Lecture Notes for CSCI 311: Algorithms
Set 9-Dynamic Programming Example: Matrix Chain Multiplication

Professor Talmage

October 7, 2024

---

1 Setup

We will now return to dynamic programming examples, working through several more complex problems. Through each, try to keep in mind the overall principles and solution outline for dynamic programming, and think about how it changes for specific problems, or might change for others you may see in the future.

The first problem we will consider focuses on choosing one among many orders. Suppose we want to multiply a sequence of several matrices (For example, we may be combining multiple image filters or effects into a single operation, which we often want to do so we can apply the same filter repeatedly, such as to every frame of a video. Multiplying many large matrices is also a core step in machine learning and big data processing.) First, we note that matrix multiplication is associate. That means that the order in which we do individual multiplications does not affect the final product value.

- Example: Suppose we have five matrices A, B, C, D, E and we want to compute the product $A \cdot B \cdot C \cdot D \cdot E$. Then the following are all equal ways to compute that product:

$$(((AB)C)D)E = (A(BC))(DE) = ((AB)C)(DE) = \cdots$$

- We call each of these expressions *fully-parenthesized*, meaning that there are is at most one multiplication in any pair of parentheses (counting sub-parenthesized expressions as a single matrix since they will be evaluated first).

We care about the associativity of matrix multiplication because, while the result matrix will be the same regardless of the order in which we compute the individual products, the runtime will vary widely.

- Recall the basic algorithm for multiplying two matrices: For every row of the first, for every column of the second, save the dot product of that row and column as a single element of the product matrix.
- The precondition for this is that the number of columns of the first matrix must equal the number of rows of the second, or the dot product is undefined. So we can multiply a 10×8 matrix by a 8×17 matrix, but not 8×10 by 8×17.
- The runtime to compute a single product AB, where A is $w \times h$, B is $h \times z$ is the number of rows of A times the number of columns of B times the number of columns of A: whz.
  - Note that the resulting matrix will be $w \times z$, with the common dimension disappearing.
- Because one of the dimensions disappears every time we perform a multiplication, the order in which we perform a chain of multiplications affects the runtime. Intuitively, if we drop larger dimensions sooner, they will appear fewer times, leading to a smaller total runtime.

- Example: Suppose we want to compute ABC, with dimensions $[10, 100, 5, 50]$.
  - That is, A is 10×100, B is 100×5, and C is 5×50. We can specify the number of dimensions as a list of $n+1$ integers, since the dimensions must overlap.
  - **Exercise:** Consider the two possible parenthesizations $(AB)C$ and $A(BC)$ and calculate how many scalar multiplications are required for each.

  1. $(AB)C = DC$, where $D = AB$ requires $(10)(100)(5) = 5000$ scalar multiplications. Then computing DC takes $(10)(5)(50) = 2500$ multiplications, for a total of 7500 scalar multiplications.
  2. $A(BC) = AE$, where $E = BC$ takes $(100)(5)(50) = 25,000$ multiplications and computing AE takes $(10)(100)(50) = 50,000$ multiplications, for a total of $75,000$ scalar multiplications.

  - Even with the shortest possible chain, we get an order of magnitude difference in the amount of work required between different orders of computing the product.
- If we want to apply such transformations repeatedly, we need to find a way to minimize the cost of multiplying chains of matrices by determining the best possible ordering.

2 Problem Statement

Input: Compatible chain $A_1, A_2, \ldots, A_n$ of matrices to multiply. Denote their dimensions by $d_0, \ldots, d_n$, where each A_i is $d_{i-1} \times d_i$.

Output: Full parenthesization of the product $A_1 A_2 \cdots A_n$ minimizing the number of scalar multiplications required to compute the product.

First, we need to understand the recursive structure of an optimal solution. At this point, we also consider whether a simple recursive solution will work. We call this a *brute-force* approach, since it tries all possibilities. We can reduce the problem of parenthesizing a chain of n matrices by choosing which of the $n-1$ individual products we will compute *last*, then recursively solving the subchains before and after that point. That is, if we choose the kth product to be last, we break the product $A_1 A_2 \cdots A_n$ into $(A_1 A_2 \cdots A_k)(A_{k+1} \cdots A_n)$, by putting parentheses around the parts of the chain before and after the chosen multiplication. If we continue doing this until our recursive chains are length at most 2, then we have fully specified the order in which to compute the entire chain product.

If we coded this directly, we would check every possible parenthesization, minimizing over the choice of k between 1 and n. To estimate the runtime of this approach, we can express the number of possible parenthesizations of a chain of n matrices as $P(n)$ and describe a recurrence for it:

$$P(n) = \begin{cases} 1 & n \leq 2 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 2 \end{cases}$$

- Here, k represents the point in the chain where we split, making the kth product last. $P(k)$ is then the number of possible parenthesizations of the part of the chain before the split, $P(n-k)$ the number of parenthesizations of the part of the chain after the split, and the total number of parenthesizations is the product of these, because they are independent.
- The sequence of values this defines, as n increases, is known as the Catalan numbers C_n.
- There is a direct formula for the Catalan numbers, like that for Fibonacci numbers:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{1}{n+1}\left(\frac{(2n)!}{n!n!}\right)$$

- The Catalan numbers grow as $\Omega(4^n/n^{1.5})$, which is exponential (worse than 2^n).

This suggests that we do not want to explore all possible parenthesizations. It should be apparent, however, that different ways to split the chain will lead to the same subproblems, so dynamic programming should be able to reduce our runtime.

3 Dynamic Programming Solution

We proceed through the steps of our dynamic programming outline. If you prefer CLRS terminology, those names for the steps are in parentheses.

1. Find recursive structure (characterize optimal solution):

   If we first split at the kth multiplication, the chain product $A_1A_2\cdots A_n$ with dimensions $[d_0,\ldots,d_n]$ breaks into the product of two smaller chains: $(A_1A_2\cdots A_k)(A_{k+1}\cdots A_n) = P_1P_2$, where P_1 is a $d_0\times d_k$ matrix and P_2 is a $d_k\times d_n$ matrix. The cost of this split depends on the optimal parenthesization of each smaller chain, as improving a subchain would improve the overall solution, so we have optimal substructure.

2. Write the recurrence for the optimal value (recursively define the value of the optimal solution):

   Define $MCM(a,b)$ as the minimum number of scalar multiplications required to compute the chain product $A_a\cdots A_b$, where $1\le a\le b\le n$.

   - The minimum cost to compute P_1 is $MCM(1,k)$.
   - The minimum cost to compute P_2 is $MCM(k+1,n)$.
   - The final multiplication P_1P_2 costs $d_0d_kd_n$.

   This gives us the following recursion for the smallest number of scalar multiplications:

   $$MCM(1,n) = \min_{1\le k<n}(MCM(1,k) + MCM(k+1,n) + d_0d_kd_n)$$

   We need to generalize a bit, since not every subproblem starts at index 1 or ends at index n:

   $$MCM(i,j) = \min_{i\le k<j}(MCM(i,k) + MCM(k+1,j) + d_{i-1}d_kd_j$$

   The base case is if $i=j$, we have a chain of one matrix, so there is no work to do, and $MCM(i,i)=0$.

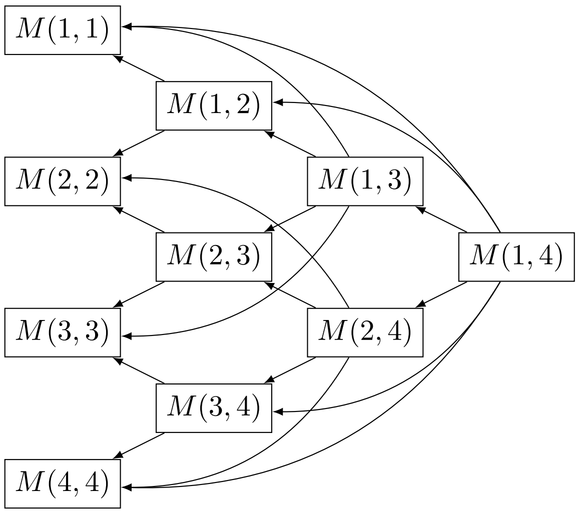
   We can now understand the space of possible subproblems. Since there are two parameters, each ranging from 1 to n, we have an $n\times n$ table of possible subproblems:

$i\backslash j$	1	2	$\cdots$	$n-1$	n
1	0				
2	▨	0			
$\vdots$	▨	▨	$\ddots$		
$n-1$	▨	▨	▨	0	
n	▨	▨	▨	▨	0

   - Base cases $(i=j)$ are the main diagonal.
   - Below the diagonal, $i>j$, which is meaningless in terms of chain products, so we ignore those cells.
   - Whole problem is $(1,n)$, which is the top-right cell.

- Each cell depends on all cells to the left (smaller j, subchain before split) and all cells below (larger i, subchain after split). For each k, we will add one cell to the left and one below, plus the cost of the kth multiplication.
  - **Exercise:** What order should we use for bottom-up computation?
    - Diagonals, moving up and right from the base cases

We can also draw the subproblem graph. Consider what it looks like for $n = 4$:



3. Write code (compute the value of an optimal solution):

   We leave top-down code as an exercise, since it follows the recurrence directly, with the addition of memoization. For bottom-up code, we fill the table by diagonals, starting from the base cases. Note that in each diagonal, the different between i and j is constant, and as we move away from the base cases, it increases by one for each successive diagonal. This allows us to compute the indices of each cell we will fill.

```
function BU-MCM($A_1, \dots A_n$)
    create $M[i,j], sol[i,j]$, $n \times n$ tables, initially empty
    for $i = 1$ to $n$ do
        $M[i,i] = 0$                                    ▷ Base cases
    end for
    for $diag = 2$ to $n$ do                            ▷ Diagonal counter
        for $row = 1$ to $n - diag + 1$ do
            $col = row + diag - 1$
            $M[i,j] = \infty$
            for $split = row$ to $col - 1$ do
                if $M[row, split] + M[split, col] + d_{row-1}d_{split}d_{col} < M[row, col]$ then
                    $M[row, col] = M[row, split] + M[split, col] + d_{row-1}d_{split}d_{col}$
                    $sol[row, col] = split$
                end if
            end for
        end for
    end for
    return $M, sol$
end function
```

Runtime: $O(n^3)$, from the $n^2/2$ cells to fill, each requiring up to n work to try all possible split points.

Correctness: From our recurrence and the fact that all dependencies are solved before used.

Example Let $n = 4, D = [30, 1, 40, 10, 25]$.

	1	2	3	4
1	0	1200_1	700_1	1400_1
2		0	400_2	650_3
3			0	10000_3
4				0

Footnotes denote the choice of *split* that led to the optimal solution.

4. Save choices that led to the optimal solution (generate the optimal solution):

   By adding a *sol* table, we can store the value of *split* that gave the best solution for each subproblem. We can then use that value to reconstruct the optimal solution as follows:

```
function MCM-Print($sol, row, col$)
    if $i == j$ then print "$A_i$"
    else
        $split = sol[row, col]$
        print "(" + MCM-Print($sol, row, split$) + MCM-Print($sol, split + 1, col$) + ")"
    end if
end function
```

   If we call MCM-Print$(sol, 1, n)$, this function will print the parenthesized chain.

Exercise: Determine the value of an optimal parenthesization for input $n = 5$, $D = [30, 35, 15, 5, 10, 20]$