Estruturas de Informação

Relatório

2º Trabalho Prático

Turma 2DL
Ricardo Mesquita 1190995
Gonçalo Jordão 1190633

isep Instituto Superior de Engenharia do Porto

Requisitos

1. Armazenamento de informação do ficheiro

Neste projeto, a classe utilizada para a leitura de informação dos ficheiros é denominada *FileReader.* Nesta classe, estão presentes todos os métodos de leitura (readUsers, readRelationships, readCountries, readBorders) e as respetivas verificações necessárias para o armazenamento de informação.

Construtor:

Esta apresenta também um construtor que possui como parâmetros constantes definidas numa classe própria (*Constants*), e o objeto da aplicação contém as estruturas de informação onde vão ser guardados todos os diferentes tipos de dados.

```
public FileReader(Application app, String FILE_USER_PATH, String FILE_COUNTRY_PATH, String FILE_RELATIONSHIP_PATH, String FILE_BORDERS_PATH) {
    this.userList = app.getUserList();
    this.countriesList = app.getCountryList();
    this.userRelationships = app.getUserRelationships();
    this.capitalsRelation = app.getCapitalsRelation();
    this.FILE_USER_PATH = FILE_USER_PATH;
    this.FILE_COUNTRY_PATH = FILE_COUNTRY_PATH;
    this.FILE_RELATIONSHIP_PATH = FILE_RELATIONSHIP_PATH;
    this.FILE_BORDERS_PATH = FILE_BORDERS_PATH;
}
```

Método readUsers:

Neste método é realizada a leitura dos usuários existentes no ficheiro *(ex: u1, 27, brasilia).* Para isso, foi utilizada a classe *User* no âmbito de criação do objeto. Paralelamente a isto, é também utilizada a classe *UsersList* para fazer o controlo/gestão de todos os usuários inseridos na mesma.

```
void readUsers() {
    try {
        BufferedReader reader = new BufferedReader(new java.io.FileReader(this.FILE_USER_PATH));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] data = line.split(Constants.FILE_SPLIT);
            User u = this.userList.newUser(data[0].trim(), data[1].trim(), data[2].trim());
            if (!this.userList.exitsUser(u)) {
                this.userList.registerUser(u);
            }
        }
    } catch (IOException io) {
        flag = false;
        System.out.println("\nERROR: File \"users\" has no information, please change the file and restart the Application!");
    } catch (ArrayIndexOutOfBoundsException ai) {
        flag = false;
        System.out.println("\nERROR: File \"users\" does not have all the necessary information, please change the file and restart the Application!");
    } catch (NumberFormatException nf) {
        flag = false;
        System.out.println("\nERROR: File \"users\" does not have the information placed correctly, please change the file and restart the Application!");
    }
}
```

Método readRelationships:

Para o seguinte método, foi usada a classe *UsersList* para retornar o usuário, passando como parâmetro o id do mesmo (*ex: u1, u2*). Seguidamente é feita uma verificação no que toca à existência de ambos os usuários. Caso existam, recorre-se a uma outra verificação, neste caso quanto à sua existência na rede de amizades (matriz de adjacências: userRelationships). Se não existirem são adicionados a essa mesma rede, onde posteriormente é feita a ligação entre os mesmos. Além disto, incrementamos uma unidade no contador de amigos de cada usuário.

```java
void readRelationships() {
    try {
        BufferedReader reader = new BufferedReader(new java.io.FileReader(this.FILE_RELATIONSHIP_PATH));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] data = line.split(Constants.FILE_SPLIT);
            User u1 = this.usersList.getUserById(data[0].trim());
            User u2 = this.usersList.getUserById(data[1].trim());
            if (this.usersList.exitsUser(u1) && this.usersList.exitsUser(u2)) {
                if (!this.usersRelationships.checkVertex(u1)) {
                    this.usersRelationships.insertVertex(u1);
                }
                if (!this.usersRelationships.checkVertex(u2)) {
                    this.usersRelationships.insertVertex(u2);
                }
                this.usersRelationships.insertEdge(u1, u2,  newEdge: 1);
                u1.addFriend();
                u2.addFriend();
            }
        }
    } catch (IOException io) {
        flag = false;
        System.out.println("\nERROR: File \"relationships\" has no information, or does not exist, please change the file and restart the Application!");
    } catch (ArrayIndexOutOfBoundsException ai) {
        flag = false;
        System.out.println("\nERROR: File \"relationships\" does not have all the necessary information, please change the file and restart the Application!");
    } catch (NumberFormatException nf) {
        flag = false;
        System.out.println("\nERROR: File \"relationships\" does not have the information placed correctly, please change the file and restart the Application!");
    }
}
```

Método readCountries:

Neste método é realizada a leitura dos países existentes no ficheiro *(ex: argentina, americasul, 41.67, buenosaires, -34.6131500, -58.3772300).* Para isso, foi utilizada a classe *Country* no âmbito de criação do objeto. Paralelamente a isto, é também utilizada a classe *CountriesList* para fazer o controlo/gestão de todos os países inseridos na mesma.

```java
void readCountries() {
    try {
        BufferedReader reader = new BufferedReader(new java.io.FileReader(this.FILE_COUNTRY_PATH));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] data = line.split(Constants.FILE_SPLIT);
            Country c = this.countriesList.newCountry(data[0].trim(), data[1].trim(), data[2].trim(), data[3].trim(), data[4].trim(), data[5].trim());
            if (!this.countriesList.exitsCountry(c)) {
                this.countriesList.registerCountry(c);
            }
        }
    } catch (IOException io) {
        flag = false;
        System.out.println("\nERROR: File \"countries\" has no information, or does not exist, please change the file and restart the Application!");
    } catch (ArrayIndexOutOfBoundsException ai) {
        flag = false;
        System.out.println("\nERROR: File \"countries\" does not have all the necessary information, please change the file and restart the Application!");
    } catch (NumberFormatException nf) {
        flag = false;
        System.out.println("\nERROR: File \"countries\" does not have the information placed correctly, please change the file and restart the Application!");
    }
}
```

Método readBorders:

Para o seguinte método, foi usada a classe *CountriesList* para retornar o país, passando como parâmetro o país do mesmo (*ex: argentina, bolivia*). Seguidamente é feita uma verificação no que toca à existência de ambos os países. Caso existam, recorre-se a uma outra verificação, neste caso quanto à sua existência na rede de fronteiras(map de adjacências: capitalsRelation). Se não existirem são adicionados a essa mesma rede, onde posteriormente é feita a ligação entre os mesmos.

```
void readBorders() {
    try {
        BufferedReader reader = new BufferedReader(new java.io.FileReader(this.FILE_BORDERS_PATH));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] data = line.split(Constants.FILE_SPLIT);
            Country c1 = this.countriesList.getCountryByCountry(data[0].trim());
            Country c2 = this.countriesList.getCountryByCountry(data[1].trim());
            if (this.countriesList.exitsCountry(c1) && this.countriesList.exitsCountry(c2)) {
                if (!this.capitalsRelation.validVertex(c1.getCapital())) {
                    this.capitalsRelation.insertVertex(c1.getCapital());
                }
                if (!this.capitalsRelation.validVertex(c2.getCapital())) {
                    this.capitalsRelation.insertVertex(c2.getCapital());
                }
                this.capitalsRelation.insertEdge(c1.getCapital(), c2.getCapital(),  einf: 1, this.countriesList.getDistance(c1, c2));
            }
        }
    } catch (IOException io) {
        flag = false;
        System.out.println("\nERROR: File \"borders\" has no information, or does not exist, please change the file and restart the Application!");
    } catch (ArrayIndexOutOfBoundsException ai) {
        flag = false;
        System.out.println("\nERROR: File \"borders\" does not have all the necessary information, please change the file and restart the Application!");
    } catch (NumberFormatException nf) {
        flag = false;
        System.out.println("\nERROR: File \"borders\" does not have the information placed correctly, please change the file and restart the Application!");
    }
}
```

2. Armazenamento dos usuários mais populários e os respetivos amigos em comum

Neste método, é utilizada uma lista para armazenar todos os usuários (Lista *Users*). Primeiramente, a lista previamente criada, é ordenada pela quantidade de amigos de cada usuário. Após isso é feita uma filtragem de acordo com o ranking introduzido pelo utilizador, copiando estes usuários para uma outra lista (*topUsersAndCommonFriends*).
Finalmente, é percorrida a lista de amigos do usuário que está em último do top (pois é o que possui menor quantidade de amigos entre eles) e é verificado para cada um se é amigo dos restantes. No caso disto se suceder, estes são adicionados à lista dos demais.

```
List<User> morePopular(int amount) {
    List<User> users = new ArrayList<>(this.userList.getUsers());
    Collections.sort(users);
    List<User> topUsersAndCommonFriends = users.subList(0, amount);

    //see Common friends
    boolean flag = true;
    for (User user : this.userRelationships.adjVertices(topUsersAndCommonFriends.get(amount - 1))) {
        for (int i = 0; i < amount - 1; i++) {
            if ((this.userRelationships.getEdge(topUsersAndCommonFriends.get(i), user) == null)) {
                flag = false;
                break;
            }
        }
        if (flag) {
            topUsersAndCommonFriends.add(user);
        } else {
            flag = true;
        }
    }
    return topUsersAndCommonFriends;
}
```

3. Verificação da conectividade da rede de amizades e respetivo armazenamento do numero mínimo de ligações para qualquer utilizador se conecta a qualquer outro

Neste caso foi criada uma *LinkedList* de utilizadores a partir de um método de procura referente ao grafo de matriz de adjacências. Com isto, no caso da quantidade de vértices existentes na rede de amizades for igual à quantidade de usuários da lista auxiliar, então está provado que esta é conexa, retornando assim o número mínimo de ligações (referente a *edges* no grafo) calculado através da distância entre os dois pontos mais afastados do grafo. No caso de isto não se suceder, é retornado o valor -1, correspondente à não-conectividade do mesmo.

```java
int minimumNumberOfConnections() {
    User u = this.userList.getUsers().get(0);
    LinkedList<User> listAux = AdjacencyMatrixAlgorithms.BFS(this.userRelationships, u);
    if (this.userRelationships.numVertices() == listAux.size()) {
        return AdjacencyMatrixAlgorithms.amountOfLayers(this.userRelationships, listAux.getLast());
    }
    return -1;
}
```

4. **Armazenamento os amigos que se encontram nas proximidades de um determinado usuário, dado um determinado número de fronteiras**

Para este método, recorreu-se à utilização de um *map* para armazenar as diferentes cidades e os respetivos amigos que se encontram nelas.
No inicio, foi realizada uma verificação para ver se a cidade do usuário possui fronteiras. Após isso, foi usado um BFS (*BreathFirstSearch*) adaptado por nós, de maneira a que fosse possível retornar as cidades que estão a n distância da cidade do usuário. De seguida, caso existam cidades nessa lista, ou seja, se esta não estiver vazia, para cada amigo do usuário em questão, é realizada uma verificação no âmbito de ver se a sua cidade se encontra na lista de cidades retornada anteriormente(*citiesBordersAway*). Caso isto se verifique, é adicionada a cidade ao *map* (na eventualidade de essa não existir) e o amigo à lista de amigos pertencentes a essa cidade.

```java
Map<String, HashSet<String>> nearbyFriends(String userId, int borders) {
    Map<String, HashSet<String>> mapAux = new HashMap<>();

    User u = this.userList.getUserById(userId);

    if (!this.capitalsRelations.validVertex(u.getCity())) {
        return mapAux;
    }

    //save the cities maximum n borders away
    LinkedList<String> citiesBordersAway = GraphAlgorithms.BreadthFirstSearch(this.capitalsRelations, u.getCity(), borders);

    //save the friends for each cities
    if (!citiesBordersAway.isEmpty()) {
        for (User user : this.userRelationships.adjVertices(u)) {
            if (citiesBordersAway.contains(user.getCity())) {
                if (mapAux.isEmpty() || !mapAux.containsKey(user.getCity())) {
                    mapAux.put(user.getCity(), new HashSet<>());
                }
                mapAux.get(user.getCity()).add(user.getID());
            }
        }
    }
    return mapAux;
}
```

5. **Armazenamento das cidades com maior centralidade e onde habitem pelo menos uma certa percentagem relativa de utilizadores da rede de amizades**

Neste método, foi utilizado um *map* para armazenar, para cada cidade, a sua respetiva distância média em relação a todas as outras cidades. Posteriormente, é também utilizada uma lista para armazenar as cidades que devem ser retornadas.
Primeiramente, é calculada a distância média, para cada cidade, para que, mais tarde, se possa ordenar essa mesma lista segundo esse fator, ordenando assim as cidades de acordo com a sua centralidade.
Seguidamente, é feito um *reset* ao *map* referido anteriormente, para que este possa ser reutilizado, uma vez que não será precisa a informação previamente guardada.
No que toca à segunda parte do problema (percentagens), esta é calculada segundo dois procedimentos distintos:
- Contar, para cada cidade, a quantidade de utilizadores pertencentes à rede de amizades;
- Realizar o calculo da percentagem segundo a seguinte fórmula:

$$\frac{qtdUsersPerCountry}{totalUsers} \times 100.$$

Por último, é armazenada a quantidade pedida de cidades com a percentagem relativa de utilizadores da rede igual ou superior à inserida.

```
List<String> citiesWithGreaterCentrality(int amountOfCities, double percentage) {
    Map<String, Double> cities = new HashMap<>();
    List<String> topCountries = new ArrayList<>();

    //save the average distance from all other cities for each city
    int count = 0;
    for (Country c : this.countriesList.getCountries()) {
        String c1 = c.getCapital();
        if (cities.isEmpty() || !cities.containsKey(c1)) {
            cities.put(c1, 0.0);
        }
        for (int i = count + 1; i < this.countriesList.getCountries().size(); i++) {
            String c2 = this.countriesList.getCountries().get(i).getCapital();
            if (!cities.containsKey(c2)) {
                cities.put(c2, 0.0);
            }
            double currentDistance = cities.get(c1);
            double distance = this.countriesList.getDistance(c, this.countriesList.getCountries().get(i));
            double newDistance = currentDistance + distance;
            cities.put(c1, newDistance);
            currentDistance = cities.get(c2);
            newDistance = currentDistance + distance;
            cities.put(c2, newDistance);
        }
        double totalDistance = cities.get(c1);
        double average = totalDistance / (this.countriesList.getCountries().size() - 1);
        cities.put(c1, average);
        count++;
    }

    //sort by average distance
    Map<String, Double> citiesSorted = cities.entrySet().stream().sorted(Map.Entry.comparingByValue()).collect(Collectors.toMap(Map.Entry::getKey,
            Map.Entry::getValue, (key, content) -> content, LinkedHashMap::new));
```

```
        //reuse the map
        citiesSorted.replaceAll((s, v) -> 0.0);

        //save the amount of users in each city
        for (User u : this.userRelationships.vertices()) {
            if (citiesSorted.containsKey(u.getCity())) {
                double amountOfUser = citiesSorted.get(u.getCity()) + 1;
                citiesSorted.put(u.getCity(), amountOfUser);
            }
        }

        //save the relative percentage of users in each city
        for (String s : citiesSorted.keySet()) {
            double individualPercentage = (citiesSorted.get(s) / this.userRelationships.numVertices()) * 100;
            citiesSorted.put(s, individualPercentage);
        }

        //save n most centrality with a percentage equal to or greater than that requested
        int contAux = 0;
        for (String s : citiesSorted.keySet()) {
            if (citiesSorted.get(s) >= percentage) {
                topCountries.add(s);
                contAux++;
            }
            if (contAux == amountOfCities) {
                break;
            }
        }
        return topCountries;
    }
```

6. **Armazenamento do caminho terrestre mais curto entre dois usuários, passando obrigatoriamente por n cidades intermédias onde cada usuário tenha o maior numero de amigos (incluindo a respetiva distância)**

Neste método, é feita uma verificação para ver se a cidade dos dois utilizadores é a mesma, pois no enunciado é referido que "*as cidades origem, destino e intermédias devem ser todas distintas*". Seguidamente, é calculada a quantidade de amigos para cada cidade de cada um dos usuários. Depois, é feita uma ordenação pela qual toma como base a quantidade de amigos. Posteriormente, verificamos o ranking de n cidades para cada um dos usuários e é feita a adição dessas cidades para uma lista (*topCities*).
Nisto, são feitas todas as permutações possíveis entre as cidades intermediárias.
De seguida, é feita uma pesquisa de todos os caminhos a começar na cidade do primeiro usuário e a terminar na cidade do segundo usuário, passando por todas as permutações previamente listadas em *allPaths*.

Finalmente, para cada caminho anteriormente encontrado, é feito o cálculo da sua distância em quilómetros, comparando com os restantes, de maneira a que fique apenas o caminho de menor distância.

```
Map<LinkedList<String>, Double> shortestPathBetweenTwoUsers(String userId1, String userId2, int amount) {
    Map<LinkedList<String>, Double> pathsWithNecessaryCities = new HashMap<>();

    User u1 = this.userList.getUserById(userId1);
    String cityUser1 = u1.getCity();
    User u2 = this.userList.getUserById(userId2);
    String cityUser2 = u2.getCity();

    //two users who are from the same city cannot be done, because all cities must be different
    //if the city of one of the users doesn't have in the graph, it means that there are no connections so it's impossible to connect them
    if (cityUser1.equalsIgnoreCase(cityUser2) || !this.capitalsRelations.validVertex(u1.getCity()) ||
            !this.capitalsRelations.validVertex(u2.getCity()) || !GraphAlgorithms.BreadthFirstSearch(this.capitalsRelations, cityUser1, cityUser2)) {
        return pathsWithNecessaryCities;
    }

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

    Map<String, Integer> citiesUser1 = this.getAmountOfFriendsPerCountry(u1);
    Map<String, Integer> citiesUser2 = this.getAmountOfFriendsPerCountry(u2);

    Map<String, Integer> citiesUser1Sorted = this.sortMap(citiesUser1,  sortType: 1);
    Map<String, Integer> citiesUser2Sorted = this.sortMap(citiesUser2,  sortType: 1);

    //save the u1 top friends
    int contAux = 0;
    for (String s : citiesUser1Sorted.keySet()) {
        if (!s.equals(cityUser1) && !s.equals(cityUser2)) {
            topCities.add(s);
        }
        contAux++;
        if (contAux == amount) {
            break;
        }
    }
```

```
    //save the u2 top friends
    contAux = 0;
    for (String s : citiesUser2Sorted.keySet()) {
        if (!topCities.contains(s) && !s.equals(cityUser1) && !s.equals(cityUser2)) {
            topCities.add(s);
        }
        contAux++;
        if (contAux == amount) {
            break;
        }
    }

    //save the shortest path
    if (!topCities.isEmpty()) {
        List<List<String>> allPaths = this.permute(topCities);
        LinkedList<String> path;
        LinkedList<String> pathAux;
        LinkedList<String> minimumPath = new LinkedList<>();
        double minimumDistance = 500000000;
        double individualDistance;
        boolean flag;
        for (List<String> list : allPaths) {
            flag = true;
            path = new LinkedList<>();
            path.add(cityUser1);
            individualDistance = 0;
            for (String s : list) {
                pathAux = new LinkedList<>();
                double distance = GraphAlgorithms.shortestPath(this.capitalsRelations, path.getLast(), s, pathAux);
                individualDistance = individualDistance + distance;
                if (individualDistance > minimumDistance || distance == 0) {
                    flag = false;
                    break;
                }
                pathAux.removeFirst();
                path.addAll(pathAux);
            }
```

```
            }
            if (flag) {
                pathAux = new LinkedList<>();
                double distance = GraphAlgorithms.shortestPath(this.capitalsRelations, path.getLast(), cityUser2, pathAux);
                individualDistance = individualDistance + distance;
                if (distance != 0) {
                    pathAux.removeFirst();
                    path.addAll(pathAux);
                    if (individualDistance < minimumDistance) {
                        minimumPath = new LinkedList<>(path);
                        minimumDistance = individualDistance;
                        pathsWithNecessaryCities.clear();
                        pathsWithNecessaryCities.put(minimumPath, minimumDistance);
                    }
                }
            }
        }
    }
    return pathsWithNecessaryCities;
}
```

7. Outras Classes Utilizadas

Classe Constants:

Esta classe foi utilizada meramente para guardar variáveis que foram consideradas constantes ao longo de todo o projeto.

Assim, caso fosse necessário alterar o valor de alguma destas variáveis, bastava apenas aceder a esta classe.

```
public class Constants {

    public static final String FILE_S_USER_PATH = "Files\\smallNetwork\\susers.txt";
    public static final String FILE_S_COUNTRY_PATH = "Files\\smallNetwork\\scountries.txt";
    public static final String FILE_S_RELATIONSHIP_PATH = "Files\\smallNetwork\\srelationships.txt";
    public static final String FILE_S_BORDERS_PATH = "Files\\smallNetwork\\sborders.txt";
    public static final String FILE_B_USER_PATH = "Files\\bigNetwork\\busers.txt";
    public static final String FILE_B_COUNTRY_PATH = "Files\\bigNetwork\\bcountries.txt";
    public static final String FILE_B_RELATIONSHIP_PATH = "Files\\bigNetwork\\brelationships.txt";
    public static final String FILE_B_BORDERS_PATH = "Files\\bigNetwork\\bborders.txt";
    public static final String FILE_SPLIT = ",";
}
```

Classe User:

Nesta classe foram guardados os atributos de cada usuário.

Foi decidido criar um atributo denominado “amountOfFriends” para facilitar a resolução do segundo exercício proposto.

```
public User(String id, String age, String city) {
    this.id = id;
    this.age = Integer.parseInt(age);
    this.city = city;
    this.amountOfFriends = 0;
}
```

Classe Country:

Nesta classe foram guardados os atributos de cada cidade incluída no ficheiro.

```
public Country(String country, String continent, String population, String capital, String latitude, String longitude) {
    this.country = country;
    this.continent = continent;
    this.population = Float.parseFloat(population);
    this.capital = capital;
    this.latitude = Double.parseDouble(latitude);
    this.longitude = Double.parseDouble(longitude);
}
```

Classe UsersList:

Esta classe é responsável pelo armazenamento de todos os usuários presentes no ficheiro.

No âmbito de facilitar o modo como se iria encontrar determinados utilizadores, foi optado por criar um método que conseguisse retornar toda a informação referente aos mesmos a partir do seu id.

```java
public class UsersList {

    public List<User> m_UserLst;

    public UsersList() { this.m_UserLst = new ArrayList<>(); }

    public List<User> getUsers() { return this.m_UserLst; }

    public User newUser(String id, String age, String city) { return new User(id, age, city); }

    public void registerUser(User m_oUser) { this.m_UserLst.add(m_oUser); }

    public boolean exitsUser(User m_oUser) { return this.m_UserLst.contains(m_oUser); }

    public User getUserById(String userID) {
        for (User user : m_UserLst) {
            if (user.getID().equals(userID)) {
                return user;
            }
        }
        return null;
    }
}
```

Classe CountriesList:

Esta classe é responsável pelo armazenamento de todas os países presentes no ficheiro.

No âmbito de facilitar o modo como se iria encontrar determinados países, foi optado por criar um método que conseguisse retornar toda a informação referente aos mesmos a partir do seu nome.

Além disso, foi introduzido também um método que calculasse o valor da distância em quilómetros entre dois países, a partir de dois dos seus atributos (latitude e longitude).

```java
public class CountriesList {

    public List<Country> m_CountryLst;

    public CountriesList() { this.m_CountryLst = new ArrayList<>(); }

    public List<Country> getCountries() { return this.m_CountryLst; }

    public Country newCountry(String country, String continent, String population, String capital, String latitude, String longitude) {
        return new Country(country, continent, population, capital, latitude, longitude);
    }

    public void registerCountry(Country m_oCountry) { this.m_CountryLst.add(m_oCountry); }

    public boolean exitsCountry(Country m_oCountry) { return this.m_CountryLst.contains(m_oCountry); }

    public Country getCountryByCountry(String country) {
        for (Country c : m_CountryLst) {
            if (c.getCountry().equals(country)) {
                return c;
            }
        }
        return null;
    }

    public double getDistance(Country c1, Country c2) {
        double lat1 = c1.getLatitude();
        double lon1 = c1.getLongitude();
        double lat2 = c2.getLatitude();
        double lon2 = c2.getLongitude();
        if ((lat1 == lat2) && (lon1 == lon2)) {
            return 0;
        } else {
            double theta = lon1 - lon2;
            double dist = Math.sin(Math.toRadians(lat1)) * Math.sin(Math.toRadians(lat2)) + Math.cos(Math.toRadians(lat1)) *
                    Math.cos(Math.toRadians(lat2)) * Math.cos(Math.toRadians(theta));
            dist = Math.acos(dist);
            dist = Math.toDegrees(dist);
            dist = dist * 60 * 1.1515;
            dist = dist * 1.609344;
            return dist;
        }
    }
}
```

8. Análise de complexidade

Métodos a analisar	***Complexidade***
readUsers	O(n)
readRelationships	$O(n^2)$
readCountries	O(n)
readBorders	$O(n^2)$
morePopular	*Algoritmo não-determinístico* Melhor caso: O(n) Pior caso: $O(n^2)$

minimumNumberOfConnections	$O(n^2)$
nearbyFriends	*Algoritmo não-determinístico* Melhor caso: O(1) Pior caso: $O(n^2)$
citiesWithGreaterCentrality	*Algoritmo não-determinístico* Melhor caso: O(n) Pior caso: $O(n^2)$
shortestPathBetweenTwoUsers	*Algoritmo não-determinístico* Melhor caso: O(1) Pior caso: $O(2^n)$

9. Conclusões e outras informações relevantes

Inicialmente, foi optado por construir os alicerces do código, baseados nos ficheiros fornecidos (smallNetwork). No entanto, à medida que os métodos de leitura iam sendo realizados, ficou inerente o facto de que muitos daqueles dados não iriam ser possíveis de armazenar devido à ausência de alguma informação por parte dos ficheiros de testes. Para resolver o problema em questão, apenas foram adicionados ao grafo de amizades os utilizadores que possuíam amigos. O mesmo se sucedeu para as cidades, pois no grafo só seriam adicionadas as que possuíssem fronteiras. Mais tarde, e após terem sido fornecidos outros ficheiros (bigNetwork), ficou decidido que, no menu inicial, iria ficar uma opção de escolha a realizar por parte do utilizador (ficando os ficheiros de smallNetwork como *default*).