CSE 421  
Algorithms  
Richard Anderson  
Lecture 20  
Memory Efficient Longest Common Subsequence

Longest Common Subsequence

• \( C = c_1…c_j \) is a subsequence of \( A = a_1…a_m \) 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 \( B \)

\[ \text{occurrence} \quad \text{attacggct} \]
\[ \text{occurrence} \quad \text{tacgacca} \]

LCS Optimization

• \( A = a_1a_2…a_m \)
• \( B = b_1b_2…b_n \)
• \( \text{Opt}[i,j] \) is the length of LCS(\( a_1a_2…a_i, b_1b_2…b_j \))

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

Storing the path information

\[ A[1..m], B[1..n] \]
for \( i = 1 \) to \( m \)  \( \text{Opt}[i,0] := 0; \)
for \( j = 1 \) to \( n \)  \( \text{Opt}[0,j] := 0; \)
\( \text{Opt}[0,0] := 0; \)
for \( i = 1 \) to \( m \)
for \( j = 1 \) to \( n \)
if \( A[i] = B[j] \)  \( \{ \text{Opt}[i,j] := 1 + \text{Opt}[i-1,j-1]; \text{Best}[i,j] := \text{Diag}; \} \)
else if \( \text{Opt}[i-1,j] >= \text{Opt}[i,j-1] \)  \( \{ \text{Opt}[i,j] := \text{Opt}[i-1,j]; \text{Best}[i,j] := \text{Left}; \} \)
else  \( \{ \text{Opt}[i,j] := \text{Opt}[i,j-1]; \text{Best}[i,j] := \text{Down}; \} \)
Algorithm Performance

- O(nm) time and O(nm) space
- On current desktop machines
  - n, m < 10,000 is easy
  - n, m > 1,000,000 is prohibitive
- Space is more likely to be the bounding resource than time

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

Constrained LCS

- LCS_{i,j}(A,B): The LCS such that
  - a_1,...,a_i paired with elements of b_1,...,b_j
  - a_{i+1},...,a_m paired with elements of b_{j+1},...,b_n
- LCS_{4,3}(abbacbb, cbbaa)

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

A = RRSSRTTTRS
B=RTSRRRSTST

Compute LCS_{5,0}(A,B), LCS_{5,1}(A,B),...,LCS_{5,9}(A,B)
A = RRSSRTTRTS
B=RTSRRSTST

Compute LCS₅₁₀(A,B), LCS₅₁₁(A,B),…,LCS₅₉₉(A,B)

Computing the middle column
• From the left, compute LCS(a₁…aₘ₁₂,b₁…bₗ)
• From the right, compute LCS(aₘ₁₂+₁…aₘ,bₗ+₁…bₙ)
• Add values for corresponding j’s
  • Note – this is space efficient

Divide and Conquer
• A = a₁,…,aₘ  B = b₁,…,bₙ
• Find j such that
  – LCS(a₁…aₘ₁₂, b₁…bₗ) and
  – LCS(aₘ₁₂+₁…aₘ,bₗ+₁…bₙ) 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) <= 2cmn

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
**Shortest Path Problem**
- Dijkstra’s Single Source Shortest Paths Algorithm
  - $O(m \log n)$ time, positive cost edges
- General case – handling negative edges
- If there exists a negative cost cycle, the shortest path is not defined
- Bellman-Ford Algorithm
  - $O(mn)$ time for graphs with negative cost edges

**Lemma**
- If a graph has no negative cost cycles, then the shortest paths are simple paths
- Shortest paths have at most $n-1$ edges

**Shortest paths with a fixed number of edges**
- Find the shortest path from $v$ to $w$ with exactly $k$ edges

**Express as a recurrence**
- $\text{Opt}_k(w) = \min_x [\text{Opt}_{k-1}(x) + c_{vw}]$
- $\text{Opt}_0(w) = 0$ if $v=w$ and infinity otherwise

**Algorithm, Version 1**
```java
foreach w
    M[0, w] = infinity;
M[0, v] = 0;
for i = 1 to n-1
    foreach w
        M[i, w] = min_j [M[i-1, x] + cost(x, w)];
```