CMSC 451: Lecture 12
Dynamic Programming: Chain Matrix Multiplication

Thursday, Oct 12, 2017

Reading: Section 6.5 in DPV; not covered in KT.

Chain matrix multiplication: This problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in compiler design for code optimization and in databases for query optimization. We will study the problem in a very restricted instance, where the dynamic programming issues are easiest to see.

Suppose that we wish to multiply a series of matrices

$$C = A_1 \cdot A_2 \cdots A_n$$

Matrix multiplication is an associative but not a commutative operation. This means that we are free to parenthesize the above multiplication however we like, but we are not free to rearrange the order of the matrices. Also recall that when two (nonsquare) matrices are being multiplied, there are restrictions on the dimensions. A $p \times q$ matrix has p rows and q columns. You can multiply a $p \times q$ matrix A times a $q \times r$ matrix B, and the result will be a $p \times r$ matrix C (see Fig. 1). The number of columns of A must equal the number of rows of B. In particular for $1 \le i \le p$ and $1 \le j \le r$, we have

$$C[i, j] = \sum_{k=1}^{q} A[i, k] \cdot B[k, j].$$

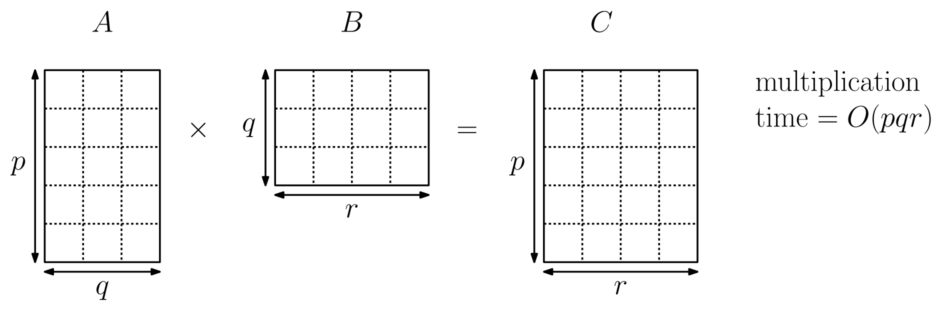

Fig. 1: Matrix Multiplication.

This corresponds to the (hopefully familiar) rule that the $[i, j]$ entry of C is the dot product of the ith (horizontal) row of A and the jth (vertical) column of B. Observe that there are pr total entries in C and each takes $O(q)$ time to compute, thus the total time to multiply these two matrices is proportional to the product of the dimensions, pqr.

Note that although any legal "parenthesization" will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices: A_1 be 5×4, A_2 be 4×6 and

A_3 be 6×2.

$$\begin{aligned}\text{cost}[((A_1A_2)A_3)] &= (5\cdot 4\cdot 6) + (5\cdot 6\cdot 2) = 180,\\ \text{cost}[(A_1(A_2A_3))] &= (4\cdot 6\cdot 2) + (5\cdot 4\cdot 2) = 88.\end{aligned}$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

Chain Matrix Multiplication Problem: Given a sequence of matrices $A_1, \ldots, A_n$ and dimensions $p_0, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication (represented, say, as a binary tree) that minimizes the number of operations.

Important Note: This algorithm *does not* perform the multiplications, it just determines the best order in which to perform the multiplications and the total number of operations.

Dynamic programming approach: A naive approach to this problem, namely that of trying all valid ways of parenthesizing the expression, will lead to an exponential running time. We will solve it through dynamic programming.

This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into subproblems, whose solutions can be combined to solve the global problem. As is common to any DP solution, we need to find some way to break the problem into smaller subproblems, and we need to determine a recursive formulation, which represents the optimum solution to each problem in terms of solutions to the subproblems. Let us think of how we can do this.

Since matrices cannot be reordered, it makes sense to think about sequences of matrices. Let $A_{i..j}$ denote the result of multiplying matrices i through j. It is easy to see that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix. (Think about this for a second to be sure you see why.) Now, in order to determine how to perform this multiplication optimally, we need to make many decisions. What we want to do is to break the problem into problems of a similar structure. In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is,

$$A_{1..n} = A_{1..k} \cdot A_{k+1..n} \qquad \text{for } 1 \le k \le n-1.$$

Thus the problem of determining the optimal sequence reduces to two decisions:

- What is the best place to split the chain? (what is k?)
- How do we parenthesize each of the subsequences $A_{1..k}$ and $A_{k+1..n}$?

Clearly, the problems of computing the two subsequences can be solved recursively, by applying the same scheme. (This is an instance of the principle of optimality. There is no advantage to be gained by solving a subproblem suboptimally.) So, let us think about the problem of determining the best value of k. At this point, you may be tempted to consider some clever ideas. For example, since we want matrices with small dimensions, pick the value of k that minimizes p_k. Although this is not a bad idea, in principle. (After all it might work. It just turns out that it doesn't in this case. This takes a bit of thinking, which you should try.)

Instead, as is the case in the other dynamic programming solutions we have seen, we will try *all possible* choices of k and take the best of them. This is not as inefficient as it might sound, since there are only $O(n^2)$ different sequences of matrices. (There are $\binom{n}{2} = n(n-1)/2$ ways of choosing i and j to form $A_{i..j}$ to be precise.) Thus, we do not encounter exponential growth in our algorithm's complexity, only polynomial growth.

Notice that our chain matrix multiplication problem satisfies the principle of optimality. In particular, once we decide to break the sequence into the product $A_{1..k} \cdot A_{k+1..n}$, it is in our best interest to compute each subsequence optimally. That is, for the global problem to be solved optimally, the subproblems should be solved optimally as well.

Recursive formulation: Let's explore how to express the optimum cost of multiplication in a recursive form. Later we will consider how to efficiently implement this recursive rule. We will subdivide the problem into subproblems by considering subsequences of matrices. In particular, for $1 \leq i \leq j \leq n$, let $m(i,j)$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The desired total cost of multiplying all the matrices is that of computing the entire chain $A_{1..n}$, which is given by $m(1,n)$. The optimum cost can be described by the following recursive formulation.

Basis: Observe that if $i = j$ then the sequence contains only one matrix, and so the cost is 0. (There is nothing to multiply.) Thus, $m(i,i) = 0$.

Step: If $i < j$, then we are asking about the product $A_{i..j}$. This can be split into two groups $A_{i..k}$ times $A_{k+1..j}$, by considering each k, $i \leq k < j$ (see Fig. 2).

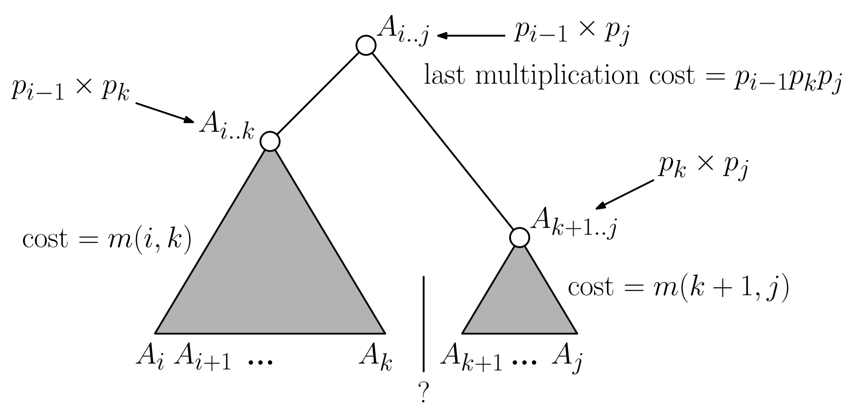

Fig. 2: Dynamic programming decision.

The optimum times to compute $A_{i..k}$ and $A_{k+1..j}$ are, by definition, $m(i,k)$ and $m(k+1,j)$, respectively. Let us assume inductively that we can compute these values. (Note that they each involve a strictly smaller number of matrices, so there is no possibility of circularity.) Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix, and $A_{k+1..j}$ is a $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1}p_kp_j$. This suggests the following recursive rule for $m(i,j)$.

$$\begin{aligned} m(i,i) &= 0 \\ m(i,j) &= \min_{i \leq k < j} (m(i,k) + m(k+1,j) + p_{i-1}p_kp_j) \qquad \text{for } i < j. \end{aligned}$$

Bottom-up implementation: As with other DP problems, there are two natural implementations of the recursive rule that will lead to an efficient algorithm. One is memoization (which we will leave as an exercise), and the other is bottom-up calculation. We will consider just the latter.

To do this, we will store the values of $m(i, j)$ in a 2-dimensional array $m[1..n, 1..n]$. The trickiest part of the process is arranging the order in which to compute the values. In the process of computing $m(i, j)$ we need to access values $m(i, k)$ and $m(k + 1, j)$ for k lying between i and j. Note that we cannot just compute the matrix in the simple row-by-row order that we used for the longest common subsequence problem. To see why, suppose that we are computing the values in row 3. When computing $m[3, 5]$, we would need to access both $m[3, 4]$ and $m[4, 5]$, but $m[4, 5]$ is in row 4, which has not yet been computed.

Instead, the trick is to compute *diagonal-by-diagonal* working out from the middle of the array. In particular, we organize our computation according to the number of matrices in the subsequence. For example, $m[3, 5]$ represents a chain of $5 - 3 + 1 = 3$ matrices, whereas $m[3, 4]$ and $m[4, 5]$ each represent chains of only two matrices. We first solve the problem for chains of length 1 (which is trivial), then chains of length 2, and so on, until we come to $m[1, n]$, which is the total chain of length n.

To do this, for $1 \leq i \leq j \leq n$, let $L = j - i + 1$ denote the length of the subchain being multiplied. How shall we set up the loops to do this? The case $L = 1$ is trivial, since there is only one matrix, and nothing needs to be multiplied, so we have $m[i, i] = 0$. Otherwise, our outer loop runs from $L = 2, \ldots, n$. If a subchain of length L starts at position i, then $j = i + L - 1$. Since $j \leq n$, we have $i + L - 1 \leq n$, or in other words, $i \leq n - L + 1$. So our inner loop will be based on i running from 1 up to $n - L + 1$. The code is presented in the code block below. (Also, see Fig. 3 for an example.) We will explain below the purpose of the s array.

Chain Matrix Multiplication

```
Matrix-Chain(p[0..n]) {
    s = array[1..n-1, 2..n]
    for (i = 1 to n) m[i, i] = 0                // initialize
    for (L = 2 to n) {                          // L = length of subchain
        for (i = 1 to n - L + 1) {
            j = i + L - 1
            m[i,j] = INFINITY
            for (k = i to j - 1) {              // check all splits
                cost = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]
                if (cost < m[i, j]) {           // found a new optimum?
                    m[i, j] = cost              // ...save its cost
                    s[i, j] = k                 // ...and the split marker
                }
            }
        }
    }
    return m[1, n] (final cost) and s (splitting markers)
}
```

The array $s[i,j]$ will be explained below. It will be used to extract the actual multiplication sequence. The running time of the procedure is $O(n^3)$. This is because we have three nested loops, and each can iterate at most n times. (A more careful analysis would show that the total number of iterations grows roughly as $n^3/6$.)

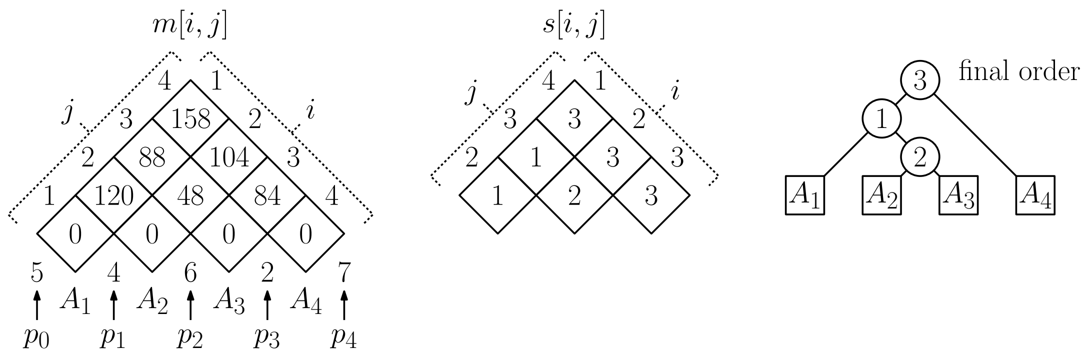

Fig. 3: Chain matrix multiplication for the product $A_1 \cdots A_4$, where A_i is of dimension $p_{i-1} \times p_i$.

Extracting the final Sequence: Extracting the actual multiplication sequence is a fairly easy extension. The basic idea is to leave a *split marker* indicating what the best split is, that is, the value of k that leads to the minimum value of $m[i,j]$. We can maintain a parallel array $s[i,j]$ in which we will store the value of k providing the optimal split. For example, suppose that $s[i,j] = k$. This tells us that the best way to multiply the subchain $A_{i..j}$ is to first multiply the subchain $A_{i..k}$ and then multiply the subchain $A_{k+1..j}$, and finally multiply these together. Intuitively, $s[i,j]$ tells us what multiplication to perform *last*. Note that we only need to store $s[i,j]$ when we have at least two matrices, that is, if $j > i$.

The actual multiplication algorithm uses the $s[i,j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1..n]$, and that $s[i,j]$ is global to this recursive procedure. The recursive procedure do-mult does this computation and below returns a matrix (see Fig. 3).

Extracting Optimum Sequence

```
do-mult(i, j) {
    if (i == j)                         // basis case
        return A[i]
    else {
        k = s[i,j]
        X = do-mult(i, k)               // X = A[i]...A[k]
        Y = do-mult(k+1, j)             // Y = A[k+1]...A[j]
        return X * Y                    // multiply matrices X and Y
    }
}
```

It's a good idea to trace through this example to be sure you understand it.