Computational Biology
Lecture 7: Database search
Saad Mneimneh

Due to latest advances and research in sequencing technology, large centralized databases have been created. When querying such large databases for sequence similarities, it is not feasible to use quadratic time algorithms like the dynamic programming ones we have seen. Instead, heuristics are used. Among those heuristics are BLAST and FAST.

In addition, simple scoring schemes like +1 for a match, -1 for a mismatch and -2 for a gap are not justifiable biologically, especially for amino acid sequences (proteins). Instead, more elaborated scoring functions are used. These scores are usually obtained as a result of analyzing chemical properties and statistical data for amino acids and DNA sequences.

For example, it is known that same size amino acids are more likely to be substituted by one another. Similarly, amino acids with same affinity to water are likely to serve the same purpose in some cases. On the other hand, some mutations are not acceptable (may lead to demise of the organism). PAM and BLOSUM matrices are amongst results of such analysis.

BLAST: Basic Local Alignment Search Tool

BLAST stands for Basic Local Alignment Search Tool. BLAST returns a list of high scoring segment pairs between the query sequence and sequences in the database. Here, a *segment* is a substring of one sequence, and a *segment pair* is a pair of segments having the same length, one from each sequence. Therefore, a segment pair represents an ungapped alignment. When presented with a query sequence, BLAST attempts to find all segment pairs between the query and the database sequences that score above a threshold S. Using the concept of segment pairs, the computation of the score does not involve gaps and, therefore, the basic version of BLAST produces ungapped alignments.

As a starting point, BLAST finds certain "*seeds*". *Seeds* are very short segment pairs between the query and the database sequence. These seeds are then extended in both directions - without gaps - until a maximal scoring segment pair is obtained. However, in order to improve the time complexity, the extension in one direction stops as soon as the score drops by a value X below the best score computed for shorter extensions (for protein sequences, the value of X is 20). Therefore, the extended segment pair may not qualify as a high scoring segment pair, even if a score $\geq S$ is possible by a larger extension. This inaccuracy can be made negligible as we will see later. In other words, there is a very small chance of missing the correct extension, but in practice the tradeoff is highly acceptable.

A natural question is now what are these seeds and how are they computed? A seed is a string (word in BLAST jargon) of size k, also known as k-mer (typically k is 3 or 4), that scores at least T when compared to some word in the query sequence. As the database is scanned, the location of those words in the query sequence determine the seeds.

- Compile a list of high scoring words, k-mers that score $\geq T$
- Scan the database for hits to this word list
- Find location of hits in query (each gives a seed)
- Extend seeds until score drops below X from highest
- Return segment pairs with score $\geq S$

One possible way of generating the k-mers is by scanning the query sequence and obtaining all length k substrings. A k-mer is kept if it scores at least T with itself (one option is to actually keep them all regardless of their score). For protein sequence, in addition to the original list of k-mers obtained this way, all *neighboring k-mers* are also generated. A neighboring k-mer is a k-mer that scores at least T with an original k-mer.

In the example of Figure 1, a possible k-mer is PQG. All of its neighboring words are generated by trying all possible replacements of each amino acid in the word at a time. For instance, Figure 1 shows replacements for amino acid Q in PQG. The ones that score $\geq T$ (13 in this case) are considered.

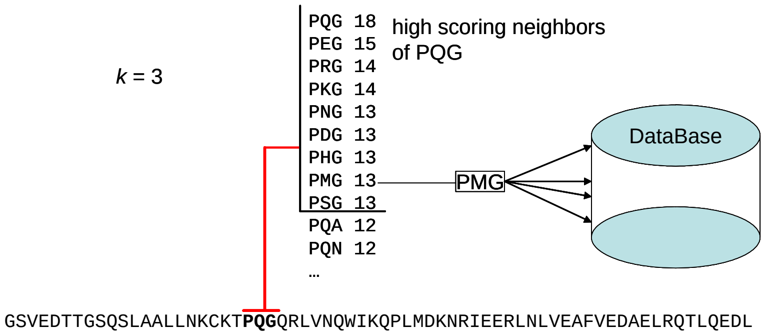
k = 3
PQG 18
PEG 15
PRG 14
PKG 14
PNG 13
PDG 13
PHG 13
PMG 13
PSG 13
PQA 12
PQN 12
…
high scoring neighbors of PQG
PMG
DataBase
GSVEDTTGSQSLAALLNKCKTPQGQRLVNQWIKQPLMDKNRIEERLNLVEAFVEDAELRQTLQEDL

Figure 1: k-mers and their neighbors

Finding the hits in the database can be efficiently achieved by indexing the database by words. Each word points to a list of all the sequences that contain it. Once a sequence is found, it is scanned sequentially for hits.

When the hit is found in the database, we need to search the query sequence for all occurrences of the k-mer. This can be done in two ways. For protein sequences for instance, each k-mer can be used to index an array of size 20^k. The i^{th} entry of the array points to the list of all occurrences in the query sequence of the i^{th} k-mer. This can be modified using hash tables to use far fewer than 20^k entries. Another way, which is the one actually used, is to build a finite state automaton for the k-mer in question, and scan the query sequence through the automaton to determine all occurrences in time linear in the size of the query. For example, here's a finite state automaton for the string *ababaca*.

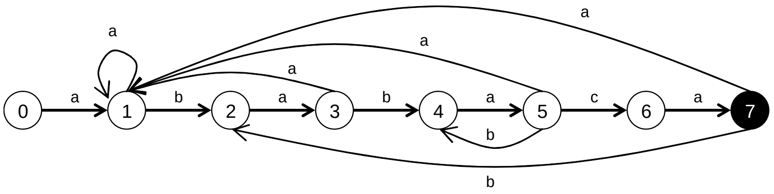

Figure 2: Finite state automaton for the string *ababaca*. If a state i has no outgoing edge labeled x for some x in the alphabet, then $\delta(i, x) = 0$.

Starting from state 0, we transition based on the characters seen in the query sequence. The finite state automaton keeps track of the longest prefix of *ababaca* that is also a suffix of the query read so far. In general, for a pattern P, let P_i be the i^{th} prefix of P. As we read the query, and transition accordingly, if we stop at state i, then P_i is the longest prefix of P that is also a suffix of the query. For instance, let's see why $\delta(5, b) = 4$. This means that if we are in state 5, and read a b, we make a transition to state 4. Being in state 5 means that $P_5 = ababa$ is the longest prefix that is also a suffix of the query. If we read a b, we must now look at $P_5b = ababab$ and determine the longest prefix that is also a suffix (it makes no sense to go back further in the query because it was already determine that P_5 is the longest prefix so far). Examining P_5b, we determine that the longest prefix of P that is also a suffix of the query is $abab = P_4$, and hence we transition to state 4.

As the above example illustrates, the finite state automaton for a k-mer can be construted in a straightforward way by explicitly computing the transition function for every state and every character of the alphabet. If the alphabet is Σ, this will take $O(k^3|\Sigma|)$ time (k^2 time for determining the longest prefix for every state and every character). However, it is possible to construct it in $O(k|\Sigma|)$ time using more elaborate techniques (Knuth-Morris-Pratt algorithm).

Every time we hit the last state, we find an occurrence of the pattern in the query (seed). Once a seed is found, it is extended in both directions (Figure 3), and stop the extension in one direction if it causes the score to drop by X below the highest score for shorter extensions.

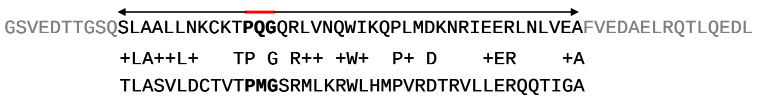
```
GSVEDTTGSQSLAALLNKCKTPQGQRLVNQWIKQPLMDKNRIEERLNLVEAFVEDAELRQTLQEDL
          +LA++L+    TP G R++ +W+  P+ D    +ER    +A
          TLASVLDCTVTPMGSRMLKRWLHMPVRDTRVLLERQQTIGA
```

Figure 3: Extending seeds

Stoping an extension as described above may result in missing a higher scoring extension, and hence missing an actual segment pair with score $\geq S$. Based on the score statistics, however, the probability of such an event can be made negligible, e.g. 0.001. To see this, we need to understand the statistical model of the scoring matrix, also known as the substitution matrix. A substitution matrix s, defined by its entries s_{ij}, determines the score of aligning character i with character j. Now consider two independent random sequences of length m and n, where character i occurs with probability p_i. We require that at least one s_{ij} is positive and

$$E[s_{ij}] = \sum_{i,j} p_i p_j s_{ij} < 0$$

Without this requirement, the segment pair with the highest score probabilistically tends to cover the entire sequence. This requirement ensures also that there exists a unique positive solution λ that satisfies the following:

$$\sum_{i,j} p_i p_j e^{\lambda s_{ij}} - 1 = 0$$

This can be verified by finding the first derivative of the above sum with respect to λ and confirming that it is negative for $\lambda = 0$. Furthermore, we can find the second derivative and confirm that it is always positive. This means that the function drops below 0 when λ moves away from 0 but inscreases back to infinity. Therefore, it must cross 1 once.

The theory then says that M, the maximal segment score satisfies:

$$Prob\{M > \frac{\ln mn}{\lambda} + x\} \approx 1 - e^{-Ke^{-\lambda x}}$$

where K is a computed constant. For large x, and by expanding $e^a = 1 + a + a^2 + \ldots$, we have

$$Prob\{M > \frac{\ln mn}{\lambda} + x\} \approx Ke^{-\lambda x}$$

Therefore, if $S = \frac{\ln mn}{\lambda} + x$, we have:

$$Prob\{M > S\} \approx Kmne^{-\lambda S}$$

Now the following is not so rigorous but if we think of $e^{-\lambda S}$ as a probability, and mn as the number of chances available, then we can percieve that the occurrence of segments with score $\geq S$ is the sum of mn independent Bernoulli trials with success probability $e^{-\lambda S}$. Of course, this is not so true because while the two sequences are assumed independent, the pairs comprising the MSP do exhibit an interdependency. Therefore, K adjusts for that and we use Kmn instead for the number of chances. With this analysis, and since $Kmn \gg e^{-\lambda S}$, the number of segments with score $\geq S$ can be closely approximated by a Poisson process with parameter $Kmne^{-\lambda S}$. Therefore, if $y = Kmne^{-\lambda S}$, the probability of having i such segments is

$$\frac{y^i e^{-y}}{i!}$$

and the probability of finding c or more such segments is

$$1 - e^{-y} \sum_{i=0}^{c-1} \frac{y^i}{i!}$$

S can then be chosen to make $1 - e^{-y}$ small enough, e.g. 0.001, which means that the probability of finding a high scoring segment is small, and hence the chance of missing it is small.

BLAST reports the score S in bits by computing $S' = \log_2 \frac{e^{\lambda S}}{K}$. Therefore, the reported score depends on the values of λ and K and not just the entries given in the substitution matrix s. This is done to make all scoring systems comparable from a statistical point of view, and we will see why this is true later. The value $Kmne^{-\lambda S} = mn/2^{S'}$ is called the e-value (number of segment pairs with score $\geq S'$ that is expected by chance).

Now recall that BLAST searches for segment pairs with score $\geq S$ by assuming that such segment pairs must contain a k-mer with a score of at least T. This decision to work with k-mers is not arbitrary. If two sequences have a level of similarity (say $L\%$), then it is guaranteed they contain a preserved k-mer for some value of k. Indeed this is justified by the pigeonhole principle. As a quick illustration of the pigeon hole principle, suppose there are 91 pigeons and 10 holes. The principle states that at least one of the holes will hold 10 or more pigeons (if all 91 pigeons are inside the holes). The proof is by contradiction. Suppose none of the holes holds 10 or more pigeons, or, equivalently, all the holes have at most 9 pigeons. Having 10 holes with at most 9 pigeons in each sums up to having a maximum of 90 pigeons, which contradicts the starting point - namely that we have 91 pigeons.

We can apply the same principle for sequences. Assume we have two sequences of length 100 with more then 90% similarity between them. We claim that there must be a preserved 10-mer between these two.

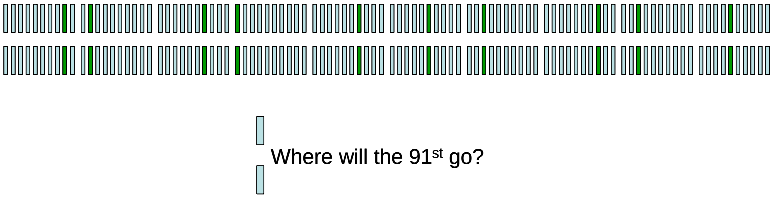

Figure 4: 100 length sequences with $> 90\%$ similarity

Figure 4 shows two sequences of length 100 divided into blocks of size 10. Since the similarity level is greater than 90%, then there are at least 91 positions where the two sequences match. By selecting these position in any way possible, at least one block will hold 10 of them, i.e. a preserved 10-mer. Therefore, stated in another way, if we are looking for sequences that are more then 90% similar to our query, then why bother searching a sequence if it does not contain a preserved 10-mer?

In case similarities are randomly distributed, the chance of having even a bigger k-mer is increased. Therefore, statistically the same principle should apply. However, a preserved k-mer does not necessarily score at least T. What is the probability that a high scoring segment pair ($\geq S$) will fail to contain a k-mer with score at least T? Experiments show that this probability decreases exponentially with increased S, which is intuitive. For instance, when $T = 17$ and $k = 4$, BLAST misses only about 20% of the high scoring segment pairs with $S = 55$, and only 10% with $S = 70$.

BLAST also knows a number of variations. Couple of these are *two-hit* BLAST (gapped) and psi-BLAST. Two-hit BLAST is based upon the following observations:

- most of the computational time is spent on extending hits (seeds)
- a high scoring segment pair is much longer than a k-mer

The first observation implies that we waste time if a hit is extended only to be dropped afterwards because it scores $< S$. On the other hand, the second observation implies that a high scoring segment pair may entail multiple hits on the same diagonal and within a relatively short distance. We can define the diagonal of a hit involving k-mers starting at positions (x_1, x_2) of the database and the query respectively as $x_1 - x_2$. Two-hit BLAST invokes an extension only when two non-overlapping hits are found within a distance A e.g. 40 of one another on the same diagonal.

- An array is used to record for each diagonal the first coordinate x_1 of the most recent hit found and its length (possibly an extended hit).
- Any hit that overlaps the most recent one is ignored. Since database sequences are scanned sequentially, the coordinate x_1 always increases for successive hits and, therefore, detecting an overlap is easy.
- When a non-overlaping hit is found to be within a distance A of the most recent hit, it is extended.

Because two hits are required to invoke an extension, the value of T is lowered to retain comparable sensitivity. Many more single hits may be found, but only a small fraction of them are extended.

If a segment pair with score greater than some threshold $S_g > S$ is found, this triggers a gapped alignment in the following way. The length-11 segment with the highest alignment score within the segment is determined and its central residue is used as a seed to start a bounded dynamic programming in both directions around the seed. The bounded dynamic programming ignores entries in the matrix for which the score drops below a threshold X_g. A gap of length k costs $10 + k$, i.e. this corresponds to $e = 11$ (opening of a gap) and $d = 1$ (extension of a gap) in the affine gap penalty model.

psi-BLAST stands for position specific iterated BLAST. With psi-BLAST, BLAST is run in iterations, but each iteration adjusts the scoring scheme based on the result of the previous iteration. The substitution matrix s (for instance a 20x20 matrix for protein sequences) is used for the first iteration. Subsequence iterations compute a 20xn matrix t known as position specific matrix. In this matrix, t_{ij} is the score of aligning amino acid i with the j^{th} amino acid of the query sequence. In other words, the score of aligning two amino acids depends on the position in the query sequence. The matrix t is computed using the high scoring segment pairs obtained from the previous iteration. For a given position j in the query, all such segments that overlap with j will contribute in adjusting the score and computing t_{ij} for $i = 1 \ldots 20$. The iterations stop when no new statistically significant (low e-value) segments are found.

FAST

FAST finds similarities between sequences in a different way, but it also attempts to find all occurrences of words of certain size k (substrings of size k) in the two sequences x and y being compared. A central term for FAST is *offset* and is used to specify the relative position of two words: if a word occurs at position i in x and position j in y, we say it occurs at an offset $i - j$. According to this definition of offset, the extreme cases are when i is maximal and j is minimal and vice versa. Since $1 \leq i \leq m$, and $1 \leq j \leq n$, the offset ranges between $1 - n$ and $m - 1$.

Here is an example. Let the word size $k = 2$, $x = AGAGAG$, and $y = AAGAGAG$. Looking at word AG, it occurs at x_1, x_3, x_5, and also y_2, y_4, and y_6. Considering the pair (x_1, y_4), the resulting offset is $-3 = 1 - 4$. What this really means is that aligning x and y at offset -3 results in a perfect alignment for the word AG (A in x matches A in y, and G in x matches G in y). The offset is essentially the same as a diagonal in BLAST. However, FAST simply finds the alignment that maximizes the number of perfectly aligned words.

Before presenting the algorithm, we take a quick look at the data structures used by FAST: a lookup table holding all substrings (words) of x of size k along with their occurrences in x. Also, an offset vector that holds for each offset value the number of times that offset occurs. Since the offset range is from $1 - n$ to $m - 1$, the length of the offset vector is $m + n - 1$.

As we scan y, we identify for every substring of size k whether a corresponding word exists in the lookup table, and if it does, we increment the appropriate entries in the offset vector based on the positions stored for that word.

- Fill the lookup table using information about x.
- Update the offset vector as you scan y
- Choose the most frequent offset
- Align x and y at that offset

So far FAST was presented as ungapped (it actually produces a semi-global alignment where gaps are present only at the extremities), where the two sequences are aligned at an offset.

Like BLAST, FAST also has variations that consider gaps. In this case the resulting offset is used initially to align the two sequences. This alignment corresponds to some diagonal in the dynamic programming table. Then a bounded dynamic programming around this diagonal is performed to find a better gapped alignment.

References

Setubal J., Meidanis, J., Introduction to Molecular Biology, Chapter 3.