CPS260/BGT204.1 Algorithms in Computational Biology **October 2, 2003**

Lecture 11: Searching in Sequence Databases

Lecturer: Pankaj K. Agarwal *Scribe: Dmitriy Morozov*

This lecture continues our discussion of sequence alignment by examining an extension of our original model: instead of finding alignments between two sequences U and V, we examine the case that is more common in practice — searching sequence databases, looking for alignments between a given query sequence and a collection of sequences that constitute a database, and returning the ones that have high-score local alignments.

Let us, first, establish the notation that we will use:

$\mathcal{V} = \{V_1, V_2, \ldots, V_N\}$	—	database, a collection of sequences (in real world databases, examples of which include GenBank, UniGene, dbSTS, UniSTS, and HTG [2], each seqeunce is complemented with additional infromation such as the origin of data, bibliographic references, etc. — in our case we are concerned only with sequences themselves), here N is large (typically 100,000s)
U	—	query sequence
$\sigma(U, V_i)$	—	score function, a measure between sequence similarity that can be computed by dynamic programming; the exact values depend on the scoring matrices that are used (e.g., BLOSUM, PAM, etc.)

We are faced with the following problem: given a query sequence U and a database $\mathcal{V}$, find one/few/all sequences in the database similar to U. The naive approach of ranking the alignments of U with all sequences in the database $\mathcal{V}$ using dynamic programming (i.e., computing $\sigma(U, V_1), \sigma(U, V_2), \ldots, \sigma(U, V_N)$) and returning top k of them is unacceptable: with the sizes of the sequence databases in the 100,000s any dynamic programming method (running in $O(n^2)$ time) will take too long to compute. Therefore, we are forced to find a compromise by finding a trade off between efficiency and accuracy.

Our goal is to develop heuristics that are: fast, simple to implement, and return "good" solutions most of the time. A number of such heuristics have been developed to address this problem; in this lecture we discuss two of them: FASTA and BLAST.

11.1 FASTA

FASTA heuristic belongs to the family of programs for sequence database search called FAST. All the heuristics in the FAST family follow three basic steps:

I. Use a heuristic to find a good local alignment of U with $V_1, \ldots, V_N$. Return only those alignments, whose score is greater than T, where T is some threshold parameter chosen in advance. Let ρ_i be the *initial score* of the alginment (returned by the heuristic) between U and V_i.

II. Sort ρ_is in non-increasing order: $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_M > T$.

III. For the highest ranking sequences, compute the *optimized score* using the dynamic programming algorithm. Return the top few alignments ranked according to their optimized score. "In practice, when the sequences are truly related, the optimized score is usually significantly higher than the initial score. This observation often helps distinguish between good alignments occuring by chance and true relationships." [3]

The general idea behind the heuristic is very simple: first, we quickly compute a coarse estimation for the scores of the sequences, and then we refine the scores for the top ranking few (using dynamic programming). It is obvious that the success of this technique depends largely on the effectiveness of the heuristic chosen for step I.

Intuition for heuristic

As the dynamic programming algorithm for sequence alignment fills out the table, the back-references along the diagonals correspond to matches in the alignment, while the vertical and horizontal back-references correspond to gaps (see Figure 11.1). Inuitively, we want to find long diagonals in the dynamic programming table without actually going through the trouble of filling it out. When good matches (long, possibly broken diagonals) are found by the heuristic, they are merged together. This behavior categorizes the whole FAST family; FASTA itself takes an extra step: "after the best regions have been selected [FASTA] tries to join nearby regions, even if they do not belong to the same diagonal" [3].

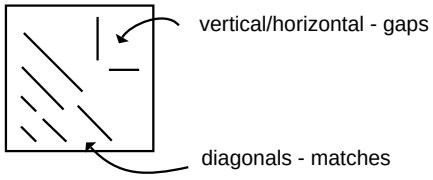

Figure 11.1: Dynamic Programming table layout: long, possibly broken diagonals correspond to matches, vertical/horizontal references correspond to gaps.

Therefore, we arrive to the following layout for an algorithm that computes ρ_i (aligning U with V_i):

I. Identify long diagonals in the dynamic programming table. If two diagonals are *close by* (next one starts after the previous one ends), merge them.

II. Choose k' largest diagonals.

III. For each of the diagonals, find its score using some scoring matrix chosen in advance.

IV. Return the top score.

The most time-consuming part of the above procedure again lies in step I, so let us examine how this step could be performed efficiently. The key observation that we need to make is that the alphabet over which the sequences are given is much smaller than the sequences themselves. Let us assume that we are comparing

sequences $U = u_1u_2 \dots u_{m-1}u_m$ and $V_i = v_1v_2 \dots v_{n-1}v_n$. Note that for any two u_i and v_j on the same diagonal, $j - i$ is the same. Let us call $j - i$ an *offset* of a diagonal (each such offset identifies a particular diagonal). Now we simply want to compute how many matches lie on a particular diagonal and then examine the diagonals that have lots of matches.

Two data structures are needed to implement the above search for diagonals with lots of matches efficiently: a lookup table of the size of the alphabet is needed (the entries will store lists of positions at which a particular character in the alphabet appears in the sequence U, see Figure 11.2), a vector indexed by offsets and initialized with 0s is needed to count the number of matches on different diagonals, see Figure 11.3.

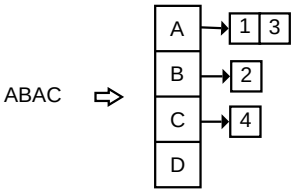

Figure 11.2: Characters of the alphabet look up table, Σ

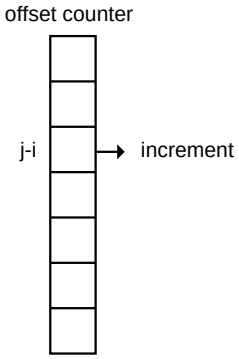

Figure 11.3: Vector indexed by offsets, O

Algorithm 1 For identifying long diagonals in the dynamic programming table for sequences U and V

```
Initialize the lookup table Σ
Initialize offset vector O
for each u_i ∈ U do
   Σ[u_i] → append(i)
end for
for each v_j ∈ V do
   for each i ∈ Σ[v_j] do
      increment O[j − i]
   end for
end for
return O
```

Once the diagonals have been identified, separate regions in them are joined together and scored with a given scoring matrix — in other words, gapless local alignments are found. It is also important to note that in practice the sequences are processed not by single characters but by k-tuples of characters [3].

While it is possible to construct an example on which the heuristic will take $O(n^2)$ time to run, in practice, the algorithm takes advantage of the fact that alignment matrices (constructed by the dynamic programming algorithm) are usually sparse (there are usually schemes in place to decide when to stop if the computation of the score gets too expensive).

11.2 BLAST

BLAST is an acronym for Basic Local Alignment Search Tool. "The BLAST programs are among the most frequently used to search sequence databases worldwide" [3]. BLAST works by finding *seeds*, "which are short segment pairs between the query and a database sequence". The seeds are then extended in both directions until the maximum possible score for an extension of the particular seed is reached (mechanisms are in place that determine when an extension should terminate without getting lost in local maxima).

The algorithm follows three major steps:

I. **Preprocessing query**. Find all words x of length w (called w-mers, where w is a parameter for the algorithm that varies based on whether DNA or protein sequences are being compared) such that there is a substring y of length w in U, such that score$(x, y) \geq T$, where T is another (threshold) parameter for the algorithm. In other words a list of high-scoring w-mers is computed; note that this list may not contain all query w-mers, e.g., if a w-mer consists of common amino acids, its score with itself may fall below T, and it may be left out. [3]

   All such words are stored in a hash table $X = \{x_1, \ldots, x_n\}$, so that membership queries $y \in X$ take $O(1)$ time.

II. **Finding hits**. Each $V_i \in \mathcal{V}$ is scanned, and all substring of length w in V_i that are in X (seeds) are recorded in a list $\{y_1, \ldots, y_k\}$.

III. **Extending seeds**. Each y_i is extended in both directions: the seeds grow as long as a match can be found, eventually, the score function starts decreasing. However, the cut-off does not happen immediately since one wants to avoid local maxima, instead the extension stops when the value of the score function goes below cS_{max} for some constant $c < 1$, where S_{max} is the maximum score seen so far. The sketch of the statistical analysis of how the cutoff point should be determined is given below. *Clearly, this step is the most expensive one since the number of the hits and the sizes of the extensions can be large.*

An improvement for step II can be achieved by replacing a hash table with a deterministic finite automaton [3] that has transitions for each character in the alphabet, and that recognizes with its states words from the list of high-scoring words. In effect, the automaton moves a window over the sequences in the database in a very computationally cheap manner: one transition is made per character. However, since the most expensive step is step III, the improvements obtained from using a finite automaton are less significant than one may expect.

[3] gives a sketch of the main points of the statistical theory behind the extensions performed by BLAST that allows one to derive various parameters needed by BLAST. Given two random sequences s and t of lengths m and n, the following approximations can be obtained:

> Given a matrix of replacement costs s_{ij} for the pairs of characters in the alphabet and the probability p_i of occurrence of each individual character in the sequences, we first compute a value λ, solving the equation
>
> $$\sum_{i,j} p_i p_j e^{\lambda s_{ij}} = 1$$
>
> The parameter λ is the unique positive solution to this equation and can be obtained by Newton's method. Once λ is known, the expected number of distinct segment pairs between s and t with

score above S is

$$Kmne^{-\lambda S}$$

where K is a calculable constant. Actually, the distribution of the number of segment pairs scoring above S is a Poisson distribution with mean given by the previous formula. From this, it is easy to derive expressions for useful quantitites like the average score, intervals where the score will fall 90% of the time, and so on.

11.3 BLAST Optimizations and PSI-BLAST

A number of optimizations for BLAST are presented in [1]:

1. The first improvement is based on the observation that the hits that correspond to some similarities between sequences (and not just random coincidencies) do not happen solitarily, but rather close together, in groups. Therefore, only hits that have other hits nearby should be expanded. The "two-hit" method that is proposed in the paper works by finding two non-overlapping word pairs on the same diagonal and within a distance A of each other, before extension is performed. To avoid missing similarities between sequences, the method requires lowering of the parameter T, which produces many more hits. However, since only few of them are extended, the running time decreases. [1] See Figures 11.4, 11.5.

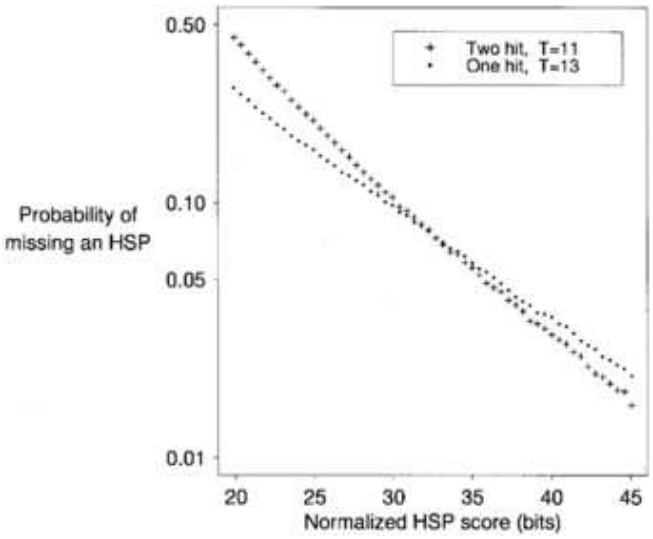

Figure 11.4: The sensitivity of the two-hit and one-hit heuristics as a function of 'high-scoring segment pair' (HSP) score. The two-hit method is more sensitive for HSPs with score at least 33 bits. [1]

2. In practice, BLAST finds several alignments of the query with a *single* database sequence, which are significant when considered together. To avoid the possibility of overlooking any one such alignment, the ability to perform gapped alignments is described in [1]. The approach suggested in the paper is to perform gapped alignments (using a dynamic programming algorithm) around the hits that exceed

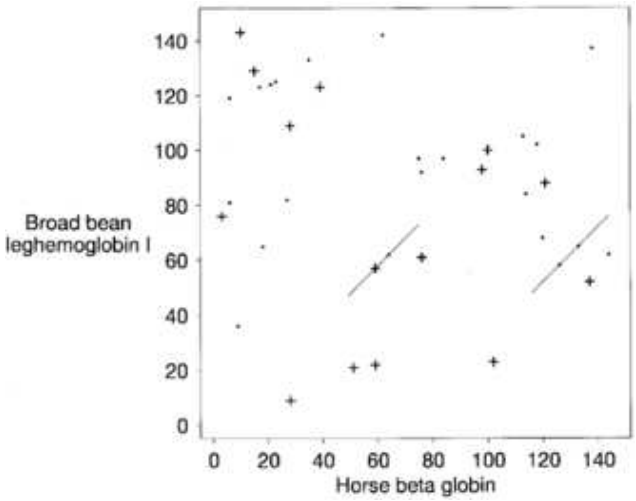

Figure 11.5: The BLAST comparison of broad bean leghemoglobin I and horse β-globin. The 15 hits with score at least 13 are indicated by plus signs. An additional 22 non-overlapping hits with score at least 11 are indicated by dots. Of these 37 hits, only the two indicated pairs are on the same diagonal and within distance 40 of one another. Thus the two-hit heuristic with $T = 11$ triggers two extensions, in place of the 15 extensions invoked by the one-hit heuristic with $T = 13$. Because this is just ony example, the relative numbers of hits and extensions at the various settings of T correspond only roughly to the ratios found in a full database search. [1]

a certain pre-defined score S_g, chosen so that no more than about one extension per 50 database sequences is invoked. Even though gapped extensions take much longer to execute, since so few of them are performed, the total running time can be kept low. Since a gapped alignment is discovered instead of a collection of ungapped ones, only one of the hits needs to be discovered to generate successful combined result.

As a result, algorithm possesses much higher tolerance for missing any single hit. As a simple example in [1] shows, if we consider just two hits, each with probability P of being missed, and if we want to find a combined result with probability at least 0.95, the orginial BLAST algorithm would need to find both segments, which would happen with probability $(1-P)^2$. Then, $2P - P^2 \leq 0.05$, or $P < 0.025$. On the other hand gapped BLAST requires only that at least one of the segments was found, which happens with probability $1 - P^2$, as a result, $P^2 \leq 0.05$, and thus P of up to 0.22 can be tolerated. The improvement is significant.

3. Position-specific iterated (**PSI**)-**BLAST**. The idea for this approach lies in constructing a position-specific scoring matrix (PSSM) out of the sequences discovered by BLAST, and then running BLAST on the sequence database with the PSSM as its input.

   PSSMs show how similar a given sequence is to the sequences used to create the scoring matrix. A query sequence of length L and a substitution matrix of dimension 20×20 (in case of protein alignment) can be replaced by a PSSM of dimension $L \times 20$ with weights for scoring a particular amino acid in any particular position calculated from the sequences obtained by searching a database using BLAST.

   As [1] shows, just a few iterations of BLAST that generate PSSMs and feed them into the next iteration

greatly increase sensitivity to weak but biological relevant sequence relationships that maybe missed using plain BLAST.

For details on how the position-specific score matrices should be constructed out of BLAST's output, and how BLAST needs to be modified to be able to run on position-specific score matrices as well as general discussion (and introduction) of PSI-BLAST, see [1].

According to [1] the above optimizations, "yield a gapped BLAST program that runs at approximately three times the speed of the original".

References

[1] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller and David J. Lipman. *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.* Nucleic Acids Research, 1997, Vol. 25, No. 17, pp. 3389-3402.

[2] *NCBI Homepage*. *http://www.ncbi.nlm.nih.gov/Database/index.html*

[3] Carlos Setubal, Joao Meidanis. *Introduction to Computational Molecular Biology.* Sections 3.5.2, 3.5.3. Brooks Cole, 1997.