Instituto Superior de Engenharia do Porto

Licenciatura em Engenharia Informática
ESINF – 2020/2021

# Relatório Estruturas de Informação

Projeto 2 – Rede Social

**Autores:**
[1191097] [Tiago Machado]
[1191111] [Tomas Flores]

**Turma:** 2DL

**Data:** [26/12/20]

# Introdução

Foi-nos proposto, para o segundo projeto da disciplina de Estruturas de Informação, o desenvolvimento de uma aplicação de uma rede social simplificada que gere uma rede de amigos bem como a rede de cidades onde estes habitam e as distancias entre estas. Para tal, aplicamos os conhecimentos adquiridos nesta disciplina sobre a Java Collections Framework e Grafos para dar uma resposta mais eficiente aos requisitos do projeto.

***1º Requisito - Construir os grafos a partir da informação fornecida nos ficheiros de texto. A capital de um país tem ligação direta com as capitais dos países com os quais faz fronteira. O cálculo da distância em Kms entre duas capitais deverá ser feito através das suas coordenadas.***

Para este primeiro requisito, decidimos armazenar a informação recolhida nos ficheiros em duas classes: User para os utilizadores e Country para os países. Ao longo dos métodos *lerFicheiroUsers()* e *lerFicheiroCountries()* estes objetos vão sendo guardados em *arraylist's, listaUsers* e *listaCountries*. De seguida, lemos os ficheiros das relações e das fronteiras, onde verificamos se os objetos que estão a ser adicionados como vértices existem nos *arraylists*, se existirem esses vértices são adicionados ao grafo bem como a suas aresta. Para o caso das relações de amizade utilizamos um grafo usando a matriz de adjacências. Para caso das cidades foi usado um map de adjacências, onde o valor da aresta entre dois vértices é a distancia em Km entre os mesmos.

Por fim, uma vez concluído o método *lerFicheiro()*, os dados dos utilizadores e países estarão armazenados em dois arraylist's, *listaUsers* e *listaCountries* e as relações e fronteiras estarão organizadas em duas redes, *relatioshipsNetwork* e *bordersNetwork.* Assim, se um utilizador ou pais não possuir nenhum relacionamento ou fronteira, respetivamente, os dados dos mesmos serão na mesma guardados no sistema para ligações futuras.

**Analise de complexidade -**
*lerFicheiroUsers()*- O(V)
*lerFicheiroCountries()*- O(V)
*lerFicheiroRelationships()*- $O(V^3)$
*lerFicheiroBorders()*- O(V)

***2º Requisito - Devolver os amigos comuns entre os n utilizadores mais populares da rede. A popularidade de um utilizador é dada pelo seu número de amizades.***
Para este requisito temos o método *getAmigosPopulares(),* que vai criar um *HashMap* em que a *key* é um utilizador e o *value* é o numero de amigos desse utilizador. Este *HashMap* vai ser preenchido com todos os utilizadores e o seu respetivo número de amigos. Depois o método *sortByValue()* vai ordenar o HashMap por valor num *LinkedHashMap* preservando apenas os n utilizadores mais populares.
De seguida é criada uma lista de todos os amigos do utilizador mais popular. Os valores desta lista são comparados com os de outras listas auxiliares, contendo os amigos dos n-1 utilizadores mais populares. Caso a lista do utilizador mais popular contenha amigos que não estão incluídos numa das listas auxiliares, estes são removidos, de modo a preservar apenas os amigos em comum entre os utilizadores.
**Analise de complexidade -** O(V log V)

***3. Requisito - Verificar se a rede de amizades é conectada e em caso positivo devolver o número mínimo de ligações necessário para nesta rede qualquer utilizador conseguir contactar um qualquer outro utilizador.***

Como sabemos, um grafo não dirigido é conexo se existir um caminho entre qualquer par de vértices. Sabendo isto, usamos o algoritmo *Breadth First Search,* que realiza uma busca exaustiva num grafo, passando por todas as arestas e vértices. Assim, se o tamanho do caminho que o *bfs* nos der for igual ao numero de vértices que tem o grafo, esse grafo é conexo.

Se o grafo for conexo passamos para a segunda parte deste requisito: ***devolver o número mínimo de ligações para qualquer utilizador conseguir contactar qualquer outro utilizador***. Para esta parte, temos de achar o caminho mais longo do grafo e depois, com os dois vértices extremos do caminho mais longo utilizamos o algoritmo *shortest path*, que como o nome indica devolve o caminho mais curto entre estes vértices. Recorremos a dois bfs, o primeiro para encontrar o vértice final do caminho mais longo e o segundo para, a partir deste vértice final encontrar o verdadeiro caminho mais longo. Por fim, como já dito anteriormente utilizamos o *shortest path* com o primeiro e último vértice do caminho encontrado anteriormente, obtendo assim o numero mínimo de ligações necessário.

**Analise de complexidade -** O(V+E)

***4. Requisito - Devolver para um utilizador os amigos que se encontrem nas proximidades, isto é, amigos que habitem***
***em cidades que distam um dado número de fronteiras da cidade desse utilizador. Devolver para cada cidade os respetivos amigos.***

Para este requisito temos o método *amigosProximidades()*, que vai criar um *HashMap*, em que a *key* é uma cidade e o *value* é o número de amigos do utilizador escolhido nessa cidade. Para preencher este *map,* vamos criar um *arraylist,* que vai ser iniciado com a cidade do utilizador escolhido, e é depois preenchido com as cidades cujo pais faz fronteira com os países da cidade da lista. Isto é repetido *nFronteiras* vezes, de modo a obter as cidades que se encontram nas proximidades. Depois, para cada cidade vai se obter uma lista dos amigos de utilizador, e estes items são postos no *HashMap*.

**Analise de complexidade -** O($V^2E$)

***5. Requisito - Devolver as n cidades com maior centralidade ou seja, as cidades que em média estão mais próximas de todas as outras cidades e onde habitem pelo menos p% dos utilizadores da rede de amizades, onde p% é a percentagem relativa de utilizadores em cada cidade.***

Para este requisito, e de maneira a ser o mais eficiente possível, logo depois da leitura dos dados decidimos calcular a centralidade para todos os países do *array listaCountries*, minimizamos assim cálculos excessivos para este requisito. Mais tarde, quando o utilizador quiser utilizar esta funcionalidade, pedimos-lhe que insira o número de cidades que deseja visualizar e a percentagem mínima de utilizadores que estas cidades devem de ter.

Depois, recorremos ao método *cidadesComPUtilizadores(),* onde obtemos apenas as cidades que tenham uma percentagem de utilizadores igual ou maior á percentagem inserida pelo utilizador. Por fim, ordenamos a lista obtida pela média de proximidade dos países.

**Analise de complexidade –**
*printExercicio5()*- $O(V^2)$

***6. Requisito - Devolver o caminho terrestre mais curto entre dois utilizadores, passando obrigatoriamente pelas n cidade(s) intermédias onde cada utilizador tenha o maior número de amigos. Note que as cidades origem, destino e intermédias devem ser todas distintas. O caminho encontrado deve indicar as cidades incluídas e a respetiva distância em km.***

Para este requisito temos o método *caminhoTerrestreCurto()* que vai começar por ir buscar os utilizadores origem e destino e adiciona-los à *List* userList e as suas cidades à *List* userList. Depois, para ambos os utilizadores, é criada uma *List* de amigos, a qual é adicionada os amigos do utilizador. Esta lista é usada para criar um *HashMap* que vai conter uma cidade como *key* e o número de amigos do utilizador nessa cidade como *value*. Depois o método *sortByValue2()* vai ordenar o HashMap por valor num *LinkedHashMap* preservando apenas n cidades intermédias. A seguir, as cidades são extraídas para um *ArrayList* de cidades, que não aceita cidades repetidas e contém a cidade origem como primeiro valor do array, e a cidade destino como segundo. Através desta lista, fazemos um novo *ArrayList* que vai conter os países correspondentes às cidades. Fazemos depois uma cópia desta *List* e removemos os 2 primeros valores (pais origem e pais destino). Ficamos assim como uma lista que vai conter os países intermédios que não contem duplicados e não contem também os países de origem.

Para cada pais intermedio, vamos invocar o método *shortestPath()*, que vai retornar o caminho mais curto entre o pais origem e o pais intermedio em questão. Deste ciclo são apenas preservados os dados do pais intermedio com a distancia mais curta da origem, sendo estes dados a distancia e o percurso. Este pais vai também ser retirado da *List* de países intermédios. Todo este processo vai ser repetido, mas em vez de utilizar o pais origem é o utilizado o pais que continha a distancia mais curta e o array de países intermédios não irá conter o pais em questão. Quando o array de países intermédios se encontrar vazio, vamos fazer o *shortestPath()* entre o ultimo pais intermedio a ter sido removido e o pais destino. Vamos assim obter o menor percurso terrestre possível e a distancia deste.

**Analise de complexidade –** O((V+E)*V)