Trabalho Prático 3
Algoritmos I

Euller Saez Lage Silva

2019054501

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

eullersaez@ufmg.br

1. Modelagem computacional

O trabalho proposto consistia em desenvolver um sistema computacional capaz de resolver o problema de ajudar um jovem universitário em Nova Iorque a gastar a menor quantia possível dentro do sistema de metrô da cidade. Basicamente o jovem tem direito a uma quantia de descontos que vão se acumulando até uma quantidade D de escalas feitas nas estações, dentro de um tempo T. Após esse tempo, é obrigatório reiniciar os descontos. Como o jovem sabe que em alguns momentos pode esperar o tempo acabar em uma estação para reiniciar os descontos, isso pode ajudá-lo a pagar um custo menor, pois quando ainda dentro do limite T de tempo, porém já tendo se passado de D estações consecutivas aplicando os descontos cumulativos, não vale a pena continuar seguindo dentro do limite de tempo, já que o preço pago nessas estações será seu preço integral, o que aumentará o custo acumulado total.

Desse modo, é possível resolver o problema dado com o artifício da programação dinâmica, sem a necessidade de fazer a mesma chamada recursiva diversas vezes, apenas consultando uma posição de memória em vez de executar diversas pilhas de recursão. Logo, podemos definir a equação de Bellman para calcular o custo de ir de uma dada estação EA (estação atual) até a estação final, considerando que o desconto começa de uma estação ED (estação do desconto) da seguinte forma:

calcula_custo(EA,ED) =
{**0**, se EA é a estação final;

∞, se (EA-ED ≥ D) ou (vetor_tempos[EA]-vetor_tempos[ED] ≥ T);

desconto_acumulado[EA-ED]*vetor_custos[EA]+min(calcula_custo(EA+1,ED), calcula_custo(EA+1,EA+1)), caso contrário.}

Ou seja, caso a estação atual não seja a última ou ainda esteja dentro do limite (obtido pela diferença entre o tempo de uma estação e outra) de tempo T e do limite D de estações passíveis de se aplicar desconto consecutivamente, calcula-se o preço do bilhete com o devido desconto na estação atual e obtém-se o menor dentre os custos de percorrer até o final ou começando com o desconto a partir da próxima estação (EA+1) ou mantendo a estação em que o desconto começou nesta iteração (ED) a partir da próxima.

2. Estrutura de dados

As estruturas de dados neste trabalho prático consistiram no armazenamento das informações relativas a cada escala em arrays e a realização do cálculo do custo acumulado por meio de uma matriz de memoização, todos implementados como atributos ou em métodos de uma classe *metro* (mais detalhes abaixo). A escolha pela resolução por programação dinâmica se deu pelo fato de que foi uma forma de evitar possíveis chamadas recursivas em duplicata, além de auxiliar no entendimento do problema. De fato, como mencionado na seção 1, apesar de ser necessário um grande espaço auxiliar extra de armazenamento para o funcionamento, essa técnica em geral ajuda a reduzir o custo computacional de execução, pois não é necessário executar diversas pilhas de recursão, bastando apenas consultar uma posição de memória pré-computada.

A classe *metro,* mencionada anteriormente, serviu como base para armazenar todos os dados necessários da entrada e a realização do cálculo do custo acumulado. Ela possui dois atributos inteiros *numero_estacoes* e *max_escalas_desconto_cumulativo* para denotar o número de estações (ou escalas) presentes no metrô e o número máximo de escalas consecutivas cujo desconto ainda pode ser aplicado. Além disso possui três arrays, sendo dois do tipo inteiro (*vetor_custos* e *vetor_tempos*) e um do tipo double (*vetor_descontos*). O *vetor_custos* armazena em cada posição i o valor integral entrado do bilhete da escala na posição i. O *vetor_tempos,* por sua vez, armazena o tempo acumulado total até uma dada escala i, e, por fim, *vetor_descontos* armazena o complemento do desconto acumulado em porcentagem (ou seja, caso o desconto acumulado seja de 25% em i, o vetor armazenará 0.75 nesta posição) para cada estação i, dentre as *max_escalas_desconto_cumulativo* permitidas.

O construtor recebe *numero_estacoes* e *max_escalas_desconto_cumulativo,* atribui os devidos valores às variáveis e aloca espaço na memória para os arrays.

Os métodos *insere_desconto_acumulado*, *insere_tempo_acumulado* e *insere_preco* agem como *setters* e recebem as posições e os descontos, tempo e preço respectivamente de cada escala, armazenando-os com pequenas manipulações no *vetor_descontos, vetor_tempos* e *vetor_custos*, respectivamente, para que fiquem estruturados de forma adequada (como descrito acima em como ficam acumulados) na hora de calcular o custo.

O método *min* recebe dois números do tipo double e retorna aquele de menor valor.

Finalmente, *calcula_custo* recebe Tmax para o tempo máximo, aloca espaço para a matriz de memoização, de dimensões [numero_estacoes+1][numero_estacoes+1], pois assim evita-se acessar posições proibidas de memória quando na primeira iteração do loop externo for necessário consultar as posições equivalentes a ir do ponto final (após a escala final) até o ponto final, que deve ser zero. Após isso, zera a memoização por completo e a percorre calculando os devidos custos de baixo para cima, começando de *numero_estacoes* até 0, limitado pela diagonal principal. Assim, para cada elemento é checado em qual condição ele se encaixa e qual valor deve ser atribuído a ele, baseado na equação de Bellman disponível na

seção 1 e a algumas implementações vistas em aula, como a do Knapsack problem, por exemplo, fazendo as devidas alterações. Como é necessário saber o custo acumulado de percorrer da estação na posição 0 até a última, dado que o desconto começará na estação 0, é impresso justamente o elemento memoização[0][0] ao final, e é liberado o espaço tomado pela matriz. Abaixo está disponível o seu pseudocódigo:

---

calcula_custo(Tmax):
memoization[numero_estacoes+1][numero_estacoes+1] //declaração e alocação de espaço
Para todo elemento i,j de memoization:
memoization[i][j] = 0
fim-para
Para EA = numero_estacoes até 0
Para ED = EA até 0
Se EA == numero_estacoes **então**
memoization[EA][ED] = 0
fim-se
Caso contrário se EA-ED ≥ max_escalas_desconto_cumulativo ou vetor_tempos[EA]-vetor_tempos[ED] ≥ Tmax **então**
memoization[EA][ED] = ∞
fim-Caso contrário se
Caso contrário
memoization[EA][ED] = vetor_descontos[EA-ED] * vetor_custos[EA] + min(memoization[EA+1][EA+1], memoization[EA+1][ED])
fim-Caso contrário
fim-para
fim-para

print(memoization[0][0])
delete memoization

---

3. Instruções de compilação e execução

O programa foi desenvolvido na linguagem C++ por meio do Visual Studio Code para Windows da Microsoft Corporation, utilizando o Windows Subsystem for Linux (WSL) e compilado pelo compilador G++ da GNU Compiler Collection.

3.1 Compilação

Para compilar o programa primeiro é necessário abrir o diretório 2019054501_EULLER/src no terminal do linux e digitar o comando “make”.

3.2 Execução

Para executar o programa, após compilado, ainda dentro do mesmo diretório 2019054501_EULLER/src no terminal do linux, basta digitar o comando “./tp03”.

3.3 Limpeza de arquivos temporários

Para limpar todos os arquivos temporários gerados durante a compilação, ainda dentro do mesmo diretório 2019054501_EULLER/src no terminal do linux, basta digitar o comando "make clean".

4. Análise de complexidade

A complexidade de tempo do programa como um todo será o custo de todas as chamadas dos métodos implementados. Vale ressaltar que todos eles, exceto *calcula_custo*, têm tempo de execução constante $O(1)$. No entanto, durante a entrada os métodos são chamados em cada execução no máximo n vezes, onde n é o número de estações (ou escalas) do metrô. Isso acrescenta um custo linear de $O(n)$ ao programa.

Além disso, *calcula_custo* tem que alocar a matriz de memoização de dimensões [numero_estacoes+1][numero_estacoes+1] e apesar desta operação ter custo linear $O(n)$, para zerar a matriz e depois percorrê-la de baixo para cima, da direita para a esquerda, sendo limitado pela diagonal principal, é necessário fazer na ordem de $O(n^2)$ operações, o que aumenta o custo total do programa.

Assim, temos que

$$O(n) + O(n^2) = O(n^2)$$

e portanto a complexidade assintótica de tempo do programa é quadrática $O(n^2)$.

References

KLEINBERG, J., TARDOS, É. Algorithm Design. 1st. Edition. Pearson, March 16, 2005.