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 being used. Among those heuristics are BLAST and FAST.

Also, simple scoring schemes like +1 for a match, -1 for a mismatch and -2 for a gap aligned with a symbol are not suitable anymore, especially for amino acid sequences (proteins). Instead, more elaborated scoring functions are being 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.

Next, BLAST is presented followed by an overview of FAST.

**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 a sequence, and a segment pair is a pair of segments having the same length. When presented with a query sequence, BLAST returns all segment pairs between the query and database sequences that score above a threshold $S$; computing the score is done without considering gaps, and this is why only same length segments are considered when forming a segment pair. The basic version of BLAST produces therefore ungapped alignments. Next, a few details about how BLAST works are revealed.

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 - and a maximum scoring segment pair is obtained. In order to improve the time complexity, the extension process stops as soon as any extension will make the score drop under a carefully computed value $X$. In doing this time optimization, there is a very small chance of missing the correct extension, but in practice this tradeoff is highly acceptable.

A natural question is now what are these seeds and how are they computed? As a first step, BLAST compiles a list of short high-scoring strings (words in BLAST jargon). Next, the database is searched for hits within the word list. Once these seeds are determined, the extension process begins. The algorithmic steps are:

- Compile list of words.
- Search for hits (each gives a seed).
- Extend seeds.
- Return segment pairs with score $> S$.

To further develop the idea, another question is to be answered: How is the list of words obtained? Depending on what kind of sequences (DNA or protein) are being compared, the process is slightly different. Next we’ll present the steps for both.

BLAST works with $k$-mers, which are simply words (substrings) of length $k$.

In the case of DNA, BLAST first generates all of the query $k$-mers (all substrings of length $k$ that are found in the query sequence), with $k$ typically 3 or 4. At this point, a filtering step is applied to eliminate very commonly occurring words. Next, these $k$-mers are compared for hits against all $k$ length words in the database.

For a protein sequence, in addition to the query $k$-mers, all neighboring $k$-mers are generated. Here, a neighboring $k$-mer is any $k$-mer that scores high with some $k$-mer of the query sequence.

In order to faster retrieve information, the database is hashed and indexed by all words of size $k$. Each word of size $k$ will point to all the locations in the database where it exists. Since we only have $4^k$ possible words in case of DNA and $20^k$ in case of proteins, it makes sense to use them as index because their number is much less than the database size.

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 above a threshold (in our case 12) are considered high scoring neighbors. These $k$-mers
(including PQG of course) are then looked for into the database (each is an index for the database), and all location in the database pointed to by these $k$ -mers will constitute seeds.

Each seed is then extended in both directions (Figure 2) until any further extension causes its score to drop below $X$.

All in all, the steps of the BLAST algorithm are:

- Split query into overlapping words of length $k$ ($k$-mers).
- For each $k$-mer find neighboring $k$-mers that score above a threshold $T$, and consider them words, too.
- Look into the database where these words occur. This will give the initial seeds.
- extend every seed until its extension causes its score to drop under a threshold $X$.
- return segment pairs that score above $S$.

The decision to work with $k$-mers is not taken at random. 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 are inside those 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 inside, 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 3 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.

In the case of a random distribution of similarities - which is better mapping the reality - the chance of having an even bigger \( k \)-mer is increased.

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?

Following, a short time complexity analysis of the algorithm: let \( n \) be the length of the query sequence, \( s \) the number of seeds, and \( L \) the length of the alignment. The running time is then \( O(n + Ls) \). It is possible, as a worst case scenario, to have the number of seeds obtained and their lengths comparable to the sequence length, i.e. \( s = O(n) \) and \( L = O(n) \) (consider for instance a database containing the query sequence itself). In this extreme case the complexity reaches \( O(N^2) \). Anyway, in practice it performs better than Smith-Waterman and its time limit is closer to being linear.

BLAST also knows a number of variations. Couple of these are 2-hit BLAST and gapped BLAST.

In case of 2-hit BLAST a speed up of the algorithm is achieved by requiring not to consider a database sequence unless it contains two seeds that are within 40 amino acids of each other. Although theoretically the worst case time complexity is the same, this performs faster in practice because it limits the search space.

As expected, gapped BLAST allows gaps to be considered in the alignments. It first finds a seed, then finds more seeds and extends them, and finally joins segments with gaps around the main seed.

**FAST**

FAST finds similarities between sequences in a different way: it first records all occurrences of windows of certain size \( k \) (substrings of size \( k \)) in the two sequences \( x \) and \( y \) being compared. Here also there is a difference in case of DNA or proteins strings: typical values of \( k \) are 1-2 for DNA and 3-4 for proteins. A central term for FAST is offset and is used to specify the relative position of two windows: if a windows 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 for what we mean by window: let the window size \( k = 2 \), \( x = AGAGAG \), and \( y = AAGAGAG \). For example, looking at window \( AG \) 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 window \( AG \) (\( A \) in \( x \) matches \( A \) in \( y \), and \( G \) in \( x \) matches \( G \) in \( y \)).

FAST simply finds the alignment that maximizes the number of perfectly aligned windows.

Before presenting the algorithm, we take a quick look at the data structures used by FAST: a lookup table holding all possible windows of size \( k \) along with their occurrences in \( x \) and \( y \). 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 \( m + n - 1 \). The offset vector is computed from the information stored in the lookup table. These two data structures are used when performing the following steps of the FAST algorithm:

- Fill the lookup table.
- Compute the offset vector: for each offset value, find how many times it occurs in the lookup table.
- Choose the most frequent offset.
- Align \( x \) and \( y \) at that offset.

Let’s take the same example and have a run of the algorithm. Figure 4 shows the lookup table and the offset vector.

It has a total of \( 4^2 \) entries in the lookup table for \( k = 2 \), but, in our particular example of the sequences \( x \) and \( y \), only the entries corresponding to \( AA \), \( AG \), and \( GA \) will have non-void lists of indexes. The vector is filled according to the values in the table. It is first initialized with zeroes, and, whenever a new offset value is computed, its corresponding value is incremented by 1.

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.
Figure 4: FAST example

References

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