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.

Organizando a informação disponível em ficheiros txt contendo os utilizadores, relações, países e fronteiras, assim como aplicar os conhecimentos adquiridos na Java Collections Framework e recorrendo a classes genéricas para responder eficientemente aos requisitos que nos foram colocados ao longo deste projeto.

1. ***Requisito nº1 – Carregar e guardar informação do ficheiro Excel.***

Para este primeiro procedimento decidimos armazenar a informação recolhida dos ficheiros Excel que nos foi disponibilizado na plataforma moodle em 5 classes, sendo elas: *City*, *CityNetwork*, *FriendNetwork*, *Graph* e *User*.

Neste primeiro procedimento decidimos utilizar as classes *City* e *User* para guardar todo o tipo de atributos que sejam atribuídos tanto a uma cidade como a um utilizador, de notar que optamos por criar a classe *City* e não *Country* porque neste caso só teremos uma cidade associada a cada país, ou seja, irá nos facilitar bastante o uso de cidades como vértices do grafo em comparação com os vértices como países.

A classe *CityNetwork* e *FriendNetwork* são ambas classes que recorrem à classe *Graph*, estas contêm todos os métodos necessários para intervir a um nível específico dentro do grafo correspondente.

A classe *Graph* é uma classe genérica e contêm todos os métodos genéricos necessários para todos os grafos construídos neste projeto. É nesta classe que se insere métodos como por exemplo: breathFirstSearch, shortestDistance, getAdjacentVertices entre outros, que são necessários para as soluções apresentadas, de forma a responder aos diversos requisitos que nos são propostos neste projeto.

1. ***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 decidimos primeiramente comparar os utilizadores pelo seu número de amigos, verificamos para cada vértice do grafo a quantidade de amigos que ele possui e inserimos numa lista, lista essa que vai conter os utilizadores ordenados por ordem decrescente de popularidade. Para isso decidimos adicionar um atributo ao utilizador denominado “numFriends”.

De seguida, recorremos a um método de sucessivas interseções para obtermos os utilizadores que são em comum com os n utilizadores mais populares, isto é, ao recorrer à lista previamente mencionada, faríamos interseções de 2 em 2 até ao n utilizador mais popular, com isto aumentamos a nossa eficiência porque não necessitamos de comparar todos os utilizadores da lista entre si.

1. ***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, de maneira a perceber se realmente o grafo era conexo, com o uso deste método, conseguíamos verificar se o número de vértices conectados era igual ao número de vértices do grafo.

Depois de verificar se a rede de amizades era conectada, necessitamos de verificar qual o caminho mais curto para os pontos mais distantes do grafo, para este processo, utilizamos novamente o método breadthFirstSearch mas desta vez 2 vezes, isto porque ao partir de um vértice qualquer conseguimos verificar o ponto mais distante e realizando o método novamente nesse ponto conseguimos com certeza verificar qual os pontos mais distantes no grafo.

Depois de obter esses dois pontos, utilizamos o método shortestDistance de maneira a obter o caminho mais curto entre esses 2 utilizadores e verificar quantas ligações são feitas para obter esse caminho.

1. ***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 recorremos a vários métodos de maneira a obter a resposta necessária. Recorremos inicialmente a um método para recolher o User inserido pelo utilizador. De seguida verificamos qual a cidade desse utilizador, para posteriormente verificar quais as cidades que estão a seu redor (que fazem fronteira com a cidade do user), utilizando um método “searchByLayers” que corresponde a um breadthFirstSearch com um n de paragem. Obtendo uma lista de cidades, vamos através do método “getNearestFriends” recorremos à lista de cidades em proximidade e ao user e verificamos se para os amigos em adjacência estão inseridos numa das cidades em proximidade ficando com uma lista de users por cidade.

A seguir só é necessário organizar a informação recolhida, através de um sort, que vai essencialmente organizar a lista alfabeticamente, ou seja, por cidades.

1. ***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 2 atributos adicionais à classe *Cidade*, “numUsers” e ”centrality” isto porque para a segunda parte necessitamos de verificar quantos utilizadores existem em cada cidade e depois descobrir a percentagem relativa a todas as cidades existentes. Para isso utilizamos o método getUserPercentage que vai retornar os valores pretendidos.

Para a primeira parte desta etapa com o atributo “centrality”, recorremos a um método getCentralities, este método é um pouco complexo, isto porque à medida que vamos ler uma cidade do ficheiro para o grafo, vamos para cada outra cidade atualizar a sua centralidade.

Depois utilizamos um comparator para podermos organizar a lista da centralidade das cidades, de maneira a obter o menor valor como o topo, porque o que tiver menor média de distâncias será a mais centralizada.

1. ***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.***

Complexidades

***Classe City:***

Algoritmos:

1. distanceForm: O(1)
2. addUser: O(1)
3. updateCentrality: O(1)
4. equals: O(1)

***Classe FriendNetwork:***

Algoritmos:

1. usersByPopularity: O(n)
2. friendsInCommon: O(n)
3. friendsInCommon: n x log(n)
4. intersections: log(n)
5. getNearestFriends: O(n)

Os restantes métodos não mencionados tem a complexidade O(1).

***Classe User:***

Algoritmos:

Todos os algoritmos que estão presentes na classe *User* tem complexidade O(1), isto porque são métodos get e set.

***Classe CityNetwork:***

Algoritmos:

1. getCitiesByBorders: O(V)
2. getCentralities: O(n)
3. getUserPercentage: O(n)
4. insertVertex: O(n)
5. removeVertex: O(n)
6. insertEdge: O(1)

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

***Classe Graph:***

Algoritmos:

1. outDegree: O(V)
2. insertVertex: O(V)
3. removeVertex: O(V)
4. getAdjacentVertices: O(V)
5. isConnected: O(V)
6. longestPath: O(V)
7. breadthFirstSearch: O(V)
8. searchByLayers: O(V)
9. shortestPath: O(V)
10. shortestDistance: O(V)

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

Conclusão

Com este trabalho ficamos a conhecer de forma mais aprofundada as ferramentas da *Java Collections Framework*, aplicando esses conhecimentos através do uso de *Maps* e indexação *Key -> Value* para tornar pesquisas mais eficientes, assim como o uso de *TreeMaps* e o método *Collections.sort* para garantir a rápida ordenação de dados.

Podemos concluir que as ferramentas da *Java Collections Framework* realmente são boas para pôr em prática porque não só conseguem proporcionar uma melhor organização de toda a informação, como podem melhorar significativamente a eficiência do programa que estamos a construir.