Instituto Superior de Engenharia do Porto

Grupo 01 – Turma 2DB

<Projeto Integrador>

Sprint 1

04/12/2022

Relatório

José Barbosa, 1211359
Rodrigo Peireso, 1211345
Inês Costa, 1210814
Joana Perpétuo, 1211148
Délcio Monjane, 1211739

Índice

1. Introdução

Este relatório tem como objetivo expor a aplicação desenvolvida no âmbito de ESINF, no contexto do Projeto Integrador da Licenciatura em Engenharia Informática no ISEP. O trabalho consiste no desenvolvimento de uma solução informática que apoie a gestão de uma empresa responsável pela gestão de uma exploração agrícola em modo biológico. Neste documento é apresentada uma visão generalizada da aplicação desenvolvida assim como a descrição das principais funcionalidades da solução, acompanhada da respetiva análise de complexidade.

2. App Overview

As funcionalidades foram desenvolvidas utilizando a linguagem *Java* e o programa está estruturado de acordo com os padrões GRASP de desenvolvimento de software (conhecimento adquirido na unidade curricular ESOFT) e com a programação orientada a objetos. Para além disso, as classes necessárias foram implementadas usando a ferramenta *Java Collection FrameWork* (JCF) e aplicando conhecimentos sobre grafos.

Esta pequena aplicação efetua algumas interações com o utilizador, nomeadamente para a inserção de *input,* e carece de uma interface gráfica - apenas pode ser corrida na consola do ambiente de desenvolvimento de *Java* (neste caso o *IntelliJ IDEA*).

Todos os métodos que são essenciais para o bom funcionamento do programa foram testados recorrendo a testes unitários.

3. Funcionalidades implementadas

3.1. US301 - Construir a rede de distribuição de cabazes a partir da informação fornecida nos ficheiros

ReadClientesProdutoresFile:

Este método permite importar ficheiros .csv relativos a clientes e Produtores, contendo informação relativa ao seu código de localização, coordenadas geográficas (latitude e longitude) e código de cliente (C) ou código de empresa (E), dependendo se se trata de um cliente ou de um produtor. É verificado se aquela empresa/cliente já pertence à rede e caso não pertença, adiciona-o (vértice).

```java
public void readClientesProdutoresFile(File filename) throws IOException {
    try{
    BufferedReader br = new BufferedReader(new FileReader(filename));
    br.readLine(); // skip first line
    String line = br.readLine();
    while ((line != null)) {
        String[] values = line.split(MAIN_SPLIT_REGEX);
        if (values.length == 1)
            values = line.split(SPLIT_REGEX);
        else {
            for (int i = 0; i < values.length; i++) {
                values[i] = values[i].replace( target: "\"",  replacement: "");
            }
        }
        ClienteProdutorEmpresa cpe = new ClienteProdutorEmpresa(values[0], Float.parseFloat(values[1]), Float.parseFloat(values[2]), values[3]);
        if(!cpeStore.containsCPE(cpe.getId())) {
            cpeStore.addCPE(cpe);
            localizacaoStore.addLocalizacao(cpe.getLocalizacao());
            graph.addVertex(cpe.getLocalizacao());
        }
        line = br.readLine();
    }
    }catch (NumberFormatException e) {
        throw new IOException("Invalid ID format");
    }catch (IllegalArgumentException e){
        throw new IOException(e.getMessage());
    }
}
```

ReadDistancesFile:

Este método importa informação presente num ficheiro .csv relativas às distâncias entre duas localizações, identificadas pelos seus respetivos códigos. Verifica se o caminho entre localizações já está marcado e caso tal não se verifique, adiciona-o (aresta).

```java
/**
 * It reads a file refering to the distances between two locations and puts it in a graph
 *
 * @param filename The file to be read
 */
2 usages   Jose_Barbosa_1211359
public void readDistancesFile(File filename) throws IOException {
    try{
    BufferedReader br = new BufferedReader(new FileReader(filename));
    br.readLine(); // skip first line
    String line = br.readLine();
    while ((line != null)) {
        String[] values = line.split(MAIN_SPLIT_REGEX);
        if (values.length == 1)
            values = line.split(SPLIT_REGEX);
        else {
            for (int i = 0; i < values.length; i++) {
                values[i] = values[i].replace( target: "\"",  replacement: "");
            }
        }

        Localizacao lc1=localizacaoStore.getLocalizacao(values[0]);
        Localizacao lc2=localizacaoStore.getLocalizacao(values[1]);
        if(lc1!=null && lc2!=null){
            if(!graph.edgeExists(lc2,lc1)){
                graph.addEdge(lc2,lc1,Integer.parseInt(values[2]));
            }
        }
        line = br.readLine();
    }
}catch (NumberFormatException e) {
        throw new IOException("Invalid ID format");
    }catch (IllegalArgumentException e){
        throw new IOException(e.getMessage());
    }
}
```

```
BEM-VINDO/A!
Selecione uma opcao:

1 - Carregar informacao a partir de ficheiros CSV
2 - Verificar se o grafo e conexo e o nr minimo de conexoes para chegar dum vertice a qualquer outro
3 - Definir os hubs da rede de distribuicao
4 - Determinar o hub mais proximo para cada cliente
5 - Determinar a rede que conecte todos os clientes e produtores com uma distancia total minima
6 - Controlador de rega
0 - Sair
1

Por favor insira o ficheiro de Clientes e produtores que quer importar, ou "0" para voltar
esinf/grafos/grafos/Small/clientes-produtores_small.csv

Ficheiro importado corretamente!

Por favor insira o ficheiro de distancias que quer importar, ou "0" para voltar
esinf/grafos/grafos/Small/distancias_small.csv

Ficheiro importado corretamente!
```

3.2. US302 - Verificar se o grafo carregado é conexo e devolver o número mínimo de ligações necessário para nesta rede qualquer cliente/produtor conseguir contactar um qualquer outro

Esta US tem como objetivo saber se grafo carregado é conexo e saber o número mínimo de ligações/conexões para chegar a qualquer cliente/produtor a partir de qualquer cliente/produtor.

```
Ficheiro importado corretamente!

Selecione uma opcao:

1 - Carregar informacao a partir de ficheiros CSV
2 - Verificar se o grafo e conexo e o nr minimo de conexoes para chegar dum vertice a qualquer outro
3 - Definir os hubs da rede de distribuicao
4 - Determinar o hub mais proximo para cada cliente
5 - Determinar a rede que conecte todos os clientes e produtores com uma distancia total minima
6 - Controlador de rega
0 - Sair
2
O grafo é conexo, e com 6 conexões conseguimos atingir qualquer Destino a partir de qualquer Origem.
```

Foram concebidas duas classes para fazer a implementação dessa funcionalidade: ConnectionsUI que faz a ligação entre a interface do utilizador e os métodos do controlador:

```java
2 usages  Delcio Monjane +1 *
public class ConnectionsUI implements  Runnable{
3 usages
private final ConnectionsController CTRL;

    Delcio Monjane
    public ConnectionsUI(ConnectionsController CTRL) { this.CTRL = CTRL; }

    1 usage  Delcio Monjane
    public ConnectionsUI() { CTRL = new ConnectionsController(); }


    Delcio Monjane +1 *
    @Override
    public void run() {
        int result = CTRL.minConnevtions();
        if(result == -1)
            System.out.println("O grafo nao e conexo");
        else
            System.out.println("O grafo é conexo, e com " + result + " conexões conseguimos atingir qualquer Destino a partir de qualquer Origem."
    }

}
```

E a classe ConnectionsController que liga os métodos do domínio. Esta classe vai aos dados da App e leva o grafo a ser analisado e usa os métodos da interface graph para ver as conexões.

```java
2 usages  Delcio Monjane
public int minConnevtions(){
    return Algorithms.minConnections(app.getGraph(), Integer::compare, Integer::sum,  zero: 0);
```

Escolheu se uma implementação genérica para o método que calcula as conexões, este método procura todos os caminhos de um vértice para o outro e assume como o menor caminho dessa iteração o que tiver menor conexões, e compara com o valor de retorno o minConnections se for maior esse valor passa a ser o valor de retorno. Faz este procedimento para todos os vértices do grafo. Caso ele não encontre nenhum caminho entre dois vértices o procedimento para.

```java
public static <V, E> int minConnections(Graph<V, E> g, Comparator<E> ce, BinaryOperator<E> sum, E zero) {
    int minConnections = 0;
    boolean[] visited = new boolean[g.numVertices()];
    for (V vert : g.vertices()) {
        for (V vert2 : g.vertices()) {
            if (!vert.equals(vert2)) {
                LinkedList<V> path = new LinkedList<>();
                ArrayList<LinkedList<V>> paths = new ArrayList<>();
                Algorithms.allPaths(g, vert, vert2, visited, path, paths);
                if (paths.size() == 0) {
                    return -1;
                } else {
                    LinkedList<V> caminhoMin = null;
                    for (LinkedList<V> caminho : paths) {
                        if (caminhoMin == null)
                            caminhoMin = caminho;
                        else if (caminhoMin.size() -1 > caminho.size() - 1)
                            caminhoMin = caminho;
                    }
                    if (minConnections < caminhoMin.size() - 1) {
                        minConnections = caminhoMin.size() - 1;
                    }
                }
            }
        }
    }
    return minConnections;
}
```

3.3. US303 - Definir os hubs da rede de distribuição

Esta US tem como objetivo encontrar as n Empresas que, em média, estão mais próximas de todos os Clientes e Produtores. Para isso foram concebidas duas classes: HubsDistributionUI e HubsDistributionController.

HubsDistributionUI:

```
package UI;

import Controller.App;
import Controller.HubsDistribuicaoController;

import java.util.Scanner;

public class HubsDistribuicaoUI {   JoanaSofia, 01/12/2022 17:23 • working on us303

    App app;
    HubsDistribuicaoController ctrl = new HubsDistribuicaoController();

    public HubsDistribuicaoUI() { app = App.getInstance(); }

    public void run() {

        int n;
        Scanner sc = new Scanner(System.in);
        System.out.println("\nIntroduza o numero de hubs que pretende definir: ");
        n = sc.nextInt();
        System.out.println("\nHubs mais proximos: ");
        ctrl.getMediaDistancia(app.getGraph(), n);

    }
}
```

Aqui é inicializado o controller que vai ser usado para chamar o seu método. O UI vai pedir ao utilizador para introduzir o número de empresas que pretende definir (n) e a seguir é chamado o método *getMediaDistancia* que recebe como parâmetro o grafo que foi construído durante a leitura dos dados dos ficheiros, e o valor de n.

```java
public Map<ClienteProdutorEmpresa, Double> getMediaDistancia(Graph<Localizacao, Integer> grafo, int n) {  Complexity is 20 You must be kidding
    if(n > getEmpresas().size() || n < 1) System.out.println("O numero de hubs inserido e invalido");
    else {
    Map<ClienteProdutorEmpresa, Double> medias = new HashMap<>();
    List<ClienteProdutorEmpresa> listaEmpresas = getEmpresas();
    ArrayList<LinkedList<Localizacao>> listaCaminhos = new ArrayList<>();
    for (ClienteProdutorEmpresa e : listaEmpresas) {
        ArrayList<Integer> distancias = new ArrayList<>();
        Double soma = (double) 0;
        Double numeroCPE = (double) 0;
        Algorithms.shortestPaths(grafo,
                e.getLocalizacao(),
                Integer::compare,
                Integer::sum,
                 zero: 0,
                listaCaminhos,
                distancias);

        for (Integer value : distancias) {
            if (value != 0) {
                soma += value;
                numeroCPE++;
            }
        }
        if(numeroCPE != 0) {
            double m = soma / numeroCPE;
            medias.put(e, m);
        }
    }
```

Neste método começamos por verificar se o n que foi inserido é um número válido no contexto do problema, ou seja, se não ultrapassa o número de Empresas contidas no ficheiro, nem é menor que 1, caso contrário o método retorna o valor *null*.

Em seguida são criadas as estruturas de dados necessárias: um mapa de *ClienteProdutorEmpresa, Double (medias)*, que irá conter cada Empresa referida no ficheiro e a respetiva distância média a todos os Clientes/Produtores. O array list de *ClienteProdutorEmpresa (listaEmpresas)* vai conter todas as Empresas contidas no ficheiro. O array list de linked lists *(listaCaminhos)* será um parâmetro do algoritmo que iremos utilizar.

Começamos então por iterar todas empresas contidas na lista de Empresas: para cada empresa existente vamos ter um array list de *Integer (distancias)* que irá conter a correspondente distância dessa Empresa mínima a cada Cliente/Produtor.

O método getEmpresas utiliza o mapa que foi criando durante a leitura dos ficheiros que contém todos os Clientes, Produtores e Empresas do ficheiro para obter as Empresas e guardá-las num array list.

```java
public class HubsDistribuicaoController {  Complexity is 21 You must be kidding

    App app;


    public HubsDistribuicaoController() { app = App.getInstance(); }


    public List<ClienteProdutorEmpresa> getEmpresas() {  Complexity is 4 Everything is cool!
        Map<String, ClienteProdutorEmpresa> mapCPE = app.getClienteProdutorEmpresaStore().getMapCPE();
        List<ClienteProdutorEmpresa> listaEmpresas = new ArrayList<>();
        for (Map.Entry<String, ClienteProdutorEmpresa> entry : mapCPE.entrySet()) {
            if (entry.getValue().isEmpresa()) {
                listaEmpresas.add(entry.getValue());
            }
        }
        return listaEmpresas;
        // 5 empresas - small
    }
```

Inicializamos uma variável soma e uma variável que servir de contador para contar o número de Clientes e Produtores. A seguir chamamos o algoritmo *shortestPaths* a partir da classe *Algorithms.* Este algoritmo, em comparação com o algoritmo *shortestPath,* reduz a complexidadede do programa e por isso tem uma melhor performance. Depois do algoritmo ser chamado os array lists já vão ter os devidos elementos, então vamos iterar o array list de distâncias e somar à variável soma todas as distâncias e incrementar em 1 o número de Clientes/Produtores (por cada iteração).

```java
                                    Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                    ArrayList<LinkedList<V>> paths, ArrayList<E> dists) {

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

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

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

    dists.clear();
    paths.clear();

    for (int i = 0; i < numVertices; i++) {
        paths.add(null);
        dists.add(null);
    }

    for (V vDest : g.vertices()) {
        int i = g.key(vDest);
        if (dist[i] != null) {
            LinkedList<V> shortPath = new LinkedList<>();
            getPath(g, vOrig, vDest, pathKeys, shortPath);
            paths.set(i, shortPath);
            dists.set(i, dist[i]);
        }
    }
    return true;
```

No final, é calculada a média e adicionada ao mapa de *ClienteProdutorEmpresa, Double,* em que a key é a Empresa e o value é a respetiva distância média dessa mesma Empresa a todos os Clientes/Produtores.

```
            System.out.println("\nAs n empresas mais proximas de cada cliente/produtora sao, em media:");
            // ordenar mapa por valor
            Map<ClienteProdutorEmpresa, Double> sorted = new LinkedHashMap<>();
            medias.entrySet() Set<Map<K, V>.Entry<ClienteProdutorEmpresa, Double>>
                    .stream() Stream<Map<K, V>.Entry<ClienteProdutorEmpresa, Double>>
                    .sorted(Map.Entry.comparingByValue())
                    .forEachOrdered(x -> sorted.put(x.getKey(), x.getValue()));
            // imprimir os n primeiros elementos do mapa ordenado
            int i = 0;
            for (Map.Entry<ClienteProdutorEmpresa, Double> entry : sorted.entrySet()) {
                if (i < n) {
                    entry.getKey().setHub();
                    System.out.println(entry.getKey().getDesignacao() + " -> " + entry.getValue());
                    i++;
                    }
                }
                return sorted;
            }
            return null;
        }
    }
```

Aqui é criado um novo mapa *ClienteProdutorEmpresa, Double* que vai conter os mesmos dados do mapa anterior, mas ordenados por ordem crescente de distância média. No final, apresenta no ecrã os n primeiros elementos desse mapa.

3.4. US304- Para cada cliente (particular ou empresa) determinar o hub mais próximo.

Esta US tem como objetivo encontrar o hub mais próximo de um cliente. Para isso foram concebidas duas classes: ClosestHubUI e ClosestHubController.

Neste relatório iremos apresentar os métodos principais: getAllClosestHubs() e getClosestHub().

```
public HashMap<ClienteProdutorEmpresa,ClienteProdutorEmpresa> getAllClosestHubs (){
    HashMap<ClienteProdutorEmpresa,ClienteProdutorEmpresa> result = new HashMap<>();
    for (ClienteProdutorEmpresa cpe: App.getInstance().getClienteProdutorEmpresaStore().getMapCPE().values()) {
        if (ClienteProdutorEmpresa.validateClienteID(cpe.getId())) {
            result.put(cpe,getClosestHub(cpe));
        }
    }
    return result;
}
```

O método getAllClosestHubs() percorre todos os ClienteProdutorEmpresa registados e para cada ClienteProdutorEmpresa que é um cliente, guarda esse cliente e o hub mais próximo dele num HashMap.

```java
public ClienteProdutorEmpresa getClosestHub(ClienteProdutorEmpresa cpe) {
    LinkedList<Localizacao> list = new LinkedList<>();
    Integer minlenpath = Integer.MAX_VALUE;
    ClienteProdutorEmpresa result = null;
    for (ClienteProdutorEmpresa cpe2: App.getInstance().getClienteProdutorEmpresaStore().getMapCPE().values()) {
        if (cpe2.isHub()) {
            if (cpe2.getLocalizacao().equals(cpe.getLocalizacao())) {
                return cpe2;
            }
            else{
                Integer tempLenPath = Algorithms.shortestPath(graph, cpe.getLocalizacao(), cpe2.getLocalizacao(), Integer::compare,Integer::sum,  zero: 0,list );
                if (tempLenPath < minlenpath) {
                    minlenpath = tempLenPath;
                    result = cpe2;
                }
            }
        }
    }
    return result;
}
```

O método getClosestHub() recebe um cliente e encontra o hub mais próximo.

Para este efeito itera-se sobre todos os ClienteProdutorEmpresa. Se a localização do cliente for igual ao hub, considera-se esse como o hub mais próximo. Senão calcula-se a distância para cada hub e escolhe-se o hub com a menor distância.

3.5. US305- Determinar a rede que conecte todos os clientes e produtores agrícolas com uma distância total mínima.

Esta US tem como objetivo determinar a rede que ligue todos os clientes e produtores agrícolas com a distância total mínima. Para tal efeito, a solução encontrada foi através da construção de uma Árvore de Extensão Mínima (*Minimum Spanning Tree,* MST) através do grafo original, e nesse sentido foram desenvolvidos dois métodos principais na classe *Algorithms* para a sua resolução: *kruskall()* e *prim()*;

```java
public static <V, E> Graph<V, E> kruskall(Graph<V, E> g, Comparator<E> ce) {
    if (g.isDirected())
        throw new IllegalArgumentException("Graph must be undirected");

    Graph<V, E> mst = new MatrixGraph<>( directed: false, g.numVertices());

    for (V vert : g.vertices()) {
        mst.addVertex(vert);
    }

    List<Edge<V, E>> lstEdges = new ArrayList<>(g.edges());
    rpeir
    lstEdges.sort(new Comparator<Edge<V, E>>() {
        rpeir
        @Override
        public int compare(Edge<V, E> o1, Edge<V, E> o2) { return ce.compare(o1.getWeight(), o2.getWeight()); }
    });

    for (Edge<V, E> edge : lstEdges) {
        V vOrig = edge.getVOrig();
        V vDest = edge.getVDest();
        List<V> connectedVerts = DepthFirstSearch(mst, vOrig);
        if (connectedVerts != null && !connectedVerts.contains(vDest)) {
            mst.addEdge(vOrig, vDest, edge.getWeight());
        }
    }
    return mst;
}
```

Como o próprio nome indica, este método utiliza o algoritmo de Kruskall, em que vai realizar a pesquisa no grafo considerando as arestas em ordem ascendente de custo e vai adicionando à MST as arestas que juntam 2 vértices que não estão conectados á MST.

```
public static <V, E> Graph<V, E> prim(Graph<V, E> g, Comparator<E> ce, E zero) {
    if (g.isDirected())
        throw new IllegalArgumentException("Graph must be undirected");

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

    int vOrigKey = 0;
    V vOrig = g.vertex( key: 0);
    dist[vOrigKey] = zero;

    while (vOrig != null) {
        vOrigKey = g.key(vOrig);
        visited[vOrigKey] = true;
        for (V vAdj : g.adjVertices(vOrig)) {
            Edge<V, E> edge = g.edge(vOrig, vAdj);
            E newDist = edge.getWeight();
            int vAdjKey = g.key(vAdj);
            if (!visited[vAdjKey]) {
                if (dist[vAdjKey] == null || ce.compare(dist[vAdjKey], newDist) > 0) {
                    dist[vAdjKey] = newDist;
                    pathKeys[vAdjKey] = vOrig;
                }
            }
        }
        vOrig = getVertMinDist(g, dist, visited, ce, zero);
    }

    return buildMst(g, pathKeys, dist);
}
```

Por sua vez, este método utiliza o algoritmo de Prim, em que vai fundamentalmente realizar uma pesquisa em largura, utilizando três vetores de suporte (visited[], pathKeys[] e dist[]) desde um vértice inicial, que vai atualizando os pesos dos vértices adjacentes não visitados, escolhe o vértice com menor peso ainda não visitado (*getVertMinDist()*), e repete o processo até todos os vértices serem visitados.

```
private static <V, E> V getVertMinDist(Graph<V, E> g, E[] dist, boolean[] visited, Comparator<E> ce, E zero) {
    E menorDistancia = zero;
    V vMenor = null;
    for (int i = 0; i < visited.length; i++) {
        if (!visited[i]) {
            if (dist[i] != null) {
                if (menorDistancia == zero) {
                    menorDistancia = dist[i];
                    vMenor = g.vertex(i);
                } else if (ce.compare(menorDistancia, dist[i]) > 0) {
                    menorDistancia = dist[i];
                    vMenor = g.vertex(i);
                }
            }
        }
    }
    return vMenor;
}
```

O método *buildMst()* constrói a MST através dos arrays pathKeys[] e dist[], preenchidos no método *Prim()*.

```java
private static <V, E> Graph<V, E> buildMst(Graph<V, E> g, V[] pathKeys, E[] dist) {
    Graph<V, E> mst = new MatrixGraph<>( directed: false);
    for (V vert : g.vertices()) {
        mst.addVertex(vert);
    }

    int numVertices = g.numVertices();
    for (int i = 0; i < numVertices; i++) {
        V vDest = pathKeys[i];
        if (vDest != null) {
            mst.addEdge(mst.vertex(i), vDest, dist[i]);
        }
    }

    return mst;
}
```

4. Testes Unitários

Todos os métodos que foram testados passam a todos os testes. Os testes baseiam-se em comparar uma saída conhecida após o processamento do método e para isso foram utilizadas as ferramentas disponibilizadas pelo *JUnit* para testes unitários.

De seguida são apresentados alguns exemplos de testes implementados.

Teste da classe ConnectionsController:

```java
1 usage
private final String pathCPsmall = "esinf/grafos/grafos/Small/clientes-produtores_small.cs
1 usage
private final String pathDsmall = "esinf/grafos/grafos/Small/distancias_small.csv";


Delcio Monjane
@BeforeEach
public void setUp() {
    ReadCSVController readCSVController = new ReadCSVController();
    try {
        readCSVController.readClientesProdutoresFile(new File(pathCPsmall));
    } catch (IOException e) {
        System.out.println("Erro ao ler o ficheiro de teste clientes-produtores_big.csv");
    }
    try {
        readCSVController.readDistancesFile(new File(pathDsmall));
    } catch (IOException e) {
        System.out.println("Erro ao ler o ficheiro de teste distancias_big.csv");
    }
}

Delcio Monjane
@Test
void minConnevtions() {
    int result = ctrl.minConnevtions();
    assertEquals( expected: 6,result);
}
}
```

Teste do método minConnections:

```
Delcio Monjane
@Test
public <V> void  testminConnections() {
    System.out.println("Test of min connections");

    int result = Algorithms.minConnections(completeMap, Integer::compare, Integer::sum,  zero: 0);
    assertEquals( expected: 5,result);


    result = Algorithms.minConnections(incompleteMap,Integer::compare, Integer::sum,  zero: 0);
    assertEquals( expected: -1,result);
}
```

5. Análise de complexidade

Métodos principais:

- ***readClientesProdutoresFile:***

Este método apresenta complexidade O(n) uma vez que é executado n vezes, sendo n, o número de linhas de um ficheiro.

- ***readDistancesFile:***

Este método apresenta complexidade O(n) uma vez que é executado n vezes, sendo n, o número de linhas de um ficheiro.

- ***getEmpresas:***

Este método utiliza um for each loop para iterar um mapa O(n) e adicionar alguns elementos num array list O(1), logo a sua complexidade é O(n), sendo n o número de linhas de um ficheiro.

- ***getMediaDistancia (shortestPaths):***

A complexidade do pior caso possível para o algoritmo de Dijkstra é de O(E + V * log(V)), em que E é o número de arestas e V é o número de vértices. Como um grafo conexo tem sempre mais arestas do que vértices, a complexidade também pode ser escrita como O(E * log(V)). Este algoritmo encontra-se dentro de um for each loop que itera as Empresas existentes num ficheiro O(n) logo, no total a complexidade fica

O(n^2 * log(n)). A ordenação do mapa e a adição de novos elementos acontece fora dos for each loops mas têm complexidade O(n).

- ***minConnections:***

Este método tem dois loops que iteram todos os vértices O(V^2) e procuram todos os caminhos com o allPaths, sendo o pior caso O(V!) logo o método tem complexidade O((V^2 – V) * (V!)) , (- V porque ele não procura caminhos para ele mesmo).

- ***getClosestHub (shortestPath)***

Este método tem um loop que itera todos os vértices O(V). Se um vértice for um hub, e o cliente não for um hub, é realizado o algoritmo shortestPath() O(V),o algoritmo de Dijkstra O(E + V * log(V)), getPath() O(V) e um loop que itera sobre as arestas do caminho mais curto O(E). A complexidade no pior caso é na ordem de O(E + V * log(V)).

- ***getAllClosestHubs***

Este método tem um loop que itera todos os vértices O(V). A complexidade é O(V).

- ***kruskall***

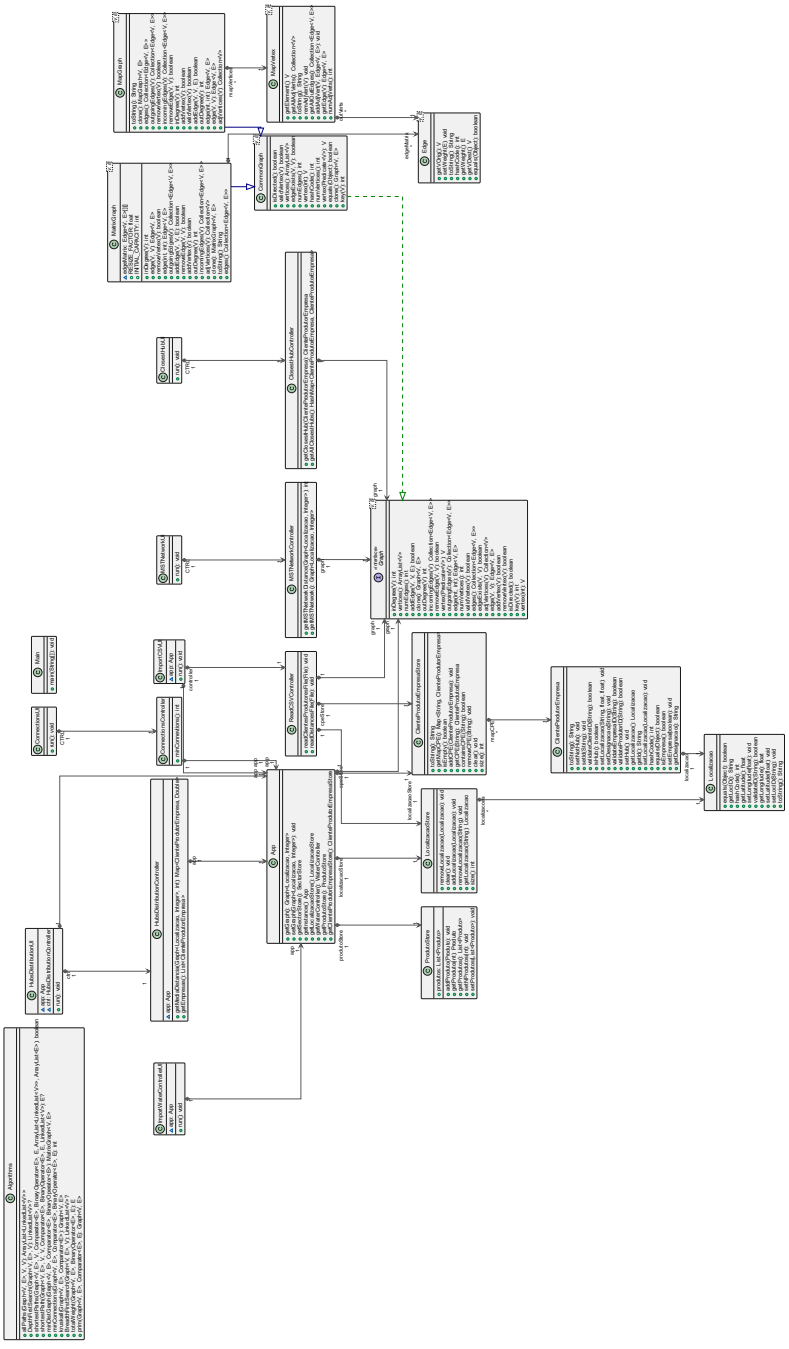
Este método tem complexidade de O(E log (V)), devido ao custo de ordenação das arestas necessária.

- ***prim***

Este método tem complexidade de O(V^2), devido ao grafo usar uma matriz ou lista de adjacências como sua representação, na pesquisa das arestas.

6. Diagrama de Classes

(o diagrama de classes também se encontra do diretório “esinf/docs” na pasta .zip do trabalho)



7. Conclusão e Melhorias Possíveis

Poderiam ter sido efetuados mais testes unitários para as funcionalidades desenvolvidas.

Divisão de tarefas:
US301: José Barbosa (1211359)

US302: Délcio Monjane (1211739)

US303: Joana Perpétuo (1211148)

US304: Inês Costa (1210814)

US305: Rodrigo Peireso (1211345)