Sequence Alignment II CH364C/CH391L Bioinformatics Marcotte Spring 2013

Aligning 2 sequences of length *n*, the number of possible alignments (allowing gaps) can be found as follows:

2 aligned sequences of symbols with gaps:

ACGTACGT ACGT or more generically, $x_1x_2x_3x_4x_5x_6x_7x_8 - x_9x_{10}x_{11}x_{12}$

ACG ACGTAACGT $y_1y_2y_3 - y_4y_5y_6y_7y_8y_9y_{10}y_{11}y_{12}$

can always be rewritten alternating symbols from the top & bottom rows, ignoring gaps:

$x_1y_1x_2y_2x_3y_3x_4x_5y_4x_6y_5x_7y_6x_8y_7y_8x_9y_9x_{10}y_{10}x_{11}y_{11}x_{12}y_{12}$

Because the indices are implicit in the order the symbols are written as, this is equivalent to: **xyxyxyxxyxyxyxyyyxyxyxyxy**

This composite sequence has length 2*n*. The number of possible ways to intercalate two sequences of length *n* to give a single sequence of length 2*n* is equivalent to calculating the number of permutations of *n x*'s and *n y*'s, which is:

$$\binom{2n}{n} = \frac{(2n)!}{(n!)(n!)} \cong \frac{2^{2n}}{\sqrt{\pi n}}$$

where the simplification on the right side of the equation comes from the use of Stirling's approximation of the factorial:

$$x! \cong \sqrt{2\pi}\, x^{x+\frac{1}{2}} e^{-x}$$

As a reminder for rusty mathematicians, the notation $\binom{x}{y}$, which you would read as "*x* choose *y*", means: from *x* objects, how many ways can a subset of *y* be chosen? The generic answer is $\frac{x!}{y!(x-y)!}$.

Obviously, we can't generate every one of these alignments and score each for quality to choose the best. Even for relatively short sequences, the number of possible alignments grows out of hand (10^{59} possible alignments for *n*=100, and 10^{600} for *n*=1000), so instead, we'll use an algorithm to find the optimal alignment (or set of alignments). The particular class of algorithm we'll use is called *dynamic programming*, which refers to a set of algorithms that allow optima to be found for problems that can be defined in a recursive manner. That is, the problems are broken into subproblems, which are in turn broken into subproblems, etc, until the simplest subproblems can be solved. For sequence alignments, this sequential dependency takes a form where the choice of optimal alignment of a sequence of length *n* is found from the solution to the optimal alignment of a sequence of length *n*-1 plus the alignment of the *n*th symbol, and the

optimal alignment of the *n*-1 case is a function of the *n*-2 case, and so on. Just as an aside, dynamic programming was developed by Richard Bellman 40-50 years ago, but then "rediscovered" by biologists aligning sequences in the 1970's.

We'll distinguish 2 types of alignments that we could make: *global* and *local*
Global alignments will require a forced match between every symbol of one string with some symbol (or gap) of the second string, e.g.

```
ACGTTATGCATGACGTA
-C---ATGCAT----T-
```

Local alignments will correspond to the best matching subsequences (including gaps). For the above example, this corresponds to:

```
     ATGCAT
     ATGCAT
```

We'll start with global alignments. For biological sequences, this is known as the *Needleman-Wunsch algorithm*, from a paper by Saul Needleman and Christian Wunsch in the 1970 *Journal of Molecular Biology*, vol. 48, 443-453, later improved by Gotoh, in *JMB* 162, 705-708 (1982).

The general idea is to construct a "path" matrix indicating the scores of different alignments between the two sequences. The matrix will be built up one element at a time. Each element of the path matrix will contain the score $F(i,j)$ of the best alignment between the subsequence of x from 1 to i with the subsequence of y from 1 to j. The path matrix is first initialized by setting $F(0,0) = 0$.

We'll follow an example from Durbin *et al.*. Here's the initial path matrix F:

```
                  i=0                 x                   i=n
                  H   E   A   G   A   W   G   H   E   E
              0
   P <-- j=0               The path matrix will be
   A                       filled from the top left
   W                       to the bottom right
y  H
   E
   A
   E <-- j=m
```

The basic idea is that given $F(i-1,j-1)$, $F(i-1,j)$ and $F(i,j-1)$, one can calculate $F(i,j)$. One of three events is possible: depending on the event, $F(i,j)$ is set equal to:

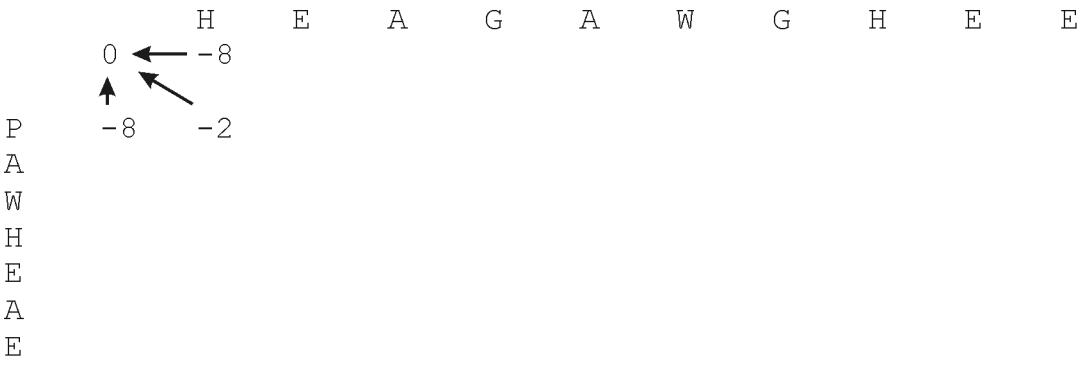
x_i is aligned to y_j	$F(i-1,j-1) + s(x_i,y_j)$	
x_i is aligned to a gap	$F(i-1,j) - d$	(e.g., for a linear gap function)
y_j is aligned to a gap	$F(i,j-1) - d$	(for this example, $d = 8$)

The event chosen is the one that leads to the largest score at $F(i,j)$. This process is repeated iteratively to populate the path matrix. At each step, we also keep track of which event was chosen (e.g. which previous cell contributed its score to the current score).

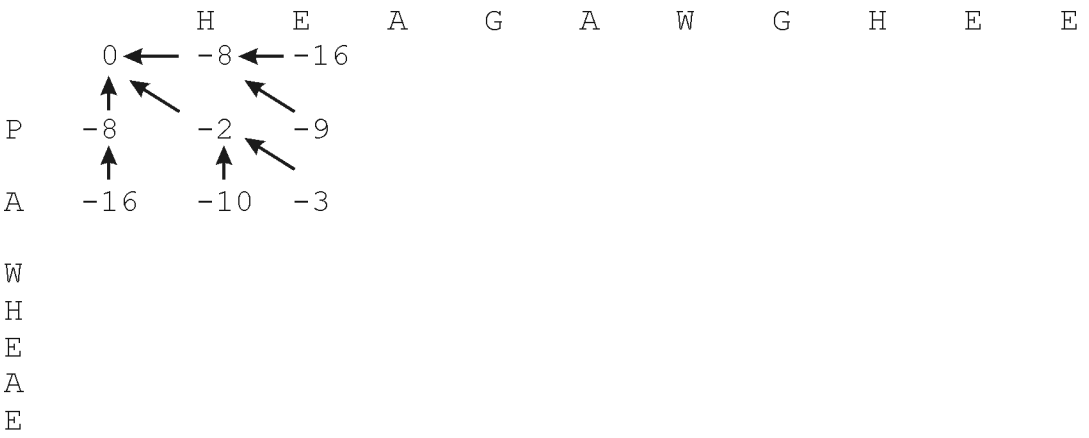
Along the edges of the matrix, we have to include some special conditions. Along the top edge, we lack $F(i\text{-}1,j\text{-}1)$ and $F(i,j\text{-}1)$ elements. Since positions along the top edge are equivalent to aligning to all gaps in the *y* sequence, each top boundary element $F(i,0)$ is set to $-id$. By a similar logic, the leftmost boundary elements $F(0,j)$ are set to $-jd$.

The bottom right element of the array $F(n,m)$

So, in our example, above, the first elements added are:



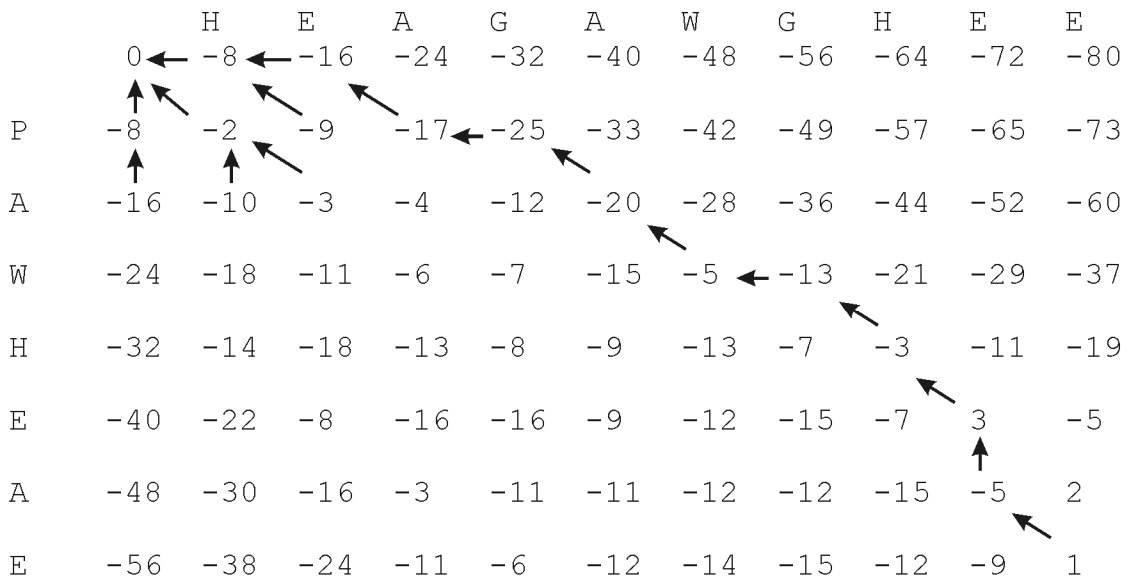
The next elements are:



and so on, until the entire matrix is filled. At this point, the optimal alignment is found by a *traceback* process, following the arrows from the bottom right element back to the top left element to define the alignment. At each step in the traceback, the symbols corresponding to that move are added to the alignment. For example, a vertical move means to add a gap in *x* and a symbol in *y*, a horizontal move is a gap in *y* and a symbol in *x*, and a diagonal move is a symbol from *x* with one of *y*. The alignment "grows" from right to left. From this example, the final alignment is:

```
HEAGAWGHE-E
--P-AW-HEAE
```

derived from the final path matrix, which looks like (with many off-path arrows omitted):



This approach finds one optimal global alignment (but more than one may exist) between the two sequences.

In reality, we often would rather have a local alignment, finding just those subsequences that align well. This approach (in biology) is named the *Smith-Waterman algorithm* after Temple Smith & Mike Waterman, *Journal of Molecular Biology* vol. 147, 195-197 (1981). The approach is essentially the same as for the global alignment, but an extra option is possible for each cell: if all of the other options provide negative scores, the cell can receive a score $F(i,j) = 0$, corresponding to the starting position of a new local alignment. We can eliminate the boundary conditions (all boundary cells are assigned scores of zero). Lastly, an alignment can now end anyplace in the path matrix. So, once the path matrix is constructed, the cell with the highest score is chosen as the starting point for the traceback (rather than choosing the bottom right element in the global alignment.) The traceback proceeds as for the global alignment (e.g., following the arrows up and to the left) and ends when a cell with zero score is reached.

Below is the local alignment matrix for the earlier set of two amino acid sequences. The optimal subsequence alignment is:

```
AWGHE
AW-HE
```

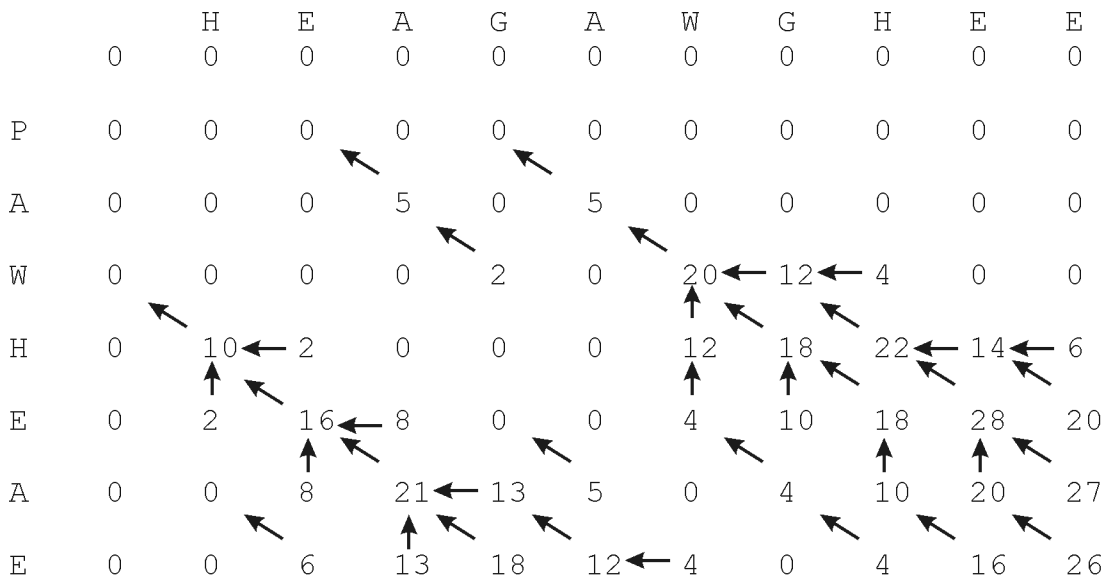
which has score of 28. Note that with this method it is easy to find many high-scoring suboptimal alignments, such as:

```
HEA
HEA
```

which scores 21.

This type of search for suboptimal alignments also turns out to be a good method to find repeating sequences.

The path matrix $F(i,j)$:



& the best local alignment highlighted: