Analysis of Algorithms, I

CSOR W4231

Eleni Drinea

Computer Science Department

Columbia University

More dynamic programming: matrix chain multiplication

Outline

Today

1. Matrix chain multiplication
2. A first attempt: brute-force
3. A second attempt: divide and conquer
4. Organizing DP computations

Matrix chain multiplication example

Example 1.

Input: matrices A_1, A_2, A_3 of dimensions 6×1, 1×5, 5×2

Output:

- a way to compute the product $A_1 A_2 A_3$ so that the number of arithmetic operations performed is minimized;
- the minimum number of arithmetic operations required.

Useful observations

Remark 1.

- *We do not want to compute the actual product.*
- *Matrix multiplication is associative but not commutative (in general). Hence a solution to our problem corresponds to a parenthesization of the product.*
- *We want the optimal parenthesization and its cost, that is, the parenthesization that minimizes the number of arithmetic operations, as well as that number.*

Estimating #arithmetic operations

- Let A, B be matrices of dimensions $m \times n$, $n \times p$.
- Let $C = AB$. Then C is an $m \times p$ matrix such that

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}.$$

⇒ c_{ij} requires n scalar multiplications, $n - 1$ additions

⇒ #arithmetic operations to compute c_{ij} is dominated by #scalar multiplications

- Total #scalar multiplications to fill in C is mnp

Minimizing #scalar multiplications for $A_1A_2A_3$

Input: A_1, A_2, A_3 of dimensions 6×1, 1×5, 5×2 respectively

Given a parenthesization of the input matrices, its cost is the total # scalar multiplications to compute the product.

Two ways of computing $A_1A_2A_3$:

1. $(A_1A_2)A_3$: first compute A_1A_2, then multiply it by A_3
   - $6 \cdot 1 \cdot 5$ scalar multiplications for A_1A_2
   - $6 \cdot 5 \cdot 2$ scalar multiplications for $(A_1A_2)A_3$

   $\Rightarrow$ 90 scalar multiplications in total

2. $A_1(A_2A_3)$: first compute A_2A_3, then multiply A_1 by A_2A_3
   - $1 \cdot 5 \cdot 2$ scalar multiplications for A_2A_3
   - $6 \cdot 1 \cdot 2$ scalar multiplications for $A_1(A_2A_3)$

   $\Rightarrow$ 22 scalar multiplications in total

Remark 2.

Parenthesization $A_1(A_2A_3)$ *improves over* $(A_1A_2)A_3$ *by over* 75%.

(Fully) Parenthesized products of matrices

Definition 2.

A product of matrices is fully parenthesized if it is

1. a single matrix; or
2. the product of two fully parenthesized matrices, surrounded by parentheses.

Examples: $((A_1 A_2) A_3)$ and $(A_1 (A_2 A_3))$ are fully parenthesized.

Remark: we will henceforth refer to a *full parenthesization* simply as a *parenthesization*.

Matrix chain multiplication

Input: n matrices $A_1, A_2, \ldots, A_n$, with dimensions $p_{i-1} \times p_i$, for $1 \leq i \leq n$.

Output:

1. an **optimal** parenthesization of the input
   (i.e., a way to compute $A_1 \cdots A_n$ incurring minimum cost)
2. its **cost**
   (i.e., total # scalar multiplications to compute $A_1 \cdots A_n$)

Example: the optimal parenthesization for Example 1 is $(A_1(A_2A_3))$ and its cost is 22.

Today

1. Matrix chain multiplication
2. A first attempt: brute-force
3. A second attempt: divide and conquer
4. Organizing DP computations

Brute-force approach

- $A_1, \ldots, A_n$ are matrices of dimensions $p_{i-1} \times p_i$ for $1 \leq i \leq n$.
- Consider the product $A_1 \cdots A_n$.
- Let $P(n) = $ #parenthesizations of the product $A_1 \cdots A_n$.
- Then $P(0) = 0$, $P(1) = 1$, $P(2) = 1$
- By Definition 2, for $n > 2$, every possible parenthesization of $A_1 \cdots A_n$ can be decomposed into the product of two parenthesized subproducts for some $1 \leq k \leq n-1$:

$$((A_1 A_2 \cdots A_k)(A_{k+1} \cdots A_n))$$

Computing #possible parenthsizations

- Given k, the number of parenthesizations for the product

$$((A_1 A_2 \cdots A_k)(A_{k+1} \cdots A_n))$$

can be computed recursively:

$$P(k) \cdot P(n-k)$$

- There are $n-1$ possible values for k. Hence

$$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n-k), \text{ for } n > 1$$

Bounding $P(n)$

- We may obtain a crude yet sufficient for our purposes **lower bound** for $P(n)$ as follows

$$\begin{aligned} P(n) &\geq P(1) \cdot P(n-1) + P(2) \cdot P(n-2) \\ &\geq P(n-1) + P(n-2) \qquad (1) \end{aligned}$$

- By strong induction on n, we can show that $P(n) \geq F_n$, the n-th Fibonacci number.

- Hence $P(n) = \Omega(2^{n/2})$.
  - In fact, $P(n) = \Omega(2^{2n}/n^{3/2})$ (e.g., see your textbook).

$\Rightarrow$ Brute force requires exponential time.

Today

1. Matrix chain multiplication
2. A first attempt: brute-force
3. A second attempt: divide and conquer
4. Organizing DP computations

A second attempt: divide and conquer

Notation: $A_{1,n}$ is the optimal parenthesization of the product $A_1 \cdots A_n$, that is, the optimal way to compute this product.

By Definition 2, there exists $1 \leq k^* \leq n-1$ such that $A_{1,n}$ may be decomposed as the product of two fully parenthesized subproducts:

$$A_{1,n} = ((A_1 \cdots A_{k^*})(A_{k^*+1} \cdots A_n))$$

Optimal substructure

Notation: $A_{i,j}$ is the optimal parenthesization of the product $A_i \cdots A_j$.

Fact 3.

There exists $1 \leq k^* \leq n-1$ such that

$$A_{1,n} = (A_{1,k^*} \cdot A_{k^*+1,n}).$$

That is, the optimal parenthesization of the input can be decomposed into the optimal parenthesizations of two subproblems.

The cost of multiplying two matrices

- Recall that matrix A_i has dimensions $p_{i-1} \times p_i$.
- Then $(A_1 \cdots A_k)$ is a $p_0 \times p_k$ matrix;
- $(A_{k+1} \cdots A_n)$ is a $p_k \times p_n$ matrix.

$\Rightarrow$ The total #scalar multiplications required for multiplying matrix $(A_1 \cdots A_k)$ by matrix $(A_{k+1} \cdots A_n)$ is

$$p_0 p_k p_n.$$

Proof of Fact 3

Notation:

- $A_{i,j}$ is the optimal parenthesization of $A_i \cdots A_j$
- $OPT(i,j)$ = cost of $A_{i,j}$ = optimal cost to compute $A_i \cdots A_j$

Since $A_{1,n} = ((A_1 \cdots A_{k^*})(A_{k^*+1} \cdots A_n))$, its cost is given by

$$OPT(1,n) = OPT(1,k^*) + OPT(k^*+1,n) + p_0 p_{k^*} p_n,$$

where

- $OPT(1,k^*), OPT(k^*+1,n)$ are the costs for optimally *(why?)* computing $A_1 \cdots A_{k^*}$, $A_{k^*+1} \cdots A_n$ respectively
- $p_0 p_{k^*} p_n$ is the fixed cost for multiplying $(A_1 \cdots A_{k^*})$ by $(A_{k^*+1} \cdots A_n)$.

Recursive computation of $OPT(1, n)$

Notation: $OPT(1, n)$ = optimal cost for computing $A_1 \cdots A_n$

- **Issue:** we do not know k^*!
- **Solution:** consider every possible value of k.

$$OPT(1, n) = \begin{cases} 0 & \text{, if } n = 1 \\ \min\limits_{1 \leq k < n} \left\{ OPT(1, k) + OPT(k+1, n) + p_0 p_k p_n \right\} & \text{, o.w.} \end{cases}$$

Remark 3.

This recurrence gives rise to an exponential recursive algorithm. However we can use dynamic programming to obtain an efficient solution.

Introducing subproblems

Notation: $OPT(i,j) =$ optimal cost for computing $A_i \cdots A_j$

$$OPT(i,j) = \begin{cases} 0 & \text{, if } i = j \\ \min\limits_{i \leq k < j} \left\{ OPT(i,k) + OPT(k+1,j) + p_{i-1}p_k p_j \right\} & \text{, if } i < j \end{cases}$$

Remark 4.

- *Only* $\Theta(n^2)$ *subproblems.*
- *If subproblems are computed from smaller to larger, then only* $\Theta(j-i) = O(n)$ *work per subproblem: each term inside the* min *computation requires time* $O(1)$ *(why?).*

Today

1. Matrix chain multiplication
2. A first attempt: brute-force
3. A second attempt: divide and conquer
4. Organizing DP computations

Bottom-up computation of subproblems

Define matrix $M[1:n, 1:n]$, $S[1:n-1, 2:n]$ such that

$$\begin{array}{rcll} M[i,j] & = & OPT(i,j), & \text{for } 1 \le i \le j \le n \\ S[i,j] & = & k, \text{ if } A_{i,j} = A_{i,k}A_{k+1,j}, & \text{for } 1 \le i < j \le n \end{array}$$

- Only need fill in the upper triangle of M, where $i \le j$
- Start from the main diagonal, proceed diagonal by diagonal
- Last entry to fill in: $M[1,n]$, the cost of the optimal parenthesization of the entire product $A_1 \cdots A_n$
- Running time: $O(n^3)$
  - $\Theta(n^2)$ entries to fill in
  - each entry requires $\Theta(j-i) = O(n)$ work
- Space: $\Theta(n^2)$

Example

Input

- 6×1 matrix A_1
- 1×5 matrix A_2
- 5×2 matrix A_3
- 2×3 matrix A_4

Output

- the cost of the optimal parenthesization of $A_1 A_2 A_3 A_4$ (by filling in the dynamic programming table M)

Computing the cost of the optimal parenthesization in $O(n^3)$ (from CLRS)

```
MATRIX-CHAIN-ORDER(p)
n = p.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i,i] = 0
for l = 2 to n                  // l is the chain length
    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
```

Reconstructing the optimal parenthesization (from CLRS)

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Memoized recursion

Use the original recursive algorithm together with M:

- initialize M to ∞ above the main diagonal and to 0 on the main diagonal.
- to solve a subproblem, look up its value in M
  - if it is ∞, solve the subproblem **and** store its cost in M;
  - else, directly use its value from M.

Remark 5.

- *The memoized recursive algorithm solves every subproblem once, thus overcoming the main source of inefficiency of the original recursive algorithm.*
- *Running time:* $O(n^3)$.

Memoized recursion pseudocode (from CLRS)

```
MEMOIZED-MATRIX-CHAIN(p)
n = p.length − 1
let m[1..n, 1..n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
              + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Dynamic programming vs Divide & Conquer

- They both combine solutions to subproblems to generate a solution to the whole problem.
- However, divide and conquer starts with a large problem and divides it into small pieces.
- While dynamic programming works from the bottom up, solving the smallest subproblems first and building optimal solutions to steadily larger problems.