Estruturas de informação 2022/2023

4 dezembro

Instituto Superior de Engenharia do Porto

Pedro Teixeira, 1211184
JoãoFernandes,1211681
David Mendonça,1211572
Afonso Machado, 1190326

isep
instituto superior de engenharia do porto

Índice

Conteúdo

Introdução

Este relatório tem o propósito de demostrar os conteúdos que foram abordados durante a realização do primeiro SPRINT do trabalho do Projeto Integrador do 3º semestre da LEI.

Relativamente à cadeira de ESINF este SPRINT tinha vários objetivos, nomeadamente todas as User Stories que envolvessem o desenvolvimento de funcionalidades que usem a linguagem JAVA e que permitam gerir a informação sobre vários clientes e produtores numa rede de produção biológica e informação relativa a equipamentos de irrigação. A informação encontra-se em arquivos de texto com formato CSV.

As funcionalidades incluem o carregamento de um ficheiro contendo todos os clientes e produtores e as respetivas localizações e ainda outro relativamente a um contolador de rega.

Foram implementadas Graphs e MapGraphs que foram desenvolvidos durante as aulas pratico-laboratoriais tendo utilizado também os algoritmos disponibilizados nas mesmas.

Para cada uma dessas funcionalidades vamos demonstrar o respetivo diagramade classes, algoritmo utilizados, análise de complexidade e possíveis melhorias.

US 301

Enunciado

- US301 - 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).

Para o propósito da leitura de ficheiro e criação de grafo necessitámos de ler 2 ficheiros fornecidos, nomeadamente o ficheiro Produtores/clientes e que irão constituir os nossos vértices e o ficheiro distâncias que é referente às arestas do grafo. Para cada um inicializámos um Scanner que divide cada linha do ficheiro nos seus respetivos blocos de informação, delimitados por vírgula, ou seja, sendo um ficheiro CSV, divide cada linha em colunas.

```java
public static void readProducerCSV(File fileVertexes, File fileEdges) {

    System.out.println("Creating graph");
    String buffer;
    int counter;

    try {

        Scanner scanner = new Scanner(fileVertexes);
        scanner.nextLine(); //skip first line of file
        counter = 2;

        while (scanner.hasNextLine()) {

            buffer = scanner.nextLine();
            String[] arrBuffer = buffer.split( regex: ",");

            if (arrBuffer.length == 4) {

                ClientsProducers clp = new ClientsProducers(arrBuffer[0], Float.parseFloat(arrBuffer[1]), Float.parseFloat(arrBuffer[2]), arrBuffer[3]);
                App.getInstance().getCompany().getClientsProducersGraph().addVertex(clp);
                //System.out.println(clp);
                counter++;
            } else {
                System.out.printf("V - Line number: %d isn't valid.\n", counter);
                counter++;
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Graph Vertexes file not found");
    }
```

Figura 1 - Código leitura de vértices

	A	B	C	D	E
1	Loc id	lat	lng	Clientes-Produtores	
2	CT1	40.6389	-8.6553	C1	

Figura 2 - Linha exemplo vértice

Após a criação dos vértices no grafo, repetimos o processo para as arestas lendo desta vez o ficheiro das distâncias.

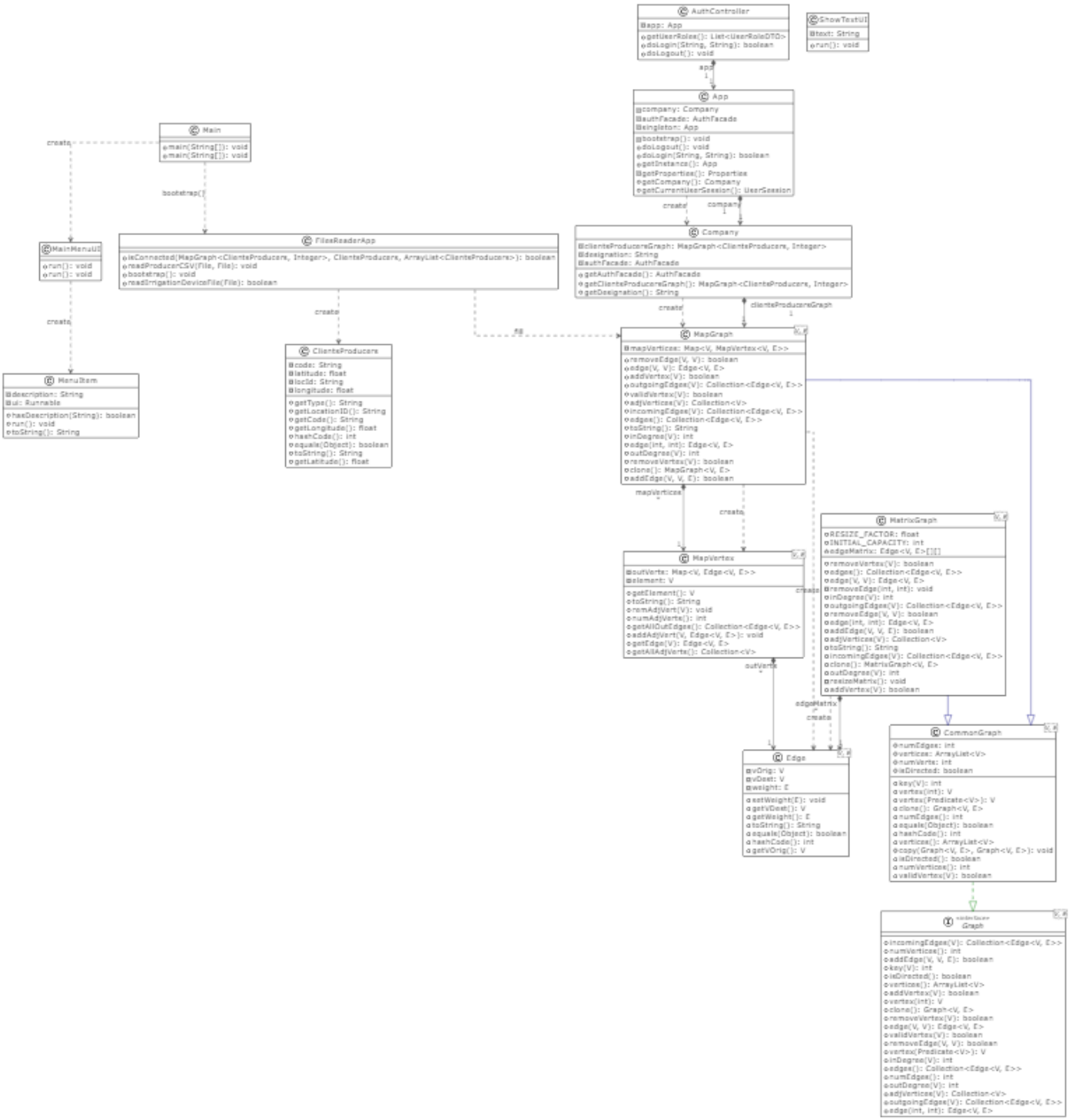
```
try {

    Scanner scanner2 = new Scanner(fileEdges);
    scanner2.nextLine(); //skip first line of file
    counter = 2;

    while (scanner2.hasNextLine()) {

        buffer = scanner2.nextLine();
        String[] arrBuffer = buffer.split( regex: ",");         //Loc id 1, Loc id 2, length (m)

        if (arrBuffer.length == 3) {

            Iterator<ClientsProducers> cp = App.getInstance().getCompany().getClientsProducersGraph().vertices().iterator();
            boolean exist = false;
            boolean exist2 = false;
            ClientsProducers aux = null;
            ClientsProducers aux2 = null;

            while (cp.hasNext()) {
                ClientsProducers c = cp.next();
                if (c.getLocationID().equals(arrBuffer[0].trim())) {
                    exist = true;
                    aux = new ClientsProducers(c);
                }
                if (c.getLocationID().equals(arrBuffer[1].trim())) {
                    exist2 = true;
                    aux2 = new ClientsProducers(c);
                }
            }

            if (exist && exist2) {
                App.getInstance().getCompany().getClientsProducersGraph().addEdge(aux, aux2, Integer.parseInt(arrBuffer[2].trim()));
            } else {
                System.out.println("\nOne of the locations didn't exist\n");
            }

            counter++;
        } else {
            System.out.printf("E - Line number: %d isn't valid.\n", counter);
            counter++;
```

Figura 3 - Código leitura de arestas

	A	B	C	D
1	Loc id 1	Loc id 2	length (m)	
2	CT10	CT13	63448	

Figura 4 - Linha exemplo aresta

Diagrama de classes



O resultado dos 2 excertos de código apresentados é um MapGraph conexo contendo toda a informação apresentada que é registado na Company a qual por si é acessível a partir de uma getInstance() da App.

Por fim, de modo a cumprir o requisito de manipulação indistinta, a classe ClientProducers, que armazena a informação relativa aos vértices do grafo, contém um método getType() que permite identificar os diferentes tipos de vértices.

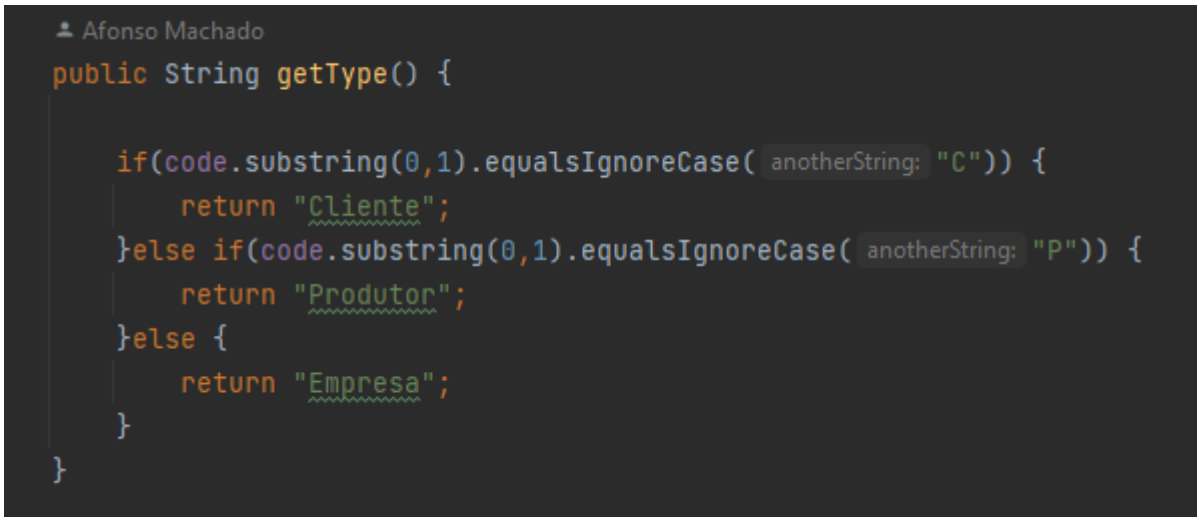


Figura 5 - Método getType()

A decisão da utilização do método em vez de 3 classes diferentes para representar cada tipo de vértice deve-se ao facto de que todas iriam armazenar informação idêntica pelo que seria ineficaz a divisão em várias classes.

A complexidade do método de leitura do ficheiro é O(V+E) em que V é o número de vértices e N o número de arestas.

Melhorias possíveis

Relativamente a melhorias, o método de leitura do ficheiro apenas verifica se cada linha fornecida contém 4 argumentos que irão ser armazenados na instância da classe ClientProducers mas não verifica o conteúdo em si.

No estado atual do código um cliente poderia ser registado com um código começado pela letra A, o que não cumpriria com os requisitos da funcionalidade.

US 302

Enunciado

- 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.

Aquando da leitura do ficheiro, o método ReadProducersCSV() corre um método isConnected() que tem por objetivo verificar se o grafo é conexo. Um grafo conexo descreve um grafo em que de qualquer um vértice é possível chegar a outro, isto é, não existem vértices isolados do resto.

```
3 usages   Afonso Machado *
public static boolean isConnected(MapGraph<ClientsProducers, Integer> cpgraph, ClientsProducers cpOrig, ArrayList<ClientsProducers> cp) {

    LinkedList<ClientsProducers> dfsResults = Algorithms.DepthFirstSearch(cpgraph, cpOrig);
    boolean connected = true;

    int i;
    for (i = 0; i < cp.size(); i++) {
        if (!dfsResults.contains(cp.get(i))) {
            //System.out.println("not connected");
            connected = false;
        }
        //System.out.println("seeing if its connected");
    }

    return connected;
}
```

Figura 6 - Método isConnected()

Para tal, o método realiza um DFS, que retorna uma lista contendo todos os vértices visitados partindo de um vértice origem e compara-a com a lista de vértices do grafo. Caso um vértice presente na lista do grafo não esteja presente no resultado do DFS, isto significa que aquele vértice não é acessível por um caminho logo o grafo não é conexo.

Por outro lado, a User Story pretende também que seja possível encontrar o menor caminho entre 2 vértices. Para tal, criámos uma classe UI que apresenta ao utilizador a lista de códigos de vértices presentes no grafo, pede-lhe para selecionar origem e destino e de seguida apresenta o menor caminho.

```
Main Menu

1. Get shortest path between Clients
2. Get smallest fully connected network
3. Define the hubs of the distribution network
4. Get closest hub
5. Register an irrigation device
6. Check current irrigation

0 - Cancel

Type your option:
1
[C125, C124, C120, P27, C55, C80, C110, C173, C131, C46, C162, P12, P39, C151, C43, C70, C52, C4, C159, P25, C115, C105, C98, C15, C20, C134, C84, P45, C167, C89, P42, P52, P15, C2, P16, C108, C137, P29, C116, C67, P8, C63, C95,

Please Select Origin Vertex from list!

c125
Location ID : CT43    Latitude : 39.1167    Longitude : -7.2833    Code : C125

Please Select Destination Vertex from list!

p25
Location ID : CT287    Latitude : 41.4878    Longitude : -7.9613    Code : P25

Smallest path from Origin to Destination is:[C125, E41, C44, C117, E72, E29, C106, C127, C22, P17, C79, P51, E9, E73, C149, C78, P32, P25] with size 18 and distance in meters 331971.
```

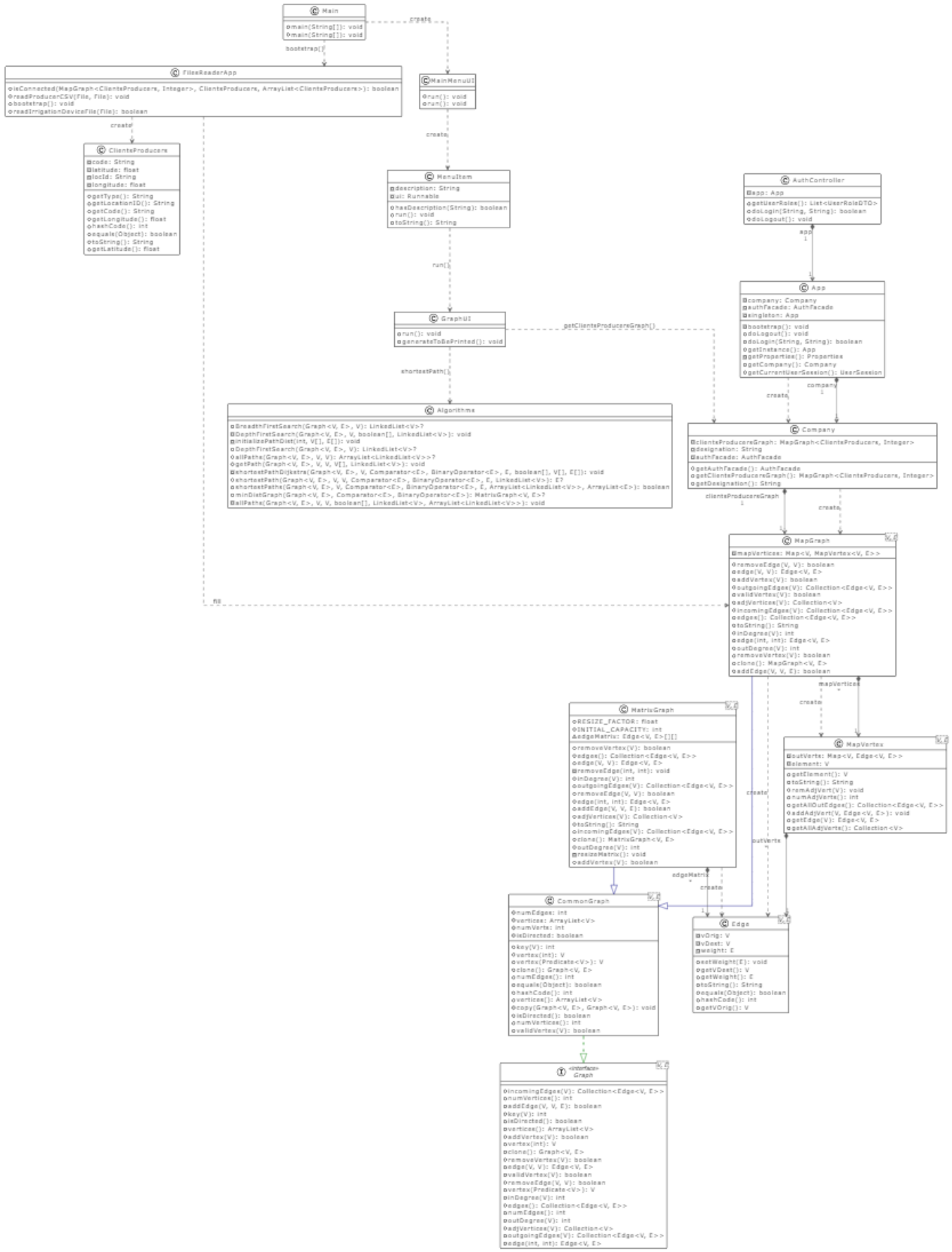
Figura 7 - Console log shortestPath

```java
String og = Utils.readLineFromConsole( prompt: "Please Select Origin Vertex from list!\n");

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

    if (og.equalsIgnoreCase(cp.get(i).getCode())) {
        clp1 = cp.get(i);
    }
}

System.out.println(clp1);

String dest = Utils.readLineFromConsole( prompt: "Please Select Destination Vertex from list!\n");

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

    if (dest.equalsIgnoreCase(cp.get(i).getCode())) {
        clp2 = cp.get(i);
    }
}

System.out.println(clp2);

Integer sPathResults = Algorithms.shortestPath(clpGraph, clp1, clp2, Integer::compare, Integer::sum,  zero: 0, path);

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

    codePath.add(path.get(i).getCode());

}

System.out.printf("Smallest path from Origin to Destination is:" + codePath + " with size %d and distance in meters %d.\n", codePath.size(), sPathResults);

toBePrinted.clear();
path.clear();
codePath.clear();
```

Figura 8 - Código shortestPathI

Tendo em conta que utiliza o algoritmo fornecido shortestPath e que este por si utiliza o algoritmo de Djikstra a complexidade desta US é $O(V^2)$ em que V é o número de Vértices.

Diagrama de classes



Melhorias possíveis

Nesta User Story não encontrámos melhorias possíveis evidentes visto ser bastante direta nas funcionalidades requerida e também fácil de implementar.

US 303

Enunciado

- 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).

De modo a cumprir as funcionalidades pedidas percorremos todos os vértices de empresas, isto é, não consideramos clientes e produtores e para cada uma obtemos a distância média para todos os outros pontos.

Após termos a distância média de cada empresa para todos os outros pontos, devolvemos as N empresas mais próximas de todos os outros pontos consoante especificado pelo utilizador.

```
public static ArrayList<Path> getCloserPoints(Graph<ClientsProducers, Integer> graph) {

    ArrayList<ClientsProducers> companies = new ArrayList<>();
    ArrayList<ClientsProducers> vertices = graph.vertices();

    for (ClientsProducers vertex : vertices) {
        if (vertex.getType().equalsIgnoreCase( anotherString: "Empresa")) {
            companies.add(vertex);
        }
    }


    ArrayList<LinkedList<ClientsProducers>> paths = new ArrayList<>();
    ArrayList<Integer> dists = new ArrayList<>();
    ArrayList<Path> pathArrayList = new ArrayList<>();

    for (ClientsProducers company : companies) {
        paths.clear();
        dists.clear();
        Algorithms.shortestPaths(graph, company, Integer::compare, Integer::sum, zero: 0, paths, dists);
        pathArrayList.add(new Path(company, paths, dists));
    }

    return pathArrayList;

}
```

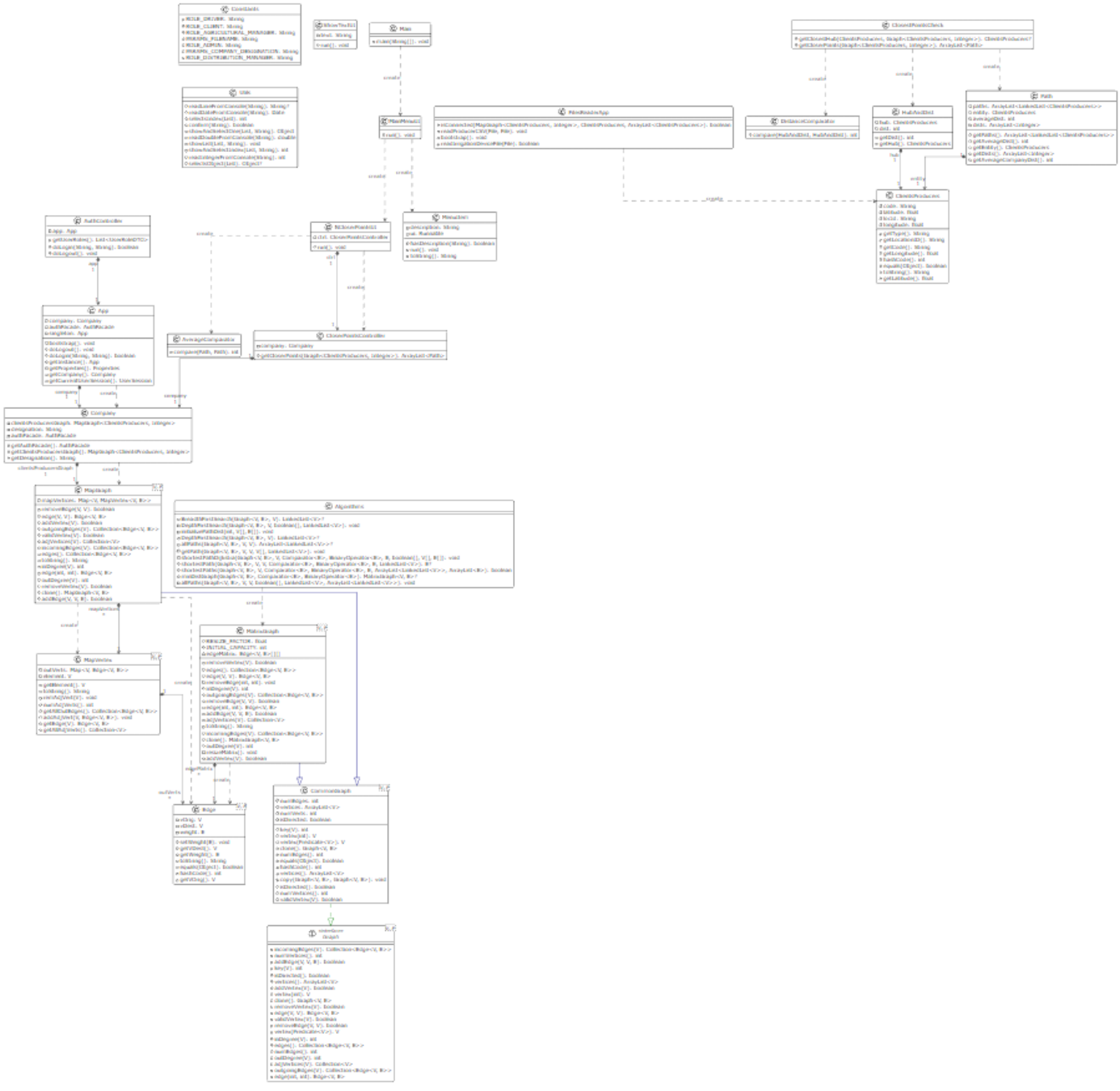
Figura 9 - Cálculo da distância média de cada empresa

```java
@Override
public void run() {

    boolean exit = false;
    int n = 0;
    Scanner scanner = new Scanner(System.in);
    do {
        try {
            System.out.println("\nTop N Closer Points\nWrite the N :");
            n = scanner.nextInt();
            if (n > 0) {
                exit = true;
            }
        } catch (NumberFormatException e) {
            System.out.println("Invalid Input");
        }
    } while (!exit);

    ArrayList<Path> closerPoints = ctrl.getCloserPoints(App.getInstance().getCompany().getClientsProducersGraph());
    Set<Path> closerPointsSet = new TreeSet<>(new AverageComparator());

    closerPointsSet.addAll(closerPoints);

    Iterator<Path> iterator = closerPointsSet.iterator();

    if (closerPointsSet.size()>=n) {
        System.out.println("\nTop " + n + " Closer Points :\n");
        for (int i = 0; i < n; i++) {
            Path path = iterator.next();
            System.out.println(i + 1 + ". " + path.getEntity() + "    Average :  " + path.getAverageDist() + " meters");
        }
    }else {
        System.out.println("\nTop " + closerPointsSet.size() + " Closer Points :\n");
        for (int i = 0; i < closerPoints.size(); i++) {
            Path path = iterator.next();
            System.out.println(i + 1 + ". " + path.getEntity() + "    Average :  " + path.getAverageDist() + " meters");
        }
    }
}
```

Figura 10 - UI

A User Story tem complexidade O(E2) em que E é o número de empresas presentes. A complexidade é O(E2) e não O(E) visto que o algoritmo shortestPath é utilizado sendo este a parte mais complexa da US.

Diagrama de Classes



Melhorias possíveis

Nesta funcionalidade, as principais melhorias possíveis seriam a alteração de algumas estruturas de dados para melhor eficiência do programa. Devido a não podermos fugir da complexidade que os algoritmos relacionados ao grafo suportam, poderíamos alterar/evitar estruturas de dados que aumentem ainda mais a complexidade.

US 304

Enunciado

- US304 – Para cada cliente (particular ou empresa) definir o hub mais próximo.

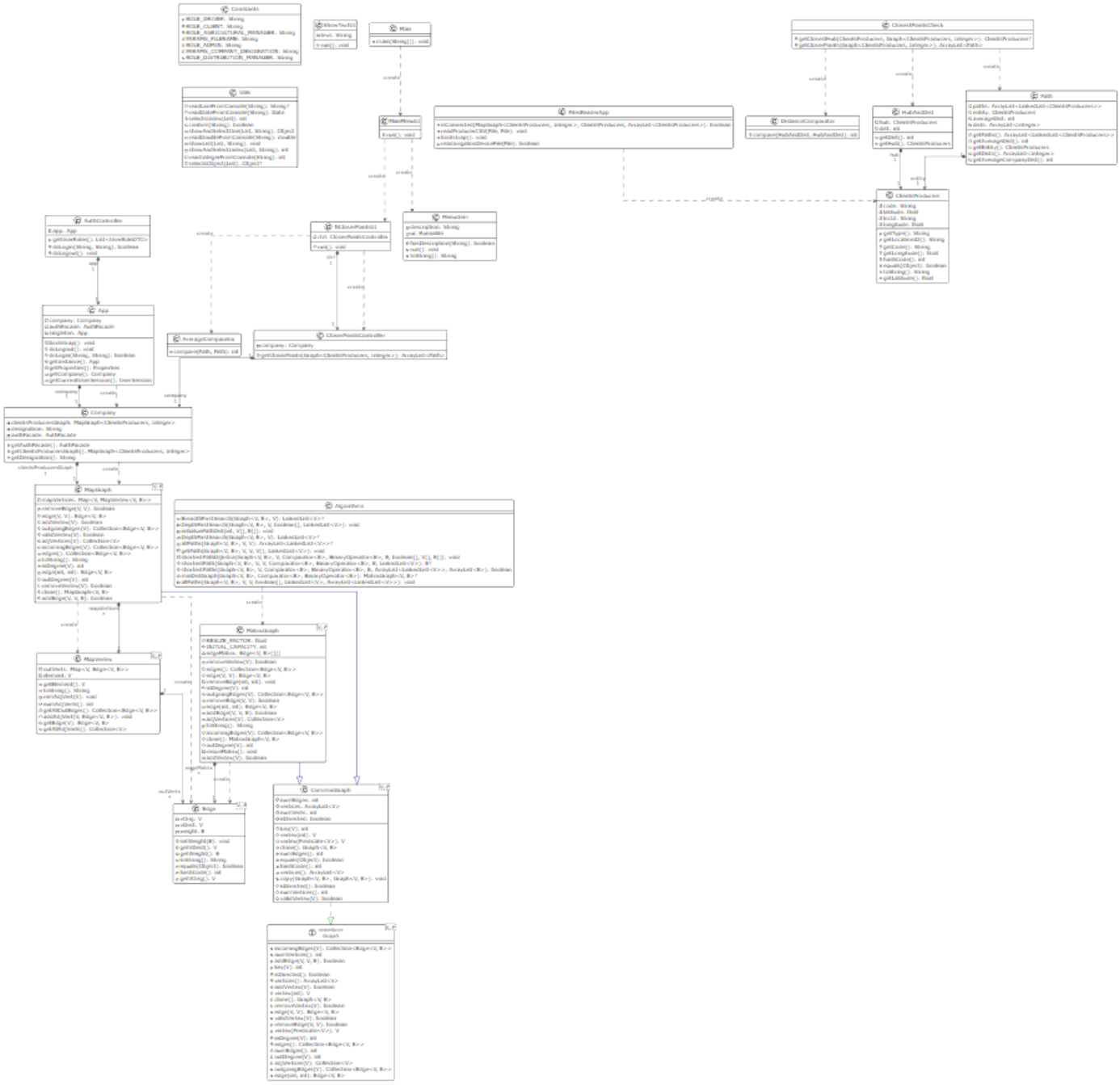
Partindo do trabalho feito na User Story anterior que nos permitiu identificar os hubs presentes na rede, precisámos apenas de fazer o Shortest Path de cada cliente para todos os Hubs possíveis.

```
2 usages  windows +1
public static ClientsProducers getClosestHub(ClientsProducers cp, Graph<ClientsProducers, Integer> graph) {

    ArrayList<LinkedList<ClientsProducers>> paths = new ArrayList<>();
    ArrayList<Integer> dists = new ArrayList<>();

    Algorithms.shortestPaths(graph, cp, Integer::compare, Integer::sum, zero: 0, paths, dists);
    Set<HubAndDist> topHubs = new TreeSet<>(new DistanceComparator());

    for (int i = 0; i < paths.size(); i++) {
        if (paths.get(i).getLast().getType().equalsIgnoreCase( anotherString: "Empresa")) {
            topHubs.add(new HubAndDist(paths.get(i).getLast(), dists.get(i)));
        }
    }
    if (topHubs.isEmpty()){
        return null;
    }
    return topHubs.iterator().next().getHub();
}
}
```

Figura 11 - Código US404

Mais uma vez, tendo em conta que o excerto de código utiliza shortestPaths podemos concluir que a sua complexidade é $O(V^2)$.

Diagrama de Classes



Melhorias possíveis

Tal como na funcionalidade anterior, não há maneira de fugir à complexidade da classe Algorithms que desenvolvemos nas aulas práticas, neste caso o shortestPath.
As principais melhorias possíveis seriam na alteração de algumas estruturas de dados para melhor eficiência do programa.

US 305

Enunciado

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

A rede que conecte todos os clientes e produtores agrícolas com uma distância total mínima é a rede que, tendo todos os vértices exceto empresa, tem o menor peso total de todas as suas arestas.

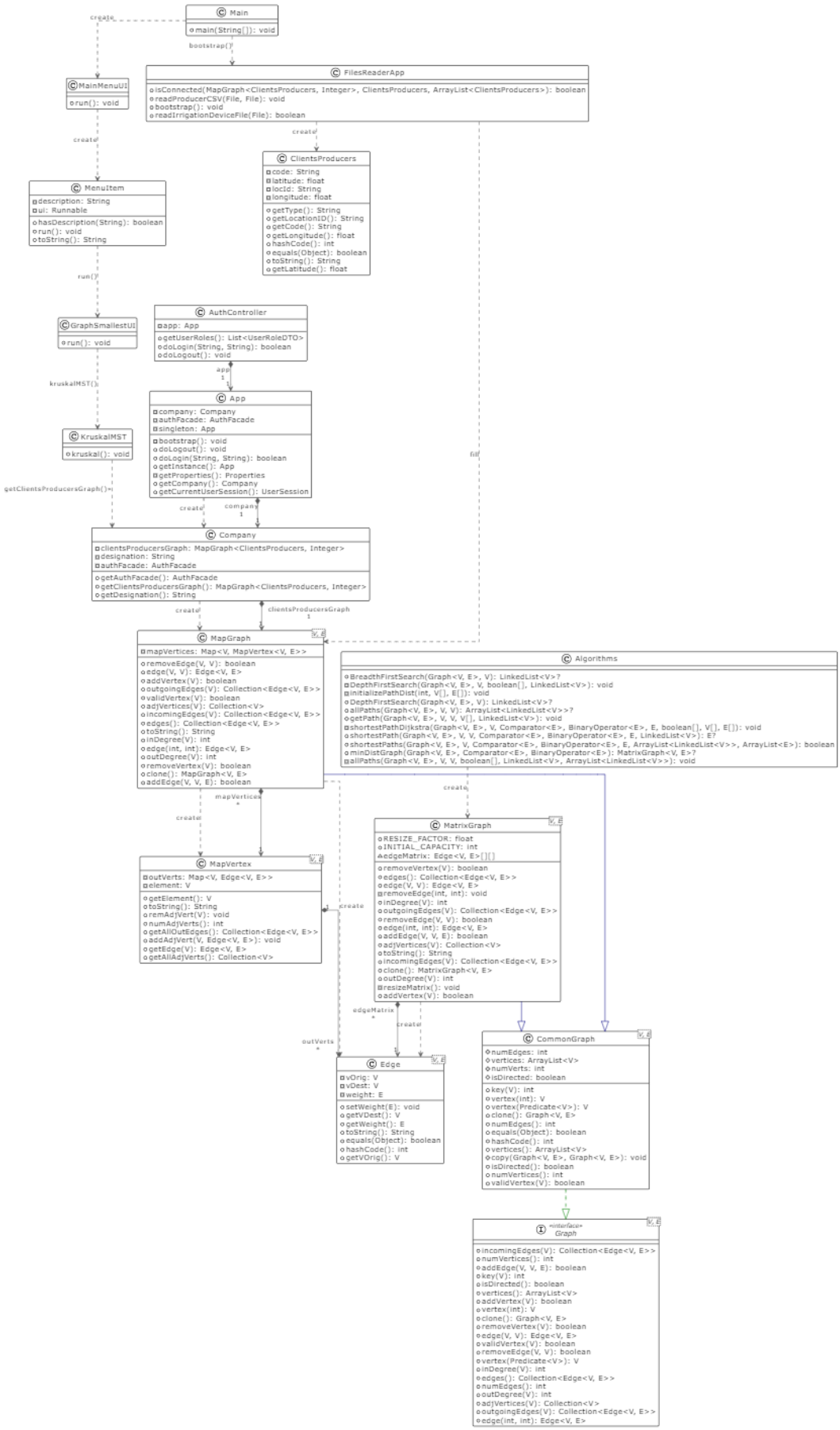
Para tal, queremos uma Minimum Spanning Tree do nosso grafo, não incluindo os vértices empresas. Sendo um grafo conexo e não direcionado poderemos utilizar o algoritmo de Kruskal ou o de Prim.

Visto já termos familiaridade com o algoritmo de Kruskal, optámos por usar este.

O código segue a estrutura fornecida nos slides de ESINF, mas visto ser demasiado vasto decidimos não incluir imagens neste relatório de modo a manter uma estrutura agradável à leitura.

A complexidade do algoritmo de Kruskal é de O(E log V) em que E são as arestas e V são os vértices.

Diagrama de Classes



Melhorias possíveis

Após mais pesquisa acabámos por concluir que em ficheiros de maior dimensão e usando a estrutura de dados Grafo o mais adequado seria o algoritmo de Prim, mas como explicado antes optámos pelo algoritmo de Kruskal visto estarmos mais familiarizados com o seu comportamento e os dados com os quais estamos a trabalhar terem reduzida dimensão pelo que as diferenças de desempenho não são detetáveis.