RELATÓRIO ESINF

PROJETO INTEGRADOR - SPRINT 2

ENGENHARIA INFORMÁTICA

2023/2024

PROFESSORA

ISABEL SAMPAIO

ELEMENTOS DO GRUPO

JOSÉ SANTOS - Nº1220738

RITA BARBOSA - Nº1220841

MATILDE VARELA - Nº1220683

ANA GUTERRES - Nº1221933

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Índice

Índice de algoritmos

Índice de tabelas

Introdução

Este relatório tem como objetivo apresentar o trabalho realizado no *Sprint* 2. Neste documento serão referidos os vários elementos que definem as funcionalidades e as complexidades dos algoritmos implementados para cada *User Story*.

US- EI01

Nesta *User Story*, pretendíamos importar os dados dos ficheiros disponibilizados e, de seguida, construir uma rede de distribuição utilizando grafos. Posto isto, analisamos os dados fornecidos e decidimos implementar um grafo não direcionado utilizando um *Adjancency Map*, que no caso apresenta a seguinte estrutura: *MapGraph<Local, Integer>.*

Assim sendo, o seguinte código foi implementado com esse propósito:

```
public static boolean importRedeDistribuicao(String locais, String distancias) throws
ExcecaoFicheiro, IOException {
        importFicheiroLocais(locais);
        importFicheiroDistancias(distancias);
        return true;
    }
```

Algoritmo 1: Implementação do método importRedeDistribuicao().

```
private static void importFicheiroLocais(String locais) throws IOException {
    RedeHub redeHub = RedeHub.getInstance();

    BufferedReader reader = new BufferedReader(new FileReader(locais));
    String currentLine;
    reader.readLine();
    String[] line;
    while ((currentLine = reader.readLine()) != null) {
        line = currentLine.split(",");
        redeHub.addHub(line[0], Double.parseDouble(line[1]), Double.parseDouble(line[2]));
    }
}
```

Algoritmo 2: Implementação do método importFicheiroLocais().

Analisando o método, notamos que a complexidade do método é $O(n^2)$. Esta análise tem em conta que o método addHub, com a implementação apresentada a seguir, tem complexidade temporal de O(n). Isto deve-se ao facto que a variável *mapVertices*, utilizada no método *validVertex()* é um *LinkedHashMap,* e por isso o método *get()* possui complexidade O(n). Adicionalmente, o método split(), utilizado em *importFicheiroLocais*, também apresenta complexidade O(n).

Assim sendo, como ambos os métodos se encontram dentro de um *loop while* que percorre todo o ficheiro, a complexidade temporal do método *importFicheiroLocais* é, conforme mencionado anteriormente, $O(n^2)$.

```
public void addHub(String numId, Double lat, Double lon) {
   Local vert = new Local(numId, lat, lon);
   redeDistribuicao.addVertex(vert);
}
public boolean addVertex(V vert) {
   if (vert == null) throw new RuntimeException("Vertices cannot be null!");
   if (validVertex(vert))
      return false;

   MapVertex<V, E> mv = new MapVertex<>(vert);
   vertices.add(vert);
   mapVertices.put(vert, mv);
   numVerts++;
   return true;
}
public boolean validVertex(V vert) {
   return (mapVertices.get(vert) != null);
}
```

Algoritmo 3: Implementação dos métodos addHub(), addVertex() e validVertex().

Adicionalmente, o método *importFicheiroDistancias,* apresentado a seguir, tem complexidade temporal $O(n^2)$. Este faz uso do método *addRoute()* que, assim como o addHub(), utiliza os métodos *addVertex()* e *validVertex()* [Algoritmo 3] . Posto isto, addRoute() possui complexidade $O(n^2)$.

```
private static void importFicheiroDistancias(String distancias) throws IOException {
   RedeHub redeHub = RedeHub.getInstance();

   BufferedReader reader = new BufferedReader(new FileReader(distancias));
   String currentLine;
   reader.readLine();
   String[] line;
   while ((currentLine = reader.readLine()) != null) {
      line = currentLine.split(",");
      redeHub.addRoute(new Local(line[0]), new Local(line[1]), Integer.parseInt(line[2]));
   }
}

public void addRoute(Local orig, Local dest, Integer distance) {
   redeDistribuicao.addEdge(orig, dest, distance); //o(n)
}
```

Algoritmo 4: Implementação dos métodos importFicheiroDistancias() e addRoute().

```
public boolean addEdge(V vOrig, V vDest, E weight) { //o(n)
   if (vOrig == null || vDest == null) throw new RuntimeException("Vertices cannot be null!");
   if (edge(vOrig, vDest) != null)
```

```
        return false;

    if (!validVertex(vOrig))
        addVertex(vOrig);

    if (!validVertex(vDest))
        addVertex(vDest);

    MapVertex<V, E> mvo = mapVertices.get(vOrig);
    MapVertex<V, E> mvd = mapVertices.get(vDest);

    Edge<V, E> newEdge = new Edge<>(mvo.getElement(), mvd.getElement(), weight);
    mvo.addAdjVert(mvd.getElement(), newEdge);
    numEdges++;

    if (!isDirected)
        if (edge(vDest, vOrig) == null) {
            Edge<V, E> otherEdge = new Edge<>(mvd.getElement(), mvo.getElement(), weight);
            mvd.addAdjVert(mvo.getElement(), otherEdge);
            numEdges++;
        }

    return true;
}
```

Algoritmo 5: Implementação do método addEdge().

Concluindo, o método "global" da funcionalidade, *importRedeDistribuicao()* [Algoritmo 1], que utiliza os métodos *importFicheiroLocais()* [Algoritmo 2] e *importFicheiroDistancias()* [Algoritmo 4] tem complexidade temporal igual a $O(n^2)$.

US- EI02

Nesta *User Story*, é pedido para determinar os vértices ideais para a localização de N hubs, com o objetivo de otimizar a rede de distribuição. Para isso são necessários ter em conta os seguintes critérios:

- influência: vértices com maior grau
- proximidade: vértices mais próximos dos restantes vértices
- centralidade: vértices com maior número de caminhos mínimos que passam por eles

Para o desenvolvimento do critério da influência, optamos pela utilização do *outDegree()*, método esse que devolve o grau dos vértices de um grafo, visto que estamos perante um grafo não direcionado. Assim foi desenvolvido o método *calculateInfluence()* que retorna uma estrutura *Map<Local, Integer>,* de modo que para Local seja guardado o seu respetivo valor de influência. Este map é um *HashMap*, visto que ainda não é necessário realizar a ordenação do *Map*.

```
public Map<Local, Integer> calculateInfluence(MapGraph<Local, Integer> graph) {
        Map<Local, Integer> influence = new HashMap<>();
        for (Local vertex : graph.vertices()) {
            influence.put(vertex, graph.outDegree(vertex));
        }
        return influence;
    }
```

Algoritmo 6: Implementação do método calculateInfluence

Analisando este método, verifica-se que a sua complexidade vai ser O(V), visto que é usado um for para percorrer todos os vértices do *MapGraph*, já que como a estrutura usada para guardar os dados é um *HashMap*, o que leva a que a complexidade do *put()* seja O(1).

De modo a implementar o critério da proximidade, foi usado os métodos *calculateProximity* e *calculateVertexProximity*, método este que vai usar o algoritmo *shortestPaths*.

```
public Map<Local, Integer> calculateProximity(MapGraph<Local, Integer> graph) {
        Map<Local, Integer> proximity = new HashMap<>();
        for (Local vertex : graph.vertices()) {
            Integer proximityValue = calculateVertexProximity(graph, vertex);
            proximity.put(vertex, proximityValue);
        }
        return proximity;
    }
```

Algoritmo 7: Implementação do método calculateProximity

```
private Integer calculateVertexProximity(MapGraph<Local, Integer> graph, Local vertex) {
        ArrayList<Integer> dists = new ArrayList<>();
        Algorithms.shortestPaths(graph, vertex, Comparator.naturalOrder(), Integer::sum, 0, new
ArrayList<>(), dists);
        int proximitySum = 0;
        for (Integer dist : dists) {
            if (dist != null) {
                proximitySum += dist;
            }
        }
        return proximitySum;
    }
```

Algoritmo 8: Implementação do método calculateVertexProximity

Analisando o método *calculateVertexProximity()*, verifica-se que é utilizado o algoritmo *shortestPath()*, algoritmo esse que usa também o *shortestPathDijkstra()*, *initializePathDist()* e *getPath()*. O *getPath()* tem uma complexidade, no pior caso de O(V), caso verifique o caminho para todos os vértices, já o *initializePathDist()* terá de complexidade O(V), visto usar um ciclo for para percorrer o número de vértices. Para além disso o método *shortestPathDijkstra()* tem uma complexidade O((V + E) log V), devido aos dois ciclos *for* utilizado, onde no primeiro são percorridos todos os *edges* a partir do *outgoingEdges()* que tem uma complexidade de O(V*E), e no segundo são percorridos todos vértices, como método *vertices(),* que tem uma complexidade de O(V), visto que utilizamos um *MapGraph*.

De seguida, para a implementação do critério da centralidade é usado o método calculateCentrality, que usa o método betweennessCentrality.

```
public Map<Local, Integer> calculateCentrality(MapGraph<Local, Integer> graph) {
    Map<Local, Integer> centrality = Algorithms.betweennessCentrality(graph);
    return centrality;
  }
```

Algoritmo 9: Implementação do método calculateCentrality

A partir da análise do método *calculateCentrality()*, verifica-se que a complexidade do método será de O(V *(V + E)), visto que no algoritmo *betweennessCentrality()* ocorre uma procura por todos os vértices, implementada através de um ciclo for, que vai ter uma complexidade de O(V). Para além disso, ainda existe outros ciclos *for*, um que vai percorrer todos os vizinhos nesse vértice, o que significa que, no pior caso, a complexidade será de O(E). O outro volta a percorrer os vértices e coloca as informações no *HashMap*, pelo que terá uma complexidade de O(V). Assim, temos a complexidade de O(V * (V + E)).

Por fim, realizamos a ordenação da informação, que de acordo com o critério de aceitação, deveria ser feita por ordem decrescente de centralidade e influência e, em caso de igualdade dos critérios anteriores, por ordem crescente de proximidade. Para isso foi implementado o método *getTopNTotal()*, que vai chamar os métodos anteriormente referidos e ainda o método *getTopNMap()*.

```
public Map<Local, List<Integer>> getTopNTotal(MapGraph<Local,Integer> graph, Integer n){
    RedeHub redeHub = RedeHub.getInstance();
    Map<Local, Integer> influence = redeHub.calculateInfluence(graph);
    Map<Local, Integer> proximity = redeHub.calculateProximity(graph);
    Map<Local, Integer> centrality = redeHub.calculateCentrality(graph);
    Map<Local, List<Integer>> finalMap = new HashMap<>();
    for (Local key : influence.keySet()) {
      List<Integer> values = new ArrayList<>();
      values.add(influence.get(key));
      values.add(proximity.get(key));
      values.add(centrality.get(key));
      finalMap.put(key, values);
    }

    return redeHub.getTopNMap(finalMap, n);
  }
```

Algoritmo 10: Implementação do método getTopNTotal

O método *getTopNMap()*, vai primeiramente ordenar de acordo com o critério de aceitação referido anteriormente, e de seguida chama o método *getTopNHubs()*, que vai selecionar os *n hubs* desejados e colocá-los num *LinkedHashMap*, de modo que a ordenação feita anteriormente seja mantida. Deste modo, o método *getTopNHubs()* vai ter uma complexidade de O(V) e o método *getTopNMap()* vai ter uma complexidade de O(V), devido à ordenação e à posterior colocação no *Map*, mas também pela chamada do método *getTopNHubs()*.

```
public Map<Local, List<Integer>> getTopNMap(Map<Local, List<Integer>> map, Integer n){
    List<Map.Entry<Local, List<Integer>>> entries = new ArrayList<>(map.entrySet());
    entries.sort((entry1, entry2) -> {
        List<Integer> values1 = entry1.getValue();
        List<Integer> values2 = entry2.getValue();
        int compareCentrality = Integer.compare(values2.get(2), values1.get(2));
        if (compareCentrality != 0) {
            return compareCentrality;
        }
        int compareInfluence = Integer.compare(values2.get(0), values1.get(0));
        if (compareInfluence != 0) {
            return compareInfluence;
        }
        return Integer.compare(values1.get(1), values2.get(1));
    });
    Map<Local, List<Integer>> sortedFinalMap = new LinkedHashMap<>();
    for (Map.Entry<Local, List<Integer>> entry : entries) {
        sortedFinalMap.put(entry.getKey(), entry.getValue());
    }

    return getTopNHubs(sortedFinalMap, n);
}
```

Algoritmo 11: Implementação do método getTopNMap

```
public Map<Local, List<Integer>> getTopNHubs(Map<Local,List<Integer>> map, Integer n){
    Map<Local, List<Integer>> topNHubsMap = new LinkedHashMap<>();
    int count = 0;
    for (Map.Entry<Local, List<Integer>> entry : map.entrySet()) {
        if (count < n) {
            topNHubsMap.put(entry.getKey(), entry.getValue());
            count++;
        } else {
            break;
        }
    }
    return topNHubsMap;
}
```

Algoritmo 12: Implementação do método getTopNHubs

Assim, a complexidade final para esta *user story* vai ser de O(V *(V + E)), devido às complexidades dos diferentes métodos que são chamados em *getTopNTotal()*.

US- EI03

A *User Story* 3 de ESINF pede-nos para encontrar os pontos mais afastados da rede e, logo depois, encontrar o percurso mais pequeno entre eles, demonstrando o ponto de partida, o ponto de chegada, o percurso, a distância percorrida e todos os pontos de carregamento dada uma certa autonomia de um veículo.

Para esta *User Story* foi utilizado o algoritmo de Dijkstra.

```
public 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 keyVAdj = g.key(edge.getVDest());
            if (!visited[keyVAdj]) {
                E s = sum.apply(dist[vKey], edge.getWeight());
                if (dist[keyVAdj] == null || ce.compare(dist[keyVAdj], s) > 0) {
                    dist[keyVAdj] = s;
                    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;
            }
        }
    }
}
```

Algoritmo 13: Implementação do método shortestPathDijkstra

Após a criação deste método cria-se um outro que encontre todos os caminhos de menor esforço para cada vértice, dado um vértice inicial, chamado *shortestPaths().*

```
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 numVertices = g.numVertices();
    boolean[] visited = new boolean[numVertices];
    V[] pathKeys = (V[]) new Object[numVertices];
    E[] dist = (E[]) new Object[numVertices];
    initializePathDist(numVertices, pathKeys, dist);

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

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

    return true;
}
```

Algoritmo 14: Implementação do método shortestPaths

Estes dois métodos serão utilizados de forma que encontrem ao mesmo tempo, não só os pontos mais afastados da rede como, também, o seu percurso.

Para encontrarmos o maior caminho de menor esforço, que será indubitavelmente entre os pontos mais afastados, temos de utilizar estas duas funções para percorrem a lista de vértices e ir considerando se o caminho que tem é menor que aquele que analisa. Para isso, foi utilizado uma função chamada de *getShortestPathForFurthestNodes().*

```
public static LinkedList<Local> getShortestPathForFurthestNodes() {
    int maxDist = 0;
    LinkedList<Local> tempPath = new LinkedList<>();
    for (Local local : instance.getRedeDistribuicao().vertices) {
        ArrayList<LinkedList<Local>> path = new ArrayList<>();
        ArrayList<Integer> dist = new ArrayList<>();

        shortestPaths(instance.getRedeDistribuicao(),local,Comparator.naturalOrder(),
Integer::sum, 0,path,dist);
        if(dist.get(getBiggestDist(dist)) > maxDist){
            maxDist = dist.get(getBiggestDist(dist));
            tempPath = path.get(getBiggestDist(dist));
        }
    }
    return tempPath;
}

public static int getBiggestDist(ArrayList<Integer> dist){
    int temp = 0, index=0;
    for (int i = 0; i < dist.size(); i++) {
        if (dist.get(i) != null && temp < dist.get(i)){
            temp = dist.get(i);
            index=i;
        }
    }
    return index;
}
```

Algoritmo 15: Implementação do método getShortestPathForFurthestNodes

Esta função, como explicado anteriormente, irá, para cada vértice, calcular o menor caminho para todos os restantes vértices e a sua distância. Depois irá encontrar o maior caminho e temporariamente guardá-lo de forma a comparar constantemente até, após percorrer todos os vértices, encontrar o maior percurso de menor esforço para os pontos mais afastados na rede.

De forma a facilitar a desmontagem dos dados pedidos ao cliente, foi criada também uma classe denominada de EstruturaDeEntregaDeDados, que conterá todos os valores pedidos pelo usuário e que serão depois demonstrados na UI.

Para saber se o carro consegue percorrer esse caminho, foi utilizada a função analyzeData(), que, tal como o nome indica, analisa os dados obtidos pelas funções já mencionadas e a partir destas compara uma variável denominada bateria que começa “carregada”. A cada aresta, é retirada o valor de “*weight*” à bateria - se o valor chegar a ser negativo quando tenta percorrer uma aresta, é testado se a bateria carregada funciona para percorrer tal caminho. Se esse não for o caso, a variável *boolean flag* passa para *false*, indicando que não é possível.

```
public static EstruturaDeEntregaDeDados analyzeData(int autonomia){
   ArrayList<Integer> indexDeCarregamentos = new ArrayList<>();
   LinkedList<Local> percurso = getShortestPathForFurthestNodes();
   int distanciaPercorrida = 0, bateria=autonomia;
   boolean flag = true;
   for (int i = 0; i < percurso.size()-1; i++) {
      int distanciaEntrePontos =
instance.getRedeDistribuicao().edge(percurso.get(i),percurso.get(i+1)).getWeight();
      distanciaPercorrida += distanciaEntrePontos;
      if(distanciaEntrePontos > bateria){
         if (distanciaEntrePontos <= autonomia){
            indexDeCarregamentos.add(i);
            bateria = autonomia;
         }else{
            flag = false;
         }
      }else{
         bateria -= distanciaEntrePontos;
      }
   }
   return new
EstruturaDeEntregaDeDados(distanciaPercorrida,percurso,indexDeCarregamentos,flag);
}
```

Algoritmo 16: Implementação do método analyzeData

Finalmente, analisaremos a complexidade espacial, como os dados são colocados numa estrutura de dados customizada para esta funcionalidade, ao analisarmos a complexidade espacial da classe EstruturaDeDadosDeEntrega saberemos a complexidade espacial.

A classe EstruturaDeDadosDeEntrega possui os seguintes atributos:

- Uma variável *Integer*;

- Uma variável *LinkedList*<Local>;

- Uma variável *ArrayList*<*Integer*>;

- Uma variável *Boolean*;

A complexidade espacial estará então na responsabilidade das duas listas presentes nos seus atributos.

Como a complexidade espacial de uma *LinkedList* é de O(V) e a complexidade da *ArrayList* é, no pior dos casos, também O(V), temos então de considerar O(V)+O(V), o que acaba por ser O(V) de complexidade espacial.

Passemos à analise temporal, a função shortestPathDijkstra() possui uma complexidade temporal de O(log(V)), como a função repete essa funcionalidade V vezes, já que o faz para todos os vértices, ficamos com O(V * log(V)). Finalmente, como a função getShortestPathForFurthestNodes() chama a função shortestPaths() V vezes, então a complexidade temporal é O(V*(V*Log(V))).

US- EI04

A User Story 4 de ESINF pedia para gerar uma rede que ligasse todas as localidades com uma distância mínima. A funcionalidade deveria retornar os locais, as distâncias entre estes e a distância total da rede. Para implementar esta *user story* recorreu-se ao algoritmo de *Kruskal*.

```
    public static <V, E extends Comparable<E>> MapGraph<V, E >
getMstWithKruskallAlgorithm(Graph<V, E> g) {

    MapGraph<V, E> mst = new MapGraph<>(false);

    ArrayList<V> vertices = g.vertices();

    for (V vert : vertices) {

      mst.addVertex(vert);

    }

    ArrayList<Edge<V, E>> edgesList = (ArrayList<Edge<V, E>>) g.edges();

    edgesList.sort(Comparator.comparing(Edge::getWeight));

    LinkedList<V> connectedVerts;

    for (Edge<V, E> edge : edgesList) {

      connectedVerts = Algorithms.DepthFirstSearch(mst, edge.getVOrig());

      assert connectedVerts != null;

      if (!connectedVerts.contains(edge.getVDest())) {

        mst.addEdge(edge.getVOrig(), edge.getVDest(), edge.getWeight());

      }

    }

    return filterMstKruskkall(mst);

  }
```

Algoritmo 17: Implementação do método getMstWithKruskallAlgorithm

Posto isto, cria-se uma instância de *MapGraph*, adicionando-se os vértices do grafo passado como parâmetro (addVertex() – O(V)). Segue-se a extração da lista das arestas (*edge*) do grafo e a sua ordenação ascendente.

Para cada *edge,* aplica-se o algoritmo *DepthFirstSearch*(), verifica-se se a *LinkedList* devolvida pelo método contém o vértice de destino da aresta. Se este não tiver, então adiciona-se a *edge* à rede de ligação mínima. Esta estrutura é utilizada, pois tem tamanho dinâmico e o algoritmo envolve muitas inserções, especialmente se o grafo tiver muitos vértices.

Infelizmente, o algoritmo não identificava arestas duplicadas, isto é, onde duas arestas partilham os vértices, embora estes estejam invertidos, e tenham o mesmo peso (*weight*). Para colmatar esta falha, criou-se um método: *filtrarMstKruskall*().

```
        private static <V, E extends Comparable<E>> MapGraph<V, E> filterMstKruskall(Graph<V,
E> mst) {
            ArrayList<Edge<V,E>> edgesList = new ArrayList<>(mst.edges());
            edgesList.sort(Comparator.comparing(Edge::getWeight));
            int i = 0;
            while (edgesList.size() != (mst.numVertices() - 1)) {
              if (mst.edges().contains(edgesList.get(i).reverse())) {
                edgesList.remove(edgesList.get(i).reverse());
              }
              ++i;
            }
            MapGraph<V, E> result = new MapGraph<>(true);
    for (V vert : vertices) {
      mst.addVertex(vert);
    }
            for (Edge<V, E> edge : edgesList) {
              result.addEdge(edge.getVOrig(), edge.getVDest(), edge.getWeight());
            }
            return result;
        }
```

Algoritmo 18: Implementação do método filterMstKruskall

Copia-se a lista de *edges* do grafo passado como parâmetro, ordenando-as ascendentemente em relação ao peso das arestas. Posto isto, verifica-se se na lista de edges encontram-se duplicados, se sim, remove-se a *edge*, até que a lista de arestas tenha tamanho V-1, onde V é o número de vértices.

Cria-se uma instância de *MapGraph*, adiciona-se os vértices da MST (*minimum spanning tree*) e adiciona-se as arestas da lista já filtrada. Contudo, este método só faz efeito se definirmos que a estrutura que será devolvida é direcionada, se não a classe *MapGraph* irá acrescentar a aresta inversa.

O *MapGraph* é um *Map* que representa um grafo. A *key* é o vértice V e o seu *value* é um outro mapa com *key* um vértice X, adjacente a V, e o *value* é o peso da aresta entre estes dois vértices. Isto permite um acesso mais eficiente a uma aresta específica, como uma expetativa de tempo O(1). O seu espaço total é de O(V+E).

Para concluir, a complexidade temporal apenas do algoritmo de *Kruskal* é $O(V^2)$ (inserção de vértices no MapGraph) + O(ElogV) (o resto do algoritmo em si). Posto isto, a complexidade desta parte é de $O(V^2)$. O método de filtragem tem complexidade de $O(E*V) + O(V^2)$, onde E representa o número de arestas.

Tendo em conta estas complexidades, pode-se afirmar que a complexidade temporal final do algoritmo de *Kruskal* varia conforme o valor de V e E. A seguinte tabela agrega os cenários possíveis.

	Gráfico Denso V > E	Gráfico Esparso E > V
Algoritmo de Kruskal | $O(V^2)$	-	-
Filtragem de Output | $O(E*V) + O(V^2)$	$O(V^2)$	$O(E*V)$
Funcionalidade	$O(V^2)$	$O(E*V)$

Tabela 1: Cenários da complexidade da USEI04.

Confirma-se que, dependendo dos valores de E e V, se o número de vértices for superior ao de arestas, então a complexidade do pior cenário é $O(V^2)$, caso contrário é $O(E*V)$.

Conclusão

Após a conclusão deste trabalho, compreendemos de forma mais aprofundada o conceito de grafos. Associado a este tema foram explorados outros conceitos como algoritmos de pesquisa transversal, de caminhos mais curtos e árvores de menor custo. No decorrer do projeto, aprendemos a distinguir as particularidades e usos apropriados de cada uma deles, dependendo dos requisitos associados a determinado contexto ou problema.

Como nos é sempre requerido, a acompanhar a descrição de cada funcionalidade está também a complexidade temporal destas, associada às estruturas usadas. Isto permitiu tomar decisões apropriadas sobre a escolha das melhores soluções para otimizar o desempenho da aplicação.