G205 Fundamentals of Computer Engineering

CLASS 20, Wed. Nov. 17 2004

Stefano Basagni

Fall 2004

M-W, 1:30pm-3:10pm

Dynamic Programming (DP)

- Algorithm design technique
- Programming → tabular method (not code writing)
- Similar to Divide & Conquer:
  - Solve a problem by combining solutions to some problems

Memento: Divide & Conquer

- Partition the problem into independent sub-problems
- Solve the sub-problems recursively
- Combine their solution to obtain the solution to the original problem
- "Kind of DULL" in the recursive division (e.g., Merge-Sort)

Dynamic Programming, 2

- To be used when the sub-problems are NOT independent
  - Sub-problems share sub-problems
  - D&C does more work than necessary
- DP solves the common sub-problems once and store the results in a table, to be re-used later

D&C vs. DP: An example

- Binomial Coefficient
  - The number of ways of choosing k objects from n: $C(n,k)$
- Recursive Definition

$$C(n,k)=C(n-1,k-1)+C(n-1,k)$$

with $k \le n$. It is also stipulated that

$$C(n,k) = 1$$

when $k \le 0$

Binomial Coefficient: Recursive Implementation

- Natural recursive implementation according to Divide & Conquer

```
RC(n,k)
  if k ≤ 0 or k = n
    return 1
  else
    return C(n-1,k-1) + C(n-1,k)
```

RC(n,k): Complexity

- The final result is computed by adding up a "bunch" of 1s
- How many? C(n,k) (number of recursive calls)
- Hence $T_{C(n,k)}(n) \in \Omega(C(n,k))$ $(k \leq n)$: Exponential!
- $T_{C(n,k)}(n)=2\ T_{C(n,k)}(n-1)+1 \in O(2^{n+1})$
- Why? Many C(i,j)s are computed over and over again: C(5,3)=C(4,2)+C(4,3) and both require C(3,2) …

Binomial Coefficient: Iterative Implementation

- DP approach: Store in a table the values that are computed more than once
  - They are accessed in O(1) time when needed
- The “table” is also known as the Pascal triangle (after Blaise Pascal, 1653 … Probability theory, … See Knuth “opera magna”)

Pascal Triangle

	0	1	2	3	4	5	…	k-1	k
0	1								
1	1	1							
2	1	2	1						
…									
n-1	…							C(n-1,k-1)	+ C(n-1,k)
n									= C(n,k)

Computing C(n,k)

```
IC(n,k)
    array c[0..n, 0..k]
    for i = 0 to n do c[i,0] =1
    for i = 0 to k do c[i,i] =1
    for i = 2 to n do
      for j = 1 to i-1 do
        c(i, j) = c(i-1, j) + c(i-1, j-1)
    return c(n,k)
```

IC(n,k) : Complexity

- Table initialization costs $\Theta(n)$
- Filling the table costs $\Theta(nk)$
- Total cost: $T_{IC(n,k)} \in \Theta(nk) = \Theta(n^2)$
- Space requirements: $\Theta(n^2)$
- EXERCISE: Implement IC(n,k) so that is only uses linear space

DP and Optimization Problems

- DP is useful for Optimization Problems
- OPs are problems with multiple solutions
  - Each solution has a value
  - We want the solution with the min or max value (optimal solution)

DP Algorithm Construction, 1

- 4 major 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 "bottom-up"
  4. Construct an optimal solution

DP Algorithm Construction, 2

- Steps 1-3 are the basis of DP
- Step 4 can be omitted if only the value of an optimal solution is required
- For step 4 usually additional structures are needed
- Example: All-Pair Shortest-Paths
  - For step 4 we have also the matrix Π

Elements of DP

- Two (three) key ingredients for an optimization problem to admit a DP solution

1. Optimal substructure
2. Overlapping sub-problems
3. (Memoization)

Optimal Substructure

- Characterize the structure of an optimal solution
- A problem has optimal substructure if an optimal solution contains optimal solution to sub-problems
- The optimal solution is built from optimal solutions to the sub-problems

Overlapping Sub-problems

- Space of sub-problems should be "small"
  - Few problems to be solved over and over again rather than new problems
  - The sub-problems are overlapping
- The total number of sub-problems is typically polynomial in the input size

Matrix-Chain Multiplication

- Input: A sequence (chain) of matrices to be multiplied:

$$\langle A_1, A_2, \ldots, A_n \rangle$$

- Output: The product

$$A_1 A_2 \ldots A_n$$

- We can use regular multiplication of pairs of matrices after we have parenthesized the chain to establish the order of multiplication

Parenthesizing Matrix Chains

- A product of matrices of fully parenthesized if and only if
  - It is a single matrix
  - It is the product of two fully parenthesized matrix products (in parenthesis)
- Example: Given $<A_1,A_2,A_3,A_4>$ the product can be fully parenthesized in 5 ways …

Example: <A_1,A_2,A_3,A_4>

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

- Since matrix product is associative all parenthesizations yields the same result

Matrix Multiplication

- Consider compatible matrices A(p x q) and B(q x r)

```
Matrix-Multiply(A,B)
  array c[1..p,1..r]
  for i = 1 to p do
    for j = 1 to r do
      c[i,j] = 0
      for k = 1 to q do
        c[i,j]=c[i,j]+A[i,k]B[k,j]
  return C
```

Cost of Matrix-Chain Multiplication

- Matrix-Multiply costs O(pqr)
- Different parenthesization yields different costs:
  - $<A_1, A_2, A_3>$ with $A_1(10x100)$, $A_2(10x5)$ and $A_3(5x50)$
  - $((A_1, A_2)A_3)$ → 7500 multiplications
  - $(A_1(A_2A_3))$ → 75000 multiplications
- One order or magnitude faster

The Matrix-Chain Multiplication Problem

- Given a chain of n matrices $<A_1, A_2, \ldots, A_n>$ with $A_i(p_i, p_{i+1})$ fully parenthesize the product $A_1 A_2 \ldots A_n$ so that the number of scalar multiplication is minimized
- The solution is the ORDER of the multiplication (→ the lowest cost), NOT the multiplication per se

An Inefficient Solution

- Check all possible parenthesizations and chose the one with lowest cost
- P(n) = number of different parenthesizations of a chain of n matrices

$$P(1) = 1$$

$$P(n) = \mathrm{SUM}(k=1,n-1)P(k)P(n-k),\ n \geq 2$$

- $P(n) \in \Omega(2^n)$ (exponential!)

A DP Solution: Optimal Substructure

- The structure of an optimal parenthesization is as follows
- Suppose that an optimal parenthesization of $A_i A_{i+1} \ldots A_j$ splits the product between A_k and A_{k+1}. Then the parenthesization of the prefix sub-chain $A_i \ldots A_k$ and of the postfix sub-chain $A_{k+1} \ldots A_j$ are optimal (by contradiction)

Optimal Substructure, 2

- Any solution requires to split the product at a certain point
- Sub-products must be optimal for obtaining an optimal solution
- We must ensure to find the correct place to split the product

Recursive Solution

- We define the cost of an optimal solution in terms of the optimal solution to sub-problems
- Let m[i,j] the minimum number of scalar multiplications for multiplying $A_i A_{i+1} \ldots A_j$
  - $m[i,i]=0$
  - $m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$
- We do not know k, but we can check all of them:
  - $\mathrm{Min}_{(i \le k < j)}\{m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}$, $i < j$

Computing the Optimal Cost

- We have to write an algorithm to compute m[1,n], which is what we want
- We have relatively few sub-problems: One problem for each choice of i and j, $1 \le i \le j \le n$: $O(n^2)$
- DP → Tabular, bottom-up approach

Computing the Optimal Cost, 2

```
Matrix-Chain-Order(p=<p_0,p_1,…,p_n>)
  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=1 to j-1 do
        q= m[i,k]+m[k+1,j]+p_{i-1}p_kp_j
        if q < m[i,j] then m[i,j]=q
                           s[i,j]=k
  return m and s
```

Matrix-Chain-Order Complexity

- The three nested for → $O(n^3)$
- In fact the algorithm is also $\Omega(n^3)$
- Space requirements → $\Theta(n^2)$ to store m and s
- Much more efficient of the exponential "exhaustive search" solution (enumerating all possible parenthesizations)

Assignments

- Textbook, Chapter 15, pages 323—347
- Updated information on the class web page:

  www.ece.neu.edu/courses/eceg205/2004fa