Algorithms to compute string similarity
String Similarity

• Finding differences or edit distance between two sequences can be alternately formulated as finding similarity between two sequences.

• Biologists usually prefer using similarity measures to study relationship between strings.

• Earlier we gave a definition of alignment as follows:
  – **Definition**: Let $v$ and $w$ be two sequences of length $n$ and $m$, respectively, over a finite alphabet $\Sigma$. An **alignment** maps the strings $v$ and $w$ into strings and that may contain indel (‘-’) characters such that removal of all indel characters leaves $v$ and $w$ intact.
Similarity using Dynamic Programming - Longest Common Subsequence Problem

If we are interested to find an alignment that maximizes $S(n,m)$, the number of matched symbols, we can assign a weight of 1 for match and a weight of 0 for both insert and delete operations. The substitution operation is considered as a delete followed by an insert operation. The score table $\delta$ consists simply of all diagonal entries to be 1 and rest are 0. The dynamic programming equations will then look like

$$S(0,0) \leftarrow 0$$

for $j = 1$ to $m$ do

for $i = 1$ to $n$ do

\{ 

$S(i,0) \leftarrow 0$ \hspace{1cm} /* delete from v */

for $j = 1$ to $m$ do

if $v_i = w_j$ match = $S(i-1, j-1) + 1$

$S(i, j) \leftarrow \max\{S(i, j-1), S(i-1, j), \text{match}\}$

\}

write "similarity score is" $S(n,m)$
Dynamic Programming Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initialize $1^{st}$ row and $1^{st}$ column to be all zeroes.

Or, to be more precise, initialize $0^{th}$ row and $0^{th}$ column to be all zeroes.
LCS via Dynamic Programming

: Example

\[ S_{i,j} = \begin{cases} 
S_{i-1,j-1} & \text{value from NW} +1, \text{ if } v_i = w_j \\
\max \{ S_{i-1,j}, S_{i,j-1} \} & \text{value from North (top)} \\
S_{i,j-1} & \text{value from West (left)} 
\end{cases} \]
Alignment: Backtracking

Arrows show where the score originated from.

↑ if from the top
← if from the left

if $v_i = w_j$
Backtracking Example

Find a match in row and column 2.

- $i=2, j=2, 5$ is a match (T).
- $j=2, i=4, 5, 7$ is a match (T).

Since $v_i = w_j$, $s_{i,j} = s_{i-1,j-1} + 1$

- $s_{2,2} = [s_{1,1} = 1] + 1$
- $s_{2,5} = [s_{1,4} = 1] + 1$
- $s_{4,2} = [s_{3,1} = 1] + 1$
- $s_{5,2} = [s_{4,1} = 1] + 1$
- $s_{7,2} = [s_{6,1} = 1] + 1$
Backtracking Example

Continuing with the dynamic programming algorithm gives this result.
LCS: Example

- **i coords:** 0 1 2 2 3 3 4 5 6 7 8
- **elements of \(v\):**
  - A
  - T
  - --
  - C
  - --
  - T
  - G
  - A
  - T
  - C
- **elements of \(w\):**
  - --
  - T
  - G
  - C
  - A
  - T
  - --
  - A
  - --
  - C
- **j coords:** 0 0 1 2 3 4 5 5 6 6 7

(0,0) → (1,0) → (2,1) → (2,2) → (3,3) → (3,4) → (4,5) → (5,5) → (6,6) → (7,6) → (8,7)

Matches shown in **red**

- positions in \(v\): 2 < 3 < 4 < 6 < 8
- positions in \(w\): 1 < 3 < 5 < 6 < 7

**Every common subsequence is a path in 2-D grid**
Imagine vertical lines for characters of sequence $w$ and horizontal lines for those of $v$. This also illustrates an alternate way to represent the “edit graph”.. It is embedded.

Every path is a common subsequence.

Every diagonal edge adds an extra element to common subsequence

**LCS Problem:** Find a path with maximum number of diagonal edges
Relationship Between Edit Distance and LCS Problem

\[ D = n + m - 2L \]
\[ D = 7 + 8 - 2 \times 5 = 5 \]
LCS Edit Graph
Computing LCS

Let $v_i = \text{prefix of } v \text{ of length } i: \; v_1 \ldots v_i$

and $w_j = \text{prefix of } w \text{ of length } j: \; w_1 \ldots w_j$

The length of $\text{LCS}(v_i, w_j)$ is computed by:

$$s_{i,j} = \max \begin{cases} 
  s_{i-1,j} \\
  s_{i,j-1} \\
  s_{i-1,j-1} + 1 \text{ if } v_i = w_j
\end{cases}$$

(It is the same definition that we presented earlier but shows that LCS has its own dynamic programming formulation independent of sequence alignment problem)
Computing LCS (cont’d)

\[ s_{i,j} = \text{MAX} \begin{cases} 
  s_{i-1,j} + 0 \\
  s_{i,j-1} + 0 \\
  s_{i-1,j-1} + 1, & \text{if } v_i = w_j 
\end{cases} \]
Every Path in the Grid Corresponds to an Alignment: Another Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V = AT - GT
W = ATCG –

0 1 2 3 4

0 1 2 3 4
LCS Runtime

• It takes $O(nm)$ time to fill in the $nxm$ dynamic programming matrix.

• Why $O(nm)$? The pseudocode consists of a nested “for” loop inside of another “for” loop to set up a $nxm$ matrix.
Similarity Definition Generalized

- We enlarge the alphabet $\Sigma$ to $\Sigma'$ including the space symbol ‘-’. Then for any two characters $x$ and $y$ in $\Sigma'$, we define a score or value obtained by aligning $x$ against $y$. For a given alignment of $S_1$ and $S_2$, let $S'_1$ and $S'_2$ denote the strings after the chosen insertion of spaces. And let $k$ denote the equal length of these two strings. Then value $V$ of alignment between $S'_1$ and $S'_2$ is defined as

$$
\sum_{i=1}^{k} \delta(S'_1(i), S'_2(i))
$$

where $\delta$ is the value or score associated with the pair of symbols $S'_1(i)$ and $S'_2(i)$. 
Maximization Problem

• In string similarity problems, the value of $\delta$ is usually set greater than zero for matched symbols and less than zero for symbol pairs that do not match or when a symbol is aligned with a ‘-‘ character.

• This reduces the problem to the problem of maximization of $V$ for all possible alignments.
Dynamic Programming Solution

• Let $V(i, j)$ be the optimal alignment of prefixes $S_1[1...i]$ and $S_2[1...j]$.

• Basis:

\[ V(0, j) = \sum_{k=1}^{j} \delta(-, S_2(k)) \]

\[ V(i, 0) = \sum_{k=1}^{i} \delta(S_1(k), -) \]

\[ V(0, 0) = 0 \]
Dynamic Programming Solution

• recurrence relation is:

\[ V(i, j) = \max[V(i - 1, j - 1) + \delta(S_1(i), S_2(j)), V(i - 1, j) + \delta(S_1(i), -), V(i, j - 1) + \delta(-, S_2(j))] \]

  replacement

  deletion

  insertion

The value of the optimal alignment is given by \( V(n, m) \)

Like for the computation of the edit distance, we can use a bottom-up method to compute the alignment matrix. The complexity is \( O(nm) \) since at each point we perform 3 comparisons, 3 look-up operations and 3 additional operations.
Dynamic Programming Solution

When mismatches are penalized by a constant $-\mu$, indels are penalized by some other constant $-\sigma$ and matches are rewarded with $+1$, the recurrence relation is

$$V(i, j) = \max[V(i - 1, j - 1) - \mu \text{ if } v_i \neq w_j, \quad \text{mismatch}]$$

$$V(i - 1, j - 1) + 1 \text{ if } v_i = w_j \quad \text{match}$$

$$V(i - 1, j) - \sigma, \quad \text{deletion}$$

$$V(i, j - 1) - \sigma] \quad \text{insertion}$$

The value of the optimal alignment is given by $V(n,m)$ which equals

$#\text{matches} - \mu. #\text{mismatches} - \sigma. #\text{indels}$

Note, the LCS problem is the Global Alignment problem with $\mu=0$ and $\sigma=0$

Like for the computation of the edit distance, we can use a bottom-up method to compute the alignment matrix. The complexity is $O(nm)$ since at each point we perform 3 comparisons, 3 look-up operations and 3 additional operations.
Maximum similarity path

• By setting up suitable pointers, once the matrix is computed, we can obtain a trace for the optimal alignment by constructing any path from the cell \((n,m)\) to the cell \((0,0)\).

• Also, the problem can be formulated as finding a maximum weighted path in a weighted acyclic graph similar to one discussed earlier. (In general, computing a longest path in arbitrary graph is NP complete).
Computation time and Storage

• The weights of the edges must correspond to specific values of $s$ for the pair of symbols. The algorithm takes $O(nm)$ space.

• This is quite expensive if the sequences are large.

• If one were interested only in the value of the alignment and not obtaining a trace, this could easily be done by keeping only the last two rows of the matrix to compute the next row.

• This will need only $O(n+m)$ space. Is it possible to reconstruct an alignment using only linear space?