Realizado por:

Fábio Fernandes 1191430

Bárbara Pinto 1191507

Síntese

Ao longo do documento iremos demonstrar o modo como gerimos a informação que nos é fornecida através de um ficheiro de Excel, bem como a explicação dos nossos métodos para responder aos diferentes requerimentos que a aplicação necessitaria ter, tendo em conta a eficiência, mostrando também a análise de complexidade de todas as funcionalidades implementadas.

Relatório de estruturas de informação

Projeto 2 – Rede Social

**Introdução**

Na unidade curricular de Estruturas de Informação foi-nos proposto, para segundo projeto, desenvolver uma aplicação que permita gerir uma rede social de amigos, com base numa rede de cidades onde habitam.

Organizamos a informação disponível em ficheiros *.txt* contendo separadamente os utilizadores, amizades, países e fronteiras, aplicando os conhecimentos adquiridos na Java Collections Framework e recorrendo a classes genéricas e herança para responder eficientemente aos requisitos que nos foram colocados ao longo deste projeto.

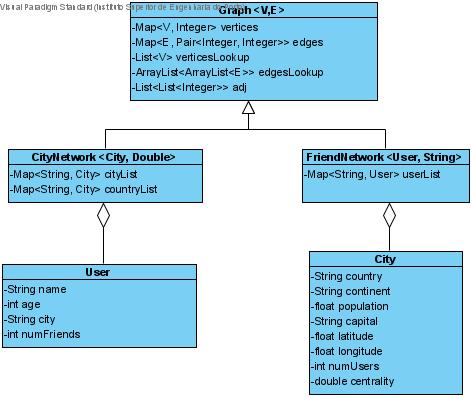


Figura 1: Diagrama de classes do projeto.

## Requisito nº1 – Carregar e guardar informação dos ficheiros de texto

Para este primeiro procedimento decidimos armazenar a informação recolhida dos ficheiros de texto que nos foram disponibilizados para o projeto em 4 classes, sendo elas: *City*, *CityNetwork*, *FriendNetwork* e *User*.

Neste primeiro procedimento decidimos utilizar as classes *City* e *User* para guardar todo o tipo de atributos, tanto geográficos (*countries.txt*) como de utilizadores (*users.txt*). De notar que optamos por criar a classe *City* - definida primeiramente pelo nome da cidade - e não *Country* - definida pelo nome do país - uma vez que a estrutura de informação fornecida e os requisitos da aplicação se regem principalmente pelas cidades de residência dos utilizadores, que assumimos serem únicas, ou seja, não existem duas capitais com o mesmo nome e cada país só tem uma cidade relevante (a capital).

Ambas as classes *CityNetwork* e *FriendNetwork* representam grafos e, por isso, herdam a estrutura e todos os métodos da classe *Graph*. Cada classe adiciona à estrutura de *Graph* especificidades necessárias ao correto processamento da informação: a classe *CityNetwork* efetua o cálculo do peso de cada aresta inserida através do *insertEdge()*, por exemplo, enquanto que a classe *FriendNetwork* incrementa o número de amigos de cada utilizador sempre que uma aresta é inserida através do mesmo método. Tal permite o uso eficiente de linhas de código (evita a existência de blocos idênticos de código em classes diferentes), conservando a capacidade de personalização para cada tipo de grafo.

A classe *Graph* é uma classe genérica que contém todos os métodos genéricos necessários para os grafos construídos neste projeto. É nesta classe que se inserem os algoritmos que implementam os conceitos mais importantes de busca da teoria de grafos, tais como: breathFirstSearch, depthFirstSearch e shortestDistance (algoritmo de Dijsktra), que são necessários para as soluções apresentadas.

## Requisito nº2 – Devolver os amigos comuns entre os n utilizadores mais populares da rede. A popularidade de um utilizador é dada pelo seu número de amizades.

Neste requisito, seguimos o seguinte processo lógico para obter a solução final:

## Obter os utilizadores mais populares (com mais amigos) da rede.

De modo a ordenar da forma mais eficiente possível os utilizadores pelo seu número de amigos, decidimos adicionar o atributo *numFriends* – número total de amigos – aos objetos da classe *User*, que é incrementado sempre que uma aresta – ou amizade – é inserida no grafo da rede social e decrementado sempre que uma aresta é removida.

Assim, com a criação de um *Comparator<User>* que ordene os utilizadores pelo seu número de amigos de forma decrescente, bastou recorrer ao método *Collections.sort()* – que utiliza o *merge* sort, com complexidade temporal *O(n log(n))* – para obter a lista de todos os utilizadores, ordenados do mais popular ao menos popular, sendo que os *n* mais populares correspondem aos *n* primeiros índices.

## Obter os amigos em comum de dois utilizadores.

Neste passo, escolhemos alavancar as funcionalidades dos conjuntos em Java (*HashSet*)de modo a eliminar automaticamente amigos duplicados e fazer uma interseção fácil através do método *retainAll()* – a documentação oficial da Oracle não indica a complexidade temporal deste método, mas aparenta ser *O(n)*.

## Fazer interseções sucessivas para obter os amigos em comum de todos os pares de utilizadores.

Com o ponto anterior, recorremos então a um método de sucessivas interseções dos amigos em comum de cada par de utilizadores, e de seguida de cada par de resultados dessas interseções, de forma recursiva. O processo assume a forma de uma árvore invertida e foi escolhido pela maior eficiência comparado a um método iterativo:

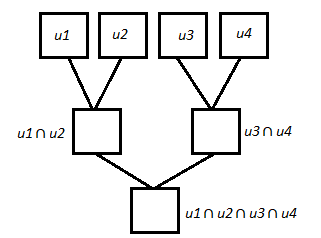


Figura 2: Esquema do método friendsInCommon(List<User>, int).  
(Cada quadrado representa o conjunto de amigos do(s) utilizador(es) identificados)

## Requisito nº3 – 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.

Para este requisito utilizamos o método *breadthFirstSearch* em primeiro lugar - com complexidade temporal *O(V+E)* - de maneira a verificar se o grafo é conexo. Com o uso deste método, conseguimos verificar se o número de vértices alcançados ao percorrer todas as arestas era igual ao número total de vértices do grafo.

Depois de verificar se a rede de amizades é conectada, é necessário verificar qual o caminho mais curto entre os pontos mais distantes do grafo. Para este processo, utilizamos novamente o método *breadthFirstSearch*, desta vez em duplicado: O primeiro BFS encontra o vértice limite do caminho mais longo a partir do vértice inicial (índice 0). Já o segundo BFS encontra o vértice limite do caminho mais longo a partir da primeira busca para encontrar o verdadeiro caminho mais longo.

Depois de obter esses dois pontos, utilizamos o algoritmo de Dijsktra – com complexidade temporal *O(E\*log(V))*, pelo método *shortestDistance*, de maneira a obter a distância (neste caso, o número de arestas) entre os dois vértices limite do caminho encontrado acima.

## Requisito nº4 – 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 responder a este requisito, seguimos os seguintes passos:

## Procurar as cidades que distam até n fronteiras da cidade do utilizador.

Para o efeito é utilizado o método *searchByLayers()*, que corresponde a um *breadthFirstSearch* com um *n* de paragem e um vetor adicional – *level[]* – que mantém registo da “camada” ou nivel do círculo concêntrico em que se insere cada vértice.

## Ordenar os amigos do utilizador pela cidade em que estes habitam.

Através do método *getNearestFriends()* faz-se uma seleção iterativa – em tempo *O(n)* – de todos os amigos que habitem numa das cidades encontradas no passo anterior através da funcionalidade *contains* presente na classe *ArrayList* – também ela executada em *O(n).*

Finalmente, e tal como em outros métodos, recorremos a um *Comparator* e ao *Collections.sort()* de modo a agrupar os amigos pela sua cidade de origem com complexidade *O(n log(n))*.

## Requisito nº5 – 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 esta etapa, decidimos adicionar dois atributos adicionais à classe *City*: *numUsers* (o número de utilizadores da rede social que habitam na cidade) e *centrality* (a soma das distâncias da cidade a todas as outras cidades inseridas na *CityNetwork*).

O primeiro atributo serve para obter, de forma eficiente, a percentagem relativa de utilizadores em cada cidade, sendo incrementado sempre que um utilizador é inserido na rede social, através do método *insertVertex()*.

O método *getUserPercentage()* é uma solução iterativa, em *O(n)*, para percorrer os *n* primeiros índices de uma dada lista de cidades e retornar aqueles que possuem um número de utilizadores superior a *p*, onde o *p* é convertido para valor absoluto multiplicando a percentagem inicial pelo número total de utilizadores.

Já o atributo *centrality* consiste uma solução para ordenar de forma mais eficiente os vários países pela sua centralidade. Recorremos ao método *getCentralities()*, cuja complexidade é *O(n2)*, uma vez que é necessário para cada cidade percorrer todas as outras e calcular a soma das respetivas distâncias.

A ordenação é feita, mais uma vez, com recurso a um *Comparator* para obter o menor tempo possível de execução e aceder facilmente às cidades com menor centralidade através dos primeiros índices da lista que daí resulta.

## Requisito nº 6 – 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.

O último requisito da aplicação mostrou ser o mais desafiante, uma vez que sendo um problema NP-completo, com alta complexidade computacional, não pode ser resolvido de forma eficiente (de facto, verificamos que quando aplicado à *big network* não gera output em tempo útil, o que consideramos normal dado a natureza do problema).

A solução ao problema envolveu a criação do método *allPaths()* que calcula todos os caminhos possíveis entre os vértices de origem e de destino. Tal como indica o enunciado, e de forma a reduzir de certa forma o tempo de execução, apenas são considerados caminhos em que os vértices origem, destino e intermédios são todos distintos.

Após encontrar todos os caminhos, é ainda necessário aplicar um algoritmo iterativo que procure um caminho que contenha todas as cidades intermédias necessárias, utilizando para o efeito o método *containsAll()* da *Java Collections Framework*, resultando numa complexidade *O(n2).*

**Complexidades (Resumo)**

Apresentamos abaixo a complexidade temporal estimada de todos os métodos implementados nas classes (exceto getters, setters, construtores e métodos override).

***Classe City:***

Algoritmos:

1. distanceFrom: *O(1)*
2. addUser: *O(1)*

***Classe FriendNetwork:***

Algoritmos:

1. getUser: *O(1)*
2. usersByPopularity: *O(n log(n))*
3. friendsInCommon(User, User): *O(n)*
4. friendsInCommon(List<User>, int): *O(n log(n))*
5. intersections: *O(log(n))*
6. getNearestFriends: *O(n)*
7. getFriendsByCity: *O(n log(n))*
8. getTopCitiesByNumFriends: *O(n)*

***Classe CityNetwork:***

Algoritmos:

1. getCity: *O(1)*
2. getCountry: *O(1)*
3. convert: *O(n)*
4. getCitiesBorders: *O(n)*
5. getCentralities: *O(n2)*
6. getUserPercentage: *O(n)*

***Classe User:***

Algoritmos:

Todos os algoritmos que estão presentes na classe *User* tem complexidade O(1).

***Classe Graph:***

Algoritmos:

1. outDegree: *O(V+E)*
2. outgoingEdges: *O(V)*
3. getAdjacentVertices: *O(V)*
4. insertVertex: *O(V)*
5. insertEdge: *O(1)*
6. removeVertex: *O(1)*
7. isConnected: *O(V)*
8. shortestPath: *O(V)*
9. breadthFirstSearch: *O(V+E)*
10. searchByLayers: *O(V)*
11. depthFirstSearch: *O(V+E)*
12. allPaths: *O(V2 \* V!)*
13. getPathAcrossAllVertices: *O(V2)*
14. longestPath: *O(V)*
15. pathLength: *O(V+E)*

Todos os outros algoritmos não mencionados são de complexidade *O(1)*.

Conclusão

Com este trabalho ficamos a compreender todo o tipo de métodos necessários para regularizar e trabalhar com grafos de maneira eficiente, podendo nem sempre o alcançar (exemplo do método allPaths), algo que futuramente pode se mostrar essencial recorrer.

Com base na revisão do trabalho anterior, o nosso objetivo para este trabalho era responder aos pontos mencionados, sendo assim, decidimos melhorar a cobertura dos testes das nossas classes (optando por utilizar os ficheiros .*txt* para recolher e abranger todos os dados), bem como aumentar o número de testes para cobrir todas as possibilidades.

Sem falta mencionar que a utilização da Java Collections Framework se mostrou bastante útil para esta etapa, assim como, a explicação dos professores aos diferentes requisitos perante o enunciado fornecido.