ICS 260—Fall, 2000–Class Notes 8: Dynamic Programming

Notes Originally Developed by Sandy Irani
Modified by Michael B. Dillencourt

This material is taken from [CLR 16.1-16.3]

1 Introduction

In a dynamic programming solution, one seeks the optimal solution to a problem by breaking the problem into subproblems and combining the solutions. This can be also said of divide-and-conquer strategies. However, dynamic programming operates in a bottom-up manner instead of a top-down manner. This is especially useful when subproblems overlap. In a dynamic programming algorithm, solutions to small subproblems are computed and stored in a table. These solutions are then combined to yield solutions to larger and larger subproblems. Since all intermediate results are stored in a table, it is not necessary to re-calculate solutions to subproblems that are used over again.

2 Example: matrix chain multiplication

Suppose we are planning to compute the product of n matrices $A_1, A_2, \ldots, A_n$. We must decide in what order to multiply the matrices. Such an ordering can be specified by *fully parenthesizing* the product of matrices so that each product is computed on two fully parenthesized products of matrices which in turn a surrounded by parentheses. For example, the product $A_1 \cdot A_2 \cdot A_3$, can be parenthesized as $(A_1 \cdot (A_2 \cdot A_3))$ or $((A_1 \cdot A_2) \cdot A_3)$. The result of performing the product in the two different orders is the same, but the time spent in performing the products can differ dramatically.

Note that in order to multiply two matrices $A \cdot B$, it must be the case that the number of columns of A must be equal to the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, then the number of scalar multiplications required to multiply A and B is pqr. We will count the cost of performing a product of matrices as the number of scalar multiplications required since this dominates the running time of the computation. Each row of A can be seen as a vector of length q and each column of B can be seen as a vector of length q. The result of multiplying $A \cdot B$ is a $p \times r$ matrix, where the entry in row i column j is the dot product of row i of A and column j of B. (The dot product of two equal length vectors $\langle a_1, \ldots, a_q \rangle$ and $\langle b_1, \ldots, b_n \rangle$ is

$$a_1 \cdot b_1 + a_2 \cdot b_2 + \cdots + a_n \cdot b_n.$$

The order in which a product of matrices is computed can effect the number of scalar multiplication operations performed.

The *matrix chain multiplication problem* is to fully parenthesize a matrix product $A_1 \cdot A_2 \cdots A_n$ so that the number of scalar multiplications performed is minimized. We will denote the size of matrix A_i by $p_{i-1} \times p_i$.

- Exhaustively checking all possibilities.

The first step in developing a dynamic programming algorithm to solve a problem is to break the problem into subproblems. The right way to break up this particular problem is to find the last product which is performed. That is, we can split the sequence of matrices between matrix A_k and A_{k+1}. We solve $A_1 \cdots A_k$ optimally and then solve $A_{k+1} \cdots A_n$ optimally, and then multiply the two resulting products. The things that must be determined, is the best choice for k and then the optimal way to compute $A_1 \cdots A_k$ and $A_{k+1} \cdots A_n$ Note that one property of problems for which dynamic programming solutions work is that the optimal solution to the whole problem is the combination of the optimal solutions to subproblems.

First some notation: for $i \leq j$, let $A_{i:j}$ denote the product for $A_i \cdot A_{i+1} \cdots A_j$. Let $m[i,j]$ denote the minimum number of scalar multiplications required to compute $A_{i:j}$. We are ultimately trying to solve $m[1,n]$. Now if $i = j$, then there is only one matrix in the product and no multiplications are required, so $m[i,i] = 0$ for all $1 \leq i \leq n$. To compute $m[i,j]$, we will use the same reasoning that the last multiplication performed in the optimal solution for $A_{i:j}$ must be between $A_{i:k}$ and $A_{k+1:j}$ for some $k \in \{i, i+1, \ldots, j-1\}$. We must determine the optimal k and the optimal way to compute $A_{i:k}$ and $A_{k+1:j}$. This leads to the following recursion:

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

To help us keep track of the actual parenthesization, we will keep another matrix $s[i,j]$ which will denote the k which achieves the minimum in the recurrence above.

It may be tempting at this point, to implement this using a recursive procedure as shown in Figure 2.

```
Procedure RecursiveMatrixChainMultiplication(p_{i-1}, p_i, ..., p_j):
if i == j
    return 0
min ← ∞
for k = i to j − 1
    m[i,k] ← RecursiveMatrixChainMultiplication(p_{i-1}, ..., p_k)
    m[k+1,j] ← RecursiveMatrixChainMultiplication(p_{k+1}, ..., p_j)
    if min > m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
        min ← m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
m[i,j] ← min
return min
```

Figure 1: A recursive algorithm for the matrix chain multiplication problem.

The main procedure would be called on RecursiveMatrixChainMultiplication$(p_0, p_1, \ldots, p_n)$ Let's look at what this procedure does on a small example.

- example

In general, the running time of the above algorithm on an input of size n would be $T(n)$, the solution to the following recurrence: $T(1) \geq 1$.

$$T(n) \geq 1 + \sum_{k=1}^{n-1} [T(k) + T(n-k) + 1].$$

It's not too hard to prove by induction that $T(n) \geq 2^{n-1}$.

Using dynamic programming, each subproblem would be solved only once and stored. The intermediate values are stored in an array $m[i, j]$. The array is filled in for every pair $1 \leq i < j \leq n$. There are $\binom{n}{2}$ such pairs, so the table has size $\Theta(n^2)$. Similarly, we will keeps a table $s[i, j]$ which indicates the optimal place to split the sequence $A_i, A_{i+1}, \ldots, A_j$.

```
Procedure DynProgMatrixChainMultiplication(p_0, p_2, ..., p_n):
for i = 1 to n
    m[i, i] ← 0
for l ← 2 to n
    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
```

Figure 2: A dynamic programming algorithm for the matrix chain multiplication problem.
There are three nested loops in the procedure, so the whole procedure takes time $\Theta(n^3)$.

- Example.
- Constructing an optimal solution.

There are two properties of a problem which allow for an efficient dynamic programming solution:

- **Optimal substructure:** The optimal solution to the problem contains within it, optimal solutions to subproblems.
- **Overlapping subproblems:** the number of subproblems must be small (polynomial), so that a recursive procedure would solve the same subproblems repeatedly.