Chained Matrix Multiplication

Agenda:

- Discussion of a dynamic programming algorithm
- An example illustrating the working of the algorithm, and
- A computer program in C++ that implements the algorithm.

Objective is to determine the minimum cost order of multiplying a string of n matrices, $M_1, M_2, \ldots, M_n$ where matrix $M_i$ has $r_{i-1}$ rows and $r_i$ columns, $1 \leq i \leq n$. Associativity ensures that $M_1 \times M_2 \times \ldots \times M_n$ itself is well-defined. Fortunately, the principle of optimality applies, i.e., if the best way of multiplying all the matrices requires us to make the first cut between the $i^{th}$ matrix and the $(i+1)^{st}$ matrix of the product, then the sub-products $M_i \times \ldots \times M_j$ and $M_{i+1} \times \ldots \times M_n$ themselves must be calculated in an optimal way. The least number of scalar multiplications to compute $M_i \times \ldots \times M_j$ is given by

$$m_{i,j} = \begin{cases} 0, & \text{if } i = j \\ \min \left\{ (m_{i,k} + m_{k+1,j} + r_{i-1} * r_k * r_j) : i \leq k \leq j-1 \right\}, & \text{if } i < j. \end{cases}$$

A dynamic programming algorithm

Input: Number $n$ of matrices, and $r_0, \ldots, r_n$, where $r_{i-1}$ and $r_i$ are dimensions of matrix $M_i$.

Output: The minimum cost of multiplying the $M_i$’s, assuming that $pqr$ operations are required to multiply a $p \times q$ matrix and a $q \times r$ matrix.

Method:

1. for $i \leftarrow 1$ to $n$ do
2. $m_{i,i} \leftarrow 0$;
3. for $p \leftarrow 1$ to $n-1$ do
4. for $i \leftarrow 1$ to $n-p$ do
5. $j \leftarrow i + p$;
6. $m_{i,j} \leftarrow \min \left\{ (m_{i,k} + m_{k+1,j} + r_{i-1} * r_k * r_j) : i \leq k \leq j-1 \right\}$;
7. write $m_{1,n}$;
Note that \((m_{ij})\) itself may be viewed as an upper triangular matrix that is constructed diagonal-by-diagonal in a bottom-up fashion. For \(i < j\), computation of \(m_{ij}\) makes effective use of the solution of the sub-problems stored in \(m_{pq}\) where \(i \leq p \leq q \leq j\) and \(0 \leq |p-q| < |i-j|\).

**Analysis of the algorithm:** The “for” loop at Step (1) includes a single assignment statement, and hence accounts for a total of \(n\) elementary operations. The major work is done within the nested “for” loops from steps (3) through (6). For \(p \geq 1\), there are \(n - p\) iterations of the inner loop, and each such iteration computes a single \(m_{ij}\) at Step 6. The \(\text{MIN}\) function acts on a set of \(j-i\) elements, and hence involves \((j - i - 1)\) comparisons. Thus the inner loop accounts for \(j - i = p\) elementary operations. The execution time of the algorithm is, therefore, in the order of

\[
\sum_{p=1}^{n-1}(n-p)p = \left(\sum_{p=1}^{n-1}p\right) - \left(\sum_{p=1}^{n-1}p^2\right) = \frac{n^2(n-1)}{2} - \frac{n(n-1)(2n-1)}{6} = \frac{n^3 - n}{6} \text{ that is } O(n^3) .
\]

We usually want to know not just the number of scalar multiplications necessary to compute the product, but also how to perform this computation efficiently. We do this by adding a second (upper triangular) array, say \(\text{best}_k\), to keep track of the choices we have made. For \(i < j\), we save in \(\text{best}_k[i, j]\) the value of \(k\) that corresponds to the minimum term among \(j-i\) elements compared at Step (6) of the algorithm. When the algorithm terminates, \(\text{best}_k[1, n]\) tells us where to make the first cut in the product. Proceeding recursively on the two terms thus produced, we can construct the optimal ways of parenthesizing \(M_1 \times M_2 \times \ldots \times M_n\).

The present algorithm is due to Godbole [1]. There exist superior algorithms [2, 3] with running time of only \(O(n \log n)\), and a parallel algorithm [4].


Example: Consider the following chain of matrices:

\[
M_1 \quad M_2 \quad M_3 \quad M_4 \quad M_5 \quad M_6 \\
5 \times 2 \quad 2 \times 3 \quad 3 \times 4 \quad 4 \times 6 \quad 6 \times 7 \quad 7 \times 8
\]

Objectives are (i) to perform the dynamic programming algorithm to determine the least number of scalar multiplications necessary to obtain \(M_1 \times M_2 \times M_3 \times M_4 \times M_5 \times M_6\), and (ii) to determine the corresponding parenthesization.

In this case, \(r_0 = 5; r_1 = 2; r_2 = 3; r_3 = 4; r_4 = 6; r_5 = 7;\) and \(r_6 = 8\). While computing \(m_{i,j}\), record the value of \(k\) corresponding to \(\min \{m_{i,k} + m_{k+1,j} + r_i \cdot r_k \cdot r_j : i \leq k \leq j - 1\}\) in \(\text{best}_k\).

![Matrix Multiplication Table]

Build the structure brick-by-brick.

The parenthesization follows.

\[
\begin{array}{c}
(M_1 \times M_2 \times M_3 \times M_4 \times M_5 \times M_6) \\
\text{Reason}
\end{array}
\]

\[
\begin{array}{c}
(M_1 \times (M_2 \times M_3 \times M_4 \times M_5 \times M_6)) \\
\text{best}_k \text{ for } m_{1,6} \text{ is equal to 1.}
\end{array}
\]

\[
\begin{array}{c}
(M_1 \times ((M_2 \times M_3 \times M_4 \times M_5) \times M_6)) \\
\text{best}_k \text{ for } m_{2,6} \text{ is equal to 5.}
\end{array}
\]

\[
\begin{array}{c}
(M_1 \times (((M_2 \times M_3 \times M_4) \times M_5) \times M_6)) \\
\text{best}_k \text{ for } m_{2,5} \text{ is equal to 4.}
\end{array}
\]

\[
\begin{array}{c}
(M_1 \times ((((M_2 \times M_3) \times M_4) \times M_5) \times M_6)) \\
\text{best}_k \text{ for } m_{2,4} \text{ is equal to 3.}
\end{array}
\]
Notice that a particular parenthesization is very naturally representable by a rooted binary tree in which individual matrices correspond to leaves, root is the final product and intermediate nodes are intermediate products. For example, the preceding parenthesization admits the following binary tree representation:

\[
\begin{array}{c}
  \text{M}_1 \\
  \text{M}_2 & \text{M}_3 \\
  \text{M}_4 & \text{M}_5 & \text{M}_6 \\
  \end{array}
\]

A program in C++ appears on the next page.
```cpp
#include <iostream>
using namespace std;

class ChainedMatrixMultiplication {
public:
    ChainedMatrixMultiplication(int n) : n_(n) {
        // Initialize cost matrix and best_k matrix
        for (int i = 1; i <= n; i++)
            cost_[i][i] = 0;
        for (int p = 1; p <= n-1; p++)
            for (int i = 1; i <= n-p; i++)
                for (int k = i+1; k <= j-1; k++)
                    if (temp < min)
                        min = temp;
            
        // Compute the best parenthesization
        string best_parenthesis;
        traceback(1, n, &best_parenthesis);
        cout << best_parenthesis << endl;
    }

private:
    int n_; // Number of matrices in the chain
    int cost_[][n+1]; // Cost matrix
    int best_k_[][n+1]; // Best k matrix

    void traceback(int i, int j, string& parenthesis) {
        if (i == j)
            parenthesis += 'M_' + to_string(i);
        else
            for (int k = i+1; k <= j-1; k++)
                if (temp < min)
                    min = temp;
            
        // Compute the best parenthesization
        string best_parenthesis;
        traceback(1, n, &best_parenthesis);
        cout << best_parenthesis << endl;
    }

    void printMatrix() {
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                cout << cost_[i][j] << '	';
        cout << endl;
    }
};
```

The cost matrix and the best_k matrix themselves may be printed by appropriately introducing the following piece of code:

```cpp
// Print the cost matrix
for (j = 1; j <= n; j++)
    i = 1;
    for (k = 1; k <= j; k++)
        cout << em [i][n+i-j] << " ";
        i = i + 1;
    } // for (k = ...)
    cout << endl << endl;
} // for (j = ...)

// Print the best_k matrix
for (j = 1; j <= n-1; j++)
    i = 1;
    for (k = 1; k <= j; k++)
        cout << best_k [i][n+i-j] << " ";
        i = i + 1;
    } // for (k = ...)
    cout << endl << endl;
} // for (j = ...)
```