Lecture 3: December 27, 1998

Lecturer: Prof. Ron Shamir *Scribe: Micky Frankel and Yossi Richter*

3.1 Introduction - Sequence Alignment Heuristics

In the second lecture we presented dynamic programming algorithms to calculate the best alignment of two strings. When we are searching, in a database of size 10^7, for the closest match to a query string of length 200-500, we can not use these algorithms because they require too much time. There are several approaches to overcome this problem:

1. Implementing the dynamic programming algorithms in hardware, thus executing them much faster.

2. Using parallel hardware, the problem can be divided efficiently to a number of processors, and the results can be integrated later.

3. Using different heuristics that work much faster than the original dynamic programming algorithm.

We next present some of the most commonly used heuristics.

3.2 FASTA

The FASTA algorithm is a heuristic method for string comparison. It was developed by *Lipman* and *Pearson* in 1985 [6] and further improved in 1988 [7].

FASTA compares a query string against a single text string. When searching the whole database for matches to a given query, we compare the query using the FASTA algorithm to every string in the database.

When looking for an alignment, we might expect to find a few segments in which there will be absolute identity between the two compared strings. The algorithm is using this property and focuses on these identical regions.

The stages in the FASTA algorithm are as follows:

1. We specify an integer parameter called *ktup* (short for *k respective tuples*), and we look for *ktup*-length matching substrings of the two strings. The standard recommended *ktup* values are six for DNA sequence matching and two for protein sequence matching. The matching *ktup*-length substrings are referred to as *hot spots*. Consecutive hot spots are located along the dynamic programming matrix diagonals. This stage can be done efficiently by using a lookup table or a hash to store all the *ktup*-length substrings from one string, and then search the table with the *ktup*-length substrings from the other string.

2. In this stage we wish to find the 10 best *diagonal runs* of hot spots in the matrix. A diagonal run is a sequence of nearby hot spots on the same diagonal (not necessarily adjacent along the diagonal, i.e., spaces between these hot spots are allowed). A run need not contain all the hot spots on its diagonal, and a diagonal may contain more than one of the 10 best runs we find.

   In order to evaluate the diagonal runs, FASTA gives each hot spot a positive score, and the space between consecutive hot spots in a run is given a negative score that decreases with the increasing distance. The score of the diagonal run is the sum of the hot spots scores and the interspot scores. FASTA finds the 10 highest scoring diagonal runs under this evaluating scheme.

3. A diagonal run specifies a pair of aligned substrings. The alignment is composed of matches (the hot spots) and mismatches (from the interspot regions), but it does not contain any indels because it is derived from a single diagonal. We next evaluate the runs using an amino acid (or nucleotide) substitution matrix, and pick the best scoring run. The single best subalignment found in this stage is called $init_1$. Apart from computing $init_1$, a filtration is performed and we discard of the diagonal runs achieving relatively low scores.

4. Until now we essentially did not allow any indels in the subalignments. We now try to combine "good" diagonal runs from close diagonals, thus achieving a subalignment with indels allowed. We take "good" subalignments from the previous stage (subalignments whose score is above some specified cutoff) and attempt to combine them into a single larger high-scoring alignment that allows some spaces. This can be done in the following way:

   We construct a directed weighted graph whose vertices are the subalignments found in the previous stage, and the weight in each vertex is the score found in the previous stage of the subalignment it represents. Next, we extend an edge from vertex u to vertex v if the subalignment represented by v starts at a lower row than where the subalignment represented by v ends. We give the edge a negative weight which depends on the number of gaps that would be created by aligning according to subalignment v followed by subalignment u. Essentially, FASTA then finds a maximum weight path in this graph. The selected alignment specifies a single local alignment between the two strings. The best alignment found in this stage is marked $init_n$. As in the previous stage, we discard alignments with relatively low score.

5. In this step FASTA computes an alternative local alignment score, in addition to $init_n$. Recall that $init_1$ defines a diagonal segment in the dynamic programming matrix. We consider a narrow diagonal band in the matrix, centered along this segment. We observe that it is highly likely that the best alignment path between the $init_1$ substrings, lies within the subtable defined by the band. We assume this is the case and compute the optimal local alignment in this band, using the ordinary dynamic programming algorithm. Assuming that the best local alignment is indeed within the defined band, the local alignment algorithm essentially merges diagonal runs found in the previous stages to achieve a local alignment which may contain indels. The band width is dependent on the *ktup* choice. The best local alignment computed in this stage is called *opt*.

6. In the last stage, the database sequences are ranked according to $init_n$ scores or *opt* scores, and the full dynamic programming algorithm is used to align the query sequence against each of the highest ranking result sequences.

Although FASTA is a heuristic, and as such it is possible to show instances in which the alignments found by the algorithm are not optimal, it is claimed (and supported by experience) that the resulting alignment scores well compare to the optimal alignment, while the FASTA algorithm is much faster than the ordinary dynamic programming alignment algorithm.

3.3 BLAST - Basic Local Alignment Search Tool

The *BLAST* algorithm was developed by *Altschul*, *Gish*, *Miller*, *Myers* and *Lipman* in 1990 [1] The motivation to the development of BLAST was the need to increase the speed of FASTA by finding fewer and better hot spots during the algorithm. The idea was to integrate the substitution matrix in the first stage of finding the hot spots.

BLAST concentrates on finding regions of high local similarity in alignments without gaps, evaluated by an alphabet-weight scoring matrix. We next define some fundamental objects concerning BLAST:

Given two strings S_1 and S_2, a *segment pair* is a pair of equal length respective substrings of S_1 and S_2, aligned without spaces.

A *locally maximal segment pair* is a segment pair whose alignment score (without spaces) can not be improved by extending it or shortening it.

A *maximal segment pair* (MSP) in S_1 and S_2 is a segment pair with the maximum score over all segment pairs in S_1, S_2.

When comparing all the sequences in the database against the query, BLAST attempts to find all the database sequences that when paired with the query contain an MSP above some cutoff score S. We choose S such that it is unlikely to find a random sequence in the database that achieves a score higher than S when compared with the query sequence.

The stages in the BLAST algorithm are as follows:

1. Given a length parameter w and a threshold parameter t, BLAST finds all the w-length substrings (called *words*) of database sequences that align with words from the query with an alignment score higher than t. Each such hot spot is called a *hit* in BLAST.

2. We extend each hit to find out if it is contained in a segment pair with score above S.

We may implement the first stage by constructing, for each w-length word α in the query sequence, all the w-length words whose similarity to α is at least t. We store these words in a data structure which is later accessed while checking the database sequences.

It is usually recommended to set the parameter w to values of 3 to 5 for amino acids, and ~ 12 for nucleotides.

Although BLAST does not allow alignments with indels, it has been shown that with the correct selection of values to the parameters used by the algorithm, it is possible to obtain all the correct alignments while saving much of the computation time compared to the standard dynamic programming method.

3.3.1 Improved BLAST

Altschul et al. suggested in 1997 [2] an improved BLAST algorithm that allows indels in the alignment.

The algorithm stages follows:

1. When considering the dynamic programming matrix to align two strings, we search along each diagonal for two w-length words such that the distance between them is $\leq A$ and their score is $\geq T$. Whenever we encounter such a pair of hits we concatenate them.

2. In the second stage we want to allow local alignments with indels, similarly to the FASTA algorithm. We allow two local alignments from different diagonals to merge

into a new local alignment composed of the first local alignment followed by some indels and then the second local alignment. This local alignment is essentially a path in the dynamic programming matrix, composed of two diagonal sections and a path connecting them which may contain gaps. Unlike in FASTA, where we only allowed the diagonal to have local shifts, restricted to a band, here we allow local alignments from different diagonals to merge as long as the resulting alignment has a score above some threshold. This method results in an alignment structure which is much less regular.

3.3.2 PSI BLAST - Position Specific Iterated BLAST

The PSI BLAST is another improved version of the BLAST algorithm [2].

When aligning a group of amino acid sequences, i.e., writing them one below the other (see section 3.5 for discussion of multiple alignment), the vector of characters in a certain column (i.e. the same position in the aligned sequences) is called a *profile*. For a certain profile we may compute the histogram of characters types and obtain the variance between them. When we align together amino acid sequences belonging to the same protein family, we will find that some regions are very similar, with profiles showing little variance. These regions, called *conserved regions*, define the structure and functionality typical to this family. We would like the substitution matrices we use to take into account the statistic information we have about how conserved is the column, in order to improve our alignment score.

In the first stage in the algorithm we perform ordinary BLAST while using a different cost vector V_i for each column i. Initially, each such vector V_i is set to the row of the substitution matrix corresponding to the i-th character in the query sequence. From the high-scoring results we get, we build the profiles for each column. We continue to perform BLAST iteratively while using as query the collection of profiles, i.e. we use a histogram at each column rather than a simple string, and compare it against the database. This is equivalent to updating the position dependent cost vectors according to the profile statistics. After each iterative step we update the profiles according to the obtained result sequences. We terminate the iterative loop when we no longer find new meaningful matches.

It should be noted that biologists regard PSI BLAST as not reliable.

3.4 Amino Acids Substitution Matrices

When we search for the best alignment between two protein sequences, the scoring (or substitution) matrix we use can largely affect the results we get. Ideally, the scores should reflect the biological phenomena that the alignment seeks to expose. For example - in the case of sequence divergence due to evolutionary mutations, the values in the scoring matrix should ideally be derived from empirical observation on ancestral sequences and their present-day descendants.

Two examples of simple substitution matrices often used that do not employ the biological phenomena are:

1. The *unit matrix*:

$$M_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

2. The *genetic code matrix*. M_{ij} equals the number of minimal base substitution needed to convert a codon of amino acid i to a codon of amino acid j. We disregard here the importance of chemical properties of the amino acids, that evidently influence the chances for their substitution, like their hydrophobicity, charge or size.

3.4.1 PAM units and PAM matrices

The methodology of PAM units and the first specific PAM matrices were developed by *Margaret Dayhoff* and al. [8, 4]

PAM units

We use PAM units to measure the amount of evolutionary distance between two amino acid sequences. Two strings S_1 and S_2 are said to be one PAM unit diverged if a series of accepted point mutations (and no insertions of deletions) has converted S_1 to S_2 with an average of one accepted point-mutation event per 100 amino acids. The term "accepted" here means a mutation that was incorporated into the protein and passed to its progeny. Therefore, either the mutation did not change the function of the protein or the change in the protein was beneficial to the organism.

Note that two strings which are one PAM unit diverged do not necessarily differ in one percent, as often mistakenly thought, because a single position may undergo more than a single mutation. The difference between the two notions grows as the number of units does.

There are two main problems with the notion of the PAM units:

1. First, practically all the sequences we can obtain today are extracted from extant organisms. We almost do not know any protein sequences where one is actually derived from the other. The lack of ancestral protein sequences is handled by assuming that amino acid mutations are reversible and equally likely in either direction. This assumption, together with the additivity property of the PAM units derived from its definition, imply that given two amino acid sequences: S_i and S_j whose mutual ancestor is S_{ij} we have:
$$d(S_i, S_j) = d(S_i, S_{ij}) + d(S_{ij}, S_j)$$
when $d(i, j)$ is the PAM distance between amino acid sequences i and j.

2. The second problem, which is more difficult to overcome, is that we disregard here insertions and deletions which may occur during evolution, hence we can not be sure of the correct correspondence between sequence positions. In order to know the exact correspondence one has to be able to identify the true historical gaps, or, at least to identify large intervals along the two sequences where the correspondence is correct. This can not always be done with certainty, especially when the two sequences are distantly diverged.

PAM matrices

PAM matrices are amino acid substitution matrices that encode the expected evolutionary change at the amino acid level. Each PAM matrix is designed to compare two sequences which are a specific number of PAM units apart. For example - the PAM120 score matrix is designed to compare between sequences that are 120 PAM units apart: The score it gives a pair of sequences is the (log of the) probabilities of such sequences evolving during 120 PAM units of evolution. For any specific pair (A_i, A_j) of amino acids the (i, j) entry in the PAM n matrix reflects the frequency at which A_i is expected to replace with A_j in two sequences that are n PAM units diverged. These frequencies should be estimated by gathering statistics on replaced amino acids.

Collecting statistics about amino acids substitution in order to compute the PAM matrices is relatively difficult for sequences that are distantly diverged, as mentioned in the previous section. But for sequences that are highly similar, i.e., the PAM divergence distance between them is small, finding the position correspondence is relatively easy since only few insertions and deletions took place. Therefore, in the first stage statistics were collected from aligned sequences that were believed to be approximately one PAM unit diverged and the PAM1 matrix could be computed based on this data, as follows: Let M_{ij} denote the observed frequency (= estimated probability) of amino acid A_i mutating into amino acid A_j during one PAM unit of evolutionary change. M is a 20×20 real matrix, with the values in each matrix column adding up to 1. There is a significant variance between the values in each column. For example, see figure 3.1, taken from [4].

	A	R	N	D	C
A	9867	2	9	10	3
R	1	9913	1	0	1
N	4	1	9822	36	0
D	6	0	42	9859	0
C	1	1	0	0	9973

Figure 3.1: The top left corner 5×5 of the PAM1 matrix. We write $10^4 M_{ij}$ for convinience.

Once M is known, the matrix M^n gives the probabilities of any amino acid mutating to any other during n PAM units. The (i, j) entry in the PAM n matrix is therefore:

$$\log \frac{f(j)M^n(i,j)}{f(i)f(j)} = \log \frac{M^n(i,j)}{f(i)}$$

where $f(i)$ and $f(j)$ are the observed frequencies of amino acids A_i and A_j respectively. This approach assumes that the frequencies of the amino acids remain constant over time, and that the mutational processes causing substitutions during an interval of one PAM unit operate in the same manner for longer periods. We take the log value of the probability in order to allow computing the total score of all substitutions using summation rather than multiplication. The PAM matrix is usually organized by dividing the amino acids to groups of relatively similar amino acids and all group members are located in consecutive columns in the matrix.

3.4.2 BLOSUM - BLOcks SUbstitution Matrix

The BLOSUM matrix is another amino acid substitution matrix, first calculated by *Henikoff* and *Henikoff* [5]. For its calculation only blocks of amino acid sequences with small change between them are considered. These blocks are called *conserved blocks* (See figure 3.2). One reason for this is that one needs to find a multiple alignment between all these sequences and it is easier to construct such an alignment with more similar sequences. Another reason is that the purpose of the matrix is to measure the probability of one amino acid to change into another, and the change between distant sequences may include also insertions and deletions of amino acids. Moreover, we are more interested in conservation of regions inside protein families, where sequences are quite similar, and therefore we restrict our examination to such.

```
AABCDA...BBCDA
DABCDA.A.BBCBB
BBBCDABA.BCCAA
AAACDAC.DCBCDB
CCBADAB.DBBDCC
AAACAA...BBCCC
```

Figure 3.2: Alignment of several sequences. The conserved blocks are marked.

The first stage of building the BLOSUM matrix is eliminating sequences, which are identical in more than x% of their amino acid sequence. This is done to avoid bias of the result in favor of a certain protein. The elimination is done either by removing sequences from the block, or by finding a cluster of similar sequences and replacing it by a new sequence that represents the cluster. The matrix built from blocks with no more the x% of similarity is called BLOSUM-x (e.g. the matrix built using sequences with no more then 50% similarity is called BLOSUM-50.)

The second stage is counting the pairs of amino acids in each column of the multiple alignment. For example in a column with the acids AABACA (as in the first column in the block in figure 3.2), there are 6 AA pairs, 4 AB pairs, 4 AC, and one BC. The probability $q_{i,j}$ for a pair of amino acids in the same column to be A_i and A_j is calculated, as well as the probability p_i of a certain amino acid to be A_i.

In the third stage the *log odd ratio* is calculated as $s_{i,j} = \log_2 \frac{q_{i,j}}{p_i p_j}$. As final result we consider the rounded $2s_{i,j}$, this value is stored in the (i,j) entry of the BLOSUM-x matrix.

In contrast to the PAM matrices, more sequences are examined in the process of computing the BLOSUM matrix. Moreover, the sequences are of specific nature of resemblance, and therefore the two sets of matrices differ.

Comparing the efficiency of two matrices is done by calculating the ratio between the number of pairs of similar sequences discovered by a certain matrix but not discovered by another one and the number of pairs missed by the first but found by the other. According to this comparison BLOSUM-62 is found to be better than other BLOSUM-x matrices as well as than PAM-x matrices.

3.5 Multiple Alignment

Definition A *multiple alignment* of strings $S_1, S_2, \ldots, S_k$ is a series of strings with blanks $S'_1, S'_2, \ldots, S'_k$ such that

1. $|S'_1| = |S'_2| = \cdots = |S'_k|$
2. S'_j is extension of S_j, obtained by insertion of blanks.

```
AC..BCDB
.CADB.D.
ACA.BCD.
```

Figure 3.3: A multiple alignment of $ACBCBD$, $CADDB$ and $ACABCD$.

We are interested in finding a common alignment of several sequences, because this multiple similarity suggests a common structure of the protein product, a common function or a common evolutionary source. A multiple alignment carries more information than a pairwise one, as a protein can be matched against a family of proteins instead of only against another one.

The best multiple alignment of r sequences is calculated using an r-dimensional hypercube D, defining $D(j_1, j_2, \ldots, j_r)$ to be the best score for aligning the prefixes of lengths

$j_1, j_2, \ldots, j_r$ of the sequences $x_1, x_2, \ldots, x_r$, respectively.
We define

$$D(0, 0, \ldots, 0) = 0$$

And we calculate

$$D(j_1, j_2, \ldots, j_r) = min_{\epsilon \in \{0,1\}^n, \epsilon \neq 0}\{D(j_1 - \epsilon_1, j_2 - \epsilon_2, \ldots, j_r - \epsilon_r) + \rho(\epsilon_1 x_{j_1}, \ldots, \epsilon_r x_{j_r})\}$$

where ρ is the cost function. The size of the hyper-cube is $O(\prod_{j=1}^{r} n_j)$, where n_j is the length of x_j, where computation of each of each entry consider $2^r - 1$ others.
If $n_1 = n_2 = \cdots = n_r = n$, the space complexity is of $O(n^r)$ and the time complexity is of $O(2^r n^r)$.

There are several known useful possibilities for measuring the divergence of a set of aligned strings, namely the total distance between them.

- *Distance from Consensus* - The consensus of an alignment is a string of the most common character in each column of the alignment. The total distance between the strings is defined as the number of characters that differ from the consensus character of their column.

- *Evolutionary Distance* - The weight of the lightest evolutionary tree that can be constructed from the sequences, with the weight of the tree defined as the number of changes between pairs of sequences that correspond to two adjacent nodes in the tree, summed over all such pairs.

- *Sum of Pairs* - The sum of pairwise distances between all the pairs of sequences.

Carrillo and *Lipman* [3] found a heuristic method for accelerating the search for the best multiple alignment. The method is based on the property that if the strings are relatively similar, the alignment path would be close to the main diagonal, therefore not all the values in the multi-dimensional cube need to be calculated, we now detail this algorithm.

Assuming an upper bound on cost of the best alignment, we will discard some alignments that are a priori known to be more expensive than the bound on the cost.

Let A be an alignment of strings $X_1, x_2, \ldots, x_r$. Denote by $A_{i,j}$ the pair of rows in A containing only x_i and x_j, and by $c(A_{i,j})$ the cost of this pairwise alignment. Denote by $c(A)$ the total cost of A, and suppose we define $c(A) = \sum_{i<j} c(A_{i,j})$. Let A^* be the optimal alignment (the one with the minimal cost), and suppose we know that $c(A^*) \leq c'$. Therefore,

$$c' \geq c(A^*) = \sum_{i<j} c(A^*_{i,j}) = c(A^*_{u,v}) + \sum_{i<j\ (i,j)\neq(u,v)} c(A^*_{i,j}) \geq c(A^*_{u,v}) + \sum_{i<j\ (i,j)\neq(u,v)} D(x_i, x_j)$$

Where $D(x,y)$ is the optimal score for aligning strings x and y. It follows that

$$c(A^*_{u,v}) \le c' - \sum_{i<j\ (i,j)\neq(u,v)} D(x_i, x_j)$$

$A^*_{u,v}$ is a projection of A^* on the uv-plain. By calculating $D(x_i, x_j)$ for each i and j, we can find $B(u,v) = c' - \sum_{i<j\ (i,j)\neq(u,v)} D(x_i, x_j)$.

Now, consider a cell $(i_1, i_2, \ldots, i_u = s, \ldots, i_v = t, \ldots, i_r)$ whose projection to the uv-plane is (s,t). If the best alignment A^* passes through this cell, then its projection $A^*_{u,v}$ passes through (s,t), and its cost $c(A^*_{u,v})$ agrees with $best^{(u,v)}_{s,t} \le c(A^*_{u,v}) \le B(u,v)$ where $best^{(u,v)}_{s,t}$ is an upper bound on the optimal score for an alignment through (s,t) in the uv-plain. We can compute such an upper bound as:

$$best^{(u,v)}_{i,j} = D(x_{u,1}x_{u,2}\ldots x_{u,i-1}, x_{v,1}x_{v,2}\ldots x_{v,j-1}) + d(x_{u,i}, x_{v,j}) + D(x_{u,i+1}\ldots x_{u,n_u}, x_{v,j+1}\ldots x_{v,n_v})$$

where $d(\kappa_1, \kappa_2)$ is the cost of matching the characters κ_1 and κ_2.

Therefore if $best^{(u,v)}_{s,t} > B(u,v)$, then the best alignment A^* cannot pass through the cell $(i_1, i_2, \ldots, i_u = s, \ldots, i_v = t, \ldots, i_r)$ for any $i_1, i_2, \ldots, i_{u-1}, i_{u+1}, \ldots, i_{v-1}, i_{v+1}, \ldots, i_r$, and these cells can be discarded from the computation.

Bibliography

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215:403–10, 1990.

[2] et al. Altschul SF. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res.*, 25(17):3389–402, 1997.

[3] H. Carrillo and D. Lipmann. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math*, 48:1073–1082, 1988.

[4] M. Dayhoff and R. Schwartz. Matrices for detecting distant relationship. *Atlas of Protein Sequences*, pages 353–358, 1979.

[5] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Science USA*, 89(22):10915–10919, November 1992.

[6] D. Lipman and W. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.

[7] D. Lipman and W. Pearson. Improved tools for biological sequence comparison. *Proc. Natl. Academy Science*, 85:2444–2448, 1988.

[8] R. Schwartz M. Dayhoff and B. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5:345–352, 1978.