Prepared By:
1201205 - Samuel Dias
1201367 - Carlos Santos
1201416 - Pedro Alves
1211136 - Bruna Costa
1211201 - Henrique Pinto

ESINF SPRINT 1

2022-2023

ÍNDICE

ÍNDICE DE FIGURAS

INTRODUÇÃO

O presente relatório apresenta todo o desenvolvimento do SPRINT 1 do Projeto Integrador, no terceiro semestre da Licenciatura em Engenharia Informática no Instituto Superior de Engenharia do Porto, no âmbito da disciplina Estruturas de Informação (ESINF).

O projeto realizado consiste na gestão de uma rede de distribuição de cabazes entre agricultores e clientes. Esta rede gere a distribuição dos produtos dos agricultores de modo a garantir a entrega dos cabazes aos clientes.

Será apresentado o diagrama de classes, com os principais requisitos de domínio e os seus componentes, de modo a contextualizar o projeto e o seu objetivo. Realizar-se-á a análise dos algoritmos de todas as funcionalidades implementas e da sua complexidade.

Posteriormente, irão ser apresentados possíveis melhoramentos capazes de otimizar a implementação das classes pedidas, garantindo que esta seja a mais eficiente e prática possível.

DESCRIÇÃO

No enunciado do Projeto Integrador, no âmbito da disciplina de Estruturas de Informação (ESINF), pretende-se, com recurso às classes que implementam a interface *Graph*, criar um conjunto de classes e testes que permitam gerir uma rede de distribuição de cabazes entre agricultores e clientes.

Os agricultores/produtores disponibilizam diariamente à rede de distribuição os produtos e respetivas quantidades que têm para vender e os clientes (particulares ou empresas) colocam encomendas (cabazes de produtos agrícolas) à rede de distribuição.

Se a totalidade de um produto disponibilizado por um agricultor para venda, num determinado dia, não for totalmente expedido, fica disponível nos dois dias seguintes, sendo eliminado após esses dois dias. Os cabazes são expedidos num determinado dia, quer sejam totalmente satisfeitos ou não, sendo necessário registar para cada produto a quantidade encomendada, a quantidade entregue e o produtor que forneceu. Cada produto de um cabaz é fornecido por um só produtor, mas um cabaz pode ser fornecido por vários produtores.

A rede gere a distribuição dos produtos dos agricultores de modo a satisfazer os cabazes a serem entregues em hubs para posterior levantamento pelos clientes. Um hub é localizado numa empresa e cada cliente (particular ou empresa) recolhe as suas encomendas no hub mais próximo.

DIAGRAMA DE CLASSES

Para o desenvolvimento do projeto, foi seguida a convenção da arquitetura MVC, dividida em três componentes essenciais: *Model*, *Controller* e *View*, como é possível observar pelo diagrama de classes proposto.

O *Model*, representado pelo *Domain*, é responsável por gerenciar e controlar a forma como os dados se comportam por meio das funções, lógica e regras de negócio estabelecidas.

Utiliza-se a *Store*, de modo a armazenar instâncias de *Domain* e apresentar as funções de *getters*/*setters* e filtros para obter os resultados pretendidos e o *Service*, para realizar a ligação à *Store* e garantir a implementação das funcionalidades e algoritmos.

O *Controller* é a camada responsável por intermediar as requisições enviadas pelo *View* com as respostas fornecidas pelo *Model*, processando e partilhando os dados que o usuário submeteu.

O *View*, aqui descrito como *UI*, é responsável por garantir a comunicação utilizador-máquina, apresentando as informações de forma visual ao usuário.

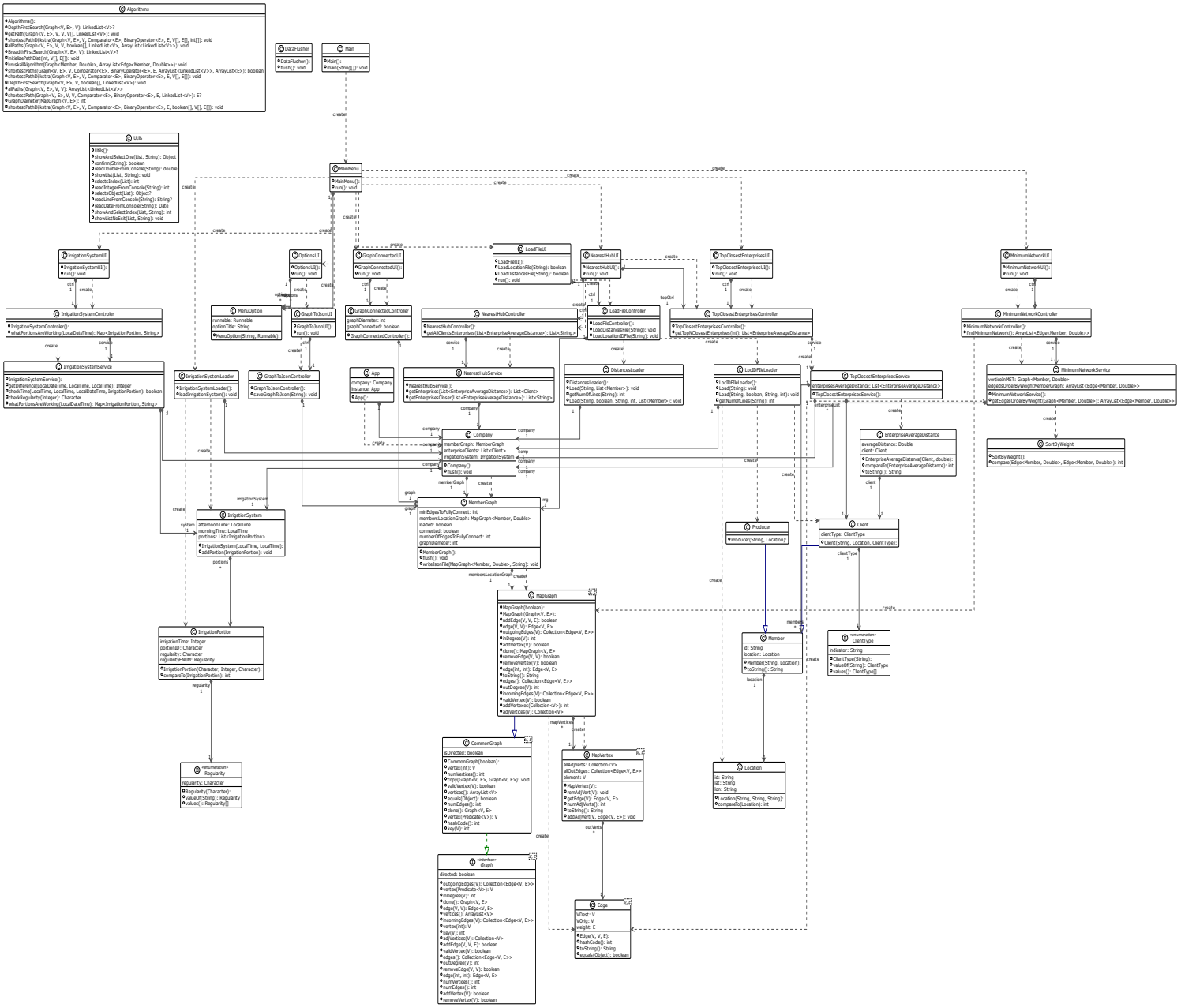

Figura 1 - Diagrama de Classes

MENU INICIAL

De modo a organizar o conjunto de classes desenvolvidas para o projeto, foi criado um menu inicial onde o utilizador consegue selecionar todas as principais funcionalidades, cuja ordem corresponde diretamente à ordem dos exercícios propostos. Todos os exercícios dependem que a primeira funcionalidade seja executada.

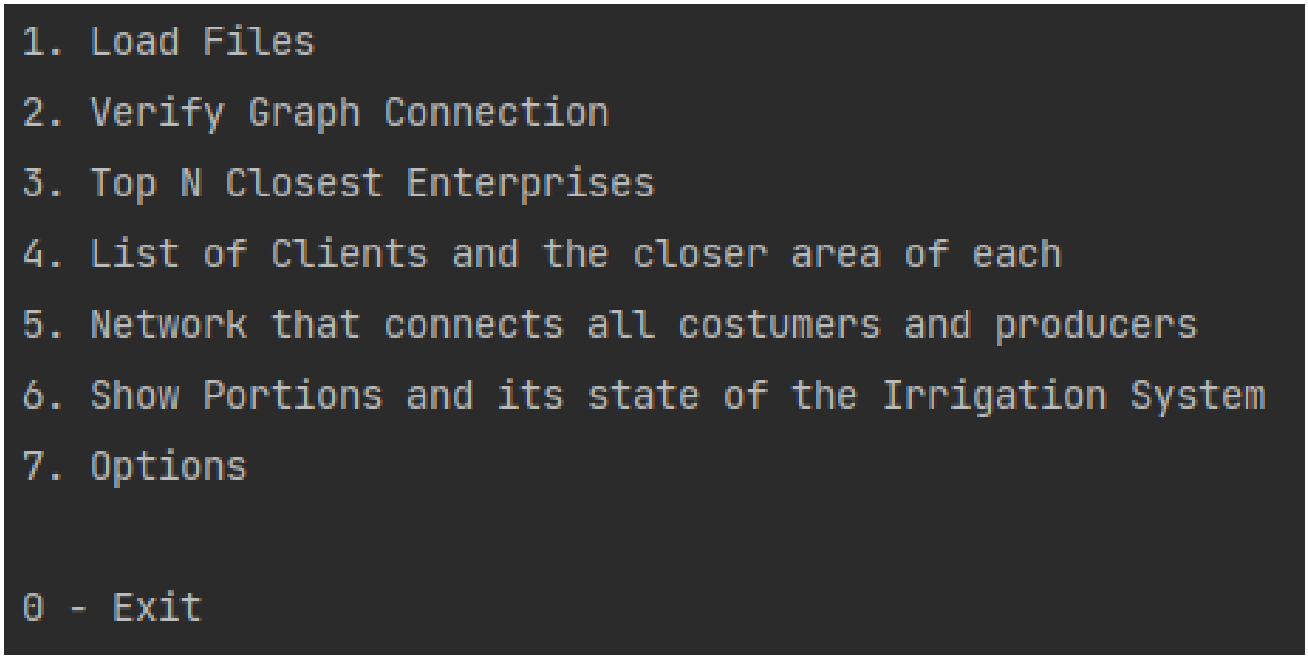

Figura 2 - Menu Inicial

1. Load Files – constrói a rede de distribuição de cabazes a partir da informação fornecida em ficheiros.
2. Verify Graph Connection – verifica se o grafo carregado é conexo e devolve o número mínimo de ligações.
3. Top N Closest Enterprises – encontra as N empresas mais próximas de todos os pontos da rede.
4. List of Clientes and the closer area of each – determina o *hub* mais próximo para cada cliente.
5. Network that connects all costumers and producers – determina a rede que conecta todos os clientes e produtores agrícolas com uma distância total mínima.
6. Show Portions and its state of the Irrigation System – esta funcionalidade não está abrangida pela disciplina de ESINF.
7. Options – permite guardar o grafo atual em um ficheiro json.
0. Exit – permite a saída do programa.

US301

DESCRIÇÃO

Construir a rede de distribuição de cabazes a partir da informação fornecida nos ficheiros. O grafo deve ser implementado usando a representação mais adequada e garantindo a manipulação indistinta dos clientes/empresas e produtores agrícolas (código cliente: C, código empresa: E, código produtor: P).

```
private static final String DEFAULT_LOCATIONS_BIG_FILE = "src/main/resources/Big/clientes-produtores_big.csv";
1 usage
private static final String DEFAULT_DISTANCES_BIG_FILE = "src/main/resources/Big/distancias_big.csv";
private static final String DEFAULT_CABAZES_BIG_FILE = "src/main/resources/Big/cabazes_big.csv";

1 usage
private static final String DEFAULT_LOCATIONS_SMALL_FILE = "src/main/resources/Small/clientes-produtores_small.csv";
1 usage
private static final String DEFAULT_DISTANCES_SMALL_FILE = "src/main/resources/Small/distancias_small.csv";
private static final String DEFAULT_CABAZES_SMALL_FILE = "src/main/resources/Small/cabazes_small.csv";
```

Figura 3 - Default Files

CLASSES DESENVOLVIDAS

Esta funcionalidade inicia-se após o utilizador selecionar a primeira opção do Menu Inicial. Na primeira fase da implementação, serão carregados dois tipos distintos de ficheiros: um que apresenta, em cada linha, duas localizações e a distância entre estas e outro que contem a localização e o código de cada membro.

De modo a garantir uma facilidade na leitura dos ficheiros, esta será automática. Ou seja, o caminho que remete à localização dos ficheiros já está presente no código e o utilizador não interage diretamente com a máquina, a não ser que pretenda utilizar um ficheiro distinto.

A classe *LoadFileController* é constituída exclusivamente por dois métodos. O primeiro, *LoadLocationIDFile*, verifica se o grafo já foi gerado. Em caso afirmativo, é invocada a função *flush*, que remove toda a informação carregada anteriormente. Logo após, é realizada, através do *load*, a ligação aos algoritmos presentes no *Service*.

O segundo método, *LoadDistancesFile*, verifica, primeiramente, se existem membros adicionados previamente. Se este valor for igual a zero, é lançada uma exceção. De seguida, assim como no método anterior, é realizada, através do *load*, a ligação aos algoritmos presentes no *Service*.

```
public class LoadFileController {

    3 usages
    MemberGraph mg = App.getInstance().getCompany().getMemberGraph();

    CarlosSantos1201367
    public LoadFileController() {}

    CarlosSantos1201367
    public void LoadLocationIDFile(String filepath) throws FileNotFoundException {
        if (mg.isLoaded()){
            mg.flush();
        }
        (new LocIDFIleLoader()).Load(filepath);
    }

    CarlosSantos1201367
    public void LoadDistancesFile(String filepath) throws FileNotFoundException, InstanceNotFoundException {
        DistancesLoader loader = new DistancesLoader();

        List<Member> members = mg.getMembersLocationGraph().vertices();
        if (members.size() == 0) {
            throw new InstanceNotFoundException("No members loaded");
        }
        loader.Load(filepath, members);
    }

}
```

Figura 4 - LoadFileController

Para carregar o primeiro ficheiro descrito anteriormente, é invocada a função *load*, na classe *LocIDFileLoader*, Através de um ciclo *while*, será verificado se o ficheiro ainda possui linhas. Em cada iteração do ciclo, a linha será dividida sempre que apresenta uma vírgula. A última coluna apresenta, no primeiro caracter, as letras ‘P’, ‘C’ ou ‘E’. Para cada linha, será verificado o tipo de membro: P para produtor, C para cliente e E para empresário e será adicionado ao vértice do gráfico.

```java
package Service;

import ...

1 usage  CarlosSantos1201367
public class LocIDFIleLoader {

    1 usage
    private static final String SEPARATOR_DEFAULT = ",";
    1 usage
    private static final boolean HAS_HEADER_DEFAULT = true;
    1 usage
    private static final int NUM_COLUMNS_DEFAULT = 4;

    1 usage
    Company comp = App.getInstance().getCompany();

    CarlosSantos1201367
    public void Load(String filepath) throws FileNotFoundException {
        Load(filepath, HAS_HEADER_DEFAULT, SEPARATOR_DEFAULT, NUM_COLUMNS_DEFAULT);
    }

    1 usage  CarlosSantos1201367
    private int getNumOfLines(String filepath) throws FileNotFoundException {
        Scanner scanner = new Scanner(new File(filepath));
        String header = scanner.nextLine();
        int lines = 0;
        while (scanner.hasNextLine()) {
            scanner.nextLine();
            lines++;
        }
        scanner.close();
        return lines;
    }

    1 usage  CarlosSantos1201367
    private void Load(String filepath, boolean hasHeaders, String separator, int numColumns) throws FileNotFoundException {
        Scanner sc = new Scanner(new File(filepath));

        if (hasHeaders) sc.nextLine();
        int i = 0;
        while(sc.hasNext()) {

            String[] line = sc.nextLine().split(separator);

            Location loc = new Location(line[0], line[1], line[2]);
            String id = line[3];
```

```
            Member m = null;
            switch (id.charAt(0)){
                case 'P':
                    m = new Producer(id, loc);
                    break;
                case 'C':
                    m = new Client(id, loc, ClientType.CLIENT);
                    break;
                case 'E':
                    m = new Client(id, loc, ClientType.ENTERPRISE);

            }
            if (m!=null){
                comp.getMemberGraph().getMembersLocationGraph().addVertex(m);
                i++;
            }
        }
        System.out.printf("Added %d vertexes of %d total \n", i, getNumOfLines(filepath));;
    }
}
```

Figura 5 - LocIDFileLoader

Para carregar o segundo ficheiro é, equivalente ao caso anterior, invocada a função *load*, na classe *DistancesLoader*. Através de um ciclo *while*, será verificado se o ficheiro ainda possui linhas. Em cada iteração do ciclo, a linha será dividida sempre que apresenta uma vírgula. Através da função *filter*, serão percorridas todas as linhas da primeira coluna, comparando os *ids*. Após encontrar uma correspondência, serão percorridas todas as linhas da segunda coluna, voltando a comparar os *ids*. No final, os dois membros e a distância entre eles serão adicionados.

```java
package Service;

import ...

1 usage  CarlosSantos1201367 +1
public class DistancesLoader {

    1 usage
    private final Company company = App.getInstance().getCompany();

    1 usage
    private static final String SEPARATOR_DEFAULT = ",";
    1 usage
    private static final boolean HAS_HEADER_DEFAULT = true;
    1 usage
    private static final int NUM_COLUMNS_DEFAULT = 3;

    CarlosSantos1201367
    public void Load(String filepath, List<Member> members) throws FileNotFoundException {
        Load(filepath, HAS_HEADER_DEFAULT, SEPARATOR_DEFAULT, NUM_COLUMNS_DEFAULT, members);
    }

    1 usage  CarlosSantos1201367
    private int getNumOfLines(String filepath) throws FileNotFoundException {
        Scanner scanner = new Scanner(new File(filepath));
        String header = scanner.nextLine();
        int lines = 0;
        while (scanner.hasNextLine()) {
            scanner.nextLine();
            lines++;
        }
        scanner.close();
        return lines;
    }

    1 usage  CarlosSantos1201367 +1
    public void Load(String filepath, boolean hasHeaders, String separator, int numColumns, List<Member> members) throws FileNotFoundException {
        Scanner sc = new Scanner(new File(filepath));
        if (hasHeaders) sc.nextLine();

        AtomicInteger i = new AtomicInteger();
        while (sc.hasNextLine()) {
            String[] line = sc.nextLine().split(",");
            String id1 = line[0];
            String id2 = line[1];
            double distance = Double.parseDouble(line[2]);
            members.stream().filter(m -> m.getLocation().getId().equals(id1)).findFirst().ifPresent(m1 -> {
                members.stream().filter(m -> m.getLocation().getId().equals(id2)).findFirst().ifPresent(m2 -> {
                    company.getMemberGraph().getMembersLocationGraph().addEdge(m1, m2, distance);
                    i.getAndIncrement();
                    //System.out.println("Added edge: " + m1.getId() + " -> " + m2.getId() + " = " + distance);
                });
            });
        }

        System.out.println("Added " + i + " of " + getNumOfLines(filepath) + " edges total");
        sc.close();
    }
}
```

Figura 6 - DistancesLoader

COMPLEXIDADE

```java
private void Load(String filepath, boolean hasHeaders, String separator, int numColumns) throws FileNotFoundException {
    Scanner sc = new Scanner(new File(filepath));

    if (hasHeaders) sc.nextLine();
    int i = 0;
    while(sc.hasNext()) {

        String[] line = sc.nextLine().split(separator);

        Location loc = new Location(line[0], line[1], line[2]);
        String id = line[3];

        Member m = null;
        switch (id.charAt(0)){
            case 'P':
                m = new Producer(id, loc);
                break;
            case 'C':
                m = new Client(id, loc, ClientType.CLIENT);
                break;
            case 'E':
                m = new Client(id, loc, ClientType.ENTERPRISE);
        }
        if (m!=null){
            comp.getMemberGraph().getMembersLocationGraph().addVertex(m);
            i++;
        }
    }
    System.out.printf("Added %d vertexes of %d total \n", i, getNumOfLines(filepath));;
}
}
```

$O(n)$

Figura 7 - Complexidade: LocIDFileLoader

```java
public void Load(String filepath, boolean hasHeaders, String separator, int numColumns, List<Member> members) throws FileNotFoundException {
    Scanner sc = new Scanner(new File(filepath));
    if (hasHeaders) sc.nextLine();

    AtomicInteger i = new AtomicInteger();
    while (sc.hasNextLine()) {
        String[] line = sc.nextLine().split( regex: ",");
        String id1 = line[0];
        String id2 = line[1];
        double distance = Double.parseDouble(line[2]);

        members.stream().filter(m -> m.getLocation().getId().equals(id1)).findFirst().ifPresent(m1 -> {
            members.stream().filter(m -> m.getLocation().getId().equals(id2)).findFirst().ifPresent(m2 -> {
                company.getMemberGraph().getMembersLocationGraph().addEdge(m1, m2, distance);
                i.getAndIncrement();
                //System.out.println("Added edge: " + m1.getId() + " -> " + m2.getId() + " = " + distance);
            });
        });
    }

    System.out.println("Added " + i + " of " + getNumOfLines(filepath) + " edges total");
    sc.close();
}
}
```

$O(E)$

$O(n^2E)$

Figura 8 - Complexidade: DistancesLoader

A complexidade da classe *LocIDFileLoader()* é $O(n)$, uma vez que tudo o que se encontra dentro do *while* tem esta complexidade. A classe *DistancesLoader()* tem de complexidade $O(n^2 \log E)$, já que a função *filter()* tem de complexidade $O(n)$. Assim sendo, a complexidade final da funcionalidade é também $O(n^2 \log E)$.

US302

DESCRIÇÃO

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.

CLASSES DESENVOLVIDAS

Esta funcionalidade inicia-se após o utilizador carregar um ficheiro através da primeira opção do Menu Inicial e, seguidamente, selecionar a opção 2.

A classe *GraphConnectedController* apenas contêm as duas funções para verificar a conexidade do grafo e o seu diâmetro, ambas presentes na classe *MemberGraph*. Ambos lançam uma exceção caso o grafo não tenha vértices ou arestas carregadas.

```
public class GraphConnectedController {

    6 usages
    MemberGraph graph = App.getInstance().getCompany().getMemberGraph();

    1 usage  CarlosSantos1201367
    public boolean isGraphConnected() throws InstanceNotFoundException {
        if (!graph.isLoaded()) {
            throw new InstanceNotFoundException("Graph doesnt have any vertices/edges loaded");
        }
        return graph.isConnected();
    }

    CarlosSantos1201367
    public int getNumberOfEdgesToFullyConnect() throws InstanceNotFoundException {
        if (!graph.isLoaded()) {
            throw new InstanceNotFoundException("Graph doesnt have any vertices/edges loaded");
        }
        return graph.getNumberOfEdgesToFullyConnect();
    }


    1 usage  CarlosSantos1201367
    public int getGraphDiameter() throws InstanceNotFoundException {
        if (!graph.isLoaded()) {
            throw new InstanceNotFoundException("Graph doesnt have any vertices/edges loaded");
        }
        return graph.getGraphDiameter();
    }

}
```

Figura 9 - GraphConnectedController

A função *isConnected()* realiza uma pesquisa em profundidade pelo grafo. Caso a lista de membros retornada tenha o mesmo tamanho do número de vértices do grafo, irá ser retornado o valor *boolean* verdadeiro ou, em caso contrário, será retornado falso.

```
public boolean isConnected() {
    LinkedList<Member> members = Algorithms.DepthFirstSearch(
            getMembersLocationGraph(),
            getMembersLocationGraph().vertices().get(0)
    );
    return (members.size() == membersLocationGraph.vertices().size());
}
```

Figura 10 – MemberGraph: isConnected()

A função *getGraphDiameter()* pesquisa por todos os caminhos mais curtos entre dois pontos do grafo de modo a obter obter o maior caminho de todos. O tamanho deste caminho escolhido representará o diâmetro do grafo.

```
public int getGraphDiameter() {
    LinkedList<Member> longestPath = null;
    for (Member m1 : membersLocationGraph.vertices()) {
        for (Member m2 : membersLocationGraph.vertices()) {
            LinkedList<Member> path = new LinkedList<>();
            Algorithms.shortestPath(membersLocationGraph, m1, m2, Double::compare, Double::sum, Double.valueOf( s: "0"), path);
            if (longestPath == null || path.size() > longestPath.size()) {
                longestPath = path;
            }
        }

    }
    return longestPath.size();
}

}
```

Figura 11 - MemberGraph: getGraphDiameter()

Por fim, a classe *GraphConnectedUI* irá disponibilizar ao utilizador a informação sobre o grafo ser ou não conexo. No primeiro caso, irá ser apresentado na interface o diâmetro do mesmo.

```java
public class GraphConnectedUI implements Runnable{

    2 usages
    GraphConnectedController ctrl = new GraphConnectedController();

    CarlosSantos1201367
    @Override
    public void run() {
        try {
            if (ctrl.isGraphConnected()){
                System.out.println("\nGraph is connected");
                System.out.println("Calculating Graph Diameter... Please wait");
                System.out.println("Graph Diameter: " + ctrl.getGraphDiameter());
                System.out.println("");
            }
            else System.out.println("\nGraph is not connected\n");


        } catch (Exception exception) {
            System.out.println("\nWARNING: " + exception.getMessage() + "\n");
        }
    }
}
```

Figura 12 - GraphConnectedUI

COMPLEXIDADE

```java
public boolean isConnected() {
    LinkedList<Member> members = Algorithms.DepthFirstSearch(
            getMembersLocationGraph(),
            getMembersLocationGraph().vertices().get(0)
    ); O(n)
    return (members.size() == membersLocationGraph.vertices().size()); O(1)
}
```

Figura 13 - Complexidade: isConnected()

Figura 14 - Complexidade: getGraphDiameter()

A complexidade da função *isConnected()* é $O(n)$. Na função *getGraphDiameter()*, é invocado o algoritmo de *Djisktra, cuja complexidade isolada é* $O(n \log n)$. No entanto, como este algoritmo é chamado no interior de dois ciclos *for*, com complexidade $O(n^2)$, a complexidade final da classe é $O(n^3 \log n)$. Assim sendo, a complexidade final da funcionalidade é também $O(n^3 \log n)$.

US303

DESCRIÇÃO

Definir os *hubs* da rede de distribuição, ou seja, encontrar as N empresas mais próximas de todos os pontos da rede (clientes e produtores agrícolas). A medida de proximidade deve ser calculada como a média do comprimento do caminho mais curto de cada empresa a todos os clientes e produtores agrícolas. (*small*, N=3)

CLASSES DESENVOLVIDAS

Esta funcionalidade inicia-se após o utilizador carregar um ficheiro através da primeira opção do Menu Inicial e, seguidamente, selecionar a opção 3. A classe *TopClosestEnterprisesUI* irá solicitar o número N de empresas a ser exibido.

A classe *TopClosestEnterprisesController* é constituída exclusivamente por um método que permite a interação com os algoritmos do *Service*, através da função *getEnterprisesAverageDistance*. Através do *Collections.sort*, a lista será ordenada por distância.

Ainda no *Controller*, será realizada uma pequena verificação que envolve o valor de N passado por parâmetro. Se este valor for igual ou inferior a zero, será lançada uma exceção. Caso este valor seja superior ao valor de empresas presentes no sistema, será também lançada uma exceção, distinta da anterior.

```
public class TopClosestEnterprisesUI implements Runnable {

    2 usages
    private TopClosestEnterprisesController ctrl;

    Henrique
    public TopClosestEnterprisesUI() { this.ctrl = new TopClosestEnterprisesController(); }

    Henrique
    @Override
    public void run() {
        try{
            Integer n = Utils.readIntegerFromConsole("Enter the number of closest enterprises to show: ");

            List<EnterpriseAverageDistance> topNClosestEnterprises = ctrl.getTopNClosestEnterprises(n);

            Utils.showListNoExit(topNClosestEnterprises, "\nTop " + n + " closest enterprises\n");
        }catch (Exception e) {
            System.out.println(e.getMessage());
        }



    }
}
```

Figura 15 - TopClosestEnterprisesUI

```java
public class TopClosestEnterprisesController {

    2 usages
    private TopClosestEnterprisesService service;

    1 usage
    public static final String ERROR_MESSAGE_MORE_ENTERPRISES
            = "The number of closest enterprises to show is greater than the number of enterprises in the system.\nTry again.\n";

    1 usage
    public static final String ERROR_MESSAGE_NEGATIVE_OR_ZERO_NUMBER
            = "The number of closest enterprises to show is negative.\nTry again.\n";

    Henrique
    public TopClosestEnterprisesController() { this.service = new TopClosestEnterprisesService(); }

    Henrique
    public List<EnterpriseAverageDistance> getTopNClosestEnterprises(int n) {

        List<EnterpriseAverageDistance> totalList = service
                .getEnterprisesAverageDistance();

        if(n > totalList.size()) {
            throw new IllegalArgumentException(ERROR_MESSAGE_MORE_ENTERPRISES);
        }else if (n <= 0){
            throw new IllegalArgumentException(ERROR_MESSAGE_NEGATIVE_OR_ZERO_NUMBER);
        }

        Collections
                .sort(totalList);

        return totalList
                .subList(0, n);
    }

}
```

Figura 16 - TopClosestEnterprisesController

No *TopClosestEnterprisesService*, em cada iteração do ciclo, serão verificados os empresários presentes na lista e criadas duas variáveis: soma e número de membros, primeiramente a zero. Aqui, será invocado o algoritmo de *Djikstra*, através do *shortestPaths*, que retornará um *ArrayList* de distâncias entre a Empresa e os Clientes. De seguida, esta lista será iterada e será calculada a distância entre a empresa e o membro, que será igualada à variável soma, e incrementado o número de membros criado anteriormente.

No final das duas iterações, será calculada a média das distâncias, através da divisão entre a soma e o número de membros e, finalmente, a lista das distâncias será organizada.

```java
public List<EnterpriseAverageDistance> getEnterprisesAverageDistance() {
    this.enterpriseList = this.company.getEnterpriseClients();

    List<EnterpriseAverageDistance> enterpriseAverageDistanceList = new ArrayList<>();

    ArrayList<LinkedList<Member>> shortPaths = new ArrayList<>();

    for(Client c : enterpriseList) {

        ArrayList<Double> distances = new ArrayList<>();
        Double sum = (double) 0;
        Double numberOfMembers = (double) 0;

        Algorithms.shortestPaths(
                this.company.getMemberGraph().getMembersLocationGraph(),
                c,
                Double::compare,
                Double::sum,
                 zero: 0.0,
                shortPaths,
                distances
        );

        for(Double value : distances) {
            if(value != 0.0) {
                sum += value;
                numberOfMembers++;
            }
        }

        if(numberOfMembers != 0) {
            double averageDistance = sum / numberOfMembers;
            enterpriseAverageDistanceList.add(new EnterpriseAverageDistance(c, averageDistance));
        }
    }

    return enterpriseAverageDistanceList;
}
```

Figura 17 - TopClosesetEnterprisesService

COMPLEXIDADE

```
public List<EnterpriseAverageDistance> getEnterprisesAverageDistance() {
    this.enterpriseList = this.company.getEnterpriseClients(); O(V)

    List<EnterpriseAverageDistance> enterpriseAverageDistanceList = new ArrayList<>(); O(1)

    ArrayList<LinkedList<Member>> shortPaths = new ArrayList<>(); O(1)

    for(Client c : enterpriseList) { O(V)

        ArrayList<Double> distances = new ArrayList<>(); O(V)
        Double sum = (double) 0; O(V)
        Double numberOfMembers = (double) 0; O(V)

        Algorithms.shortestPaths(
                this.company.getMemberGraph().getMembersLocationGraph(),
                c,
                Double::compare,
                Double::sum,
                 zero: 0.0,
                shortPaths,
                distances
        ); O(VE log V)

        for(Double value : distances) {  O(VE)
            if(value != 0.0) {
                sum += value; O(VE)
                numberOfMembers++; O(VE)
            }
        }

        if(numberOfMembers != 0) {
            double averageDistance = sum / numberOfMembers; O(V)
            enterpriseAverageDistanceList.add(new EnterpriseAverageDistance(c, averageDistance)); O(V)
        }
    }

    return enterpriseAverageDistanceList; O(1)
}
}
```

Figura 18 - Complexidade: TopClosestEnterprisesService

Figura 19 - Complexidade: TopClosestEnterprisesController

A complexidade das classes *TopClosestEnterprisesService* e *TopClosestEnterprisesController* é $O(VE \log V)$. Assim sendo, a complexidade final da funcionalidade é também $O(VE \log V)$.

US304

DESCRIÇÃO

Para cada cliente (particular ou empresa) determinar o *hub* mais próximo.

CLASSES DESENVOLVIDAS

Esta funcionalidade inicia-se após o utilizador carregar um ficheiro através da primeira opção do Menu Inicial e, seguidamente, selecionar a opção 4. A classe *NearestHubUI* meramente manda gerar a lista a partir de toda a informação que já está processada no sistema, logo, não é necessária a introdução de novos dados.

```
public class NearestHubUI implements Runnable{
    2 usages
    NearestHubController ctrl;
    1 usage
    TopClosestEnterprisesController topCtrl;
    1 usage
    Scanner sc;

    1 usage   DESKTOP-BFL3VOP\User +1
    public NearestHubUI(){
        topCtrl = new TopClosestEnterprisesController();
        ctrl = new NearestHubController();
        sc = new Scanner(System.in);
    }

    DESKTOP-BFL3VOP\User
    @Override
    public void run(){

        List<String> areaClient = ctrl.getAllClientsEnterprises();
        System.out.println("This is all the information in the system:\n Client ID -- Enterprise ID -- Distance\n");
        for(String info : areaClient){
            System.out.println(info + "\n");
        }
    }
}
```

Figura 20 - NearestHubUI

A classe *NearestHubController* é responsável por buscar a lista gerada no *NearestHubService*.

```java
public class NearestHubController {
    2 usages
    NearestHubService service;

    DESKTOP-BFL3VOP\User
    public NearestHubController(){
        service = new NearestHubService();
    }

    1201416_Pedro_Alves +1
    public List<String> getAllClientsEnterprises(List<EnterpriseAverageDistance> topNClosestEnterprises){
        return service.getEnterprisesCloser(topNClosestEnterprises);
    }
}
```

Figura 21 - NearestHubController

No *Service* serão calculados todos os valores. Inicialmente, a função irá procurar toda a informação presente na *Company*. Como são utilizados grafos para armazenar os dados de todos os membros, estes estão colocados numa lista de Membros e as empresas já são facilmente descobertas pois são instâncias do tipo Empresa.

```java
    public List<Client> getEnterpriseClients() {
        List<Member> members = memberGraph.getMembersLocationGraph().vertices();

        List<Client> enterpriseClients = new ArrayList<>();

        for (Member m : members) {
            if (m instanceof Client) {
                Client c = (Client) m;
                if (c.getClientType().equals(ClientType.ENTERPRISE)) {
                    enterpriseClients.add(c);
                }
            }
        }

        return enterpriseClients;
    }
```

Figura 22 - Company: getEnterpriseClients()

Serão definidas algumas listas, de modo a garantir o não aparecimento de um valor errado e, além disso, para garantir o armazenamento do resultado. Os valores *clientID*, *entID* e *distance* são os valores que vão ser alterados e convertidos em uma *String*, de modo a garantir que a lista seja retornada. Como o algoritmo necessita de uma *LinkedList*, é criada uma inicialmente para que seja possível realizar o cálculo.

Para que não sejam adicionadas as distâncias dos produtores às empresas, estes não entrarão na iteração do ciclo. De seguida, é confirmado se, no caso de o membro *m* ser uma empresa, que a empresa não é a mesma. Caso favorável, é confirmado se há alguma

maneira de, através de uma empresa, alcançar o membro. Se o valor for menor que o valor inicializado de distância, os valores inicializados são todos alterados para a *String* ser criada.

Cada vez que a distância temporária não for nula e for menor que a distância atribuída, são novamente trocados os valores.

O algoritmo *shortestPath* verifica se os vértices mencionados são validos e limpa a *LinkedList* enviada. Seguidamente, recolhe todos os valores e verifica se há alguma maneira desses dois pontos se juntarem através do algoritmo de *Djikstra* e, se houver, irá retornar a distância desejada entre os dois pontos.

```
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]; //default value: false
    @SuppressWarnings("unchecked")
    V[] pathKeys = (V[]) new Object [numVerts];
    @SuppressWarnings("unchecked")
    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 23 - Algorithms: shortestPath()

Para finalizar, é verificado se o membro já não está contido e se ele não é nulo. Nesse caso, é adicionado à lista de *Strings* uma nova linha e são confirmados se os valores não aparecem repetidos. O valor de distância tem sempre de ser reinicializado no final do primeiro ciclo for para que as distâncias calculadas não sejam comparadas com a distância modificada da iteração anterior. Por fim, a lista é retornada.

```java
public class NearestHubService {
    3 usages
    Company company;


    DESKTOP-BFL3VOP\User
    public NearestHubService(){
        this.company = App.getInstance().getCompany();
    }

    1201416_Pedro_Alves +1
    public List<String> getEnterprisesCloser(List<EnterpriseAverageDistance> topNClosestEnterprises) {
        List<Member> members = company.getMemberGraph().getMembersLocationGraph().vertices();
        List<Client> enterprises = getEnterprises(topNClosestEnterprises);
        List<Member> finalMembers = new ArrayList<>();
        List<String> finalList = new ArrayList<>();
        Member clientID = null;
        Client entID = null;
        double distance = Double.MAX_VALUE;
        LinkedList<Member> shortPath = new LinkedList<>();
        for (Member m : members) {
            if(!(m.getId().charAt(0)=='P')) {
                for (Client e : enterprises) {
                    if (!(m.getId().equals(e.getId()))) {
                        Double tempDistance = Algorithms.shortestPath(
                                this.company.getMemberGraph().getMembersLocationGraph(),
                                m,
                                e,
                                Double::compare,
                                Double::sum,
                                0.0,
                                shortPath);

                        if (distance > tempDistance) {
                            distance = tempDistance;
                            clientID = m;
                            entID = e;
                        }
                    }
                }
                if(!finalMembers.contains(clientID)) {
                    String enterpriseClient = clientID.getId() + " -- " + entID.getId() + " -- " + distance;
                    finalList.removeIf(enterpriseClient::equals);
                    finalList.add(enterpriseClient);
                    finalMembers.add(clientID);
```

```
                    finalList.add(enterpriseClient);
                    finalMembers.add(clientID);
                }
                distance = Double.MAX_VALUE;
            }
        }

        return finalList;
    }

    1 usage  DESKTOP-BFL3VOP\User +1
    public List<Client> getEnterprises(List<EnterpriseAverageDistance> topNClosestEnterprises){
        List<Client> enterprises = new ArrayList<>();
        for(EnterpriseAverageDistance c : topNClosestEnterprises){
            enterprises.add(c.getClient());
        }
        return enterprises;
    }

}
```

Figura 24 - NearestHubService

COMPLEXIDADE

```
public class NearestHubUI implements Runnable{
    2 usages
    NearestHubController ctrl;
    2 usages
    TopClosestEnterprisesController topCtrl;
    2 usages
    Scanner sc;

    1 usage  DESKTOP-BFL3VOP\User +1
    public NearestHubUI(){
        topCtrl = new TopClosestEnterprisesController();
        ctrl = new NearestHubController();
        sc = new Scanner(System.in);
    }

    Henrique Pinto +1
    @Override
    public void run(){

        try{

            System.out.println("Type the number of hubs you wanna define: "); O(1)
            int num = sc.nextInt(); O(1)
            List<EnterpriseAverageDistance> topNClosestEnterprises = topCtrl.getTopNClosestEnterprises(num); O(n)
            List<String> areaClient = ctrl.getAllClientsEnterprises(topNClosestEnterprises); O(n^3)

            System.out.println("This is all the information in the system:\n Client ID -- Enterprise ID -- Distance\n"); O(1)

            for(String info : areaClient) O(n)
                System.out.println(info + "\n"); O(n)


        }catch (Exception e){
            System.out.println(e.getMessage()); O(1)
        }
    }
```

Figura 25 - Complexidade: NearestHubUI

Figura 26 – Complexidade: NearestHubService

```
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)) O(1)
        return null; O(1)

    shortPath.clear();
    int numVerts = g.numVertices(); O(1)
    boolean[] visited = new boolean[numVerts]; //default value: false O(1)
    /unchecked/
    V[] pathKeys = (V[]) new Object [numVerts]; O(1)
    /unchecked/
    E[] dist = (E[]) new Object [numVerts]; O(1)
    initializePathDist(numVerts, pathKeys, dist); O(n)

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

    E lengthPath = dist[g.key(vDest)]; O(1)

    if (lengthPath != null) { O(1)
        getPath(g, vOrig, vDest, pathKeys, shortPath); O(1)
        return lengthPath; O(1)
    }
    return null;
}
```

O(1)

Figura 27 - Complexidade: shortestPath()

A complexidade da classe *NearestHubUI* é $O(n^2)$. No *NearestHubService,* é invocado o algoritmo de *Djisktra, cuja complexidade isolada é* $O(n \log n)$. No entanto, como este algoritmo é chamado no interior de dois ciclos *for*, com complexidade $O(n^2)$, a complexidade final da classe é $O(n^3 \log n)$. Assim sendo, a complexidade final da funcionalidade é também $O(n^3 \log n)$.

Apesar do alto nível de complexidade deste algoritmo, é necessário percorrer vários métodos que pesquisam dentro de listas por todos os seus elementos e comparam todas as distâncias. Deste modo, é natural que a complexidade temporal e tempo de execução sejam mais elevados.

US305

DESCRIÇÃO

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

Para ser conseguido encontrar uma rede que faça essa conexão foi utilizado o algoritmo de *Kruskal*. Resumindo, após a explicação dada anteriormente, o objetivo deste é encontrar a árvore geradora mínima do grafo criado, garantindo assim, no caso da funcionalidade proposta, o mínimo de conexões possível entre todos os clientes (particulares ou empresas) e os produtores. Após a execução do algoritmo será também apresentado o custo (baseado no *Weight* dos *Edges*) para manter esta rede.

CLASSES DESENVOLVIDAS

Esta funcionalidade inicia-se após o utilizador carregar um ficheiro através da primeira opção do Menu Inicial e, seguidamente, selecionar a opção 5.

A classe *MinimumNetworkController* é constituída por um método principal, *findMinimumNetwork*. Aqui, também é invocado o algoritmo de *Kruskal*. Uma das características principais deste algoritmo é ter a lista de *Edges* ordenada por *Weight*. Assim sendo, o *Controller*, inicialmente, chama a função *getEdgedsOrderByWeightMemberGraph*.

Posteriormente, será chamada a função *setVerticeInMST*, cuja importância será explicada abaixo. Ambas as invocações estabelecem uma ligação ao *MinimumNetworkService*.

```
public class MinimumNetworkController {
    4 usages
    private MinimumNetworkService service;

    Samuel Dias
    public MinimumNetworkController() {
        this.service = new MinimumNetworkService();
    }

    Samuel Dias
    public ArrayList<Edge<Member, Double>> findMinimumNetwork(){
        Graph<Member, Double> graphMST = new MapGraph(false);

        ArrayList<Edge<Member, Double>> edgesList = service.getEdgedsOrderByWeightMemberGraph();
        service.setVerticeInMST(graphMST);

        Algorithms.kruskallAlgorithm(graphMST, edgesList);

        ArrayList<Edge<Member, Double>> graphMSTList = service.getEdgesOrderByWeight(graphMST);

        return graphMSTList;
    }
```

Figura 28 - MinimumNetworkController

Nesta nova classe, é invocada a função genérica que ordena todos os *Edges* por peso. É importante salientar que para além de os ordenar e os colocar numa lista, são excluídas as arestas opostas, por exemplo, uma vez que o grafo é não-orientado.

Com a lista de *Edges* ordenada, o próximo passo será colocar os vértices todos no grafo *Minimum Spanning Tree*, invocando, através do Controller, a função *setVerticeInMST*.

Neste momento, o Controller terá uma lista com todos os *Edges* ordenados e o grafo MST (*Minimum Spanning Tree*) com todos os vértices adicionados. O último passo é finalizar o algoritmo de *Kruskal*, previamente criado na classe *Algorithms*. Ao percorrer a lista, irá ser realizada uma pesquisa aos vértices conectados ao vértice de origem da *Edge*.

Se, nos vértices conectados, for possível encontrar o vértice de destino, significa que não é necessário adicionarmos essa *Edge* porque a ligação já está garantida. Caso não seja encontrado irá ser adicionado à MST o vértice respetivo.

```
public class MinimumNetworkService {
    3 usages
    Company company;

    Samuel Dias
    public MinimumNetworkService() { this.company = App.getInstance().getCompany(); }

    Samuel Dias
    public ArrayList<Edge<Member, Double>> getEdgedsOrderByWeightMemberGraph(){
        return getEdgesOrderByWeight(company.getMemberGraph().getMembersLocationGraph());
    }

    1 usage  Samuel Dias
    public ArrayList<Edge<Member, Double>> getEdgesOrderByWeight(Graph<Member, Double> graph) {
        ArrayList<Edge<Member, Double>> edges = (ArrayList<Edge<Member, Double>>) graph.edges();
        ArrayList<Edge<Member, Double>> edgesOrderWeight = new ArrayList<>();

        edges.sort(new SortByWeight());

        for (Edge<Member, Double> edge : edges) {
            if(!edgesOrderWeight.contains(new Edge<>(edge.getVDest(), edge.getVOrig(), edge.getWeight()))){
                edgesOrderWeight.add(edge);
            }
        }

        return edgesOrderWeight;
    }

    Samuel Dias
    public void setVerticeInMST(Graph<Member, Double> graphMST) {
        ArrayList<Member> vertices = company.getMemberGraph().getMembersLocationGraph().vertices();

        for (Member vertice : vertices) {
            graphMST.addVertex(vertice);
        }
    }
}
```

Figura 29 - MinimumNetworkService

Para finalizar, a MST é convertida em uma lista ordenada por peso para ser mostrada ao utilizador, através do *MinimumNetworkUI*.

```java
public class MinimumNetworkUI implements Runnable {
    2 usages
    private MinimumNetworkController ctrl;

    Samuel Dias
    public MinimumNetworkUI() { this.ctrl = new MinimumNetworkController(); }

    Samuel Dias
    @Override
    public void run() {
        System.out.println();

        ArrayList<Edge<Member, Double>> graphMSTList = ctrl.findMinimumNetwork();
        int sumWeight = 0;

        if(graphMSTList.size() != 0){
            for (Edge<Member, Double> edge : graphMSTList) {
                System.out.println(edge.getVOrig().getId() + " <--> " + edge.getVDest().getId() + " : " + edge.getWeight() + "m");
                sumWeight += edge.getWeight();
            }

            System.out.println("\nMinimum Spanning Tree cost - " + sumWeight + "m\n");
        } else{
            System.out.println("\nImport files first!\n");
        }
    }
}
```

Figura 30 - MinimumNetworkUI

COMPLEXIDADE

```
public class MinimumNetworkUI implements Runnable {
    2 usages
    private MinimumNetworkController ctrl;

    1 usage  Samuel Dias
    public MinimumNetworkUI() {
        this.ctrl = new MinimumNetworkController();
    }

    Samuel Dias
    @Override
    public void run() {
        System.out.println();                                                          O(1)

        ArrayList<Edge<Member, Double>> graphMSTList = ctrl.findMinimumNetwork();      O(E log V)
        int sumWeight = 0;                                                             O(1)

        if(graphMSTList.size() != 0){                                                  O(1)
            for (Edge<Member, Double> edge : graphMSTList) {                           O(n)
                System.out.println(edge.getVOrig().getId() + " <--> " + edge.getVDest().getId() + " : " + edge.getWeight() + "m");   O(n)
                sumWeight += edge.getWeight();                                         O(n)
            }

            System.out.println("\nMinimum Spanning Tree cost - " + sumWeight + "m\n");   O(1)
        } else{
            System.out.println("\nImport files first!\n");                             O(1)
        }
    }
```

Figura 31 - Complexidade: MinimumNetworkUI

```
public class MinimumNetworkController {
    4 usages
    private MinimumNetworkService service;

    1 usage  Samuel Dias
    public MinimumNetworkController() { this.service = new MinimumNetworkService(); }

    1 usage  Samuel Dias
    public ArrayList<Edge<Member, Double>> findMinimumNetwork(){
        Graph<Member, Double> graphMST = new MapGraph( directed: false);   O(1)

        ArrayList<Edge<Member, Double>> edgesList = service.getEdgedsOrderByWeightMemberGraph();   O(n log n)
        service.setVerticeInMST(graphMST);                                 O(n)

        Algorithms.kruskallAlgorithm(graphMST, edgesList);                 O(E log V)

        ArrayList<Edge<Member, Double>> graphMSTList = service.getEdgesOrderByWeight(graphMST);    O(n log n)

        return graphMSTList;                                               O(1)
    }
}
```

Figura 32 - Complexidade: MinimumNetworkController

```
public class MinimumNetworkService {
    3 usages
    Company company;

    1 usage  Samuel Dias
    public MinimumNetworkService() {
        this.company = App.getInstance().getCompany();
    }

    1 usage  Samuel Dias
    public ArrayList<Edge<Member, Double>> getEdgedsOrderByWeightMemberGraph(){
        return getEdgesOrderByWeight(company.getMemberGraph().getMembersLocationGraph()); O(n log n)
    }

    2 usages  Samuel Dias
    public ArrayList<Edge<Member, Double>> getEdgesOrderByWeight(Graph<Member, Double> graph) {
        ArrayList<Edge<Member, Double>> edges = (ArrayList<Edge<Member, Double>>) graph.edges(); O(1)
        ArrayList<Edge<Member, Double>> edgesOrderWeight = new ArrayList<>(); O(1)

        edges.sort(new SortByWeight()); O(n log n)

        for (Edge<Member, Double> edge : edges) { O(n)
            if(!edgesOrderWeight.contains(new Edge<>(edge.getVDest(), edge.getVOrig(), edge.getWeight()))){ O(n)
                edgesOrderWeight.add(edge); O(n)
            }
        }

        return edgesOrderWeight; O(1)
    }
```

```
    public void setVerticeInMST(Graph<Member, Double> graphMST) {
        ArrayList<Member> vertices = company.getMemberGraph().getMembersLocationGraph().vertices(); O(1)

        for (Member vertice : vertices) { O(n)
            graphMST.addVertex(vertice); O(n)
        }
    }
}
```

Figura 33 – Complexidade: MinimumNetworkService

```
public static void kruskallAlgorithm(Graph<Member, Double> graphMST, ArrayList<Edge<Member, Double>> edgesList){
    for (Edge<Member, Double> edge: edgesList) { O(n)
        LinkedList<Member> connectedVets = Algorithms.DepthFirstSearch(graphMST, edge.getVOrig()); O(E log V)

        if(!connectedVets.contains(edge.getVDest())){ O(n)
            graphMST.addEdge(edge.getVOrig(), edge.getVDest(), edge.getWeight()); O(n)
        }
    }
}
```

Figura 34 - Complexidade: kruskallAlgorithm()

As complexidades das classes *MinimumNetworkUI*, *MinimumNetworkController* e do algoritmo de *Kruskal* utilizado na resolução da funcionalidade são iguais, tendo de complexidade $O(E \log V)$, onde *E* representa os *Edges* e *V* representa os vértices. A

complexidade da classe *MinimumNetworkService* é $O(n \log n)$. Assim sendo, no final, a complexidade total é $O(E \log V)$.

MELHORIAS

Após a análise de todo o projeto e da complexidade de todos os algoritmos das funcionalidades implementadas, consegue-se chegar à conclusão de que existem alguns melhoramentos possíveis para aprimorar o trabalho realizado.

O algoritmo desenvolvido invoca o método *DepthFirstSearch* para verificar até onde um determinado vértice consegue chegar, sabendo assim quais são as conexões que existem até ao momento para aquele determinado vértice. De seguida, é verificado se o destino daquele vértice está contido já nas conexões. Quanto maior for o grafo a ser utilizado, mais lenta é a pesquisa.

De tal modo, poderiam ser exploradas outras técnicas que facilitariam saber se já existe forma de chegar ao vértice pretendido. Uma das soluções seria, por exemplo, a criação de um *HashMap* que identifica os parentes de cada um dos vértices. Sempre que for adicionado um vértice à MST, todos os parentes desse vértice serão atualizados. Assim, a verificação se eles já se encontram conectados seria facilitada e mais otimizada.