Dynamic Programming: LCS (19)

1 REVIEW

1.1 Maximum Probability Segmentation
Recall how we developed an algorithm for this.

- We began with a recursive formulation: The best way to break up a sequence is the best combination of a first word with the best way to break up the remainder of the sequence.

- Then, we noticed that implementing this directly recursively would result in lots of wasted work.

- So, we decided to "cache" the results of our work and compute answers in reverse order to fill in the table. That way, whenever we need the solution to a subproblem, it's already in our table.

- Actually, a lot like solving problems on a DAG (since you can write the computational dependencies as a graph and work in reverse topological order).

- Running time was $O(n^2)$, since each of the $n$ table entries takes $O(n)$ to fill in.

2 LONGEST COMMON SUBSEQUENCE

2.1 Problem
Given a text file and a variation of the file, identify lines that have been deleted, inserted, or changed.
I use this all the time in the form of the UNIX diff command.

- source code control: efficiently store multiple versions of a large program by keeping changes as "diffs."

- collaborative authoring: focus attention on new edits.

- software distribution: send updates as "diffs" instead of resending entire tree.

- debugging: compare the output of a newly compiled program to the correct output (can be used for grading also).
2.2 Formal Definition

A sequence is a list \( X = \langle x_1, x_2, \ldots, x_m \rangle \) (e.g., \( \langle A, B, C, B, D, A, B \rangle \)).

A subsequence of \( X \) is an ordered sublist of \( X \) (e.g., \( \langle B, C, D, B \rangle \), but not \( \langle D, C, B \rangle \)).

A common subsequence of two sequences \( X = \langle B, D, C, A, B \rangle \) is a subsequence of both of them.

The LCS, or longest common subsequence of \( X \) and \( Y \) is, well, their longest possible common subsequence. What is it?

We’ll also use \( X_i \) to mean the \( i \)-element prefix of \( X \). So \( X_m = X \) if \( X \) is length \( m \).

2.3 Algorithmic Ideas

How would you solve this? Hint, it will involve filling in a table!

- Think of “optimal substructure” property (like when we talked about paths).
- Think of a recursive solution.

2.4 Optimal Substructure Theorem

Let \( X = \langle x_1, x_2, \ldots, x_m \rangle \) and \( Y = \langle y_1, y_2, \ldots, y_n \rangle \) be sequences, and let \( Z = \langle z_1, z_2, \ldots, z_k \rangle \) be any LCS of \( X \) and \( Y \).

- 1. If \( x_m = y_n \), then \( z_k = x_m = y_n \) and \( Z_{k-1} \) is an LCS of \( X_{m-1} \) and \( Y_{n-1} \).
- 2. If \( x_m \neq y_n \), then \( z_k \neq x_m \) implies that \( Z \) is an LCS of \( X_{m-1} \) and \( Y \).
- 3. If \( x_m \neq y_n \), then \( z_k \neq y_n \) implies that \( Z \) is an LCS of \( X \) and \( Y_{n-1} \).

2.5 Recursive Formula

Let \( c[i, j] \) be the length of the LCS of \( X_i \) and \( Y_j \) (prefixes).

\[
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0, \\
c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\
\max(c[i, j-1], c[i-1, j]) & \text{otherwise.}
\end{cases}
\]

2.6 Algorithm

\text{LCS-LENGTH}(X, Y)
1. \( m \leftarrow \text{length}[X] \)
2. \( n \leftarrow \text{length}[Y] \)
3. \textbf{for } \( i \leftarrow 1 \) \textbf{ to } \( m \)
4. \hspace{1em} \textbf{do } \( c[i, 0] \leftarrow 0 \)
5. \textbf{for } \( j \leftarrow 1 \) \textbf{ to } \( n \)
6. \hspace{1em} \textbf{do } \( c[0, j] \leftarrow 0 \)
7. \textbf{for } \( i \leftarrow 1 \) \textbf{ to } \( m \)
do for $j \leftarrow 1$ to $n$
  do if $x_i = y_j$
    then $c[i, j] \leftarrow c[i-1, j-1] + 1$
    $b[i, j] \leftarrow \text{"\"} \ 12$
    else if $c[i-1, j] \geq c[i, j-1]$
      then $c[i, j] \leftarrow c[i-1, j]$
      $b[i, j] \leftarrow \text{"\"} \ 14$
    else $c[i, j] \leftarrow c[i, j-1]$
      $b[i, j] \leftarrow \text{"\"} \ 16$
return $c$ and $b$

2.7 General Running-Time Analysis for Dynamic Programming

Nearly any DP algorithm can be analyzed by multiplying the size of the table by the time it takes to fill in a single cell of the table.

- segmentation: $n$ table entries, $O(n)$ time to fill in, $O(n^2)$ total.
- LCS: $nm$ table entries, $O(1)$ time to fill in, $O(nm)$ total.

2.8 Beam Search

In practice, it doesn’t make sense to fill in the whole table. Instead, consider a limited window (size $k$) at any one time. Not optimal, since might be more than $k$ added or deleted lines. Works well in practice, and brings running time down to $O(nk)$.

2.9 Memoization

Can make the recursive formulation work, as long as you don’t let yourself compute the answer to the same question repeatedly.

- Create a hash table for each subroutine associating inputs to answers.
- No side effects, so same input means same output.
- Each time we compute an output, store it in the hash table.
- Before we try to compute a new answer, see if that one’s already in the hash table (and return right away if it is).
- Get same worst-case bounds (often better best case).
- (Can do the same with DFS to identify “reachable” subproblems.)

Turn an exponential algorithm into a quadratic one!
3 OTHER PROBLEMS

3.1 Other Problems

If time, we could do optimal matrix chain. Or stochastic shortest paths.