de divideGraph é $O(N + N*V^3 + V^2)$ onde N é o numero de hubs, e V é o número de vértices no grafo MST.

A complexidade de getSC é determinada principalmente pelos loops que determinam os valores de a(i) e b(I) e o uso do algoritmo de dijkstra para determinar as distâncias.

Assim a complexidade de getSC é $O(N^2 * V^4)$.

Melhoramentos Possíveis

Após a finalização do trabalho, apercebemo-nos de que poderíamos ter adotado os seguintes pontos:

- Implementação de mais algoritmos alternativos
- Uso de métodos mais eficientes

Conclusão

Em suma, foram elaborados com sucesso todos os exercícios propostos para a realização do Segundo Projeto e através do relatório foi atingido o objetivo de expor o trabalho realizado pelos membros da equipa na aplicação desenvolvida.

A aplicação foi concretizada com conhecimentos adquiridos na disciplina ESINF que foram alargados através de pesquisa utilizando diferentes recursos.