EECS 3101 M: Design and Analysis of Algorithms

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/3101M`
Also on Moodle

Optimal Matrix Multiplication

- Recall: Two matrices, A: $n \times m$ matrix, and B: $m \times k$ matrix, can be multiplied to get C with dimensions $n \times k$, using nmk scalar multiplications
- Matrix multiplication is associative: $(AB)C = A(BC)$
- Order of multiplication affects efficiency:
  e.g.: $A_1 = 20x30$, $A_2 = 30x60$, $A_3 = 60x40$,
  $((A_1 A_2) A_3) : 20x30x60 + 20x60x40 = 84000$
  $(A_1(A_2 A_3)) : 20x30x40 + 30x60x40 = 96000$
- Problem: compute $A_1 A_2 \ldots A_n$ using the fewest number of multiplications

Alternative View: Optimal Parenthesization

- Consider $A \times B \times C \times D$, where A is 30×1, B is 1×40, C is 40×10, D is 10×25
- Costs:
  - $(AB)C)D = 1200 + 12000 + 7500 = 20700$
  - $(AB)(CD) = 1200 + 10000 + 30000 = 41200$
  - $A((BC)D) = 400 + 250 + 750 = 1400$
- We need to optimally parenthesize $A_1 \times A_2 \times \ldots \times A_n$, where A_i is a $d_{i-1} \times d_i$ matrix

Optimal Parenthesization: Details

Let $M(i, j)$ be the minimum number of multiplications necessary to compute $\prod_{k=i}^{j} A_k$

Observations:

- The outermost parenthesis partition the chain of matrices (i, j) at some k, $(i \leq k < j)$: $(A_i \dots A_k)(A_{k+1} \dots A_j)$
- The optimal parenthesization of matrices (i, j) has optimal parenthesizations on either side of k, i.e., for matrices (i, k) and $(k + 1, j)$
- Since we do not know k, we try all possible values

Optimal Parenthesization: Details - 2

Recurrence:
$M(i, i) = 0$, and for $j > i$,
$M(i, j) = \min_{i \leq k < j}\{M(i, k) + M(k + 1, j) + d_{i-1} d_k d_j\}$

- A direct recursive implementation takes exponential time – there is a lot of duplicated work (why?)
- But there are only $\binom{n}{2} + n = \Theta(n^2)$ different sub-problems (i, j), where $1 \leq i \leq j \leq n$
- Thus, it requires only $\Theta(n^2)$ space to store the optimal cost $M(i, j)$ for each of the sub-problems: about half of a 2-d array $M[1..n, 1..n]$.

Optimal Parenthesization: Details - 3

Steps of the solution

- Which array element has the final solution? $M[1, n]$
- Which array elements can be initialized directly? $M[i, i]$ for $1 \leq i \leq n$
- What order should the table be filled?
  Tricky: the RHS of the recurrence must be available when LHS is evaluated
  So, the table must be filled diagonally

Optimal Parenthesization: Details - 4

Algorithm: Starting with the main diagonal, and proceeding diagonally, fill the upper triangular half of the table

- Complexity: Each entry is computed in $O(n)$ time, so $O(n^3)$ algorithm. Argue that it is $\Theta(n^3)$
- A simple recursive algorithm $Print - Optimal - Parenthesization(c, i, j)$ can be used to reconstruct an optimal parenthesization.
  For this need to record the minimum k found for each table entry
- Can also use memoized recursion

Exercise: Hand run the algorithm on $d = [10, 20, 3, 5, 30]$

Comments about Dynamic Programming

- Compute the value of an optimal solution in a bottom-up fashion, so that you always have the necessary sub-results pre-computed (or use memoization)
- Construct an optimal solution from computed information (which records a sequence of choices made that lead to an optimal solution)
- Let us study when this works

When does Dynamic Programming Work?

To apply dynamic programming, we have to:

- Show optimal substructure property – an optimal solution to the problem contains within it optimal solutions to sub-problems
- This is a subtle point. It involves taking an optimal solution and checking that subproblems are solved optimally
- The easiest way is to use a “cut-and-paste” argument
- Best seen through examples

Longest Common Subsequence (LCS)

Background:

- Computing the similarity between strings is useful in many applications and areas: e.g. spell checkers, test retrieval, bioinformatics
- Different applications require different notions of similarity
- The longest common subsequence is one measure of similarity
- Dynamic programming is useful for computing other measures as well

LCS : definitions

- Z is a subsequence of X, if it is possible to generate Z by skipping zero or more characters from X
- For example: $X =$ "$ACGGTTA$", $Y =$ "$CGTAT$", $LCS(X, Y) =$ "$CGTA$" or "$CGTT$"
- To solve a LCS problem we have to find "skips" that generate $LCS(X, Y)$ from X, and "skips" that generate $LCS(X, Y)$ from Y

LCS: Optimal Substructure

Subtle point: depends on the definition of subproblems.
Here we define $LCS(i,j)$ as the subproblem – this is the LCS of $X[1..i]$, $Y[1..j]$

- Let $Z[1..k]$ be the LCS of of $X[1..m]$ and $Y[1..n]$
- If $X[m] = Y[n]$, then $Z[k] = X[m] = Y[n]$. Is $Z[1..(k-1)]$ an LCS of $X[1..(m-1)]$, $Y[1..(n-1)]$, i.e., $LCS(m-1, n-1)$?
- If $X[m] \neq Y[n]$ and $Z[k] \neq X[m]$, then $Z = LCS(m-1, n)$?
- If $X[m] \neq Y[n]$ and $Z[k] \neq Y[n]$, then $Z = LCS(m, n-1)$?
- "Cut-and-paste" argument in each of the last 3 steps

LCS: Recurrence

Let $c[i,j] = |LCS(i,j)|$
$c[i,j] = 0$ if $i = 0$ or $j = 0$
$c[i,j] = c[i-1, j-1] + 1$ if $i, j > 0$ and $X[i] = Y[j]$
$c[i,j] = \max(c[i-1, j], c[i, j-1])$ if $i, j > 0$ and $X[i] \neq Y[j]$

- Order of filling cells?
- Complexity?
  Constant work per cell
- Actual LCS can be generated by remembering which choice gave the maximum, as before

Exercise:Compute LCS of $X = \text{"}ACGGTTA\text{"}, Y = \text{"}CGTAT\text{"}$