

## 3.1 Introduction

### 3.1.1 Biological Database

Biological database is the database of sequences. As we know, there are three different kinds of biological sequences: protein, DNA and RNA. In recent years biological data is doubled in size every 15 or 16 months. The number of everyday queries has also increased to 40,000 queries per day. So we should have some good database search methods. Otherwise, we cannot use the biological database efficiently.

### 3.1.2 How To Perform a Database-searching

Considering that there is a database  $D$  of genomic sequences and a query string  $Q$ , how can we look for a string  $S$  in  $D$  which is the closest match to the query string  $Q$ . There are two meanings for **close match**:

1.  $S$  and  $Q$  has a semi-global alignment (forgive the space on the two ends of  $Q$ ).
2.  $S$  and  $Q$  have a local alignment.

The main goal in searching is to find relevant information and to avoid irrelevant information. Formally its goodness is measured by:

1. **Sensitivity**: 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 measured as the probability of finding the matches such that the query and the database sequence has at least  $x$  percent similarity.
2. **Specificity**: The ability to reject '*false positive*' matches. The most specific search will return only true matches, but might have lots of '*false negatives*' i.e missed correct matches.

When one chooses which algorithm to use, there is a trade off between these two figures of merit. A good algorithm should be both sensitive and specific.

### 3.1.3 Different Algorithms

There are many biological search methods. The most basic methods are exhaustive methods. One such example is Smith-Waterman algorithm. In the real world, however, implementations of these methods take far too long to go through the large databases. In an attempt to gain speed with acceptable sacrifice of sensitivity, heuristic methods have derived from these algorithms. These algorithms, include FastA, BLAST and PatternHunter. And LSH and QUASTAR are filters and refined approaches.

## 3.2 Smith-Waterman Algorithm

### 3.2.1 Introduction

The **Smith-Waterman Algorithm** [AGM81] is the de facto standard for searching databases. Smith-Waterman searching method compares the query to each sequence in database using the full Smith-Waterman algorithm for pairwise comparisons. It also uses search results to generate statistics. It employs the **Dynamic Programming** (DP) algorithm between the query sequence and every database sequence, so it is the most sensitive method for finding all the closest matches in a database. But, because the DP algorithm effectively makes every possible pairwise comparison between the query (or reference) sequence and the library (or database) sequences, it is also the slowest method for similarity searches of sequence databases.

### 3.2.2 Algorithm

When given a database of total length  $n$  and the query  $Q$  of length  $m$ , Smith-Waterman algorithm will report all closest matches based on local alignment. This algorithm is described as follows:

1. For every sequence  $S$  in the database
  - Use Smith-Waterman algorithm to compute the best local alignment between  $S$  and  $Q$ .
2. Return all alignments with the best score

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

## 3.3 FASTA

### 3.3.1 What is FASTA?

FASTA [LP88] 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 each sequences in the database and return some good alignments. It is based on the assumption that good local alignment should have some exact match subsequences.

### 3.3.2 History of FASTA

In 1983, Wilbur-Lipman algorithm [WL84] was proposed for the analysis of protein and DNA sequence similarity that achieved a balance of sensitivity and selectivity on the one hand and speed and memory requirement on the other. In 1985, FastP [LP85] program was described for searching amino acid sequence databases, which uses a rapid technique for finding identities shared between two sequences and exploits the biological constraints on molecular evolution. Then in 1988, a new version of FastP, FastA [LP88] was developed, which uses an improved algorithm that increases sensitivity with a small loss of selectivity and a negligible decrease in speed.

### 3.3.3 FASTP

The search algorithm proceeds through four steps in determining a score for pairwise similarity.

#### Step1: Look for hot spots

For every k-tuple (length-k substring) of the query and every k-tuple of the database sequence, if they are the same, the pair is called a **hot spot**. This step looks for all hot spots. The parameter k determines how many consecutive identities are required in a match. The larger the value of k, the algorithm is faster but less sensitive. Usually, k= 4-6 for DNA sequence and k= 1-2 for protein sequence. See Figure 3.1.

#### Step2: 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 steps identifies

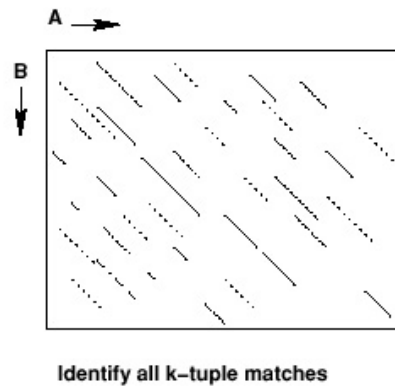


Figure 3.1: Hot spots appear as continuous diagonals on the Dynamic Programming Table

the 10 highest scoring diagonal runs for each database sequence.

### Step3: Rescore the 10 best diagonal runs for every database sequence

Rescore the 10 best diagonals 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. We will refer to this subregion as the “initial region” and it contains no gap. The best score among the 10 sub-regions is called the *init1* score for that sequence. See Figure 3.2.

### Step4: Rank the sequence

In the fourth step the sequences are ranked based on their *init1* scores.

## 3.3.4 FASTA

FastA uses the same first 3 steps of FastP. Then, it executes 2 more steps.

### Step4: Attempt to join the sub-regions by allowing indels

FASTA goes one step further during a database search; it checks to see whether several initial regions may be joined together. For the 10 sub-regions in Step 3,

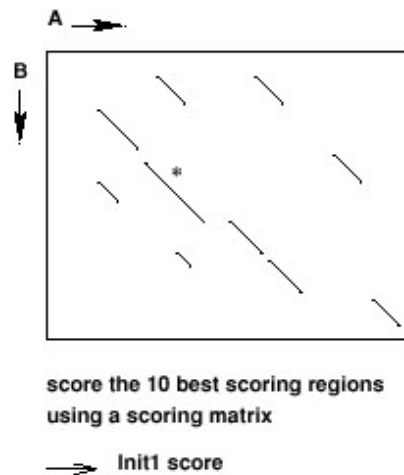


Figure 3.2: The best among these 10 subregions contribute to the init1 score

discard those with scores smaller than a given threshold. For the remaining subregions, attempts to join them by allowing insert and delete operations. Given the locations of the initial regions, their respective scores, and a “joining” penalty (analogous to a gap penalty), FASTA calculates an optimal alignment of initial regions as a combination of compatible regions with maximal score. The score of the combined regions is the sum of the scores of the sub-regions minus the penalty for gaps. Then FASTA uses the resulting score 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. See Figure 3.3.

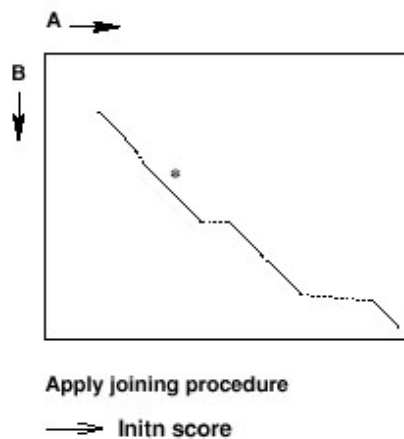


Figure 3.3: Join the gaps to get an initn score

### Step5: Banded Smith-Waterman DP

Sequences with *initns* smaller than a threshold are discarded. For the remaining sequences, apply banded Smith-Waterman dynamic programming to get the optimum alignment and score. Finally, rank the sequences based on their optimum scores. See Figure 3.4.

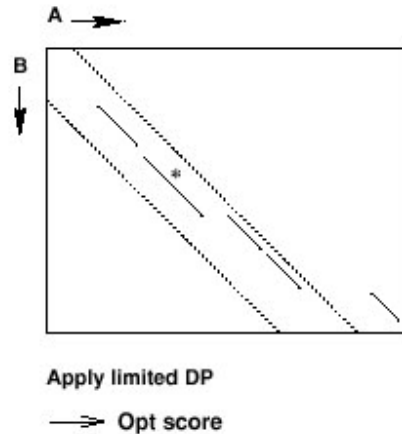


Figure 3.4: Apply banded Dynamic Programming to get the optimal score

## 3.4 BLAST

**Definition:** 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) would fall either by expanding or shortening the segment on either side. A *maximal segment pair* (MSP) in  $S_1, S_2$  is a segment pair with *maximum score* over all segment pairs in  $S_1, S_2$ .

### 3.4.1 What is BLAST?

BLAST stands for Basic Local Alignment Search Tool [Gal00]. The BLAST algorithm directly approximates alignments that optimised the maximal segment pair (MSP) score. This heuristic algorithm can be applied in a variety of contexts including straight forward DNA and protein sequence database searches, motif

searches, gene identification searches, and in the analysis of multiple regions of similarity in long DNA sequences.

### 3.4.2 History of BLAST

BLAST1 [AGM90] was created in 1990, which is very fast and dedicated to the search for regions of local similarity without gaps. BLAST2 [AMS97] was created as an extension of BLAST1, by allowing the insertion of gaps. Two versions of BLAST2 were independently developed, namely, NCBI-BLAST2 [NCBIBLAST] by National Center for Biotechnology Information in 1997 and WU-BLAST2 [WUBLAST] by Washington University in 1996.

### 3.4.3 Algorithm of BLAST1

BLAST1 is a heuristic method, which searches for local similarity without gap. 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. The algorithm can be divided into 3 steps:

- **Step1: Query preprocessing;**
- **Step2: Scan the database for hits;**
- **Step3: Extension of hits.**

#### Step1: Preprocessing of the query

In Figure 3.5, we see that for every position  $\mathbf{p}$  of the query sequence, 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  $\mathbf{p}$ . This list of words are called **neighbors**.

BLAST uses a word size of 3 for peptide sequences (protein); a word size of 11 for nucleic acid sequences (DNA). In the simplified example shown in Figure 3.6 and Figure 3.7, a word size of  $w = 2$  and a threshold of  $T = 8$  are used.

In the BLAST heuristic, this list of words (see Figure 3.6) is expanded in order to recover the sensitivity lost by matching only identical words. Any word that scores at least  $T$  when aligned with any of the initial list of words is added to the list. BLAST then examines the database sequences for words that exactly match any of the expanded list. The expanded list below consists of a total of 47 words derived from the initial 7. The expanded list contains any word that scores at least eight when aligned with the initial word and scored with the PAM

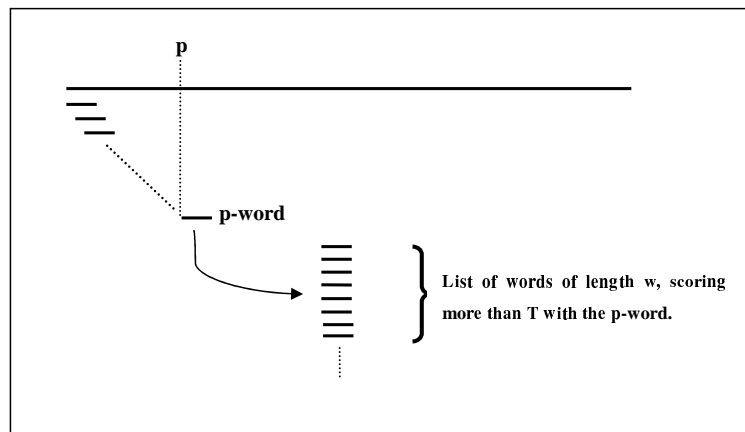


Figure 3.5: Preprocessing of the query

### 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 3.6: Creating an initial word list for a peptide sequence, word size = 2

120 similarity table.

**Definition:** 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 our context, PAM is used to refer to certain amino acid substitution matrices (scoring matrices), whose scores has a relationship to PAM units.

### Step2: Scan the database for hits

After preprocessing of the query, query is now represented by lists of neighbors, one list at each position of the query. Then we can scan the database DB to find whether there is an exact match between the neighbors of the  $w$ -tuple at **p** and a  $w$ -tuple in DB. If it does, a **hit** is made.(see Figure 3.8). All the possible hits between the query sequence and DB sequences are found in that way. A hit is characterized by the positions in both query and DB sequences.(see Figure 3.9).

**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 3.7: Generation of neighbours to create an expanded word list

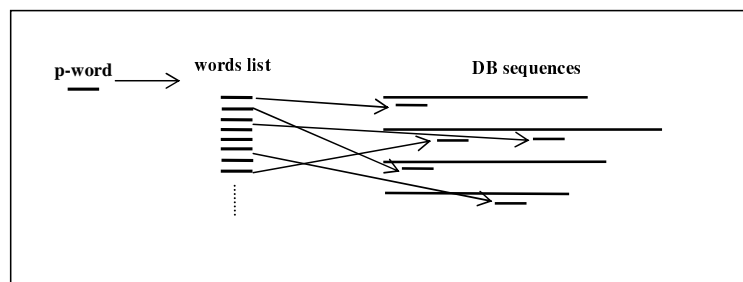


Figure 3.8: Generation of hits

**Step3: 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 now extended **in both directions, without gaps** (See Figure 3.9), but allowing mismatches.

To speed up this step, any extension is truncated as soon as the score decreases by more than X (the value chosen for X is a parameter of the program) from the highest score found so far for a shorter extension. 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 HSP). 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 program), it is called a **HSP** (High scoring Segment Pair). Then, they will be reported. For every sequence in the database, the best scoring HSP is the **MSP** (Maximal segment pair).

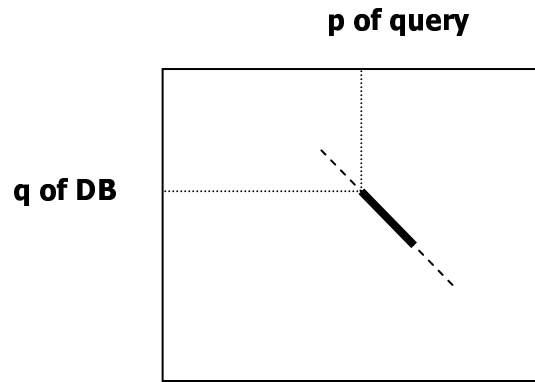


Figure 3.9: Extension of a hit

### 3.4.4 NCBI-BLAST2

NCBI-BLAST2 has been developed at NCBI (National Center for Biotechnology Information). 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 BLAST2 we only select some of these hits 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 program,  $A=40$  in DNA). This process is illustrated in Figure 3.10.

To achieve comparable sensitivity, **T** must be lowered, yielding more hits than previously. However, because a small fraction of these hits are extended, the average amount of computation required decreases.

- **Gapped extension**

BLAST2 performs an ungapped extension on all the hits satisfying the “two-hits requirement”. Among the generated HSP, we perform **gapped extension** for those segment pairs with score above some threshold: they are used as the starting points for performing dynamic programming local

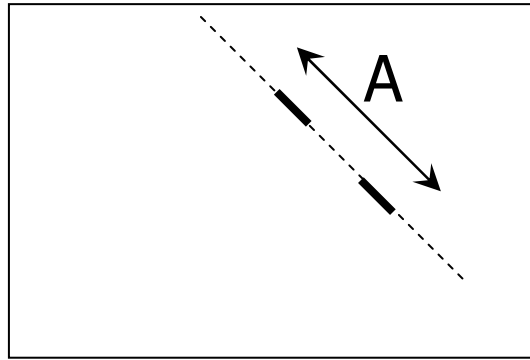


Figure 3.10: “two-hits” requirement

alignments.

The gapped extension algorithm allows gaps (deletions and insertions) 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 (see Figure 3.11(d)) starting from the middle point of the *combined hit*. In addition, when the alignment score drops off by more than  $Xg$ , the extension will be truncated.

### 3.4.5 BLAST1 VS. NCBI-BLAST2

BLAST1 spends 90 percent of its time on extension, this shows that the extension in step 3 is computationally expensive.

NCBI-BLAST2 is about 3 times faster than BLAST1. The authors attributed this speedup 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.

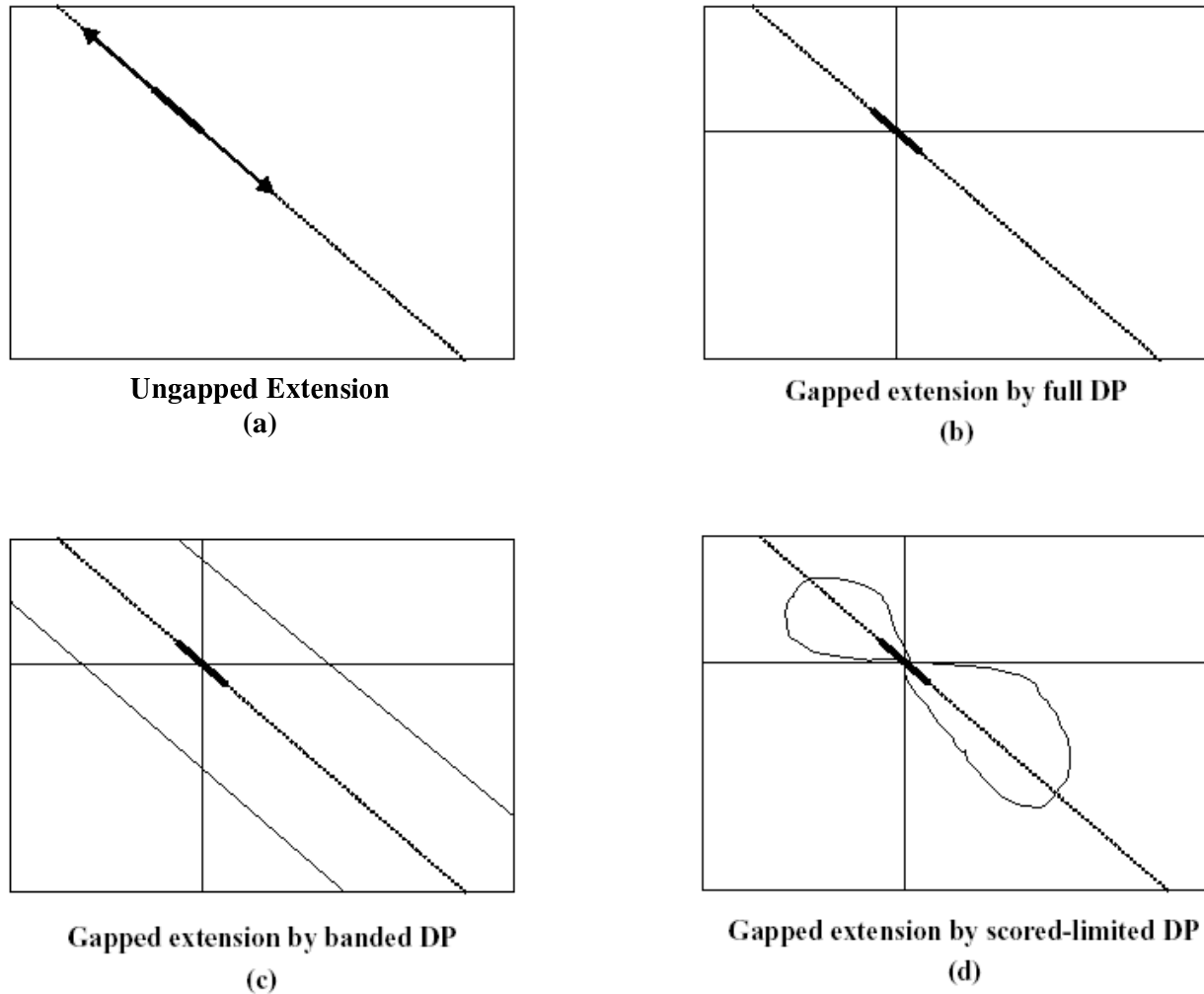


Figure 3.11: Gapped and ungapped extensions

### 3.4.6 BLAST VS. FASTA

BLAST is an order of magnitude faster than FASTA, at the expense of decrease in sensitivity. BLAST may be a bit less effective than FASTA in identifying important sequence matches, particularly when the most significant alignments contain spaces. Current comparisons between BLAST and FASTA showed a much smaller difference in speed (probably due to optimization in FASTA), although both remained significantly faster than running dynamic programming.

Figures 3.12 and 3.13 summarize the main ideas behind the algorithms of FASTA and BLAST respectively. Both the figures were obtained from <http://barton.ebi.ac.uk/new/publications/bookchapters.html>.

## FASTA Algorithm

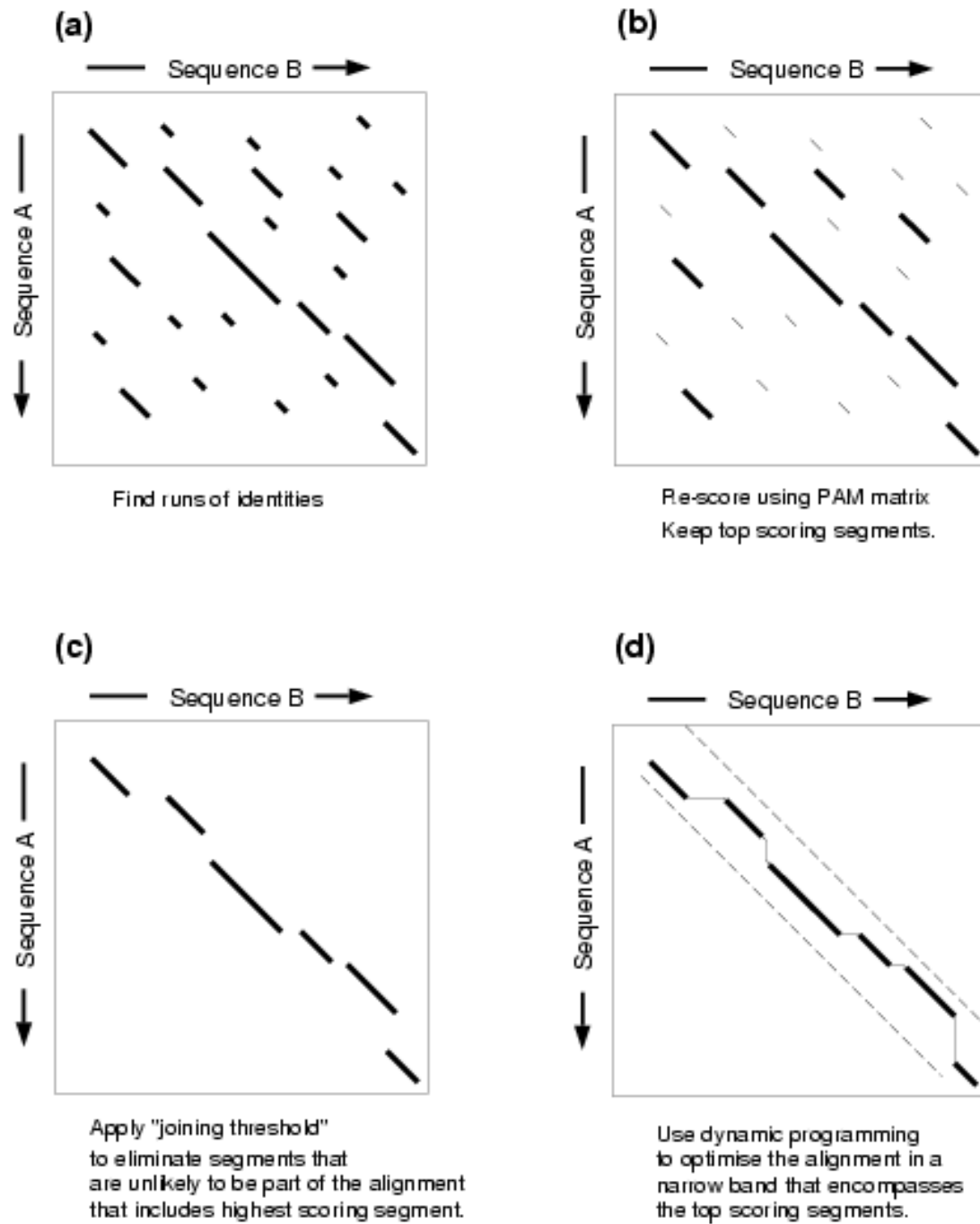
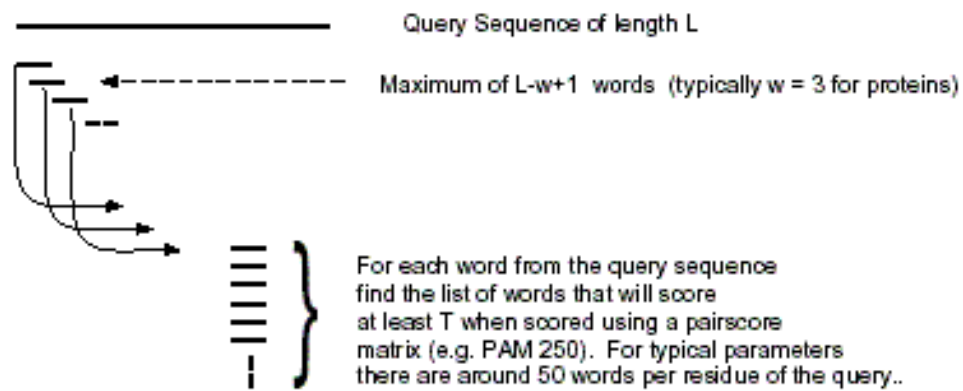


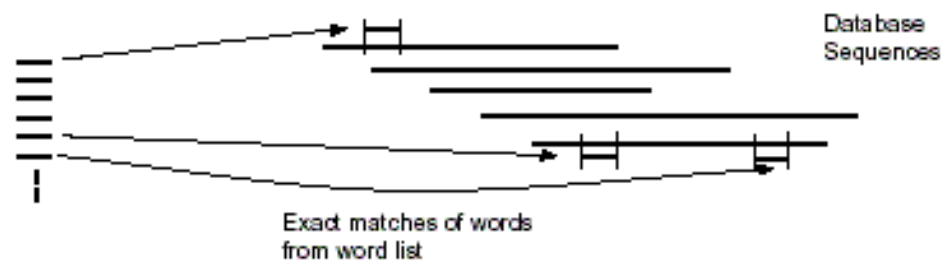
Figure 3.12: FASTA Algorithm

## 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  $S$ .

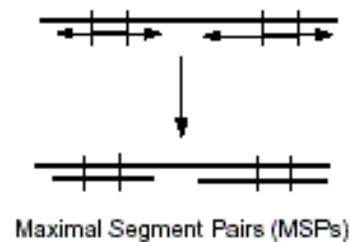


Figure 3.13: BLAST Algorithm

### 3.4.7 Variation of the BLAST

- **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<sup>+</sup>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 Search**

MegaBLAST [ZSWM00] implements a greedy algorithm for the DNA gapped sequence alignment search. And MegaBLAST can only work with DNA sequences. For DNA, in BLAST, set  $w=11$  by default. To improve efficiency, MegaBLAST uses longer  $w$ -tuples (by default,  $w=28$ ).

MegaBLAST takes as input a set of FASTA formatted DNA query sequences. These can be either pasted into a provided text area, or downloaded from a file. It is preferable to submit many query sequences at a time, but not more than 16383. 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. The database input for MegaBLAST is any “BLASTable” database, obtained from the ftp server or via format-db program from a FASTA formatted file.

Unlike BLAST, MegaBLAST is most efficient in both speed and memory requirements with non-affine gap penalties. These values of gapping parameters are default. 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. The cost of MegaBLAST is the reduction in sensitivity.

- **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.

- Where 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.
- Where 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.
- Where 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 two-hit and  $w = 11$ . It has been noted that BLAT is less sensitive than BLAST, but more sensitive than MegaBLAST.

## 3.5 PatternHunter

### 3.5.1 Introduction

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 and data structures, and the java language used to create it. The Java language version of PatternHunter is just 40 KB, only 1 percent the size of Blast, while offering a large portion of its functionality.

On very long sequences it runs faster than MegaBLAST while being more sensitive than BLASTn (at its default settings).

### 3.5.2 Concept

BLAST searches matches of  $W$  (default  $W = 11$  in BLAST and  $W = 28$  in MegaBLAST) consecutive letters as seeds. PatternHunter is similar to BLAST. Moreover, it uses non-consecutive  $W$  letters as seeds.

It has been found that gapped (non-consecutive)  $W$ -tuple can significantly increase hit to homologous region while reduce bad hits. In other words, it can increase the sensitivity and reduce the number of random hits. For  $W = 11$ , 111010010100110111 is considered to be the optimal model.

For example,

```

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

```

Figure 3.14: 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.

### 3.5.3 Advantage of gapped $W$ -tuple

- Advantage 1 — 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.

If a model and a shifted copy share many 1s 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. Figure 3.15 and 3.16 show the sensitivity of different models.

- Advantage 2 — Reduce 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 in the following Lemma.

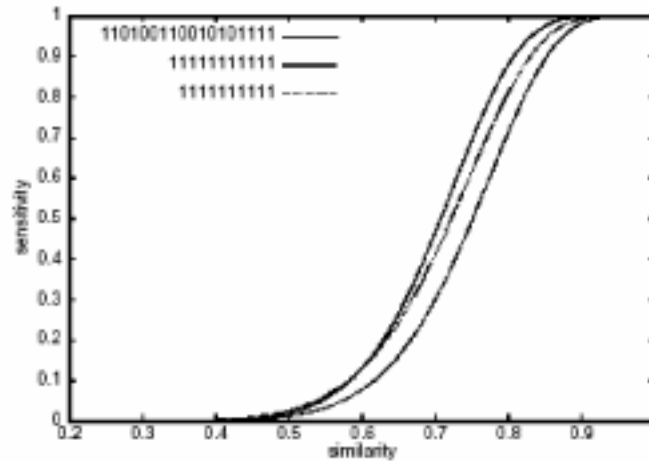


Figure 3.15: 1-hit performance of weight 11 spaced model versus weight 11 and weight 10 consecutive models

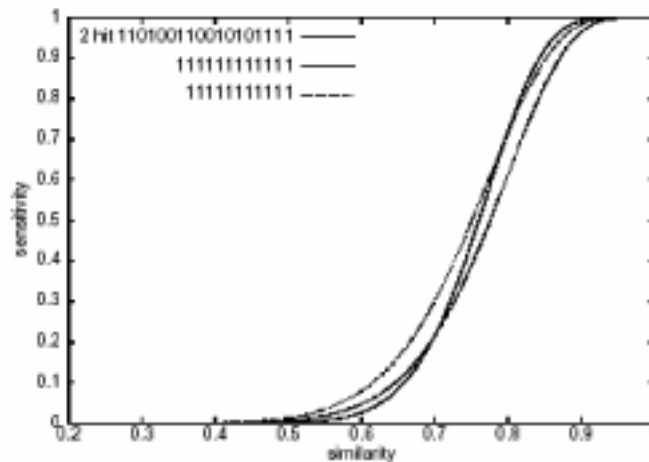


Figure 3.16: 2-hit performance of weight 11 spaced model versus single hit weight 11 and weight 12 consecutive models

**Lemma 3.1** *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  $(L-M+1)$  possible 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 14%.

## 3.6 QUASAR

There is an increasing amount of DNA and protein sequences deposited in public databases. In molecular biology, high-similarity search has become a basic operation for databases.

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

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. [JU91][HS94] Q-gram is a length q substring. 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<sup>+</sup>99]. The design of this filter plays a crucial point for QUASAR.

### 3.6.1 The 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 problem

(Approximate matching problem with  $k$  differences and window length (length of substring)  $w$ )

- 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*. To solve approximate matching, we reduce it to exact matching of short substrings of length  $q$  (called  $q$ -grams).

The  $q$ -gram is a substring of length  $q$ . The basic  $q$ -gram 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. We call such a pair 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 (see, e.g., [JU91][HS94]). However, they all have the same *threshold*, the significant number of  $q$ -grams. This number is given by the  $q$ -gram lemma.

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<sup>+</sup>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 (see, e.g., [JU91][HS94]).

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

### 3.6.2 The Algorithm

**Lemma 3.2** *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  $Y$  has  $r$  differences;

- $X$  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 difference if  $X$  and  $Y$  does not share that  $q$ -gram;
- For each difference, there are at most  $q$   $q$ -grams 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 is tight in the sense that using any lower value might miss an occurrence. For example, strings ACAGCTTA and ACACCTTA have edit distance 1 and have  $8 - 3(1+1) + 1 = 3$  common  $q$ -grams: ACA, CTT and TTA.

Lemma 3.2 gives a necessary 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*.

### Algorithm for finding potential approximate matches of $S$ in $D$

For  $X = S[i..i + w - 1]$  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*, that is, 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 an alignment algorithms.

### 3.6.3 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.

Use the suffix array: a *suffix array*  $SA$  for a database  $D$  is an array of length  $|D|$  storing the lexicographically order of all 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 (see Figure 3.17). The suffix array for  $D$  is constructed in a preprocessing step.

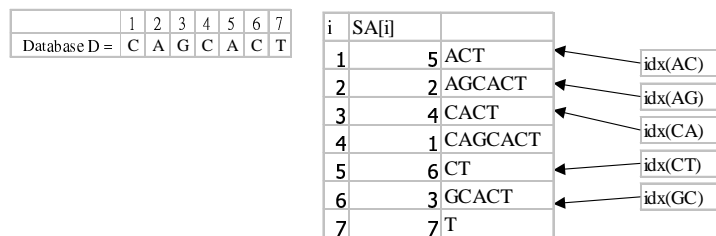


Figure 3.17: Suffix array  $SA$  of the database  $D$ .

We are only interested in the occurrences of  $q$ -grams, thus we may precompute the positions of the hitlists in the suffix array  $SA$  for all possible  $q$ -grams and store them in an auxiliary search array  $idx$ . This allows us to find the start position  $idx[Q]$  of the hitlist for any given query  $q$ -gram  $Q$  in constant time.

### 3.6.4 Speedup Feature

#### 3.6.4.1 Window shifting

The algorithm Suffix Array as Index Data Structure builds the counter list for every  $S[i..w + i - 1]$ , where  $i = 1, 2, \dots, n - w + 1$ . This is very time consuming. Window shifting reduce this time complexity. Given the counters list for  $S[i..w + i - 1]$ , can we determine the counters list for  $S[i + 1..w - i]$  easily?

Suppose the counters list for the window  $S[1..w]$  are given. In order to determine the approximate matches for the next window  $S[2..w+1]$ , we only have to consider the “old”  $q$ -gram  $S[1..q]$  and the “new”  $q$ -gram  $S[w - q + 2..w + 1]$  (see Figure 3.18).

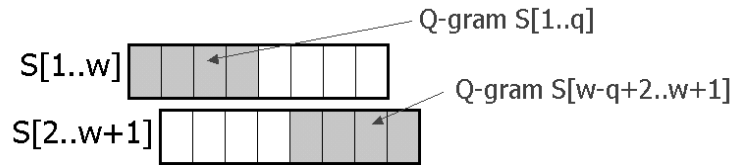
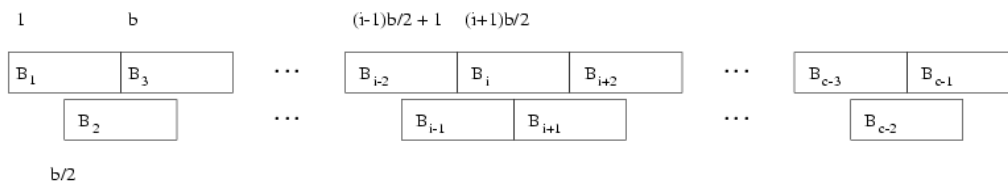


Figure 3.18: Windows Shifting Example.

First we 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. In this way we marks all candidate windows already found. Then, we 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.

### 3.6.4.2 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.

Figure 3.19: Partition of the database  $D$  into overlapping blocks of size  $b$ .

On the other hand, we will miss candidates for approximate matches that cross block boundaries. In a worst case scenario, the occurrences of  $q$ -grams 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, we use a second block decomposition of the database, i.e. a second block array. The second block decomposition is shifted by half the length of a block ( $b/2$ ) (see Figure 3.19). Then, if a situation as described above occurs for blocks  $B_1$  and  $B_3$ , block  $B_2$  contains the potential candidate.

### 3.6.5 Complexity

First, we analyze the time complexity of QUASAR. 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, our 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, in the construction phase, we need  $9|D|$  space. In the query phase, the algorithm consumes  $5|D|$  space.

## 3.7 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, locality-sensitive hashing. 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].

### 3.7.1 Locality-Sensitive Hash Function

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

To detect similarity between two strings, we first construct the following randomized filter. Consider an  $w$ -mer  $s$ , we choose  $k$  indices  $i_1, i_2, \dots, i_k$  uniformly at random from the set  $\{1, 2, \dots, w\}$ ; assume that the indices are sampled with replacement, so that an index can be chosen multiple times. Define the function  $\pi(s) = (s[i_1], s[i_2], \dots, s[i_k])$ . This function is called the *locality-sensitive hash*

function.

Now, consider two  $w$ -mers  $s_1$  and  $s_2$  (see Figure 3.20), the more similar are they, the higher probability that  $\pi(s_1) = \pi(s_2)$ . More precisely, if the hamming distance of  $s_1$  and  $s_2$  equals  $d$ ,

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

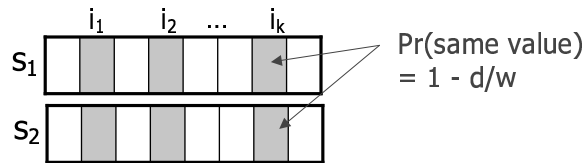


Figure 3.20: LSH Function: 2  $w$ -mers  $s_1$  and  $s_2$ .

Hence,  $s_1$  and  $s_2$  are similar if  $\pi(s_1) = \pi(s_2)$ . However, we may have false positive and false negative:

- 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  $\pi(s_1) \neq \pi(s_2)$ .
  - We cannot detect false negative.
  - We can only reduce the number of false negatives by repeating the test using different  $\pi()$  functions.

### 3.7.2 LSH-ALL-PAIRS Algorithm

The LSH-ALL-PAIRS algorithm is presented as the following steps,

1. Generate  $m$  random locality-sensitive hash functions  $\pi_1(), \pi_2(), \dots, \pi_m()$ ;
2. For every  $w$ -mers, 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 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  $\pi()$  yield many false positives. The number of string comparisons performed can therefore in theory be as large as  $O(N^2)$ .

### 3.8 Conclusion

This lecture presents some database searching methods, including FASTA, BLAST, PatternHunter, QUASAR and LSH. In fact, there are many other methods, such as CAF, FLASH, RAMdb, FD, suffix tree, suffix array, and compressed suffix array etc.

### References

- [AGM<sup>+</sup>81] SMITH, T.F. and WATERMAN, M.S., "The identification of common molecular subsequences", *Journal of Molecular Biology*, 147, pp 195-197.
- [AGM<sup>+</sup>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<sup>+</sup>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<sup>+</sup>85] LIPMAN DAVID J., PEARSON WILLIAM R., "Rapid and Sensitive Protein Similarity Searches", *Science*, Vol. 227. 1985. pp 1435-1441.
- [LP<sup>+</sup>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<sup>+</sup>97] S.-F. ALTSCHUL, T.-L. MADDEN, A.-A. SCHAFFER, 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<sup>+</sup>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/>"