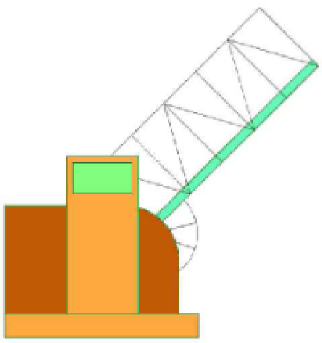
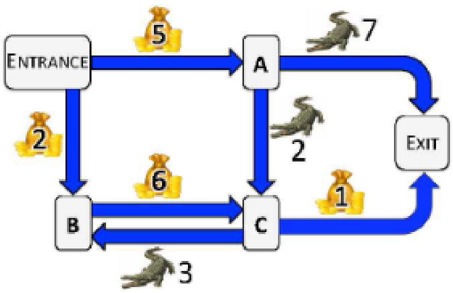
Relatório do 3º Trabalho Prático

Mazy Luck

EDA 2





Realizado por:
Pedro Grilo (43012)
Diogo Castanho (42496)

Ano Letivo 2020/2021

1 Introdução

O Dirk vai enfrentar um novo desafio. O desafio consiste em andar num labirinto, com vários corredores ligados entre si, que fazem a conexão entre vários quartos existentes.

Na entrada, o Dirk recebe um cartão com um total de 0 créditos. Em cada corredor existe ou uma bolsa com moedas, ou uma ponte sobre um lago com crocodilos. Se o Dirk quiser passar esse corredor de crocodilos, terá de pagar um conjunto de moedas pedido para que a ponte seja baixa.

Se ele não tiver moedas suficientes, terá que usar o seu cartão de crédito, podendo ficar com saldo negativo.

2 Objetivos

Foi-nos proposta a realização de um trabalho que consistia na implementação de um programa que tem como objetivo receber um input onde nos é disponibilizado o número de quartos e o número de corredores do labirinto que o Dirk vai percorrer.

Também devem ter os corredores e pesos respetivos existentes (exemplo: Quarto1 para Quarto2, com 5 moedas, ou Quarto2 para Quarto4, com ponte de custo 3 moedas).

Com estes dados, devemos conseguir dizer se o Dirk ao chegar ao quarto final, perdeu ou não dinheiro.

3 Descrição do Algoritmo

Após algumas interpretações falhadas na abordagem à resolução do problema, percebemos que o algoritmo mais adequado seria o algoritmo de **Bellman Ford**, uma vez que é um bom algoritmo para cálculo do caminho mais curto de um nó **Source** para os restantes nós do grafo.

Por sabermos que teríamos de pensar no problema como um grafo, criámos 2 classes: uma para os **Vértices** e outra para os **Arcos**.

Os vértices são constituídos por uma **Label**, que reprensenta o número dos mesmos, por um Vertice **Predecessor**, que irá conter o Vértice anterior a ele, e por uma **Distância**, que reprensenta a distância do caminho mais curto desde o vértice inicial até ele mesmo.

Os arcos são constituídos por dois vértices: um vértice **Source** e um vértice **Destination**. Para além disso tem um **peso**, que representa o custo de ir do vértice source para o vértice destination.

Na leitura do input fazemos logo a criação de todos os vértices (desde 0 até o valor do número de quartos - 1), colocando todos os vértices num array.

De seguida, mediante os seguintes inputs (dos arcos corredores, e do seu peso), críamos os arcos correspondentes a esses valores, colocando-os também num outro array.

Como no input temos duas letras: **B** para caminho com moedas, e **C** para caminho com ponte, tivemos que alterar o valor do peso para negativo quando fosse a letra C, facilitando a criação dos arcos com o peso correto.

De seguida chamamos a função **Bellman Ford**, onde o algoritmo vai ser efetuado, com os vértices e arcos gerados anteriormente.

Inicializamos um array auxiliar, que irá conter as distâncias de cada vértice até ao vértice inicial. Para esse array damos logo à posição inicial o valor 0 (pois o vértice inicial tem distância 0).

CONTINUAÇÃO DA DESCRIÇÃO DO ALGORITMO

Como feito na aula, tivemos que inicializar todos os valores dos vértices numa função **Initializa-Single-Source**, dando ao vértice source a distância 0. Aos outros vértices damos o valor de **+infinito**, dando ao atributo predecessor o valor **null**.

Após a atribuição correta dos valores aos vértices, passamos para a parte de **Relaxamento**, que podia ser numa função Relax à parte, mas decidimos fazê-la no código em si.

Nesta parte, vamos ter dois ciclos: o **exterior**, onde vamos de 1 até ao número de vértices existentes, e no ciclo **interior** vamos de 0 até ao número de arcos existentes.

Dentro deste ciclo, se o **valor da distância do vértice source do arco + o peso do grafo** for **menor que a distância do vetor destination do arco**, fazemos as **novas atribuições ao valor da distância do vértice destination desse arco, como também o seu vértice predecessor**.

Adicionamos também ao **array das distâncias do indíce do vértice destination**, o **alor da distância do vértice source + o valor do peso do arco** correspondente.

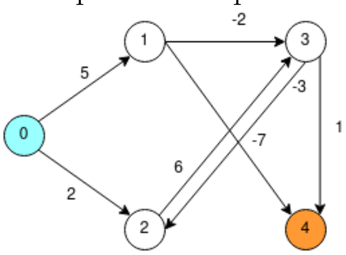
Para ver se existem ciclos negativos, fizemos mais outro for desde 0 até ao número de arcos, repetindo o if anterior, e para o caso de haver, retornar true pois quer dizer que encontra um deles.

Depois disto tudo, fizemos um **if** para ver se o **valor da distância do vértice final** até ao **vértice iniciall** era **negativa**, porque se fosse, o Dirk teria **perdido dinheiro**, retornando **true**.

Se nada destes ifs acontecesse, retornamos **false** pois quer dizer que o Dirk conseguiu fazer um caminho sem perda de dinheiro.

4 Descrição dos Grafos

A partir do input inicial do primeiro exemplo, temos o seguinte grafo:



0	0
1	+inf
2	+inf
3	+inf
4	+inf

Primeiro Relaxamento - O **Vértice 0 irá "alargar"** para **os seus adjacentes 1 2**, onde estes passam a ter **distâncias 5 e 2** respetivamente.

Segundo Relaxamento - De seguida o **Vértice 1 irá alargar** dando ao vértice 3 o valor da **distância 3** e ao vértice 4 a **distância -2**.

Terceiro Relaxamento - Posteriormente o **Vértice 2 irá alargar** não alterando o valor do seu vértice adjacente 3 (pois ficaria com distância superior à atual).

Quarto Relaxamento - O **Vértice 3 irá alargar** alterando o **valor de 4** que estava com **distância negativa**, passando a ter 3+1 = **4** de distância. Como o valor da distância do vértice 2 (adjacente de 3) estava em 2, este irá alterar o valor deste para 0 porque **5+(-2)+(-3) = 0 menor que 2**.

CONTINUAÇÃO DA DESCRIÇÃO DOS GRAFOS

Pelo que, as **distâncias finais** seriam:

Vértice	Distância
0	0
1	5
2	0
3	3
4	-2

5 Análise de Complexidades

5.1 Complexidade Temporal

Para a complexidade temporal a função que irá ter **mais influência** sobre esta será a **função Bellman-Ford**.
Na função temos dois ciclos for: o ciclo exterior que vai de 1 até ao número de vértices (que representa o número de quartos do input). O ciclo interior vai de 0 até ao número de arcos disponíveis (que representa o número de corredores do input).

Pelo que o **ciclo exterior** irá ter **complexidade de O(n)**, onde **n** representa o **número de vértices** do grafo, e o **ciclo interior** terá **complexidade de O(m)**, onde **m** representa o **número de arcos**.

Podemos então assim dizer que a **complexidade temporal do programa** será de **O(m * n)**.

5.2 Complexidade Espacial

Assumindo a Complexidade Espacial de acessos a arrays e a métodos de classes (como get.source de um arco) tem sempre complexidade constante (O(1)), não interferindo no resultado final desta.

Pelo que o que terá **importância** é:
O **array de vértices** com **tamanho n** (sendo n o número de vértices) tem **complexidade espacial O(n)**. O **array de arcos** com **tamanho m** (sendo m o número de arcos) com **complexidade espacial O(m)**. O **array das distâncias** com **d** (sendo d o número de vértices novamente) com **complexidade espacial O(d)**.

Por isso, a **complexidade espacial** será **O(n * m * d)**.

6 Conclusão e Comentários Extra

Concluindo, o grupo teve algumas dificuldades iniciais na escolha do algoritmo, e posteriormente numa boa implementação do mesmo (problemas relacionados com os ciclos do algoritmo), tendo também problemas com a complexidade do mesmo, pois tinhamos métodos que para cada vértice que queríamos procurar, fazia um for para encontrar a sua label, o que aumentava significativamente a mesma (resolvido com a criação dos arrays para vértices e arcos, uma abordagem muito mais simples e eficaz).

Para além disso, o grupo ficou a entender muito melhor o algoritmo dado, tendo a certeza que o trabalho ajudará futuramente em próximas avaliações teóricas que o contenham.