CSE 421, Introduction to Algorithms

Lecture 15, Winter 2024
Dynamic Programming
Longest Common Subsequence
Announcements

• Dynamic Programming Reading:
  – Weighted Interval Scheduling, Segmented Least Squares, Knapsack and Subset Sum
  – 6.6 String Alignment
  – 6.8 Shortest Paths (Bellman-Ford)

• Midterm, Friday, Feb 9
  – Material through 6.3 and HW 5
  – Feb 8 Section will be Midterm review
    • Old exam problems on course homepage
  – Homework 6 due Feb 14
Longest Common Subsequence

- $C = c_1 \ldots c_g$ is a subsequence of $A = a_1 \ldots a_m$ if $C$ can be obtained by removing elements from $A$ (but retaining order).
- $\text{LCS}(A, B)$: A maximum length sequence that is a subsequence of both $A$ and $B$.

\[
\text{LCS}('\text{BARTHOLEMEWSIMPSON}', '\text{KRUSTYTHECLOWN}') = '\text{RTHOWN}'
\]
Optimization recurrence

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

If \( a_j \neq b_k \), \( \text{Opt}[ j,k] = \max(\text{Opt}[ j-1,k], \text{Opt}[ j,k-1]) \)
Dynamic Programming
Computation
Code to compute Opt[ n, m]

for (int i = 0; i < n; i++)
    for (int j = 0; j < m; 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-1, j ];
        else
            Opt[ i, j ] := Opt[ i, j-1];
Storing the path information

A[1..m], B[1..n]
for i := 1 to m \hspace{1cm} \text{Opt}[i, 0] := 0;
for j := 1 to n \hspace{1cm} \text{Opt}[0,j] := 0;
\text{Opt}[0,0] := 0;
for i := 1 to m
  for j := 1 to n
    \text{Best}[i,j] := \text{Diag};
    if A[i] = B[j] \hspace{1cm} \text{Opt}[i,j] := 1 + \text{Opt}[i-1,j-1];
    else if \text{Opt}[i-1, j] \geq \text{Opt}[i, j-1]
      \text{Opt}[i, j] := \text{Opt}[i-1, j], \text{Best}[i,j] := \text{Left};
    else
      \text{Opt}[i, j] := \text{Opt}[i, j-1], \text{Best}[i,j] := \text{Down};
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.
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];
}
N = 17000

Runtime should be about 5 seconds*

* Personal PC, 10 years old
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++) {
        currRow[0] = 0;
        for (int j = 1; j <= m; j++) {
            if (str1[i - 1] == str2[j - 1])
                currRow[j] = prevRow[j - 1] + 1;
            else if (prevRow[j] >= currRow[j - 1])
                currRow[j] = prevRow[j];
            else
                currRow[j] = currRow[j - 1];
        }
        for (int j = 1; j <= m; j++)
            prevRow[j] = currRow[j];
    }

    return currRow[m];
}
N = 300000

N: 10000 Base 2 Length: 8096  Gamma: 0.8096  Runtime:00:00:01.86
N: 20000 Base 2 Length: 16231  Gamma: 0.81155  Runtime:00:00:07.45
N: 30000 Base 2 Length: 24317  Gamma: 0.8105667  Runtime:00:00:16.82
N: 40000 Base 2 Length: 32510  Gamma: 0.81275  Runtime:00:00:29.84
N: 50000 Base 2 Length: 40563  Gamma: 0.81126  Runtime:00:00:46.78
N: 60000 Base 2 Length: 48700  Gamma: 0.8116667  Runtime:00:01:08.06
N: 70000 Base 2 Length: 56824  Gamma: 0.8117715  Runtime:00:01:33.36

N: 300000 Base 2 Length: 243605  Gamma: 0.8120167  Runtime:00:28:07.32

Chvatal-Sankoff Constant: $\gamma_k$
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 algorithm can be run from either end of the strings
Computing LCS in $O(nm)$ time and $O(n+m)$ space

- Divide and conquer algorithm
- Recomputing values used to save space
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 $b_j$
Constrained LCS

• \( \text{LCS}_{i,j}(A,B) \): The LCS such that
  - \( a_1, \ldots, a_i \) paired with elements of \( b_1, \ldots, b_j \)
  - \( a_{i+1}, \ldots, a_m \) paired with elements of \( b_{j+1}, \ldots, b_n \)

• \( \text{LCS}_{4,3}(abbacbb, cbbaa) \)
A = RRSSR T T R TS
B = RTSRRSTST

Compute \( \text{LCS}_{5,0}(A,B) \), \( \text{LCS}_{5,1}(A,B) \), …, \( \text{LCS}_{5,9}(A,B) \)
A = RRSSRTTTRS
B = RTSRRSTST

Compute $\text{LCS}_{5,0}(A,B)$, $\text{LCS}_{5,1}(A,B)$, ..., $\text{LCS}_{5,9}(A,B)$

<table>
<thead>
<tr>
<th>j</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Computing the middle column

- From the left, compute LCS($a_1 \ldots a_{m/2}, b_1 \ldots b_j$)
- From the right, compute LCS($a_{m/2+1} \ldots a_m, b_{j+1} \ldots b_n$)
- Add values for corresponding j’s

- Note – this is space efficient
Divide and Conquer

- A = $a_1, \ldots, a_m$  \hspace{1cm} B = $b_1, \ldots, b_n$
- Find $j$ such that
  - LCS($a_1 \ldots a_{m/2}, b_1 \ldots b_j$) and
  - LCS($a_{m/2+1} \ldots a_m, b_{j+1} \ldots b_n$) yield optimal solution
- Recurse
Algorithm Analysis

- \( T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm \)
Prove by induction that
\[ T(m,n) \leq 2^c m n \]

\[ T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm \]
Memory Efficient LCS Summary

• We can afford $O(nm)$ time, but we can’t afford $O(nm)$ space
• If we only want to compute the length of the LCS, we can easily reduce space to $O(n+m)$
• Avoid storing the value by recomputing values
  – Divide and conquer used to reduce problem sizes
String Alignment Problem

- Align sequences with gaps

\[
\begin{array}{c}
\text{CAT TGA} \\
\text{CAGAT AGGA}
\end{array}
\]

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

Note: the problem is often expressed as a minimization problem, with \( \gamma_{xx} = 0 \) and \( \delta_x > 0 \)
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
- $\text{Dist}(\text{‘a’}, \text{‘s’}) = 1$
- $\text{Dist}(\text{‘a’}, \text{‘u’}) = 6$
- Capture the likelihood of mistyping characters