Dynamic Programming

Dr. G P Gupta

Dynamic Programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- dynamic programming applies when the subproblems overlap—that is, ***when subproblems share subsubproblems***.
- A dynamic-programming algorithm ***solves each subsubproblem just once*** and then **saves its answer in a table**,
  - thereby **avoiding the work of recomputing** the answer every time it solves each subsubproblem.

Dynamic Programming

- Dynamic Programming(DP) applies to optimization problems
  - Such problems can have many possible solutions.
  - ***Each solution has a value***, and
  - we wish to ***find a solution with the optimal*** (minimum or maximum) ***value***.
  –

Developing a dynamic- programming algorithm

- ***follow a sequence of four steps:***

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information.

Divide & Conquer vs. Dynamic Programming

- Divide and Conquer algorithms partition the problem into ***independent subproblems***.
- Dynamic Programming is applicable when the ***subproblems are not independent.***(In this case DP algorithm does more work than necessary)
- Dynamic Programming algorithm ***solves every subproblem just once*** and then ***saves its answer in a table***.

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….
- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

The steps of a dynamic programming

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed information

7

Dynamic Programming

- Example
  - Matrix-chain multiplication
  - Longest common subsequence

8

Matrix-chain multiplication Problem

9

Matrix-Chain multiplication

- given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$

  of n matrices to be multiplied, and
- we wish to compute the product

$$A_1 A_2 \ldots A_n$$

10

Matrix-Chain multiplication cont..

- Matrix multiplication is ***assosiative***, and so all parenthesizations yield the same product.
- For example, if the chain of matrices is $\langle A_1 A_2 \ldots A_4 \rangle$ then the product $\langle A_1 A_2 A_3 A_4 \rangle$ ***can be fully paranthesized in five distinct way:***

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

11

Matrix-Chain multiplication

MATRIX-MULTIPLY (A,B)
if *columns* [A] $\neq$ *rows* [B]
  then error "incompatible dimensions"
    else for $i \leftarrow 1$ **to** *rows* [A]
      do for $j \leftarrow 1$ **to** *columns* [B]
        do C[i, j]$\leftarrow$0
          for $k \leftarrow 1$ **to** *columns* [A]
            do C[i, j]$\leftarrow$ C[i, j] +A[i, k]*B[k, j]
return C

12

Matrix-Chain multiplication cont..

Cost of the matrix multiplication:

An example: $\langle A_1 A_2 A_3 \rangle$

$A_1: \ 10 \times 100$

$A_2: \ 100 \times 5$

$A_3: \ 5 \times 50$

13

Matrix-Chain multiplication cont..

If we multiply $((A_1 A_2) A_3)$ we perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications.

If we multiply $(A_1 (A_2 A_3))$ we perform $100 \cdot 5 \cdot 50 = 25\,000$ scalar multiplications to compute the 100×50 matrix product $A_2 A_3$, plus another $10 \cdot 100 \cdot 50 = 50\,000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75 000 scalar multiplications.

14

Matrix-Chain multiplication cont..

- ***The problem:***

Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where matrix A_i has dimension p_{i-1} x p_i, fully paranthesize the product $A_1 A_2 \ldots A_n$ in a way that ***minimizes the number of scalar multiplications.***

15

Matrix-Chain multiplication cont..

- Counting the ***number of alternative paranthesization*** : b_n

$$b_n = \begin{cases} 1 & \text{if } n = 1 \text{, there is only one matrix} \\ \sum_{k=1}^{n-1} b_k b_{n-k} & \text{if } n \geq 2 \end{cases}$$

$$b_n = \Omega(2^n)$$

16

Matrix-Chain multiplication cont..

Step 1: The structure of an optimal aranthesization(op)

- Find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.
- Let $A_{i..j}$ where $i \leq j$, denote the matrix product $A_i A_{i+1} \ldots A_j$
- Any parenthesization of $A_i A_{i+1} \ldots A_j$ must split the product between A_k and A_{k+1} for $i \leq k < j$.

17

Matrix-Chain multiplication cont..

The optimal substructure of the problem:

- Suppose that an ***op*** of $A_i A_{i+1} \ldots A_j$ splits the product between A_k and A_{k+1} then the paranthesization of the subchain $A_i A_{i+1} \ldots A_k$ within this parantesization of $A_i A_{i+1} \ldots A_j$ must be an ***op*** of $A_i A_{i+1} \ldots A_k$

18

Matrix-Chain multiplication cont..

Step 2: A recursive solution:

- Let $m[i,j]$ be the ***minimum number of scalar multiplications*** needed to compute the matrix $A_{i...j}$ where $1 \le i \le j \le n$.
- Thus, ***the cost of a cheapest way to compute*** $A_{1...n}$ would be $m[1,n]$.
- Assume that the ***op*** splits the product $A_{i..j}$ between A_k and A_{k+1}.where $i \le k < j$.
- Then **$m[i,j]$ =The minimum cost for computing $A_{i...k}$ and $A_{k+1...j}$ + the cost of multiplying these two matrices**.

Matrix-Chain multiplication cont..

Recursive defination for the minimum cost of paranthesization:

$$m[i,j] = \begin{cases} 0 & \text{if } \ i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } \ i < j. \end{cases}$$

Matrix-Chain multiplication cont..

- To help us keep track of ***how to constrct an optimal solution***
- we **define *s*[*i,j*]** to be a value of k at which we can split the product $A_{i...j}$ to obtain an optimal paranthesization.

That is ***s*[*i,j*] equals a value *k*** such that

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

$$s[i,j] = k$$

Matrix-Chain multiplication cont..

Step 3: Computing the optimal costs

It is easy to write a recursive algorithm based on recurrence for computing $m[i,j]$.

But the running time will be exponential!...

Matrix-Chain multiplication cont..

Step 3: Computing the optimal costs

We ***compute the optimal cost by using a tabular, bottom-up approach.***

Matrix-Chain multiplication cont..

```
MATRIX-CHAIN-ORDER(p)
n←length[p]-1
for i←1 to n
    do m[i,i]←0
for l←2 to n
      do for i←1  to  n-l+1
              do  j←i+l-1
                   m[i,j]← ∞
                    for k←i  to  j-1
                           do q←m[i,k] + m[k+1,j]+p_{i-1} p_k p_j
                               if   q < m[i,j]
                                   then m[i,j] ←q
                                          s[i,j] ←k
return m and s
```

Matrix-Chain multiplication cont..

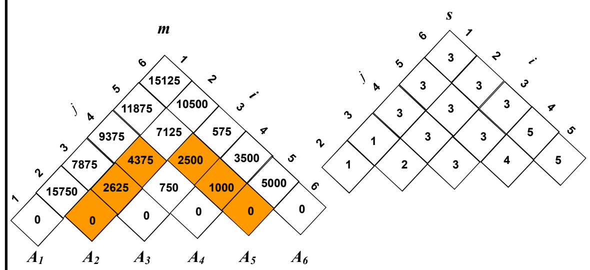
An example:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p 4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= (7125)$$

Matrix-Chain multiplication cont..



Matrix-Chain multiplication cont..

Step 4: Constructing an optimal solution

- An optimal solution can be constructed from the computed information stored in the table $s[1...n, 1...n]$.
- We know that the final matrix multiplication is

$$A_{1...s[1,n]} A_{s[1,n]+1...n}$$

The earlier matrix multiplication can be computed recursively.

Matrix-Chain multiplication cont..

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


Matrix-Chain multiplication cont..

RUNNING TIME:

- Recursive solution takes exponential time.
- ***Matrix-chain order yields a running time of*** $O(n^3)$

Elements of dynamic programming

When should we apply the method of Dynamic Programming?

Two key ingredients:
- Optimal substructure
- Overlapping subproblems

Elements of dynamic programming cont..

Optimal substructure (os):

- A problem exhibits **os** if an optimal solution to the problem contains within it optimal solutions to subproblems.
- Whenever a problem exhibits **os**, it is a good clue that dynamic programming might apply.
- In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems.
- Dynamic programming uses optimal substructure in a bottom-up fashion.

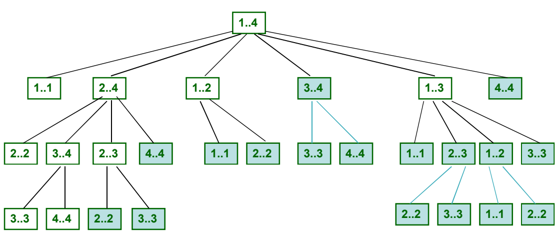
Elements of dynamic programming cont..

Overlapping subproblems:

- When a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has ***overlapping subproblems***.
- In contrast , a ***divide-and-conquer*** approach is suitable usually generates brand new problems at each step of recursion.
- Dynamic programming algorithms take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed.

Elements of dynamic programming cont..

Overlapping subproblems: (cont.)



The recursion tree of RECURSIVE-MATRIX-CHAIN(*p*, 1, 4). The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN(*p*, 1, 4).

Elements of dynamic programming cont..

```
RECURSIVE-MATRIX-CHAIN (p, i, j)
if i = j
    then return 0
m[i,j] ←∞
for k←i to j-1
     do q←RECURSIVE-MATRIX-CHAIN (p, i, k)
             + RECURSIVE-MATRIX-CHAIN (p, k+1, j)+ p_{i-1} p_k p_j
          if q < m[i,j]
               then m[i,j] ←q
 return m[i,j]
```


Elements of dynamic programming cont..

Memoization

- There is a variation of dynamic programming that often offers the efficiency of the usual dynamic-programming approach while maintaining a top-down strategy.
- The idea is to ***memoize*** the the natural, but inefficient, recursive algorithm.
- We maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.

Elements of dynamic programming cont..

- **Memoization (cont.)**
- An entry in a table for the solution to each subproblem is maintained.
- Eech table entry initially contains a special value to indicate that the entry has yet to be filled.
- When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table.
- Each subsequent time that the problem is encountered, the value stored in the table is simply looked up and returned.

Elements of dynamic programming cont..

```
MEMOIZED-MATRIX-CHAIN(p)
n←length[p]-1
for i←1  to  n
     do for j←i  to  n
          do  m[i,j] ←∞
return LOOKUP-CHAIN(p,1,n)
```


Elements of dynamic programming cont..

Memoization (cont.)

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