15-451: Algorithms

Lecture 12: Dynamic Programming (notes KS)

1 Dynamic Programming

Many algorithms are based on solving large problem instances by systematically solving smaller subinstances. Typical example: divide-and-conquer as used in quicksort and mergesort. The crucial point here is that the subinstances are independent of each other.

Problems arise when the subinstances are not independent: two different subinstances may well contain the same sub-subinstance. Have to carefully keep track of already computed answers, otherwise there will be exponential blowup.

Two main methods to keep track of the subinstances, and their corresponding solutions.

- Memoizing: store partial results in hashtable for later reuse.
- Dynamic programming: construct a table of partial results.

Memoizing is easier from the user's point: just use a recursive function. Incurs overhead. If the subinstances follow some nice pattern, it is more efficient is to construct a table.

2 Matrix Chains

Let A be an $p \times q$ matrix, and B a $q \times r$ matrix. Their product is defined to be the $p \times r$ matrix C where

$$C(i,j) = \sum_{k=1}^{q} A(i,k) \cdot B(k,j)$$

Brute force calculation is $\Theta(pqr)$. Can use divide-and-conquer for speed-up, but we won't get involved at this point. Since matrix multiplication is *associative* we can evaluate a long product

$$A_1 \cdot A_2 \cdot A_3 \ldots A_{n-1} \cdot A_n$$

by placing parentheses wherever we like.

How many ways are there to fully parenthesize this product? Equivalently: How many full binary trees on n leaves are there? Need to solve the following convolution recurrence:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{i=1}^{n-1} P(i)P(n-i) & \text{otherwise.} \end{cases}$$

Has solution $P(n) = C_{n-1}$ (Catalan numbers):

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \Omega(n^{-3/2}4^n)$$

Suppose we have matrices $A_1, A_2, \ldots, A_{n-1}, A_n$ of size $p_0, p_1, \ldots, p_n$, so that A_i has dimension $p_{i-1} \times p_i$. Choosing a particular parenthesization affects the total number of multiplication.

Example:

$$A_1 : 10 \times 100$$
$$A_2 : 100 \times 5$$
$$A_3 : 5 \times 50$$

$$(A_1 A_2) A_3 : 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$$
$$A_1 (A_2 A_3) : 100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$$

This is vaguely reminescent of Huffman coding, or merging a collection of already sorted lists. But there are important differences: we must keep the matrices in order. We can choose the shape of the tree, but not the leaf-order.

Also, the cost function is more complicated: If A is $p \times q$ matrix, B $q \times r$ then the cost of multiplying them is $p \cdot q \cdot r$ and the result has dimension $p \times r$. As a consequence, the simple Greedy Algorithm in Huffman coding won't work here.

Suppose

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

is an optimal parenthesization. Total cost:

$$cost(1,k) + cost(k+1,n) + p_0 \cdot p_k \cdot p_n.$$

Then clearly both $(A_1 \cdot A_2 \cdots A_k)$ and $(A_{k+1} \cdots A_{n-1} \cdot A_n)$ must be optimal.

2.1 First Attack: Memoizing

Compute optimal parenthesization recursively, top-down.

Let $C(i,j)$ be the optimal number of multiplications for $A_i \cdot A_{i+1} \cdots A_{j-1} \cdot A_j$, where $1 \leq i \leq j \leq n$.

We need $C(1,n)$. Recursion is easy:

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

Must avoid multiple calls to $C(i,j)$: memoize.

2.2 Second Attack: Dynamic Programming

Bottom-up approach: compute an array `C[i,j]`.

For $i = j$ can set $C(i,i) = 0$.

Then compute all $C(i,j)$ for $d = j - i$, $d = 1, 2, \ldots, n-1$.

```
for d = 1,..,n−1 do
for i = 1,..,n−d do
   j = i + d;
   for k = i,..,j−1 do
   {
      c = C[i,k] + C[k+1,j] + p[i−1] p[k] p[j];
      if( c < C[i,j] )
         C[i,j] = c;
   }
```

LHS: `C[i,j]`

RHS: `C[i,k]` and `C[k+1,j]` where $i \leq k < j$.

But then $j - i = d$ and $k - i, j - (k+1) < d$. Hence we can prove correctness by induction on d. Running time is clearly $O(n^3)$ and in fact $\Theta(n^3)$.

Table requires more thought, but is slightly faster than recursion and memoizing.

3 Longest Common Subsequence

Consider a sequence $A = (a_1, a_2, \ldots, a_n)$. $B = (b_1, b_2, \ldots, b_k)$ is a *subsequence* of A iff there is an index sequence $1 \leq p_1 < p_2 < \ldots < p_k \leq n$ such that

$$a_{p_i} = b_i.$$

Note: scattered subsequence, need not be contiguous. Given two sequences A and B, S is a *longest common subsequence (LCS)* iff S is a subsequence of A and of B, and has maximal length.

Example:

$A = (a, b, c, b, d, a, b)$,

$B = (b, d, c, a, b, a)$.

$S = (b, d, a, b)$ or $S = (b, c, b, a)$.

For dynamic programming we need a way to solve LCS for A and B by solving subproblems. Let A_i be the prefix of A of length i. $A = (a_1, a_2, \ldots, a_m)$, $B = (b_1, b_2, \ldots, b_n)$, $S = (s_1, s_2, \ldots, s_k)$ a LCS of A and B.

It is easy to get a recursive description of LCSs:

- $a_m = b_n$ implies $s_k = a_m$, S_{k-1} is LCS of A_{m-1} and B_{n-1},
- $a_m \neq b_n$ and $s_k \neq a_m$ implies S is LCS of A_{m-1} and B,
- $a_m \neq b_n$ and $s_k \neq b_n$ implies S is LCS of A and B_{n-1}.

Let $a \neq b$ and write Aa for a sequence whose last term is a.

$$LCS(Aa, Ba) = LCS(A, B)\, a$$
$$LCS(Aa, Bb) = LCS(A, Bb) \text{ or } LCS(Aa, B)$$

LCS is not really a function, but a multi-valued function.

3.1 Memoizing: top-down

Compute the length of a LCS only. Define

$$C(i, j) = \text{length of an LCS for } A_i \text{ and } B_j.$$

then

$$C(i,j) = \begin{cases} 0 & \text{if } i \cdot j = 0, \\ C(i-1, j-1) + 1 & \text{if } a_i = b_j, \\ \max\big(C(i, j-1), C(i-1, j)\big) & \text{otherwise.} \end{cases}$$

In conjunction with memoizing this is produces an $\Theta(nm)$ algorithm. Note: have either one or two recursive calls.

3.2 Dynamic programming: bottom-up

Compute array C, initialize $C(i, 0) = 0$ and $C(0, j) = 0$.

```
for( i = 1; i <= m; ++i )
for( j = 1; j <= n; ++j )
   if( a[i]==b[j] )
      C[i,j] = C[i-1,j-1] + 1;
   else
      C[i,j] = max( C[i-1,j], C[i,j-1] );
```

Note: all values on right hand side really are already computed. Running time is trivially $\Theta(nm)$.

4 All Pairs Shortest Path

Given a directed graph and a positive weight function for the edges, we want to compute the distance between any two vertices. It is less than clear what the structure of the subinstances should be in this case. Clever trick:

$$dist(a, b, k) = \text{ distance using only intermediate points} \leq k$$

Assume vertices $1, 2, \ldots, n$.

- $dist(a, b, 0)$ is the cost of edge (a, b), or infinity
- $dist(a, b, n)$ is the real distance from a to b.

Now we get a recursive description:

$$dist(i, j, k) = \min\big(dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1)\big);$$

Immediately yields an algorithm:

```
for( k = 1; i <= n; k++ )
for( i = 1; i <= n; i++ )
for( j = 1; i <= n; j++ )
    dist[i,j,k] = min( dist[i,j,k−1], dist[i,k,k−1] + dist[k,j,k−1] );
```

Analysis is straightforward: $\Theta(n^3)$, with small constants. Note that space requirement is $\Theta(n^2)$: only need level $k-1$ to get level k.

5 Traveling Salesman

Dynamic programming may also be used to speed up computations without producing really efficient algorithms. The TSP is a notorious example. Brute force computation is $\Theta(n \cdot n!)$, though one could shave off the n term by generating all permutations through transpositions. A better, but still exponential algorithm goes like this. Assume the cities are $[n]$, and a distance function $d : [n] \times [n] \to \mathbb{R}$.

For all $i \in S \subseteq [n]$ let

$$C(S, i) = \text{ shortest Hamiltonian path } 1 \xrightarrow{S} i$$

where by $1 \xrightarrow{S} i$ we mean a Hamiltonian path that starts at 1, ends at i, and touches every point in S.

$$C(\{i\}, i) = d(1, i)$$
$$C(S \cup \{j\}, j) = \min\big(\, C(S, i) + d(i, j) \bigm| i \in S \,\big)$$

where in the second equation $j \notin S$.

Then the shortest tour is given by

$$\min\big(\, C([2:n], i) + d(i, 1) \bigm| i \in [2:n] \,\big)$$

We have to consider $\Theta(n \cdot 2^n)$ pointed subsets of $[2:n]$, so running time is $\Theta(n^2 \cdot 2^n) = o(n!)$.