Departamento de Informática
Estruturas de Dados e Algoritmos II
Ano Letivo de 2020/2021

Mazy Luck

Trabalho por:
Miguel Rodrigues - 45424
André Rato - 45517

Docente:
Vasco Pedro
Mooshak: g304

Índice

1. Introdução

Este relatório é referente ao problema "Mazy Luck".

O problema consiste em verificar se é possível perder dinheiro ao atravessar-se um labirinto. Dirk, o jogador, inicialmente não possui moedas, mas possui um cartão ilimitado com zero de crédito. O objetivo de Dirk é atravessar o labirinto desde a entrada até à saída, passando por várias salas e corredores. Cada corredor pode dar ao Dirk moedas, se tiver um saco com moedas, ou retirar crédito, se for um fosso com crocodilos, onde a única maneira de o atravessar é pagar para fazer descer uma ponte levadiça.

2. Análise do Problema

Para a realização deste trabalho foi utilizado um grafo orientado pesado, implementado através de um array de vértices. Cada sala do labirinto é representada por um vértice do grafo, enquanto que cada corredor é representado por uma aresta com o peso positivo, caso este seja um saco de moedas, ou negativo, caso este seja uma ponte levadiça.

O grafo (Figura 1) que representa o problema é construído a partir de um *input*. Após a construção do mesmo, é necessário determinar se Dirk pode perder dinheiro em todos os percursos possíveis do labirinto, ou seja, determinar o caminho mais curto (menos pesado) entre o vértice de entrada e o de saída (Figura 2), e verificar se este caminho tem peso total negativo ou positivo.

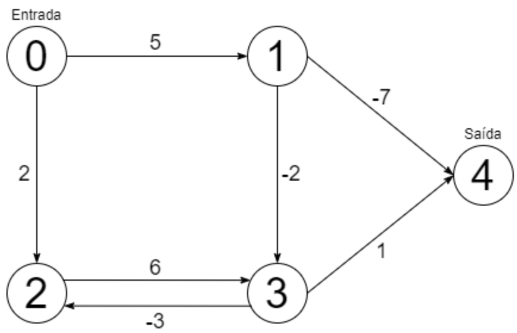

Figura 2 - Caminho mais curto de um grafo

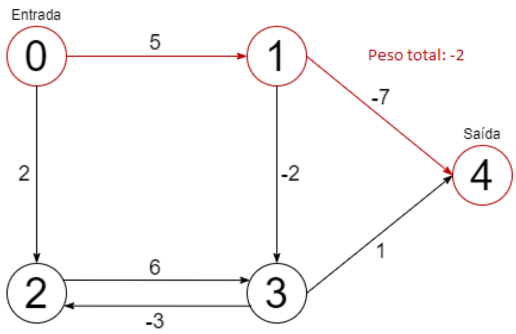

Figura 1 - Exemplo de um grafo representativo do problema

3. Estruturas Utilizadas

Foi implementada uma classe designada por *Vertex*, responsável por representar os vértices do grafo. A classe possuí quatro variáveis de classe:

- int number: número da sala;
- int distance: inteiro utilizado no algoritmo de *Bellman-Ford*;
- ArrayList<Edge> edges: lista com todos os vértices adjacentes ao vértice;
- Vertex predecessor: vértice utilizado no algoritmo de *Bellman-Ford*.

Foi implementada uma classe designada por *Edge*, responsável por representar as arestas do grafo. A classe possuí duas variáveis de classe:

- int weight: peso da aresta;
- int destination: número do vértice de destino da aresta.

Para além das classes *Vertex* e *Edge*, também se implementou a classe *Graph*, representativa de um grafo. A classe tem as seguintes variáveis de classe e métodos:

- Vertex[] V: *array* com todos os vértices que constituem o grafo;
- int numberOfVertexes: número de vértices do grafo;
- void addAdj(int origin, Edge edge): método que adiciona uma aresta a lista de adjacências do vértice de número *origin*.

4. Algoritmo

Com o objetivo de determinar se é possível ou não perder dinheiro, foi utilizado o algoritmo de *Bellman-Ford*, cujo pseudocódigo foi fornecido durante as aulas da disciplina.

O algoritmo começa por chamar a função *initializeSingleSource* para inicializar as variáveis *distance* e *predecessor* de todos os vértices, fazendo distinção do vértice passado como argumento.

Após esta inicialização, todas as arestas são "relaxadas" (verificar se é possível encontrar um caminho mais curto do que o caminho já encontrado), utilizando o método *relax*, |V| - 1 vezes. Isto acontece, pois, o caminho mais curto do vértice fonte para qualquer outro vértice pode ter no máximo |V| - 1 arestas.

Por fim, percorrem-se novamente as arestas, com o objetivo de identificar ciclos com peso total negativo. Caso seja encontrado um "ciclo negativo", o algoritmo retorna *false*; caso contrário, é retornado *true*.

Depois de aplicado o algoritmo de *Bellman-Ford*, o valor da variável *distance* do vértice de número |V| - 1 (ou seja, a saída) é avaliado:

- se o valor for negativo ou o algoritmo tenha retornado *false*, é enviado para o *output* a *string* "**yes**", ou seja, é possível perder dinheiro;
- caso contrário, envia "**no**", ou seja, não é possível perder dinheiro.

5. Código

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList.

public class MazyLuck {
    public static class Graph {
        int numberOfVertexes;
        Vertex[] V;

         public static class Vertex {
            int number, distance;
            ArrayList<Edge> edges;
            Vertex predecessor;

            public Vertex(int number) {
                this.number = number;
                this.edges = new ArrayList<>();
            }
        }

        public static class Edge {
            int weight;
            int destination;

            public Edge(int weight, int destination) {
                this.weight = weight;
                this.destination = destination;
            }
        }

        public Graph(int numberOfVertexes) {
            this.V = new Vertex[numberOfVertexes];
            this.numberOfVertexes = numberOfVertexes;

            for (int i = 0; i < numberOfVertexes; i++) {
                this.V[i] = new Vertex(i);
            }
        }

        void addAdj(int origin, Edge edge) {
            this.V[origin].edges.add(edge);
        }

        private static void initializeSingleSource(Graph G, Vertex s) {
            for (Vertex v : G.V) {
                v.distance = Integer.MAX_VALUE;
                v.predecessor = null;
            }
            s.distance = 0;
        }

        private static void relax(Vertex u, Vertex v, int w) {
            if (u.distance != Integer.MAX_VALUE && u.distance + w < v.distance) {
                v.distance = u.distance + w;
                v.predecessor = u;
            }
        }

        public boolean bellmanFord(Graph G, Vertex s) {
            initializeSingleSource(G, s);
            for (int i = 1; i <= G.V.length - 1; i++)
                for (Vertex u : G.V)
                    for (Edge e : u.edges)
                        relax(u, G.V[e.destination], e.weight);
            for (Vertex u : G.V)
                for (Edge e : u.edges)
```

```
57                      if (u.distance + e.weight < G.V[e.destination].distance)
58                          return false;
59              return true;
60          }
61      }

62      public static void main(String[] args) throws IOException {
63          BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
64          int weight, R, C, origin, destination;
65          boolean a;
66          char c;
67          String[] line;
68          Graph G;

69          line = input.readLine().split(" ");

70          R = Integer.parseInt(line[0]);
71          C = Integer.parseInt(line[1]);

72          G = new Graph(R);

73          for (int i = 0; i < C; i++) {
74              line = input.readLine().split(" ");

75              origin = Integer.parseInt(line[0]);
76              destination = Integer.parseInt(line[1]);
77              c = line[2].charAt(0);
78              weight = Integer.parseInt(line[3]);

79               if (c == 'C')
80                  weight *= -1;

81              G.addAdj(origin, new Graph.Edge(weight, destination));
82          }

83          a = G.bellmanFord(G, G.V[0]);

84          if (G.V[R - 1].distance < 0 || !a)
85              System.out.println("yes");
86          else
87              System.out.println("no");
88      }
89  }
```

6. Complexidade Temporal

No método *initializeSingleSource* (linhas 36-42), o ciclo executado |V| vezes, sendo que |V| corresponde ao número de vértices, e tem apenas operações de complexidade temporal $O(1)$; logo a complexidade temporal do método é:

$$|V| \cdot O(1) = \theta(|V|)$$

No método *relax* (linhas 43-48), todas as operações (comparações, adições e afetações) têm complexidade temporal constante, logo a complexidade temporal do método é:

$$O(1) + O(1) + O(1) + O(1) = \theta(1)$$

O método *bellmanFord* (linhas 49-61) pode ser dividido em três partes:

- na linha 50, onde é chamada a função *initializeSingleSource*, a complexidade temporal é $\theta(|V|)$;
- da linha 51 à linha 54 estão presentes três ciclos:
  - o ciclo exterior é executado |V| - 1 vezes;
  - o ciclo intermédio é executado |V| vezes;
  - o ciclo interior é executado |E| vezes, sendo |E| o número de arestas.

Deste modo, e sabendo que a operação executada no ciclo interior (*relax*) tem complexidade temporal $\theta(1)$, é possível concluir que a complexidade temporal deste bloco é:

$$\theta(1) \,.\, (|V| \,.\, (|V| + |E|)) = \theta(|V|^2)$$

- *da linha 55 à linha 58 estão presentes dois ciclos:*
    - o ciclo exterior é executado |V| vezes;
    - o ciclo interior é executado |E| vezes.

Assim, e sabendo que as operações de comparação e de retorno têm ambas complexidade temporal $O(1)$, a complexidade temporal deste bloco é:

$$(O(1) + O(1)) \,.\, (|V| + |E|) = \theta(|V| + |E|)$$

Deste modo conclui-se que a complexidade temporal do algoritmo utilizado é:

$$\theta(|V|) + \theta(1) + \theta(|V|^2) + \theta(|V| + |E|) = \theta(|V|^2)$$

Relativamente à classe do grafo (*Graph*) podemos concluir que, no construtor da classe *Vertex*, o ciclo que cria os vários vértices do grafo, dado o tamanho, tem complexidade temporal $O(|V|)$. As restantes operações têm todas complexidade temporal $O(1)$.

Na *main* do programa, todas as operações têm complexidade temporal $O(1)$, exceto o ciclo *for*, que tem complexidade $O(C)$, sendo C o número de corredores/arestas (mesmo significado de |E|).

Conclui-se assim que o programa tem complexidade temporal $\theta(|V|^2)$.

7. Complexidade Espacial

No grafo, cada vértice tem, uma variável de classe do tipo *Vertex*, dois valores escalares, *number* e *distance*, e tem uma lista de tamanho |E|, no pior dos casos (se o vértice for adjacente a todos os outros) logo, a complexidade espacial do *array* de vértices é $\theta(|V| + |E|)$.

Como não são utilizadas mais nenhumas estruturas para armazenamento de dados, pode concluir-se que a complexidade espacial do programa é $\theta(|V| + |E|)$.

8. Bibliografia

https://www.bigocheatsheet.com/ - consulta de complexidades tabeladas;
Slides 1-10 e 136-141 das aulas teóricas de EDA2, fornecidos pelo docente.