

## Sequence Alignment

### 1 Background

Over 300 complete genomes have been sequenced – everything from bacteria to mammals and everything in between – and more continue to be sequenced every day. (*Fun facts: The amoeba has the largest known genome, with ~670 billion bps. The human genome has just under 3 billion bps.*) One of the reasons we sequence so many genomes is to study evolution through comparison. Genomes evolve through mutation of the DNA sequence, which can occur as a result of environmental stimuli such as radiation or chemicals, or from random errors in the DNA replication process (all of these must happen in the germ line, or reproductive cells, in order to be passed on to the next generation).

Evolutionary events at the sequence level, or “sequence edits”, fall under two main categories: single letter changes, and large-scale changes. Single letter changes can be substitutions, where one letter is changed to another; deletions, where one letter is removed; or insertions, where one letter is added.

Substitution: ATCG → A**A**CG

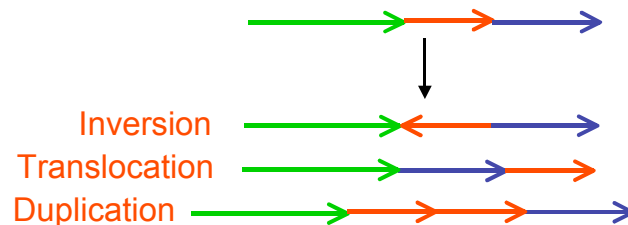
Deletion: A**T**CG → ACG

Insertion: ACG → A**T**CG

Large-scale changes can be inversions, where a segment of DNA is transformed into its reverse complement; translocations, where two or more segments are reordered; and duplications, where a segment of DNA is copied and inserted either in the same or distant region of the genome. In an inversion, for example:

ACGG**TT**TA becomes ACA**ACC**TA  
TG**CCAA**AT TG**TTGG**AT

Below is a graphical representation of the three types of large scale changes.



Different parts of the genome mutate with different frequency due to biological importance. Regions that are important for the function of that organism are less likely to accept changes to the DNA sequence and will evolve more slowly, while “junk” or non-functional DNA will evolve at a faster rate. Therefore, sequence conservation between species implies that the sequence has important biological function. This is why we study multiple genomes, and why sequence alignment is an important tool in computational biology.

## 2 Introduction to Sequence Alignment

Basic problem formulation: Place gaps in sequences (but never align two gaps) to reveal similarity between them.

### 2.1. Definition

Given two strings  $x = x_1, \dots, x_M$  and  $y = y_1, \dots, y_N$ , an alignment is defined as an assignment of gaps to positions  $0, \dots, M$  in  $x$  and  $0, \dots, N$  in  $y$  so as to line up each letter in one sequence with either a letter or a gap in the other sequence. This will result in two sequences of equal length.

We can also define sequence alignment based on the concept of “edit distance” (see next section).

### 2.2. Evaluating alignments

Many alignments can be constructed for the same pair of sequences, so we need a way to evaluate them. For example, the following are all valid alignments of the strings AGGCTAG and AGCGAAGTTT :

```

AGGCTAGTT-   AGGCTA-GTT-   AGGC-TA-GTT-
AGCGAAGTTT   AG-CGAAGTTT   AG-CG-AAGTTT
    
```

Alignments are evaluated based on the number of matches, mismatches, and gaps they introduce between sequences. If matches are given a positive value  $m$ , mismatches are given a negative value  $s$ , and gaps are given a negative value  $d$ , we can define a scoring function:

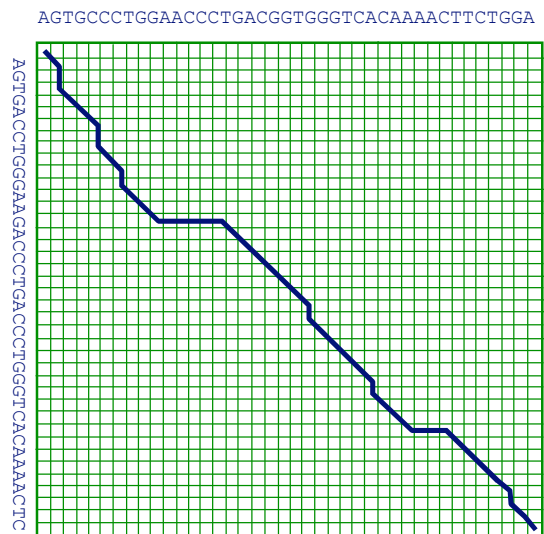
$$F = (\# \text{ matches} \times m) - (\# \text{ mismatches} \times s) - (\# \text{ gaps} \times d).$$

This function can also be said to describe the “edit distance” between two sequences; that is, the number of changes that have been made to one sequence to transform it into the other. *Minimal edit distance gives us an alternate definition of sequence alignment: Given 2 strings  $x$  and  $y$ , find the minimum number of edits to transform one to the other.*

### 2.3. Computing the best alignment

If  $x$  is placed across the top and  $y$  is placed down the left of a matrix, then an alignment is a path through the matrix (see figure below), where

- 1 diagonal move = alignment of 2 letters (either match or mismatch)
- 1 downwards move = alignment of a letter of  $y$  with a gap in  $x$
- 1 right-horizontal move = alignment of a letter of  $x$  with a gap in  $y$ .



There are  $\gg 2^N$  possible paths, so it's not feasible to do an exhaustive search. In order to find the best alignment (according to some scoring function), we need to make use of **dynamic programming**, as well as the **additive property of alignments**. Alignments are additive, meaning that if  $x_1, \dots, x_i$  aligns to  $y_1, \dots, y_j$ , and  $x_{i+1}, \dots, x_M$  aligns to  $y_{j+1}, \dots, y_N$ , then  $x_1, \dots, x_M$  will align to  $y_1, \dots, y_N$  with a score equal to the sum of the two sub-scores. More specifically,

$$F(x[1:M], y[1:N]) = F(x[1:i], y[1:j]) + F(x[i+1:M], y[j+1:N]).$$

### 3 Dynamic programming for sequence alignments

Dynamic programming is a fast way to solve a large problem by breaking the problem into smaller, more feasible sub-problems. There are a few requirements for dynamic programming to work:

1. The problem must have a polynomial number of subproblems.
2. The original problem must be one of the subproblems.
3. Each subproblem must be easily solved from smaller subproblems.

In the case of sequence alignment, the subproblems are the alignments of subsequences (a total of  $(m \times n)$  possible subsequences) of which the full-length alignment is also a subproblem, and subproblems are solved using the additive property of alignments.

#### 3.1. Using dynamic programming for sequence alignments

Let  $F(i, j)$  be the optimal score for aligning  $x_1 \dots x_i$  to  $y_1 \dots y_j$ . There are then three possible end-cases:

1.  $x_i$  aligns to  $y_j$  (match or mismatch)
2.  $x_i$  aligns to a gap
3.  $y_j$  aligns to a gap.

This leaves us with the following three alignment sub-problems:

1.  $x_1 \dots x_{i-1}$  to  $y_1 \dots y_{j-1}$
2.  $x_1 \dots x_{i-1}$  to  $y_1 \dots y_j$
3.  $x_1 \dots x_i$  to  $y_1 \dots y_{j-1}$ .

It also leaves us with three possibilities for  $F(i, j)$ :

1.  $F(i, j) = F(i-1, j-1) + \{m \text{ if } x_i = y_j, s \text{ otherwise} \}$
2.  $F(i, j) = F(i-1, j) - d$
3.  $F(i, j) = F(i, j-1) - d$

We will always choose the maximum of these three, that is:

$$F(i, j) = \max \{ F(i-1, j-1) + s(x_i, y_j), \\ F(i-1, j) - d, \\ F(i, j-1) - d \}, \text{ where } s(x_i, y_j) = m \text{ if } x_i \text{ equals } y_j, \text{ and } -s \text{ otherwise.}$$

(The term  $s(x_i, y_j)$  is known as the *substitution score*.)

### 3.2. A simple example

Let  $x = AGTA$ ,  $y = ATA$ ,  $m = 1$ ,  $s = -1$ , and  $d = -1$ .

We first construct the score matrix, filling in the base cases in the top row and leftmost column (left). Then, starting from (1,1) and moving towards the bottom right corner, we fill in the values using the score formula described in the previous section. For example, at (1,1), the value in the cell is:

$$F(1,1) = \max \left\{ \begin{array}{l} F(0,0) + s(A,A) = 1, \\ F(0,1) - d = -2, \\ F(1,0) - d = -2 \end{array} \right\} = 1.$$

Repeating this for all cells gives us the final score matrix on the right.

i	0	1	2	3	4
j		A	G	T	A
0	0	-1	-2	-3	-4
1	A	-1			
2	T	-2			
3	A	-3			

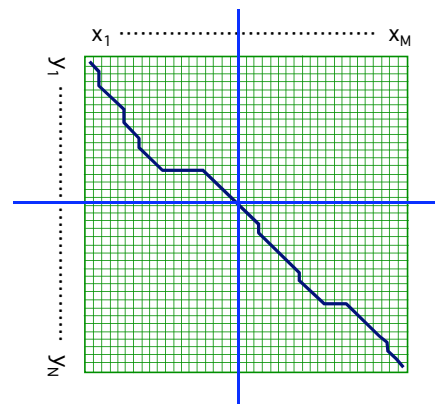
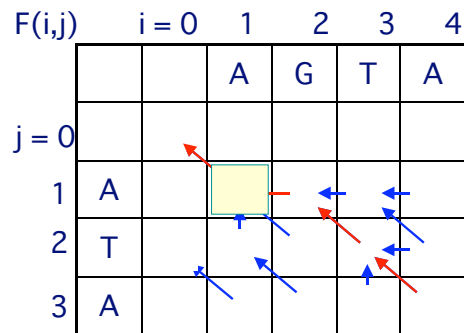
==>

i	0	1	2	3	4	
j		A	G	T	A	
0	0	-1	-2	-3	-4	
1	A	-1	1	0	-1	-2
2	T	-2	0	0	1	0
3	A	-3	-1	-1	0	2

The value of the cell in the bottom right corner is the optimal score for the best alignment between  $x$  and  $y$ . In order to reconstruct the alignment, however, we need another matrix, similar to the score matrix, which stores pointers to the cells from which the optimal scores originated (see right). To find the optimal alignment, start at the bottom right corner and trace back through the pointers to the top left (red arrows).

In the case of ties, some criteria can be applied to choose one, or all of them can be kept and reported. Also, every nondecreasing path from (0,0) to (M,N) – that is, every path that moves to the right and/or down but never up or left – corresponds to an alignment of the two sequences  $x$  and  $y$ .

Notice that an optimal alignment can always be composed of optimal sub-alignments, as shown at the right.



### 3.3. The Needleman-Wunsch algorithm

The matrix method applied in the previous section is known as the Needleman-Wunsch algorithm. More formally, the algorithm consists of the following parts:

1. Initialization: filling in the base cases along the top row and leftmost column

$$F(0,0) = 0$$

$$F(0,j) = -j \times d$$

$$F(i,0) = -i \times d$$

2. Main iteration: filling in the partial alignments

For each  $i = 1 \dots M$ :

For each  $j = 1 \dots N$ :

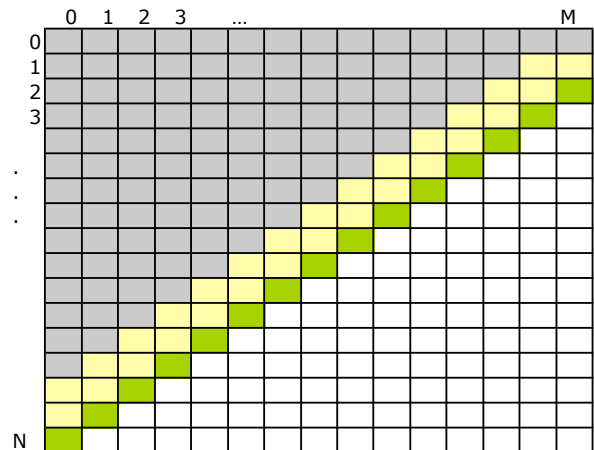
$$F(i-1, j-1) + s(x_i, y_j) \quad [\text{case 1}]$$

$$F(i, j) = \max \{ F(i-1, j) - d \quad [\text{case 2}]$$

$$F(i, j-1) - d \} \quad [\text{case 3}]$$

$$\text{Ptr}(i, j) = \begin{matrix} \text{DIAG,} & \text{if} & [\text{case 1}] \\ \text{LEFT,} & \text{if} & [\text{case 2}] \\ \text{UP,} & \text{if} & [\text{case 3}] \end{matrix}$$

- this is always done left to right, but can be done row by row, column by column, or along the diagonal
- filling in along the diagonal allows parallelization of the algorithm because these cells only rely on 3 other cells for their values, and can be calculated simultaneously (see figure at right).



cells that can be calculated simultaneously  
 cells necessary to calculate scores  
 cells that have already been calculated  
 cells yet to be calculated

3. Termination: The optimal score is  $F(M,N)$  and tracing back through the pointer matrix gives you the best alignment.

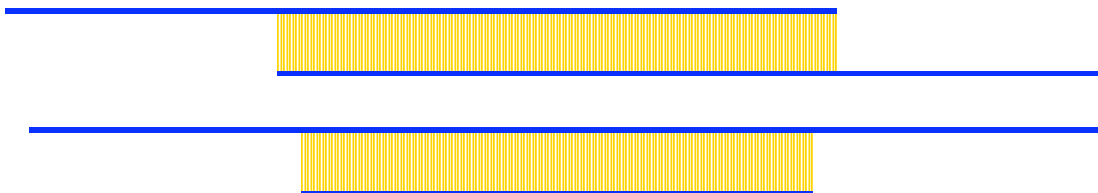
This algorithm has time complexity and space requirements of  $O(MN)$ .

### 3.4. Variants of Needleman-Wunsch

Several variants of the basic sequence alignment algorithm exist to fit particular alignment applications.

#### 3.4.1. Overlap detection

There are times when we don't want to penalize gaps at the ends of sequences, such as when we're aligning overlapping reads from genome sequencing (top picture), or searching for a gene in a genome (bottom picture). This is when we use the *overlap detection* variant of the Needleman-Wunsch algorithm.



The alignment can start anywhere in the first row or column, and end anywhere in the last row or column (see figure below). The two changes to the algorithm occur in the initialization and termination phases. During initialization, all base cases are set to zero rather than decreasing from cell to cell. At termination, the optimal score  $F$  is the maximum of the maximum values in the last row and last column. More formally:

**Initialization**

For all  $i, j$ :

$$F(i,0) = 0$$

$$F(0,j) = 0$$

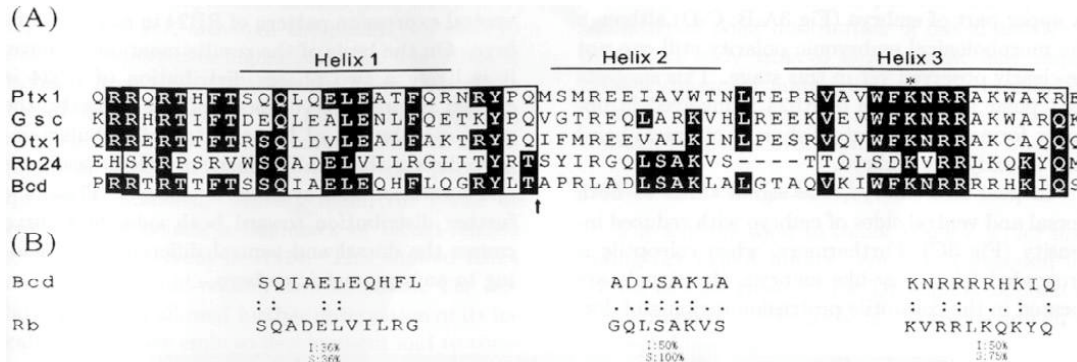
**Termination**

$$F_{opt} = \max \left\{ \begin{array}{l} \max F(i,N) \\ \max F(M,j) \end{array} \right\}$$



### 3.5. Local Alignments

Sometimes we care less about the global similarity between two sequences, but are more interested in smaller, well-conserved regions that they share. For example, there are stretches in protein sequences – called domains – that are associated with certain biological functions; often, we want to find conserved domains between proteins.



Problems of this type are examples of local alignment, and it is defined as follows:

Given 2 strings  $x$  and  $y$ , find substrings  $x'$  and  $y'$  whose optimal global alignment has the maximum score.

The solution to this problem is a subpath in the matrix that can start and end anywhere – that is, we define subproblems starting at all  $(i,j)$  and ending at  $(i',j')$ , apply the Needleman-Wunsch algorithm as before, and take the maximum of all the subproblem solutions.

#### 3.5.1. The Smith-Waterman algorithm

The most famous algorithm used in local alignment is the Smith-Waterman algorithm. This is the algorithm used in the BLAST alignment tool, which is the most cited resource in the biomedical literature. The basic idea is to ignore poorly aligning regions (see figure below).

##### Initialization

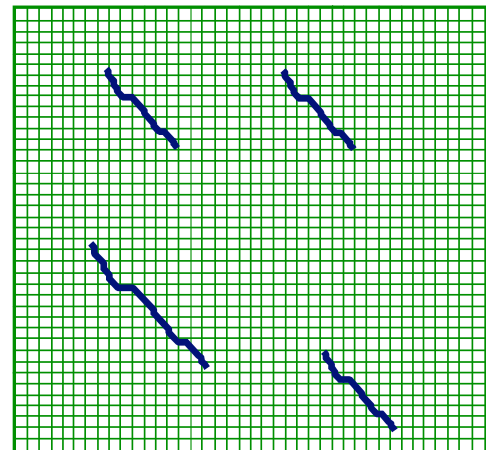
$$F(i,0) = F(0,j) = 0$$

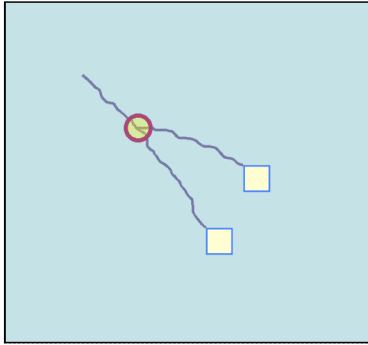
##### Main iteration

$$F(i,j) = \max \left\{ \begin{array}{l} 0 \\ F(i-1,j) - d \\ F(i,j-1) - d \\ F(i-1,j-1) + s(x_i, y_j) \end{array} \right\}$$

##### Termination

If we want the best local alignment,  $F_{opt} = \max F(i,j)$ . We simply find  $F_{opt}$  and trace back.





If we want all local alignments scoring higher than some threshold  $t$ , it becomes more complicated – we can't just find all  $F(i,j) > t$  and trace back because there are many alignments that may overlap (see figure at left). Finding multiple local alignments will not be covered in this course, but the Waterman-Eggert algorithm developed in 1987 solves this problem.

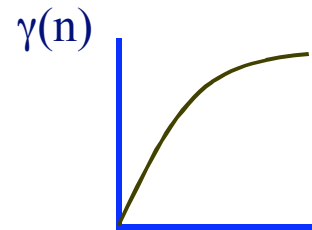
Waterman MS, Eggert M: A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. (1987) J Mol Biol. 197:723-725.

### 3.6. Scoring gaps

Biologically, gaps are more likely to occur as a consecutive group rather than interspersed singly. This is due to the fact that large-scale mutations can occur, and also because transposons (mobile DNA sequences that can replicate and insert copies of themselves into the genome) are rather numerous. For these reasons, we need more intelligent gap scoring functions ( $\gamma(n)$ ) that take these realities into account.

#### 3.6.1. Convex gap scoring functions

One variant for scoring gaps is to use a convex function rather than a linear one – instead of a constant penalty ( $n \times d$ ), penalties decrease as the gap gets longer. This results in small gaps being penalized proportionately more than large gaps.



$$\gamma(n): \text{ for all } n, \gamma(n + 1) - \gamma(n) \leq \gamma(n) - \gamma(n - 1)$$

Changes to the alignment algorithm occur only in the main iteration phase:

$$F(i,j) = \max \left\{ \begin{array}{l} F(i-1,j-1) + s(x_i,y_j) \\ \max_{k=0 \dots i-1} F(k,j) - \gamma(i-k) \\ \max_{k=0 \dots j-1} F(i,k) - \gamma(j-k) \end{array} \right.$$

Initialization and termination remain the same, and the running time using this modification is now  $O(N^2M)$ , assuming that  $N > M$ .

#### 3.6.2. Affine gap scoring functions

Affine scoring compromises between the two previously discussed scoring methods. With affine gaps, we give gap openings a constant penalty, but score gap extensions differently, for example:  $\gamma(n) = d + e(n-1)$ , where  $n$  is the length of the gap.

More on affine gaps will be covered in the next lecture.