



UNIVERSIDADE  
DE ÉVORA

# **Relatório - 3º Trabalho - Estruturas de Dados e Algoritmos II**

**Trabalho realizado pelo grupo g326 (Mooshak) composto por:**

- Rui Roque nº42720
- Tomás Dias nº42784

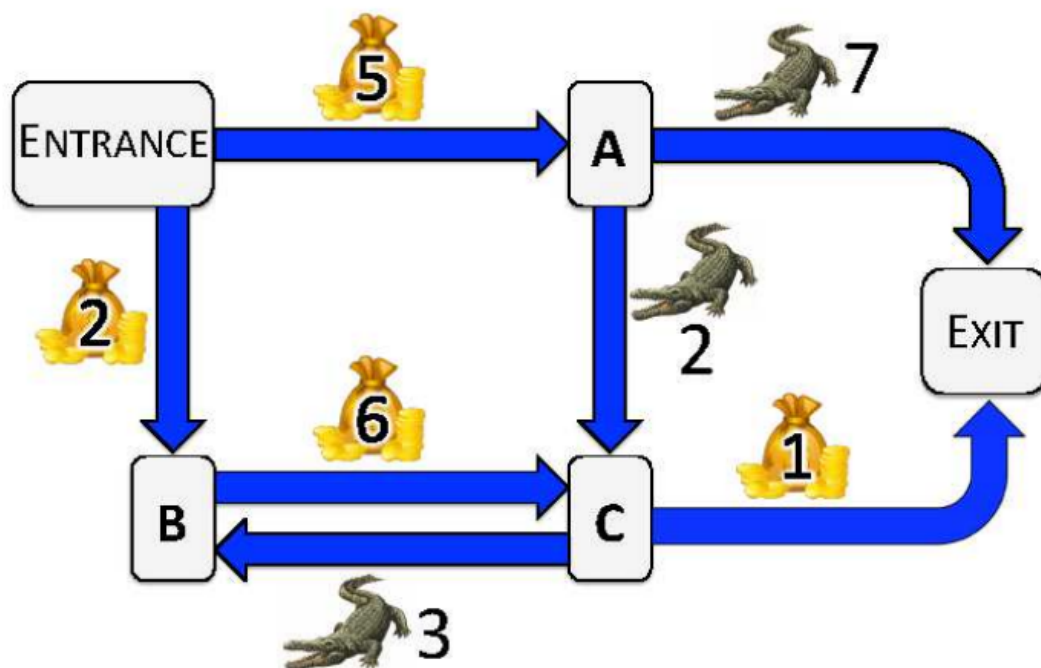
## Introdução ao problema

Este trabalho teve como objetivo implementar uma solução para o problema *Mazy Luck*.

Para tal, foi pedido um programa que dado o número de salas, o número de corredores e a informação relativa a cada corredor (salas que este liga e o seu peso que, no contexto do problema, corresponde ao número de moedas ganhas ou perdidas), indicar se seria possível perder moedas ao atravessar o maze descrito ou não.

## Construção e descrição do algoritmo

De forma a elaborar o algoritmo implementado, focou-se no grafo de exemplo apresentado no enunciado:



Recebendo a informação relativa ao grafo (vértices e arcos), o algoritmo começa por o construir recorrendo à classe **Maze** implementada. Esta classe recebe o número de vértices (salas) e de arcos (corredores), sendo que cada arco é um objeto da classe **Corridor** que tem como atributos o vértice de origem do arco (número da sala em que o corredor se inicia), o vértice de destino (número da sala em que o corredor acaba) e o peso (número de moedas ganhas, se for positivo, ou perdidas, se for negativo).

Para a resolução do problema principal, optou-se por utilizar o algoritmo de **Bellman-Ford**. De entre todos os algoritmos que calculam os caminhos mais curtos num grafo (dos que foram lecionados), este era o mais indicado dado o perfil do problema por ser o único que suporta **grafos com pesos negativos** (ao contrário do Dijkstra) e que tem como característica ser **single-source** (ao contrário do Floyd-Warshall).

De forma geral, o algoritmo **Bellman-Ford** sobrestima o comprimento do caminho desde o vértice inicial até todos os outros vértices. Depois “relaxa” iterativamente essas estimativas ao encontrar novos caminhos mais curtos do que os caminhos anteriormente sobrestimados. Ao fazer isso repetidamente para todos os vértices, o resultado é otimizado.

Mais detalhadamente em relação à implementação, começa-se por inicializar as distâncias do **source** (entrada do *maze*) a todos os vértices do grafo (restantes salas). Estas distâncias são guardadas no array **distances** em que o seu tamanho é o número de vértices do grafo (número de salas no *maze*).

De seguida, é feito o *relax* de todos os arcos (caminhos do *maze*), ou seja, são calculados os caminhos mais curtos para cada vértice. Este procedimento é feito de modo cíclico, em que na primeira iteração são calculados os caminhos mais curtos para os vértices até à distância de **um arco**, sendo atualizado o array **distances** com essas distâncias, em que na segunda iteração são calculados os caminhos mais curtos para vértices até à distância de **dois arcos**, sendo **distances** novamente atualizado, e assim sucessivamente durante  **$V - 1$**  vezes (número de salas - 1 no contexto do problema).

No final de serem calculados os caminhos mais curtos, o grafo é novamente percorrido de forma a verificar se existem ciclos de pesos negativos. Se for obtido um caminho mais curto para qualquer vértice em relação aos calculados anteriormente, existe um ciclo de pesos negativos. No contexto do problema, é concluído que são perdidas moedas.

Se não existirem ciclos de pesos negativos, é verificado se a distância do vértice correspondente à entrada do *maze* ao vértice correspondente à saída do *maze* é maior ou igual a 0. Se a condição for verdadeira, não são perdidas moedas e se for falsa, são perdidas moedas.

## Análise das complexidades temporal e espacial

Por se tratar de uma adaptação direta do algoritmo de Bellman-Ford lecionado nas aulas teóricas, as complexidades temporal e espacial serão idênticas às calculadas nas mesmas. Sendo **R** o número de salas do *maze* e **C** o número de corredores do *maze* as complexidades encontram-se calculadas abaixo.

Inicialmente, utilizando as classes **Maze** e **Corridor** para criar o grafo pesado, é utilizado o array **corridor** para guardar a informação relativa aos arcos do grafo, pelo que a sua **complexidade espacial** é de:  **$O(C)$** .

De forma a guardar as distâncias calculadas, é utilizado o array **distances** sendo que a sua **complexidade espacial** é de:  **$O(R)$** .


Então, o total da **complexidade espacial** é:  **$O(C) + O(R) = O(C+R)$** .

A **complexidade temporal** do código seguinte correspondente ao algoritmo Bellman-Ford é:

```
// initialize
for (i = 0; i < R; ++i)
    distances[i] = Integer.MAX_VALUE;
distances[0] = 0;

// relax
for (i = 1; i < R; i++)
{
    for (j = 0; j < C; j++)
    {
        r = maze.corridor[j].current_room;
        c = maze.corridor[j].next_room;
        g = maze.corridor[j].gold_coins;
        if (distances[r] != Integer.MAX_VALUE && distances[r] + g < distances[c])
            distances[c] = distances[r] + g;
    }
}

// check for negative-weight cycles
for (j = 0; j < C; j++)
{
    r = maze.corridor[j].current_room;
    c = maze.corridor[j].next_room;
    g = maze.corridor[j].gold_coins;
    if (distances[r] != Integer.MAX_VALUE && distances[r] + g < distances[c])
        return false;
}
```



Ou seja,  **$O(R) + O(RC) + O(C)$** .

Acrescentando a **complexidade temporal** da leitura do ficheiro,  $O(C)$ , a **complexidade temporal** total é:  $O(R) + O(RC) + O(C) + O(C) = O(RC)$ .