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)
- LCS(\( A, B \)): A maximum length sequence that is a subsequence of both \( A \) and \( B \)

\[
\begin{align*}
\text{occuranc}e & \quad \text{attacggtc} \\
\text{occurrence} & \quad \text{tacgacca}
\end{align*}
\]

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

\[
\begin{align*}
A[1..m], \ B[1..n] \\
\text{for } i := 1 \text{ to } m & \quad \text{Opt}[i, 0] := 0; \\
\text{for } j := 1 \text{ to } n & \quad \text{Opt}[0, j] := 0; \\
\text{for } i := 1 \text{ to } m & \\
\text{for } j := 1 \text{ to } n & \\
\text{if } A[i] = B[j] & \quad \{ \text{Opt}[i,j] := 1 + \text{Opt}[i-1,j-1]; \text{Best}[i,j] := \text{Diag}; \} \\
\text{else if } \text{Opt}[i-1,j] > \text{Opt}[i,j-1] & \quad \{ \text{Opt}[i,j] := \text{Opt}[i-1,j]; \text{Best}[i,j] := \text{Left}; \} \\
\text{else} & \quad \{ \text{Opt}[i,j] := \text{Opt}[i,j-1]; \text{Best}[i,j] := \text{Down}; \}
\end{align*}
\]

How good is this algorithm?

- Is it feasible to compute the LCS of two strings of length 100,000 on a standard desktop PC? Why or why not.
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 $b_j$.

Divide and Conquer

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

Algorithm Analysis

- $T(m, n) = T(m/2, j) + T(m/2, n-j) + cnm$

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 Paths with Dynamic Programming

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

Express as a recurrence

• $\text{Opt}_k(w) = \min_x [\text{Opt}_{k-1}(x) + c_{xw}]$
• $\text{Opt}_0(w) = 0$ if $v=w$ and infinity otherwise

Algorithm, Version 1

foreach $w$
  $M[0, w] = \infty$;
  $M[0, v] = 0$;
for $i = 1$ to $n-1$
  foreach $w$
    $M[i, w] = \min_x (M[i-1, x] + \text{cost}(x, w))$;
Algorithm, Version 2

foreach w
    M[0, w] = infinity;
    M[0, v] = 0;
for i = 1 to n-1
    foreach w
        M[i, w] = min(M[i-1, w], minx(M[i-1,x] + cost[x,w]))

Algorithm, Version 3

foreach w
    M[w] = infinity;
    M[v] = 0;
for i = 1 to n-1
    foreach w
        M[w] = min(M[w], minx(M[x] + cost[x,w]))

Correctness Proof for Algorithm 3

• Key lemma – at the end of iteration i, for all w, M[w] <= M[i, w]:

• Reconstructing the path:
  – Set P[w] = x, whenever M[w] is updated from vertex x

If the pointer graph has a cycle, then the graph has a negative cost cycle

• If P[w] = x then M[w] >= M[x] + cost(x,w)
  – Equal when w is updated
  – M[x] could be reduced after update
• Let v₁, v₂,…vₖ be a cycle in the pointer graph with (vᵦ, vᵦ) the last edge added
  – Just before the update
    • M[vᵦ] >= M[vᵦ] + cost(vᵦ, vᵦ) for j < k
    • M[vᵦ] > M[v₁] + cost(v₁, vᵦ)
  – Adding everything up
    • 0 > cost(v₁,v₂) + cost(v₂,v₃) + … + cost(vₖ, v₁)

Negative Cycles

• If the pointer graph has a cycle, then the graph has a negative cycle
• Therefore: if the graph has no negative cycles, then the pointer graph has no negative cycles

Finding negative cost cycles

• What if you want to find negative cost cycles?
Foreign Exchange Arbitrage

<table>
<thead>
<tr>
<th></th>
<th>USD</th>
<th>EUR</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>------</td>
<td>0.8</td>
<td>1.2</td>
</tr>
<tr>
<td>EUR</td>
<td>1.2</td>
<td>------</td>
<td>1.6</td>
</tr>
<tr>
<td>CAD</td>
<td>0.8</td>
<td>0.6</td>
<td>------</td>
</tr>
</tbody>
</table>