Lecture notes on alignment—I

Usman Roshan
Department of Computer Science
New Jersey Institute of Technology

February 22, 2005

1 Basics

We assume we are given an alphabet of characters, which in the case of DNA is $\Sigma = \{A, C, G, T\}$. We also have a special character called a *gap* which is denoted by '-'. Given two strings strings S and T an alignment is a mapping $S \rightarrow S', T \rightarrow T'$ such that

- $|S'| = |T'|$, and
- removal of gaps from S' gives S and from T' gives T.

An example alignment of $S = AGA$ and $T = ACGG$ is

$$\begin{array}{cc} S & A-GA \\ T & ACGG \end{array}$$

An alignment is evaluated using a scoring function σ. We use the simple scoring function

$$\sigma(x,y) = \begin{cases} 1 & \text{if } x = y, x \neq -, y \neq - \\ 0 & \text{if } x \neq y, x \neq -, y \neq - \\ -1 & \text{if } x = -, y = - \end{cases}$$

The score of our example alignment then, according to σ, is +1-1+1+0=1. Given two sequences we are interested in finding the optimal alignment, i.e., the alignment which optimizes a given scoring function σ.

2 Pairwise alignment

2.1 Dynamic programming algorithm for optimal alignment

We are given sequences S of length n and T of length m and want to compute the optimal alignment under a scoring function σ. If we let $V(i,j)$ be the score of the optimal alignment between $S_{1\ldots i}$ and $T_{1\ldots j}$, then we can write the recursion as:

Base conditions : $V(i,0) = \sum_{k=0}^{i} \sigma(S_k, -)$
$V(0,j) = \sum_{k=j}^{i} \sigma(-, T_k)$

Recurrence : for $1 \leq i \leq n, 1 \leq j \leq m$

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

This algorithm takes $O(nm)$ space and time. $V(n,m)$ will give us the score of the optimal alignment between S and T.

2.2 Local alignment

If we are interested in the best matching between substrings of S and T that can be done by modifying the previous algorithm. We modify the recursive relations as follows:

Base conditions : $\forall i, j, V(i,0) = 0, V(0,j) = 0$

Recurrence : for $1 \leq i \leq n, 1 \leq j \leq m$

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

Compute optimal substrings : $V(i^*, j^*) = \max_{1 \leq i \leq n, 1 \leq j \leq m} V(i,j)$

2.3 Scoring matrices—PAM and BLOSUM

The quality of the alignment depends largely on the scoring matrix use. In our previous example we used the unit matrix where we assign gap penalties

of $0, 1$ or -1. In practice scoring matrices are derived from real data and two popular ones are Point Accepted Mutations (PAM) and Blocks Substitution Matrix (BLOSUM). We discuss how the PAM matrices are derived.

2.3.1 PAM matrices

Margaret Dayhoff and her co-workers developed the PAM matrices in 1978. They examined 1572 mutations between 71 families of closely related sequences of proteins. The first step in computing the matrix is to compute a *probability transition* matrix M. This defines the probability of amino acid x mutating into amino acid y.

To compute M we need

- a list of *accepted mutations*, and
- the probabilities of occurrence p_a of each amino acid a (also known as *background* probabilities)

p_a can be estimated by counting the occurrence of amino acid a in all the sequences and then dividing by the total number of amino acids encountered. Accepted mutations are those which were positively selected by the environment and did not cause the demise of the organism. They can be collected by aligning sequences from very closely related sequences, which is what Dayhoff and her group did. Note that accepted mutations are undirected events, i.e., we do not know if a mutated into b or vice-versa. All we know is that a mutation occurred.

Given the accepted mutations we compute f_{ab}, the number of times the mutation $a \rightarrow b$ occurred. Note that $f_{ab} = f_{ba}$ since we are dealing with undirected mutations. We define $f_a = \sum_{b \neq a} f_{ab}$ and thus the total number of mutations in which a was involved is $f = \sum_a f_a$.

We are now ready to compute M_{ab}, the probability of amino acid a changing to amino acid b. We first compute M_{aa} by defining the relative mutability

$$m_a = \frac{f_a}{100 f p_a}$$

which is the probability of a changing. Note the additional term $100p_a$. Thus, m_a is the probability that a will change per 100 amino acids on the average. Given m_a, we can compute $M_{aa} = 1 - m_a$ and M_{ab} as

$$M_{ab} = Pr(a \rightarrow b | a \text{ changed }) Pr(a \text{ changed }) = \frac{f_{ab}}{f_a} m_a.$$

In this model the unit of evolution is amount of evolution that will change 1 in 100 amino acids on the average. We call this unit a 1 PAM evolutionary distance. To compute the probability that a will change into b in 2 PAM units, we have to sum the probability of a changing into any amino acid c and then c changing into b, which is just M^2_{ab}. In general M^k is the transition probability matrix for k PAM units of evolution. Note that PAM k does not mean there are k observed differences. The observed differences will be smaller since there are intermediate changes which we cannot see.

We now define the PAM scoring matrices. These matrices contain the ratio between the probability of a changing to b as opposed to it being a random occurrence. M_{ab} is the probability of a changing to b whereas there is a p_b chance of b occurring randomly. Therefore the ratio is $\frac{M_{ab}}{p_b}$. In practice we take the log of this value and multiply by 10 so that we sum the scores instead of taking products. The 10 factor is to reduce rounding errors. The scoring matrix for k PAM distance is thus defined as $10 \log_{10} \frac{M^k_{ab}}{p_b}$.

2.3.2 BLOSUM matrices

BLOSUM matrices are computed from alignments of shorter sequences—blocks of sequences with sufficient similarity. PAM, on the other hand, comes from global alignments.

2.4 FASTA

The FASTA algorithm is a heuristic for string comparison. In practice we may need to search through a database of sequences to find ones similar to a given query. Doing the optimal pairwise alignment for each comparison can be very costly, especially when dealing with databases of tens of thousands of sequences.

The FASTA heuristic looks for local matching subsequences and works well in practice. We first define hot spots and diagonal runs.

- *Hot spots*: matching substrings of length k from the query and the database sequences. Note that these do not contain any indels.
- *Diagonal run*: a sequence of nearby hot spots on the same diagonal of the dynamic programming matrix. These are not necessarily adjacent on the diagonal, spaces are allowed.

We now outline the heuristic.

1. Look for *hot spots*.

   This can be done efficiently by preprocessing the database sequences and for each sequence store each *ktup* (k-length substring) in hash-table for efficient lookup. We can then use a sliding window of length k along the query to find all the hot spots for each database sequence.

2. Find top 10 best diagonal runs.

   We evaluate a diagonal run by giving each hot spot a positive score and the space between hot-spots a negative score. The score of the diagonal run is the sum of the hot spot scores and of the *inter spot* scores.

3. Re-evaluate the diagonal runs.

   A diagonal run specifies a sub-alignment composed of only matches and mismatches. There are no indels because everything is along one diagonal. We evaluate each sub-alignment specified by the diagonal run using a scoring matrix, such as PAM 250.

4. Combine good close diagonal runs.

   We combine diagonal runs to form an alignment in the following way. We construct a directed weighted graph $G = (V, E)$. We set V to be the sub-alignments computed in the previous step and the score of each vertex to be the score of the subalignment (also computed in the previous step). We add an edge from subalignment u to subalignment v if v starts at a higher row and column than those at which u ends. We give the edge a negative weight depending upon the number of gaps created in the alignment by combining subalignments u and v. We now search for a maximum weight path in this graph which can be done in polynomial time. We take the best alignments found and discard those below a specified score.

5. Band alignment.

As a last step we compute another alignment. We take the best diagonal run from step 2 and consider a narrow diagonal band around it. We compute an optimal alignment constrained to this diagonal and call this the band alignment.

6. Full alignment.

   As a last step the alignments computed in steps 4 and 5 are ranked according to their scores and a full dynamic programming alignment is computed for the highest ranked database sequences.

2.5 BLAST

BLAST was developed as an improvement over FASTA to find better and fewer hot spots. The idea is to use the scoring matrix in the first stage of finding the hot spots. We first define some terms before outlining the BLAST strategy.

Given two strings S_1 and S_2, a *segment pair* is a pair of equal length substrings of S_1 and S_2 aligned without gaps. A *locally maximal segment pair* is one whose alignment score cannot be improved by extending or shortening it. A *maximum segment pair* in S_1 and S_2 is the pair with the maximum score over all segment pairs in S_1 and S_2.

We now outline BLAST.

1. For each database sequences s, find all $w$length substrings (called words) of s that align with words from the query. Matches with a score higher than t are called *hits* and are kept. Note that matches here are evaluated using a scoring matrix. Each hit is really a segment pair. This step can be done efficiently using the preprocessing that was described earlier in FASTA.

2. Extend each hit to a locally maximal segment pair and keep those whose score is above a certain threshold. This is done by extending each hit until the drop in score, relative to the maximum encountered so far, exceeds a drop-off threshold.

Improved BLAST Note that this original BLAST did not allow indels. The improved BLAST in 1997 by Altschul allows for indels and is similar to FASTA but with less constraints.

PSI BLAST—position specific iterated BLAST We first define a *profile*. A profile P is used to represent a collection of aligned sequences A. Each site of the alignment A_i is represented by a vector of amino acid frequencies P_i. We denote the frequency of amino acid x in P_i as P_i^x. Then the score between two profile sites P_i and Q_j is

$$\sum_{x,y} P_i^x Q_j^y S(x,y)$$

where x and y range over the 20 amino acids and $S(x,y)$ is the amino acid scoring matrix. Having defined this we can compute the alignment score between two profiles using the optimal pairwise alignment or BLAST or FASTA.

As the name suggests, PSI-BLAST this is an iterated version of BLAST which attempts to find better alignments. PSI BLAST searches the database for closest matches to a profile. PSI BLAST computes a profile for a given query sequence by setting each P_i to be the i^{th} row of the scoring matrix. It then searches the database, as described previously, for closely matching sequences. Note that a profile can be constructed for a single sequence as well; therefore are searches are well-defined. PSI-BLAST then takes the best matching sequences, builds a profile for them, and continues the search until no more close sequences are found.