Lecture 9:
Sequence Alignments

Study Chapter 6.4-6.8
Outline

• Edit Distances
• Longest Common Subsequence
• Global Sequence Alignment
• Scoring Matrices
• Local Sequence Alignment
• Alignment with Affine Gap Penalties
Review

- **Dynamic Programming** is a technique for computing recurrence relations efficiently by storing partial results.

- Three keys:
  1. Formulate the answer as a recurrence relation.
  2. Consider all instances of the recurrence at each step.
  3. Order evaluations so you will always have the needed partial results.
A Biological DP Problem

• How to measure the similarity between a pair of nucleotide or amino acid sequences
• In the Motif-Searching Problem (Chapter 4) we used Hamming distance as our measure
• Is Hamming distance the best measure?
• How can we distinguish matches that occur by chance from slightly modified patterns?
• What sorts of modifications should we allow?
Best Sequence Matches

- Depends on how you define “Best”
- Consider the two DNA sequences $v$ and $w$:

\[
\begin{align*}
 v & : \text{ATATATATAT} \\
 w & : \text{TATATATATA}
\end{align*}
\]

- The Hamming distance: $d_H(v, w) = 8$ is large but the sequences are very similar
- What if we allowed insertions and deletions?
Allowing Insertions and Deletions

By shifting one sequence over one position:

\[ \nu : \text{ATATATAT} \quad \text{--} \]
\[ \omega : \text{--TATATATA} \]

- The edit distance: \( d_H(\nu, \omega) = 2 \).
- Hamming distance neglects insertions and deletions in DNA.
Levenshtein (1966) introduced the notion of an “edit distance” between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) to transform one string into the other.

\[ d(v, w) = \text{MIN number of elementary operations to transform } v \rightarrow w \]

(But, he gave no solution)
Edit Distance vs Hamming Distance

Hamming distance always compares $i^{th}$ letter of $v$ with $i^{th}$ letter of $w$

$V = \text{ATATATAT}$ \hspace{1cm} $W = \text{TATATATA}$

Hamming distance: $d(v, w) = 8$

Computing Hamming distance is a trivial task

	Hamming distance always compares $i^{th}$ letter of $v$ with $i^{th}$ letter of $w$

Edit distance may compare $i^{th}$ letter of $v$ with $j^{th}$ letter of $w$

$V = - \text{ATATATAT}$ \hspace{1cm} $W = \text{TATATATA}$

Edit distance: $d(v, w) = 2$

Computing edit distance is a non-trivial task

Just one shift

Lines them up
Edit Distance: Example

TGCATAT \rightarrow ATCCCGAT in 5 steps

- TGCATAT \rightarrow (DELETE last T)
- TGCATA \rightarrow (DELETE last A)
- TGCAT \rightarrow (INSERT A at front)
- ATGCAT \rightarrow (SUBSTITUTE C for 3rd G)
- ATCCCAT \rightarrow (INSERT G before last A)
- ATCCGAT \rightarrow (Done)

What is the edit distance? 5?
Edit Distance: Example (cont’d)

TGCATAT $\rightarrow$ ATCCCGAT in 4 steps

TGCATAT $\rightarrow$ (INSERT $A$ at front)
ATGCA$T$AT $\rightarrow$ (DELETE 6$^{th}$ $T$)
ATGC$A$AT $\rightarrow$ (SUBSTITUTE $G$ for 5$^{th}$ $A$)
ATGC$C$GAT $\rightarrow$ (SUBSTITUTE $C$ for 3$^{rd}$ $G$)
ATCCGAT (Done)

Is 4 the minimum edit distance? 3?

A little jargon: Since the effect of insertion in one string can be accomplished via a deletion in the other string these two operations are quite similar. Often algorithms will consider them together as a single operation called INDEL
Longest Common Subsequence

- A special case of edit distance where no substitutions are allowed
- A subsequence need not be contiguous, but order must be preserved
  Ex. If v = ATTAGCTA then AGCA and TTTA are subsequences of v, but TGTT and ACGA are not
- The length of the LCS, s, is related to the strings edit distance, d, by:

\[ d(u,w) = \text{len}(v) + \text{len}(w) - 2 \cdot s(u,w) \]
LCS as a Dynamic Program

- All possible possible alignments can be represented as a path from the string’s beginning (source) to its end (destination)
- Horizontal edges add gaps in v. Vertical edges add gaps in w. Diagonal edges are a match
- Notice that we’ve only included valid diagonal edges in our graph
Various Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
V & A & T & _ & G & T & T & A & T \\
W & A & T & C & G & T & _ & A & _ & C \\
0 & 1 & 2 & 3 & 4 & 5 & 5 & 6 & 6 & 7 \\
\end{array}
\]

Path:
(0,0), (1,1), (2,2), (2,3), (3,4), (4,5), (5,5), (6,6), (7,6), (7,7)
Various Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 2 & 3 & 4 & 5 & 6 & 6 & 7 \\
v & A & T & _ & G & T & T & A & _ & T \\
w & A & T & C & G & _ & T & A & C & _ \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 7 \\
\end{array}
\]

Path:
(0,0), (1,1), (2,2), (2,3), (3,4), (4,4), (5,5), (6,6), (6,7), (7,7)
Even Bad Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

Path:

\[(0,0), (0,1), (0,2), (0,3), (0,4), (0,5), (1,6), (2,6), (3,6), (4,6), (5,6), (6,6), (7,6), (7,7)\]
What makes a Good Alignment?

- Using as many diagonal segments (matches) as possible
- The end of a good alignment from (j...k) begins with a good alignment from (i..j)
- Same as Manhattan Tourist problem, where the sites are the diagonal streets!
- Set diagonal street weights = 1, and horizontal and vertical weights = 0
Alignment: Dynamic Program

\[ S_{i,j} = \max \left\{ S_{i-1,j-1} + 1 \quad \text{if} \quad v_i = w_i, \right. \]

\[ \left. S_{i-1,j}, \quad S_{i,j-1} \right\} \]
Dynamic Programming Example

<table>
<thead>
<tr>
<th></th>
<th>W</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>v</td>
<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>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initialize 1st row and 1st column to be all zeroes.

Or, to be more precise, initialize 0th row and 0th column to be all zeroes.
Dynamic Programming Example

\[
s_{i,j} = \begin{cases} 
    s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\
    s_{i-1,j} & \\
    s_{i,j-1} & 
\end{cases}
\]

<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>v</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>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Programming Example

\[ s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\ s_{i-1,j} & \\ s_{i,j-1} & \end{cases} \]
### Dynamic Programming Example

#### Matrix

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

#### Equation

\[
s_{i,j} = \begin{cases} 
  s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\
  s_{i-1,j} & \\
  s_{i,j-1} & 
\end{cases}
\]
Dynamic Programming Example

\[
s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\
  s_{i-1,j} & \\
  s_{i,j-1}
\end{cases}
\]
### Dynamic Programming Example

<table>
<thead>
<tr>
<th></th>
<th>w</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>v</td>
<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>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
s_{i,j} = \max \begin{cases} 
    s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\
    s_{i-1,j} \\
    s_{i,j-1}
\end{cases}
\]
### Dynamic Programming Example

The dynamic programming example illustrates the process of aligning two sequences, `v` and `w`, where `v` and `w` are strings of nucleotides (A, T, C, G). The alignment is done using a matrix with entries `s_{i,j}` that represent the score of aligning prefixes of `v` and `w` up to positions `i` and `j`, respectively.

The recurrence relation for `s_{i,j}` is:

\[
s_{i,j} = \begin{cases} 
  s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\
  s_{i-1,j} & \\
  s_{i,j-1} & 
\end{cases}
\]

The matrix and corresponding arrows illustrate the dynamic programming solution path, where each cell contains the maximum score that can be achieved for aligning prefixes of `v` and `w` up to that point.

Here is the matrix:

<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>v</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>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>T</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>T</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

The arrows indicate the path taken to compute the scores in the matrix, reflecting the dynamic programming solution.
Dynamic Programming Example

\[
s_{i,j} = \begin{cases} 
    s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\
    s_{i-1,j} & \\
    s_{i,j-1} & 
\end{cases}
\]
### Dynamic Programming Example

**Matrix Representation**

<table>
<thead>
<tr>
<th></th>
<th>w</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>v</td>
<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>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Formulation**

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

\[w = \text{ATCGT-A-C}\]

\[v = \text{AT-GTTAT-}\]
Alignment: Backtracking

<table>
<thead>
<tr>
<th>Arrows</th>
<th>Show where the score came from.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If from the top</td>
</tr>
<tr>
<td></td>
<td>If from the left</td>
</tr>
<tr>
<td></td>
<td>If $v_i = w_j$</td>
</tr>
</tbody>
</table>

Our table only keeps track of the longest common subsequence so far. How do we figure out what the subsequence is?

We'll need a second table to keep track of the decisions we made... and we'll use it to backtrack to our answer.
def LCS(v, w):
    s = zeros((len(v)+1,len(w)+1), Int)
    b = zeros((len(v)+1,len(w)+1), Int)
    for i in xrange(1,len(v)+1):
        for j in xrange(1,len(w)+1):
            # find best score at each vertex
            if (v[i-1] == w[j-1]):
                s[i,j] = max(s[i-1,j], s[i,j-1], s[i-1,j-1] + 1)
            else:
                s[i,j] = max(s[i-1,j], s[i,j-1])
            # save direction by comparing best scores
            if (s[i,j] == s[i,j-1]):
                b[i,j] = 1             # from above
            elif (s[i,j] == s[i-1,j]):
                b[i,j] = 2             # from left
            else:
                b[i,j] = 3             # along diagonal
    return (s[i,j], b)
def PrintLCS(b,v,i,j):
    if ((i == 0) or (j == 0)):
        return
    if (b[i,j] == 3):
        PrintLCS(b,v,i-1,j-1)
        print v[i-1],
    else:
        if (b[i,j] == 2):
            PrintLCS(b,v,i-1,j)
        else:
            PrintLCS(b,v,i,j-1)
Changing the Scoring

• Longest Common Subsequence (LCS) problem
  – the simplest form of sequence alignment
  – allows only insertions and deletions (no mismatches).
• In the LCS Problem, we scored 1 for matches and 0 for indels
• Consider penalizing indels and mismatches with negative scores
• Simplest scoring schema:
  +1 : match premium
  -\( \mu \) : mismatch penalty
  -\( \sigma \) : indel penalty
Simple Scoring

- When mismatches are penalized by $-\mu$
- indels are penalized by $-\sigma$
- matches are rewarded with +1

The resulting score is:

$$\text{score} = \#\text{matches} - \mu (\#\text{mismatches}) - \sigma (\#\text{indels})$$
The Global Alignment Problem

Find the best alignment between two strings under a given scoring schema

**Input**: Strings \( v \) and \( w \) and a scoring schema

**Output**: Alignment of maximum score

\[
\begin{align*}
\uparrow \text{ or } \rightarrow &= -\sigma \\
&= 1 \text{ if match} \\
&= -\mu \text{ if mismatch}
\end{align*}
\]

\[
s_{i,j} = \max \begin{cases}
s_{i-1,j-1} + 1 \text{ if } v_i = w_j \\
s_{i-1,j-1} -\mu \text{ if } v_i \neq w_j \\
s_{i-1,j} - \sigma \\
s_{i,j-1} - \sigma
\end{cases}
\]

\( \mu \): mismatch penalty

\( \sigma \): indel penalty
Scoring Matrices

To generalize scoring, consider a $4+1 \times 4+1$ scoring matrix $\delta$.

In the case of an amino acid sequence alignment, the scoring matrix would be a $(20+1) \times (20+1)$ size.

The addition of 1 is to include the score for comparison of a gap character “-”.

This will simplify the algorithm as follows:

$$s_{i,j} = \max\left\{ s_{i-1,j-1} + \delta(v_i, w_j), s_{i-1,j} + \delta(v_i, -), s_{i,j-1} + \delta(-, w_j) \right\}$$
Making a Scoring Matrix

- Scoring matrices are created based on biological evidence.
- Alignments can be thought of as two sequences that differ due to mutations.
- Some of these mutations have little effect on the protein’s function, therefore some penalties, $\delta(v_i, w_j)$, are less harsh than others.
Notice that although R (arginine) and K (lysine) are different amino acids, they have a positive score. Why? They are both positively charged amino acids and hydrophilic may not greatly change function of protein.
Conservation

• Amino acid changes that tend to preserve the electro-chemical properties of the original residue
  − Polar to polar
    • aspartate $\rightarrow$ glutamate
  − Nonpolar to nonpolar
    • alanine $\rightarrow$ valine
  − Similarly behaving residues
    • leucine to isoleucine
Scoring matrices

- Amino acid substitution matrices
  - PAM
  - BLOSUM

- DNA substitution matrices
  - DNA is less conserved than protein sequences
  - Less effective to compare coding regions at nucleotide level
PAM

- Point Accepted Mutation (Dayhoff et al.)
- 1 PAM = PAM\(_1\) = 1% average change of all amino acid positions
  - After 100 PAMs of evolution, not every residue will have changed
    - some residues may have mutated several times
    - some residues may have returned to their original state
    - some residues may not changed at all
PAM\(_x\)

- \(PAM_x = PAM_1^x\)
  - \(PAM_{250} = PAM_1^{250}\)
- \(PAM_{250}\) is a widely used scoring matrix:

<table>
<thead>
<tr>
<th></th>
<th>Ala</th>
<th>Arg</th>
<th>Asn</th>
<th>Asp</th>
<th>Cys</th>
<th>Gln</th>
<th>Glu</th>
<th>Gly</th>
<th>His</th>
<th>Ile</th>
<th>Leu</th>
<th>Lys</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ala</td>
<td>13</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>...</td>
</tr>
<tr>
<td>Arg</td>
<td>3</td>
<td>17</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>...</td>
</tr>
<tr>
<td>Asn</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>Asp</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>11</td>
<td>1</td>
<td>7</td>
<td>10</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>Cys</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>52</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>Gln</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trp</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Tyr</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>Val</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>15</td>
<td>10</td>
<td>...</td>
</tr>
</tbody>
</table>
BLOSUM

• **Blocks Substitution Matrix**
• Scores derived from *observations* of the frequencies of substitutions in blocks of local alignments in related proteins
• Matrix name indicates evolutionary distance
  – BLOSUM62 was created using sequences sharing no more than 62% identity
The Blossum50 Scoring Matrix

|   | A | R | N | D | C | Q | E | G | H | I | L | K | M | F | P | S | T | W | Y | V | B | Z | X |
| A | 5 | -2 | -1 | -2 | -1 | -1 | -1 | 0 | -2 | -1 | -2 | -1 | -1 | -1 | -1 | 1 | 1 | 0 | -3 | 2 | 0 | -2 | -1 | -1 | -5 |
| R | -2 | 7 | 1 | -2 | 4 | 1 | 0 | 3 | 0 | -4 | -3 | -3 | -2 | -3 | -3 | -1 | -1 | -1 | -3 | -1 | -3 | -1 | -3 | 1 | 0 | 1 | 5 |
| N | -1 | 1 | 7 | 2 | 2 | 0 | 0 | 0 | 1 | 3 | 4 | 0 | 2 | 4 | 2 | 1 | 0 | 2 | 3 | 4 | 0 | 1 | 5 | 1 | 2 | 0 | 1 |
| D | -2 | -2 | 2 | 8 | -4 | 0 | 2 | -1 | -1 | 4 | -4 | -1 | 4 | -5 | 1 | 0 | -1 | -5 | 3 | 4 | 5 | 1 | 1 | -5 | -5 | -5 | -5 |
| C | -1 | 4 | -2 | 4 | 13 | -3 | -3 | 3 | 3 | 2 | -2 | 3 | -2 | -2 | 4 | -1 | -1 | -4 | -1 | -5 | -3 | -1 | -3 | -3 | -2 | -2 |
| Q | -1 | 1 | 0 | 0 | 5 | 7 | 2 | 2 | 1 | 3 | 2 | 2 | 0 | 4 | -1 | 0 | 1 | -1 | -1 | 1 | 5 | 0 | 4 | 1 | 5 | 1 | 5 |
| E | -1 | 0 | 0 | 2 | -3 | 2 | 6 | 3 | 0 | -4 | -3 | -1 | -2 | -3 | -1 | -1 | -1 | -3 | -2 | -3 | 1 | 1 | 5 | 1 | 5 | 1 | 5 |
| G | 0 | -3 | 0 | -1 | -3 | -2 | 3 | -8 | -2 | 4 | -4 | -2 | 3 | -4 | -2 | -2 | -0 | 2 | -3 | -3 | 4 | -1 | -2 | -2 | 5 | 1 | 5 |
| H | -2 | 0 | 1 | -1 | 3 | 1 | 0 | 2 | 10 | -4 | 0 | 0 | 2 | 1 | -2 | 1 | -2 | 3 | 2 | -4 | 0 | 0 | 1 | 5 | 1 | 5 |
| I | -1 | 1 | -4 | -3 | -4 | -2 | -3 | -4 | -4 | -4 | -5 | 2 | 3 | 0 | 2 | 0 | -3 | -3 | -1 | -3 | -1 | -4 | -4 | -3 | -1 | 5 |
| L | -2 | 3 | -4 | -2 | 3 | -2 | 3 | -4 | 3 | -2 | 5 | 3 | 3 | 3 | 1 | -4 | 3 | 1 | -2 | -1 | 1 | 4 | -3 | -1 | 5 | 5 |
| K | -1 | 3 | 0 | 1 | -3 | 2 | 1 | 2 | 0 | 3 | 3 | 6 | 2 | 4 | 1 | 0 | 1 | -3 | 2 | 3 | 0 | 1 | 5 | -5 | -5 | -5 |
| M | -1 | 2 | -2 | 2 | 0 | -2 | -3 | -1 | 2 | 3 | -2 | 7 | 0 | -3 | 2 | -1 | -0 | 1 | -3 | -1 | -1 | 5 | 1 | 5 | 1 | 5 |
| F | -3 | -3 | -4 | -5 | -2 | 4 | -3 | 4 | -4 | -1 | 0 | 1 | 4 | 0 | 8 | 4 | 3 | -2 | 4 | 1 | 4 | -4 | 2 | 5 | 1 | 5 |
| P | -1 | 3 | -2 | 1 | -4 | -1 | 2 | 1 | -2 | 3 | 4 | 1 | 1 | 3 | 4 | 10 | -1 | -1 | -4 | -3 | 3 | 2 | -1 | 2 | 5 | 5 |
| S | 1 | 1 | 0 | -1 | 0 | -1 | 0 | -1 | 1 | -3 | -3 | 0 | -2 | 3 | -1 | 5 | 1 | 2 | -4 | -2 | -2 | 0 | 0 | -1 | 1 | 5 |
| T | 0 | -1 | 0 | 1 | -1 | -1 | -2 | 2 | 1 | 1 | 1 | 1 | 2 | -1 | 2 | 2 | 5 | 3 | -2 | 0 | 0 | 1 | 0 | 5 | 1 | 5 |
| W | -3 | -3 | -4 | -5 | -5 | -4 | -5 | -3 | -3 | -2 | -3 | -1 | 1 | -4 | 4 | 3 | 15 | 2 | 3 | 5 | -2 | 3 | 5 | 2 | -3 | 5 |
| Y | -2 | -1 | -2 | -3 | -3 | -1 | 2 | 3 | -2 | 1 | -2 | 0 | 4 | 3 | -2 | 2 | 2 | 8 | -1 | -3 | -2 | 1 | 5 | 1 | 5 | 1 | 5 |
| V | 0 | 3 | -3 | -4 | -1 | -3 | -4 | 4 | 4 | 1 | 3 | 1 | -1 | 3 | -2 | 0 | -3 | 1 | 5 | 4 | -3 | 1 | 5 | 1 | 5 | 1 | 5 |
| B | -2 | -1 | 4 | 5 | -3 | 0 | 1 | -1 | 0 | -4 | 0 | -3 | -4 | -2 | 0 | 0 | -5 | -3 | 4 | 5 | 2 | 1 | 5 | -5 | -5 | -5 |
| Z | -1 | 0 | 0 | -1 | 3 | 4 | 5 | -2 | 0 | 3 | 3 | 1 | -4 | -1 | 0 | -1 | -2 | 2 | -3 | 2 | 5 | 1 | -5 | -5 | -5 | -5 |
| X | -1 | 1 | -1 | 1 | 2 | 1 | -1 | 1 | 1 | 1 | 2 | 2 | 1 | 0 | -3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

2/9/15  Comp 555  Spring 2015  41
Local vs. Global Alignment

- The **Global Alignment Problem** tries to find the longest path between vertices \((0,0)\) and \((n,m)\) in the edit graph.

- The **Local Alignment Problem** tries to find the longest path among paths between arbitrary vertices \((i,j)\) and \((i', j')\) in the edit graph.
Local vs. Global Alignment

- The **Global Alignment Problem** tries to find the longest path between vertices \((0,0)\) and \((n,m)\) in the edit graph.

- The **Local Alignment Problem** tries to find the longest path among paths between *arbitrary vertices* \((i,j)\) and \((i',j')\) in the edit graph.

- In the edit graph with negatively-scored edges, Local Alignment may score higher than Global Alignment.
Local vs. Global Alignment (cont’d)

- **Global Alignment**
  
  \[\text{--T--CC-C-AGT--TATGT-CAGGGGACACG--A--GCATGCAGA--GAC} \]
  
  \[\text{AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C} \]

- **Local Alignment** — better alignment to find conserved segment

  \[\text{tccCAGTTATGTCA}\text{Ggggacacgagcatgcagagac} \]
  
  \[\text{aattgcgcgcgtcgc} \text{tttttcagCAGTTATGTCA}\text{Gatc} \]
Local Alignment: Example

Global alignment

Local alignment

Compute a “mini” Global Alignment to get Local
Local Alignments: Why?

• Two genes in different species may be similar over short conserved regions and dissimilar over remaining regions.

• Example:
  – Homeobox genes have a short region called the *homeodomain* that is highly conserved between species.
  – A global alignment would not find the homeodomain because it would try to align the ENTIRE sequence.
The Local Alignment Problem

- **Goal**: Find the best local alignment between two strings
- **Input**: Strings $v$, $w$ and scoring matrix $\delta$
- **Output**: Alignment of substrings of $v$ and $w$ whose alignment score is maximum among all possible alignment of all possible substrings
Local Alignment: Example

Global alignment

Local alignment

Compute a “mini” Global Alignment to get Local
Local Alignment: Example
Local Alignment: Example
Local Alignment: Example
Local Alignment: Example
Local Alignment: Example
Local Alignment: Running Time

- Long run time $O(n^4)$:
  - In the grid of size $n \times n$ there are $\sim n^2$ vertices $(i,j)$ that may serve as a source.
  - For each such vertex computing alignments from $(i,j)$ to $(i',j')$ takes $O(n^2)$ time.
- This can be remedied by giving free rides
Local Alignment: Free Rides

Vertex (0,0)

The dashed edges represent the free rides from (0,0) to every other node.

Yeah, a free ride!
The Local Alignment Recurrence

• The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment (not $s_{n-1,m-1}$).

• The recurrence:

$$s_{i,j} = \max \begin{cases} 
0 & \\
 0 \cdot 1_{i,j-1} + \delta(v_i, w_j) \\
 0 \cdot 1_{i-1,j} + \delta(v_i, -) \\
 s_{i-1,j-1} + \delta(-, w_j) \\
 s_{i,j-1} + \delta(-, w_j) \\
 \end{cases}$$

Notice there is only this change from the original recurrence of a Global Alignment.
The Local Alignment Recurrence

• The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.

• The recurrence:

$$s_{i,j} = \max \begin{cases} 
0 \\
 s_{i-1,j-1} + \delta(v_i, w_j) \\
 s_{i-1,j} + \delta(v_i, -) \\
 s_{i,j-1} + \delta(-, w_j)
\end{cases}$$

Power of ZERO: there is only this change from the original recurrence of a Global Alignment - since there is only one “free ride” edge entering into every vertex
Next Time

• We finish Dynamic programming
• Alignment with Gap Penalties
• Multiple Alignment problem
• Gene Prediction
  – Statistical Approaches
  – Similarity Approaches
• Splice Alignments