Matrix-Chain Multiplication

If the chain of matrices is (A1, A2, A3, A4), then we can fully parenthesize the product A1A2A3A4 in five distinct ways:

(A1(A2(A3A4)));

(A1((A2A3)A4));

((A1A2)(A3A4));

((A1(A2A3))A4);

(((A1A2)A3)A4);

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B.

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain (A1, A2, A3) of three matrices. Suppose that the dimensions of the matrices are 10 x 100, 100 x 5, and 5 x 50, respectively. If we multiply according to the parenthesization ((A1A2)A3), we perform 10 x 100 x 5 = 5000 scalar multiplications to compute the 10 x 5 matrix product A1A2, plus another 10 x 5 x 50 = 2500 scalar multiplications to multiply this matrix by A3, for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization (A1(A2A3)), we perform 100 x 5 x 50 = 25,000 scalar multiplications to compute the 100 x 50 matrix product A2A3, plus another 10 x 100 x 50 = 50,000 scalar multiplications to multiply A1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the ***matrix-chain multiplication problem*** as follows: given a chain (A1,A2, An) of n matrices, where for i = 1,2,.....,n, matrix A_i has dimension p_{i-1} x pi, fully parenthesize the product A1A2.....An in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

Applying dynamic programming

We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain. In so doing, we shall follow the four-step sequence that we stated at the beginning of this chapter:

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
4. Construct an optimal solution from computed information.

Step 1: The structure of an optimal parenthesization

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_iA_{i+1......}$ Aj, we split the product between A_k and A_{k+1}. Then the way we parenthesize the "prefix" subchain $A_iA_{i+1}.......A_k$ within this optimal parenthesization of $A_iA_{i+1}.........$Aj must be an optimal parenthesization of $A_iA_{i+1}.......$Ak. Why? If there were a less costly way to parenthesize $A_iA_{i+1}......$Ak, then we could substitute that parenthesization in the optimal parenthesization of $A_iA_{i+1}.........A_j$ to produce another way to parenthesize $A_iA_{i+1.......}$Aj whose cost was lower than the optimum: a contradiction. A similar observation holds for how we parenthesize the subchain $A_{k+1}A_{k+2}.........$Aj in the optimal parenthesization of $A_iA_{i+1}......$Aj: it must be an optimal parenthesization of $A_{k+1}A_{k+2}........$Aj.

Step 2: A recursive solution

For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing A_iA_{i+1} ....Aj for $1\leq i \leq j \leq n$. Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i.......j}$; for the full problem, the lowest cost way to compute $A_{1...n}$ would thus be m[1, n].

We can define m[i, j] recursively as follows. If i = j, the problem is trivial; the chain consists of just one matrix $A_{i....i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, m[i, i] = 0 for i = 1,2,.......,n. To compute m[i, j] when i < j, we take advantage of the structure of an optimal solution from step 1. Let us assume that to optimally parenthesize, we split the product $A_iA_{i+1}......A_j$ between A_k and A_{k+1}, where $i \leq k < j$. Then, m[i, j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1........j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is p_{i-1} x p_i, we see that computing the matrix product $A_{i....k}A_{k+1.....j}$ takes $p_{i-1}p_kp_j$ scalar multiplications. Thus, we obtain

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

Thus, our recursive definition for the minimum cost of parenthesizing the product $A_iA_{i+1 \ldots\ldots\ldots\ldots}A_j$ becomes

$$\mathbf{m[i, j]} = \begin{cases} \mathbf{0} & \textbf{if } \mathbf{i = j,} \\ \mathbf{\min_{i \le k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\}} & \textbf{if } \mathbf{i < j.} \end{cases} \quad \mathbf{(1)}$$

The m[i, j] values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define s[i, j] to be a value of k at which we split the product $A_iA_{i+1 \ldots\ldots}A_j$ in an optimal parenthesization. That is, s[i, j] equals a value k such that

$$\mathbf{m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_{j.}}$$

Step 3: Computing the optimal costs

At this point, we could easily write a recursive algorithm based on recurrence (1) to compute the minimum cost m[1, n] for multiplying A1A2......An.

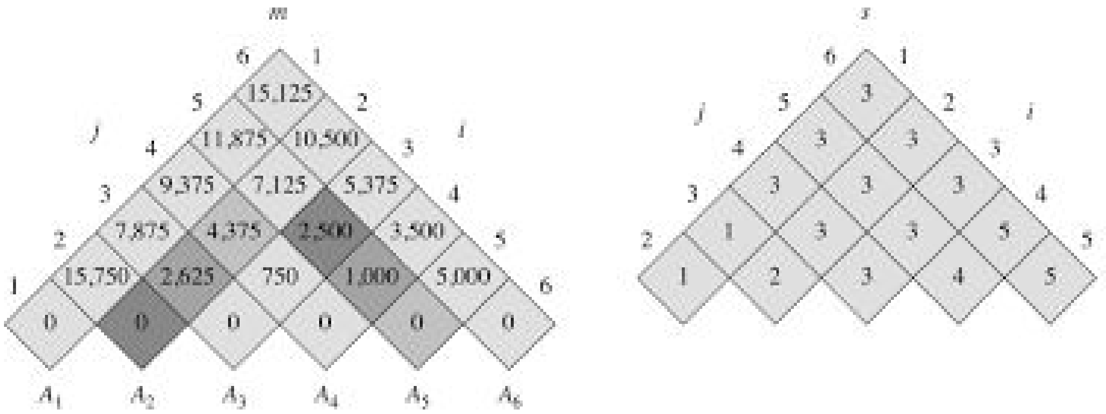
Instead of computing the solution to recurrence (1) recursively, we compute the optimal cost by using a tabular, bottom-up approach in the following

```
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
}
```

Problem

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is 30, 35, 15, 5, 10, 20, 25

Solution



p0=30, p1=35,p2=15, p3=5, p4=10, p5=20, p6=25

m[1,2]=min(m[1,1] + m[2,2] + p0xp1xp2)

=min(0+0+30x35x15)

=15750

similarly solve m[2,3], m[3,4], m[4,5], m[5,6]

m[1,3]=min((m[1,1]+m[2,3]+p0xp1xp3),(m[1,2]+m[3,3]+p0xp2xp3))

=min((0+2625+30x35x5),(15750+0+30x15x5))

=min(7875,18000)

=7875

similarly solve m[2,4],m[3,5],m[4,6]

m[1,4]=min((m[1,1]+m[2,4]+p0xp1xp4),(m[1,2]+m[3,4]+p0xp2xp4),(m[1,3]+m[4,4]+p0xp3xp4))

=min((0+4375+30x35x10),(15750+750+30x15x10),(7875+0+30x5x10))

=min(14875,21000,9375)

=9375

similarly solve m[2,5],m[3,6],m[1,5],m[2,6],m[1,6]

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show

how to multiply the matrices. The table s[1.....n – 1, 2....n] gives us the information

we need to do so. Each entry s[i, j] records a value of k such that an optimal parenthesization of $A_iA_{i+1.......}A_j$ splits the product between A_k and A_{k+1}. Thus, we know

that the final matrix multiplication in computing $A_{1.....n}$ optimally is

A1...s[1,n]As[1,n]+1....n. The following recursive procedure prints an optimal parenthesization of (A_i,........A_{i+1},.......,A_j) given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j. The initial call PRINT-OPTIMAL-PARENS(s, 1, n) prints an optimal parenthesization of (A_1,A_2,.......,A_n).

```
PRINT-OPTIMAL-PARENS(s, i, j)
    if i == j
        print "A"i;
    else print "("
        PRINT-OPTIMAL-PARENS.s; i; sOEi; j /
        PRINT-OPTIMAL-PARENS.s; sOEi; j  C 1; j /
        print ")"
```

In the above problem the call PRINT-OPTIMAL-PARENS(s, 1, 6) prints

the parenthesization ((A1(A2A3))((A4A5)A6)).

Time Complexity

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm.