CSE 417 Algorithms and Complexity
Winter 2023
Lecture 21
Longest Common Subsequence

Announcements

- Lecture plans
  - Monday: Longest Common Subsequence
  - Wednesday: Shortest Paths
  - Rest of the course: NP-Completeness

Longest Common Subsequence

- C = c₁…c₉ is a subsequence of A = a₁…aₘ if C can be obtained by removing elements from A (but retaining order)
- LCS(A, B): A maximum length sequence that is a subsequence of both A and B

Determine the LCS of the following strings

BARTHOLEMEWSIMPSON
KRUSTYTHECLOWN

String Alignment Problem

- Align sequences with gaps
  
  CAT TGA AT
  
  CAGAT AGGA

- Charge δₓ if character x is unmatched
- Charge γₓᵧ if character x is matched to character y

Note: the problem is often expressed as a minimization problem, with γₓᵧ = 0 and δₓ > 0

Recursive Version

LCS(a₁a₂…aₘ, b₁b₂…bₙ){
  if (aₘ == bₙ)
    return LCS(a₁a₂…aₘ⁻¹, b₁b₂…bₙ⁻¹) + 1;
  else
    return max(LCS(a₁a₂…aₘ⁻¹, b₁b₂…bₙ⁻¹), LCS(a₁a₂…aₘ, b₁b₂…bₙ⁻¹));
}
LCS Optimization

- \( A = a_1a_2...a_m \)
- \( B = b_1b_2...b_n \)
- \( \text{Opt}[j, k] \) is the length of LCS\((a_1a_2...a_j, b_1b_2...b_k)\)

Optimization recurrence

If \( a_j = b_k \), \( \text{Opt}[j, k] = 1 + \text{Opt}[j-1, k-1] \)

If \( a_j \neq b_k \), \( \text{Opt}[j, k] = \max(\text{Opt}[j-1, k], \text{Opt}[j, k-1]) \)

Give the Optimization Recurrence for the String Alignment Problem

- Charge \( \delta_x \) if character \( x \) is unmatched
- Charge \( \gamma_{xy} \) if character \( x \) is matched to character \( y \)

\[ \text{Opt}[j, k] = \]

Let \( a_j = x \) and \( b_k = y \)
Express as minimization

String edit with Typo Distance

- Find closest dictionary word to typed word
- Dist('a', 's') = 1
- Dist('a', 'u') = 6
- Capture the likelihood of mistyping characters

Dynamic Programming Computation

Code to compute \( \text{Opt}[n, m] \)

```c
for (int i = 0; i < n; i++)
for (int j = 0; j < m; j++)
if (A[i] == B[j])
    Opt[i, j] = Opt[i-1, j-1] + 1;
else if (Opt[i-1, j] >= Opt[i, j-1])
    Opt[i, j] = Opt[i, j-1];
else
    Opt[i, j] = Opt[i-1, j];
```
Storing the path information

\[ A[1..m], B[1..n] \]

for \( i := 1 \) to \( m \)
for \( j := 1 \) to \( n \)
for \( i := 1 \) to \( m \)
for \( j := 1 \) to \( n \)

\[ \text{Opt}[i, j] := 0; \]

\[ \text{Opt}[0,0] := 0; \]

\[ \text{Opt}[0, j] := 0; \]

if \( A[i] = B[j] \) \{ \[ \text{Opt}[i, j] := 1 + \text{Opt}[i-1, j-1]; \]

if \( \text{Opt}[i-1, j] \geq \text{Opt}[i, j-1] \) \{ \[ \text{Opt}[i, j] := \text{Opt}[i-1, j]; \]

if \( \text{Opt}[i-1, j] < \text{Opt}[i, j-1] \) \{ \[ \text{Opt}[i, j] := \text{Opt}[i, j-1]; \]

Reconstructing Path from Distances

How good is this algorithm?

• Is it feasible to compute the LCS of two strings of length 300,000 on a standard desktop PC? Why or why not.

N = 17000

Runtime should be about 5 seconds*

Implementation 1

```java
public int ComputeLCS() {
    int n = str1.Length;
    int m = str2.Length;
    int[,] opt = new int[n + 1, m + 1];
    for (int i = 0; i <= n; i++)
        opt[i, 0] = 0;
    for (int j = 0; j <= m; j++)
        opt[0, j] = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (str1[i - 1] == str2[j - 1])
                opt[i, j] = opt[i - 1, j - 1] + 1;
            else if (opt[i - 1, j] >= opt[i, j - 1])
                opt[i, j] = opt[i - 1, j];
            else
                opt[i, j] = opt[i, j - 1];
    return opt[n, m];
}
```

Implementation 2

```java
public int SpaceEfficientLCS() {
    int n = str1.Length;
    int m = str2.Length;
    int[] prevRow = new int[m + 1];
    int[] currRow = new int[m + 1];
    for (int j = 0; j <= m; j++)
        prevRow[j] = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (str1[i - 1] == str2[j - 1])
                currRow[j] = prevRow[j - 1] + 1;
            else if (prevRow[j] >= prevRow[j - 1])
                currRow[j] = prevRow[j];
            else
                currRow[j] = currRow[j - 1];
    return currRow[m];
}
```


**Observations about the Algorithm**

- The computation can be done in $O(m+n)$ space if we only need one column of the Opt values or Best Values.
- The computation requires $O(nm)$ space if we store all of the string information.

**Computing LCS in $O(nm)$ time and $O(n+m)$ space**

- Divide and conquer algorithm
- Recomputing values used to save space
- Section 6.7 of the text, but we will not have time to cover in detail (so you are not responsible for section 6.7)

**Divide and Conquer Algorithm**

- Where does the best path cross the middle column?
- For a fixed $i$, and for each $j$, compute the LCS that has $a_i$ matched with $b_j$

**Algorithm Analysis**

- $T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm$
- Solution: $T(m,n) \leq 2cnm$