Relatório TP2

Tiago Fernandes 57677 | António Ferreira 58330 | ADA

Explicação:

Para resolver o desafio, decidimos usar o algoritmo de Bellman-Ford, com o objetivo de determinar o caminho do challenge inicial para o final (wizard), calculando os melhores pesos de todos os nós a partir do vértice inicial.

O algoritmo de Bellman-ford é feito de forma semelhante ao lecionado nas aulas teóricas, embora tenha código adicional para o casos em que um nó consiga chegar ao wizard a partir de um caminho.

Foi criado um set canReachFinal (de modo a não existir nós repetidos) que guarda os nós que podem chegar ao nó final. O set tem inicialmente o nó final.

Sempre que o updateLengths é invocado, verifica-se se o secondNode da edge corrente existe no set e, neste caso, o firstNode é adicionado ao canReachFinal, pois se o secondNode chega ao nó final e se o firstNode se encontra numa edge direcionada para o secondNode, então o firstNode também consegue chegar ao nó final.

No final do algoritmo Bellman-Ford, aquando da verificação de ciclos de peso negativo, caso o peso de algum nó tenha mudado e este consiga chegar ao wizard, é lançada a exceção NegativeWeigthCycleException, que devolve 'Full of energy'.

O algoritmo devolve o peso final do nó correspondente ao wizard. Caso este valor seja menor ou igual a zero, é lançada a exceção NegativeWeigthCycleException. Caso contrário, a resposta é o máximo entre zero e a subtração entre a energia inicial (intialEnergy) e energia consumida durante o caminho (resultado do algoritmo).

Complexidade temporal

AwesomeWarriorGame.AwesomeWarriorGame:

No construtotor a complexidade temporal é O(1), pois faz-se apenas inicialização de variáveis, todas com complexidade **O(1)**.

AwesomeWarriorGame.handleConnection:

É feita uma inicialização de uma variável e uma atribuição a uma variável, ambas com complexidade O(1), logo o método tem complexidade **O(1).**

AwesomeWarriorGame.processFinalLine:

São feitas 3 inicializações de variáveis, todas com complexidade O(1), logo o método tem complexidade **O(1).**

AwesomeWarriorGame.updateLengths:

Como optámos por utilizar uma lista de arcos para representar o grafo, o ciclo terá uma complexidade temporal de **O(|decisions|),** sendo decisions o equivalente ao número de edges.

AwesomeWarriorGame.bellmanFord:

É feito o algoritmo de Bellman-Ford.

O ciclo que preenche o vetor length com long.MAX_VALUE tem complexidade O(|numNodes|), pois o ciclo for é feito numNodes vezes.

O ciclo que invoca o método updateLengths tem complexidade O(|numNodes| * |decisions|), pois o ciclo corre numNodes vezes e é invocado um método com complexidade temporal O(|decisions|), sendo que decisions é o equivalente ao número de edges.

O ciclo que verifica se existem ciclos de peso negativo pode correr até numNodes vezes, logo tem complexidade temporal de O(|numNodes|).

Como todas as outras operação têm complexidade O(1), a complexidade temporal será O(|numNodes|) + O(|numNodes| * |decisions|) + O(|numNodes|) = **O(|numNodes| * |decisions|).**

Função	Caso Médio	Melhor Caso	Pior Caso
AwesomeWarriorGame	O (1)	O (1)	O (1)
handleConnection	O (1)	O (1)	O (1)
processFinalLine	O (1)	O (1)	O (1)
updateLengths	O(|decisions|)	O(|decisions|)	O(|decisions|)
bellmanFord	O (|numNodes|* |decisions|)	O (|numNodes|* |decisions|)	O (|numNodes|* |decisions|)

Complexidade espacial

AwesomeWarriorGame.AwesomeWarriorGame:

É inicializado o vetor de edges com tamanho **decisions** e como todas as outras variáveis têm sempre o mesmo tamanho e consequentemente complexidade de O(1), a complexidade espacial total do construtor é de **O(|decisions|).**

AwesomeWarriorGame.handleConnection:

A complexidade espacial é de **O(|decisions|),** pois o tamanho do vetor edges depende do número de decisions.

AwesomeWarriorGame.processFinalLine:

Todas as variáveis do método têm tamanho constante, logo a sua complexidade é de O(1).

AwesomeWarriorGame.updateLengths:

O método recebe como argumentos 3 variáveis de tamanho variável. O tamanho do vetor edges depende do número de decisions, logo tem O(|decisions|). O tamanho do vetor len depende do numNodes, logo tem O(|numNodes|). O tamanho do set canReachFinal pode ser no máximo numNodes, logo a complexidade é de O(|numNodes|).

Portanto, como o resto das complexidades são de O(1), a complexidade espacial total do método é:

O(|decisions|) + 2 * O(|numNodes|) = max(O(|numNodes|), O(|decisions|)) = **O(|numNodes|).**

AwesomeWarriorGame.bellmanFord:

As complexidades deste métodos são as mesmas do método anterior pois tem as mesmas variáveis com tamanho variável: edges, length e canReachFinal.

Existe um vetor prevLengths que é uma cópia de lengths. Logo, como têm o mesmo tamanho, O(2*|numNodes|) = O(|numNodes|), logo a complexidade espacial final será é a mesma: **O(|numNodes|).**

Função	Caso Médio	Melhor Caso	Pior Caso
AwesomeWarriorGame	O(|decisions|)	O(|decisions|)	O(|decisions|)
handleConnection	O(|decisions|)	O(|decisions|)	O(|decisions|)
processFinalLine	O (1)	O (1)	O (1)
updateLengths	O(|numNodes|)	O(|numNodes|)	O(|numNodes|)
bellmanFord	O(|numNodes|)	O(|numNodes|)	O(|numNodes|)

Conclusão

Para representar o grafo do problema começámos por utilizar uma lista de adjacências. No entanto, percebemos que desta forma a complexidade seria desnecessariamente maior e por esse motivo, alterámos o programa de modo a utilizar uma lista de arcos.

Depois de discutirmos ideias com docentes, foi-nos sugerida a hipótese de uma solução em que não haveria qualquer alteração ao algoritmo de Bellman-Ford. No entanto, depois de pensarmos em formas de resolver o problema deste modo, não chegámos a uma resposta. Optámos então por uma solução com recurso a uma pequena adição ao algoritmo, não alterando a complexidade do mesmo, usando um set que guarda os nós que chegam ao nó final.

Anexo

Main:

```
public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new
InputStreamReader((System.in)));

        String[] aux = in.readLine().split(" ");
        int challenges = Integer.parseInt(aux[0]);
        int decisions = Integer.parseInt(aux[1]);

        AwesomeWarriorGame game = new AwesomeWarriorGame(challenges,
decisions);

        for (int i = 0; i < decisions; i++) {
            aux = in.readLine().split(" ");
            game.handleConnection(Integer.parseInt(aux[0]), aux[1],
Integer.parseInt(aux[2]), Integer.parseInt(aux[3]));
        }

        aux = in.readLine().split(" ");
        game.processFinalLine(Integer.parseInt(aux[0]),
Integer.parseInt(aux[1]), Integer.parseInt(aux[2]));
        try {
            System.out.println(game.solve());
        } catch (NegativeWeightCycleException e) {
            System.out.println(e.getMessage());
        }
    }


}
```

AwesomeWarriorGame:

```
public class AwesomeWarriorGame {

    private final String PAYS = "Pays";

    private final Edge[] edges;
    private final int numNodes;
    private int numEdges;

    private int initialChallenge;
    private int finalChallenge;
    private int initialEnergy;

    public AwesomeWarriorGame(int challenges, int decisions) {
        this.numNodes = challenges;
        this.edges = new Edge[decisions];
        this.numEdges = 0;
    }

    public void handleConnection(int finishedChallenge, String
action, int energy, int newChallenge) {
        int weight = action.equals(PAYS) ? energy : -energy;
        edges[numEdges++] = new Edge(finishedChallenge, weight,
newChallenge);
    }

    public void processFinalLine(int initialChallenge, int
finalChallenge, int initialEnergy) {
        this.finalChallenge = finalChallenge;
        this.initialChallenge = initialChallenge;
        this.initialEnergy = initialEnergy;
    }

    private long bellmanFord(Edge[] edges, int origin) throws
NegativeWeightCycleException {
        long[] length = new long[this.numNodes];
        for (int node = 0; node < this.numNodes; node++)
            length[node] = Long.MAX_VALUE;
        length[origin] = 0;

        boolean changes = false;

        Set<Integer> canReachFinal = new HashSet<>();
        canReachFinal.add(finalChallenge);

        for (int i = 1; i < this.numNodes; i++) {
            changes = updateLengths(edges, length, canReachFinal);
            if (!changes)
                break;
        }

        long[] prevLength = length.clone();
        if (changes && updateLengths(edges, length, canReachFinal))
            for (int i = 0; i < this.numNodes; i++)
                if (prevLength[i] != length[i] &&
canReachFinal.contains(i))
                    throw new NegativeWeightCycleException();

        return length[finalChallenge];
```

```
    }

    private boolean updateLengths(Edge[] edges, long[] len,
Set<Integer> canReachFinal) {
        boolean changes = false;
        for (Edge e : edges) {
            if (canReachFinal.contains(e.secondNode))
                canReachFinal.add(e.firstNode);
            if (len[e.firstNode] < Integer.MAX_VALUE) {
                long newLen = len[e.firstNode] + e.weight;
                if (newLen < len[e.secondNode]) {
                    len[e.secondNode] = newLen;
                    changes = true;
                }
            }
        }
        return changes;
    }

    public long solve() throws NegativeWeightCycleException {
        long energyConsumed = this.bellmanFord(this.edges,
this.initialChallenge);
        if (energyConsumed <= 0)
            throw new NegativeWeightCycleException();
        return Math.max(initialEnergy - energyConsumed, 0);
    }

    private static class Edge {
        private final int firstNode;
        private final int secondNode;
        private final int weight;

        public Edge(int firstNode, int weight, int secondNode) {
            this.firstNode = firstNode;
            this.weight = weight;
            this.secondNode = secondNode;
        }
    }
}
```