Matrix chain multiplication

Consider 3 matrices A_1, A_2, A_3 of sizes n x m, m x l, l x q.

Recall the associative property of matrix product $(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$: it can be that by changing the parenthesization we can decrease (or increase) the number of scalar multiplication required.

Consider a **chain** of matrices $(A_1, ..., A_n)$ which is compatible for the matrix multiplication, i.e. A_i has dimesion $p_{i-1} \times p_i \ \forall i \in [1, n]$

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

We wanto to compute a **parenthesization** that minimize the number of scalar multiplications required.

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.

Naive algorithm

We try to search among all possible parenthesization in a recursive way:

- n = 1 obvious;
- n > 1 $\forall k \in [1, .., n-1]$ we recursively produce the parenthesization $(A_1, .., A_k)(A_{k+1}, ..., A_n)$, so that $(A_1 \times ... \times A_k) \times (A_{k+1} \times ... \times A_n)$.

The number of possible alternative parenthesization of $(A_1, ..., A_n)$ P(n) is recursively computed as:

- 1, if n = 1
- if we fix k and we selected a parenthesization for the first part of the chain $(A_1...A_k)$, the number of parenthesization is P(n-k), (with n-k size of the second part of the chain), so we have $P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n-k)$, if n > 1

It can be proved that $P(n) \in \Omega(2^n)$, still too many to be enumerated. Indeed: we unroll the sum to get $P(n) = P(1) \cdot P(n-1) + ... + P(n-1) \cdot P(n-(n-1)) =$
$= P(1) \cdot P(n-1) + ... + P(n-1) \cdot P(1)$
$\geq 2 \cdot P(1) \cdot P(n-1) = 2 \cdot P(n-1)$

Hence:

- $P(n) \geq P(1)$, if n = 1
- $P(n) \geq 2 \cdot P(n-1)$, if n > 1

So that $P(n) \in \Omega(2^n)$, this is not acceptable.

Observations:

- We can split the overall computation of finding a better parenthesization in subproblems; indeed we can see that if the optimal parenthesization for the chian is $(A_1, ..., A_k)(A_{k+1}, ..., A_n)$, then $(A_1, ..., A_k)$ is optimal for the first part of the chain, $(A_{k+1}, ..., A_n)$ is optimal for the second part of the chain, so first we can compute the matrices $A_{1...k}$ and;$A_{k+1,...,n}$ and then multilply them together to get the final product;
- many branches of the naive algorithm performs exactly the same computation, indeed: for every parenthesization of $(A_1, ..., A_k)$, the parenthesization for the second part $(A_{k+1}, ..., A_n)$ are recomputed.

Idea is to use *dynamic programming* which consists in memoazing the result of the previous recursive call. In this way we end up in a subexponential solution.

Dynamic programming solution

Take a matrix m, in this matrix we store the minimum number of products for all the sub-chains, i.e. $m[i, j]$ is the minimum number of scalar multiplication nedded to compute the matrix $A_{i...j}$. Hence for the full problem, the lowest cost way to compute $A_{1..n}$ would be $m[1, n]$.

We can define this matrix recursively:

- $m[i, j] = 0$ if i = j (trivial, the chain is just one matrix, no scalar multiplications are required to compute the product);
- $m[i, j] = \min_{k\in[i,j-1]}\{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\}$ if i < j (for what we observed before, we assume that to optimally parenthesize, we split $A_{i..j}$ in $A_{i..k}$ and $A_{k+1..j}$, then $m[i, j]$ equals the the minimum cost for computing the subproducts $A_{i..k}, A_{k+1,..j}$ plus the cost of multiplying these two matrices together. Since we don't know the true value of k, we minimize among all the possible values that k can take).

It is symmetric, as follows for the observations we have done before.

So $m[i, j]$ gives the cost of optimal solution to subproblems, but it doesn't provide all the information we need to construct an optimal solution.

For this reason we compute another matrix s that store the position in which parenthesis are set, namely $s[i, j]$ is the value of k at which we split the product $A_{i..j}$ in an optimal parenthesization.

That is $s[i, j] = k$ such that $k = argmin\{m[i, k]+m[k+1, j]+p_{i-1}p_kp_j\}$, i.e. the parenthesization for the current level.

This is a recursive process, but it is easy to do it iteratively, indeed this recursive algorithm may encounter each subproblem many times in different branches of the recursion tree, as already observed (overlapping subproblems).

At the end the computation of $m[i, j]$ and $s[i, j]$ has complexity $\Theta(n^3)$, indeed finding the $k \in [i, j - 1]$ that minimizes the expression has constant complexity

$\Theta(1)$ at each step, that is $\sum_{k=i}^{j-1} \Theta(1 = \Theta(j-1))$, since $i \in [1,n]$ and $j \in [i,n]$ we get

$T_C(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \Theta(j-1) =$
$= \Theta(\sum_{i=1}^{n} (\sum_{j=i}^{n} j) - n \cdot i) =$
$= \Theta(\sum_{i=1}^{n} \frac{n \cdot (n+1)}{2} - \frac{i \cdot (i+1)}{2} - n \cdot i) =$
$= \Theta(n^3)$

Thanks to dynamic programming we reduced from an exponential complexity to a cubic one.