Projeto Integrador

Gabriel Gonçalves - 1191296

Tiago Leite - 1191369

João Durães - 1211314

Francisco Bogalho - 1211304

António Bernardo - 1210805

Introdução

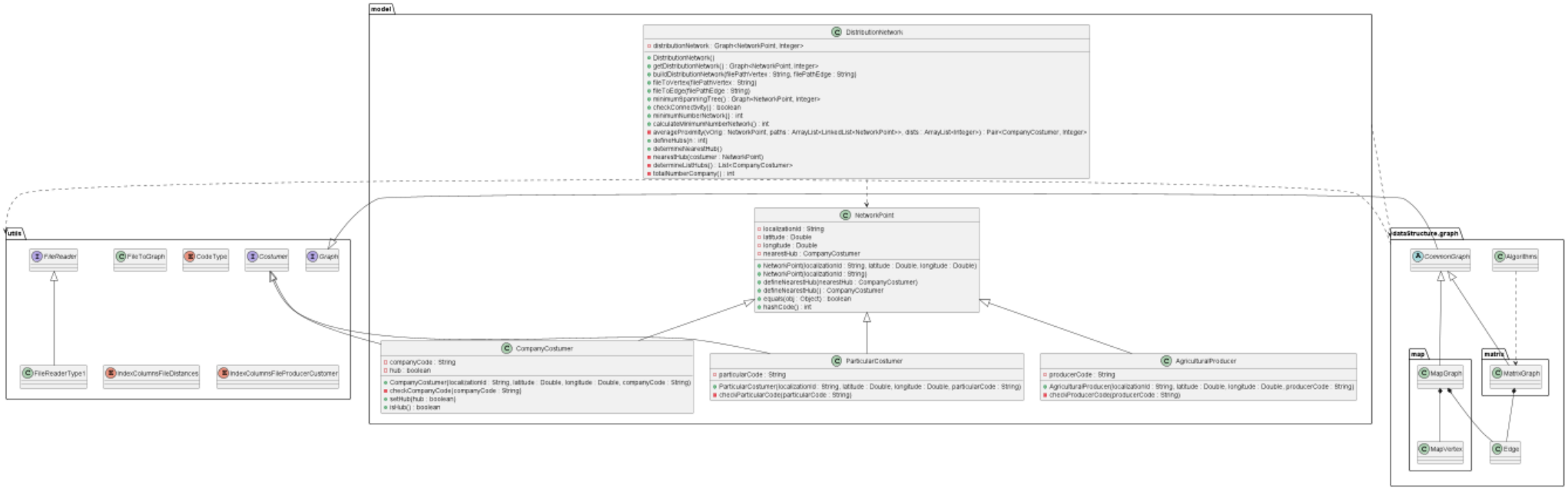
Este projeto tem como objetivo desenvolver um software com um conjunto de funcionalidades que permita gerir uma rede de distribuição de cabazes entre agricultores e clientes.

As funcionalidades requisitadas para a realização deste projeto são:

- Construir a rede de distribuição de cabazes a partir da informação fornecida nos ficheiros;
- 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;
- 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);
- Para cada cliente (particular ou empresa) determinar o hub mais próximo;
- Determinar a rede que conecte todos os clientes e produtores agrícolas com uma distância total mínima.

Estas funcionalidades devem ser implementadas da forma mais eficiente possível através do uso de um grafo que deve ser implementado usando a representação mais adequada e garantindo a manipulação indistinta dos clientes/empresas e produtores agrícolas.

Diagrama de classes



Para a estruturação das classes optou-se pela criação de 4 classes designadas por *NetworkPoint*, *CompanyCostumer*, *ParticularCostumer* e *AgriculturalProducer* no package *models*. Estas 3 últimas extendem da classe *NetworkPoint* que representa um ponto da rede.

A classe *CompanyCostumer* representa um cliente do tipo empresarial e a classe ParticularCostumer um cliente do tipo particular. Ambas as classes implementam uma interface designada *Costumer*. Existe também a classe *AgriculturalProducer* que representa um produtor agrícola.

A estrutura de dados principal adotada é um grafo onde os vértices são do tipo *NetworkPoint* e as arestas do tipo *Integer*, esta estrutura encontra-se na classe *DistributionNetwork*.

Todos os métodos necessários para a resolução dos exercícios foram desenvolvidos na classe *DistributionNetwork*.

Algoritmos – Análise de Complexidade

US301

Estrutura de dados adequada para armazenar as informações necessárias

Para o armazenamento da informação necessária para resolver a US301 e consequentemente todas as restantes US's do projeto, criou-se as classes necessárias para representar a estrutura de dados denominada de Grafos. Com a implementação desta estrutura, torna-se possível armazenar a informação proveniente de ficheiros da melhor forma para que esta informação seja utilizada posteriormente na resolução dos problemas expostos nas restantes US's.

Para o desenvolvimento desta estrutura, a equipa optou pela abordagem da matriz de adjacência, pois concordamos que seria mais prático.

Para realizar esta parte, foi utilizado as classes fornecidas nas aulas práticas laboratoriais de ESINF, em que apenas foram acrescentados mais métodos e também foram desenvolvidos aqueles métodos que ainda estavam em falta.

Foram desenvolvidos os seguintes algoritmos em conformidade com esta parte da US:

- BreadthFirstSearch;
- DepthFirstSearch;
- allPaths;
- shortestPathDijkstra;
- shortestPath;
- shortestPaths;
- getPath.

O algoritmo **BreadthFirstSearch** é um método iterativo semelhante à travessia em largura de uma árvore, onde a sua raiz contém todos os vértices alcançáveis pelo vértice s.

```java
public static <V, E> LinkedList<V> BreadthFirstSearch(Graph<V, E> g, V vert) {
    if (!g.validVertex(vert))
        return null;

    boolean[] visited = new boolean[g.numVertices()];

    LinkedList<V> qbfs = new LinkedList<>();
    qbfs.add(vert);

    LinkedList<V> qaux = new LinkedList<>();
    qaux.add(vert);

    visited[g.key(vert)] = true;

    while (!qaux.isEmpty()) {
        vert = qaux.removeFirst();
        for (V vAdj : g.adjVertices(vert)) {
            int vAdjKey = g.key(vAdj);
            if (!visited[vAdjKey]) {
                qbfs.add(vAdj);
                qaux.add(vAdj);
                visited[vAdjKey] = true;
            }
        }
    }

    return qbfs;
}
```

Este método é determinístico pois temos sempre de visitar todos os vértices e verificar todas as arestas do grafo. Desta forma, podemos afirmar que a complexidade temporal é O(V+E).

Já o algoritmo **DepthFirstSearch** é um método recursivo, em que é denominado de procura em profundidade, pois o começamos por um dado vértice e exploramos o máximo possível por cada aresta antes de fazer backtracking.

```java
private static <V, E> void DepthFirstSearch(Graph<V, E> g, V vOrig, boolean[] visited, LinkedList<V> qdfs) {
    int vOrigKey = g.key(vOrig);
    if (visited[vOrigKey])
        return;

    qdfs.add(vOrig);

    visited[vOrigKey] = true;

    for (V vAdj : g.adjVertices(vOrig)) {
        DepthFirstSearch(g, vAdj, visited, qdfs);
    }
}
```

```java
5 usages
public static <V, E> LinkedList<V> DepthFirstSearch(Graph<V, E> g, V vert) {
    if (!g.validVertex(vert))
        return null;

    LinkedList<V> qdfs = new LinkedList<>();
    boolean[] visited = new boolean[g.numVertices()];

    DepthFirstSearch(g, vert, visited, qdfs);
    return qdfs;
}
```

Este método, tal como o ***BreadthFirstSearch***, visita todos os vértices e verifica todas as arestas do grafo uma vez. Este método possui também uma complexidade temporal igual a O (V + E). Tanto este método com o anteriormente falado, podem ser representados por uma lista de adjacências.

O método ***allPaths*** retorna todos os caminhos possíveis para chegar do vértice de origem até um dado vértice de destino.

```java
private static <V, E> void allPaths(Graph<V, E> g, V vOrig, V vDest, boolean[] visited,
                                    LinkedList<V> path, ArrayList<LinkedList<V>> paths) {

    path.add(vOrig);
    visited[g.key(vOrig)] = true;

    for (V vAdj : g.adjVertices(vOrig)) {
        if (vAdj.equals(vDest)) {
            path.add(vDest);
            paths.add(path);
            path.removeLast();
        } else {
            if (!visited[g.key(vAdj)])
                allPaths(g, vAdj, vDest, visited, path, paths);
        }
    }

    path.removeLast();

}
```

Este método é não determinístico porque podemos ter um vértice de origem que tenha apenas um vértice até ao vértice de destino, tendo assim desta forma o melhor tempo de complexidade que é O(1). Para o pior tempo de complexidade deste método temos O(V).

O método ***shortestPathDijkstr***a procura o caminho mais curto deste o vértice de origem até todos os vértices que são de possível alcance, sendo que o grafo não pode conter vértices de peso negativos. Esta implementação tem por base o algoritmo de Dijkstra's.

```
private static <V, E> void shortestPathDijkstra(Graph<V, E> g, V vOrig,
                                                Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                                boolean[] visited, V[] pathKeys, E[] dist) {

    int vKey = g.key(vOrig);
    dist[vKey] = zero;
    pathKeys[vKey] = vOrig;

    while (vOrig != null) {
        int keyVOrig = g.key(vOrig);
        visited[keyVOrig] = true;
        for (Edge<V, E> edge : g.outgoingEdges(vOrig)) {
            int keyVAdj = g.key(edge.getVDest());
            E sumLength = sum.apply(dist[keyVOrig], edge.getWeight());
            if (!visited[keyVAdj] && (dist[keyVAdj] == null || ce.compare(dist[keyVAdj], sumLength) > 0)) {
                dist[keyVAdj] = sumLength;
                pathKeys[keyVAdj] = vOrig;
            }
        }

        E minDist = null;
        vOrig = null;
        for (V vertex : g.vertices()) {
            int vertexKey = g.key(vertex);
            if (!visited[vertexKey] && dist[vertexKey] != null && (minDist == null || ce.compare(dist[vertexKey], minDist) < 0)) {
                minDist = dist[vertexKey];
                vOrig = vertex;
            }
        }
    }

}
```

Este algoritmo é não determinístico pois depende de se todos os vértices podem ou não ser alcançados a partir do vértice de origem. A melhor complexidade temporal é O(V*V) e a pior complexidade é (V+E)log(V).

O algoritmo **shortestPath** encontram o caminho mais curto entre dois vértices, o vértice origem e o vértice destino.

```
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();

    boolean[] visited = new boolean[g.numVertices()];
    V[] pathKeys = (V[]) Array.newInstance(vOrig.getClass().getSuperclass(), g.numVertices());
    E[] dist = (E[]) Array.newInstance(zero.getClass().getSuperclass(), g.numVertices());


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

    E lengthToReachDest = dist[g.key(vDest)];
    if (lengthToReachDest == null)
        return null;

    getPath(g, vOrig, vDest, pathKeys, shortPath);

    return lengthToReachDest;
}
```

Este algoritmo como implementa o método *shortestPathDijkstra* e ele é não determinístico, este método é automaticamente não determinístico também. O pior tempo de complexidade deste algoritmo é O(V*V), o mesmo pior caso do algoritmo *shortestPathDijkstra*. O melhor caso de complexidade temporal é de O(((V+E)log(V))).

O método **shortestPaths** calcula o caminho mais curto entre um dado vértice e todos os outros vértices do grafo.

```
public static <V, E> boolean shortestPaths(Graph<V, E> g, V vOrig,
                                           Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                           ArrayList<LinkedList<V>> paths, ArrayList<E> dists) {
    if (!g.validVertex(vOrig))
        return false;

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

    int numVertices = g.numVertices();

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

    boolean[] visited = new boolean[numVertices];
    V[] pathKeys = (V[]) Array.newInstance(vOrig.getClass().getSuperclass(), g.numVertices());
    E[] dist = (E[]) Array.newInstance(zero.getClass().getSuperclass(), g.numVertices());

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

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

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

        return true;
    }
```

Este método como implementa métodos não determinísticos, é não determinístico também. Dados isto, podemos então dizer que com os algoritmos aqui implementados, no pior caso de complexidade temporal, este método tem O(V*V). Já no melhor tempo de complexidade temporal, o algoritmo tem O((V+E)log(V)).

O método **getPath** constrói o caminho entre dois vértices em que esse caminho é feito desde o fim até ao início.

```
private static <V, E> void getPath(Graph<V, E> g, V vOrig, V vDest,
                                   V[] pathKeys, LinkedList<V> path) {

    if (vOrig.equals(vDest))
        path.push(vDest);
    else {
        path.push(vDest);
        int keyVDest = g.key(vDest);
        vDest = pathKeys[keyVDest];
        getPath(g, vOrig, vDest, pathKeys, path);
    }
}
```

Este algoritmo é determinístico pois ele tem de passar sempre pelos vértices todos existentes na *linked list* para construir o caminho entre os dois vértices. Dado isto, podemos afirmar que este algoritmo tem complexidade temporal de O(V).

Leitura do ficheiro e colocação dos dados em estruturas adequadas

Na *US301* é necessário construir a rede de distribuição a partir da informação fornecida nos ficheiros. O grafo implementado garante a manipulação indistinta dos clientes/empresas e produtores agrícolas.

Para realizar a leitura do ficheiro optou-se pela criação da classe ***FileReaderType1*** que contém o método ***readFile*** que dado o caminho do ficheiro retorna caso este exista uma lista que contêm um *array* de *strings*. A divisão de cada linha do ficheiro em colunas realizou-se através de uma constante definida que contem o caracter ",".

```
public List<String[]> readFile(String filePath) throws FileNotFoundException {
    if (filePath == null) {
        throw new FileNotFoundException();
    }

    File file = new File(filePath);
    Scanner scannerFile = new Scanner(file);

    // Ler o cabeçalho
    scannerFile.nextLine();

    List<String[]> fileData = new ArrayList<>();
    String sentence;

    while (scannerFile.hasNext()) {
        sentence = scannerFile.nextLine();
        fileData.add(sentence.split(DATA_SEPARATOR));
    }

    scannerFile.close();

    return fileData;
}
```

Para uma melhor organização do projeto criou-se 2 enumerados que possuem a posição de cada campo no ficheiro:

- ***IndexColumnsFileProducerCustomer*** – Contém a posição de cada campo do ficheiro de cliente-produtores;
- ***IndexColumnsFileDistances*** – Contém a posição de cada campo dos ficheiros de distâncias.

Na classe ***FilToGraph*** foi criado o método ***readFileToVertex*** que recebe como parâmetro o grafo que pretende-se adicionar os vértices e o caminho do ficheiro.

Para guardar a informação do ficheiro de clientes-produtores nos vértices do grafo foi criado um ciclo for para percorrer a lista retornada pelo método de leitura do ficheiro, seguidamente em cada iteração é criado um objeto *NetworkPoint* que pode ser do tipo *CompanyCostumer*, *ParticularCostumer* ou *AgriculturalProducer* dependendo do código. Caso este não seja nulo é adicionado ao vértice do grafo.

Neste método é lançado uma exceção, caso exista algum problema com o ficheiro passado por parâmetro.

```
public static void readFileToVertex(Graph<NetworkPoint, Integer> distributionNetwork, String filePathVertex) {
    try {
        List<String[]> contentFile = new FileReaderType1().readFile(filePathVertex);

        for (String[] columns : contentFile) {
            if (columns.length == IndexColumnsFileProducerCustomer.TOTAL_COLUMNS.getIndex()) {

                NetworkPoint networkPoint = createNetworkPoint(columns[IndexColumnsFileProducerCustomer.LOCALIZATION_ID
                        columns[IndexColumnsFileProducerCustomer.LATITUDE.getIndex()],
                        columns[IndexColumnsFileProducerCustomer.LONGITUDE.getIndex()],
                        columns[IndexColumnsFileProducerCustomer.NAME.getIndex()]);

                if (networkPoint != null) {
                    distributionNetwork.addVertex(networkPoint);
                }
            }
        }

    } catch (FileNotFoundException e) {
        throw new IllegalArgumentException("Não existe nenhum ficheiro no caminho indicado.");
    }
}
```

Existe também um enumerado denominado *CodeType* que é bastante importante para o domínio de negócio, pois através do uso dele é possível verificar em cada um dos *models CompanyCostumer*, *ParticularCostumer* e *AgriculturalProducer* se o código contém na primeira posição o respetivo caracter exigido.

No caso do código do cliente deve ser o caracter C, no código da empresa o E e no código do produtor o P.

Ao criar uma nova instância de um dos *models* referido anteriormente o construtor do mesmo chama um método *check* que verifica se o respetivo código encontra-se com a estrutura exigida. Caso não aconteça é lançada uma exceção do tipo *IllegalArgumentException*.

Na figura seguinte encontra-se um exemplo da verificação realizada ao código do cliente do tipo particular.

```
private void checkParticularCode(String particularCode) {
    char charParticularCode = CodeType.COSTUMER_PARTICULAR.getCharacter();

    if (particularCode.charAt(0) != charParticularCode) {
        throw new IllegalArgumentException("O código do cliente do 'tipo' particular tem de começar obrigatoriamente " +
                "com a letra " + charParticularCode + ".");
    }
}
```

Existe também o método ***readFileToEdge*** que recebe como parâmetro o grafo que pretende-se adicionar as arestas e o caminho do ficheiro.

Este método tem um comportamento muito semelhante ao método *readFileToVertex* com a devida diferença de em cada iteração criar um objeto *Integer* que representa o peso da aresta.

Ao adicionar uma aresta ao grafo é passado por parâmetro o vértice de origem e fim e também o devido peso.

```
public static void readFileToEdge(Graph<NetworkPoint, Integer> distributionNetwork, String filePathEdge) {
    try {
        List<String[]> contentFile = new FileReaderType1().readFile(filePathEdge);

        for (String[] columns : contentFile) {
            if (columns.length == IndexColumnsFileDistances.TOTAL_COLUMNS.getIndex()) {

                NetworkPoint networkPoint1 = new NetworkPoint(columns[IndexColumnsFileDistances.LOCALIZATION_ID_1.getIndex()]);
                NetworkPoint networkPoint2 = new NetworkPoint(columns[IndexColumnsFileDistances.LOCALIZATION_ID_2.getIndex()]);
                Integer distancie = parseStringToInteger(columns[IndexColumnsFileDistances.DISTANCES.getIndex()]);

                distributionNetwork.addEdge(networkPoint1, networkPoint2, distancie);
            }
        }

    } catch (FileNotFoundException e) {
        throw new IllegalArgumentException("Não existe nenhum ficheiro no caminho indicado.");
    }
}
```

US302

Na **US302** foi necessário determinar 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, para isso decidiu-se implementar o algoritmo de *Floyd-Warshall* e a *DepthFirstSearch*.

O algoritmo de *Floyd-Warshall* presente no método minDistGr**aph** calcula os caminhos mais curtos entre todos os pares de vértices de um grafo direcionado e ponderado que eventualmente possua arcos com peso negativo, mas que não possua ciclos de custo negativo.

Este algoritmo possui complexidade de tempo igual a V^3 pois existem 3 ciclos que vão percorrer todos os vértices do grafo.

```
public static <V, E> MatrixGraph<V, E> minDistGraph(Graph<V, E> g, Comparator<E> ce, BinaryOperator<E> sum) {
    int numVerts = g.numVertices();
    if (numVerts == 0)
        return null;

    E[][] mat = (E[][]) new Object[numVerts][numVerts];
    for (int i = 0; i < numVerts; i++) {
        for (int j = 0; j < numVerts; j++) {
            Edge<V, E> edge = g.edge(i, j);
            if (edge != null)
                mat[i][j] = edge.getWeight();
        }
    }

    for (int k = 0; k < numVerts; k++) {
        for (int i = 0; i < numVerts; i++) {
            if (i != k && mat[i][k] != null) {
                for (int j = 0; j < numVerts; j++) {
                    if (j != k && j != i && mat[k][j] != null) {
                        E s = sum.apply(mat[i][k], mat[k][j]);
                        if (mat[i][j] == null || ce.compare(mat[i][j], s) > 0) {
                            mat[i][j] = s;
                        }
                    }
                }
            }
        }
    }
    return new MatrixGraph<>(g.isDirected(), g.vertices(), mat);
}
```

Criamos um método com o nome ***graphDiameter*** onde chamara o algoritmo antes referido, e percorrerá a matriz por ele returnado para achar o número mínimo de ligações necessário para nesta rede qualquer cliente/produtor conseguir contactar um qualquer outro. A complexidade deste metodo

```
public static <V, E> E graphDiameter(Graph<V, E> g, Comparator<E> comp, BinaryOperator<E> sum, E zero) {
    E max = zero;
    MatrixGraph<V, E> result = minDistGraph(g, comp, sum);
    int numVerts = result.numVerts;

    for (int i = 0; i < numVerts; i++) {
        for (int j = 0; j < numVerts; j++) {
            if (result.edge(i, j) != null && comp.compare(result.edge(i, j).getWeight(), max) > 0) {
                max = result.edge(i, j).getWeight();
            }
        }
    }
    return max;

}
```

Apos a visita em profundidade, criamos o método **isConnected** para percorrer o *Array* de *boolean* e verificar se existe algum valor *false*. Caso exista um valor falso o grafo não é conexo, pois não existe ligações entre todas os vértices.

```
public static <V, E> boolean isConnected(Graph<V, E> g) {

    int vertices = g.vertices().size();
    boolean[] visited = new boolean[vertices];

    V firstVertice = g.vertices().get(0);
    LinkedList<V> qdfs = new LinkedList<>();

    DepthFirstSearch(g, firstVertice, visited, qdfs);

    boolean connected = true;

    for (int i = 0; i < visited.length; i++) {
        if (!visited[i]) {
            connected = false;
            break;
        }
    }

    return connected;
}
```

US303

Na **US303** é necessário definir os hubs da rede de distribuição, que consistem dos n clientes do tipo empresa com a medida de proximidade mais baixa, que se obtém quando se calcula a média do comprimento do caminho mais curto de cada empresa a todos os clientes e produtores agrícolas.

Serão alteradas n, empresas para hub, se o número total de clientes do tipo empresa for maior que n, se houver menos de n empresas para se alterar o método só altera o número total de empresas disponíveis.

Para se poder classificar um cliente do tipo empresa como um hub foi criada, na classe CompanyCostumer, a variável hub do tipo booleano para poder guardar o estado de uma instância de CompanyCostumer, se a variável estiver como verdadeira essa empresa é um hub ou se a variável estiver como falsa essa empresa não é um hub.

```
public class CompanyCostumer extends NetworkPoint implements Costumer {
    /**
     * Código do cliente do 'tipo' empresa.
     */
    1 usage
    private String companyCode;

    /**
     * Representa se o cliente do 'tipo' empresa é um huh
     */
    3 usages
    private boolean hub;
```

Também dentro da classe CompanyCostumer foram criados dois métodos, setHub e isHub que servem para manipular e verificar o estado do mesmo.

```
/**
 * Método para definir se o cliente do 'tipo' empresa é um huh.
 *
 * @param hub true se for um hub, caso contrário falso.
 */
1 usage   francisco_lascasas_bogalho
public void setHub(boolean hub) {
    this.hub = hub;
}

/**
 * Função para indicar se o cliente do 'tipo' empresa é um huh.
 *
 * @return true se for um hub, caso contrário falso.
 */
31 usages   francisco_lascasas_bogalho
public boolean isHub() { return hub; }
```

Já na classe DistributionNetwork onde se encontram os métodos principais desta user storie, temos os métodos defineHubs e averageProximity.

No método defineHubs, que recebe um inteiro, n, como parâmetro, que representa o número de hubs que se quer criar, começa-se por criar duas listas uma para guardar todos os vértices do grafo e uma para guardar um par, de uma instância do tipo de CompanyCostumer e um valor inteiro.

De seguida iterasse por todos os vértices guardados na lista vertices e para cada iteração chamar-se á o método shortestPaths para calcular todos os caminhos do vértice origem para todos os outros vértices e também devolve um valor booleano, true se for possível de chegar a todos os vértices a partir do origem e falso se não for possível.

Se for possível chegar a todos os vértices a partir do vértice origem chamar-se á o método averageProximity e guarda-se o valor retornado na variável potencialHub.

Depois, iterasse por todos os elementos da lista selectedHub e verificasse se o hub potencial já existe na lista selectedHubs, se já existir verifica-se se o hub potencial tem uma média de proximidade mais baixa do que aquela guardada na lista selectedHubs se for menor trocam-se os valores.

De seguida verifica-se se o par é original se for adiciona-se á lista selectedHubs. Depois ordena-se a lista por ordem crescente de média de próximidade e altera-se a variável hub das n primeiras empresas, para true. Este método como utiliza outros métodos determinísticos tem um pior caso e um melhor caso. A complexidade temporal do melhor caso é $O(V * ((V + E)log(V))$, porque se não for possível chegar a todos os vértices a partir do vértice origem não se fara o bloco do *if*, já no pior caso a complexidade temporal é de $O(V^3)$, porque se for possível chegar a todos os vértices a partir do vértice origem faz-se o bloco do *if.*

```java
public void defineHubs(int n) {
    boolean flag;
    int j = 0;
    //Create a list to store all the vertices
    List<NetworkPoint> vertices = distributionNetwork.vertices();
    //Create a list to store an companyCostumer and its average proximity to the inteded point
    List<Pair<CompanyCostumer, Integer>> selectedHubs = new LinkedList<>();

    //Iterates through all the vertices in the graph
    for (int i = 0; i < vertices.size(); i++) {
        //Creates a list to receive all the paths between two vertices
        ArrayList<LinkedList<NetworkPoint>> paths = new ArrayList<>();
        //Creates a list to receive all the weithed values of the paths
        ArrayList<Integer> dists = new ArrayList<>();

        //Assign to the variable flag true if it is  possible to calculate a path between two vertices, assign false if not
        flag = Algorithms.shortestPaths(distributionNetwork, vertices.get(i), Integer::compare, Integer::sum,  zero: 0, paths, dists);
        //Calls averageProximity only if it is possible to create a path between the two previous vertices
        if (flag) {
            //Calculates the average proximity of two vertices
            Pair<CompanyCostumer, Integer> potencialHub = averageProximity(vertices.get(i), paths, dists);
            boolean existingHub = false;

            //Iterates the list of selected hubs
            for (Pair<CompanyCostumer, Integer> hub : selectedHubs) {
                //Verifys if there is already a hub equal to the potencial in the selectedHubs list
                if (hub.equals(potencialHub)) {
                    existingHub = true;

                    //As the hub already exists in the selectedHubs list verifys if it is nessessary to change the weight
                    if (hub.getValue() > potencialHub.getValue()) {
                        hub.setValue(potencialHub.getValue());
                    }
                }
            }


            //In case that the selected hub dos not exist it is added
            if (!existingHub) {
                selectedHubs.add(averageProximity(vertices.get(i), paths, dists));
            }
        }
    }
    //Sorts the list selectedHubs
    Collections.sort(selectedHubs);

    //For every n value or for every selectedHub existing
    while (j < n && j < selectedHubs.size()) {
        //Updates the value of the variable hub of the companyCostumer to true
        selectedHubs.get(j).getKey().setHub(true);
        j++;
    }
}
```

No método averageProximity, que recebe um vértice, um arrayList com todos os caminhos possíveis do vértice origem para todos os outros vértices e um arrayList com os valores de distância para todos os caminhos desde o vértice inicial até todos os outros vértices, começa-se por iterar por todos os vértices possíveis, e se o vértice em analise não for igual ao vértice origem e for um vértice do tipo CompanyCostumer, guarda-se a distância do vértice original ao vértice em analise na variável distance. Se a distância em distance for menor que a em minDistance, troca-se o valor em minDistance pelo em distance e atualiza-se o valor de costumer para o

vértice em atualmente em analise. Calcula-se o valor médio da proximidade entre o vértice original e o vértice do tipo CompanyCostumer mais perto. Por fim devolve um par com um vértice do tipo CompanyCostumer e um inteiro. Este método tem complexidade temporal de $O(V)$, porque o método inclui um for que itera por todos os vértices.

```
private Pair<CompanyCostumer, Integer> averageProximity(NetworkPoint vOrig, ArrayList<LinkedList<NetworkPoint>> paths, ArrayList<Integer> dists) {
    List<NetworkPoint> vertices = distributionNetwork.vertices();
    Pair<CompanyCostumer, Integer> potencialHub;
    int distance;
    int minDistance = Integer.MAX_VALUE;
    int totalPaths = 0;
    int average;
    CompanyCostumer costumer = null;

    //Iterates through all the vertices in the graph
    for (int i = 0; i < vertices.size(); i++) {
        //Verifys if the starting vertice is the same as the ending vertice and if the ending vertice is a companyCostumer
        if (vOrig != vertices.get(i) && (vertices.get(i) instanceof CompanyCostumer)) {
            //Assign to the variable the distance between two vertices
            distance = dists.get(distributionNetwork.key(vertices.get(i)));
            //Updates the value of the minDistance
            if (minDistance > distance) {
                minDistance = distance;
                totalPaths = paths.get(distributionNetwork.key(vertices.get(i))).size();
                costumer = (CompanyCostumer) vertices.get(i);
            }
        }
    }

    //Calculates the average proximity
    average = minDistance / totalPaths;
    potencialHub = new Pair<>(costumer, average);

    return potencialHub;
}
```

US304

Nesta *User Storie* era pedido para determinarmos para cada cliente o hub mais próximo. E para isso, utilizamos a classe ***DistributionNetwork***, começamos por chamar um método da *User Storie* anterior (US303), sendo que vai definir os hubs existentes, definindo assim todas as empresas como hub. De seguida, vamos percorrer todos os vértices do grafo e verificar se o vértice atual é um cliente, posto isto, irá encontrar o hub mais próximo e assim sendo irá definir o hub mais próximo.

Análise de complexidade para este método é: pior caso -> O (/V*(V+(VV)+V) + V+V+V(VW)) = O (VWWV); melhor caso -> O ((V (V + ((V+E) log(V)) + V) + V+V+V) + V log(V) + V + (V((V+E) log(V)))) = O()

```java
public void determineNearestHub() {
    // Calls US303's method to define existing hubs, defining all companies as a hub
    defineHubs(totalNumberCompany());

    //Traverse all vertices of the graph
    for (NetworkPoint networkPoint : distributionNetwork.vertices()) {
        //Checks if the current vertex is a customer
        if (networkPoint instanceof Costumer) {
            //Find the nearest hub
            CompanyCostumer nearestHub = nearestHub(networkPoint);

            //Defines the nearest hub
            networkPoint.defineNearestHub(nearestHub);
        }
    }
}
```

Optamos por criar outra função que nos permitirá determinar o hub mais próximo de um ponto de rede. Em que inicialmente, começamos por criar uma lista de hubs para posteriormente percorrê-la para verificar se o ponto da rede passado por parâmetro não é um hub. Depois chamamos o algoritmo **shortestPath** que retorna a distância entre dois pontos. De seguida, vamos fazer uma verificação para saber a distância, para atualizar o hub mais próximo se necessário, para que no fim possamos retornar o hub mais próximo.

A análise de complexidade para *shortestPath* é: *shortestPast* -> pior caso: O ((VV) + V = O (W)); melhor caso: O (((V + E) log(V))).

```java
private CompanyCostumer nearestHub(NetworkPoint costumer) {
    //Calls the method to get a list of hubs
    List<CompanyCostumer> listHubs = determineListHubs();

    int shorterDistance = Integer.MAX_VALUE;
    CompanyCostumer nearestHub = null;

    //Scrolls through the list of hubs
    for (CompanyCostumer hub : listHubs) {
        //Checks if the network point passed by parameter is not a hub
        if (costumer != hub) {
            LinkedList<NetworkPoint> shortPath = new LinkedList<>();
            //Calls the shortestPath algorithm that returns the distance between two points
            Integer distanceToHub = Algorithms.shortestPath(distributionNetwork, costumer, hub, Integer::compare, Integer::sum, zero: 0, shortPath);

            //Checks the distance, to update the nearest hub if necessary
            if (shorterDistance > distanceToHub) {
                shorterDistance = distanceToHub;
                nearestHub = hub;
            }
        }
    }

    //Returns the nearest hub
    return nearestHub;
}
```

Criamos também uma função para podermos encontrar e retornar a lista de hubs existentes. Começa por percorrer todos os vértices do grafo, depois vai verificar se o ponto da rede é ou não uma empresa e, caso seja um hub vai adicionar à lista. Por fim, irá retornar a lista de hubs.

A complexidade para este método é: *determineListHubs* -> O(V).

```java
private List<CompanyCostumer> determineListHubs() {
    List<CompanyCostumer> determineListHubs = new ArrayList<>();

    //Traverse all vertices of the graph
    for (NetworkPoint networkPoint : distributionNetwork.vertices()) {

        //Checks if the network point is a company
        if (networkPoint instanceof CompanyCostumer) {
            //See if the company is a hub
            boolean isHub = ((CompanyCostumer) networkPoint).isHub();

            //If it is a hub, add it to the list
            if (isHub) {
                determineListHubs.add((CompanyCostumer) networkPoint);
            }
        }
    }

    //Returns the list of hubs
    return determineListHubs;
}
```

Relativamente à última função referente a esta *User Storie* esta serve para contar o número de empresas existentes no grafo que por fim irá retornar o número de empresas existentes na rede de distribuição.

Começamos então por percorrer todos os vértices da rede de distribuição para de seguida verificarmos se esse vértice é uma empresa, se sim irá somar +1 ao total. Por fim, retorna o total.

```
private int totalNumberCompany() {
    int total = 0;

    //It traverses all vertices of the distribution network
    for (NetworkPoint networkPoint : distributionNetwork.vertices()) {
        //Checks if this vertex is a company, if so add +1 to the total
        if (networkPoint instanceof CompanyCostumer) {
            total++;
        }
    }

    return total;
}
```

US305

Na **US305** foi necessário determinar a rede que conecte todos os clientes e produtores agrícolas com uma distância total mínima para isso decidiu-se implementar o algoritmo de *Kruskal*.

O algoritmo de *Kruskal* é um algoritmo que procura uma árvore geradora mínima para um grafo conexo com pesos. Este algoritmo encontra um subconjunto das arestas que forma uma árvore que inclui todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore, é minimizado.

O algoritmo implementado tem a seguinte estrutura:

1. Colocação dos vértices na árvore geradora mínima;
2. Colocação das arestas numa lista;
3. Ordenação da lista das arestas;
4. Percorre a lista de arestas e para cada aresta realiza uma procura em profundidade na árvore geradora mínima para verificar se estas já contêm a aresta. Caso não contenha, a aresta é adicionada à arvore.
5. Retorna a árvore geradora mínima.

Este algoritmo possui complexidade de tempo igual a O (e log n), onde e representa o número de arestas e n o número de vértices.

```
public static <V, E extends Comparable<E>> Graph<V, E> kruskal(Graph<V, E> g) {
    Graph<V, E> minimumSpanningTree = new MapGraph<>( directed: false);
    LinkedList<Edge<V, E>> listEdges = new LinkedList<>();

    for (V vertex : g.vertices()) {
        minimumSpanningTree.addVertex(vertex);
    }

    for (Edge<V, E> edge : g.edges()) {
        listEdges.add(edge);
    }

    Collections.sort(listEdges);

    LinkedList<V> connectedVertex;
    for (Edge<V, E> edge : listEdges) {
        connectedVertex = DepthFirstSearch(minimumSpanningTree, edge.getVOrig());

        if (!connectedVertex.contains(edge.getVDest())) {
            minimumSpanningTree.addEdge(edge.getVOrig(), edge.getVDest(), edge.getWeight());
        }
    }

    return minimumSpanningTree;
}
```

Por fim, no *model DistributionNetwork* foi criado um método denominado **minimumSpanningTree** que chama o algoritmo de *Kruskal* explicado anteriormente e devolve a árvore de extensão mínima que conecta todos os clientes e produtores agrícolas com uma distância total mínima.

```
public Graph<NetworkPoint, Distance> minimumSpanningTree() {
    return Algorithms.kruskal(distributionNetwork);
}
```

Divisão de trabalho

Antes de elaborarmos o trabalho e, após uma análise em grupo, optamos por dividir as *User Stories* pelos diferentes membros da equipa da seguinte forma:

- US301 - Gabriel Gonçalves;
- US302 - João Durães;
- US303 - Francisco Bogalho;
- US304 - António Bernardo;
- US305 - Tiago Leite.

Melhoramentos possíveis

Em relação a melhoramentos, podemos indicar os seguintes pontos a serem melhorados neste projeto:

- Complexidade Temporal;

Conclusão

Em suma, neste projeto desenvolvemos as nossas capacidades ao usar as utilidades dos grafos. Podemos concluir ainda, que quanto mais estudado for o caso e melhor planeado for o trabalho mais facilmente se absorvem as alterações que acontecem ao longo do processo.

Posto isto, consideramos que, apesar de ser possível efetuar algumas melhorias neste trabalho, o projeto encontra-se de acordo com os requisitos pedidos.