ICS 260—Fall, 2000–Class Notes 9: Dynamic Programming, Continued

Notes Originally Developed by Sandy Irani
Modified by Michael B. Dillencourt

This material is taken from [CLR 16.1-16.3]

1 Longest Common Subsequence

Consider finite length sequences over a finite alphabet Σ. Given a sequence $A = a_1 a_2 \ldots a_m$, $C = c_1 c_2 \ldots c_k$ is a subsequence of A if there is a strictly increasing sequence $i_1, i_2, \ldots, i_k$ such that $a_{i_j} = c_j$.

Two sequences A and B have a common subsequence C, if C is a subsequence of both A and B. Given two sequences A and B, we would like to find the longest common subsequence of the two sequences.

Another way of looking at the longest common subsequence problem is that we want to align the sequence so that we get the most overlap. That is, we want to align the two sequences so that if a_i is aligned with b_j, then $a_i = b_j$. We want to align the sequences so that we can find the highest degree of overlap. This is essentially what the "diff' operator does.

Yet another way of looking at this problem is that we want to find the distance between the two sequences. The distance between two sequences is the number of operation required to transform one sequence into the other sequence, where the operations which can be performed is to delete an element or to insert an element. The distance between the two sequences is $|A| + |B| - (2 \cdot \text{length of lcs})$, where $|X|$ denotes the length of a sequence X.

Let $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots, b_n$. We would like to find an algorithm which determines the longest common subsequence of A and B. Fortunately, the LCS problem has the optimal-substructure property.

Let A_i denote the *prefix* of A consisting of the first i elements: $A_i = a_1, a_2, \ldots, a_i$.

Theorem 1 *Let $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots, b_n$ be two sequences, and let $C = c_1, c_2, \ldots, c_k$ be any LCS of A and B. Then*

1. *If $a_m = b_n$, then $c_k = a_m = b_n$ and C_{k-1} is an LCS of A_{m-1} and B_{n-1}.*
2. *If $a_m \neq b_n$, then if $a_m \neq c_k$, C is an LCS of A_{m-1} and B.*
3. *If $a_m \neq b_n$, then if $b_n \neq c_k$, C is an LCS of A and B_{n-1}.*

Proof.

1. Suppose that $a_m = b_n$ and $c_k \neq a_m$. Then we can extend C by adding a_m to the end of the sequence. This contradicts the optimality of C. This tells us that $c_k = a_m = b_n$.

Now suppose that C_{k-1} is not an LCS for A_{m-1} and B_{n-1}. If there were a longer subsequence C' pf A_{m-1} and B_{n-1}, then $C'c_k$ would be a subsequence of A and B which would contradict the optimality of C.

2. If $c_k \neq a_m$, then C is a subsequence of both A_{m-1} and B. If there were a longer common subsequence for A_{m-1} and B, then there would be a longer common subsequence for A and B.

3. Analogous to (2).

■

The theorem gives rise to the following recurrence. Let $c[i,j]$ denote the length of the LCS of A_i and B_j.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } a_i = b_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } i,j > 0 \text{ and } a_i \neq b_j \end{cases}$$

Solving the recurrence recursively, would result in an exponential time algorithm because the algorithm would be solving the same subproblem repeatedly. With dynamic programming, we will fill in a chart to the c values. We will also fill a chart in with directions on how to construct the actual subsequence (and not just the length of the subsequence).

```
Procedure DynProgLCS(A, B):
for i = 1 to n
    c[i, 0] ← 0
for j = 1 to m
    c[0, j] ← 0
for i ← 1 to n
    for j ← 1 to m
        if a_i = b_j
            c[i, j] ← c[i − 1, j − 1] + 1
            b[i, j] ← D
        else
            if c[i − 1, j] > c[i, j − 1],
                c[i, j] ← c[i − 1, j]
                b[i, j] ← U
            if c[i − 1, j] ≤ c[i, j − 1],
                c[i, j] ← c[i, j − 1]
                b[i, j] ← L
return c and m
```

Figure 1: A dynamic programming algorithm for finding the longest common subsequence of two sequences.

Running time: O(mn). Space O(mn). The space can be improved by just filling in the matrix one row at a time. Then we need only O(m) space, but can't recompute the actual LCS. Actually, can get space O(m+n), time O(mn) and reconstruct the sequence using a divide-and-conquer approach due to our very own Dan Hirschberg.

1.1 Applications

As we discussed earlier, the longest common subsequence problem can also be seen also as the string alignment problem, where two characters can only be aligned if they are identical. Moreover, the cost of aligning a non-blank character with a blank character is 1. We already mentioned one application of this problem which is the 'diff' utility. In this case, each character is a line of text. The diff utility, finds the best alignment and then outputs all the line insertions and deletions which must be performed to transform the first text into the second text.

One can imagine variants of this problem as follows: let Σ be the alphabet over which the sequence is defined. Let $\Sigma' = \Sigma \cup \{B\}$, where B is the blank character. Once can define an arbitrary cost function $c : \Sigma' \times \Sigma' \to \mathbf{R}$, where $c(x, y)$ indicates the cost of aligning x and y. We will assume that $c(B, B) = \infty$ in order to discourage padding the alignment with extra blank characters. We will say that the distance between two sequences X and Y (denoted $d(X, Y)$) is the total cost of the best alignment. If we use the following cost function, then $d(X, Y) = |X| + |Y| - (2 \cdot$ length of LCS). Here is the cost function:

$$c[x, y] = \begin{cases} 0 & \text{if } x = y \text{ and } x \neq B \\ \infty & \text{if } x = y = B \\ 1 & \text{if } x \neq y \text{ and either } x = B \text{ or } y = B \\ \infty & \text{if } x \neq y \text{ and } x \neq B \text{ and } y \neq B \end{cases}$$

In general, $d(A, B)$ can be computed using the following recurrence:

$$d[A_i, B_j] = \min\{d[A_{i-1}, B_{j-1}] + c[a_i, b_j], d[A_{i-1}, B_j] + c[a_i, B], d[A_i, B_{j-1}] + c[B, b_j]\}.$$

Sequence alignment is a very important problem in computational biology. Usually the sequences being aligned are DNA, RNA or amino acid sequences. If the sequences are DNA or RNA, then the alphabet consists of four base pairs $\{A, C, G, T\}$. In amino acid sequences, the alphabet consists of 22 amino acids.

Amino acid sequences fold into a three dimensional protein which presumably carries out some function in the organism. A fundamental goal in biology is to determine the function of a particular protein. One way to go about this is to align the sequence with all the sequences in a database of proteins for which the function is already known. If a given amino acid sequence is very close to that of a protein with a known function, then that gives a very good conjecture about the function of the protein in question. This conjecture can then be tested in the laboratory.

Furthermore, if it is known what part of the sequence with the known function is involved in the function, then the alignment allows the corresponding region to be identified in the unknown sequence, thereby providing a hypothesis about what part of the unknown sequence is involved in the function. Also, even if nothing is known about the function in either sequence, often the regions where the alignment is close identify functionally important regions of both sequences (because they are conserved, while unimportant regions diverge).

Similar techniques are also used for DNA sequences.

Also, alignment is sometimes used to conjecture three dimensional structure. By aligning a sequences to a sequence whose structure is known, one can place the sequence in question in a three dimensional configuration by identifying matching amino acids in their location in three space. This gives a reasonable start for determining the 3D structure of the new sequence.

Alignment is also used to determine the distance between DNA sequence is phylogenic analysis. Closer sequences are presumed to be more closely related in their evolution.

Sequence alignment is used for a host of other purposes as well. At this point, it is the most widely used algorithm in molecular biology.

The cost functions that are used, is the subject of debate among biologists. Some possibilities are:

- **Identity** Use the lcs cost function, where an amino acid are viewed simply as being the same or different.
- **Genetic Code:** assign costs that are related to the number of DNA changes required to convert the DNA triples that code for that amino acid. (Each amino acid is coded using three base pairs of DNA).
- **Chemical Similarity:** take into account the relative shapes, sizes and charges of different amino acids.
- **Experimental:** use empirical data which indicates the observed properties of amino acids and their observed effect on the tertiary structure of the protein.

2 General Dynamic Programming

Dynamic programming problems often fall into the two basic 'formats' which we have seen in the two examples, but it is also a more general technique.

We have already seen an example of dynamic programming. Recall that in the computation of the minimum mean cycle of a graph, we discussed Karp's algorithm which is based on the observation that the mean weight of the minimum mean weight cycle is

$$\min_{v \in V} \max_{0<k\leq n-1} \frac{D_n(v) - D_k(v)}{n-k},$$

where $D_k(v)$ is the length of the shortest path with exactly k edges from s to v. If no such path exists, then $D_k(v) = +\infty$. One can then compute the min mean weight cycle then by computing the $D_v(\cdot)$'s and then evaluating the above expression. The algorithm which we discussed is dynamic programming method for finding the $D_v(\cdot)$'s.

3 Greedy vs. Dynamic Programming

[CLR 17.2]

Both the greedy algorithm and dynamic programming use the optimal substructure property, so how can one determine when the greedy algorithm is sufficient and when dynamic programming is necessary?

We will examine a classical optimization problem called the *Knapsack Problem*. In this problem a thief is robbing a store and can only carry W pounds of merchandise out of the store. Each item i has a value v_i and a weight w_i. There are n possible items to take. The thief wants to maximize the total value of the objects he takes such that the total weight of the objects is at most W.

We will consider the **0-1** version of the problem where the thief has to either take the object or leave it in the store, in contrast with the fractional version of the problem where the thief is allowed to take a portion of an object.

Both problems have the optimal substructure property. If the optimal solution includes object j, If object j taken out of the knapsack, the remaining items are an optimal solution for the problem on the remaining $n-1$ items where $W - w_j$ pounds can be carried.

However, the greedy algorithm works for the fractional problem and does not work for the **0-1** problem. For the fractional problem, sort the items by decreasing value per pound (v_i/w_i). The thief will take as much of each item possible with the greatest value per pound. It is left as an exercise to prove that this is the greedy strategy.

This method doesn't work for the 0-1 version of the problem. Consider the following example. $W = 200$. There are two items, each with value \$200 and weight 100. Then there is a third item with value \$5 and weight 1. The best strategy for the 0-1 problem is to take the two heavier items. However, the greedy strategy will start with the lighter item and will only be able to fit in one heavier item.

For a dynamic programming version of the problem, for $0 \le i \le n$ and $0 \le w \le W$, let $v(i,w)$, be the maximum value that can be taken in a knapsack which can hold at most w pounds when selecting from items $1, 2, \ldots, i$. The solution to the problem is $w(n,W)$. A table will all the $v()$ values can be filled in using the following recurrence:

$$v(i,w) = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ v(i-1,w) & \text{if } w_i > w \\ \max\{v(i-1,w), v(i-1,w-w_i) + v_i\} & \text{otherwise} \end{cases}$$