MATRIX CHAIN MULTIPLICATION

Overview

- What is Dynamic Programming?
- Matrix-Chain Multiplication

What is Dynamic Programming?

- Dynamic Programming is a technique for algorithm design.
- It is a tabular method in which it uses divide-and-conquer to solve problems.
- dynamic programming is applicable when the subproblems are not independent.
- So to solve a given problem, we need to solve different parts of the problem.

Dynamic Programming Steps

- A dynamic programming approach consists of a sequence of 4 steps

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information

MATRIX CHAIN MULTIPLICATION

Input: a chain of matrices to be multiplied

Output: a parenthesizing of the chain

Objective: minimize number of steps needed for the multiplication

- Suppose we have a sequence or chain $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied

  That is, we want to compute the product $A_1 A_2 \ldots A_n$

- There are many possible ways (parenthesizations) to compute the product

Matrix Chain Multiplication cont..

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
  - 5 different orderings = 5 different parenthesizations

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

- Matrix multiplication is **associative** , e.g.,

$$A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3),$$

so parenthenization does not change result.

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

Algorithm..

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
    return C
```

•The time to compute C is dominated by the number of scalar multiplications.

•To illustrate the different costs incurred by different paranthesization of a matrix product.

Example: Consider three matrices $\mathbf{A_{10\times100}}$, $\mathbf{B_{100\times5}}$, **and** $\mathbf{C_{5\times50}}$ There are 2 ways to parenthesize

$((AB)C) = D_{10\times5} \cdot C_{5\times50}$

$AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications

$DC \Rightarrow 10\cdot5\cdot50 = 2{,}500$ scalar multiplications

Total: 7,500

$(A(BC)) = A_{10\times100} \cdot E_{100\times50}$

$BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications

$AE \Rightarrow 10\cdot100\cdot50 = 50{,}000$ scalar multiplications

Total: 75,000

- Matrix-chain multiplication problem
  - ✓ Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for $i=1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$
  - ✓ Parenthesize the product $A_1A_2\ldots A_n$ such that the total number of scalar multiplications is minimized

Counting the Number of Parenthesizations

- Before solving by Dynamic programming exhaustively check all paranthesizations.
- P(n) : paranthesization of a sequence of n matrices

- We can split sequence between kth and $(k+1)$st matrices for any $k=1, 2, \dots, n-1$, then parenthesize the two resulting sequences independently, i.e.,

$$(A_1 A_2 A_3 \dots A_k)(A_{k+1} A_{k+2} \dots A_n)$$

- We obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2\,. \end{cases}$$

- The recurrence generates the sequence of Catalan Numbers
- Solution is $P(n) = C(n\text{-}1)$ where

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega(4^n/n^{3/2})$$

- The number of solutions is exponential in n
- Therefore, brute force approach is a poor strategy

1.The Structure of an Optimal Parenthesization

Step 1: Characterize the structure of an optimal solution

- $A_{i..j}$: matrix that results from evaluating the product $A_i\ A_{i+1}\ A_{i+2} \ldots A_j$
- An optimal parenthesization of the product $A_1A_2 \ldots A_n$
  - Splits the product between A_k and A_{k+1}, for some $1 \leq k < n$

    $(A_1A_2A_3 \ldots A_k) \cdot (A_{k+1}A_{k+2} \ldots A_n)$
  - i.e., first compute $A_{1..k}$ and $A_{k+1..n}$ and then multiply these two
- The cost of this optimal parenthesization

  Cost of computing $A_{1..k}$

  + Cost of computing $A_{k+1..n}$

  + Cost of multiplying $A_{1..k} \cdot A_{k+1..n}$

Optimal (sub)structure:

• Suppose that optimal parenthesization of Ai;j splits between Ak and Ak+1.

• Then, parenthesizations of Ai;k andAk+1;j must be optimal, too

(otherwise, enhance overall solution — subproblems are independent!).

➢ Construct optimal solution:

1. split into subproblems (using optimal split!),
2. parenthesize them optimally,
3. combine optimal subproblem solutions.

2. Recursively define value of optimal solution

•Let m[i; j] denote **minimum number of scalar multiplications** needed to compute Ai;j = Ai*Ai+1….Aj (full problem: m[1; n]).

Recursive definition of m[i; j]:

- if i = j, then

m[i; j] = m[i; i] = 0

(Ai;i = Ai, no mult. needed).

- if $i < j$, assume optimal split at k, $i \leq k < j$. $A_{i,k}$ is $p_{i-1} \times p_k$ and $A_{k+1,j}$ is $p_k \times p_j$, hence

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j.$$

- $m_{ij} = m_{ik} + m_{k+1,j} + p_{i-1} \times p_k \times p_j$
  - We do not know k, but there are j-i possible values for k; $k = i, i+1, i+2, \ldots, j-1$

$$m[i,j] = \begin{cases} 0 & if\ i = j \\ \min\limits_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} & if\ i < j \end{cases}$$

- We also keep track of optimal splits:

$$s[i,j] = k \iff m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$$

3.Computing the Optimal Cost

An important observation:

- We have relatively few subproblems
  - one problem for each choice of i and j satisfying $1 \le i \le j \le n$
  - total $n + (n-1) + \ldots + 2 + 1 = \frac{1}{2}n(n+1) = \Theta(n^2)$ subproblems
- We can write a recursive algorithm based on recurrence.
- However, a recursive algorithm may encounter each subproblem many times in different branches of the recursion tree
- This property, overlapping subproblems, is the second important feature for applicability of dynamic programming

Computing the Optimal Cost(cont..)

Compute the value of an optimal solution in a bottom-up fashion

- matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$
- the input is a sequence $\langle p_0, p_1, \ldots, p_n \rangle$ where length[p] = $n + 1$

Procedure uses the following auxiliary tables:

- $m[1 \ldots n, 1 \ldots n]$: for storing the $m[i, j]$ costs
- $s[1 \ldots n, 1 \ldots n]$: records which index of k achieved the optimal cost in computing $m[i, j]$

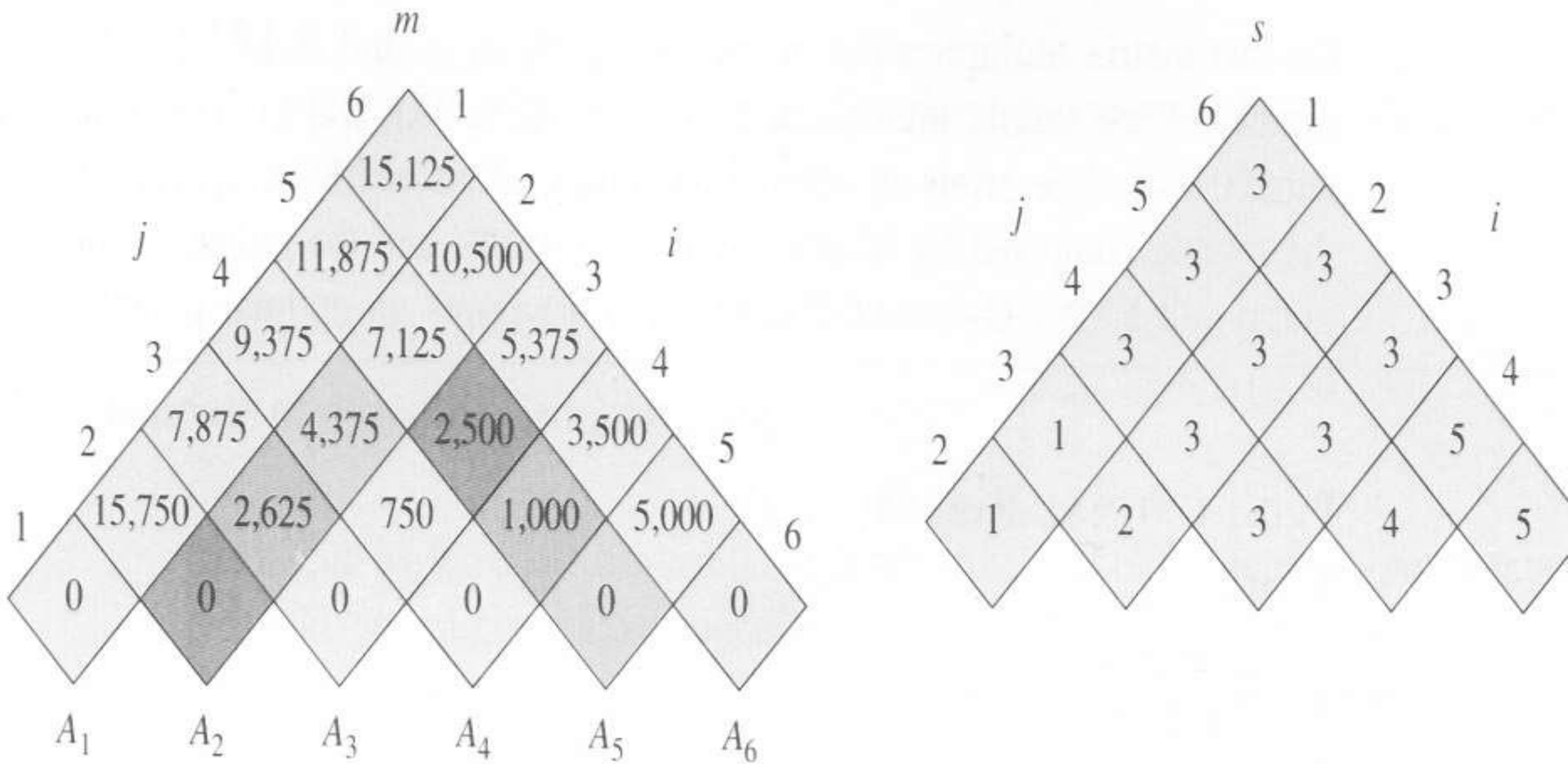
Algorithm for Computing the Optimal Costs

```
MATRIX-CHAIN-ORDER(p)
    n ← length[p] -1
    for i ← 1 to n do
        m[i, i] ← 0
    for ℓ ← 2 to n do
        for i ← 1 to n - ℓ + 1 do
            j ← i + ℓ - 1
            m[i, j] ← ∞
            for k ← i to j-1 do
                q ← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
                if q < m[i, j] then
                    m[i, j] ← q
                    s[i, j] ← k
    return m and s
```

Example:

$$
\begin{array}{lll}
A_1 & 30\times 35 & = p_0 \times p_1 \\
A_2 & 35\times 15 & = p_1 \times p_2 \\
A_3 & 15\times 5 & = p_2 \times p_3 \\
A_4 & 5\times 10 & = p_3 \times p_4 \\
A_5 & 10\times 20 & = p_4 \times p_5 \\
A_6 & 20\times 25 & = p_5 \times p_6
\end{array}
$$

The m and s table computed by MATRIX- CHAIN-ORDER for n=6



$m[2,5]=$

$\mathbf{min\{}m[2,2]+m[3,5]+p_1p_2p_5=0+2500+35\times15\times20$

$=13000,$

$m[2,3]+m[4,5]+p_1p_3p_5=2625+1000+35\times5\times$

$20=7125,$

$m[2,4]+m[5,5]+p_1p_4p_5=4375+0+35\times10\times20$

$=11374$

$\mathbf{\}}$

$=7125$

4.Constructing an Optimal Solution

- MATRIX-CHAIN-ORDER determines the optimal # of scalar mults/adds
  - needed to compute a matrix-chain product
  - it does not directly show how to multiply the matrices
- That is,
  - it determines the cost of the optimal solution(s)
  - it does not show how to obtain an optimal solution
- Each entry $s[i, j]$ records the value of k such that optimal parenthesization of $A_i \dots A_j$ splits the product between A_k & A_{k+1}
- We know that the final matrix multiplication in computing $A_{1 \dots n}$ optimally is $A_{1 \dots s[1,n]} \times A_{s[1,n]+1,n}$

Earlier optimal matrix multiplications can be computed recursively

Given:

- the chain of matrices $A = \langle A_1, A_2, \ldots A_n \rangle$
- the s table computed by MATRIX-CHAIN-ORDER

The following recursive procedure computes the matrix-chain product $A_{i..j}$

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
    if j > i then
        X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
        Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j]+1, j)
        return MATRIX-MUTIPLY(X, Y)
    else
        return A_i
```

Invocation: MATRIX-CHAIN-MULTIPLY(A, s, 1, n)

```
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 ")"
```

Thank you