Algoritmos e Estruturas de Dados

3ª Série

(Problema)

Exercício 2

46973 Jorge Alexandre Luzio Simões

47199 Tiago Luís Lima da Silva

Licenciatura em Engenharia Informática e de Computadores

Semestre de Inverno 2020/2021

01/02/2021

1. Introdução

Pretende-se desenvolver uma aplicação que permita realizar um conjunto de operações que tem como resultado o menor caminho entre 2 pontos num determinado plano cuja informação estará dividida em 2 ficheiros.

- recebe como parâmetro dois ficheiros de texto (o primeiro com extensão gr e o 2 com extensão .co);
- No ficheiro .gr estão as ligações entre pontos adjacentes (arestas), no ficheiro .co estão a identificação e coordenadas de cada ponto (vértices);
- Imprime na consola o caminho completo e o sua distância e coordenadas de cada ponto;

2. Maior número de ocorrências

A resolução deste exercício foi divida em três partes analise do problema, estruturas de dados utilizadas e análise de complexidade.

2.1 Análise do problema

Numa primeira análise do exercício proposto pelos docentes, como nos deparamos com um problema de descobrir qual o caminho mais curto, decidimos que o melhor algoritmo a usar era o de *Dijsktra*, pois esse algoritmo trata precisamente o tipo de problema em questão, com uma boa eficiência e um desempenho muito alto comparado com outras soluções.

Devido a escolha do algoritmo referido anteriormente, com uma implementação eficiente, na determinação do caminho entre os pontos usamos a Busca em Largura ou, em inglês, *Breadth-First Search* (BFS). O objetivo dessa pesquisa é percorrer todos os nós adjacentes (a começar no nó inicial) até achar o nó pretendido. A forma de guardar os nós que iremos percorrer implementámos uma *MinHeap* cujo a função é guardar todos os nós e manter o nó com distância mais curta no topo da *Heap*, com o intuito de tratar-mos sempre as distâncias mais curtas em primeiro lugar e assim aumentar a eficiência da aplicação pois, no primeiro instante que se chega ao ponto de término definido pelo utilizador e o próximo ponto na *Heap* for o mesmo pode-se parar de analisar o restos do nós, visto que o algoritmo garante que este é o caminho mais curto, assim evitando buscas extras desnecessárias.

Para mostrar o resultado, guardamos o caminho numa lista simplesmente ligada (Linked List), para no fim percorrer todos os nós nessa lista e mostrá-los no ecrã.

2.2 Estruturas de Dados

Como foi referido no ponto anterior, implementámos várias estruturas de dados:

- 2 ***HashMaps*** – Uma na classe *Vertex* para guardar os vértices adjacentes. Outra HashMap na classe *Graph* para se guardar todos os vértices que constituem o grafo que iremos usar para a procura do caminho.

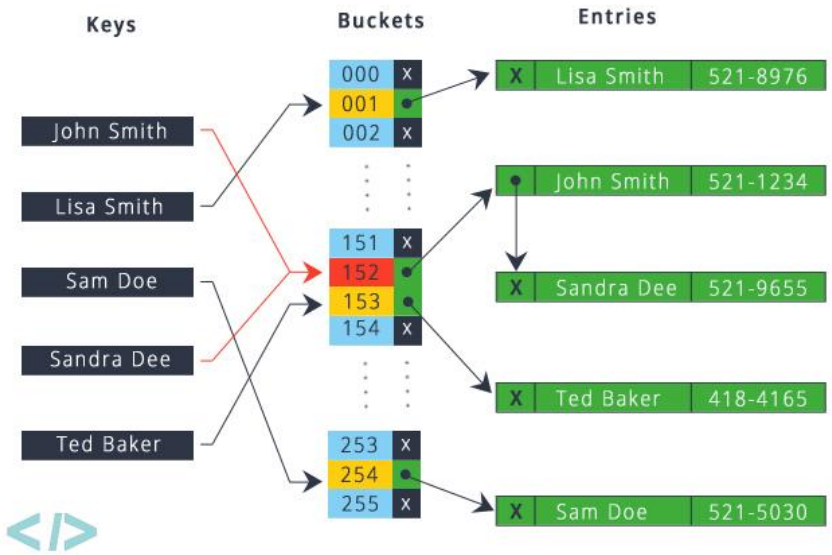

Figura 1- Imagem de *HashMap* com uso de lista simplesmente ligada para resolução de conflitos.

- ***Heap*** – Neste exercício usámos uma *Heap* criada por nós, cujo funcionamento é muito idêntico ao PriorityQueue do java com a diferença de alguns métodos extra que implementámos de modo a facilitar a, nomeadamente um método que podemos chamar para fazer o MinHeapify que facilita a atualização da *Heap* sem ser preciso inserir e remover nenhum elemento quando existe alteração da sua distância.

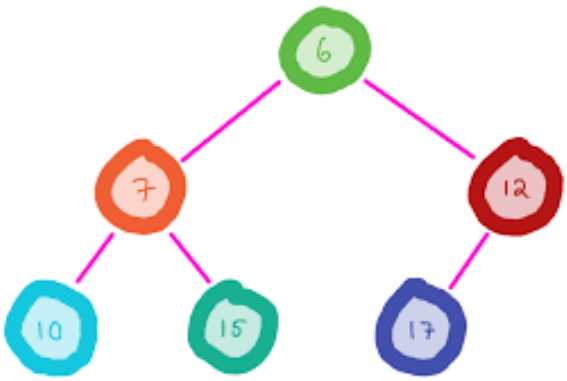

Figura 2 - Imagem de uma Heap.

- ***LinkedList*** – A *LinkedList* que usámos tem como resultado a amostragem final do caminho que obtemos desde o vértice inicial ao vértice final. Como, quando estamos a introduzir os valores na *LinkedList* começamos no vértice terminal, então, após se inserir

os vértices todos na lista, de seguida inverte-se a lista com o método *reverse* da classe Collections do java.

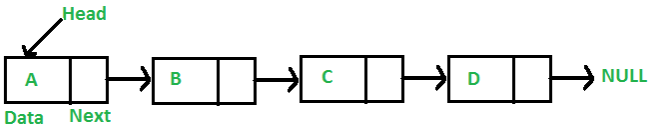

Figura 3 - Imagem de lista simplesmente ligada.

2.3 Algoritmos e análise da complexidade

Nos Algoritmos, como foi referenciado anteriormente, usámos a Busca em Largura (BFS), como esta busca poderá percorrer todos os vértices do grafo então em termos de complexidade temos:

- Tempo: O(|V|) em que V são os números de vértices que temos de percorrer, que no pior caso poderão ser todos os vértices do ficheiro.
- Espaço: O(|V|) em que V são os números de vértices que guardamos na *Heap*, no pior caso irão ser os todos os vértices do ficheiro.

A nível de complexidade de espaço da aplicação, como iremos ter de adicionar todos os vértices do ficheiro no grafo de maneira a registar todos os caminhos possíveis entre eles então a sua complexidade é O(|V|) em que são todos os vértices do ficheiro.

A nível de complexidade de tempo da aplicação, ao introduzir os vértices no grafo é O(2|V|) em que V são os vértices do ficheiro. Temos de percorrer 2 vezes pois temos que ler os 2 ficheiros, 1 para obter a identificação e coordenadas e o outro para obter os caminhos entre eles.

Para procurar o caminho, como usamos o algoritmo de *Dijkstra* então a complexidade será O (|A| + |V|log(|V|)) em que A são os vértices adjacentes e V os vértices do caminho.

3. Avaliação Experimental

A avaliação experimental foi feita com os seguintes parâmetros:

- Com o ficheiro de grafos de Califórnia e New York
- Cada par de vértices foi corrido cinco vezes para cada tipo de ficheiro com cinco vértices por ficheiro.

Tabela 1 - Tempos em milissegundos do algoritmo no ficheiro de Califórnia.

CAL	206426 -> 51017	4976 -> 254815	83105 -> 97656	Tempo de Construção de grafo
1	2137.79	840.53	331.98	11261
2	1806.78	1085.31	479.1	13408.5
3	1629.92	1322.27	315.6	11760.15
4	1721.51	767.49	313.37	11799.43
5	1638.45	811.61	319.55	12128.12

Tabela 2 - Tempos em milissegundos do algoritmo no ficheiro de New York.

NY	145528 -> 12170	114279 -> 74477	209546 -> 197854	Tempo de Construção de grafo
1	182.98	242.66	56.03	2197.42
2	126.78	236.27	45.26	2021.13
3	222.6	241.09	45.57	1989.98
4	97.74	199.18	68.38	1849.53
5	108.23	199.29	77.77	2495.17

4. Conclusões

Podemos que concluir grafos e os algoritmos para operar os grafos são uma excelente ferramenta para executar determinadas tarefas como por exemplo o cálculo da menor distancia entre dois locais.

Com o algoritmo de *Dijkstra* conseguimos calcular o caminho mais curto entre dois pontos em pouco tempo, não contando com o tempo ou até possibilidade de ler um ficheiro de vetores e conexões inteiras para memoria.