

12. Dynamic Programming for Matrix Chain Multiplication

CPSC 535

Kevin A. Wortman



CALIFORNIA STATE UNIVERSITY
FULLERTON



This work is licensed under a Creative Commons Attribution 4.0 International License.

Big Idea: 2D Table

- ▶ *Recall:* dynamic programming
 - ▶ problem has recursive structure
 - ▶ overlapping subproblems
 - ▶ use table to store solutions, avoid duplicated effort
 - ▶ top-down or bottom-up
- ▶ so far: **1D table** – has one index
- ▶ now: **2D table** – has *two* indices

Matrix Multiplication

for matrices A_1, A_2 :

$$A_1 A_2$$

Recall:

$$\begin{bmatrix} 5 & 12 & 5 \\ 16 & 9 & 4 \end{bmatrix} \times \begin{bmatrix} 19 & 2 \\ 9 & 5 \\ 8 & 11 \end{bmatrix} = \begin{bmatrix} 5 \times 19 + 12 \times 9 + 5 \times 8 & 125 \\ & 417 \\ & 121 \end{bmatrix}$$

Matrix Multiplication Algorithms

Recall:

- ▶ Naïve algorithm: three nested loops, $O(n^3)$
- ▶ Strassen's algorithm: divide-and-conquer, $\approx O(n^{2.8074})$
- ▶ Those analyses assumed A_1, A_2 are both square $n \times n$ matrices
- ▶ Now: matrix sizes may differ
- ▶ **Compatible:** A_1 and A_2 are compatible when $A_1.columns = A_2.rows$

Naïve Matrix Multiplication Algorithm

```
1: function MATRIX-MULTIPLY(A, B)
2:   C = new A.rows × B.columns matrix
3:   for i from 1 to A.rows do
4:     for j from 1 to B.columns do
5:       cij = 0
6:       for k from 1 to A.columns do
7:         cij = cij + aik · bkj
8:       end for
9:     end for
10:  end for
11:  return C
12: end function
```

Analysis: $\Theta(A.rows \cdot A.columns \cdot B.columns)$

Matrix Chain Multiplication

Given n compatible matrices A_1, A_2, \dots, A_n , compute

$$A_1 A_2 \dots A_n$$

- ▶ Recall: matrix multiplication is **associative**
- ▶ May parenthesize $A_1 A_2 \dots A_n$ in any order
- ▶ Q: which order is most efficient?

Equivalent Parenthesizations

$$\begin{aligned}A_1 A_2 A_3 A_4 &= A_1 (A_2 (A_3 A_4)) \\&= A_1 ((A_2 A_3) A_4) \\&= (A_1 A_2) (A_3 A_4) \\&= (A_1 (A_2 A_3)) A_4 \\&= ((A_1 A_2) A_3) A_4\end{aligned}$$

Total runtime depends on the dimensions of $A_1 \dots A_4$.

Example: Different Runtimes

Given three matrices A_1, A_2, A_3 with dimensions

matrix	rows	columns
A_1	10	100
A_2	100	5
A_3	5	50

- ▶ $((A_1 A_2) A_3)$ costs
 $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5,000 + 2,500 = 7,500$ scalar multiplies
- ▶ $(A_1 (A_2 A_3))$ costs
 $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25,000 + 50,000 = 75,000$ scalar multiplies
- ▶ first is order of magnitude faster

Matrix Chain Multiplication Problem

matrix chain multiplication problem

input: a sequence $\langle A_1, A_2, \dots, A_n \rangle$ of $n > 0$ compatible matrices, and sequence $p = \langle p_0, p_1, \dots, p_n \rangle$ of integers, where matrix A_i has p_{i-1} rows and p_i columns

output: a parenthesization of $A_1 A_2 \dots A_n$ that minimizes scalar multiplications

matrix chain multiplication value problem

input: a sequence $\langle A_1, A_2, \dots, A_n \rangle$ of $n > 0$ compatible matrices, and sequence $p = \langle p_0, p_1, \dots, p_n \rangle$ of integers, where matrix A_i has p_{i-1} rows and p_i columns

output: the minimum number of scalar multiplies necessary to multiply $A_1 A_2 \dots A_n$

Design Process

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
2. Derive a **recurrence** for an optimal value.
3. Design a divide-and-conquer algorithm that computes an **optimal value**.
4. Design a dynamic programming algorithm that computes an **optimal value**.
 - 4.1 **top-down** alternative: add table base case (**memoization**)
 - 4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

Matrix Chain Multiplication Step 1

1. Identify the problem's **solution** and **value**, and note which is our **goal**.

matrix chain multiplication value problem

input: a sequence $\langle A_1, A_2, \dots, A_n \rangle$ of $n > 0$ compatible matrices, and sequence $p = \langle p_0, p_1, \dots, p_n \rangle$ of integers, where matrix A_i has p_{i-1} rows and p_i columns

output: the minimum number of scalar multiplies necessary to multiply $A_1 A_2 \dots A_n$

- ▶ **solution:** parenthesized expression e.g. $(A_1(A_2 A_3))(A_4 A_5)$
- ▶ **value:** number of multiplications e.g. 75,000
- ▶ goal: **value**

Matrix Chain Multiplication Step 2

2. Derive a **recurrence** for an optimal value.

- ▶ define $r_{i,j}$ = minimum number of multiplies for $A_i A_{i+1} \dots A_j$
- ▶ (note: **two** indices)
- ▶ solution to whole problem is $r_{1,n}$
- ▶ base case: A_i by itself; so when $i = j$, $r_{i,j} = 0$
- ▶ general case:
 - ▶ **think** divide-and-conquer; define $r_{i,j}$ in terms of $r_{i,<j}$
 - ▶ make the problem **one piece** smaller
 - ▶ given $A_i A_{i+1} \dots A_j$, split w/ parenthesis at index k :

$$A_i A_{i+1} \dots A_j = (A_i A_{i+1} \dots A_k)(A_{k+1} A_{k+2} \dots A_j)$$

- ▶ try every option and keep the optimal one

$$r_{i,j} = \min_{1 \leq k \leq j} r_{i,k} + r_{k+1,j} + p_i p_k p_j$$

Matrix Chain Multiplication Step 3

3. Design a divide-and-conquer algorithm that computes an **optimal value**.

```
1: function MATRIX-CHAIN-VALUE-DC( $p[0..n]$ )
2:   return MC-DC( $p, 0, n$ )
3: end function
4: function MC-DC( $p[0..n], i, j$ )
5:   if  $i == j$  then
6:     return 0
7:   end if
8:    $q = \infty$ 
9:   for  $k$  from 1 to  $j - 1$  do
10:     $q = \min(q, \text{MC-DC}(p, i, k) + \text{MC-DC}(p, k + 1, j) + p[i] \times p[k] \times p[j])$ 
11:  end for
12:  return  $q$ 
13: end function
```

Sidebar: Analysis of MATRIX-CHAIN-VALUE-DC

- ▶ MC-DC-REC calls itself $O(n)$ times in general case
- ▶ like CUT-ROD-DC
- ▶ exponential time
- ▶ again, dynamic programming will circumvent all this recursion

Matrix Chain Multiplication Step 4.a

4. Design a dynamic programming algorithm that computes an **optimal value**.

4.1 **top-down** alternative: add table base case (**memoization**)

- ▶ Recall **memoization**: use a hash dictionary to make a “memo” of pre-calculated solutions
- ▶ create hash table T
- ▶ use pair (i, j) as key in table T , storing $r_{i,j}$

Matrix Chain Multiplication Step 4.a

```
1: function MATRIX-CHAIN-VALUE-MEMOIZED( $p[0..n]$ )
2:   HASH-TABLE-CREATE( $T$ )
3:   return MC-M( $T, p, 0, n$ )
4: end function
5: function MC-M( $T, p[0..n], i, j$ )
6:    $q = \text{HASH-TABLE-SEARCH}(T, (i, j))$ 
7:   if  $q \neq \text{NIL}$  then
8:     return  $q$ 
9:   end if
10:  if  $i == j$  then
11:     $q = 0$ 
12:  else
13:     $q = \infty$ 
14:    for  $k$  from 1 to  $j - 1$  do
15:       $q = \min(q, \text{MC-M}(p, i, k) + \text{MC-M}(p, k + 1, j) + p[i] \times p[k] \times p[j])$ 
16:    end for
17:  end if
18:   $q.\text{key} = (i, j)$ 
19:  HASH-TABLE-INSERT( $q$ )
20:  return  $q$ 
21: end function
```


Memoized Algorithm Analysis

- ▶ T contains $\Theta(n^2)$ pairs (i, j)
- ▶ each entry is inserted exactly once
- ▶ in the general case, MC-M takes $\Theta(n)$ expected time
- ▶ \Rightarrow MATRIX-CHAIN-VALUE-MEMOIZED takes $\Theta(n^3)$ expected time

Matrix Chain Multiplication Step 4.b

4. Design a dynamic programming algorithm that computes an **optimal value**.
 - 4.1 **top-down** alternative: add table base case (**memoization**)
 - 4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
- ▶ create 2D array m where $m[i][j] = r_{i,j}$
 - ▶ **bottom-up**: write an explicit **for** loop that computes and stores every general case
 - ▶ need to order loops so we never use an uninitialized element
 - ▶ \therefore initialize chain length 1(base case), 2, \dots , n

Matrix Chain Multiplication Step 4.b

```
1: function MATRIX-CHAIN-BU( $p[0..n]$ )
2:   Create array  $m[1..n][1..n]$ 
3:   for  $i$  from 1 to  $n$  do
4:      $m[i][i] = 0$  ▷ base case, length=1
5:   end for
6:   for  $\ell$  from 2 to  $n$  do ▷  $\ell$  = general-case length
7:     for  $i$  from 1 to  $(n - \ell + 1)$  do
8:        $j = i + \ell - 1$ 
9:        $q = \infty$ 
10:      for  $k$  from  $i$  to  $j - 1$  do
11:         $q = \min(q, m[i][k] + m[k + 1][j] + p[i] \times p[k] \times p[j])$ 
12:      end for
13:       $m[i][j] = q$ 
14:    end for
15:  end for
16:  return  $m[1, n]$ 
17: end function
```

Matrix Chain Multiplication Analysis

- ▶ MATRIX-CHAIN-BU is clearly $\Theta(n^3)$ time
- ▶ top-down memoized algorithm: $\Theta(n^3)$ expected time
- ▶ bottom-up algorithm: $\Theta(n^3)$ time with faster constant factors

Matrix Chain Multiplication Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

matrix chain multiplication value problem

input: a sequence $\langle A_1, A_2, \dots, A_n \rangle$ of $n > 0$ compatible matrices, and sequence $p = \langle p_0, p_1, \dots, p_n \rangle$ of integers, where matrix A_i has p_{i-1} rows and p_i columns

output: the minimum number of scalar multiplies necessary to multiply $A_1 A_2 \dots A_n$

matrix chain multiplication problem

input: (same)

output: a parenthesization of $A_1 A_2 \dots A_n$ that minimizes scalar multiplications

Matrix Chain Multiplication Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

- ▶ **idea:** for each (i, j) , record which k defines the minimum $m[i][j]$
- ▶ happens inside the inner-most k loop
- ▶ define

$s[i][j]$ = the index k that minimizes $r_{i,k} + r_{k+1,j} + p_i p_k p_j$

- ▶ rewrite $\min(q, \dots)$ statement as an **if** so we can update $s[i][j]$

Matrix Chain Multiplication Step 5

```
1: function MATRIX-CHAIN-SOLUTION( $p[0..n]$ )
2:   Create arrays  $m[1..n][1..n]$  and  $s[1..n][1..n]$ 
3:   for  $i$  from 1 to  $n$  do
4:      $m[i][i] = 0$  ▷ base case, length=1
5:   end for
6:   for  $\ell$  from 2 to  $n$  do ▷  $\ell$  = general-case length
7:     for  $i$  from 1 to  $(n - \ell + 1)$  do
8:        $j = i + \ell - 1$ 
9:        $q = \infty$ 
10:      for  $k$  from  $i$  to  $j - 1$  do
11:         $q' = m[i][k] + m[k + 1][j] + p[i] \times p[k] \times p[j]$ 
12:        if  $q' < q$  then
13:           $q = q'$ 
14:           $s[i][j] = k$ 
15:        end if
16:      end for
17:       $m[i][j] = q$ 
18:    end for
19:  end for
20:  return MC-BTRACK( $s, 1, n$ )
21: end function
```

Matrix Chain Multiplication Step 5

```
1: function MC-PARENS( $s[1..n][1..n]$ ,  $i, j$ )  
2:   if  $i == j$  then                                     ▷ single matrix  
3:     return " $A_i$ "  
4:   end if  
5:    $k = s[i][j]$   
6:   return "(" + MC-BTRACK( $s, i, k$ ) + "()" + MC-BTRACK( $s, k, j$ ) + ")"  
7: end function
```