

2.1 Introduction

2.1.1 Biological Database

Biological data are organized as a database of sequences. There are three types of biological sequences: Proteins, DNAs and RNAs. Large amounts of biological sequences are deposited in public databases. In recent years, the size of the biological database has doubled in every 15 or 16 months and in molecular biology, high-similarity search has become a basic operation on the databases. The number of queries that access the database has also increased to about 40,000 per day. This necessitates the development of efficient database search methods.

2.1.2 Database searching

The Database searching could be formally defined as follows:

Given a database D of genomic (or protein) sequences and a query string Q , we look for a string S in D which is the closest match to the query string Q .

The term **closest match** means either

- S and Q have a semi-global alignment (ignoring the spaces on the two ends of Q), or
- S and Q have a local alignment.

The main goal of database search is to find the relevant information and at the same time avoid the irrelevant information. The goodness of a search algorithm could be measured by its Sensitivity and Specificity.

- **Sensitivity** is the ability to detect '*true positives*' i.e. *correct matches*. The most sensitive search finds all true matches, but might have lots of '*false positives*' i.e. *erroneous matches* detected. Sensitivity can be defined as the probability of finding the matches such that the query and the matched database sequences have at least $x\%$ similarity.

- **Specificity** is the ability to reject '*false positives*'. The most specific search will return only true matches, but might have lots of '*false negatives*' i.e. *missed correct matches*.

In general, selecting search algorithms is a trade off between these two figures of merit. A good search algorithm should be sensitive and at the same time specific.

2.1.3 Algorithms Classification

Most of the biological database searching methods are based on "local alignment" and they could be classified as follows:

- Exhaustive method
This is the most basic and also the most sensitive method. The disadvantage of this method is that it consumes too much time to search a large database. E.g. Smith-Waterman algorithm.
- Heuristic methods
These methods attempt to speed up the searching while maintaining acceptable levels of sensitivity. E.g. FastP, FastA, BLAST, BLAT, PatternHunter, ...
- Filter based and other refined methods
These methods attempt to design a filter to select candidate positions in the database where the query sequence possibly occurs with a high level of similarity. E.g. QUASAR, LSH, ...

There are also many other search methods.

2.2 Smith-Waterman Algorithm

The **Smith-Waterman Algorithm** [AGM⁺81] is the de facto standard for database searching methods. This search method compares the query with every sequence in the database using the Smith-Waterman Dynamic Programming algorithm for pairwise comparisons. This is a brute force algorithm and is the most sensitive method for finding all the closest matches in a database. Moreover, since it essentially does every possible pairwise comparison between the query sequence and the library (or the database) sequences, it is also the slowest method.

2.2.1 Algorithm

This algorithm could be described as follows:

Given a database D of total length n and the query Q of length m ,

- For every sequence S in the database, by Smith-Waterman Dynamic Programming algorithm, we compute the best local alignment between S and Q .
- Return all alignments with the best score.

The time complexity is in the order of $O(mn)$.

2.3 FastA

FastA [LP⁺88] is a heuristic database search method heavily used before the advent of BLAST. Given a database and a query, FastA does pairwise local alignment with all sequences in the database and returns the good alignments, based on the assumption that a good local alignment should have some exact match subsequences. This algorithm is much faster, but is less sensitive than the Smith-Waterman algorithm.

2.3.1 History of FastA

In 1983, Wilbur-Lipman algorithm [WL⁺84] was proposed for the analysis of protein and DNA sequence similarity. It achieved a balance between sensitivity and specificity on the one hand and speed on the other. In 1985, FastP [LP⁺85] algorithm was proposed for searching amino acid sequence databases. It identifies good alignments with no insert and delete operations (indels). Then in 1988, a new version, or in other words an improved version, of FastP, called FastA [LP⁺88] was proposed. This algorithm can identify good alignments with gaps. It increased the sensitivity with a small loss of specificity and was also as fast as the FastP algorithm.

2.3.2 FastP Algorithm

In this algorithm, there are four steps to determine the pair-wise similarity score.

Step 1: Look for hot spots

For every k -tuple (length- k substring) of the query and every k -tuple of the database sequences, if they are identical, the pair is called an **hot spot**. This step looks for all the hot spots as shown in Figure 2.1. The parameter k of the algorithm determines how many consecutive identities are required in a match.

The larger the value of k , the faster the searching is, but less will be the sensitivity. Generally,

- $k = 4 - 6$ for DNA sequences, and
- $k = 1 - 2$ for protein sequences.

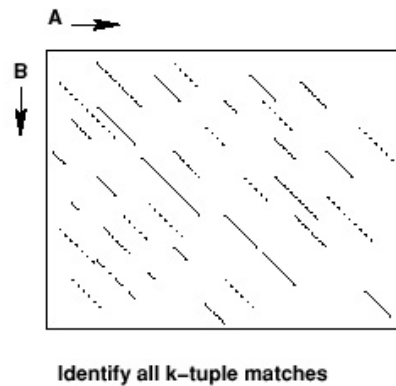


Figure 2.1: Hot spots appear as continuous diagonals on the Dynamic Programming Table

Step 2: Find the 10 best diagonal runs for every database sequence

Diagonal run is a sequence of nearby hot spots on the same diagonal (spaces are allowed between hot spots). Each hot spot is assigned a positive score. Interspot space is given a negative score that decrease with length. The score of a diagonal run is the sum of scores for hot spots and interspot spaces. This step identifies the 10 highest scoring diagonal runs for each database sequence.

Step 3: Re-score the 10 best diagonal runs for every database sequence

Re-score the 10 best diagonal runs with a substitution matrix for amino acid (or nucleotide). This substitution matrix is a scoring matrix that allows conservative replacements and runs of identities shorter than k -tuple to contribute to the similarity score. This has the effect of trimming ends that do not contribute to the score. For each of these diagonal region, a subregion with the highest score is identified. This subregion is called the “initial region” and it contains no gaps. The best score among the 10 initial regions is called the *init1* score for that sequence as shown in Figure 2.2.

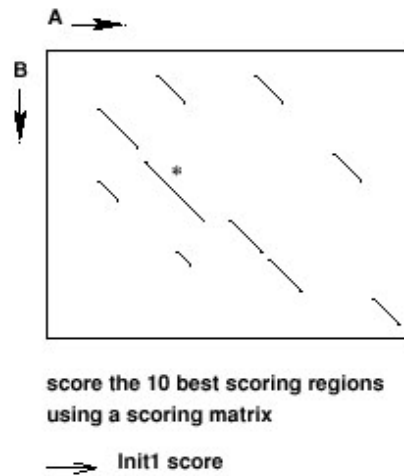


Figure 2.2: The best among these 10 subregions contribute to the *init1* score

Step 4: Rank the sequence

Then the sequences are ranked based on their *init1* scores.

2.3.3 FastA Algorithm

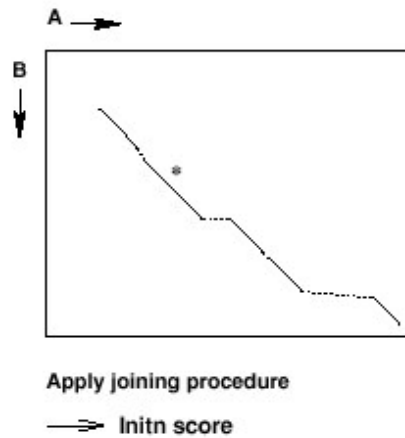
FastA algorithm attempts to find alignments with gaps. It uses the first 3 steps of the FastP algorithm. Then, it executes 2 additional steps.

Step 4: Attempt to join the sub-regions by allowing indels

After the 10 best initial regions for every database sequence, we check whether several initial regions could be joined together. First, we discard those initial regions with scores smaller than a given threshold. Then, we attempt to join the remaining initial regions by allowing indels. Given the locations of the initial regions, their respective scores, and a “joining” penalty (analogous to a gap penalty), we calculate an optimal alignment of initial regions as a combination of compatible regions with maximal score. An example is shown in Figure 2.3. The score of the combined regions is the sum of the scores of the sub-regions minus the penalty for gaps. Then the resulting score is used to rank the database sequences. The best score at this stage is still not fully optimized and is called the *initn* score for this sequence.

Step 5: Banded Smith-Waterman Dynamic Programming

Sequences with *initn* scores smaller than a threshold are discarded. Then, for the

Figure 2.3: Join the gaps to get an *initn* score

remaining sequences, banded Smith-Waterman dynamic programming algorithm is applied to get the optimum alignment and score, as shown in Figure 2.4. Finally, the sequences are ranked based on their optimum scores.

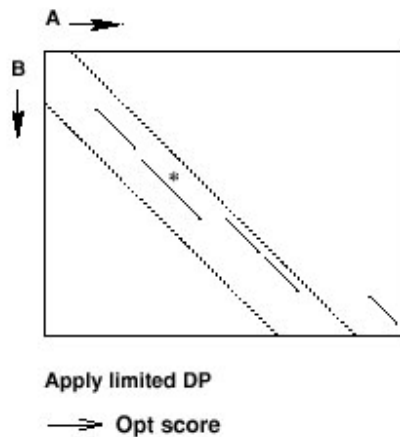


Figure 2.4: Apply banded Smith-Waterman Dynamic Programming algorithm to get the optimal score

2.4 BLAST

BLAST stands for Basic Local Alignment Search Tool [Gal00]. Given, a sequence database D and a query sequence s , the BLAST algorithm directly approximates

alignments that optimizes the *maximal segment pair* (MSP) score. First, it is essential to understand the term MSP.

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 spaces.
- A *locally maximal segment* is a segment pair whose alignment score (without spaces) will be reduced by expanding or shortening the segment on either side.
- A MSP in S_1, S_2 is a segment pair with *maximum score* over all segment pairs in S_1, S_2 .

This heuristic algorithm can be applied in a variety of contexts including DNA and protein sequence database searches, motif searches, gene identification searches, and in the analysis of multiple regions of similarity in long DNA sequences. BLAST is designed for speed and is less sensitive.

2.4.1 History of BLAST

BLAST1 [AGM⁺90] was proposed in 1990. It was very fast and was dedicated to the search for regions of local similarity without gaps. BLAST2 [AMS⁺97] was created as an extension of BLAST1, by allowing the insertion of gaps. Two versions of BLAST2 were independently developed, namely, WU-BLAST2 [WUBLAST] by Washington University in 1996 and NCBI-BLAST2 [NCBIBLAST] by National Center for Biotechnology Information in 1997.

2.4.2 BLAST1 Algorithm

BLAST1 is a heuristic method, which searches for local similarity without gaps. It permits a tradeoff between speed and sensitivity, with the setting of a threshold parameter, T . A higher T yields greater speed, but also an increased probability of missing weak similarities. There are 3 steps.

Step 1: Query Preprocessing

For every position p of the query sequence s , BLAST finds the list of w -tuples (length- w substring) scoring more than a similarity threshold T , when compared with the word of the query starting at position p as shown in Figure 2.5. This list of words are called **neighbors**.

BLAST uses a word size $w = 3$ for peptide sequences (protein) and $w = 11$ for nucleic acid sequences (DNA). Generally, the similarity threshold T is set to

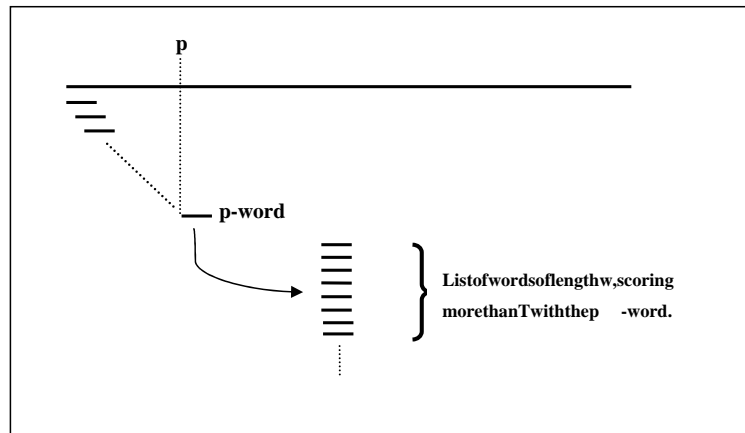


Figure 2.5: Preprocessing of the query

a value of “0”.

Adipokinetic hormone II - Migratory locust

```

q l n f s a g w
q l
  l n
    n f
      f s
        s a
          a g
            g w

```

Figure 2.6: Creating an initial word list for a peptide sequence, word size $w = 2$

In the simplified example shown in Figures 2.6 and 2.7, a word size $w = 2$ and a threshold $T = 8$ are used. Figure 2.6 shows the list of initial words while Figure 2.7 shows the list of expanded words which consists of 47 words. The expanded list contains any word that scores at least eight when aligned with the initial word and is scored with the PAM 120 similarity table. The term **PAM**, is an acronym for “point accepted mutation” or “percent accepted mutation”. It is used as a *unit* to measure the amount of evolutionary divergence (or evolutionary distance) between two amino acid sequences. In biological database searching, PAM is used to refer to certain amino acid substitution matrices (scoring matrices), whose scores has a relationship to PAM units.

Initial

Word	Expanded List
ql:	ql, qm, hl, zl
ln:	ln, lb
nf:	nf, af, ny, df, qf, ef, gf, hf, kf, sf, tf, bf, zf
fs:	fs, fa, fn, fd, fg, fp, ft, fb, ys
sa:	no words score 8 or more, including the initial word sa
ag:	ag
gw:	gw, aw, rw, nw, dw, qw, ew, hw, iw, kw, mw, pw, sw, tw, vw, bw, zw, xw

Figure 2.7: Generation of neighbors to create an expanded word list

Step 2: Scan the database for hits

At the end of Step 1, the query is represented by lists of neighbors, one list at each position of the query. In Step 2, the database D is scanned to find whether there is an exact match between the neighbors of the w -tuple at p and a w -tuple in D . If it does, a **hit** is made. A hit is characterized by the positions in both query and database sequences. All the possible hits between the query sequence and database sequences are found as shown in Figure 2.8.

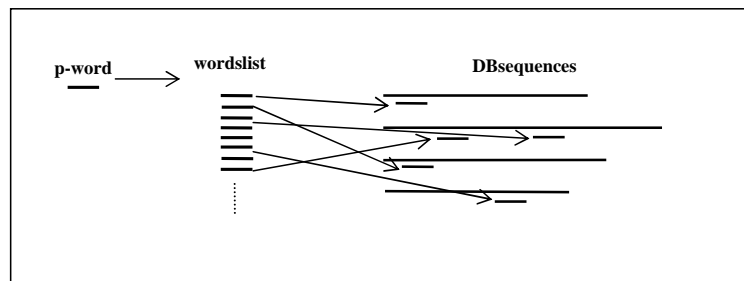


Figure 2.8: Generation of hits

Step 3: Extension of the hits

To determine whether each hit may be part of a longer segment pair with higher score, every hit that has been generated is extended **in both directions, without gaps**, but allowing mismatches as shown in Figure 2.9.

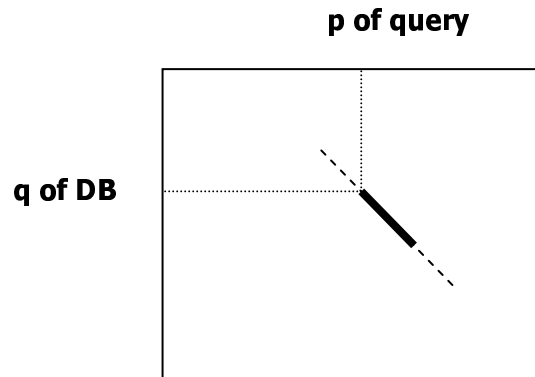


Figure 2.9: Extension of a hit

To speed up this step, any extension is truncated as soon as the score decreases by more than X (X is a parameter of the algorithm) from the highest score found so far. Because of this truncation and the use of the threshold T , BLAST is not guaranteed to find **all** segment pairs that scored better or equal to S (the cutoff score for a high scoring segment pair). In other words, some qualified segment pairs might not be detected.

If the extended segment pair has score better than or equal to the cutoff score S (set as a parameter of the algorithm), it is called an **HSP** (High scoring Segment Pair), they will be reported. For every sequence in the database, the best scoring HSP is called the **MSP**.

2.4.3 NCBI-BLAST2

NCBI-BLAST2 was developed at National Center for Biotechnology Information (NCBI). The most important feature of NCBI-BLAST2 is that it allows local alignment with gaps. The first two steps, leading to the generation of primary hits are the same as those in BLAST1. The third step includes two major refinements:

- **Two-hits requirement**

In Step 3 of BLAST1, all hits will be extended. However, in NCBI-BLAST2 only some of these hits are selected for extension. An additional requirement for a hit to be extended is the existence of another *non-overlapping hit*, on the same diagonal, within a distance smaller than A (a user-specified parameter of the algorithm, $A = 40$ for DNA). This process is illustrated in Figure 2.10. Since only a small fraction of these hits are extended, the computation time is reduced.

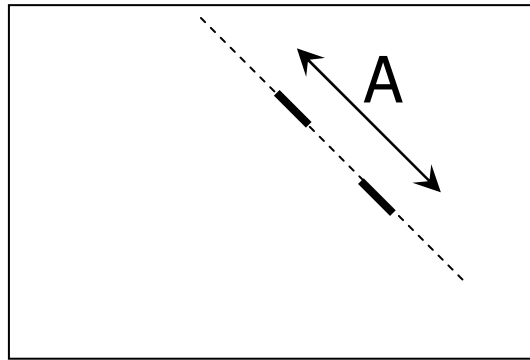


Figure 2.10: “two-hits” requirement

- **Gapped extension**

NCBI-BLAST2 performs an ungapped extension on all the hits satisfying the “two-hits requirement”. Among the generated HSP, a **gapped extension** is performed for those segment pairs with score above some threshold. They are used as the starting points for applying Smith-Waterman dynamic programming algorithm for local alignments.

The gapped extension algorithm allows gaps (insertions and deletions) to be introduced into the alignments that are returned. Allowing gaps means that similar regions are not broken into several segments. The scoring of these gapped alignments tends to reflect biological relationships more closely.

The algorithm used for computing these local gapped alignments is a modified Smith-Waterman algorithm. The Dynamic Programming matrix is explored in both directions (Refer Figure 2.11) starting from the mid-point of the *combined hit*. In addition, when the alignment score drops off by more than X_g , the extension will be truncated.

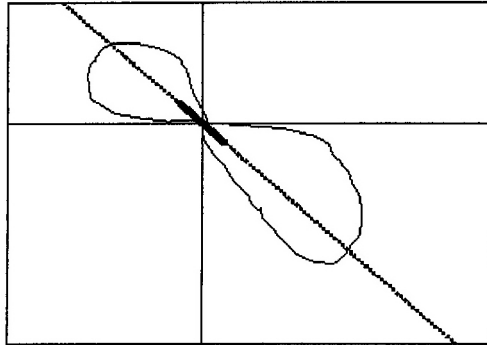


Figure 2.11: Gapped extension by score-limited Dynamic Programming

2.4.4 BLAST1 vs NCBI-BLAST2

BLAST1 spends 90 percent of its time on extension of the hits (Step 3). This shows that this is a computationally expensive operation.

NCBI-BLAST2 is about 3 times faster than BLAST1. This speedup is attributed to the “two-hit requirement”, which reduces the number of extensions significantly. The “two-hit requirement” is formulated based upon the observation that a HSP of interest is much longer than a single word pair, and may therefore entail multiple hits on the same diagonal and within a relatively short distance of one another.

Generally, with NCBI-BLAST2 the “two-hit requirement” is used for searching peptide sequences (protein) only and is not used with DNA sequences. Hence, for DNA sequence searching, NCBI-BLAST2 is slower than BLAST1.

2.4.5 BLAST vs FastA

BLAST is an order of magnitude faster than FastA, at the expense of reducing the sensitivity. BLAST may be a bit less effective than FastA in identifying important sequence matches, particularly when the most significant alignments contain spaces. Both BLAST and FastA are significantly faster than Smith-Waterman algorithm. Figures 2.12 and 2.13 summarize the main ideas behind FastA and BLAST algorithms.

FASTA Algorithm

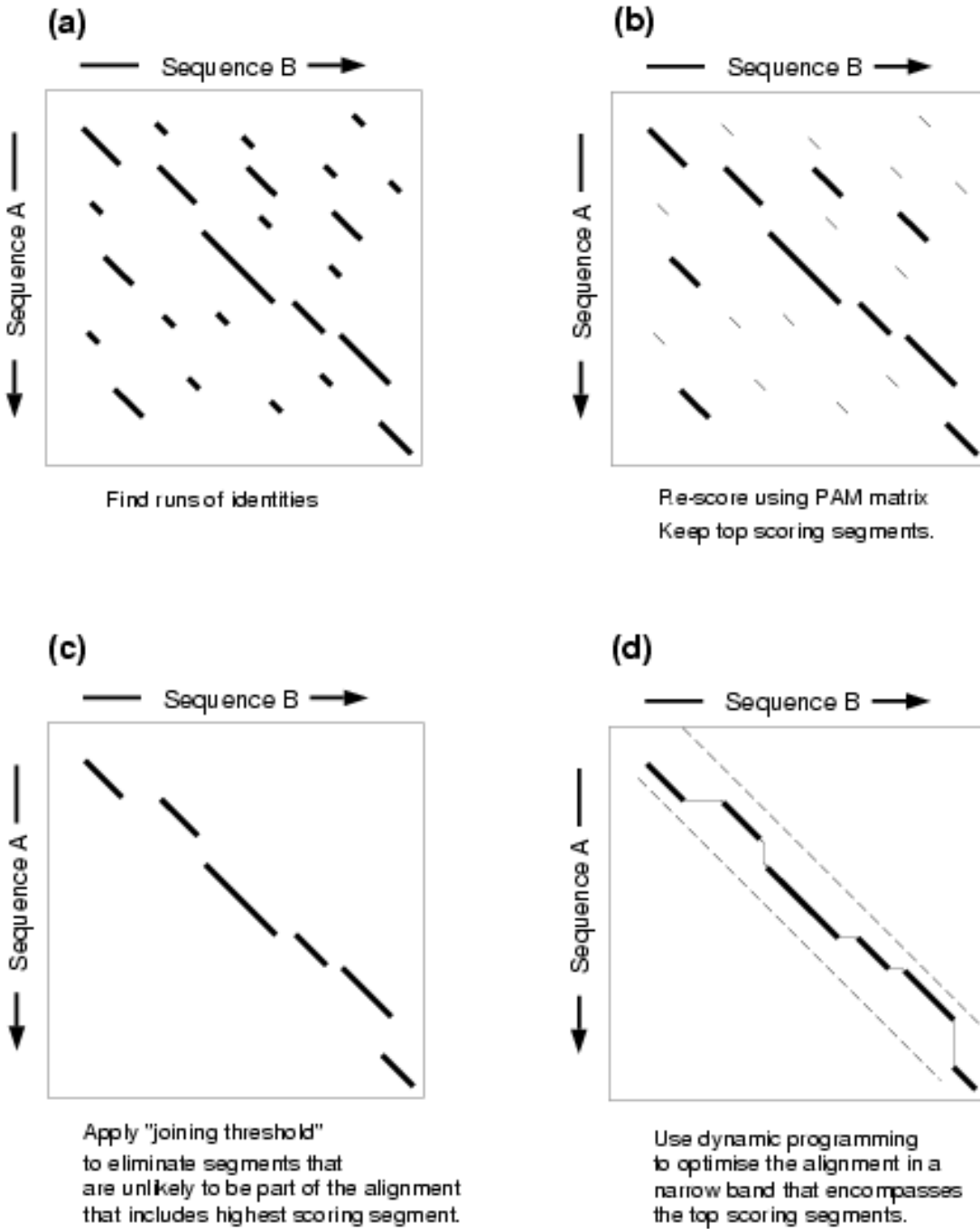
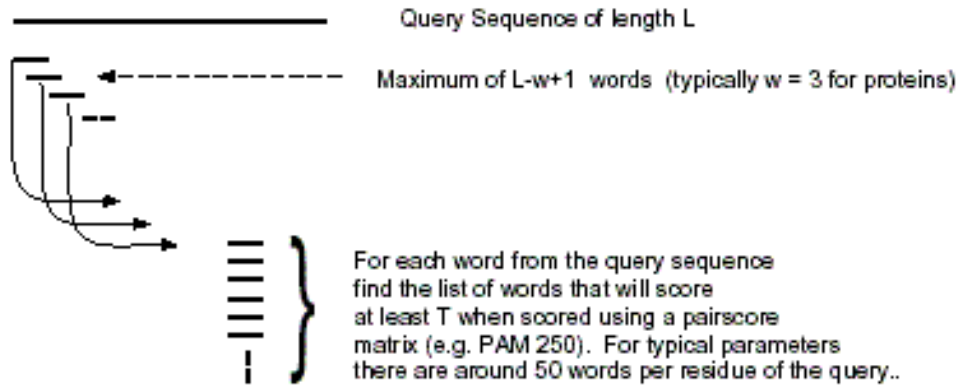


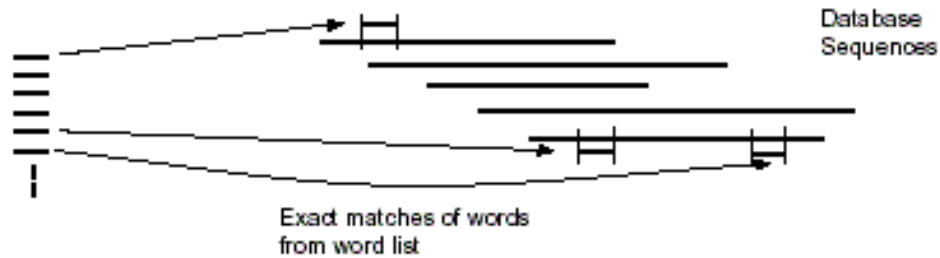
Figure 2.12: FastA Algorithm

BLAST Algorithm

(1) For the query find the list of high scoring words of length w .



(2) Compare the word list to the database and identify exact matches.



(3) For each word match, extend alignment in both directions to find alignments that score greater than score threshold S .

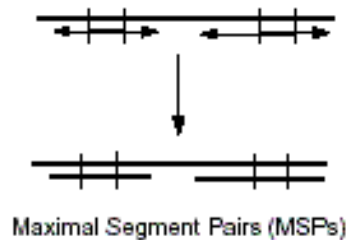


Figure 2.13: BLAST Algorithm

2.4.6 Variations of the BLAST Algorithm

- **PSI-BLAST (Position-specific iterated BLAST)**

Position-Specific Iterated BLAST (PSI-BLAST) provides an automated version of a “profile” search, which is a sensitive way to look for sequence homologies [AMS⁺97]. The program first performs a gapped BLAST database search. The PSI-BLAST program uses the information from any significant alignments returned to construct a position-specific score matrix, which replaces the query sequence for the next round of searching. PSI-BLAST may be iterated until no new significant alignments are found. PSI-BLAST is much more sensitive to weak but biologically relevant sequences.

- **MegaBLAST**

MegaBLAST [ZSWM00] is a greedy algorithm used for the DNA gapped sequence alignment search. MegaBLAST can only work with DNA sequences. To improve efficiency, MegaBLAST uses longer w -tuples (by default, $w = 28$). But, this reduces the sensitivity of the algorithm.

MegaBLAST takes as input a set of DNA query sequences. The algorithm concatenates all the query sequences together and performs search on the obtained long single sequence. After the search is done, the results are re-sorted by query sequence.

Unlike BLAST, MegaBLAST by default uses non-affine gap penalties. To set the affine penalties, advanced options should be used. It is not recommended to use the affine version of MegaBLAST with large databases or very long query sequences.

- **BLAT**

BLAT [KWJ02] stands for “BLAST-like alignment tool”. BLAT is similar in many ways to BLAST. The program rapidly scans for relatively short matches (hits), and extends these into high-scoring pairs (HSPs).

However, BLAT differs from BLAST in some significant ways as follows.

- While BLAST builds an index of the query sequence and then scans linearly through the database, BLAT builds an index of the database and then scans linearly through the query sequence.

- While BLAST triggers an extension when one or two hits occur in proximity to each other, BLAT can trigger extensions on any number of perfect or near-perfect hits.
- While BLAST returns each area of homology between two sequences as separate alignments, BLAT stitches them together into a larger alignment.
- BLAT has a special code to handle introns in RNA/DNA alignments. Therefore, whereas BLAST delivers a list of exons sorted by exon size, with alignments extending slightly beyond the edge of each exon, BLAT effectively “unsplices” mRNA onto the genome - giving a single alignment that uses each base of the mRNA only once, and which correctly positions splice sites.

By default, for DNA, BLAT uses the “two-hit requirement” and $w = 11$. It has been noted that BLAT is less sensitive than BLAST, but it is more sensitive than MegaBLAST.

2.5 PatternHunter

PatternHunter [MTL01] is a highly sensitive **Java program** which finds the homologies within one, or between two DNA sequences. PatternHunter can identify all approximate repeats in a complete genome, in a short time, using little memory on a desktop computer. Its features are its advanced patented algorithm, data structures and the usage of java language to create it. It uses a variety of advanced data structures including priority queues, a variation of red-black tree, queues and hash tables. It also uses a new method of sequence alignment. The Java language version of PatternHunter is just 40 KB, just 1% of the size of BLAST, while offering a large portion of its functionality.

In comparison with BLASTn algorithms (at their default settings), PatternHunter runs faster, on very long sequences, while being more sensitive than them. But PatternHunter is slower than MegaBLAST.

2.5.1 Concept

BLAST searches uses w consecutive letters as seeds and the default value for w is 11. PatternHunter is similar to BLAST. But, it uses non-consecutive w letters as seeds.

It has been found that gapped (non-consecutive) w -tuple can significantly increase hits to homologous region while reducing bad hits. In other words, it can increase

the sensitivity and reduce the number of random hits. For $w = 11$, the pattern 111010010100110111 is considered to be an optimal pattern. For example,

```

111010010100110111
ACTCCGATATGCGGTAAC
| | | - | - - | - | - - | | - | | |
ACTTCACTGTGAGGCAAC

```

Figure 2.14: Example of two substrings which are matched according to the model

This approach is called the **Spaced Seed approach**. Hence, using *novel seeding schemes* and *hit-processing methods*, sensitivity and speed are improved simultaneously.

2.5.2 Advantage of gapped w -tuple

- Improving the sensitivity

The reason for the increased sensitivity is that the events (of having a match at different positions) become more independent for spaced models. For example, as shown in the figures 2.15 and 2.16, two adjacent ungapped 11-tuples share 10 symbols where as, two adjacent gapped 11-tuples share just 5 symbols.

If a model and a shifted copy share many “1”s in the same position, then a base mismatch at any of these shared positions will make both matches fail, hence the corresponding matching events are far from independent. Independent events are better at pooling their success probabilities together. If the w -tuples are more independent, the probability of having at least one hit in a homologous region is higher. Figures 2.17 and 2.18 depicts the sensitivity of different models.

- Reducing the number of hits

Efficiency can be increased by decreasing the number of hits. The expected number of hits in a region can be easily calculated as shown in the Lemma 2.1.

```

1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 1 1

```

Figure 2.15: Two adjacent ungapped 11-tuples

```

1 1 1 0 1 0 0 1 0 1 0 0 1 1 0 1 1 1
  1 1 1 0 1 0 0 1 0 1 0 0 1 1 0 1 1 1
    
```

Figure 2.16: Two adjacent gapped 11-tuples

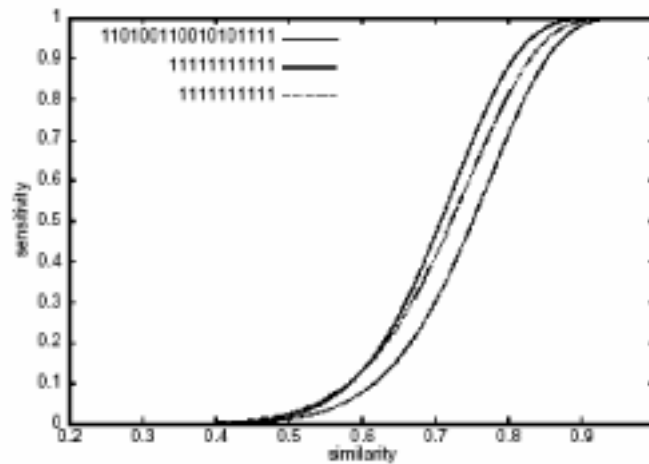


Figure 2.17: 1-hit performance of weight 11 spaced model vs weight 11 and weight 10 consecutive models

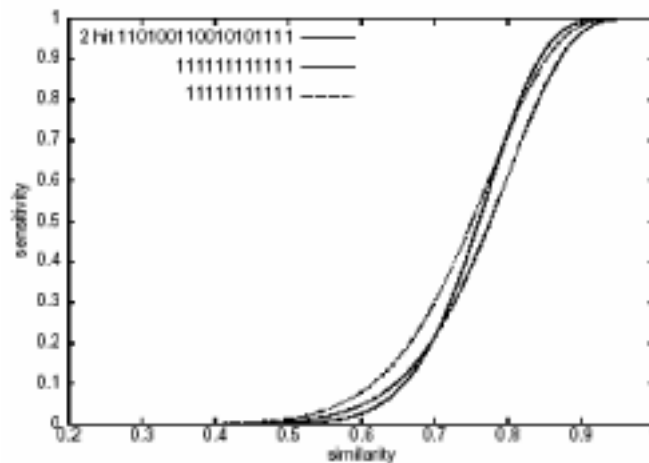


Figure 2.18: 2-hit performance of weight 11 spaced model vs single hit weight 11 and weight 12 consecutive models

Lemma 2.1 *The expected number of hits of a weight W length M seed model within a length L region with similarity $p(0 \leq p \leq 1)$, is $(L-M+1)p^w$.*

Proof: The expected number of hits is the sum, over the $(L-M+1)$ possi-

ble positions of fitting the model within the region, of the probability of W specific matches, the latter being p^w . ■

Example: In a region of length 64 with 0.7 similarity, PatternHunter has probability of 0.466 to get hits while BLAST has probability of 0.3 to get hits. So the probability of getting hits increases 50%. On the other hand, by above lemma, the expected number of hits in BLAST is 1.07, while the expected number of hits in PatternHunter is 0.93. So, the expected number of hits decreases by 14%.

2.6 Q-gram Alignment based on Suffix ARrays (QUASAR)

In 1999, Stefan Burkhardt, Andreas Crauser, Paolo Ferragina, Hans-Peter Lenhof, Éric Rivals and Martin Vingron proposed a new database searching algorithm called “Q-gram Alignment based on Suffix ARrays (QUASAR)”. QUASAR was designed to quickly detect sequences with strong similarity, especially if the searches are conducted on one database [BCF⁺99].

2.6.1 Algorithm

QUASAR is developed and implemented as an approximate matching algorithm for determining all sequences in a database that have a local similarity to a query sequence.

The approximate matching problem with k differences and window length (length of substring) w , could be formally defined as:

- Input: a database D , a query S , k , w
- Output: a set of (X, Y) where
 - X and Y are length- w substring in D and S , respectively
 - edit $dist(X, Y) \leq k$

A pair of substrings with the above properties is called an *approximate match*. The approximate matching problem is solved by reducing it to exact matching problem of short substrings of length q (called q -grams).

The approach is based on the following observation: *if two sequences have an edit distance below a certain bound, one can guarantee that they share a certain number of q -grams*. This observation allows us to design a filter that selects

candidate positions from the database where the query sequence possibly occurs with a high level of similarity [BCF⁺99].

The basic q -gram filtration method works as follows:

- First, find all matching q -grams between the pattern and the text. That is, find all pairs (i, j) such that the q -gram at position i in the pattern is identical to the q -gram at position j in the text. Such a pair is called a hit.
- Second, identify the text areas that have enough hits. These areas are passed to the verification phase. There are different ways of defining the text areas and counting the hits in them [JU91][HS94].

However, they all have the same threshold (significant number of q -grams). This number is given by the lemma 2.2.

Lemma 2.2 *Given two length- w sequences X and Y , if their edit distance $\leq k$, then they share at least t common q -grams (length- q substrings) where $t = w + 1 - (k + 1)q$.*

Proof: Suppose, X and Y have r differences and X has $(w + 1 - q)$ q -grams. Let X' be the string with the r differences annotated. Note that a q -gram in X overlaps with some of the r differences, if X and Y does not share that q -gram. For each difference, there are at most q q -grams that overlap with the difference. In total, rq q -grams overlap with the r differences. Thus, X and Y share $(w + 1 - q - rq)$ q -grams, which is larger than $t = w + 1 - (k + 1)q$. ■

The threshold given by the lemma 2.2 is tight in the sense that using any lower value might miss an occurrence. For example, strings ACAGCTTA and ACACCTTA have an edit distance of 1 and have $8 - 3(1 + 1) + 1 = 3$ common q -grams: ACA, CTT and TTA.

Lemma 2.2 gives a necessary and sufficient condition for a subsequence of D to be a candidate for an approximate match with $S[1..w]$. At least $t = w + 1 - (k + 1)q$ of the q -grams contained in $S[1..w]$ occur in a substring of D with length w . Substrings of D with this property are *potential approximate matches*.

2.6.1.1 Algorithm for finding potential approximate matches

Given, $X = S[i..i + w - 1]$ where $i = 1, 2, \dots$

- For every length- w substring Y in D , associate a counter with it and initialize it to zero

- For each q -gram Q in X
 - Find the *hitlist*, i.e., the list of positions in D so that Q occurs
 - Increment the counter for every length- w substring Y in D , which contains Q
- For every length- w substring Y in D with *counter* $> t$, X and Y are a *potential approximate match*. This will be checked with a sequence alignment algorithm.

2.6.2 Suffix Array as Index Data Structure

QUASAR uses an index data structure for all q -grams in D to direct the search for Q towards small portions of D without scanning the whole database. It uses a full-text indexing data structure so that it is not necessary to rebuild the index if Q is changed.

A *suffix array SA* for a database D is an array of length $|D|$ storing all the lexicographically ordered suffixes of D . Entry $SA[j]$ contains the text position where the j -th smallest suffix of D starts. Therefore SA requires storing exactly one pointer per text position as shown in Figure 2.19. The suffix array for D is constructed in the preprocessing step.

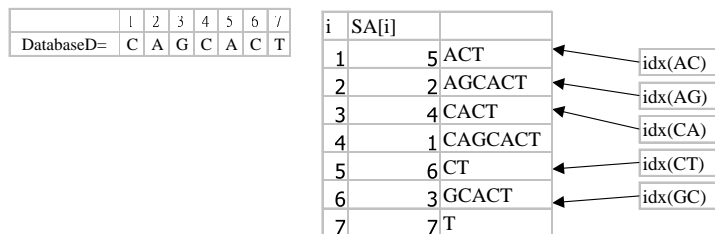


Figure 2.19: Suffix array SA of the database D .

Since, only the occurrences of q -grams is of interest, the positions of the hitlists in the suffix array SA for all possible q -grams could be pre-computed and stored in an auxiliary search array idx . This allows for finding the start position $idx[Q]$ of the hitlist for any given query q -gram Q in constant time.

2.6.3 Speedup Feature

2.6.3.1 Window shifting

The algorithm described in Section 2.6.2 builds the counter list for every $S[i..(w+i-1)]$, where $i = 1, 2, \dots, (n-w+1)$. This is a very time consuming process. Window shifting reduces this time complexity.

Suppose the counters list for the window $S[1..w]$ are given, in order to determine the approximate matches for the next window $S[2..(w+1)]$, only the “old” q -gram $S[1..q]$ and the “new” q -gram $S[(w-q+2)..(w+1)]$ needs to be considered as shown in Figure 2.20.

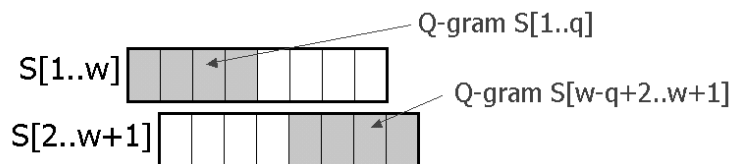


Figure 2.20: Windows Shifting

First decrement the counter values of all windows that contain the q -gram $S[1..q]$ and that have not reached the threshold t , i.e., if a counter for a window has already reached t , leave it unmodified. This marks all candidate windows already found. Then, use the suffix array to search for all occurrences of the “new” q -gram $S[(w-q+1)..(w+1)]$ and increment the corresponding window counters. Shift the window of length w over the string S until the end is reached.

2.6.3.2 Block Addressing

Block Addressing reduces the amount of space required to store counters used by Window shifting. The database D is conceptually divided into blocks of fixed size b ($b \geq 2w$). A counter is assigned to each block. This counter will be incremented whenever a search for a q -gram Q reports an occurrence inside the block. After processing all q -grams in $S[1..w]$, the counter of a certain block indicates how many q -grams from $S[1..w]$ are contained in this segment of the database. These counter values are stored in an array of size $|D|/b$. If a block contains more than t q -grams, this block has to be checked for approximate matches using a sequence alignment algorithm.

On the other hand, the candidates for approximate matches that cross block boundaries will be missed. In a worst case scenario, the occurrences of q -grams

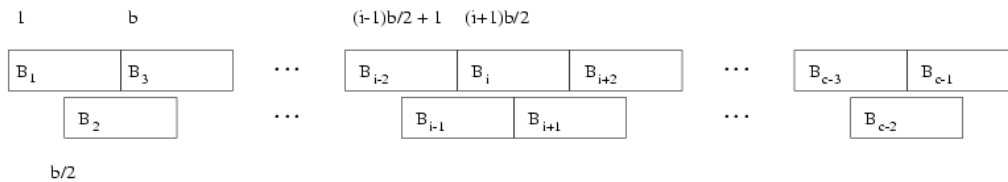


Figure 2.21: Partition of the database D into overlapping blocks of size b

from $S[1..w]$ are spread among two adjacent blocks and none of these block counters reaches the threshold t . In order to avoid this problem, a second block decomposition of the database, i.e., a second block array, is used. The second block decomposition is shifted by half the length of a block ($b/2$) as shown in Figure 2.21. Then, if a situation as described above occurs for blocks B_1 and B_3 , block B_2 contains the potential candidate.

2.6.4 Complexity

The preprocessing-step (the construction of the suffix array and the precomputation of the search array) can be done in $O(|D|\log|D|)$ time [MM93]. Searching for a specific q -gram requires constant time but the number of reported occurrences can be linear in $|D|$. As there are $O(|S|)$ q -grams, QUASAR approach takes $O(|S| \cdot |D|)$ time. If at the end c blocks reach the threshold t , the alignment with BLAST takes further $O(c \cdot b \cdot |S|)$ time. QUASAR has an extensive memory requirement. Its space complexity is dominated by the space used for the suffix array, which is $O(|D|\log|D|)$. More precisely, the algorithm requires $9|D|$ space in the construction phase and $5|D|$ space in the query phase.

2.7 Locality-Sensitive Hashing

LSH-ALL-PAIRS is designed to find ungapped local alignments in genomic sequence with up to a specified fraction of substitutions. The algorithm finds ungapped alignments efficiently using a randomized search technique called as locality-sensitive hashing (LSH). It is efficient and sensitive for finding local similarities with as little as 63 percent identity in mammalian genomic sequences up to tens of megabases [JB01].

2.7.1 Locality-Sensitive Hash Function

The LSH-ALL-PAIRS algorithm uses LSH to reduce the problem of string matching with substitutions to a more tractable exact matching problem.

To detect similarity between two strings, a randomized filter needs to be constructed. Consider a w -mer s , and choose k indices i_1, i_2, \dots, i_k uniformly at random from the set $\{1, 2, \dots, w\}$. It is assumed that the indices are sampled with replacement, so that an index can be chosen multiple times. A function $\pi(s) = (s[i_1], s[i_2], \dots, s[i_k])$ is defined. This function is called the *locality-sensitive hash function*.

Now, consider two w -mers s_1 and s_2 as shown in Figure 2.22. The more similar they are, higher will be the probability that $\pi(s_1) = \pi(s_2)$. More precisely, if the hamming distance of s_1 and s_2 equals d ,

$$\Pr[\pi(s_1) = \pi(s_2)] = \sum_{j=1, \dots, k} \Pr[s_1[i_j] = s_2[i_j]] = (1 - d/w)^k.$$

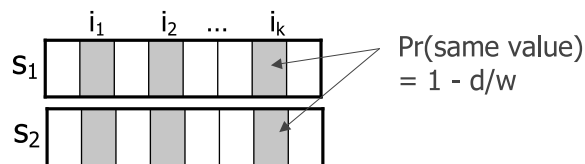


Figure 2.22: LSH Function: 2 w -mers s_1 and s_2 .

Hence, s_1 and s_2 are similar if $\pi(s_1) = \pi(s_2)$. However, there may be false positives and false negatives.

- False positive: s_1 and s_2 are dissimilar but $\pi(s_1) = \pi(s_2)$.
 - False positive can be distinguished from true positive by computing hamming distance between s_1 and s_2 .
- False negative: s_1 and s_2 are similar but $\pi(s_1) \neq \pi(s_2)$.
 - False negatives cannot be detected.
 - But, the number of false negatives could be reduced, by repeating the test using different $\pi()$ functions.

2.7.2 LSH-ALL-PAIRS Algorithm

There are 3 steps in this algorithm.

1. Generate m random locality-sensitive hash functions $\pi_1(), \pi_2(), \dots, \pi_m()$
2. For every w -mer s , compute $\pi_j(s)$ for $1 \leq j \leq m$

3. For every pair of w -mers s and t such that $\pi_j(s) = \pi_j(t)$ for some j , if $\text{hammingDist}(s, t) < d$, report (s, t) -pair.

LSH-ALL-PAIRS algorithm is sound as it outputs only similar pairs of w -mers. However, the algorithm is only guaranteed to find all pairs that match exactly. It will miss similar pairs that happen to be false negatives for every hash function chosen. The number of missed pairs can be controlled by performing more iterations or by allowing more w -mers to hash together in each iteration.

In the worst case, $O(N)$ w -mers might hash to the same LSH value, either because they are all pairwise similar or because the hash functions $\pi()$ yielded many false positives. The number of string comparisons performed can therefore in theory be as large as $O(N^2)$.

2.8 Conclusion

This lecture presents some database searching methods, like, Smith-Waterman algorithm, FastP, FastA, BLAST, PatternHunter, QUASAR and LSH. In fact, there are many other methods, such as, FLASH, RAMdb, FD, suffix tree, suffix array, compressed suffix array etc.

References

- [AGM⁺81] SMITH, T.F. and WATERMAN, M.S., "The identification of common molecular subsequences", *Journal of Molecular Biology*, 147, pp 195-197.
- [AGM⁺90] S.-F. ALTSCHUL, W. GISH, W. MILLER, E.-W. MEYERS and D.-J. LIPMAN, "Basic Local Alignment Search Tool", *Journal of Molecular Biology*, Vol. 215. 1990.
- [WL⁺84] W.J. WILBUR, DAVID J. LIPMAN, "The Context Dependent Comparison of Biological Sequence", *SIAM J. Applied Maths*, Vol. 44(3). 1984. pp 557-567.
- [LP⁺85] LIPMAN DAVID J., PEARSON WILLIAM R., "Rapid and Sensitive Protein Similarity Searches", *Science*, Vol. 227. 1985. pp 1435-1441.
- [LP⁺88] LIPMAN DAVID J., PEARSON WILLIAM R., "Improved tools for biological sequence comparison", *Proceedings of the National Academy of Science*, Vol. 85. 1988. pp 2444-2448.

- [AMS⁺97] S.-F. ALTSCHUL, T.-L. MADDEN, A.-A. SCHAFER, J. ZHANG, Z. ZHANG, W. MILLER and D.-J. LIPMAN, "Gapped BLAST and PSI-BLAST: a New Generation of Protein Database Search Programs", *Nuclear Acids Research*, Vol. 25. 1997.
- [BCF⁺99] S. BURKHARDT, A. CRAUSER, P. FERRAGINA, H.-P. LENHOF, E. RIVALS, and M. VINGRON, "Q-gram Based Database Searching Using a Suffix Array", In S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology (RECOMB-99)*, Lyon, France, ACM Press, 1999, pp. 77–83.
- [BK01] S. BURKHARDT and J. KARKKAINEN, "Better Filtering with Gapped q-Grams", *Combinatorial Pattern Matching (CPM2001)*, Jerusalem, Israel, 2001, pp. 73–85.
- [Buh01] J. BUHLER, "Efficient Large-Scale Sequence Comparison by Locality-Sensitive Hashing", *Appearing in Bioinformatics*, 17(5), 2001, pp. 419–428.
- [Gal00] F. GALISSON, "The Fasta and BLAST programs" , *Manuscript* , 2000.
- [HS94] N. HOLSTI and E. SUTINEN, "Approximate string matching using q-gram places", *In Proceedings of the 7th Finnish Symposium on Computer Science*, 1994, pp. 23–32.
- [JB01] P. JEREMY BUHLER "Efficient Large-Scale Sequence Comparison by Locality-Sensitive Hashing ", by *Department of Computer Science and Engineering University of Washington Seattle, WA 98195-2350, USA*
- [JU91] P. JOKINEN and E. UKKONEN, "Two algorithms for approximate string matching in static texts ", In A. Tarlecki, *Proceedings of the 16th Symposium on Mathematical Foundations of Computer Science*, number 520 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1991, pp. 240–248.
- [MM93] U. MANBER and E. W. MYERS, "Suffix Arrays. A new method for on-line string searches.", *SIAM Journal on Computing*, 22(5). 935–948, 1993.
- [Ukk92] E. UKKONEN, "Approximate string-matching with q-grams and maximal matches.", *Theoretical Computer Science*, 92(1).191-211, 1992.
- [ZSWM00] ZHANG Z, SCHWARTZ S, WAGNER L, MILLER W, "A greedy algorithm for aligning DNA sequences.", *Journal of Computational Biology*, 2000, 203-214.

[KWJ02] W. JAMES KENT, "BLAT-the BLAST-like alignment tool", *Genome Res*, 2002, 656-664.

[MTL01] B. MA, J. TROMP and M. LI, "PatternHunter: Faster And More Sensitive Homology Search." , *Bioinformatics* , 2001.

[WUBLAST] <http://blast.wustl.edu/>

[NCBIBLAST] <http://www.ncbi.nlm.nih.gov/BLAST/>