Relatório Sprint 1

Diogo Araújo – 1200967
João Batista – 1211396
David Dias – 1211415
Ezequiel Estima – 1211417
Marco Andrade - 1211469

Conteúdo

Índice de Imagens

Class Diagram

Em anexo com a submissão deste ficheiro encontra-se o Diagrama de Classes com o nome Class_Diagram.svg

US 301

Nesta US, dado um ficheiro com o formato CSV deveria ser carregada a sua informação, para a construção de um *Graph*. Para esta funcionalidade, criámos o método *loadGraph*, que dadas as listas com informações sobre os membros da rede e sobre os caminhos estabelecidos entre os mesmos, realiza a criação da estrutura pretendida, utilizando duas interfaces privadas para o efeito. O método garante ainda que antes de carregar a *network* (o nosso *Graph*) é limpa:

```
public void loadGraph(List<String> membersInfo, List<String> trackInfo){
    network = new MapGraph<>( directed: false);
    insertVertices(membersInfo);
    insertEdges(trackInfo);
}
```

Figura 1 - Método loadGraph

O método *insertVertices*, como o nome indica, é responsável pela inserção dos vértices no *Graph*, apoiando-se na hierarquia de classes para guardar os diferentes tipos de membro no mesmo:

```
private void insertVertices(List<String> membersInfo) {
    for (String s: membersInfo) {
        String[] values = s.split( regex: ",");
        if(s.charAt(0)!='L') {
            if(values[3].charAt(0)=='C') network.addVertex(new Client(values[0],Double.parseDouble(values[1]),Double.parseDouble(values[2]),values[3]));
            else if(values[3].charAt(0)=='E') network.addVertex(new Company(values[0],Double.parseDouble(values[1]),Double.parseDouble(values[2]),values[3]));
            else network.addVertex(new Producer(values[0],Double.parseDouble(values[1]),Double.parseDouble(values[2]),values[3]));
        }
    }
}
```

Figura 2 - Método insertVertices

Por sua vez, o método *insertEdges*, é responsável pela inserção das *edges* no *Graph*, utilizando o *locId* dos membros como referência para criar a conexão:

```
private void insertEdges(List<String> trackInfo) {
    for (String s: trackInfo) {
        String[] values = s.split( regex: ",");
        if(s.charAt(0)!='L') network.addEdge(new DistributionNetworkMember(values[0]),new DistributionNetworkMember(values[1]) ,new Track(Double.parseDouble(values[2])));
    }
}
```

Figura 3 - Método insertEdges

Para testar a criação do *Graph* foram criadas um conjunto de interfaces que permitem a testagem da estrutura tanto de forma integral como por amostragem sem quebrar os princípios de Information Expert e Encapsulamento:

```java
public boolean integralGraphCheck(List<DistributionNetworkMember> verticesList, List<List<DistributionNetworkMember>> edgesList){
    return checkGraphVertices(verticesList)&checkGraphEdges(edgesList);
}

public boolean checkGraphVertices(List<DistributionNetworkMember> verticesList){
    Graph<DistributionNetworkMember, Track> clonedGraph = network.clone();
    for (DistributionNetworkMember distributionNetworkMember : verticesList) {
        clonedGraph.removeVertex(distributionNetworkMember);
    }
    if(clonedGraph.vertices().isEmpty()) return true;
    return false;
}

public boolean checkGraphEdges(List<List<DistributionNetworkMember>> edgesList){
    Graph<DistributionNetworkMember, Track> clonedGraph = network.clone();
    for (List<DistributionNetworkMember> edge : edgesList) {
        clonedGraph.removeEdge(edge.get(0),edge.get(1));
        clonedGraph.removeEdge(edge.get(1),edge.get(0));
    }
    if(clonedGraph.edges().isEmpty()) return true;
    return false;
}
```

Figura 4 - Métodos para auxiliar testes

Análise da Complexidade

Consideremos um grafo com *V* vértices e *E* edges.

A complexidade temporal de criação de um *MapGraph* será a junção das *V* chamadas do *insertVertices* juntamente com as *E* chamadas do *insertEdges*, o que resulta numa complexidade O(*E*), porque o número de *Edges* será o fator que irá, numa escala muito superior ao número de *Vértices*, ditar quantas execuções esta operação irá ter no pior caso. Sendo que para cada *Vértices* irão existir, comumente, múltiplas *Edges*.

US 302

Na US302 foi pedido para verificar se o grafo era conexo e devolver o número mínimo de ligações necessário para que nesta rede qualquer cliente/produtor conseguir contactar um qualquer outro.

Para conseguir realizar esta US primeiramente foi criado o método *isConex*, que começa por verificar se o grafo é conexo e posteriormente retorna o número mínimo de ligações.

```java
public int isConex(){
    boolean isConex = Algorithms.isConnected(network);
    if(!isConex){
        return -1;
    }
    ArrayList<Integer> shortestpath = new ArrayList<>();
    for (int i = 0;i<network.vertices().size()-1;i++) {
        for (int j = i + 1; j < network.vertices().size(); j++) {
            shortestpath.add(Algorithms.minEdgeBFS(network, network.vertices().get(i), network.vertices().get(j)));
        }
    }
    int diameter = Integer.MIN_VALUE;
    for (int g = 0;g<shortestpath.size();g++){
        if (shortestpath.get(g)> diameter) diameter = shortestpath.get(g);
    }
    return diameter;
}
```

Figura 5 - Método isConnex

Este método utiliza o método *isConnected* da classe *Algorithms,* para verificar se o grafo é conexo, que retorna verdadeiro caso o número de vértices do grafo seja igual ao tamanho da lista retornada pelo algoritmo *DepthFirstSearch.*

```java
public static <V,E> boolean isConnected(Graph<V,E> g){
    if (g == null){
        return false;
    }
    if (g.numVertices() == 0){
        return false;
    }
    LinkedList<V> vertices = Algorithms.DepthFirstSearch(g,g.vertices().get(0));
    return vertices.size() == g.numVertices();
}
```

Figura 6 - Método isConnected

Também utiliza o método *minEdgeBFS* da classe *Algorithms* que chama o método BFSminEdge, que devolve o número de ligações mínimas de um vértice para outro.

```java
public static <V, E> int minEdgeBFS(Graph<V, E> g, V vOrg, V vDes){
    if(g==null||vOrg==null||vDes==null) return 0;
    int[] dist= new int[g.numVertices()];
    if (BFSminEdge(g,vOrg,vDes,dist)) {
        return dist[g.key(vDes)];
    }
    return 0;
}

private static <V, E>  boolean BFSminEdge(Graph<V, E> g, V vOrg, V vDes, int[] dist) {
    LinkedList<V> queue = new LinkedList<>();
    boolean[] visited = new boolean[g.numVertices()];
    Arrays.fill(dist,Integer.MAX_VALUE);

    visited[g.key(vOrg)]= true;
    dist[g.key(vOrg)]=0;
    queue.add(vOrg);

    while (!queue.isEmpty()) {
        V u = queue.remove();
        for (V adj: g.adjVertices(u)){
            if ( !visited[g.key(adj)]){
                visited[g.key(adj)] = true;
                dist[g.key(adj)] = dist[g.key(u)] +1;
                queue.add(adj);

                if (adj.equals(vDes)){
                    return true;
                }
            }
        }
    }
    return false;
}
```

Figura 7 - Método minEdgeBFS

Sendo este método chamado para descobrir o número mínimo de ligações que conectam todos os pares de Vértices do grafo. Por fim damos *return* ao maior número.

Análise da Complexidade

Consideremos um grafo com V vértices e E edges.

Nesse método, podemos observar que o bloco de código que vai ditar qual a complexidade vai ser do bloco *for* que será de $\mathrm{O}(\ \mathrm{C}(\mathrm{V}|2) \times \mathrm{V} + \mathrm{E})$.

US 303

Na US303 é pedido para definir os N hubs mais próximos de todos os clientes e produtores agrícolas da rede, com medida de proximidade sendo a média de todos os caminhos mais curtos do hub a cada cliente e produtor.

Primeiro obtemos todos os vértices *DistributionNetworkMember* do nosso grafo com a função *network.vertices()*, os quais representam todos os membros da nossa rede de distribuição. Filtramos então os membros desta rede por empresas, com o operador if verificando se é uma instância da classe *Company*, e se for, adicionar ao *ArrayList vOrigins* do tipo *DistributionNetworkMember*.

```
9 usages   1211415 +1
public List<DistributionNetworkMember> defineHubs(int n){
    hubs=new ArrayList<>();
    if(n<=0){
        return null;
    }
    ArrayList<DistributionNetworkMember> vOrigins = new ArrayList<>();
    ArrayList<LinkedList<DistributionNetworkMember>> paths = new ArrayList<>();
    ArrayList<Track> dists = new ArrayList<>();
    for(DistributionNetworkMember v : network.vertices()){
        if(v instanceof Company){
            vOrigins.add(v);
        }
    }
```

Figura 8 – 1º Parte método defineHubs

Depois, para cada empresa guardada no *ArrayList vOrigins*, chamamos o método *shortestPaths* da classe *Algorithms* e obtemos as distâncias mínimas dessa empresa (*Company*) para todas os clientes (*Clients*), empresas e produtores (*Producer*) do nosso grafo (*DistributionNetwork*). Depois em cada iteração fazem-se as médias de todos os caminhos de menor custo e guardam se num novo *ArrayList* de médias de caminhos de menor custo.

No caso de existir uma empresa sem caminhos de menor custo para nenhum cliente, empresa ou produtor no grafo, é registada a sua posição no array e posteriormente removida do *vOrigins*.

```java
ArrayList<Double> results = new ArrayList<>();
double proximidade = 0;
double shortPath=0;
int num=0;
int k=0;
ArrayList<Integer> remove = new ArrayList<>();
for (DistributionNetworkMember vOrigin : vOrigins) {

    if (Algorithms.shortestPaths(network, vOrigin, trackComparator, binaryOperator, zero, paths,dists)){
        for (int i = 0; i < dists.size(); i++) {
            if(dists.get(i)!=null) {
                shortPath = dists.get(i).getMiles();
                proximidade = proximidade + shortPath;
                if(shortPath!=0) {num++;}
            }
        }

        if (num!=0) {
            proximidade = proximidade / num;
            results.add(proximidade);
        }
        else {remove.add(k);}
        num=0;
        proximidade=0;

    }
    k++;
}

for (int i = 0; i < remove.size(); i++) {
    k = remove.get(i);
    vOrigins.remove(k);
}
```

Figura 9 - 2º parte método defineHubs

Para obtermos os caminhos de menor custo foi implementado o algoritmo *shortestPaths*, que computa todos os caminhos de menor custo entre um vértice e todos os vértices do grafo, desta forma obtemos todas as distâncias mínimas para todos os vértices conectados para esse dado vértice.

```java
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;}

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

    int numVerts = g.numVertices();
    boolean [] visited = new boolean[numVerts];
    @SuppressWarnings("unchecked")
    V[] pathKeys = (V[]) new Object[numVerts];
    @SuppressWarnings("unchecked")
    E[] dist = (E[]) new Object[numVerts];

    initializePathDist(numVerts,pathKeys,dist);
    shortestPathDijkstra(g,vOrig,ce,sum,zero,visited,pathKeys,dist);
    dists.clear();
    paths.clear();

    for (int i = 0; i < numVerts; i++) {
        paths.add(null);
        dists.add(null);
    }
    for (V vDest: g.vertices()) {
        int key = g.key(vDest);
        if (dist[key]!=null){
            LinkedList<V> shortPath = new LinkedList<>();
            getPath(g,vOrig,vDest,pathKeys,shortPath);
            paths.set(key,shortPath);
            dists.set(key,dist[key]);
        }
    }
    return true;
}
```

Figura 10 - Método shortestPaths

Uma vez obtidas as médias dos caminhos mais curtos e as respetivas empresas que possuem essas médias, ordenamos as empresas em função das médias de forma descendente e depois adicionamos estas a uma lista de *hubs* do tipo *List* que atualiza os elementos dentro do *DistributionNetwork* e retorna essa lista.

```java
        double temp = 0;
        DistributionNetworkMember tempCompany;

        for(int i = 0 ; i<results.size(); i++){
            for(int j = i; j<results.size(); j++){
                if(results.get(i)>results.get(j)){
                    temp = results.get(i);
                    results.set(i, results.get(j));
                    results.set(j, temp);
                    tempCompany = vOrigins.get(i);
                    vOrigins.set(i,vOrigins.get(j));
                    vOrigins.set(j,tempCompany);
                }
            }
        }
        if (n>vOrigins.size()){n= vOrigins.size();}
        for (int i = 0; i <n ; i++) {
            hubs.add(vOrigins.get(i));
        }

        return hubs;
    }
}
```

Figura 11 - 3º Parte método defineHubs

Analise de Complexidade

Se consideramos N como o número de *hubs*, V como o número de vértices no grafo e E como o número de *edges*, então:

Na função *defineHubs* é chamado o Algoritmo de *shortestPaths* V vezes, como este algoritmo chama a função de *Dijkstra* de complexidade temporal O $(V^2 + E)$ sendo o processo com maior complexidade dentro do algoritmo do qual este depende, então este terá essa mesma complexidade. $O\ (V^2 + E + 2V)\ =\ O\ (V^2)$

Para cada V ao qual pode-se aplicar o algoritmo *shortestPaths*, obtêm-se as médias dos caminhos para todos os vértices do grafo. Ficaria uma complexidade temporal no pior caso de O $((V^2 + E) \times V)\ =\ O\ (V^3 + VE)$

No caso de ordenar os vértices, eliminar vértices e inseri-los na lista de *hubs* o pior caso de complexidade temporal respetiva para cada uma delas é uma complexidade polinomial de primeiro grau, pelo que em resumo, a função *defineHubs* terá uma complexidade do tipo $O\ (V^3 + VE)$

US304

Foi pedido nesta US que fosse determinada a *Hub* mais próximo de uma de um cliente (particular ou empresa). Para tal foi criada uma função chamada *getClosestHub().*

Essa função retorna um *map* que em que as *keys* são os vértices do grafo, exceto os já *hubs* determinados, e o *value* associado é o *hub* mais próximo.

```java
1211469
public Map<DistributionNetworkMember, DistributionNetworkMember> getClosestHub(){
    LinkedList<DistributionNetworkMember> shortPath = new LinkedList<>();
    double size;
    Map<DistributionNetworkMember, DistributionNetworkMember> closestHub= new HashMap<>();
    for(DistributionNetworkMember v:network.vertices()){
        double min = Double.MAX_VALUE;
        DistributionNetworkMember vMin = null;
        if (!hubs.contains(v)){
            for(DistributionNetworkMember hub : hubs){
                Track shortPathEdge = Algorithms.shortestPath(network,v,hub,trackComparator,binaryOperator,zero,shortPath);
                if(shortPathEdge != null){
                    size = shortPathEdge.getMiles();
                    if(size < min){
                        min = size;
                        vMin = hub;
                    }
                    closestHub.put(v,vMin);
                }
            }
        }
    }
    if(closestHub.isEmpty()){
        return null;
    }else{
        return closestHub;
    }
}
```

Figura 12 - Método getClosestHub

Essa função tira partido da função *shortestPath()* que encontra o caminho mais curto entre um vértice origem e um vértice destino e retorna o seu peso, para tal é utilizado o algoritimo de Dijkstra que está implementado na função *shortestPathDijkstra().*

```java
1211469
public static <V, E> E shortestPath(Graph<V, E> g, V vOrig, V vDest,
                                    Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                    LinkedList<V> shortPath) {
    if (!g.validVertex(vOrig) || !g.validVertex(vDest)) {
        return null;
    }

    shortPath.clear();
    int numVerts = g.numVertices();
    boolean[] visited = new boolean[numVerts];
    V[] pathKeys = (V[]) new Object[numVerts];
    E[] dist = (E[]) new Object[numVerts];
    initializePathDist(numVerts, pathKeys, dist);

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

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

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

Figura 13 - Método shortestPath

```
2 usages  1211469 +1
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) {
        vkey = g.key(vOrig);
        visited[vkey] = true;
        for (Edge<V, E> edge : g.outgoingEdges(vOrig)) {
            int vkeyAdj = g.key(edge.getVDest());
            if (!visited[vkeyAdj]) {
                E s = sum.apply(dist[vkey], edge.getWeight());
                if (dist[vkeyAdj] == null || ce.compare(dist[vkeyAdj], s) > 0) {
                    dist[vkeyAdj] = s;
                    pathKeys[vkeyAdj] = vOrig;
                }
            }
        }
        E minDist = null;
        vOrig = null;
        for (V vert : g.vertices()) {
            int i = g.key(vert);
            if (!visited[i] && (dist[i] != null) && ((minDist == null) || ce.compare(dist[i], minDist) < 0)) {
                minDist = dist[i];
                vOrig = vert;
            }
        }
    }
}
```

Figura 14 - Método shortestPathDijkstra

Esta implementação do Algoritimo de Dijkstra não tem retorno, no entanto atualiza a LinkedList que é composto pelas distâncias mínimas de um certo vértice até os outros vértices do grafo, o array *visited* que indica se um certo vértice já foi visitado e o array *pathKeys* que indica o path seguido de um vértice origem a um destino com o peso mínimo possível. Para além destes parâmteros existem o *graph*, o *vOrig* por onde é passado o vértice origem, o ce que é um comparator para poder comparar as *Edges* e o *Binary Operator* que é utilizado para poder realizar somas com os pesos.

O algoritimo de Dijkstra é um algortimo "greedy", ou seja, prioriza sempre as edges com distâncias mínimas o que pode signifcar que resultar não é sempre o mais correto, no entanto é mais eficiente do que um algoritimo que verificasse todos os caminhos.

O que acontece neste algoritimo é que para cada vértice que está a anlisar (*vOrig*) coloca-o como visitado, vê as *edges* que estão conectadas e verifica se é possível diminuir a distância para esse vértice adjacente. Depois vai ver se existe ainda algum vértice por visitar, se existir vai para o que ainda não foi visitado e com a distância mais curta.

Análise de complexidade

Sendo V o número de vértices e H o número de *hubs*. Tanto o método que chama o Dijkstra como o próprio têm uma complexidade temporal $O(V^2)$. O método do *getClosestHub()* tem uma complexidade $O(V^3 \times H)$, porque para além de chamar o método de complexidade $O(V^2)$ é percorrido para cada vértice, isto porque percorre todos os vértices não *hubs* e para além desse também percorre os *hubs* para cada vértice não *hub*, assim acabando por visitar todos os vértices. Até então este método teria complexidade $O(V^3)$, mas para não utilizar o dijsktra para todos os vértices existentes no grafo existe uma verficação se ele é *hub* ou não, e essa verficação ocorre com um *contains* de um *ArrayList* com todos os hubs to grafo, então podemos concluir que a complexidade total do método *getClosestHub()* no pior caso é $O(V^3 \times H)$.

O algorítimo de Dijkstra tem uma complexidade temporal $O(V^2)$ para o pior caso pois percorre para cada vértice todos os vértices.

US305

Na US305 é pedido para determinar a rede conecta todos os clientes e produtores agrícolas com uma distância total mínima.

A árvore geradora de custo mínimo de um grafo é o subgrafo acíclico e conexo que contem todos os vértices do grafo inicial, e tem a menor soma de pesos das suas arestas. Como, para o problema em questão, temos um grafo em que os vértices são os clientes/produtores e as arestas são as distâncias entre eles, basta aplicar um algoritmo para calcular a árvore geradora de menor custo do grafo inicial.

Para isso foi implementado o algoritmo de Kruskal:

```
public static <V,E> Graph<V,E> minimumSpanningTreeKruskal(Graph<V,E> g , Comparator<E> cmp){
    if (g == null || g.isDirected() || !Algorithms.isConnected(g)){
        return null;
    }

    Graph<V,E> mst = new MatrixGraph<>( directed: false);
    ArrayList<Edge<V, E>> edgeList = new ArrayList<>(g.edges());
    for (V vert : g.vertices()){
        mst.addVertex(vert);
    }
    edgeList.sort( (e1,e2) -> cmp.compare(e1.getWeight(), e2.getWeight()) );

    for (Edge<V, E> edge : edgeList){
        LinkedList<V> connectedVerts = DepthFirstSearch(mst,edge.getVOrig());
        if (connectedVerts !=  null) {
            if (!connectedVerts.contains(edge.getVDest())){
                mst.addEdge(edge.getVOrig(),edge.getVDest(), edge.getWeight());
            }
        }
    }
    return mst;
}
```

Figura 15 - Método minimumSpanningTreeKruskal

Primeiro é feita a validação do grafo inicial: verificar se não é direcionado e é conexo. Depois todos os vértices do grafo inicial são colocados no grafo resultante e todas as arestas são colocadas num *ArrayList*, por ordem crescente dos seus pesos. Finalmente, para todas as arestas, é realizada uma visita em profundidade pelo seu vértice de origem e, se esta aresta liga dois vértices que ainda não estão conectados, ela é adicionada ao grafo resultante.

Depois de implementado o algoritmo, basta aplicá-lo ao grafo *network* para obter a solução pretendida.

```
public Graph<DistributionNetworkMember, Track> minimumSpanningNetwork(){
    return Algorithms.minimumSpanningTreeKruskal(network,trackComparator);
}
```

Figura 16 - Método minimumSpanningNetwork

Análise de complexidade

Consideremos um grafo com *V* vértices e *E* arestas.

No método desenvolvido, a complexidade será dependente da complexidade do método *minimumSpanningTreeKruskal* (já que apenas chama esse método).

Nesse método, podemos observar que o bloco de código que vai ditar qual a complexidade vai ser o bloco *for each* que percorre todas as arestas, realizando uma visita em profundidade para cada iteração.

Como a visita em profundidade tem complexidade $O(V + E)$, a complexidade final do algoritmo é $O((V + E) \times E)$.