Trabalho Prático 3

Vitor Rodarte Ricoy - 2019007112

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

vitorricoy@ufmg.br

1. Introdução

O problema proposto foi implementar um programa para determinar o menor custo de uma viagem de metrô. Essa viagem é feita por meio de uma sequência de escalas e o preço da passagem de cada escala está condicionado a um sistema de descontos cumulativos de acordo com o número de escalas dentro de um intervalo de tempo pré-definido. Também é definido que é permitido esperar o tempo que for necessário para reiniciar o ciclo de descontos, ou seja, é possível esperar que o período em que um desconto é válido termine para reiniciar a contagem dos descontos cumulativos antes de pagar a próxima escala.

Para resolver o problema é fornecida a lista de escalas que serão usadas, as porcentagens da progressão acumulada de descontos, o custo da passagem de cada escala, o tempo de viagem de cada escala, o valor do maior intervalo de tempo possível para a duração do desconto acumulado, e o número máximo de escalas em que um desconto se aplica.

Por fim, o programa deve imprimir apenas o menor custo possível da viagem, considerando os descontos, com duas casas decimais.

2. Modelagem Computacional do Problema

Para modelar o problema foi pensado nos seguintes casos:

- Não existem mais escalas a serem tomadas, logo o custo da viagem é zero.
- Existe uma escala a ser tomada, mas um desconto inválido iniciado em outra escala é usado, portanto o custo da viagem pode ser visto como sendo infinito, já que o desconto inválido torna essa alternativa inviável.
- Existe uma escala a ser tomada com um desconto válido, logo o custo para tomá-la até o destino final é igual ao preço da passagem com o desconto atual somado ao preço das escalas seguintes.
    - A opção ótima para a escala seguinte será o menor custo entre iniciar um novo ciclo de descontos e continuar o ciclo de descontos atual. Vale notar que o resultado igual a infinito de um par de escala e desconto inválidos trata o caso em que o desconto atual é inválido para a próxima escala.

Com base nesses casos, a solução do problema foi modelada com uma equação de Bellman, para possibilitar que ele seja resolvido com a técnica de programação dinâmica. A equação modelada para o problema foi a seguinte:

$$OPT(i,u) = \begin{cases} 0 & \text{se } i = n \\ \infty & \text{se } tempo_{[e,i)} \geq t \ \vee i < u \ \vee u = d \\ (1 - descontos_u) * valor_i + & \\ min\{OPT(i+1,0), OPT(i+1,u+1)\} & \text{caso contrario} \end{cases}$$

A equação define a função $OPT(i, u)$, que retorna o custo mínimo para alcançar o final da viagem a partir da escala i, usando um desconto iniciado há u escalas, ou seja, o desconto usado na escala i é o u-ésimo desconto cumulativo. Vale notar que i é indexado a partir de zero, variando entre 0 e n, enquanto u varia de 0 a d, que são, respectivamente, o número mínimo e máximo de descontos cumulativos.

Também percebe-se que foi definido o segundo caso, em que o valor da função é infinito, para indicar um par i e u inválidos, pois um desconto iniciado há u escalas não pode ser aplicado a i. Isso acontece devido ao tempo gasto nas u viagens até i exceder o limite de tempo de um desconto; por ser impossível que um desconto em i seja iniciado há u escalas, ou seja, por ter que $i < u$; ou por u indicar um desconto que, ao ser usado em i, excede o número máximo de descontos cumulativos. O valor infinito faz com que essa ação nunca seja escolhida como ótima.

Também foram definidos os seguintes termos para a equação: e indica a escala em que o desconto aplicado a i foi iniciado; $tempo_{[e, i)}$ indica o tempo gasto da escala e até o momento do embarque para a escala i; $descontos_u$ indica o valor do u-ésimo desconto acumulado; $valor_i$ indica o preço da passagem da escala i; t indica a duração do intervalo em que um desconto é válido; d indica o número máximo de descontos consecutivos; e, por fim, n indica o número total de escalas.

Com a definição dessa equação de Bellman, podemos concluir que a solução do problema é dada pelo valor de $OPT(0, 0)$.

3. Estrutura de Dados e Algoritmos

3.1. Estrutura de Dados

Para a solução foram utilizados três $vectors$ para guardar as listas dos valores da entrada, sendo que o vetor dos percentuais de desconto é representado como um vetor de soma de prefixos, para que a posição j indique a porcentagem exata do desconto da j-ésima escala com desconto, já que estes são cumulativos.

Também foi utilizado um $vector$, para implementar um vetor de soma de prefixos para o tempo de viagem das escalas, e dois $vectors$ aninhados para implementar a matriz que guarda os dados da programação dinâmica.

3.1.1. Soma de Prefixos

A soma de prefixos foi implementada em duas situações no código. A primeira para gerar o valor final do desconto cumulativo da i-ésima escala de um trecho com desconto. Essa soma de prefixo é a mais simples e é representada pelo seguinte pseudo-código:

```
lê o vetor descontoPercentual de tamanho d
para i de 1 até d-1
    descontoPercentual[i] += descontoPercentual[i-1]
    descontoPercentual[i] = min(descontoPercentual[i], 1)
```

Foi escolhida a soma de prefixo para esse caso para evitar o cálculo repetitivo da soma das porcentagens de desconto desde a escala que iniciou o desconto. Vale notar que o valores das porcentagens foram convertidos para decimal, e que é feita uma verificação para que a soma não passe de 100%.

A segunda situação em que a soma de prefixos foi utilizada foi para calcular a soma dos tempos das viagens de um trecho em tempo constante. O vetor dessa soma foi criado, diferente do restante do código, indexado de 1, para que se tenha o índice zero com o valor zero, o que facilita o cálculo da soma de um intervalo do vetor. Segue o pseudo-código desta estrutura:

```
le o vetor tempoViagem de tamanho n

declara o vetor somaPrefixoTempo de tamanho n+1

somaPrefixoTempo[0] = 0

para i de 1 até n
    somaPrefixoTempo[i] = tempoViagem[i-1] + somaPrefixoTempo[i-1]
```

O pseudo-código do cálculo da soma dos valores de um intervalo utilizando essa estrutura, considerando que $inicio$ e fim são indexados de zero, é o seguinte:

```
somaTemposViagem(inicio, fim)

    retorne somaPrefixoTempo[fim+1]-somaPrefixoTempo[inicio]
```

Foi escolhida a soma de prefixo para esse caso porque, pela construção da solução com programação dinâmica, será necessário saber o tempo de uma viagem entre as escalas e e $i - 1$, somando todos os tempos de viagem desse intervalo, e a soma de prefixo é uma estrutura simples que realiza essa operação em $O(1)$.

3.2. Algoritmos

3.2.1. Solução da Equação de Bellman

O principal algoritmo implementado foi a solução bottom-up de programação dinâmica para a equação de Bellman apresentada na seção anterior.

Vale destacar que a solução foi implementada com uma otimização de espaço em que a matriz da programação dinâmica salva apenas os valores para os parâmetros i e $i + 1$. Essa otimização é possível devido ao cálculo da função para a escala i depender apenas dos cálculos para a escala $i + 1$. Tal modificação reduz a complexidade de espaço da matriz de $O(nd)$ para $O(d)$, tendo como única desvantagem a impossibilidade de reconstruir as ações tomadas para gerar a solução, o que não é necessário para esse problema. A otimização foi implementada utilizando a paridade do identificador de cada escala, modificando-se pouco o código original.

Primeiramente, é implementado o caso base da equação de Bellman, que é dado pelo identificador da escala igual a n. Esse caso é implementado ao inicializar a matriz da programação dinâmica com zeros.

Já para o cálculo dos outros casos da equação são executados dois loops for aninhados. O primeiro for indica os valores do argumento i da equação de Bellman e é executado, com incremento decrescente, a partir da escala $n - 1$ até a escala 0. Essa ordem foi escolhida pelo fato do cálculo da escala i depender apenas do valor da função para a escala $i + 1$. Já o segundo for indica os valores do argumento u e é executado, com incremento decrescente, a partir de d até 0. Mesmo que alguns valores escolhidos para u trivialmente caiam no caso em que a função é igual a infinito, a implementação foi feita dessa forma pois ela é mais simples e executa no tempo esperado.

Para cada iteração interna da construção bottom-up é calculada a escala em que o desconto usado em i foi iniciado e o tempo gasto para ir dessa escala até $i - 1$, por meio do vetor de soma de prefixos.

Depois de calculados esses valores auxiliares, é verificado se o tempo gasto para ir da escala que iniciou o desconto até $i - 1$excede o tempo máximo de viagem, se é impossível se ter um desconto no qual i seja a u-ésima escala visitada, ou se o desconto usado em i excede o número máximo de descontos cumulativos. Caso algum desses casos aconteça, a posição sendo calculada da matriz da programação dinâmica recebe infinito e o próximo passo do loop é executado.

Caso contrário, é salvo nessa posição da matriz da programação dinâmica o custo de comprar a passagem da escala i com o u-ésimo desconto cumulativo. Além disso, também é adicionado na posição da matriz o valor mínimo entre dois custos: o de ir de $i + 1$ até o fim da viagem iniciando outro desconto em $i + 1$, e o de ir de $i + 1$ até o fim da viagem mantendo o ciclo de descontos utilizado em i.

Após executados os cálculos, a resposta do problema está na primeira linha e na primeira coluna da matriz da programação dinâmica. Para deixar o resultado em duas casas decimais foi usado o $setprecision$ no comando $cout$. Segue o pseudo-código da programação dinâmica implementada:

```
inicializa uma matriz pd 2 por d+1 com zeros

para escala decrescente de n-1 até 0
    para descontosConsecutivos decrescente de d até 0
        escalaDesconto = escala - descontosConsecutivos
        se escalaDesconto < 0 ou descontosConsecutivos == d
            pd[escala%2][descontosConsecutivos ] = infinito
            continua o loop na sua próxima iteração
        tempoUltDesc = somaPrefixoTempo[escala]-somaPrefixoTempo[escalaDesconto]
        se tempoUltDesc < t
            preco = (1-descontoPercentual[descontosConsecutivos])*custoBilhete[escala]
            pd[escala%2][descontosConsecutivos ] = preco
            valorDescontoAtual = pd[(escala+1)%2][descontosConsecutivos+1]
            valorNovoDesconto = pd[(escala+1)%2][0]
            pd[escala%2][descontosConsecutivos ] += min(valorDescontoAtual, valorNovoDesconto)
        senao
            pd[escala%2][descontosConsecutivos ] = infinito
resultado = pd[0][0]
```

Foi escolhida a implementação bottom-up da programação dinâmica por ela possibilitar a otimização de espaço utilizada e por ter uma implementação mais fácil de ser analisada do que a implementação top-down.

4. Análise de Complexidade de Tempo

Para a análise da complexidade de tempo do programa, primeiramente vamos analisar a complexidade de algumas partes que o compõem, para facilitar a conclusão final. Também para auxiliar a análise, iremos definir n como o número de escalas da viagem, t como a duração máxima de um desconto e d como o número máximo de escalas com o desconto cumulativo. As análises feitas são referentes ao custo assintótico de tempo no pior caso.

4.1. Entrada dos dados

A entrada dos dados é dada por um comando de leitura do valor de n, t e de d, por um loop para ler o percentual de cada desconto cumulativo e outro loop para ler o tempo de viagem e o custo do bilhete de cada escala.

A leitura das primeiras três variáveis tem custo $O(1)$, já que é feita apenas com um cin. O primeiro loop tem custo $O(d)$, já que é executado um cin para cada percentual de desconto. Por fim, o segundo loop tem custo $O(n)$, já que executa somente um cin para cada escala. Portanto, toda a leitura dos dados tem complexidade $O(1 + d + n) = O(n + d)$.

4.2. Cálculo das Somas de Prefixo

O cálculo das somas de prefixo é feito por meio de dois comandos for. O primeiro, que calcula a soma de prefixos dos descontos, é executado $d - 1$ vezes, logo é da ordem de $O(d)$. O segundo, que calcula a soma de prefixos dos tempos das viagens, é executado n vezes, portanto é da ordem de $O(n)$. Assim, temos que os cálculos das somas de prefixo têm um custo total da ordem de $O(d + n)$.

4.3. Execução da Programação Dinâmica

O primeiro passo da execução da programação dinâmica é declarar a matriz, o que custa $O(d)$, já que a matriz tem um tamanho de dois por $d + 1$ e é inicializada com zeros. Após a declaração da matriz é executado o algoritmo da programação dinâmica de fato.

O algoritmo é composto por dois loops for. O primeiro deles é executado n vezes e o segundo é executado $d + 1$ vezes. Podemos perceber que todos os comandos executados no corpo do segundo for são $O(1)$, já que todas as operações realizadas nele são trivialmente $O(1)$.

Dessa forma, concluímos que o custo da execução da programação dinâmica depende dos limites dos loops for e do custo para declarar a matriz. Assim, o custo é igual a $O(nd + d)$.

4.4. Saída de Dados

A saída de dados executa apenas um comando $cout$, que custa $O(1)$, para o resultado. Logo, tem custo $O(1)$.

4.5. Análise do Programa Completo

Por fim, temos que a leitura de dados é $O(n + d)$, o cálculo dos vetores de soma de prefixos é $O(n + d)$, a execução da programação dinâmica é $O(nd + d)$ e a saída de dados é $O(1)$.

Também temos alguns trechos que são trivialmente menos complexos do que a execução dessas seções, como a inicialização dos vetores utilizados. Como o programa completo consiste em, basicamente, executar tais trechos de código, sua complexidade é igual a $O(n + d + n + d + nd + d + 1) = O(nd + n + d)$.

Assim, concluímos que a complexidade de tempo da solução é igual a $O(nd + n + d)$.

Referências

Kleinberg, Jon (2006). Algorithm Design: Chapter 6: Dynamic Programming. Pearson Education India.

Almeida, Jussara. Slides da Disciplina Algoritmos I. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2021.