Estruturas de Informação

Sprint 2

Feito por:

André Ferreira, 1190378

Bernardo Barbosa, 1220741

Diogo Cunha, 1221071

Tomás Peixoto,1221948

Professores:

Alberto Sampaio (ACS)

20/11/23

Índice

Glossário

Aresta: Ligação entre dois vértices em um grafo, representando a relação entre eles.

Algoritmo de Kruskal: Algoritmo utilizado para encontrar a árvore geradora de custo mínimo em um grafo.

Cabaz: Cesto com conjunto de produtos agrícola colhidos (ex tomates, Cenouras).

Caminho de Menor Custo: Rota entre dois vértices em um grafo que possui o menor custo possível.

Complexidade Temporal: Uma medida do número de operações ou recursos necessários para executar um algoritmo, geralmente expressa em termos de tempo (melhor caso, pior caso).

Grafo: Estrutura que representa um conjunto de vértices e arestas que os conectam.

GFH: Operador logístico responsável pela distribuição de cabazes com produtos agrícolas em rede.

Hub: Instituição ou empresa onde são entregues e posteriormente levantados os cabazes.

Localidade: Representação de uma área geográfica onde podem existir hubs de distribuição.

Rede de Distribuição: Estrutura constituída por vários vértices representativos de localidades, onde podem existir hubs, e arestas representativas das distâncias entre essas localidades.

Veículo Elétrico: Meio de transporte utilizado para transportar os cabazes, com autonomia limitada e disponível nos hubs para posterior recolha.

Vértice: Ponto fundamental em um grafo que representa uma entidade ou local.

Introdução

Neste relatório, pretendemos apresentar a logística contida na resolução de cada uma das funcionalidades propostas para o segundo *Sprint* 2 do **Projeto Integrador**, no contexto da unidade curricular **Estruturas de Informação**.

O principal objetivo durante as nossas resoluções foi a aplicação dos conceitos e técnicas aprendidas nessa mesma unidade curricular (nomeadamente, os grafos), mantendo as boas práticas de programação orientada a objetos (*OO Programming*).

O grupo pretendeu obter a máxima compreensão e simplicidade ao longo do relatório, mantendo a sua eficácia. Para alcançar este objetivo, foi criado um diagrama de classes.

Cada *User Story* contido neste relatório tem um título, figura (ou figuras) e uma breve, mas concisa explicação de como o algoritmo funciona, resolvendo a questão proposta. A análise de complexidade para cada funcionalidade também é incluída.

Por fim, o relatório apresenta as fontes das quais foram adquiridas conhecimento utilizado para a resolução e melhor compreensão to trabalho.

Diagrama de Classes

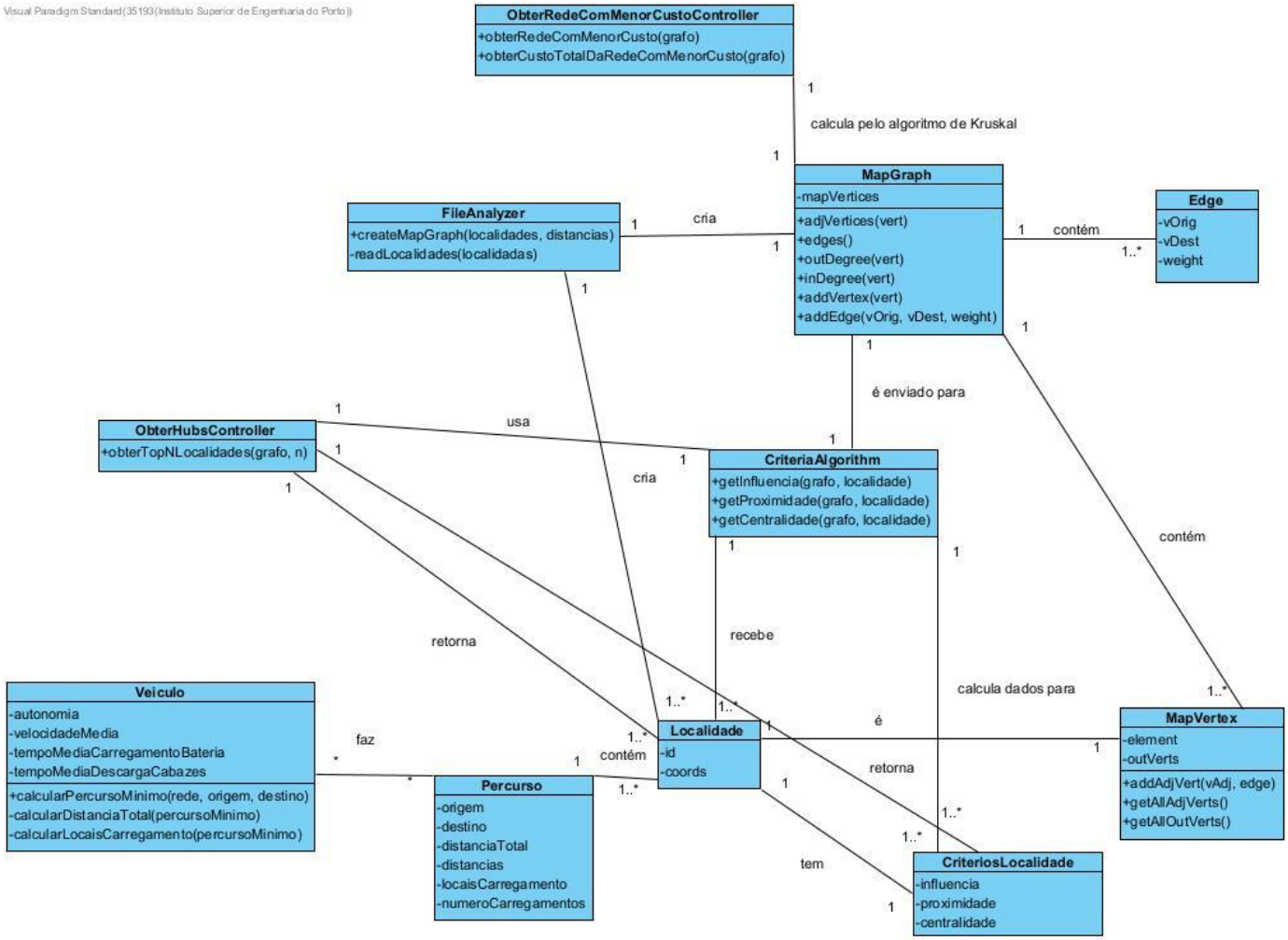

Figura 1. Diagrama de Classes

Aqui, vemos o diagrama de classes elaborado para esta secção do projeto. As várias classes e relações estabelecidas estão apresentadas aqui.

As interações entre o ***MapGraph*** e os objetos de domínio, o ***FileAnalyzer*** e os *Controllers* são de alta importância para a nossa resolução. Cada uma destas relações será devidamente elaborada na sua respetiva secção.

USEI01 – Construir Rede de Distribuição

(Responsável:Diogo Cunha)

Explicação da Resolução

```
public static MapGraph<Localidade, Integer> createMapGraph(File localidadesPath, File distanciasPath) throws FileNotFoundException{
    MapGraph<Localidade, Integer> graph = new MapGraph<>( directed: false);
    List<Localidade> localidades = readLocalidades(localidadesPath);
    for(Localidade localidade : localidades)
        graph.addVertex(localidade);
    Scanner fileReader = new Scanner(distanciasPath);
    fileReader.nextLine(); //ignore Header
    while(fileReader.hasNext()){
        String[] itemsPerLine = fileReader.nextLine().split( regex: ",");
        graph.addEdge(Utils.getLocalidadeById(itemsPerLine[0], localidades),
                Utils.getLocalidadeById(itemsPerLine[1], localidades), Integer.parseInt(itemsPerLine[2]));
    }
    return graph;
}
```

*Figura 2. Método **createMapGraph***

Este método tem como objetivo ler os ficheiros fornecidos pelo utilizador, o das localidades e o das distâncias, colocando os locais como vértices e as distâncias entre locais como arestas dum ***MapGraph***.

Foi escolhido o ***MapGraph*** em vez da ***MatrixGraph*** visto que a primeira é mais eficiente em vários aspetos importantes para os nossos objetivos. Ambas as opções têm as suas vantagens, mas decidimos que as do ***MapGraph*** são mais favoráveis.

Para alcançar este objetivo, é criado um ***MapGraph*** não direcionado. O método recebe uma lista de localidades através do método ***readLocalidades*** (explicado depois deste), todos estes locais são adicionados ao grafo como um vértice. Depois disto, o método irá ler o ficheiro das distâncias, criando uma *Edge* com ambos os locais contidos na mesma para todas as linhas do ficheiro mencionado.

```
private static List<Localidade> readLocalidades(File localidadesPath) throws FileNotFoundException {
    Scanner fileReader = new Scanner(localidadesPath);
    fileReader.nextLine(); //ignore header
    List<Localidade> localidades = new ArrayList<>();
    while(fileReader.hasNext()){
        String[] itemsPerLine = fileReader.nextLine().split( regex: ",");
        localidades.add(new Localidade(itemsPerLine[0],
                new Point2D.Double(Double.parseDouble(itemsPerLine[1]), Double.parseDouble(itemsPerLine[2]))));
    }
    return localidades;
}
```

*Figura 3. Método **readLocalidades***

Este método analisa o ficheiro com as localidades, passando linha a linha do método, criando um objeto "Localidade" e adicionando-o à lista de localidades que será retornada no fim do mesmo.

Análise de Complexidade

Começando pelo método ***readLocalidades***, vemos que este método tem uma ***Worst-Case Time Complexity*** linear, ou seja, **O(n)** (**n** sendo o número de locais no ficheiro). Isto deve-se ao facto que o método terá de atravessar todas os locais, adicionando-os à lista que será retornada.

Já o método ***createMapGraph*** tem uma ***Worst-Case Time Complexity*** de **O(V * E)** (**V** sendo o número de locais e **E** o número de caminhos entre locais). Isto deve-se ao facto que, no meio de um *loop* que passa por todas as linhas do ficheiro com as distâncias (daí o **V**), o método ***getLocalidadeById***, que atravessa a lista inteira de locais até encontrar o que tem o *id* igual ao que procuramos atualmente (daí o **E**).

No estudo de grafos, esta complexidade trata-se duma **Complexidade Linear**.

USEI02:

(Responsável: Tomás Peixoto)

Explicação da Resolução

```java
public static Map<CriteriosLocalidade, Localidade> obterTopNLocalidades(Graph<Localidade, Integer> grafo, int n){
    List<Localidade> localidades = grafo.vertices();
    Comparator<CriteriosLocalidade> cmp = (o1, o2) -> {
        int centralidadeDiff = o2.getCentralidade() - o1.getCentralidade();
        if(centralidadeDiff != 0)
            return centralidadeDiff;
        else
            return o2.getInfluencia() - o1.getInfluencia();
    };
    Map<CriteriosLocalidade, Localidade> map = new TreeMap<>(cmp);
    for(Localidade localidade : localidades){
        map.put(new CriteriosLocalidade(CriteriaAlgorithm.getInfluencia(grafo,localidade),
                CriteriaAlgorithm.getProximidade(grafo,localidade),
                CriteriaAlgorithm.getCentralidade(grafo,localidade)), localidade);
    }
    if(n > map.size())
        n = map.size();
    Iterator<Map.Entry<CriteriosLocalidade, Localidade>> iterator = map.entrySet().iterator();
    int count = 0;
    while (iterator.hasNext()) {
        iterator.next();
        if (count >= n) {
            iterator.remove();
        }
        count++;
    }
    return map;
}
```

*Figura 4. Método **obterTopNLocalidades***

Nesta *User Story*, o método principal é o ***obterTopNLocalidades***, que recebe um grafo e um inteiro *n* como parâmetros.

A partir do grafo recebido, o método calcula os critérios de **influência**, **proximidade** e **centralidade** para todos os vértices do grafo. Desta forma, o programa poderá colocá-los numa ***TreeMap*** e ordená-la por ordem decrescente da **centralidade** e da **influência**, como foi pedido.

Posteriormente, o método irá eliminar todos os membros da ***TreeMap*** exceto os *n* primeiros, retornando o objeto resultante.

Nota: Foi decidido o uso duma ***TreeMap*** aqui pelo facto que permite retornar dois objetos de tipos diferentes numa lista, ordenada pelo critério que o programador estabelecer.

```java
public static int getInfluencia(Graph<Localidade, Integer> grafo, Localidade local){
    return grafo.outDegree(local) + grafo.inDegree(local);
}
```

*Figura 5. Método **getInfluencia***

Este método visa retornar o valor de **influência** de uma localidade, dentro do contexto do grafo onde pertence.

Para alcançar este objetivo, retornamos o seu grau de entrada somado com o seu grau de saída, ou seja, *2E* (visto que este grafo é não direcionado).

```java
public static int getProximidade(Graph<Localidade, Integer> grafo, Localidade local){
    int proximidade = 0;
    List<Localidade> vertices = grafo.vertices();
    for (int i = 0; i < vertices.size() - 1; i++)
        if(!vertices.get(i).equals(local)) {
            double shortestDifference = Double.MAX_VALUE;
            Localidade shortestDifferenceLocalidade = null;
            for(int j = i + 1; j < vertices.size(); j++){
                double difference = Utils.haversineDistance(vertices.get(i).getCoords(), vertices.get(j).getCoords());
                if(difference < shortestDifference || (difference <= shortestDifference && Objects.equals(shortestDifferenceLocalidade, local))){
                    shortestDifference = difference;
                    shortestDifferenceLocalidade = vertices.get(j);
                }
            }
            if(shortestDifferenceLocalidade.equals(local))
                proximidade++;
        }
    return proximidade;
}
```

*Figura 6. Método **getProximidade***

Este método visa retornar o valor de **proximidade** de uma localidade, dentro do contexto do grafo onde pertence.

Para alcançar este objetivo, atravessamos todos os vértices do grafo, calculando a distância entre cada um desses vértices (excluindo o que enviamos como parâmetro) com os outros. Se o vértice com menor distância ao que estamos a analisar for o que enviamos como parâmetro, incrementamos o valor que vamos retornar, a variável **proximidade**.

```java
public static int getCentralidade(Graph<Localidade, Integer> grafo, Localidade local){
    int centralidade = 0;
    ShortestPathMapGraph<Localidade, Integer> shortestPathMaker = new ShortestPathMapGraph<>();
    List<Localidade> localidades = grafo.vertices();
    for(int i = 0; i < localidades.size() - 1; i++)
        for(int j = i + 1; j < localidades.size(); j++)
            for(Edge<Localidade, Integer> edge : shortestPathMaker.getShortestPath(grafo, localidades.get(i), localidades.get(j)))
                if(edge.getVOrig().equals(local) || edge.getVDest().equals(local))
                    centralidade++;
    return centralidade / 2;
}
```

*Figura 7. Método **getCentralidade***

Este método visa retornar o valor de **centralidade** de uma localidade, dentro do contexto do grafo onde pertence.

Para alcançar este objetivo, teremos de calcular todos os caminhos de menor custo entre todas as combinações de dois vértices, incrementando a variável de retorno **centralidade** quando encontramos um caminho que inclui o vértice que passamos como parâmetro.

A necessidade de calcular os caminhos mais curtos entre todas as combinações de dois vértices torna este método bastante custoso numa questão de consumo de tempo.

Análise de Complexidade

Começando pelo método ***getInfluencia***, vemos que este método tem uma ***Worst-Case Time Complexity*** de **O(1) / O(VxE)** de acordo com os *slides* oficiais da unidade curricular relativas a grafos (Imagem mostrada na secção "**Restrições e Observações**").

O método ***getProximidade*** terá um custo mais elevado, visto que teremos de calcular o vértice mais próximo de todos os vértices. Visto que este método tem dois *loops* que atravessam os vértices do grafo (nomeadamente, os *for* começados nas linhas 19 e 23). Isto dá uma ***Worst-Case Time Complexity*** de **O(V^2)**.

O método **getCentralidade** será o mais custoso destes três, devido à natureza do critério. Vendo que temos de calcular o caminho mínimo entre quaisquer dois vértices do grafo (daí os dois *for* nas linhas 39 e 40), e para cada caminho mínimo, ver se o vértice que passamos como parâmetro está contido (daí o *for* na linha 41), podemos concluir que este método tem uma ***Worst-Case Time Complexity*** de **O(V^2 x E)**.

Já o método principal, o ***obterTopNLocalidades*** tem uma ***Worst-Case Time Complexity*** de **O(V^3xE)**. Isto deve-se ao facto que o método tem de calcular os três critérios anteriormente mencionados para todos os vértices do grafo para poder fazer o seu objetivo (daí o *for* começado na linha 21 do seu método). Um desses critérios é a **centralidade**, por isso vamos ter de multiplicar essa complexidade anteriormente mencionada pelo número de vértices, obtendo uma ***Worst-Case Time Complexity*** final de **O(V^3xE)**.

USEI03

(Responsável: André Ferreira)

Método Principal

```
82      public Percurso calcularPercursoMinimo(MapGraph<Localidade, Integer> redeDistribuicao) {
83          Point2D.Double[] furthestPoints = Utils.findFurthestPoints(getCoordinates(redeDistribuicao));
84          Localidade origem = Localidade.findLocalidadeByCoordinates(redeDistribuicao, furthestPoints[0]);
85          Localidade destino = Localidade.findLocalidadeByCoordinates(redeDistribuicao, furthestPoints[1]);
86          ShortestPathMapGraph<Localidade, Integer> shortestPathAlgorithm = new ShortestPathMapGraph<>();
87          List<Edge<Localidade, Integer>> percursoMinimo = shortestPathAlgorithm.getShortestPath(redeDistribuicao, origem, destino);
88          int distanciaTotal = calcularDistanciaTotal(percursoMinimo);
89          List<Localidade> locaisCarregamento = calcularLocaisCarregamento(percursoMinimo);
90          return new Percurso(origem, destino, distanciaTotal, percursoMinimo, locaisCarregamento);
91      }
92
```

*Figura 8. Método **calcularPercursoMinimo***

O método principal usado para a resolução desta US é o ***calcularPercursoMinimo***. Devolve um objeto **Percurso** associado ao veículo que contêm as informações pedidas no enunciado relativamente ao percurso mínimo, nomeadamente:

- a origem,
- o destino,
- a distância total,
- as distâncias individuais,
- os locais de carregamento,
- o número de carregamentos;

O método recebe como parâmetros a rede de distribuição como um ***MapGraph*** de **Localidade** e **Integer**. Isto é representa um grafo das localidades e distâncias entre as mesmas. Recebe também a origem e o destino a calcular o percurso mínimo.

Execução do Algoritmo

- Linha 83 - Calcula os pontos mais afastados da rede de distribuição e retorna num *array* de tamanho 2.
- Linha 84 – Encontra a localidade do ponto de origem.
- Linha 85 – Encontra a localidade do ponto destino
- Linha 86 – Inicia uma instância do algoritmo que calcula o caminho mais curto.
- Linha 87 – Inicia e preenche uma lista das distâncias do caminho mais curto chamando o algoritmo a partir da instância iniciada na linha anterior e passando como parâmetros os mesmos do método.
- Linha 88 – Chama um método auxiliar para calcular a distância total do percurso.
- Linha 89 – Chama um método auxiliar para calcular os locais de carregamento durante o percurso.
- Linha 90 – Retorna uma instância **Percurso** com os *fields* preenchidos.

Métodos Auxiliares

```java
    private List<Point2D.Double> getCoordinates(MapGraph<Localidade, Integer> redeDistribuicao) {
        List<Localidade> localidadesList = redeDistribuicao.vertices();
        List<Point2D.Double> points = new ArrayList<>();
        for (Localidade current : localidadesList) {
            points.add(current.getCoords());
        }
        return points;
    }

    1 usage   andredsferreira *
    private int calcularDistanciaTotal(List<Edge<Localidade, Integer>> percursoMinimo) {
        Integer total = 0;
        for (Edge distancia : percursoMinimo) {
            Integer current = (Integer) distancia.getWeight();
            total += current;
        }
        return total;
    }

    private List<Localidade> calcularLocaisCarregamento(List<Edge<Localidade, Integer>> percursoMinimo) {
        List<Localidade> locaisCarregamento = new ArrayList<>();
        double autonomiaTemp = getAutonomiaMetros();
        for (Edge<Localidade, Integer> distancia : percursoMinimo) {
            if (autonomiaTemp - distancia.getWeight() < 0) {
                locaisCarregamento.add(distancia.getVOrig());
                autonomiaTemp = getAutonomiaMetros();
            } else {
                autonomiaTemp -= getAutonomiaMetros();
            }
        }
        return locaisCarregamento;
    }
```

*Figura 9. Métodos **getCoordinates**, **calcularDistanciaTotal** e **calcularLocaisCarregamento***

Análise de Complexidade

Os métodos auxiliares têm todos uma complexidade linear para o pior caso. No melhor caso, têm complexidade constante, mas isto significará que o veículo não saiu do sítio. A análise é relativamente simples. Para o método ***calcularDistanciaTotal***, recebemos uma lista com as distâncias do percurso mínimo e percorremo-la o que dita a complexidade linear. Para o método ***calcularLocaisCarregamente***, recebemos a mesma lista e calculamos os locais de carregamento pelo que apresenta também complexidade linear.

A complexidade do algoritmo para calcular o percurso mínimo, uma vez que utiliza o algoritmo de ***Dijkstra***, é quadrática. Sendo que depende do número de vértices do grafo da rede de distribuição, isto é, do número de localidades.

Concluindo, o método apresenta uma complexidade de *n* elevado a 5.

USEI04 - Determinar a rede de todas as localidades com menor distância.

(Responsável: Bernardo Barbosa)

Método Principal: obterRedeComMenorCusto:

```
public static Graph <Localidade,Integer> obterRedeComMenorCusto(Graph<Localidade,Integer> grafoObtido){
   Comparator <Edge <Localidade,Integer>> comparator = Comparator.comparingInt(Edge::getWeight);

   return Kruskal.kruskall(grafoObtido,comparator);
  }
```

*Figura 10. Método **obterRedeComMenorCusto***

O método ***obterRedeComMenorCusto*** é responsável por calcular a rede (árvore geradora de menor custo) em um grafo ponderado, utilizando o algoritmo de ***Kruskal***. O método recebe um grafo do tipo ***Graph<Localidade, Integer>*** em que, as localidades são interpretadas para o âmbito da resolução do exercício como vértices, e as distâncias como arestas.

O método cria um objeto do tipo comparador do tipo ***Edge***, uma vez que o algoritmo de ***Kruskal*** desenvolvido recebe um como parâmetro. Este irá comparar os pesos de cada aresta.

Por fim o método retorna a arvore geradora mínima obtida a partir do algoritmo ***Kruskal***.

No entanto este apenas retornada o grafo da árvore geradora mínima, e, o exercício pedido requere não só o grafo e os seus vértices e arestas (localidades e distâncias) mas também a **distância total de rede** que se encontra em falta aqui.

Desta forma foi criado o método secundário para a sua obtenção:

Método Secundário: obterCustoTotalDeRedeComMenorCusto:

```
public static int obterCustoTotalDaRedeComMenorCusto(Graph <Localidade,Integer> grafoObtido) {
    int custoTotal = 0;
    for (Edge <Localidade,Integer> aresta : grafoObtido.edges())
        custoTotal += aresta.getWeight();


    return  custoTotal/2;
}
```

*Figura 11. Método **obterCustoTotalDeRedeComMenorCusto***

O método ***obterCustoTotalDaRedeComMenorCusto*** calcula o **custo total** da árvore geradora de menor custo obtida pelo método principal. Este recebe a árvore geradora mínima (**grafoObtido**) e retorna o custo total.

Primeiramente o método inicia o valor de custo total como zero caso os custos sejam nulos. De seguida, é realizado um ciclo em que, por cada aresta existente no **grafoObtido**, será somado o peso ao custo total.

O custo total é dividido por dois tendo em conta uma condição de implementação do especificada em **restrições e observações**.

Algoritmo De *Kruskal*:

O algoritmo de ***Kruskal*** foi o algoritmo escolhido pelo grupo para encontrar a arvore geradora de menor custo. A partir do pseudopódico fornecido:

Kruskal's – Algorithm

```
Algorithm Graph<V,E> kruskall (Graph<V,E> g) {
  for (all V vertices in g)
       Add vertex V to mst
  for (all E edges in g)
       Add edge into a lstEdges
  sort lstEdges              // in ascending order of weight
  for (each Edge e=(VOrig,vDst) of lstEdges)
      connectedVerts=DepthFirstSearch(mst, vOrig);
      if (!connectedVerts.contains(vDst))
        add Edge to mst

  return mst;
}
```

*Figura 12. Pseudo-Código do Algoritmo de **Kruskal***

O seguinte algoritmo foi traduzido e implementado em java:

```
public static <V,E> Graph<V,E> kruskall (Graph<V,E> grafo,Comparator <Edge<V,E>> ce) {
   Graph <V,E> mst = new MapGraph<>( directed: false);

    for (V vertex: grafo.vertices())
        mst.addVertex(vertex);

    List<Edge<V, E>> edgesList = new ArrayList<>(grafo.edges());
    edgesList.sort(ce);

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

*Figura 13. Algoritmo de **Kruskal***

.

O algoritmo recebe um grafo, e retorna outro grafo gerador desse mesmo (árvore geradora de custo mínimo). Neste caso em concreto, a implementação do método requer um objeto do tipo ***Comparator <Edge<V,E>> ce***, uma vez que terá de comparar cada vértice para que os possa ordenar como vemos em ***edgeslist.sort(ce)***.

O algoritmo inicialmente cria um objeto do tipo ***Graph <V,E> mst*** que será a árvore de menor custo devolvida .

Durante a execução, o algoritmo percorre as arestas do grafo de entrada, ordenadas pelo comparador fornecido. Para cada aresta, verifica se adicionar a aresta à árvore geradora criaria um ciclo utilizando uma busca em profundidade a partir do método fornecido na classe ***Algorithms (DepthFirstSearch).*** Se não criar um ciclo, a aresta é adicionada à árvore geradora.

Por fim, o método retorna a arvore geradora de menor custo criada inicialmente ***mst*.**

Análise de Complexidade

O método principal **obterRedeComMenorCusto** em essencialmente complexidade do **algoritmo de Kruskal** criado sendo este um algoritmo conhecido, embora este utilize o método auxiliar ***DepthFirstSearch.*** Este tem a complexidade ***O* (*E* log (*E*)**) onde **E** corresponde ao número de arestas (***Edges***).

O método secundário **obterCustoTotalDeRedeComMenorCusto** tem uma ***Worst-Case Time Complexity*** de **O(E),** uma vez que irá percorrer todas as arestas (***Edges***) do grafo recebido para somar o custo de cada uma delas ao custo Total.

Sendo ambos os métodos independentes a complexidade temporal será classificada como a maior complexidade de ambos os métodos ao seja: **O (E log(E))**.

Restrições e Observações

Na **USEI01** (escolha de ficheiros) Os dois ficheiros inseridos têm obrigatoriamente de ser compatíveis um com o outro. Ou seja, não poderá inserir um ficheiro de distâncias com locais não presentes no ficheiro fornecido para os locais.

-Os ficheiros lidos na USEI01 tem de ser to tipo **csv.**

-A estrutura utilizada para a implementação de grafos no grupo foi um mapa: **MapGraph.**

Asymptotic performance of graph data structures

	Edge List	Adjacency List	Adjacency Map	Adjacency Matrix
Space	V + E	V + E	V + E	V^2
numVertices(), numEdges()	O(1)	O(1)	O(1)	O(1)
vertices()	O(V)	O(V)	O(V)	O(V)
getEdge(*u*, *v*)	O(E)	O(min(d_u,d_v))	O(1)	O(1)
outDegree(v) inDegree(v)	1	O(1) / O(V×E)	O(1) / O(VxE)	O(V)
outgoingEdges(v) incomingEdges(v)	O(E)	O(d_v) / O(V×E)	O(d_v) / O(VxE)	O(V)
insertVertex(*x*)	O(1)	O(1)	O(1)	O(V^2)
removeVertex(*v*)	O(E)	O(d_v)	O(d_v)	O(1)
insertEdge(*u*, *v*, *x*) removeEdge(x)	O(1)	O(1)	O(1)	O(1)

27

Figura 14. Diferenças de eficácia entre os tipos de grafo

Na USEI04 o custo total é divido por dois uma vez que na criação de um grafo não direcionado as arestas serão duplicadas.

Recursos

Informação Global do Projeto:

- Ficheiro ISEP Moodle descrição do Projeto:
  - *Enunciado Projecto Integrador - Versao 1.2 (10/nov)Ficheiro*

- Outros ficheiros de informação no Moodle:
  - *distancias_big.csv*
  - *distancias_small.csv*
  - *localidades_big.csv*
  - *localidades_small.csv*

Teoria e implementação de Grafos:

***Time-Complexity* dos vários tipos de grafo:**

- Moodle --» ESINF --» Graphs --» Lecture Slides: **27**.

Algoritmo de Kruskal:

- Moodle --» ESINF --» Graphs --» Lecture Slides: **98-106**.

Outras informações importantes sobre grafos:

- YouTube Video: "**Closure and composition: Transitive closure**" de *Douglas Weathers* Link: https://youtu.be/3uD1ftia8I8?si=lC-l5gg5HGnNsVYU.

- YouTube Video: "**Data Structure: Transitive closure of a graph**" de *vinay singh* Link https://youtu.be/-elv-GJNEYE?si=CK8GoOMUtLCl7ABq.

Recursos adicionais usados para implementações de métodos Java:

- YouTube Video: "**java Unit Testing with JUnit - Tutorial - How to Create And Use Unit Tests**" de *Coding with John* Link: https://youtu.be/gN29Y600k5g