DCC206 – Algoritmos 1

Aula 16 – Programação Dinâmica – Parte 2

Professor Renato Vimieiro

DCC/ICEx/UFMG

Introdução

- Na aula passada, iniciamos nossos estudos sobre programação dinâmica
- Vimos que essa estratégia é adequada para situações em que o problema principal pode ser dividido em subproblemas não independentes
- Outro ponto discutido foi que o desenvolvimento de um algoritmo de programação dinâmica envolvia, em geral, quatro etapas
    1. Caracterizar a estrutura de uma solução ótima
    2. Definir uma relação de recorrência para o valor de uma solução ótima
    3. Calcular o valor de uma solução ótima por um procedimento bottom-up
    4. Construir uma solução ótima a partir dos valores computados

Introdução

- Seguiremos investigando a aplicação de PD em outros problemas práticos na aula de hoje
- Tentaremos explicitar os quatro passos necessários para a elaboração da solução
- Vamos analisar os problemas e estudar como o objetivo de cada etapa pode ser conquistado

Multiplicação de cadeias de matrizes

- Vamos iniciar por um problema simples, mas, como discutido anteriormente, de muito apelo prático atualmente
- Suponha que queremos multiplicar *n* matrizes $A_1A_2...A_n$
- Sabemos que a multiplicação de matrizes é associativa
- Portanto, podemos executar a tarefa de diferentes formas
- Diferentes formas de multiplicar levam a diferentes custos

Multiplicação de cadeias de matrizes

- Por exemplo, suponha que temos três matrizes A_1 com dimensões 10x100, A_2, com dimensões 100x5, e A_3, com dimensões 5x50
- Queremos calcular o produto das três matrizes $A_1A_2A_3$
- Novamente, como a operação é associativa, podemos fazer $(A_1A_2)A_3$ ou $A_1(A_2A_3)$
- Supondo que usaremos o algoritmo tradicional para multiplicação de matrizes, a primeira requer 10x100x5 multiplicações de escalares para obter (A1A2), mais 10x5x50 multiplicações para obter o produto final
  - Totalizando 7500 multiplicações
- A segunda escolha requer 100x5x50 multiplicações para obter (A2A3) e outras 10x100x50 multiplicações para obter o produto final
  - Totalizando 75000
- Ou seja, a primeira escolha é 10x mais rápida que a segunda

Multiplicação de cadeias de matrizes

- Veja que nosso problema não é efetivamente computar o produto das matrizes, mas sim elaborar uma estratégia ótima para computá-lo
- Considerando *n* matrizes $A_1A_2..A_n$ e que cada matriz A_i tem dimensões p_{i-1}x p_i, nosso problema é obter uma parentização da sequência de forma a otimizar o produto das matrizes
- É fácil perceber que cada forma de parentizar representa uma árvore binária diferente com *n* folhas
- Assim, resolver esse problema com um algoritmo ingênuo tem custo $O(2^n)$
- Vamos tentar construir um algoritmo de PD para resolver o problema
- Logo, vamos seguir os quatro passos para elaborar a solução

Multiplicação de cadeias de matrizes

- O primeiro passo na elaboração do algoritmo é mostrar que o problema possui subestrutura ótima
- Suponha que queremos parentizar o produto $A_iA_{i+1}..A_j$, para $i \le j$
  - Para simplificar a notação, vamos denotar esse produto por $A_{i..j}$
- Se $i=j$, então não resta nada a fazer
- No caso de $i < j$, devemos encontrar um $i \le k < j$ para parentizar os dois produtos $A_{i..k}\ A_{k+1..j}$
- Nossa escolha de k deve ser guiada pela minimização do número de operações

Multiplicação de cadeias de matrizes

- Sabemos que o custo total do produto $A_{i..j}$ é a soma das multiplicações realizadas em $A_{i..k}$, mais as multiplicações em $A_{k+1..j}$ e as multiplicações do produto entre os dois resultados
- Suponha que a escolha do k tenha sido ótima, então temos que a parentização de $A_{i..k}$ também deve ser ótima
- Caso contrário, teríamos uma outra forma de parentizar cujo total de multiplicações em $A_{i..j}$ seria menor
  - O que seria uma contradição
- O mesmo raciocínio se aplica a $A_{k+1..j}$
- Portanto, o problema apresenta subestrutura ótima

Multiplicação de cadeias de matrizes

- O segundo passo consiste em desenhar uma relação de recorrência para computar a solução ótima a partir da solução dos subproblemas
- Vamos denotar por m[i,j] o menor número de multiplicações necessárias para computar $A_{i..j}$
- Como deduzido anteriormente, m[i,i] = 0 para $1 \leq i \leq n$
- Para o caso em que $i < j$, temos que o custo de Ai..j é o custo de Ai..k, mais o custo de Ak+1..j, e o custo do produto dos dois resultados que será $p_{i-1} \times p_k \times p_j$
  - $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j$
- Como precisamos encontrar o k ótimo, temos a seguinte relação
- $m[i,j] = \begin{cases} 0, & i = j \\ min_k \{m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j,\} & i < j \end{cases}$

Multiplicação de cadeias de matrizes

- O terceiro passo é obter um procedimento bottom-up para computar o valor
- Vimos que é possível desenvolver um algoritmo recursivo diretamente a partir da relação de recorrência
- Contudo, esse algoritmo teria custo proibitivo
- Então, vamos desenhar um algoritmo iterativo para resolver o problema
- Um requisito para tal é que o número de subproblemas seja polinomial
- Para calcular m[1,n], precisamos potencialmente de todos os valores de m, ou seja m[i,j] para $1 \leq i \leq j \leq n$
- Isso resulta em um número polinomial de subproblemas
- Assim, podemos seguir em frente

Multiplicação de cadeias de matrizes

- Note que, para resolver o produto $A_{i..j}$, temos uma sequência de tamanho j-i+1
- Para resolver $A_{i..k}$, temos uma sequência de tamanho k-i+1 < j-i+1
- Analogamente, para resolver $A_{k+1..j}$, temos uma sequência j - k < j-i+1
- Em outras palavras, resolver uma sequência com mais matrizes requer a solução de sequências com menos matrizes
- Portanto, nossa solução bottom-up deve resolver os problemas com sequências de tamanho 1, depois 2, ... até chegar em n
  - Intuitivamente, primeiro descobrimos as formas ótimas de multiplicar duas matrizes para então obter a forma ótima para até 3 matrizes

Multiplicação de cadeias de matrizes

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                  // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Multiplicação de cadeias de matrizes

- O quarto passo é a (re)construção da solução ótima
- A matriz m guarda somente os valores mínimos para computar o produto $A_{i..j}$
- Assim, precisamos de uma variável auxiliar que nos permita reconstruir a solução ótima
- Vamos armazenar a escolha do k ótima para $A_{i..j}$ em dois subproblemas em s[i,j]
- Como s[i,j] guarda o valor de k tal que $A_{i..k}$ e $A_{k+1..j}$ resulta no menor número de multiplicações, sabemos que $A_{1..n}$ é dividido em $A_{1..s[1,n]}$ e $A_{s[1,n]+1..n}$
- Se aplicarmos o raciocínio recursivamente, conseguimos reconstruir a solução ótima

Multiplicação de cadeias de matrizes

- Assim, chegamos ao algoritmo

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Multiplicação de cadeias de matrizes

- Exemplo:

A1	A2	A3	A4
30 x 35	35 x 15	15 x 5	5 x 10

Multiplicação de cadeias de matrizes

- Em termos de custo, percebemos que existem três laços aninhados no algoritmo principal cada um executando O(n) vezes
- Assim, o custo do algoritmo é $O(n^3)$
  - Ainda assim é melhor que o custo exponencial do algoritmo ingênuo
- A aplicação do algoritmo é vantajosa pois reduz o custo da multiplicação de matrizes
  - Que é dado em função das dimensões (e não do número de matrizes!)
- O algoritmo requer $O(n^2)$ espaço para armazenar as matrizes m e s

Programação de uma linha de montagem

- Vamos analisar outros problemas e verificar soluções com programação dinâmica para os mesmos
- Começaremos com o problema de uma linha de montagem
- Suponha que uma indústria possui duas linhas de montagem com *n* estações para fabricar seu produto
- Cada estação $1 \leq j \leq n$ de uma linha i = 1 ou 2 é denotada por S_{ij}
- Cada estação possui um tempo de montagem distinto (a_{ij})

Programação de uma linha de montagem

- Para ser montado, as peças que compõem o produto devem ser transportadas do depósito até à linha de montagem
  - O tempo de transporte de entrada é dado por e_i
- Da mesma forma, ao ser finalizado, o produto deve ser transportado para um armazém
  - O tempo de transporte de cada linha até o armazém é x_i
- O produto avança de uma estação para outra com tempo desprezível
  - No entanto, se a próxima estação estiver sobrecarregada, o produto pode ser deslocado para a estação equivalente na outra linha em tempo t_{ij}

Programação de uma linha de montagem

- O problema é minimizar o tempo necessário para a montagem de um produto
- Em outras palavras, quais estações nas linhas 1 e 2 devem ser escolhidas
  - Um algoritmo exaustivo (força-bruta) teria custo $O(2^n)$ (cada estação pode ou não fazer parte da solução)
- Podemos fazer melhor usando programação dinâmica
  - Para isso, devemos seguir as 4 etapas mencionadas anteriormente

Programação de uma linha de montagem

1. Caracterizar a estrutura de uma solução ótima

- A montagem mais rápida para o produto passando pela estação S_{1j} pode ser facilmente computada para j=1
- Para $2 \leq j \leq n$, existem duas opções:
  - O produto veio de S_{1j-1} com tempo desprezível de transição; ou
  - O produto veio de S_{2j-1} com tempo de transição t_{2j-1}
- O caminho mais rápido passando por S_{1j} envolve o caminho mais rápido até S_{1j-1} e S_{2j-1}
  - O mesmo raciocínio se aplica a S_{2j}
- O problema apresenta subestrutura ótima

Programação de uma linha de montagem

2. Solução recursiva

- A solução ótima passando por uma estação em uma linha envolve as soluções ótimas passando pela estação anterior em ambas as linhas
- Seja f[i,j] o menor tempo gasto na montagem desde a origem até a estação S_{ij}
- A solução ótima do problema f* é
  - $f^* = \min(f[1,n] + x_1, f[2,n] + x_2)$
- O tempo necessário até a estação 1 em cada linha é
  - $f[i,1] = e_i + a_{i1}$

Programação de uma linha de montagem

- O tempo necessário até uma estação S_{1j}
  - $f[1,j] = \begin{cases} e_1 + a_{11}, & se\ j = 1 \\ \min(f[1,j-1] + a_{1j}, f[2,j-1] + a_{1j} + t_{2j-1}, & se\ j \geq 2 \end{cases}$
- O tempo necessário até uma estação S_{2j}
  - $f[2,j] = \begin{cases} e_2 + a_{21}, & se\ j = 1 \\ \min(f[2,j-1] + a_{2j}, f[1,j-1] + a_{2j} + t_{1j-1}, & se\ j \geq 2 \end{cases}$

Programação de uma linha de montagem

- Podemos novamente derivar um algoritmo recursivo diretamente da definição de menor tempo necessário até uma estação para calcular a solução ótima
  - Mais uma vez, vários valores $f[i,j]$ serão computados desnecessariamente diversas vezes
- Alternativamente podemos aplicar programação dinâmica para solucionar o problema
  - Armazenamos $f[i,j]$ em uma tabela
  - Computamos os valores para $j=1$ até n para as duas linhas iterativamente

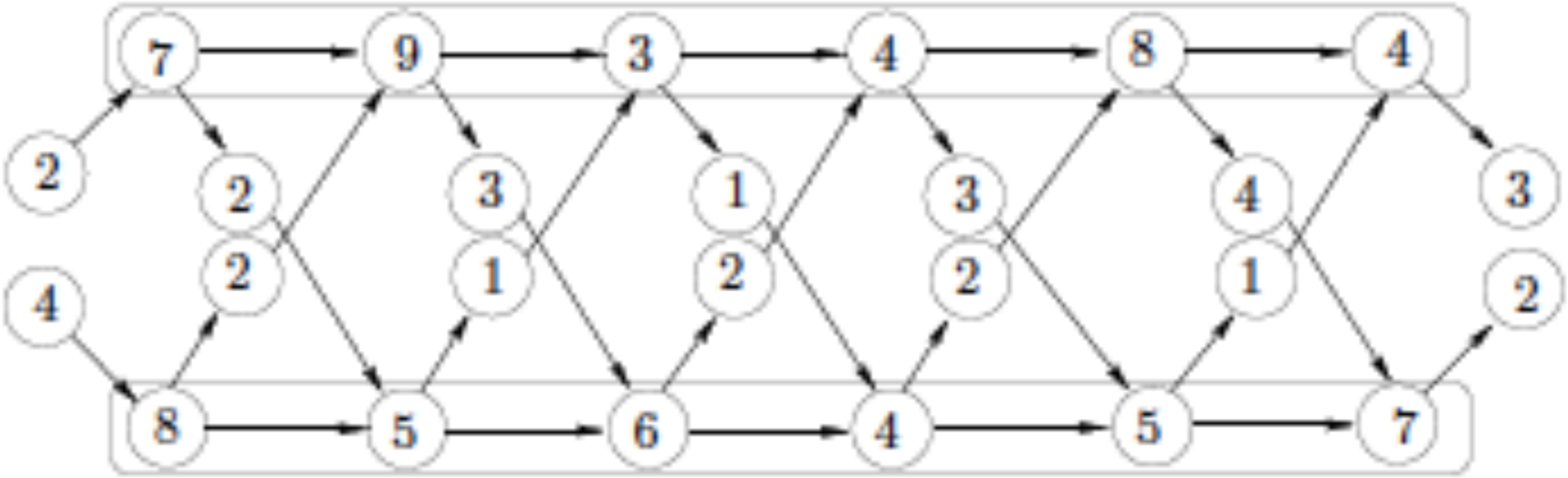
Programação de uma linha de montagem

```
Montagem(a,t,e,x,n)
    f[1,1] = e[1] + a[1,1]
    f[2,1] = e[2] + a[2,1]
    para j=2 até n
        f[1,j] = min(f[1,j-1]+a[1,j],f[2,j-1]+a[1,j]+t[2,j-1])
        f[2,j] = min(f[2,j-1]+a[2,j],f[1,j-1]+a[2,j]+t[1,j-1])
    f* = min(f[1,n]+x[1],f[2,n]+x[2])
```

Programação de uma linha de montagem

```
imprimeCaminho(i,j)
    se j > 1
        se f[i][j] == f[i][j-1]+a[i][j-1]
            imprimeCaminho(i,j-1)
        senão
            imprimeCaminho((i+1)%2,j-1)
    imprimir “linha” i “estacao” j
```

Programação de uma linha de montagem



Leitura

- Capítulo 15 (CLRS)
- Capítulo 6 (Kleinberg e Tardos)
- Capítulo 8 (Livitin)