

Lecture 11: Dynamic Programming Matrix-chain Multiplication Problem

Course Learning Outcome

- Use **algorithm design methods**, such as exhaustive search, divide-and-conquer and **dynamic programming**, to develop efficient algorithms.

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

Read Chapter 15 of *Introduction to Algorithms*

Dynamic Programming

Dynamic programming is a metatechnique (not an algorithm) like the divide-and-conquer method. It is used to create algorithms for problems that can be solved by combining solutions to smaller subproblems.

However, it is a method that is most effective when a subproblems recur again and again in other subproblems.

In such a case, the divide-and-conquer techniques would redo the subproblems each time, resulting in much unnecessary work.

Dynamic programming is often used for solving **optimization problems**, in which a set of choices must be made to arrive at an optimal solution to a problem (minimizing or maximizing some value associated with the problem). Note that there may be more than one optimal solution to a problem.

- Optimization problems: e.g find the shortest path ...
- Decision problems (YES/NO): e.g is there is a path of k (threshold or less) ...

Dynamic Programming *continued*

Examples:

- Minimizing scalar multiplications in a chain of matrix multiplications
- Scheduling problems
- Longest common subsequence in two sequences

Dynamic Programming Steps

The development of a dynamic programming algorithm is characterized by four 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. Compute an optimal solution from the computed information.

The first three steps form the basis of dynamic programming, and the last is needed only if we want to report an optimal solution, not just return its value.

Example: Matrix-chain Multiplication

We will first discuss the dynamic programming method in terms of a chain of n matrices to be multiplied. For example, the following chain

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4$$

can be computed in the following ways:

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))) \text{ or}$$

$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4)) \text{ or}$$

$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4)) \text{ or}$$

$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4) \text{ or}$$

$$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$$

This parenthesization can have a dramatic impact on the cost of evaluating the product.

Example: Matrix-chain Multiplication

The main idea of this dynamic programming example is to answer the following question: Which parenthesization will give the best (the cheapest cost) i.e. How many basic operations at minimum will it take under the best parenthesization to multiply the sequence A_1, A_2, \dots, A_n .

Thus we have two questions to answer: (1) what does it cost to get the best parenthesization and (2) How to get it.

- Natural way: try all possible parenthesization
- So, for each parenthesize (division point), make a recursive call on the left side of the division point and another recursive call on the right side. Then compute the combined cost remembering (memoization) if this division point is the best so far (the cheapest).
- A recursion call is terminated if the problem is small enough and give the trivial answer (optimal solution).
- Return the best cost found

Example: Matrix-chain Multiplication

Note that: The idea in dynamic programming is different than quick sort (i.e. divide and conquer)

- In quick sort, we only look at the location of one pivot.
- In dynamic programming, we look at all of the pivot's possible locations to answer the question of what is the best pivot placement in the quick sort algorithm.

Note that, as we will see in this example

Dynamic programming = optimal division point + overlapping subproblems.

The above in red defines the Elements of Dynamic Programming

Elements of Dynamic Programming

- **Optimal Substructure:** The optimal solution is built from optimal solutions to subproblems. In the case of the matrix-chain algorithm, if $A_1 \cdot A_2 \cdot \dots \cdot A_k$ is a prefix subchain of an optimal parenthesization of $A_1 \cdot A_2 \cdot \dots \cdot A_n$, then $A_1 \cdot A_2 \cdot \dots \cdot A_k$ and $A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_n$ must be optimally parenthesized. (This allows using divide and conquer approach i.e. when independent smaller optimization problem of the same form contribute to the solution of the original problem)
- **Overlapping Subproblems:** The space of subproblems must be small (in the sense that the recursive algorithm solves many of the subproblems over and over again) for this method to be useful. The number of distinct problems should be polynomial. In the case of the matrix-chain algorithm, the number of subproblems as we will see, is $\Theta(n^2)$.

Example: Matrix-chain Multiplication

The general approach of Dynamic programming (Top down approach) can be summarized as follows:

- **Memoization:** Check if the subproblem was already solved (trivial solution)
 - For each parenthesize (choice or division point)
 - * Divide and Conquer: make a recursive call on the left side of the division point and another recursive call on the right side.
 - * Compute the combined cost checking if this division point (choice) was the best so far (the cheapest).
 - Return and remember the best cost (solution value) found and the choice that led to it.

Counting Scalar Multiplications in Matrix Multiplication

MATRIX-MULTIPLY(A, B)

1. **if** $columns[A] \neq rows[B]$
2. **then error** "incompatible dimensions"
3. **else for** $i \leftarrow 1$ to $rows[A]$
4. **do for** $j \leftarrow 1$ to $columns[B]$
5. **do** $C[i, j] \leftarrow 0$
6. **for** $k \leftarrow 1$ to $columns[A]$
7. **do** $C[i, j] = C[i, j] + A[i, k] \cdot B[k, j]$
8. **return** C

$$\boxed{n \times m} \cdot \boxed{m \times l} = \boxed{n \times l}$$

The number of scalar multiplications is: $n \times m \times l$

The Matrix-Chain Multiplication Problem

The matrix-chain multiplication problem: given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices, such that for $i = 1, 2, \dots, n$, matrix A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdot \dots \cdot A_n$ in a way to minimize the number of scalar multiplications.

Example: $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ such that the dimensions of A_1 is 10×20 , A_2 is 20×50 , A_3 is 50×1 , and A_4 is 1×100 .

$A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)) \rightarrow 125,000$ scalar multiplications

$(A_1 \cdot (A_2 \cdot A_3)) \cdot A_4 \rightarrow 2,200$ scalar multiplications

How to Choose the Best Parenthesization

Exhaustive checking will not provide an efficient algorithm because this would result in an exponential running time algorithm. To see this let's devise a recurrence:

Let $P(n)$ be the number of alternative parenthesizations for an n matrix chain. We can split the sequence between the k th and $(k+1)$ st matrix for any $k = 1, 2, \dots, (n-1)$, and then parenthesize the remaining subsequences, giving the recurrence:

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

Note that this recurrence has a closed form solution of $P(n) = C(n-1)$, where $C(n)$ is the n th Catalan number, and $C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega(\frac{4^n}{n^{3/2}})$.

1. Characterizing the Structure of an Optimal Parenthesization

To characterize the structure of an optimal solution for the matrix-chain multiplication problem, we must split the product $A_1 \cdot A_2 \cdot \dots \cdot A_n$ between A_k and A_{k+1} for some integer $1 \leq k < n$.

$A_{i..j}$ denotes the matrix resulting from multiplying $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$.

Using this notation, the cost of the optimal parenthesization is the cost of computing $A_{1..k}$ and $A_{(k+1)..n}$ along with the cost of multiplying the two matrices together.

Note that if $A_1 \cdot A_2 \cdot \dots \cdot A_k$ is a prefix subchain of an optimal parenthesization of $A_1 \cdot A_2 \cdot \dots \cdot A_n$, then $A_1 \cdot A_2 \cdot \dots \cdot A_k$ must be optimally parenthesized. If not, then we could provide a lower cost alternative. An optimal solution to the matrix-chain multiplication problem must contain optimal solutions to subproblems.

2. Defining the Value of an Optimal Solution Recursively

Let $m[i, j]$ denote the minimum number of scalar multiplications to compute $A_{i..j}$; hence, $m[1, n]$ is the minimum number to compute $A_{1..n}$. Note that $A_{i..i} = A_i$ requires no scalar multiplications; hence, $m[i, i] = 0$ for $i = 1, 2, \dots, n$.

Assuming that the optimal parenthesization splits the product $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ between k and $k + 1$ where $i \leq k < j$, then $m[i, j]$ is the minimum cost of computing the subproducts $A_{i..k}$ and $A_{(k+1)..j}$ plus the cost of multiplying them together, $p_{i-1}p_kp_j$:

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$$

Note that there are $j - 1 - i + 1 = j - i$ possible values of k to check for a given i and j . Below is the recursive definition for the minimum cost of parenthesizing the product $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$:

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

3. Compute the Optimal Solution Bottom-Up

MATRIX-CHAIN-ORDER is a bottom-up algorithm for computing the optimal solution, which takes a single argument $p = \langle p_0, p_1, \dots, p_n \rangle$, where the dimension of A_i is $p_{i-1} \times p_i$, for $i = 1, 2, \dots, n$.

It uses an auxiliary table $m[1..n, 1..n]$ to store the $m[i, j]$ costs and $s[1..n, 1..n]$ to record the index of k that achieves the optimal cost in computing $m[i, j]$.

MATRIX-CHAIN-ORDER(p)

1. $n \leftarrow \text{length}[p] - 1$
2. **for** $i \leftarrow 1$ to n
3. **do** $m[i, i] \leftarrow 0$

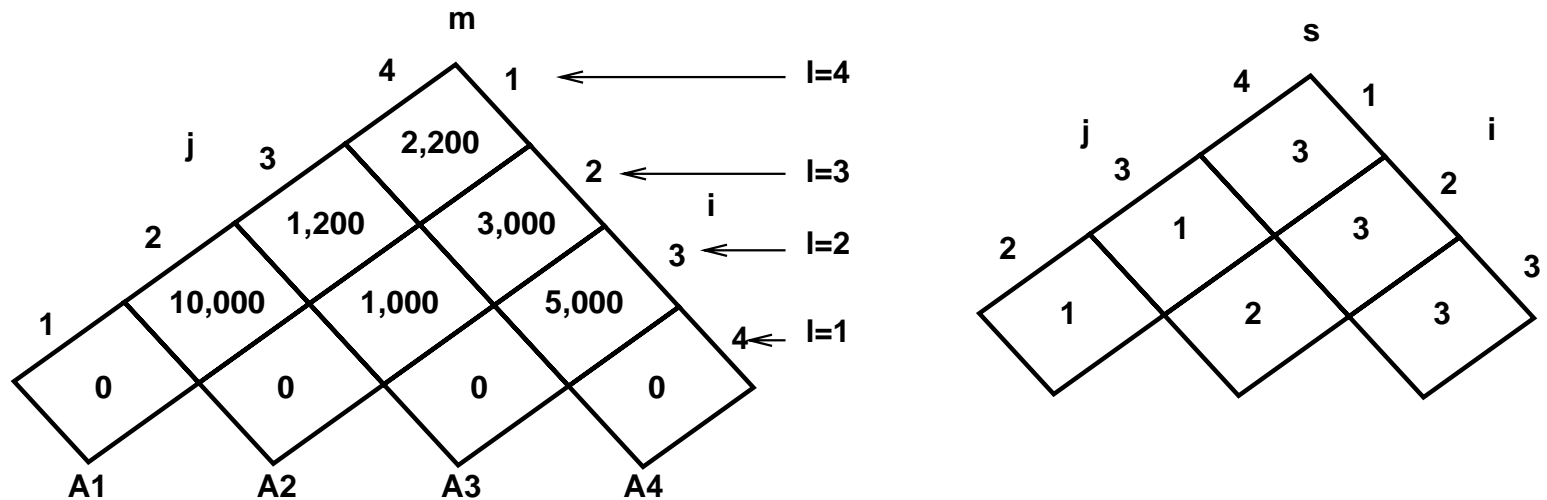
3. Compute the Optimal Solution Bottom-Up *continued*

```
4. for  $l \leftarrow 2$  to  $n$ 
5.   do for  $i \leftarrow 1$  to  $n - l + 1$ 
6.     do  $j \leftarrow i + l - 1$ 
7.        $m[i, j] \leftarrow \infty$ 
8.       for  $k \leftarrow i$  to  $j - 1$ 
9.         do  $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
10.        if  $q < m[i, j]$ 
11.          then  $m[i, j] \leftarrow q$ 
12.             $s[i, j] \leftarrow k$ 
13. return  $m$  and  $s$ 
```

At each step, the $m[i, j]$ computed in lines 9-12 depends only on the table entries $m[i, k]$ and $m[k + 1, j]$ already computed.

MATRIX-CHAIN-ORDER Example

Consider MATRIX-CHAIN-ORDER(p), where $p = \langle 10, 20, 50, 1, 100 \rangle$.



MATRIX-CHAIN-ORDER **Example** *continued*

The computations are provided below:

$$m[1,2] = \min_{1 \leq k < 2} \{m[1,1] + m[2,2] + p_0 p_1 p_2\} = 10,000$$

$$m[2,3] = \min_{2 \leq k < 3} \{m[2,2] + m[3,3] + p_1 p_2 p_3\} = 1,000$$

$$m[3,4] = \min_{3 \leq k < 4} \{m[3,3] + m[4,4] + p_2 p_3 p_4\} = 5,000$$

$$m[1,3] = \min_{1 \leq k < 3} \{m[1,1] + m[2,3] + p_0 p_1 p_3 = 1,200, \\ m[1,2] + m[3,3] + p_0 p_2 p_3 = 10,500\} = 1,200$$

$$m[2,4] = \min_{2 \leq k < 4} \{m[2,2] + m[3,4] + p_1 p_2 p_4 = 105,000, \\ m[2,3] + m[4,4] + p_1 p_3 p_4 = 3,000\} = 3,000$$

$$m[1,4] = \min_{1 \leq k < 4} \{m[1,1] + m[2,4] + p_0 p_1 p_4 = 23,000, \\ m[1,2] + m[3,4] + p_0 p_2 p_4 = 65,000, \\ m[1,3] + m[4,4] + p_0 p_3 p_4 = 2,200\} = 2,200$$

4. Constructing the Optimal Solution

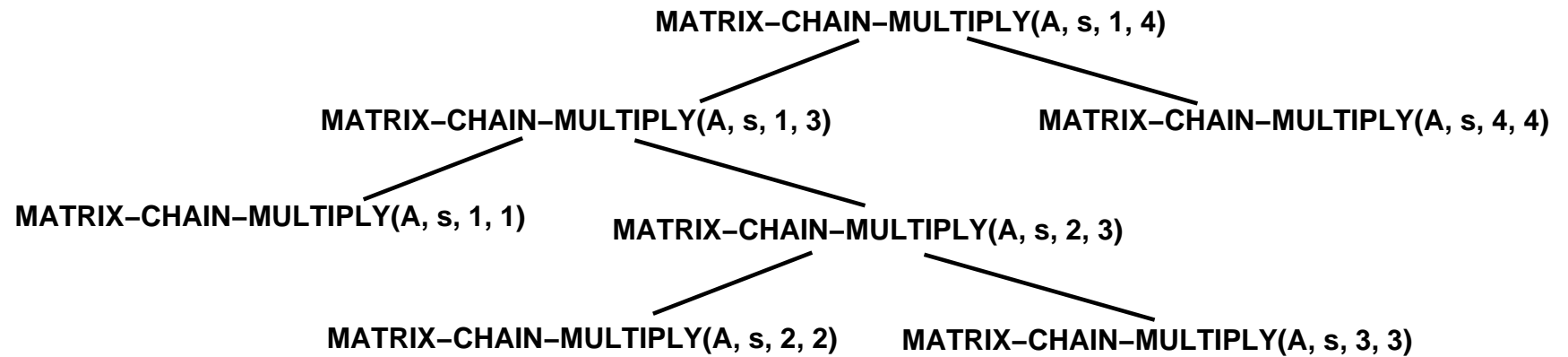
The following algorithm uses the table $s[1..n, 1..n]$ to determine the best way to multiply the matrices.

MATRIX-CHAIN-MULTIPLY(A, s, i, j)

1. **if** $j > i$
2. **then** $X \leftarrow \text{MATRIX-CHAIN-MULTIPLY}(A, s, i, s[i, j])$
3. $Y \leftarrow \text{MATRIX-CHAIN-MULTIPLY}(A, s, s[i, j] + 1, j)$
4. **return** MATRIX-MULTIPLY(X, Y)
5. **else return** A_i

For our example, this computes the matrix multiplication as $((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$.

Constructing the Optimal Solution for the Example



Complexity of MATRIX-CHAIN-ORDER

The complexity of the algorithm for $2 \leq l < n$ is computed as follows:

$$T(n) = \sum_{l=2}^n \sum_{i=1}^{n-l+1} \sum_{k=i}^{j-1} 1 \quad (1)$$

$$= \sum_{l=2}^n \sum_{i=1}^{n-l+1} (j-i) \quad (2)$$

$$= \sum_{l=2}^n \sum_{i=1}^{n-l+1} (i+l-1-i) \quad (3)$$

$$= \sum_{l=2}^n \sum_{i=1}^{n-l+1} (l-1) \quad (4)$$

$$= \sum_{l=2}^n (l-1)(n-l+1) \quad (5)$$

$$= \sum_{l=1}^{n-1} (n-l) \cdot l = n \sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2 \quad (6)$$

$$= \frac{n^2(n-1)}{2} - \frac{n(n-1)(2n-1)}{6} \quad (7)$$

$$= \frac{n^3 - n}{6} = O(n^3) \quad (8)$$