



Instituto Superior de
Engenharia do Porto

Relatório das funcionalidades implementadas no Sprint1 - Correções na Entrega do Sprint 2

Projeto realizado pelo Grupo 102:

- 1190701 João Sousa
- 1201942 Fábio Silva
- 1190569 Filipe Morais
- 1200618 Jorge Cunha
- 1191576 Francisco Sampaio

Index

Contextualização	3
Class Diagram	3
UC 301	4
Enunciado:	4
Proposta de Solução:	4
Explicação:	5
UC302	6
Enunciado:	6
Proposta de solução:	6
Explicação:	8
UC303	9
Enunciado:	9
Proposta de Solução	9
Explicação:	10
UC304	10
Enunciado:	10
Explicação:	11
UC305	11
Enunciado:	11
Proposta de Solução:	11
Explicação:	11
Funções adicionais:	12
VertCycle	12
Proposta de solução:	12
Explicação	12
Proposta de solução:	13
Explicação:	13

Contextualização

Class Diagram

Os diagramas de classes foi entregue em conjunto com este documento. Devido ao seu tamanho, incluí-lo neste ficheiro torná-lo-ia muito difícil de ler, razão pela qual foi entregue em separado, num formato de SVG.

Depois da leitura do enunciado do projeto, identificámos as classes base que modelam as entidades do negócio: *Pessoa*, *Cliente*, *Empresa*, *Produtor*. Estas classes seriam armazenadas em grafos, para tal implementamos as classes algoritmos, graphs e MapGraphs.

As classes de grafos e os seus algoritmos usadas foram as desenvolvidas durante as aulas. Optámos por usar o MapGraph, pois em deterimento do problema em questão é mais eficiente comparando com as outras opções.

A classe responsável pela importação do CSV é a *ReadFromCSV*, que depois de processar os dados, cria objetos pessoa e distancias que são armazenadas em listas para mais tarde serem utilizadas na criação dos grafos.

De forma a ter maior organização do projeto, o grupo decidiu criar um ui e um controller, o que nos permite obter os vários métodos sem estes serem todos static.

Por fim, foram criadas algumas classes que albergam os dados de retorno dos exercícios.

UC 301

Enunciado:

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).

Proposta de Solução:

```
1 usage 1 Jorge Cunha +1
public List<Pessoa> lerClienteProdutor(String caminho) throws FileNotFoundException {
    Scanner read = new Scanner(new File(caminho));
    read.nextLine();
    while (read.hasNextLine()) {
        String[] line = read.nextLine().split(" ");
        if (line[CLIENTEPRODUTOR_NOME].contains("E")) { // 0(n)
            Empresa empresaAux = new Empresa(line[CLIENTEPRODUTOR_LOC], line[CLIENTEPRODUTOR_NOME], new Point2D.Double(Double.parseDouble(line[CLIENTEPRODUTOR_LAT]), Double.parseDouble(line[CLIENTEPRODUTOR_LONG])));
            if (!storeLists.getPessoaList().contains(empresaAux)) { // 0(n)
                storeLists.addPessoaList(empresaAux);
            }
        }
        if (line[CLIENTEPRODUTOR_NOME].contains("P")) { // 0(n)
            Produtor produtorAux = new Produtor(line[CLIENTEPRODUTOR_LOC], line[CLIENTEPRODUTOR_NOME], new Point2D.Double(Double.parseDouble(line[CLIENTEPRODUTOR_LAT]), Double.parseDouble(line[CLIENTEPRODUTOR_LONG])));
            if (!storeLists.getPessoaList().contains(produtorAux)) { // 0(n)
                storeLists.addPessoaList(produtorAux);
            }
        }
        if (line[CLIENTEPRODUTOR_NOME].contains("C")) { // 0(n)
            Cliente clienteAux = new Cliente(line[CLIENTEPRODUTOR_LOC], line[CLIENTEPRODUTOR_NOME], new Point2D.Double(Double.parseDouble(line[CLIENTEPRODUTOR_LAT]), Double.parseDouble(line[CLIENTEPRODUTOR_LONG])));
            if (!storeLists.getPessoaList().contains(clienteAux)) { // 0(n)
                storeLists.addPessoaList(clienteAux);
            }
        }
    }
    return storeLists.getPessoaList();
} // 0(n^2)
```

```
86 */
1 usage 1 Jorge Cunha +1
87 public List<Helper> lerDistancias(String caminho) throws FileNotFoundException {
88     Scanner read = new Scanner(new File(caminho));
89     read.nextLine();
90     while (read.hasNextLine()) {
91         String[] line = read.nextLine().split(" ");
92         Helper helperaux = new Helper(line[DISTANCIAS_UM], line[DISTANCIAS_DOIS], Integer.parseInt(line[DISTANCIAS_TAMANHO]));
93         if (!storeLists.getLocalidadeList().contains(helperaux)) { // 0(n)
94             storeLists.addLocalidadeList(helperaux);
95         }
96     }
97 }
98
99     return storeLists.getLocalidadeList();
100 } // 0(n)
```

```
1 usage  + Fábio Silva +1
public MapGraph<Pessoa, Integer> createGraph() {
    List<Pessoa> listPessoa = storeLists.getPessoaList();
    List<Helper> listHelper = storeLists.getLocalidadeList();

    for (Pessoa pessoa : listPessoa) { // 0(n)
        graph.addVertex(pessoa); // 0(n)
    }

    // 0(n^3)
    for (Helper helper : listHelper) { // 0(m)
        for (Pessoa pessoa : listPessoa) { // 0(n)
            if (helper.getLocalidade1().equals(pessoa.getLocalidade())) {
                Pessoa auxPessoa = findPessoaByLocal(helper.getLocalidade2()); // 0(n)
                if (auxPessoa != null) {
                    graph.addEdge(pessoa, auxPessoa, helper.getDistancia()); // 0(1)
                }
            }
        }
    }

    return graph;
} // 0(n^3)
```

```
/**
 * Find a person by location.
 *
 * @param local The location of the person.
 * @return A list of Pessoa objects.
 */
2 usages  + Fábio Silva +1
public Pessoa findPessoaByLocal(String local) {
    List<Pessoa> listPessoa = storeLists.getPessoaList();

    for (Pessoa pessoa : listPessoa) { // 0(n)
        if (pessoa.getLocalidade().equals(local)) {
            return pessoa;
        }
    }

    return null;
} // 0(n)
}
```

Explicação:

No método lerClienteProdutor, o ficheiro é lido e armazenado na storelist para depois ser usado.

No método lerDistancias, o ficheiro é lido e armazenado na storelist para depois ser usado.

Após serem lidos ambos os ficheiros é usado o método createGraph que pega na lista de clientes Produtores e vai adicionando como vértices e em cada vértice adiciona o edge com o ficheiro das distâncias.

UC302

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.

Proposta de solução:

```
/**
 * It checks if the graph is connected by doing a Breadth First Search from each vertex and checking if the number of
 * vertices in the graph is equal to the number of vertices in the list returned by the BFS
 *
 * @return The method returns a boolean value, true if the graph is connected and false if it is not.
 */
1 usage  Jorge Cunha +1
public boolean verificarGraphConexo() {

    List<Pessoa> pessoaList = mapGraphPessoaStore.getMapGraph().vertices(); // O(V)

    for (Pessoa pessoaAux : pessoaList) { // O(n)
        List<Pessoa> listAux = Algorithms.BreadthFirstSearch(mapGraphPessoaStore.getMapGraph(), pessoaAux); // O(|E| + |V|)

        if (listAux == null) {
            System.out.println("Grafo nao conexo");
            return false;
        }
        if (listAux.size() == pessoaList.size()) {
            System.out.println("Grafo conexo");
            System.out.println(listAux.size() + " nr de vertices"); // O(1)
            return true;
        }
    }
    System.out.println("Grafo nao conexo");
    return false;
} // O(|E| + |V|)
```

Método inicialmente usado da US302:

```
/**
 * It gets the graph, gets all the vertices, and for each vertex it gets all the cycles, and then it gets the biggest
 * size of the cycles
 *
 * @return The number of people in the biggest group of friends.
 */
1 usage 1 Fábio Silva +1
public int numeroLigacoesDiam(){
    MapGraph<Pessoa, Integer> mapGraph = mapGraphPessoaStore.getMapGraph();
    int max = -1, helper;

    if (mapGraph == null){
        return max;
    }

    List<Pessoa> pessoaList = mapGraphPessoaStore.getMapGraph().vertices();// 0(V)

    // 0(n^6)
    for (Pessoa pessoa : pessoaList) { // 0(n)
        ArrayList<LinkedList<Pessoa>> listAux = Algorithms.vertCycles(mapGraph, pessoa, comparator); // 0(n^4)
        helper = biggestSize(listAux); // 0(n)
        if (max < helper) {
            max = helper;
        }
    }
    return max;
} // 0(n^6)
```

```
/**
 * It returns the biggest size of a list in a list of lists
 *
 * @param listAux is the list of lists that will be used to create the tree.
 * @return The biggest size of the list.
 */
1 usage 1 Jorge_Cunha07 +1
private int biggestSize(ArrayList<LinkedList<Pessoa>> listAux) {
    OptionalInt max = listAux.stream().mapToInt(LinkedList::size).max(); // 0(n)

    return max.getAsInt() - 1;
}
```

Método Alternativo da US 302:

```
// Diameter
1 usage  Jorge Cunha
public int numeroLigacoesDiamOpcaoDois() {
    MapGraph<Pessoa, Integer> mapGraph = new MapGraph<>(mapGraphPessoaStone.getMapGraph());

    ArrayList<Pessoa> pessoasList = mapGraph.vertices();// 0(V)
    Collection<Edge<Pessoa, Integer>> edges = mapGraph.edges();// 0(E)

    for (Edge<Pessoa, Integer> edge:edges) { // 0(E)
        edge.setWeight(1);
    }

    MatrixGraph<Pessoa, Integer> minDistGraph = Algorithms.minDistGraph(mapGraph, comparator, binaryOperator);// 0(n^3)
    int max = 0;

    for (Pessoa pessoa1:pessoasList) { // 0(n)
        for (Pessoa pessoa2:pessoasList) { // 0(n)
            if (pessoa1 != pessoa2){
                int aux = minDistGraph.edge(pessoa1, pessoa2).getWeight();
                if (aux > max){
                    max = aux;
                }
            }
        }
    }

    return max;
} // 0(n^3)
```

Explicação:

No método verificarGraphConexo, nós vamos buscar os dos vértices e fazemos um for each e em cada vértice usamos o BreadthFirstSearch para retornar um caminho e verificamos se existe pelo menos um caminho que contenha todos os vértices e se tiver o grafo é conexo.

No método numeroLigacoesDiam vamos buscar os vértices outra vez e fazemos um for each e em cada vértice usamos o vertCycles que nos ira dar todos os caminhos pelo o método depthFirstSearch e adiciona numa lista, depois procuramos a lista com o maior número de ligações e retornamos esse número.

UC303

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)

Proposta de Solução

```
/**
 * It returns a list of the n people who are closest to the average of their closest hubs
 *
 * @param n number of people to return
 * @return A list of the n people with the lowest average distance to their closest hub.
 */
1 usage  1 Jorge Cunha +1
public List<Pessoa> averageClosestHub(int n) {
    Map<Pessoa, Map<Pessoa, Integer>> map = closestHub();
    Map<Double, Pessoa> averageMap = new TreeMap<>();
    List<Pessoa> list = new ArrayList<>();

    int distance;
    double average;

    for (Pessoa pessoa : map.keySet()) { //O(n)
        Map<Pessoa, Integer> auxMap = map.get(pessoa);
        distance = 0;
        for (Map.Entry<Pessoa, Integer> entry : auxMap.entrySet()) { //O(n)
            distance += entry.getValue();
        }
        average = (double) distance / auxMap.size();
        averageMap.put(average, pessoa);
    }

    int i = 0;
    for (Map.Entry<Double, Pessoa> entry : averageMap.entrySet()) { //O(n)
        list.add(entry.getValue());
        i++;
        if (i >= n) {
            return list;
        }
    }

    return list;
} //O(n^2)
```

```
/**
 * It finds the shortest path between every hub and every other node in the graph
 *
 * @return A map with the hubs as keys and a map with the closest pessoas as values.
 */
1 usage  1 Jorge Cunha +2
public Map<Pessoa, Map<Pessoa, Integer>> closestHub() {
    MapGraph<Pessoa, Integer> graph = mapGraphPessoaStore.getMapGraph();
    List<Empresa> hubsList = findAllHubs();
    Map<Pessoa, Map<Pessoa, Integer>> pessoaMap = new HashMap<>();
    ArrayList<LinkedList<Pessoa>> paths = new ArrayList<>();
    ArrayList<Integer> dists = new ArrayList<>();

    for (Empresa hub : hubsList) { //O(n)
        Algorithms.shortestPaths(graph, hub, comparator, binaryOperator, zero, 0, paths, dists); //O(n^2)
        pessoaMap.put(hub, new HashMap<>());
        for (int i = 0; i < dists.size(); i++) { //O(n)
            if (paths.get(i).getLast() != hub && (paths.get(i).getLast() instanceof Empresa || paths.get(i).getLast() instanceof Cliente)) {
                pessoaMap.get(hub).put(paths.get(i).getLast(), dists.get(i));
            }
        }
    }

    return pessoaMap;
} //O(n^3)
```

Explicação:

Neste método findNearestHub, ele começa por fazer o Algoritmo minDistGraph que faz com que retorne a matrix graph (ajudas nos a converter os weights e torna a execução do programa mais rápido), logo de seguida vamos buscar os vértices e criamos um hash map.

Fazemos um for each para todos os vértices que sejam Empresa para ver as ligações com os outros que não são Empresa.

UC304

Enunciado:

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

Proposta de Solução:

```
/**
 * For each client, get the n nearest hubs and their distances
 *
 * @param auxMap a map of clients and their nearest hubs, sorted by distance
 * @param n      number of nearest hubs to be returned
 * @return A map of clients and their nearest hubs.
 */
2 usages  ▲ Jorge Cunha +1
public Map<Pessoa, Map<Pessoa, Integer>> clientHubN(Map<Pessoa, TreeMap<Integer, Pessoa>> auxMap, int n) {
    Map<Pessoa, Map<Pessoa, Integer>> nearestHubsMap = new HashMap<>();

    for (Pessoa client : auxMap.keySet()) { //O(n)
        Map<Pessoa, Integer> nearestHubs = new HashMap<>();
        TreeMap<Integer, Pessoa> sortedMap = auxMap.get(client);
        int i = 0;
        for (Integer distance : sortedMap.keySet()) { //O(n)
            if (i >= n) {
                break;
            }
            nearestHubs.put(sortedMap.get(distance), distance);
            i++;
        }
        nearestHubsMap.put(client, nearestHubs);
    }
    return nearestHubsMap;
} //O(n^2)
```

```
//US303

/**
 * It returns a map of clients and their respective distances to the companies they are connected to
 *
 * @return A map with the clients as keys and a tree map as values. The tree map has the distance as key and the
 * company as value.
 */
2 usages  ▲ Jorge Cunha +1
public Map<Pessoa, TreeMap<Integer, Pessoa>> clientHub() {
    MapGraph<Pessoa, Integer> graph = mapGraphPessoaStore.getMapGraph();
    List<Pessoa> listClients = findAllClients(); //O(n)
    Map<Pessoa, TreeMap<Integer, Pessoa>> pessoaTreeMap = new HashMap<>();
    ArrayList<LinkedList<Pessoa>> paths = new ArrayList<>();
    ArrayList<Integer> dists = new ArrayList<>();

    for (Pessoa client : listClients) { //O(n)
        Algorithms.shortestPaths(graph, client, comparator, binaryOperator, zero, 0, paths, dists);
        pessoaTreeMap.put(client, new TreeMap<>());
        for (int i = 0; i < dists.size(); i++) { //O(n)
            if (paths.get(i).getLast() != client && paths.get(i).getLast() instanceof Empresa) {
                pessoaTreeMap.get(client).put(dists.get(i), paths.get(i).getLast()); //O(logn)
            }
        }
    }
    return pessoaTreeMap;
} //O(n^2 * logn)
```

Explicação:

No método findNearestHubEachClient, logo de início chamamos o minDistGraph pois calcula o grafo de distancia mínima usando o Floyd-Warshall. Logo de seguida percorremos a lista de clientes e vemos as distancias entre todas as empresas e guardamos num mapa auxiliar onde depois vai ser mandado para uma Lista de Strings .

UC305

Enunciado:

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

Proposta de Solução:

```
//US305
/**
 * It finds the client that is closer to all the other clients
 *
 * @return The person who is closer to all the clients.
 */
1 usage  A Fábio Silva
public Map<List<Pessoa>, Integer> findNetworkWithMinimumDistance(MapGraph<Pessoa>, Integer> graph) {
    if (graph == null) {
        return null;
    }

    MatrixGraph<Pessoa, Integer> mg = Algorithms.minDistGraph(graph, comparator, binaryOperator);

    boolean[] visited;
    int sum, minDist = 0, sumMinDist = -1;
    List<Pessoa> minPath = new ArrayList<>();
    List<Pessoa> path;
    for (Pessoa pessoa : mg.vertices()) {
        path = new ArrayList<>();
        visited = new boolean[mg.vertices().size()];
        Pessoa vOrig = pessoa;
        path.add(vOrig);
        Pessoa vOrigAux = null;
        int counter = 1;
        sum = 0;
        while (counter < mg.vertices().size()) {...}
        if (sum < sumMinDist || sumMinDist < 0) {
            sumMinDist = sum;
            minPath = path;
        }
    }

    Map<List<Pessoa>, Integer> returnableMap = new HashMap<>();
    returnableMap.put(minPath, minDist);

    return returnableMap;
} //O(n^4)
```

Explicação:

No método findNetworkWithMinimumDistance nos chamamos o minDistGraph para calcular o grafo de distancia mínima e passar para matrixGraph e fazemos o algoritmo Kruskal e retornamos a rede ainda em matriz.

Funções adicionais:

VertCycle

Proposta de solução:

```
2 usages  Jorge Cunha *  
public static <V, E> ArrayList<LinkedList<V>> vertCycles(Graph<V, E> g, V vOrig, Comparator<E> ce) {  
    boolean[] visited = new boolean[g.numVertices()]; // O(n)  
    ArrayList<LinkedList<V>> paths = new ArrayList<>();  
    for (V v : g.vertices()) { // O(V)  
        Arrays.fill(visited, false);  
        allPathNoBacktracking(g, v, v, visited, new LinkedList<>(), paths, ce); // O(n^4)  
    }  
    paths.removeIf(path -> !path.contains(vOrig)); // O(n)  
    return paths;  
} // O(n^4 * V)
```

Explicação

Este método executa uma versão modificada do algoritmo allPaths onde agora não retrocede, desta forma ele faz todos os ciclos que um simples depthFirstSearch pode encontrar em uma primeira leitura e os adiciona a uma única lista, após a qual o método retorna apenas os ciclos que contenham o vértice fornecido, pois estes podem ser movidos de forma que o vértice esteja no início.

AllPathNoBacktracking:

Proposta de solução:

```

2 usages  ± Jorge Cunha +1
private static <V, E> void allPathNoBacktracking(Graph<V, E> g, V vOrig, V vDest, boolean[] visited,
                                                    LinkedList<V> path, ArrayList<LinkedList<V>> paths, Comparator<E> ce) {

    path.push(vOrig);
    visited[g.key(vOrig)] = true;
    ArrayList<V> adjVertices = (ArrayList<V>) g.adjVertices(vOrig);
    ± Jorge Cunha
    adjVertices.sort(new Comparator<V>() {
        ± Jorge Cunha
        @Override
        public int compare(V v, V t1) {
            return ce.compare(g.edge(vOrig, v).getWeight(), g.edge(vOrig, t1).getWeight());
        }
    });

    // O(n^4)
    for (V vAdj : adjVertices) { // O(n)
        if (vAdj == vDest) {
            path.push(vDest);
            LinkedList<V> reversed = new LinkedList<V>();
            for (V v : path) { // O(m)
                reversed.push(v);
            }
            paths.add(reversed);
            path.pop(); // O(k)
        } else if (!visited[g.key(vAdj)])
            allPathNoBacktracking(g, vAdj, vDest, visited, path, paths, ce); // O(n)
    }
    path.pop(); // O(n)
} // O(n^4)

```

Explicação:

O algoritmo inicial do allPath tem uma complexidade que é inutilizável para grafos tão grandes quanto o que estamos usando, portanto, este método não retrocede, mas é compensado pelo fato de ser executado para cada, vértice no gráfico para obter uma melhor certeza.