Instituto Superior de Engenharia do Porto

Relatório das funcionalidades implementadas

Projeto realizado pelo Grupo 102:

-1190701 João Sousa

- 1201942 Fábio Silva

-1190569 Filipe Morais

-1200618 Jorge Cunha

-1191576 Francisco Sampaio

Index

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  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( regex: ",");
        if (line[CLIENTEPRODUTOR_NOME].contains("E")) {// O(n)
            Empresa empresaAux = new Empresa(line[CLIENTEPRODUTOR_LOC], line[CLIENTEPRODUTOR_NOME], new Point2D.Double(Double.parseDouble(line[CLIENTEPRODUTOR_LAT
            if (!storeLists.getPessoaList().contains(empresaAux)) {// O(n)
                storeLists.addPessoaList(empresaAux);
            }
        }
        if (line[CLIENTEPRODUTOR_NOME].contains("P")) {// O(n)
            Produtor produtorAux = new Produtor(line[CLIENTEPRODUTOR_LOC], line[CLIENTEPRODUTOR_NOME], new Point2D.Double(Double.parseDouble(line[CLIENTEPRODUTOR_
            if (!storeLists.getPessoaList().contains(produtorAux)) {// O(n)
                storeLists.addPessoaList(produtorAux);
            }
        }
        if (line[CLIENTEPRODUTOR_NOME].contains("C")) {// O(n)
            Cliente clienteAux = new Cliente(line[CLIENTEPRODUTOR_LOC], line[CLIENTEPRODUTOR_NOME], new Point2D.Double(Double.parseDouble(line[CLIENTEPRODUTOR_LAT
            if (!storeLists.getPessoaList().contains(clienteAux)) {// O(n)
                storeLists.addPessoaList(clienteAux);
            }
        }
    }
    return storeLists.getPessoaList();
}// O(n^2)
```

```
86      */
        1 usage  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( regex: ",");
92              Helper helperaux = new Helper(line[DISTANCIAS_UM], line[DISTANCIAS_DOIS], Integer.parseInt(line[DISTANCIAS_TAMANHO
93              if (!storeLists.getLocalidadeList().contains(helperaux)) {// O(n)
94                  storeLists.addLocalidadeList(helperaux);
95              }
96
97          }
98
99          return storeLists.getLocalidadeList();
100     }//O(n)
```

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

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

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

```java
    /**
     * 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) {// O(n)
            if (pessoa.getLocalidade().equals(local)) {
                return pessoa;
            }
        }
        return null;
    }// O(n)
}
```

Explicação:

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

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

Apos serem lidos ambos 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:

```java
/**
 * 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|)
```

```java
/**
 * 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  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();// O(V)

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

```java
/**
 * 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  Jorge_Cunha07 +1
private int biggestSize(ArrayList<LinkedList<Pessoa>> listAux) {
    OptionalInt max = listAux.stream().mapToInt(LinkedList::size).max(); // O(n)

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

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

```java
public List<Map.Entry<Pessoa, Double>> findNearestHub() {
    MatrixGraph<Pessoa, Integer> graph = Algorithms.minDistGraph(mapGraphPessoaStore.getMapGraph(), comparator, binaryOperator);// O(n^3)
    if (graph == null) {
        return null;
    }
    List<Pessoa> vertex = graph.vertices(); // O(V)
    Map<Pessoa, Double> map = new HashMap<>();
    double media;
    int counter;
    // O(n^2)
    for (Pessoa pessoa : vertex) { // O(n)
        media = 0;
        counter = 0;
        if (pessoa instanceof Empresa) {
            for (Pessoa pessoa1 : vertex) { // O(n)
                if (!(pessoa1 instanceof Empresa)) {
                    Edge<Pessoa, Integer> edge = graph.edge(pessoa, pessoa1);// O(1)
                    if (edge != null) {
                        int distance = edge.getWeight();
                        media += distance;
                        counter++;
                    }
                }
            }
            media = media / counter;
            map.put(pessoa, media);
        }
    }
    List<Map.Entry<Pessoa, Double>> list = new ArrayList<>(map.entrySet());
    // O(n)
    list.sort(Comparator.comparing(Map.Entry::getValue));

    return list;
}// 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.

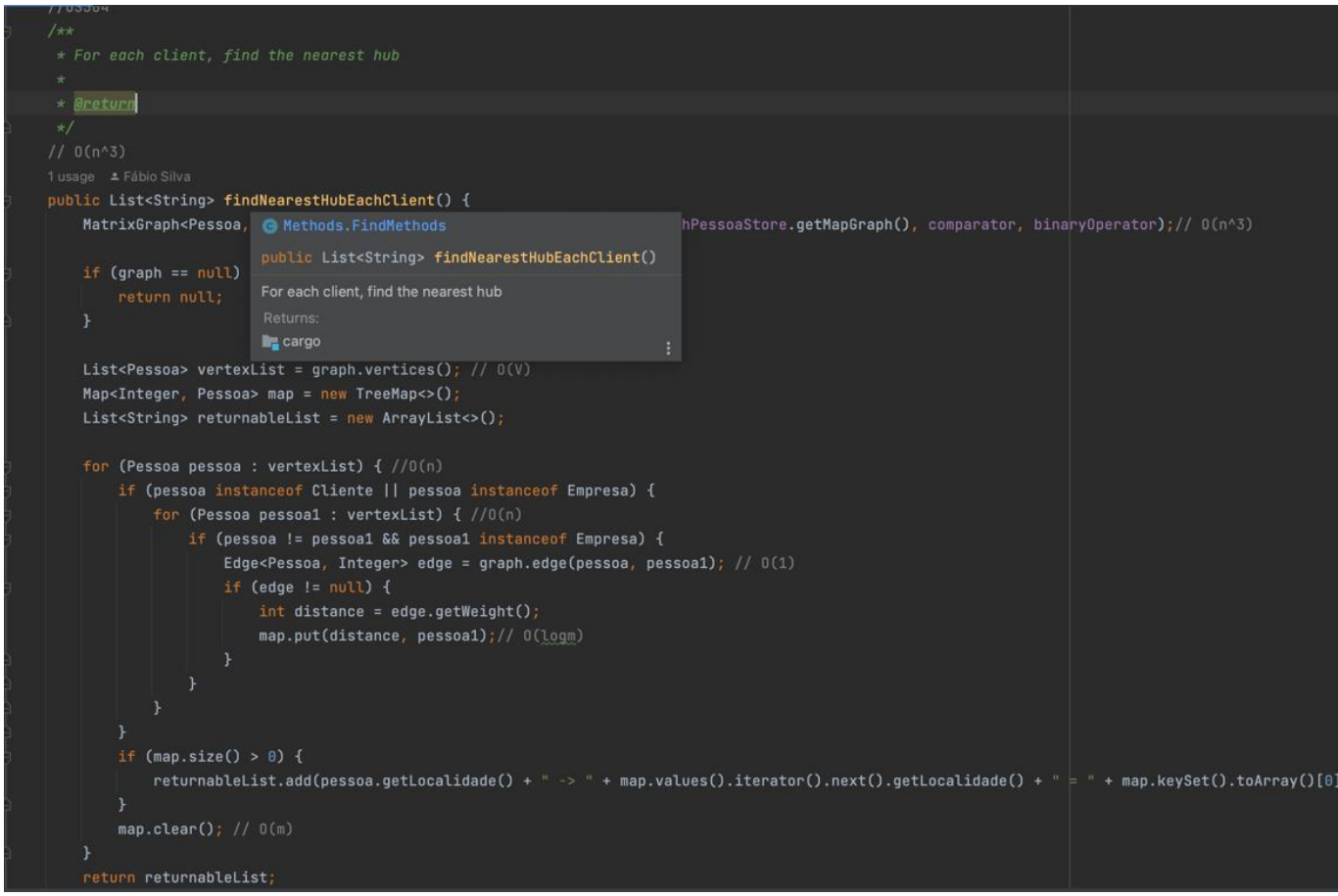
Fazemos um for each para todos os vértices que sejam Empresa para ver as ligações com os outros que não são Empresa.

UC304

Enunciado:

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

Proposta de Solução:



```
/**
 * For each client, find the nearest hub
 *
 * @return
 */
// O(n^3)
1 usage  Fábio Silva
public List<String> findNearestHubEachClient() {
    MatrixGraph<Pessoa,  Methods.FindMethods                                    hPessoaStore.getMapGraph(), comparator, binaryOperator);// O(n^3)
                         public List<String> findNearestHubEachClient()
    if (graph == null)   For each client, find the nearest hub
        return null;     Returns:
    }                    cargo

    List<Pessoa> vertexList = graph.vertices(); // O(V)
    Map<Integer, Pessoa> map = new TreeMap<>();
    List<String> returnableList = new ArrayList<>();

    for (Pessoa pessoa : vertexList) { //O(n)
        if (pessoa instanceof Cliente || pessoa instanceof Empresa) {
            for (Pessoa pessoa1 : vertexList) { //O(n)
                if (pessoa != pessoa1 && pessoa1 instanceof Empresa) {
                    Edge<Pessoa, Integer> edge = graph.edge(pessoa, pessoa1); // O(1)
                    if (edge != null) {
                        int distance = edge.getWeight();
                        map.put(distance, pessoa1);// O(logm)
                    }
                }
            }
        }
        if (map.size() > 0) {
            returnableList.add(pessoa.getLocalidade() + " -> " + map.values().iterator().next().getLocalidade() + " = " + map.keySet().toArray()[0]
        }
        map.clear(); // O(m)
    }
    return returnableList;
```

Explicação:

No método findNearestHubEachClient,logo de inicio 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:

```java
/**
 * It finds the client that is closer to all the other clients
 *
 * @return The person who is closer to all the clients.
 */
1 usage  Fábio Silva
public MatrixGraph<Pessoa, Integer> findNetworkWithMinimumDistance() {
    MapGraph<Pessoa, Integer> graph = mapGraphPessoaStore.getMapGraph();

    if (graph == null) {
        return null;
    }

    List<Pessoa> vertexList = graph.vertices(); // O(V)
    MatrixGraph<Pessoa, Integer> mg = new MatrixGraph<>( directed: false);
    ArrayList<Edge<Pessoa, Integer>> edgeList = (ArrayList<Edge<Pessoa, Integer>>) graph.edges();// O(n^3)

    edgeList.sort(Comparator.comparing(Edge::getWeight)); // O(n)

    for (Pessoa pessoa : vertexList) {// O(n)
        mg.addVertex(pessoa);
    }

    for (Edge<Pessoa, Integer> edge : edgeList) {// O(n)
        Pessoa pessoa = edge.getVOrig();
        LinkedList<Pessoa> connectedVerts = Algorithms.DepthFirstSearch(mg, pessoa); // O(m^2)
        if (!connectedVerts.contains(edge.getVDest())) { // O(m)
            mg.addEdge(edge.getVOrig(), edge.getVDest(), edge.getWeight());
        }
    }
    return mg;
}//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 matrz

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, val: 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.