Dynamic Programming: Longest Common Subsequence

Sergei Uversky
2/3/2015

1 Dynamic programming

Dynamic programming is an algorithmic approach to problems that have two key properties: overlapping subproblems and optimal substructure (an optimal solution for the problem can be efficiently constructed from optimal solutions of the subproblems).

Recall that the divide and conquer approach works by splitting a problem into independent subproblems, solving them recursively and combining the solutions into a cohesive whole. Dynamic programming is similar, but the split involves subproblems whose solutions overlap; therefore, it is important to store sub-solutions in a table. This avoids doing the unnecessary work of repeatedly recomputing an already-solved subproblem.

2 Longest common subsequence

Suppose we have two strings, and we want to find the longest common subsequence (LCS) of characters in those strings, e.g. the strings

\[ A = \text{ENTROPY} \]
\[ B = \text{THORNY} \]

have ”TRY” and ”TOY” as their LCSes. We can state this problem more formally as follows.

Let \( A = a_1a_2...a_m \) and \( B = b_1b_2...b_n \) be strings. Let \( A = a_1a_2...a_i, \ i \leq m \) and \( B = b_1b_2...b_j, \ j \leq n \) be prefixes of \( A \) and \( B \), respectively. Let a subsequence of a string be the same string with zero or more elements omitted. Let \( M_{ij} \) be the length of a longest common subsequence (LCS) of \( A_i \) and \( B_j \).

We desire \( M_{mn} \), and a corresponding LCS with length \( M_{mn} \).
2.1 Approach 1: brute force

A naive solution to this problem would involve listing all subsequences of \( A \) and seeing if they are also subsequences of \( B \), keeping track of which is the longest. There are \( 2^m \) such subsequences of \( A \), so this approach is \( O(2^m) \).

2.2 Approach 2: dynamic programming

Intuitively, the LCS for a pair of strings is dependent on the LCS of the prefixes of those strings; also, one can easily get the LCS at \((i, j)\) given the LCSes for prior indexes. Thus, we can attempt a dynamic programming approach.

We can define a recurrence for computing \( M_{ij} \) as follows:

\[
\begin{align*}
0, & \quad i = 0, j = 0 \\
M_{ij} = M_{i-1,j-1} + 1, & \quad i > 0, j > 0, a_i = b_j \\
\max\{M_{i-1,j}, M_{i,j-1}\} & \quad \text{otherwise}
\end{align*}
\]

In laymen’s terms: if both strings are empty, their LCS must also be empty. If we have found a matching character at location \((i, j)\), then the LCS so far is one character longer than the LCS at \((i - 1, j - 1)\). Otherwise, our LCS can be no longer than the LCS at \((i - 1, j)\) or \((i, j - 1)\).

2.2.1 Length of LCS

We can write an algorithm for finding \( M_{ij} \) by using a back-pointer technique as follows. More details are given on the following page.

\[
\text{LCS-LENGTH}(A, B)
\]

1: \((m, n) \leftarrow (A.length, B.length)
2: \text{for } j \leftarrow 0 \text{ to } n \text{ do}
3: \quad (M_{0,j}, P_{0,j}) \leftarrow (0, (0, 0))
4: \text{end for}
5: \text{for } i \leftarrow 0 \text{ to } m \text{ do}
6: \quad (M_{i,0}, P_{i,0}) \leftarrow (0, (0, 0))
7: \quad \text{for } j \leftarrow 1 \text{ to } n \text{ do}
8: \quad \quad \text{if } a_i = b_j \text{ then}
9: \quad \quad \quad (M_{ij}, P_{ij}) \leftarrow (M_{i-1,j-1} + 1, (i, j))
10: \quad \quad \text{else if } M_{i-1,j} \geq M_{i,j-1} \text{ then}
11: \quad \quad \quad (M_{ij}, P_{ij}) \leftarrow (M_{i-1,j}, P_{i-1,j})
12: \quad \quad \text{else}
13: \quad \quad \quad (M_{ij}, P_{ij}) \leftarrow (M_{i,j-1}, P_{i,j-1})
14: \quad \quad \text{end if}
15: \quad \text{end if}
16: \text{end for}
17: \text{return } M, P
\( P_{ij} \) essentially holds the indexes of the last matching character for the LCS at \((i, j)\). The algorithm essentially follows the recurrence:

- When a matching character is found, \( M_{ij} \) is appropriately updated, and \( P_{ij} \) refers to the current character;
- If there is no match, then \( M_{ij} \) and \( P_{ij} \) are appropriately updated based on \( \max \{ M_{i-1,j}, M_{i,j-1} \} \).

### 2.2.2 Getting the LCS from the length

There is a simple algorithm for printing out the LCS, given \( M \) and \( P \) found above.

\[
\text{PRINT-LCS}(M, P, A, i, j) \\
1: \text{if } M_{ij} > 0 \text{ then} \\
2: \quad (i, j) \leftarrow P_{ij} \\
3: \quad \text{PRINT-LCS}(M, P, A, i-1, j-1) \\
4: \quad \text{print } a_i \\
5: \text{end if}
\]

To print a LCS, the initial call would be \( \text{PRINT-LCS}(M, P, A, A \cdot \text{length}, B \cdot \text{length}) \).

### 2.2.3 Analysis

Finding the length of the LCS intuitively takes \( O(mn) \) time, as for each \( m \), the algorithm iterates \( n \) times. Additionally, this algorithm takes \( O(mn) \) space, as both of the arrays \( M \) and \( P \) are of size \( m \times n \).

### 2.3 Approach 3: dynamic programming, improving on space

We can improve on the space used by this approach by essentially using dynamic programming to find a split point, and then applying divide and conquer to further simplify the problem.

There are two possible cases for \( M_{mn} \), dependent on the following definition. Let \( i' \leq \lceil \frac{m}{2} \rceil \) be the largest value s.t. \( \exists (i', j') \) where \( a_{i'} = b_{j'} \).

1. If \( (i', j') \) exists, then \( M_{mn} = M_{i'-1,j'-1} + 1 + \text{LCS}(a_{i'+1} \ldots a_m, b_{j'+1} \ldots b_n) \).
2. Otherwise, \( M_{mn} = \text{LCS}(a_{i' \lceil \frac{m}{2} \rceil} \ldots a_m, b_1 \ldots b_n) \).

Essentially, we are saying that if an appropriate \( (i', j') \) pair exists, we must have at least one character in a LCS in the first half of \( A \); thus, we can divide and conquer by splitting \( A \) and \( B \) on \( (i', j') \). Otherwise, we can discard half of \( A \), as we know we have a LCS where the first character is in the second half of \( A \).
Therefore, in Case 1, we are essentially doing a divide and conquer by finding the following recursively:

\[ L_1 = \text{LCS}(a_1, a_{i-1}, b_1, \ldots, b_{j-1}) \]
\[ L_2 = \text{LCS}(a_{\lceil m/2 \rceil}, a_m, b_{j+1}, \ldots, b_n) \]

and then concatenating \( L_1 :: a'_i :: L_2 \). In Case 2, we are doing a simple recursion, and can find some split point \( n' \) in \( B \).

The recurrence for the running time of this algorithm, \( T(m, n) \), is as follows:

\[
T(0, n) = 0 \quad T(m, 0) = 0 \\
T(1, n) = cn \quad T(m, 1) = cm \\
T(m, n) \leq cmn + T(\lfloor m/2 \rfloor, n') + T(\lfloor m/2 \rfloor, n - n' - 1)
\]

The algorithm follows. We use arrays \( R[0..n] \) and \( S[0..n] \) of records of two fields, value and back: \( (v, b) \).

**LCS-LENGTH-2(A, B)**
1. \((m, n) \leftarrow (A.length, B.length)\)
2. \( \text{for } j \leftarrow 0 \text{ to } n \text{ do} \)
3. \( R_j \leftarrow (0, (0, 0)) \)
4. \( \text{end for} \)
5. \( S_0 \leftarrow (0, (0, 0)) \)
6. \( \text{for } i \leftarrow 1 \text{ to } m \text{ do} \)
7. \( \text{for } j \leftarrow 1 \text{ to } n \text{ do} \)
8. \( \text{if } a_i = b_j \text{ then} \)
9. \( S_{j,v} \leftarrow R_{j-1,v} + 1 \)
10. \( \text{if } i \leq \lfloor m/2 \rfloor \text{ then} \)
11. \( S_{j,b} \leftarrow (i, j) \)
12. \( \text{else} \)
13. \( S_{j,b} \leftarrow R_{j-1,b} \)
14. \( \text{end if} \)
15. \( \text{else if } R_{j,v} \geq S_{j-1,v} \text{ then} \)
16. \( S_j \leftarrow R_j \)
17. \( \text{else} \)
18. \( S_j \leftarrow S_{j-1} \)
19. \( \text{end if} \)
20. \( \text{end for} \)
21. \( \text{for } j \leftarrow 1 \text{ to } n \text{ do} \)
22. \( R_j \leftarrow S_j \)
23. \( \text{end for} \)
24. \( \text{return } S \)
3 Related reading

The chapter on dynamic programming (and particularly the section on the longest common subsequence problem) in the Cormen textbook is highly relevant here. LCS is in §15.4 (page 350) in the second edition.