CS 284A: Algorithms for Computational Biology
Notes on Lecture:

Sequence alignment

Memisevic Vesna and Milenkovic Tijana
Based on presentation by Xiaohui Xie on February 6th and 11th 2008

A *sequence alignment* is a way of arranging the primary sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Imagine if in the early evolution a sequence G A C T T A had existed, and during evolution it changed to T G A C C T, due to the insertion of the nucleotide T in the first position and the mutation of nucleotide T into C. Additionally, in a different branch (for a different species), the sequence could have evolved differently. For example, a sequence G A T T A could have occurred due to the deletion of nucleotide C. This is illustrated in the figure bellow.

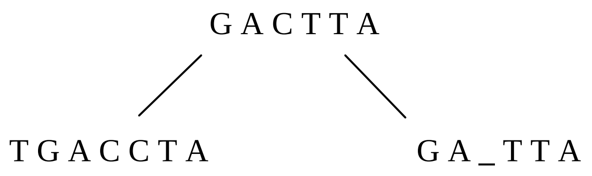

Figure 1: An example of sequence evolution.

The question we want to answer is whether it is possible to find the similarity between two successor sequences and determine if they evolved from the same sequence, given that the initial sequence is not available. Since ancestor sequence is unavailable, the two new sequences need to be aligned in order to find a measure of similarity. Two sequences S and T need to be aligned in such a way that the smallest number of operations happens, where an operation can be any of the following:

1. *Mutation*/*substitution*: an event in which one nucleotide is substituted with another ($a \rightarrow b$)
2. *Insertion*
3. *Deletion*

Since it is hard to distinguish between insertion of a nucleotide into one sequence and its deletion from another sequence, these two operations are together commonly referred to as *indels*.

We formulate the problem as follows: given two sequences S and T, what is the minimum number of operations to transform S to T?

When defining the similarity between a pair of sequences, we first need to assign a weight σ for each of the operations:

1. σ (a, b) – weight of a substitution (a → b),
2. σ (a, -) – weight of an indel,
3. σ (-, a) – weight of an indel.

Then, we need to compute the score of the alignment representing the sum of weights at each position and choose the alignment with the maximum score. For simplicity, we assume the following weights for possible operations:

σ (a, a) = 2, σ (a, b) = -1, σ (a, -) = -1, and σ (-, b) = -1.

In the simple example presented in figure 1, i.e., for the two sequences S and T:

S:	T	G	A	C	C	T	A
T:	_	G	A	_	T	T	A

we need to compute the following weights: σ (T, -), σ (G, G), σ (A, A), σ (C, C), σ (C, T), σ (T, T), and σ (A, A). The score of the alignment shown above is (-1) + (2) + (2) + (-1) + (-1) + (2) + (2) = 5.

Here, we assume that we are given the scoring scheme (i.e., the weights). However, the highest scoring alignment will significantly depend on the choice of these weights. Additionally, we assume the symmetry between sequences S and T, i.e., it is irrelevant whether we are trying to align S with T or T with S. In reality, the "order" of sequences is important.

In general, we try to align sequence *S* of length *n* with sequence *T* of length *m*. We can define $V(i, j)$ to be the maximal score for alignment between the first *i* characters in sequence *S* and the first *j* characters in sequence *T*, i.e., between $S_{1,2,\ldots,i}$ and $T_{1,2,\ldots,j}$. The challenge is to find $V(n, m)$. In our simple example, V (1, 1) = V (T, G) = -1, V (2, 1) = V (T, —) + V (G, G) = -1 + 2 = 1, and so on.

In general, we have:

(a) $$V(0,j) = \sum_{k=1}^{j} \sigma(-,T_k) = -j$$

(b) $$V(i,0) = \sum_{k=1}^{i} \sigma(S_k,-) = -i$$

(c) $$V(i, j) \text{ - three scenarios: } \begin{cases} V(i-1, j-1) \\ V(i-1, j) \\ V(i, j-1) \end{cases}$$

Now we observe all possibilities for (c):

1) position *i* is aligned to position *j*:

   S: 1 ____________ *i*

   T: 1 ____________ *j*

   In this case, $V(i, j) = V(i\text{-}1, j\text{-}1) + \sigma(S_i, T_j)$.

2) position *i* is aligned to position *j-1*:

   S: 1 ____________ i _

   T : 1 ____________ j-1 j

   In this case, $V(i, j) = V(i, j\text{-}1) + \sigma(\_, T_j)$.

3) position *i-1* is aligned to position *j*:

   S: 1 ____________ i-1 i

   T: 1 ____________ j _

   In this case, $V(i, j) = V(i\text{-}1, j) + \sigma(S_i, \_)$.

Optimal choice for (c) will be the maximum of the three possibilities:

$$V(i, j)^* = \max \begin{cases} V(i\text{-}1, j\text{-}1) + \sigma(S_i, T_j) \\ V(i, j\text{-}1) + \sigma(\_, T_j) \\ V(i\text{-}1, j) + \sigma(S_i, -) \end{cases}$$

An example:

S: T G A C C T A

T: G A T T A

S	0	1	2	3	4	5	6	7
T	_	T	G	A	C	C	T	A
0 _	0	-1 ↘	-2	-3	-4	-5	-6	-7
1 G	-1	-1	1 →	0				
2 A	-2							
3 T	-3							
4 T	-4							
5 A	-5							

Note: 0 is an empty string.

The scores are calculated as follows:

$$V(1, 1) = \max \begin{cases} -1 \\ -2 \\ -2 \end{cases} = -1$$

V (2, 1) = 1 (as explained bellow).

1.	T G _ G	2 + (-1) = 1	The best score
2.	T G _ _ _ G	-3	
3.	T G G _	-2	

V (3, 1) = 0 (as explained bellow).

1.	T G A _ _ G	-3	
2.	T G A _ _ _ _ G	-4	
3.	T G A _ G _	-1 + 2 – 1 = 0	The best score

The process continues until all values in the table are computed. Note that to calculate a value in the table it is enough to know values of the neighbor fields in the table. After computing all values and obtaining the optimal score of an alignment, the actual alignment that produced this score can be found by remembering the “direction” from which the score was computed and by tracing that path back.

S T	0 _	1 T	2 G	3 A
0 _	0	-1 ↘	-2	-3
1 G	-1	-1	1 ↘	0
2 A	-2	-2	0	3 ↓
3 T	-3	0	-1	2

By tracing backward, we get:

T G A _
_ G A T

If arrow is vertical, a letter is aligned to the gap, and if it is diagonal, a letter is aligned to another letter.

The time and space complexity of the proposed algorithm is O(mn). Thus, the sequence alignment is solvable in polynomial time by using recursion and solving smaller problems. However, space complexity is problematic when sequences being aligned are long. One solution would be to reduce the space complexity to be linear.

The proposed alignment solution of aligning two whole sequences is referred to as the global *alignment*. However, we sometimes might be interested in aligning only parts of sequences. This approach is known as the *local alignment*.

Local alignment

Computational approaches to sequence alignment generally fall into two categories: global alignments and local alignments. Calculating a global alignment is a form of global optimization that "forces" the alignment to span the entire length of all query sequences. By contrast, local alignments identify regions of similarity within long sequences that are often widely divergent overall.

Let us observe two very long strings S: ($S_1S_2S_3\ldots S_n$) and T: ($T_1T_2T_3\ldots T_m$). The simple example is given bellow:

S: A A A A T G A C T T T T T
T: T A C C

Our goal is to find a substring α from string S and substring β from string T such that the score V(α,β) is maximal over all possible αs and βs. The question we want to answer is how to find best sets of substrings α from S and β from T, which will produce maximal score.

$$S = max_{\alpha,\beta} V(\alpha, \beta)$$

In our simple example, if α = T G A C and β = T A C, then V(α, β) = 2 + (-1) + 2+ 2 = 5.

However, trying all possible αs and βs and scoring all possible alignments requires exponential complexity. Thus, we try to simplify the problem. We first assume α is a suffix of S and β is a suffix of T. The problem now becomes the global alignment of substrings α and β, instead of entire strings S and T. However, unlike the global alignment, any mismatches in positions before α and β are scored 0, i.e.., for all prefixes of S and T that do not include α and β, score is set to zero:

$$V(i, 0) = 0 \quad \text{and} \quad V(0, j) = 0$$

Therefore, the best value for V (i, j) with local alignment is defined as:

$$V(i,j) = \max \begin{cases} V(i\text{-}1, j\text{-}1) + \sigma(i,j) \\ V(i, j\text{-}1) + \sigma(\_, T_j) \\ V(i\text{-}1, j) + \sigma(S_i, -) \\ 0 \end{cases}$$

In order for the algorithm to identify local alignments, the score for aligning unrelated sequence segments should typically be negative. Otherwise true optimal local alignments will be extended beyond their correct ends or have lower scores than longer alignments between unrelated regions.

The problem described above is called local suffix alignment problem, and it represents a subproblem of the local alignment problem. The complete solution includes first selecting a prefix of S, $S_{1,2,\ldots,j}$, and a prefix of T, $T_{1,2,3,\ldots,k}$, and then finding the best suffices α and β in these prefixes, respectively. More specifically:

$$\alpha^*, \beta^* = max_{j,k} (\alpha_j, \beta_k) = max\ [S_{1,2,\ldots,j,}\ T_{1,2,3,\ldots,k}]$$

This solution is known as Smith-Waterman algorithm and it is a well-known algorithm for performing local sequence alignment, that is, for determining similar regions between two sequences. Instead of looking at the total sequence, the Smith-Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure. For local sequence alignment we wish to find what regions (sub-sequences) in the compared pair of sequences will give the best alignment score. However, since it is necessary to repeat the process of finding optimal α and β for every choice of prefixes of S and T, this algorithm is very time-complex. The space complexity remains polynomial as in the case of global alignment.

Affine gap penalty

Thus far, we assumed the independence among positions within a sequence and we simply summed up the scores at each position to get the total score of an alignment. Instead, we can use gap scoring model (*affine gap penalty*) to compute the alignment score.

For example, in the following two sequences

S: ATAAAAT
T:AT_ _ _ _ T

perhaps "AAAA" part of the sequence should not be penalized as 4 gaps. In biological sequences, for example, it is much more likely that one big gap of length 10 occurs in one sequence, due to a single insertion or deletion event, than it is that 10 small gaps of length 1 are made.

Whereas a regular gap extension method would assign a fixed cost per gap, affine gap penalty model penalizes insertions and deletions using a linear function in which one term is length independent, and the other is length dependent. Affine gap penalty is computed as follows:

$$d = W_g + gW_e$$

where W_g is a penalty for gap opening (length independent), and W_e is penalty for gap extension (length dependent). Note that in this model, substitutions are scored in the same way as in a regular gap extension method.

Deriving substitution matrix

Another interesting question is how to choose the substitution matrix. For example, should $\sigma(A, G)$ have the same score as $\sigma(A, T)$? There are many ways to determine this. Here, we present probabilistic models. We observe two sequences S and T:

S: $S_1 \ldots S_n$
T: $T_1 \ldots T_n$

which are aligned to each other and we assume no gaps exist between these two sequences.

There are two models for deriving scoring matrix:

1. Random model
2. Match model

1. <u>Random model:</u>

In this model, we assume that sequences S and T are random, i.e., unrelated to each other.

Probability of generating string S given a random model is:

$$p(S|R) = \prod_{i=1}^{n} p(S_i)$$

where $p(S_i)$ is a probability of a letter at position *i*.

Similarly, a probability of generating string T given a random model is:

$$p(T|R) = \prod_{i=1}^{n} q(T_i)$$

where $q(T_i)$ is a probability of a letter at position *i*.

Thus, p and q are distributions of probabilities (which might be the same). In the simplest case, $(p_A, p_C, p_G, p_T) = (q_A, q_C, q_G, q_T) = (1/4, 1/4, 1/4, 1/4)$.

Therefore, random model assumes that occurrence of each letter is independent:

- Probability is product of probability (frequency) of each letter,
- $P(x,y|R) = p(x)p(y)$.

2. <u>Match model:</u>

Unlike random model, match model assumes that letters are aligned due to evolutionary relationship.

Probability of generating strings S and T given a match model is:

$$p(S,T|M) = \prod_{i=1}^{n} p(S_i,T_i)$$

To test which of the two proposed models is better, we can compute *log-odds ratio*:

$$S = log\frac{p(S,T|M)}{p(S|R)p(T|R)} = \sum_{i=1}^{n} log\frac{p(S_i,T_i)}{p(S_i)q(T_i)}$$

from which we can get substitution matrix.

An example (for amino-acid sequences):

- *Random Model*
    - Assume even mix of amino acids (with 20 different amino acids);
    - Probability of any amino acid = 1/20 = 0.05;
    - Probability of leucine (L) being align with isoleucine (I):
        - P(L,I|R) = p(L)p(I) = 0.05 * 0.05 = 0.0025 (0.25%).
- *Match Model*
    - Leucine and isoleucine are very similar;
    - Empirically, they are paired fairly often in alignments of closely related sequences;
    - For example, probability of L and I being paired is 0.5%:
        - P(L,I|M)= 0.005 = 0.5%
- The substitution matrix entry for the pair leucine/isoleucine will be the log odds ratio. That is, the relative log likelihood ratio of the residue pair (L, I) occurring as an aligned pair, as opposed to an unaligned pair (by chance):
    - ln(0.005/0.0025) = ln(2) = 0.69.
    - Therefore, the matrix entry for this pair will be +1 (rounded up for better computation efficiency).
    - Similar calculations are then performed for all possible pairs of amino acids.