Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

[Série 3

44795 : Miguel Rebelo da Costa (A44795@alunos.isel.pt)

47654 : Pedro Afonso Assis Vaz (A47654@alunos.isel.pt)

Relatório para a Unidade Curricular de Sistemas de Informação 1 da Licenciatura em Engenharia Informática e de Computadores

Professoras : Prof.ª Maria Paula Graça

7 – Janeiro – 2022

Índice

Lista de Figuras

Introdução

A estrutura de dados e os algoritmos são dois dos aspectos mais importantes das ciências da computação. As estruturas de dados permitem-nos organizar e armazenar dados, enquanto os algoritmos nos permitem processar esses dados de maneira significativa. O problema desta série incorpora estes dois campos.

Problema: Tourist Path

2.1. Intodução ao Problema

Pretende-se desenvolver uma aplicação que permita obter o caminho de menor custo entre dois cruzamentos de uma determinada cidade, em que o custo é definido pela distância entre os mesmos.

O problema é descrito por:

- Um conjunto C de m cruzamentos identificados de 1 a m;
- Cada cruzamento tem associado as suas coordenadas geográficas no plano, isto é o seu ponto correspondente (coordenadas x e y);
- Cada rua é descrita como um triplo com o identificador dos dois cruzamentos e a distância entre esses dois cruzamentos.

O objetivo deste trabalho é portanto a realização de um programa que permita:

- dados dois identificadores de cruzamentos, obter o caminho mais curto entre ambos e retornar a listagem dos identificadores dos cruzamentos que compõem esse caminho, assim como o custo total do mesmo.

2.2. Análise ao Problema

2.2.1. Funcionalidades a implementar

Como referido anteriormente, pretende-se desenvolver uma aplicação que permita obter o caminho de menor custo entre dois cruzamentos de uma determinada cidade. Para realizar isto é preciso implementar as seguintes funcionalidades:

1. Carregamento da informação das ruas e cruzamentos, presentes num ficheiro de texto de formato .gr, dado o nome do ficheiro. Um ficheiro de formato .gr é composto por linhas de:
   1.1. comentário - estas linhas começam com o caracter c para denotar que é uma linha de comentário;
   1.2. de definição do problema - só existindo uma linha destas em cada ficheiro e tem o formato p sp n m em que n é o número cruzamentos e m ú o número de ruas;
   1.3. descrição de ruas - são linhas com o formato a u v w em que u e v são cruzamentos que definem uma rua e w é a distância entre esses dois cruzamentos. Os caracteres a negrito são os caracteres identificadores de cada linha.
2. Listagem do caminho de menor custo entre dois cruzamentos, isto é, listagem dos identificadores dos cruzamentos que compõem esse caminho, assim como o custo total do mesmo.

2.2.2. Parâmetros de Execução

Para iniciar a execução da aplicação a desenvolver, terá de se executar:

- kotlin TouristPathKt fileName.gr

Durante a sua execução, a aplicação deverá processar os seguinte comandos:

- path id1 id2 output.txt

que corresponde á sua funcionalidade.

- e

que termina a aplicação.

2.2.3. Formato dos Ficheiros

Para este projecto podem ser usados os ficheiros com redes rodoviárias presentes em

- http://www.dis.uniroma1.it/~challenge9/download.shtml

Os formatos destes ficheiros est˜ao descritos em

- http://www.dis.uniroma1.it/~challenge9/format.shtml

2.3. Ler ficheiros preenchendo um GraphStructure

Foi utilizado um GraphStructure mutável que vai armazenando os cruzamentos e as ruas, enquanto as lê. O GraphStructure que guarda o cruzamento e as suas ligações a outros cruzamentos (ruas) utiliza um HashMap<I, Vertex<I, D>>.

A lógica por trás do algoritmo de armazenamento é: Lê uma entrada, e adicionamos essa entrada composta por dois Vertex e um Edge ao Mapa utilizando as funções addEdge e addVertex dentro da função Read.

- A função Read tem complexidade temporal de O(E) , sendo E o número de edges presentes no grafo e também o número de linhas do ficheiro de input. Visto que addEdge e addVertex tem custo O(1) , a função Read fica-se por O(E).
- A complexidade temporal da função addEdge é O(1), uma vez que anexa um elemento ao fim de uma lista ligada (LinkedList) utilizando a função add, que tem uma complexidade constante em tempo.
- A complexidade temporal da função addVertex é O(1), uma vez que utiliza uma tabela de hash (HashMap) para armazenar os vértices e tem uma complexidade temporal constante para a inserção de um elemento.

```
fun read(file : String,g : GraphStructure<Int, Int>){
    File(DIR + file).forEachLine {
        if(it[0]=='p') {
            val arguments = it.split(" ").toList()
            noNodes = arguments[2].toInt()
            noArcs = arguments[3].toInt()
        }
        if(it[0]=='a'){
            val arguments = it.split(" ").toList()
            g.addVertex(arguments[1].toInt(),arguments[1].toInt())
            g.addVertex(arguments[2].toInt(),arguments[2].toInt())
g.addEdge(arguments[1].toInt(),arguments[2].toInt(),arguments[3].toInt())
        }
}}
```

Figura 2.1: Função de leitura e armazenamento de dados

2.4. Find the Shortest Path

```
fun findShortestPath( idS: Int, idT: Int,graph: GraphStructure<Int, Int>):
Pair<Int,List<Int>> {
    val pair = dijkstra(idS , idT ,graph)
    val dist = pair.first
    val prev = pair.second
    val path = LinkedList<Int>()
    if(dist[idT]==Int.MAX_VALUE) return Pair(0,path)
    var at = idT
    while(at != idS){
        path.add(at)
        if(prev[at] == null) return Pair(0,emptyList())
        at = prev[at]!!
    }
    path.add(idS)

    return Pair(dist[idT],path.reversed())
}
```

Figura 2.2: Processo de procura do caminho mais curto

- A complexidade temporal da função findShortestPath é O(V), onde V é o número de vértices no gráfico. Isto porque a função chama a função dijkstra, que tem uma complexidade temporal de O((V + E) log V), e depois realiza uma operação de tempo linear (iteração sobre a matriz prev) para construir o caminho mais curto desde o nó de origem até ao nó alvo.

2.4.1. Algoritmo Dijkstra

O Algoritmo de Dijkstra começa num nó escolhido (o nó de origem) e analisa o grafo para encontrar o caminho mais curto entre esse nó e todos os outros nós do grafo.
O algoritmo rastreia a distância mais curta atualmente conhecida de cada nó até o nó de origem e atualiza esses valores se encontrar um caminho mais curto.
Depois que o algoritmo encontra o caminho mais curto entre o nó de origem e outro nó, esse nó é marcado como "visitado" e adicionado ao caminho.
O processo continua até que todos os nós no grafo tenham sido adicionados ao caminho. Desta forma, temos um caminho que conecta o nó fonte a todos os outros nós seguindo o caminho mais curto possível para chegar a cada nó.
O Algoritmo de Dijkstra só pode trabalhar com grafos que tenham pesos positivos. Isso

porque, durante o processo, os pesos das arestas precisam ser somados para encontrar o caminho mais curto.

Se houver um peso negativo no gráfico, o algoritmo não funcionará corretamente. Depois que um nó é marcado como "visitado", o caminho atual para esse nó é marcado como o caminho mais curto para chegar a esse nó. E pesos negativos podem alterar isso se o peso total puder ser diminuído após a ocorrência dessa etapa.

```
fun  dijkstra(idS : Int , idT : Int , graph: GraphStructure<Int, Int>):
Pair<IntArray,Array<Int?>> {
   val visited = BooleanArray(graph.vertices.size+10)
    val dist = IntArray(graph.vertices.size+10){ Int.MAX_VALUE}
    val prev = arrayOfNulls<Int>(graph.vertices.size+10)
    dist[idS]=0
    val q =  PriorityQueue {x: Pair<Int, Int>, y: Pair<Int, Int> ->
x.second.compareTo(y.second)}
    q.add(Pair(idS,0))
    while(!q.isEmpty()){
        val pair = q.poll()
        val idx = pair.first
        val minValue = pair.second
        visited[idx]=true
        if(dist[idx] < minValue) continue
        val edges = graph.vertices[idx]!!.edges
        for(adj in edges){
            if(visited[adj.adjacent]) continue
            val newDist = dist[idx] + adj.weight
            if(newDist < dist[adj.adjacent]){
                prev[adj.adjacent]= idx
                dist[adj.adjacent] = newDist
                q.add(Pair(adj.adjacent, newDist))
            }
            if(idx == idT) return Pair(dist , prev)
        }
    }
    return Pair(dist,prev)
}
```

Figura 2.3: Função seguindo o algoritmo Dijkstra de pesquisa

- A complexidade temporal desta função é O((V + E) log V), onde V é o número de vértices no gráfico e E é o número de arestas. Isto porque a função utiliza uma fila prioritária para armazenar e classificar as distâncias dos nós não visitados, e a complexidade temporal de inserir um elemento numa fila prioritária e remover o elemento mínimo é O((log V). A função também realiza uma operação de tempo constante (verificação se o q está vazio) e uma operação de tempo linear (iteração sobre as bordas de cada nó) para cada vértice no gráfico, levando a uma

complexidade de tempo adicional de O(V + E).

- A complexidade espacial desta função é O(V), uma vez que armazena a matriz ´visited´, a matriz 'dist´, a matriz 'prev', e a fila de prioridade q, todas elas com um tamanho proporcional ao número de vértices no gráfico.

2.5. Output Result

```
fun writeOutput(output: String, idStart : Int , idEnd : Int , cost : Int ,
path : List<Int>){
    val fileName = DIR + output
    val writer = createWriter(fileName)
    if(path.isEmpty()){
        writer.println("Não há caminho entre $idStart e $idEnd ")
    }
    else {
        writer.println("O caminho mais curto entre $idStart e $idEnd tem
custo $cost e é:")
        for (v in path) {
            writer.println(v)
        }
    }
    writer.close()
}
fun createWriter(fileName: String) = PrintWriter(fileName)
```

Figura 2.4: Função de escrita em ficheiro

- A função writeOutput tem uma complexidade O(V) visto que itera pelo path dado por findShortestPath.

Conclusão

Este trabalho permitiu-nos aprofundar o nosso conhecimento sobre o algoritmo Dijkstra de pesquisa (greedy) e aprofundar os nosso conhecimnetos na utilização da estrutura de dados HashMap , sendo esta a escolhida para desenvolver o trabalho por ser a que se adequa melhor, tendo em conta o objetivo do problema.

Referências

https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/

"Disciplina: Algoritmos e Estruturas de Dados - 2223SV," Moodle 2022/23. https://2223moodle.isel.pt.