Relatório do Trabalho Prático 2

Instituto Superior de Engenharia do Porto

Estruturas de Informação

2020/2021

1181743- Guilherme Daniel

1171589- Lucas Sousa

Classes Definidas:

Para a elaboração do trabalho prático foram definidas 5 classes diferentes- 1 classe para cada tipo de ficheiro de texto a ser lido (countries, borders, users, relationships) e 1 classe Rede Social:

- **Country**: possui o nome do país, continente, população, capital, latitude e longitude;
- **Border**: tem o nome do país origem e do país destino, que partilham uma fronteira;
- **User**: possui o id do utilizador, idade e cidade;
- **Relation**: tem um utilizador origem e um utilizador destino, que são amigos.
- **RedeSocial**: faz a leitura dos ficheiros de texto, constrói os grafos e possui os diferentes algoritmos necessários à execução das alíneas do trabalho prático.

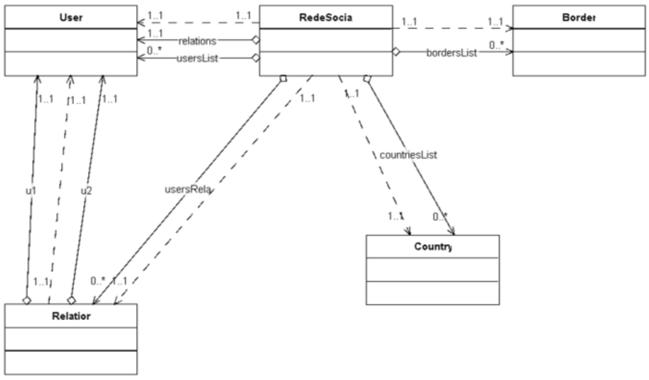

Figura 1- Excerto do digrama de classes

A criação destas classes facilitou a leitura dos ficheiros de texto e toda a criação dos grafos pretendidos.

Para além das 5 classes definidas, são também implementadas as classes base de grafos, para a rede de cidades e para a rede de amigos. Estas classes para além das essenciais, para a criação dos diferentes grafos, possuem também as classes **GraphAlgorithms** e **AdjacencyGraphAlgorithms**, essenciais à realizações de **todas** as alíneas do trabalho prático.

Análise de Complexidade:

Exercício 1:

Para evitar a repetição da análise algorítmica nos métodos utilizados no exercício 1, apenas serão demonstrados algumas das funcionalidades desta alínea, sendo que há uma grande quantidade de métodos pertencentes à resolução do exercício:

```java
private void addCountries(List<String> countryInfo) {

    for (String s : countryInfo) {
        String data[] = s.split(",");
        data[0] = data[0].replaceAll("\\s+", "");
        data[1] = data[1].replaceAll("\\s+", "");
        data[2] = data[2].replaceAll("\\s+", "");
        data[3] = data[3].replaceAll("\\s+", "");
        data[4] = data[4].replaceAll("\\s+", "");
        data[5] = data[5].replaceAll("\\s+", "");
        Country c = new Country(data[0], data[1], Double.parseDouble(data[2]), data[3], Double.parseDouble(data[4]), Double.parseDouble(data[5]));
        countriesList.add(c);
    }

}

private List<Border> insertConnections(List<String> bordersInfo) {
    List<Border> borders = new ArrayList<>();

    for (String s : bordersInfo) {

        String data[] = s.split(",");
        data[1] = data[1].replaceAll("\\s+", "");
        Border b = new Border(data[0], data[1]);
        borders.add(b);
    }

    return borders;
}
```

Figura 2- 2 métodos pertencentes ao primeiro exercício

Em ambos os métodos, são realizadas n (neste caso, s), iterações para cada lista do tipo String, pelo que terão uma complexidade temporal de O(n).

```java
/**
 * Adiciona uma ligação entre dois países
 * @param borders
 */
private void addConnection(List<Border> borders) {
    Country a = null;
    Country b = null;
    int incr = 0;
    for (Border bor : borders) {

        for (Country c : countriesList) {
            if (c.getCountry().equals(bor.getCountryA())) {
                a = c;
            }
            if (c.getCountry().equals(bor.getCountryB())) {
                b = c;
            }
        }
        citiesCon.insertEdge(a.getCapital(), b.getCapital(), incr, calculateDistance(a, b));
        incr++;
    }
}
```

Figura 3- Método que adiciona ligações entre países

Trata-se de um nested loop, de duas listas diferentes, pelo que serão feitas n (bor) iterações pelo número de borders existentes na lista, e n iterações (c), pelo número de países que existe na lista de países. O melhor caso possível seria serem encontrados desde início os dois países que estão a ser procurados, mas sabendo que estes nunca são iguais, terá sempre uma complexidade temporal quadrática, de O(n^2).

Mais uma vez, apenas foi demonstrada a complexidade de 3 métodos para evitar a sobrelotação desnecessária no relatório de métodos que têm, quase sempre, a mesma complexidade temporal.

Exercício 2:

Na resolução desta alínea, criamos o método “amigosComuns”, que recebe por parâmetro a quantidade de utilizadores mais populares que vamos encontrar, para posteriormente serem encontrados os seus amigos em comum. Primeiramente fazemos uma validação para ver se a quantidade de utilizadores mais populares não é negativa nem superior ao número total de utilizadores registados na matriz “relations”. Apos a validação referida anteriormente preenchemos a lista de users “usersMaisPopulares” com os n utilizadores mais populares que introduzimos por parâmetro, fazendo recurso ao método “utlzMaisPopulares”. A maneira que encontramos de descobrir se um utilizador era o mais popular foi preencher um LinkedHashMap “utlznumAmigos” em que a key é o User e o value a quantidade de edges que saem desse vertex. De seguida ordenamos “utlznumAmigos” por ordem descendente e retornamos os n utilizadores desejados.

Já com os utilizadores mais populares vamos encontrar todos os amigos, preenchendo a lista “amigosComum”, depois verificamos se os amigos da lista “amigosComum”, têm ligação com todos os utilizadores mais populares se isso não se verificar são introduzidos para a lista “amigosARemover” para finalmente sem removidos da lista “amigosComum”, está que será a retornada pelo método.

```java
//-------------------------------------------------------------------------------------------
//--------------------------    EXERCICIO 2 -------------------------------------------------
/**
 * Devolve os amigos em comum entre os n utilizadores mais populares da rede
 *
 * @param qtdPopulares numero de utilizadores mais populares a ser retornado
 * @return amigos em comum entre os n utilizadores mais populares
 */
    public List<User> amigosComuns(int qtdPopulares) {

    if ((!(relations.numVertices() >= qtdPopulares)) || qtdPopulares <= 0) {
        return null;
    }

    List<User> usersMaisPopulares = utlzMaisPopulares(qtdPopulares);
    List<User> amigosComum = new ArrayList<>();
    List<User> amigosARemover = new ArrayList<>();

    for (User popular : usersMaisPopulares) {
        for (User amigo : relations.directConnections(popular)) {
            if (!amigosComum.contains(amigo)) {
                amigosComum.add(amigo);
            }
        }
    }

    for (User popular : usersMaisPopulares) {
        for (User amigo : amigosComum) {
            if (relations.getEdge(popular, amigo) == null) {
                amigosARemover.add(amigo);
            }
        }
    }

    for (User u : amigosARemover) {
        amigosComum.remove(u);
    }

    return amigosComum;
}
```

Vamos analisar a Complexidade algorítmica por detrás do método “amigosComuns”, começando pelo método “utlzMaisPopulares” este que tem uma complexidade de **O(V)** no primeiro ciclo for, o segundo ciclo for é não determinístico tendo uma complexidade de **O(1)** no melhor cenário e **O(V)** no pior. Portanto a complexidade deste método é **O(V)**. No resto do método temos complexidade **O(V^2)** nos ciclos de preenchimentos das listas e finalmente no último ciclo temos novamente complexidade **O(V)**. Ou seja, todo o nosso algoritmo tem complexidade temporal de **O(V^2)**.

```java
/**
 * Devolve a lista dos n utilizadores mais populares
 *
 * @param qtdPopulares numero de utilizadores mais populares a ser retornado
 * @return n utilizadores mais populares
 */
protected List<User> utlzMaisPopulares(int qtdPopulares) {

    int numPopulares = qtdPopulares;
    List<User> usersMaisPopu = new ArrayList<>();
    LinkedHashMap< User, Integer> utlznumAmigos = new LinkedHashMap<>();

    for (User u : relations.vertices()) {
        utlznumAmigos.put(u, relations.outDegree(u));
    }

    LinkedHashMap utlznumAmgOrde = sortMap(utlznumAmigos);

    for (Object u : utlznumAmgOrde.keySet()) {
        if (numPopulares == 0) {
            break;
        }
        numPopulares--;
        usersMaisPopu.add(((User) u));
    }

    return usersMaisPopu;
}
```

Exercício 3:

Para a elaboração deste exercício, seria necessário primeiro determinar se o grafo era conectado ou não, para isso utilizou-se o seguinte método:

```
protected boolean isConnected() {
    User firstVertex = usersList.get(0);

    List<User> path = AdjacencyGraphAlgorithms.DFS(relations, firstVertex);

    if (path.size() == relations.numVertices()) {
        return true;
    }
    return false;
}
```

Figura 4- Método utilizado para determinar se um grafo é conectado ou não

Analisando o método, é possível reparar que se utiliza o algoritmo **Depth-First Search,** que recorre à stack para efetuar a procura. Se o tamanho do caminho (número de vértices) encontrado recorrendo ao algoritmo for igual ao número de vértices do grafo, o grafo é conectado. Para determinar a complexidade do DFS, analisemos o seguinte algoritmo:

```
/**
 * Performs depth-first search of the graph starting at vertex.
 * Calls package recursive version of the method.
 * @param graph Graph object
 * @param vertex Vertex of graph that will be the source of the search
 * @return queue of vertices found by search (empty if none), null if vertex does not exist
 */
public static <V,E> LinkedList<V> DFS(AdjacencyMatrixGraph<V,E> graph, V vertex) {
    int index = graph.toIndex(vertex);
    if(index == -1)
        return null;
    boolean visited[] = new boolean[graph.numVertices];
    LinkedList<V> verticesQueue = new LinkedList<>();
    DFS(graph,index,verticesQueue,visited);
    return verticesQueue;
}

/**
 * Actual depth-first search of the graph starting at vertex.
 * The method adds discovered vertices (including vertex) to the queue of vertices
 * @param graph Graph object
 * @param index Index of vertex of graph that will be the source of the search
 * @param verticesQueue queue of vertices found by search
 *
 */
static <V,E> void DFS(AdjacencyMatrixGraph<V,E> graph, int index, LinkedList<V> verticesQueue, boolean visited[]) {
    V vertice = graph.vertices.get(index);
    verticesQueue.add(vertice);
    visited[index] = true;
    for (V vert: graph.directConnections(vertice)) {
        if(!visited[graph.toIndex(vert)]){
            DFS(graph,graph.toIndex(vert),verticesQueue,visited);
        }
    }
}
```

Figura 5- Algoritmo Depth-First Search

Analisando o algoritmo, constamos que tem uma complexidade temporal de O(VxE), onde V é o número de vértices e E é o número de ramos do grafo, sendo que se trata de um algoritmo recursivo. Logo, o método de verificação da conectividade do grafo terá complexidade temporal de O(VxE) e espacial de O(V).

Depois de verificar se o grafo é conectado, será feito o algoritmo **Floyd-Warshall**, que irá calcular o caminho mais curto entre todos os pares de vértices do grafo, sendo devolvida a maior das ligações:

```java
public Double minConncectionsToReachAllUsers() {
    Double maxConnection = 0.0;

    if (!isConnected()) {
        return 0.0;
    }

    int V = relations.numVertices();
    Double dist[][] = new Double[V][V];
    int iC = 0, jC = 0, kC = 0;

    for (User i : relations.vertices()) {
        jC = 0;
        for (User j : relations.vertices()) {
            dist[iC][jC] = relations.getEdge(i, j);
            jC++;
        }
        iC++;
    }

    AdjacencyMatrixGraph copy = relations;
    iC = 0;
    jC = 0;

    for (User k : relations.vertices()) {
        iC = 0;
        for (User i : relations.vertices()) {
            jC = 0;
            for (User j : relations.vertices()) {
                if (i == j) {
                    dist[iC][jC] = 0.0;
                }
                if (dist[iC][kC] != null && dist[kC][jC] != null && dist[iC][jC] != null) {
                    if (dist[iC][kC] + dist[kC][jC] < dist[iC][jC]) {
                        dist[iC][jC] = dist[iC][kC] + dist[kC][jC];
                    }
                }
                if (dist[iC][jC] == null && dist[iC][kC] != null && dist[kC][jC] != null) {
                    dist[iC][jC] = dist[iC][kC] + dist[kC][jC];
                }
                jC++;
            }
            iC++;
        }
        kC++;
    }

    // Descobre qual o maior número de ligacoes de um utilizador para outro utilizador
    for (int i = 0; i < dist.length; i++) {
        for (int j = 0; j < dist.length; j++) {
            if (dist[i][j] > maxConnection) {
                maxConnection = dist[i][j];
            }
        }
    }

    return maxConnection;
}
```

O(V^2)

Algoritmo de Floyd-Warshall (complexidade de O(V^3))

O(n^2)

Figura 6- Método que determina o número mínimo de ligações para chegar de um vértice a outro qualquer

O par de ciclos for que não pertence à cerne do algoritmo de Floyd-Warshall foi identificado na figura, tendo ambos uma complexidade temporal quadrática. O algoritmo de Floyd-Warshall, por ter 3 ciclos for seguidos, apresentará uma complexidade temporal de **O(V^3),** onde V é o número de vértices. Logo, pela teoria da complexidade algorítmica, o método terá uma complexidade temporal de **O(V^3).**

Exercício 4:

Para a elaboração desta alínea começamos por procurar um método que nos retornasse a distância (numero de "edges") mínima entre um vértice origem e todos os restantes. E o algoritmo que nos dava a melhor complexidade de todos era o de Dijkstra, este que foi implementado no método "shortesPathEdges". Graças a este método sabemos a quantidade de fronteiras "edges" mínimas entre a localização do User "principal" e todas as restantes existentes no grafo "citiesCon". Finalmente só tivemos de agrupar num LinkedHasMap "utlzPorCidade" todos os utilizadores que são amigos do "principal", recebido por parâmetro, e verificar se essa cidade está dentro do número de fronteiras máximo "numFronteiras" recebido também por parâmetro. No final retornamos o LinkedHasMap "utlzPorCidade" com o id do utilizador e a sua respetiva cidade.

```java
/**
 * Devolve os amigos que se encontram nas proximidades de um utilizador
 *
 * @param principal user a serem devolvidos os amigos nas suas proximidades
 * @param numFronteiras numero de fronteiras
 * @return amigos que se encontram nas proximidades de um utilizador
 */
public LinkedHashMap<String, String> amigosNasProximidades(User principal, int numFronteiras) {

    if ( numFronteiras < 0 || !this.relations.checkVertex(principal)) {
        return null;
    }

    LinkedHashMap<String, String> utlzPorCidade = new LinkedHashMap<>();
    String cityPrincipal = principal.getCity();
    int dist[];

    dist = shortestPathEdges(cityPrincipal);

    for (String city : this.citiesCon.vertices()) {
        for (User u : relations.directConnections(principal)) {
            if (city.contains(u.getCity()) && dist[citiesCon.getKey(city)] <= numFronteiras) {
                utlzPorCidade.put(u.getUser(), city);
            }
        }
    }
    return utlzPorCidade;
}
```

Figura 7- Método que determina os amigos nas proximidades do utilizador

Vamos analisar a Complexidade algorítmica por detrás do método "amigosNasProximidades", começando pelo método "shortesPathEdges" este que tem uma complexidade de **O(V)**, para o primeiro ciclo em que percorre todos os vértices do grafo "citiesCon". A complexidade no ciclo while com o for embutido é de **O(V x E)** uma vez que para cada vértice "V" vamos ver as suas arestas "E". Sendo, portanto, esta a complexidade do algoritmo "shortesPathEdges". Já no "amigosNasProximidades" temos, **O(V)** uma vez que estamos a percorrer todos os vértices do "citiesCon" e O(E) uma vez que estamos a ver todos os vértices que tem conceção direta com o User "principal", logo **O(V x E)** novamente, sendo esta a complexidade do nosso algoritmo.

```java
/**
 * Devolve todos os edges mais curtos para chegar a cada um dos outros vértices
 * @param vOrig vértice a serem devolvidos todos os edges mais curtoes
 * @return edges mais curtoes para chegar a cada um dos outros vértices
 */
private int[] shortestPathEdges(String vOrig) {

    String orig;
    int dist[] = new int[citiesCon.numVertices()];
    LinkedList<String> queueAux = new LinkedList<>();

    for (String vertices : citiesCon.vertices()) {
        dist[citiesCon.getKey(vertices)] = Integer.MAX_VALUE;
    }

    queueAux.add(vOrig);
    dist[citiesCon.getKey(vOrig)] = 0;

    while (!(queueAux.isEmpty())) {
        orig = queueAux.pop();
        for (String vAdj : citiesCon.adjVertices(orig)) {
            if (dist[citiesCon.getKey(vAdj)] == Integer.MAX_VALUE) {
                dist[citiesCon.getKey(vAdj)] = dist[citiesCon.getKey(orig)] + 1;
                queueAux.add(vAdj);
            }
        }
    }

    return dist;
}
```

Figura 8- Método que encontra os ramos mais curtos a partir de um vértice

Exercício 5:

Para determinar a complexidade do método, serão efetuadas análise de algoritmos passo a passo:

```
public List<String> citiesWithMoreCentrality(int nPopular, double pLeast) {
    Map<String, Double> cities = new HashMap<>();
    Map<String, Double> citiesCopy = new HashMap<>();

    List<String> s = new ArrayList<>();

    ArrayList<LinkedList<String>> paths = new ArrayList<>();
    ArrayList<Double> dists = new ArrayList<>();

    for (String capital : citiesCon.vertices()) {
        double sum = 0;

        GraphAlgorithms.shortestPaths(citiesCon, capital, paths, dists);
        for (int i = 0; i < dists.size(); i++) {
            if (dists.get(i) != Double.MAX_VALUE) {     // se o valor for sup
1               sum += dists.get(i);
            }
        }

        cities.put(capital, sum / citiesCon.numVertices());
    }

    Iterator<Map.Entry<String, Double>> iter = cities.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry<String, Double> entry = iter.next();
2       if (relativeFrequencyUsersPerCity(entry.getKey()) < pLeast) {
            iter.remove();
        }
    }

    Map sortedMap = sortByValue(cities);

    int cnt = 0;
    for (Object key : sortedMap.keySet()) {
        if (cnt == nPopular) {
3           break;
        }
        citiesCopy.put((String) key, (Double) sortedMap.get(key));
        cnt++;
    }
    Map sorted = sortByValue(citiesCopy);
    List<String> returnedList = new ArrayList<>(sorted.keySet());

    return returnedList;
}
```

Figura 9- Método que determina as cidades com mais centralidade

1- A primeira etapa do método, que adiciona para um map, como chave a capital e valor a média de distância entre ela e as outras cidades. Este método depende de um outro método da classe GraphAlgorithms- **shortestPaths**- e de seguida, será analisada a sua complexidade:

```
public static <V, E> boolean shortestPaths(Graph<V, E> g, V vOrig, ArrayList<LinkedList<V>> paths, ArrayList<Double> dists) {
    if (!g.validVertex(vOrig)) return false;

    int nverts = g.numVertices();
    boolean[] visited = new boolean[nverts];
    V[] pathKeys = (V[]) Array.newInstance(vOrig.getClass(), nverts);
    double[] dist = new double[nverts];

    for (int i = 0; i < nverts; i++) {
        dist[i] = Double.MAX_VALUE;
        pathKeys[i] = null;                    O(V)
    }

    shortestPathLength(g, vOrig, visited, pathKeys, dist);        1.1
    dists.clear();
    paths.clear();

    for (int i = 0; i < nverts; i++) {
        paths.add(null);
        dists.add(Double.MAX_VALUE);           O(V)
    }

    for (V vDst : g.vertices()) {
        int i = g.getKey(vDst);
        if (dist[i] != Double.MAX_VALUE) {
            LinkedList<V> shortPath = new LinkedList<>();
            getPath(g, vOrig, vDst, pathKeys, shortPath);           1.2
            paths.set(i, shortPath);
            dists.set(i, dist[i]);
        }
    }

    return true;
}
```

*Figura 10- Método **ShortestPaths**, chamado no método da centralidade*

1.1-Trata-se de um método pertencente também à classe GraphAlgorithms, e tem uma complexidade temporal de **O(ExV)** por se tratar de um método que recorre ao algoritmo de Djisktra.

1.2-

```
protected static <V, E> void getPath(Graph<V, E> g, V vOrig, V vDest, V[] pathKeys, LinkedList<V> path) {
    if (vOrig.equals(vDest)) {
        path.push(vOrig);
    } else {
        path.push(vDest);
        int vKey = g.getKey(vDest);
        vDest = pathKeys[vKey];
        getPath(g, vOrig, vDest, pathKeys, path);
    }
}
```

Figura 11- Método getPath

No melhor caso, a complexidade seria de O(1), caso o vértice de origem fosse igual ao método de destino. No pior, trata-se de um método recursivo para todos os vértices do grafo, tendo uma complexidade de O(VxE)

Concluímos portante que a complexidade temporal do método ShortestPaths é de **O(VxE).** Logo, a complexidade do método será de **O(V^2*E^2).**

2- Nesta parte do método, serão eliminadas todas as capitais que tenham um p% inferior ao esperado. A complexidade de O(Vxn^2) (relativeFrequencyUsersPerCity tem complexidade de O(n^2))

3- O(n)

Concluindo, o método citiesWithMoreCentrality tem uma complexidade temporal de **O(V^2xE^2)**

Exercício 6:

Nesta alínea seria necessário determinar o caminho mais curto passando por n vértices intermédios, que correspondem às cidades em que os dois utilizadores têm mais amigos.

Para determinar a complexidade e explicar a funcionalidade do método, serão explicadas passo a passo cada porção do método:

```
public Map<Integer, LinkedHashMap<String, Double>> shortestPathWithIntermediateVertexs(User u1, User u2, LinkedList<String> intermediate, int n) {

    if (intermediate.isEmpty()) { //so para efeitos de testes unitarios
        intermediate = intermidiateVertices(u1, u2, n);                          1
    }

    Iterator<String> iter = intermediate.iterator();
    while (iter.hasNext()) {
        String str = iter.next();
        if (str.equals(u1.getCity()) || str.equals(u2.getCity())) {            2
            iter.remove();
        }
    }

    intermediate.addFirst(u1.getCity());
    intermediate.addLast(u2.getCity());

    Map<Integer, LinkedHashMap<String, Double>> paths = new HashMap<>();

    for (int i = 1; i < intermediate.size(); i++) {
        LinkedHashMap<String, Double> map2 = new LinkedHashMap<>();
        LinkedList<String> info = new LinkedList<>();
        GraphAlgorithms.shortestPath(citiesCon, intermediate.get(i - 1), intermediate.get(i), info);

        LinkedList<String> copyCity = new LinkedList<>();

        //insere os vertices
        for (int b = 0; b < info.size(); b++) {

            copyCity.add(info.get(b));
            map2.put(info.get(b), 0.0);

        }
                                                                                             3
        //insere a distancia entre os vertices para o map a ser retornado
        for (int cc = 0; cc < copyCity.size(); cc++) {
            if (cc > 0) {
                for (String mm : map2.keySet()) {
                    if (copyCity.get(cc).equals(mm)) {
                        Country a = getCountryByCity(copyCity.get(cc - 1));
                        Country b = getCountryByCity(copyCity.get(cc));
                        Double distance = calculateDistance(a, b);
                        map2.put(mm, distance);
                    }
                }
            }
        }

        paths.put(i, map2);
    }
    for (Integer j : paths.keySet()) {
        if (paths.get(j).isEmpty()) {
            return null;
        }
    }

    return paths;

}
```

Figura 12- Método shortestPathWithIntermediateVertexs que determina o caminho mais curto com n vértices intermédios

1- Nesta parte inicial do método, apenas para efeitos de testes unitários, é possível já inserir vértices intermédios por parâmetro, caso não existam vértices intermédios, é realizado o método que os descobre, sendo que estes são as cidades em que cada utilizador tem mais amigos. O método é o seguinte:

```java
private LinkedList<String> intermidiateVertices(User u1, User u2, int n) {
    LinkedList<String> l1 = userCitiesWithMoreFriends(u1, n);
    LinkedList<String> l2 = userCitiesWithMoreFriends(u2, n);
    LinkedList<String> mergedLists = mergeLists(l1, l2);

    return mergedLists;
}
```

Figura 13- Método que determina os vértices intermédios

```java
public LinkedList<String> userCitiesWithMoreFriends(User us, int n) {
    Map<String, Integer> mapCountries = new HashMap<>();

    LinkedList<String> friendsCities = new LinkedList<>();
    LinkedList<String> returnedCities = new LinkedList<String>();

    if (!usersList.contains(us)) {
        return null;
    }

    //ligacoes diretas do user, ou seja, os seus amigos
    for (User u : relations.directConnections(us)) {
        friendsCities.add(u.getCity());
    }
    //conta a frequencia de cada cidade nos amigos do utilizador us
    for (String s : friendsCities) {
        LinkedList<String> shortPath = new LinkedList<>();
        double shortestPath = GraphAlgorithms.shortestPath(citiesCon, s, us.getCity(), shortPath);

        if (shortestPath != 0) { // ha cidades em que nao ha caminho terrestre possivel, e preciso
            mapCountries.put(s, Collections.frequency(friendsCities, s));
        }

    }

    Map order = sortByValue(mapCountries);
    int count = 0;
    Iterator<Map.Entry<String, Integer>> iter = order.entrySet().iterator();
    while (iter.hasNext()) {
        Entry<String, Integer> entry = iter.next();
        if (count >= n) {
            break;
        }
        returnedCities.add(entry.getKey());
        count++;
    }

    return returnedCities;
}
```

Figura 14- Método que determina as cidades com mais amigos de um utilizador

O método que determina os vértices intermédios chama dois métodos, o primeiro **userCitiesWithMoreFriends** tem uma complexidade de O(s x (VxE)), sendo que o segundo ciclo for é o ciclo com complexidade temporal mais significativa. O método mergeLists é um método que apenas se limita a juntar duas listas e a eliminar valores repitidos, e tem uma complexidade de O(s), logo a complexidade temporal do método completo **intermediateVertices** é de O(s x (VxE)).

2- Este ciclo limita-se a eliminar da lista que contém os vértices intermédios aqueles que são iguais ao vértice de origem e de destino (se existirem) e têm uma complexidade temporal de O(n).

3- Esta parte do método é a chave para o bom funcionamento do algoritmo. Serão calculados os caminhos mais curtos entre os vértices (por exemplo: vértice origem – intermedio1; intermedio1 – intermedio2; intermedio2 – vértice destino), fazendo desta forma o algoritmo de Djisktra que tem uma complexidade temporal de O(V x E). Depois serão inseridos a um map temporário os vértices (O(V)) e depois o caminho a distância entre o vértice atual e o anterior (O(V^2)).

Conclui-se então que o método tem uma complexidade temporal de O(V^3 x E).