CSC 373 H 1 Y — Summer 2007

University of Toronto — St. George Campus

Lecture Summary for Week 3

This summary is not a replacement for the lecture. If you miss a class, please arrange with a friend to take note for you.

2.1 Longest Common Subsequence (continued)

The algorithm LCS2 presented last week uses an $n \times m$ array S to store the solutions to the subprolems; each entry $S[i, j]$ of S is a longest common subsequence of $X[1 \ldots i]$ and $Y[1 \ldots j]$. So S is essentially a 3-dimensional array of size $n \times m \times n$.

It turns out that the longest common subsequence of X and Y can be computed from the lengths of $S[i, j]$. So we only need to store these lengths (as opposed to the actual sequences). In other words, we need only a 2-dimensional array L of size $n \times m$ where each entry is a natural number $\leq \min(n, m)$. The length $L[i, j]$ of $S[i, j]$ satisfies the following initialization and recurrence relation:

$$L[0, j] = L[i, 0] = 0 \qquad \text{for } 0 \leq i \leq n, 0 \leq j \leq m \tag{1}$$

$$L[i, j] = \begin{cases} L[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ \max\{L[i-1, j], L[i, j-1]\} & \text{otherwise} \end{cases} \tag{2}$$

The following program computes L:

1. L: $n \times m$ array
2. For $j = 1$ to m do $L[0, j] \leftarrow 0$
3. For $i = 1$ to n do $L[i, 0] \leftarrow 0$
4. For $i = 1$ to n do
5. For $j = 1$ to m do
6. If $X[i] = Y[j]$ do $L[i, j] \leftarrow L[i-1, j-1] + 1$
7. Else $L[i, j] \leftarrow$ max of $\{L[i-1, j], L[i, j-1]\}$
8. End If
9. End For
10. End For

Now $L[n, m]$ is the length of a longest common subsequence of X and Y. We will compute a longest common subsequence of X and Y by looking at how $L[n, m]$ is obtained from $L[i, j]$ for $0 \leq i \leq n$, $0 \leq j \leq m$. The following program computes a longest common subsequence of X and Y using L:

1. $\ell \leftarrow L[n, m]$ % length of a longest common subsequence of X and Y.
2. Z: array of length ℓ % the output common subsequence
3. $i \leftarrow n$, $j \leftarrow m$
4. While $l > 0$ do
5. If $X[i] = Y[j]$ then
6. $Z[\ell] \leftarrow X[i]$
7. $i \leftarrow i - 1, j \leftarrow j - 1, \ell \leftarrow \ell - 1$

8. Elseif $L[i,j] = L[i-1,j]$ then
9. $i \leftarrow i-1$
10. Else $j \leftarrow j-1$
11. End If
12. End While
13. Output Z.

Over all, the running time for computing a longest common subsequence of X and Y is $\mathcal{O}(nm)$. For space complexity: we use an $n \times m$ array of natural numbers $\leq \min(n,m)$.

2.2 Dynamic Programming Technique

The problem of computing a longest common subsequence that we have seen can be solved recursively, where the solutions of subproblems are re-used many times. Many other problems have this property, and they can be solved in the same way, i.e., solving subproblems in the bottom-up fashion and storing their solutions to avoid re-computation.

To solve the Longest Common Subsequence problem, we first solve the related problem of computing the length of a longest common subsequence. (This problem also has the property we mention above.) The advantage is some saving on the time and space complexity. In general, we may have to make a detour to another related but simpler problem (e.g. computing the length of a longest common subsequence, instead of computing a longest such common subsequence), and use the solution to this simpler problem to obtain the solution to the original problem (e.g., using L to compute a longest common subsequence of X and Y).

Here are four steps in designing an algorithm using the dynamic programming technique:

1. Define an array for solving the (related) problem. (For example, the array L in the LCS problem.)
2. Give the initialization and recursive formulas for computing the elements of the array. (For example, the initial values in (1) and recursive formula in (2).)
3. Give a program for computing the array.
4. Give a program for computing the solution to our problem using the array.

We will see more examples in this lecture and the next.

2.3 The Subset Sum Problem [Section 6.4]

Input A set of n items $\{1,\ldots,n\}$, where item i has nonnegative integer weight $w(i)$, and a bound W.

Output A subset S of the items with maximum total weight which is $\leq W$. That is, $S \subseteq \{1,\ldots,n\}$ so that

$$\sum_{i \in S} w(i) \leq W$$

and $\sum_{i \in S} w(i)$ is maximum.

There are 2^n subsets of $\{1,2,\ldots,n\}$. Here we will see how dynamic programming technique can help to avoid going through all these 2^n subsets.

The problem can be solved recursively by looking at the subproblems specified by the subset of items $\{1, 2, \ldots, i\}$ and a bound m where $m \leq W$. In other words, we look at the (sub)problems of computing a subset $S \subseteq \{1, 2, \ldots, i\}$ so that

$$\sum_{i \in S} w(i) \leq m$$

and $\sum_{i \in S} w(i)$ is maximum.

There are optimal solutions $OPT[i, m]$ to the above problems that satisfy:

$$OPT[0, m] = 0$$

$$OPT[i, m] = \begin{cases} OPT[i-1, m] & \text{if } m < w(i) \\ \text{“better of } OPT[i-1, m], OPT[i-1, m-w(i)] \cup \{i\}\text{”} & \text{if } m > w(i) \end{cases}$$

To compute $OPT[n, W]$, each $OPT[i, m]$ may be used many times. So we will use the dynamic programming technique. Also, instead of storing the subsets $OPT[i, m]$, we will only store their total weights in an $n \times W$ array A. Once all entries $A[i, m]$ have been computed, the optimal subset $S = OPT[n, W]$ will be computed by examining how $A[n, W]$ is obtained from other entries $A[i, m]$. Here are the four steps of designing an algorithm using the Dynamic Programming technique for this problem:

1. Let A be an $n \times W$ array, where each entry $A[i, m]$ is the total weight of an optimal solution for the subproblem defined by the set of items $\{1, \ldots, i\}$ and bound m. (Here $0 \leq i \leq n, 0 \leq m \leq W$.)

2. Initial values and recurrence:

$$A[0, m] = 0 \text{ for } m \leq W,$$

$$A[i, m] = \begin{cases} A[i-1, m] & \text{if } w(i) > m \\ max\{A[i-1, m], w(i) + A[i-1, m-w(i)]\} & \text{otherwise} \end{cases}$$

3. Program:

```
1. For m = 0 to W do A[0, m] ← 0
2. For i = 1 to n do
3.     For m = 0 to W do
4.         If w(i) > m then A[i, m] ← A[i − 1, m]
5.         Else A[i, m] ← max{A[i − 1, m], w(i) + A[i − 1, m − w(i)]}
6.         End If
7.     End For
8. End For
```

4. Computing an optimal solution using A:

```
1. S ← ∅   % solution
2. i ← n, m ← W
3. While i ≥ 0 do
4.     If A[i, m] = A[i − 1, m] then i ← i − 1
5.     Else
6.         S ← S ∪ {i}
```

7. $m \leftarrow m - w(i)$
8. $i \leftarrow i - 1$
9. End If
10. End While
11. Return S.

The running time of the above algorithm is $\mathcal{O}(nW)$. Note that this is NOT a polynomial in the length of the input W.

2.4 Matrix Chain Multiplication Problem [CLRS Section 30.2]

For this problem, assume that for an $n \times k$ matrix A and a $k \times m$ matrix B, computing $A \times B$ requires $\Theta(nkm)$ operations.

Matrix multiplication is associative, i.e., for matrices A, B, C of appropriate size,

$$(A \times B) \times C = A \times (B \times C)$$

So we can write $A \times B \times C$ to mean the above product. In general, for a sequence of matrices of appropriate size

$$A_1, A_2 \ldots, A_n$$

the product $A_1 \times A_2 \times \ldots \times A_n$ is uniquely defined. However, the order of carrying out this multiplication is important, because it determines the number operations needed.

For example, suppose that A is a 1×10 matrix, B is 10×10, and C is 10×100. Then by carrying out the multiplication in the order of

$$(A \times B) \times C$$

we use

$$1 \times 10 \times 10 + 1 \times 10 \times 100 = 1100$$

operations. On the other hand, if we compute

$$A \times (B \times C)$$

then we need

$$10 \times 10 \times 100 + 1 \times 10 \times 100 = 11000$$

operations.

The Matrix Chain Multiplication problem is the following problem:

Input Given a sequence of matrices $A_1, \ldots, A_n$, where A_i has dimension $d_i \times d_{i+1}$, for $1 \leq i \leq n$.
Output An order of multiplying the matrices with smallest number of operations.

Notice that by the above assumption, multiplying A_i and A_{i+1} requires $\Theta(d_i d_{i+1} d_{i+2})$ operations.

Consider solving the problem recursively. We want to find the best way of parenthesizing the product

$$A_i \times \ldots \times A_j \qquad \text{for } i \leq j.$$

So let $B[i, j]$ be k such that

$$(A_i \times A_{i+1} \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_j)$$

is the best way of multiplying $A_i \times \ldots \times A_j$. To compute $B[i,j]$ we use a 2-dimensional array N where $N[i,j]$ is the smallest number of operations required for computing

$$A_i \times \ldots \times A_j$$

In step 3 below we compute B at the same time as we compute N. The running time for this step is $\Theta(n^3)$.

1. **Array:** $N[i,j]$ is the optimal cost of multiplying $A_i \times \ldots \times A_j$, for $i \leq j$.
2. **Initialization & Recurrence:** We look at the length of the subsequence $A_i, A_{i+1}, \ldots, A_j$. For the initialization, we consider the sequence of only one matrix:

   $$N[i,i] = 0 \qquad \text{for } 1 \leq i \leq n$$

   The recurrence:

   $$N[i,j] = min\{N[i,k] + N[k+1,j] + d_i d_{k+1} d_{j+1} \ : \ \text{for } i \leq k < j\}$$

   Here $d_i d_{k+1} d_{j+1}$ is the cost of multiplying the two products $(A_i \times \ldots \times A_k)$ and $(A_{k+1} \times \ldots \times A_j)$.
3. **Program:** We will compute the best break point $B[i,j]$ for the sequence $A_i, \ldots, A_j$, as well as the best cost $N[i,j]$. In the program below, the length of the subsequence under consideration is $\ell + 1$.

```
1. For i from 1 to n   do  n[i,i] ← 0  End For
2. For ℓ from 1 to n − 1   do
3.     For i from 1 to n − ℓ   do   % consider the sequence A_i × ... × A_{i+ℓ}
4.         j ← i + ℓ
5.         N[i,j] ← ∞
6.         For k from i to j − 1 do
7.             v ← N[i,k] + N[k+1,j] + d_i d_{k+1} d_{j+1}
8.             If v < N[i,j] do
9.                 N[i,j] ← v
10.                B[i,j] ← k   % break point
11.            End If
12.        End For
13.    End For
14. End For
```

4. **Computing the solution:** Here is a way of computing $A_1 \times \ldots \times A_n$ using the array B. We call $\text{MM}(A, B, 1, n)$, where MM is the program:

   MM(A, B, i, j):

```
1. If i = j return A_i
2. Else
3.     k ← B[i,j]
4.     M_1 ← MM(A, B, i, k)
5.     M_2 ← MM(A, B, k+1, j)
6.     return M_1 × M_2
7. End If
```