Longest Common Subsequence

Inspiration

- Biological applications often need to compare the DNA of two (or more) different organisms
- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine
- each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set {A, C, G, T}

Inspiration

- For example, the DNA of one organism may be S_1= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and the DNA of another organism may be S_2= GTCGTTCGGAATGCCGTTGCTCTGTAAA.
- One reason to compare two strands of DNA is to determine how “similar the two strands are, as some measure of how closely related the two organisms are

Inspiration

- We can, and do, define similarity in many different ways
- For example, we can say that two DNA strands are similar if one is a substring of the other.
- In our example, neither S_1 nor S_2 is a substring of the other.
- Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small.

Inspiration

- another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3
- In which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively
- The longer the strand S_3 we can find, the more similar S_1 and S_2 are

Inspiration

- S_1= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- S_{2} = GTCGTTCGGAATGCCGTTGCTCTGTAAA
- S_3 is GTCGTCGGAAGCCGGCCGAA

Problem Statement

- A subsequence of a given sequence is just the given sequence with zero or more elements left out
- Formally, given a sequence $X = <x_1,x_2,\ldots,x_m>$, another sequence $Z =<z_1,z_2,\ldots,z_k>$ is a subsequence of X if there exists a strictly increasing sequence $<i_1,i_2,\ldots,i_k>$ of indices of X such that for all $j = 1,2,\ldots,k$, we have $x_{i_j}= z_j$

Problem Statement

- For example, Z = <B, C, D, B> is a subsequence of X = <A, B, C, B, D, A, B> with corresponding index sequence <2, 3, 5, 7>
- Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y

Problem Statement

- For example, if X = <A, B, C, B, D, A, B> and Y = <B, D, C, A, B, A>, the sequence <B, C, A> is a common subsequence of both X and Y
- But not a longest common subsequence (LCS) of X and Y
- Sequence <B, C, B, A>, which is also common to both X and Y , has length 4 is the LCS
- Since X and Y have no common subsequence of length 5 or greater

Step 1: Characterizing a longest common subsequence

- Brute–force approach to solve LCS problem:
  - Enumerate all subsequences of X
  - Check each subsequence to see whether it is also a subsequence of Y
  - Keeping track of the longest subsequence we find.
- Each subsequence of X corresponds to a subset of the indices {1, 2,…,m} of X
- Because X has 2^m subsequences, this approach requires exponential time, making it impractical

Basis of Optimal substructure of an LCS

- Given a sequence $X = <x_1, x_2, \ldots, x_m>$, we define the ith prefix of X , for $i = 0,1,\ldots,m$, as $X_i = <x_1, x_2, \ldots, x_i>$
- For example, if X = <A, B, C, B, D, A, B>, then X4 = <A, B, C, B> and X_0 is the empty sequence

Theorem 15.1 Optimal substructure of an LCS

- Let $X = <x_1, x_2,...,x_m>$ and $Y = <y_1, y_2,...,y_n>$ be sequences, and let $Z = <z_1, z_2,..., z_k>$ 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}

Proof of Theorem 15.1

- (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length k + 1, contradicting the supposition that Z is a longest common subsequence of X and Y . Thus, we must have $z_k = x_m = y_n$.
- Now, the prefix Z_{k-1} is a length–(k –1) common subsequence of X_{m-1} and Y_{n-1}

Proof of Theorem 15.1

- We wish to show that it is an LCS
- Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k-1$
- Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction

Proof of Theorem 15.1

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y

- If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y
- (3) The proof is symmetric to (2)

Step 2: A recursive solution

- Theorem 15.1 implies that we should examine either one or two subproblems when finding an LCS of X = <x1, x2,…,xm> and Y= <y1, y2,…,yn>
- If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}
- Appending $x_m = y_n$ to this LCS yields an LCS of X and Y
- If $x_m \neq y_n$, then we must solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}.

Step 2: A recursive solution

- Whichever of these two LCSs is longer is an LCS of X and Y
- Because these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must appear within an LCS of X and Y .

Step 2: Overlapping Subproblem

- To find an LCS of X and Y, we may need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y
- But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}
- Many other subproblems share subsubproblems.

Step 2: Overlapping Subproblem

- Let us define c[i, j] to be the length of an LCS of the sequences X_i and Y_j
- either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0

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

Step 3: Computing the length of an LCS

- LCS problem has only θ (m*n) distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.
- We maintain two 2D tables c and b for dynamic programming
- c table maintains the length of the common sub sequence
- b table helps to construct the solution

Step 3: Computing the length of an LCS

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each "↖" on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Step 4: Constructing an LCS

- b table returned by LCS–LENGTH enables us to quickly construct an LCS for $X = <x_1, x_2,\ldots,x_m>$ and $Y = <y_1, y_2,\ldots,y_n>$
- We simply begin at b[m, n] and trace through the table by following the arrows
- Whenever we encounter a "↖" in entry b[i,j], it implies that $x_i = y_j$ is an element of the LCS that LCS–L ENGTH found.

Step 4: Constructing an LCS

- With this method, we encounter the elements of this LCS in reverse order.
- The following recursive procedure prints out an LCS of X and Y in the proper, forward order
- The initial call is PRINT –LCS(b, X, X.length, Y.length)

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
        return
if b[i, j] == "↖"
        PRINT-LCS(b, X, i − 1, j − 1)
        print x_i
elseif b[i, j] == "↑"
        PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

- For the b table in Figure 15.8 this procedure prints BCBA
  The procedure takes time O(m + n) since it decrements at least one of i and j in each recursive call

Improving the code

- Once you have developed an algorithm, you will often find that you can improve on the time or space it uses
- Some changes can simplify the code and improve constant factors but otherwise yield no asymptotic improvement in performance.
- Others can yield substantial asymptotic savings in time and space.

Improving the code

- In the LCS algorithm, for example, we can eliminate the b table altogether. Each c[i, j] entry depends on only three other c table entries: c[i –1, j– 1], c[i – 1, j], and c[i, j –1].
- Given the value of c[i, j], we can determine in O(1) time which of these three values was used to compute c[i,j], without inspecting table b.

Improving the code

- Thus, we can reconstruct an LCS in $O(m+n)$ time using a procedure similar to PRINT –LCS.
- Although we save θ (mn) space by this method, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since we need θ (mn) space for the c table anyway.

Improving the code

- We can, however, reduce the asymptotic space requirements for LCS–LENGTH , since it needs only two rows of table c at a time: the row being computed and the previous row.
- This improvement works if we need only the length of an LCS; if we need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace our steps in $O(m + n)$ time