CS5238 Combinatorial methods in bioinformatics 2006/2007 Semester 1

Lecture 3: Searching Biological Database - Sep 1, 2006

Lecturer: Wing-Kin Sung *Scribe: Ye Nan*

3.1 Introduction

3.1.1 Biological Database

In the last few years, there has been an exponential growth (doubling every 15 to 18 years) in the number of biological sequences (protein, DNA and RNA sequence). Often these sequences are submitted to public biological data databases like GenBank operated by the National Center for Biotechnology Information (NCBI). On August 22nd, 2005, it was announced that over 100 Gigabase pairs from over 165,000 organisms had been submitted to these public biological databases (Figure 3.1).

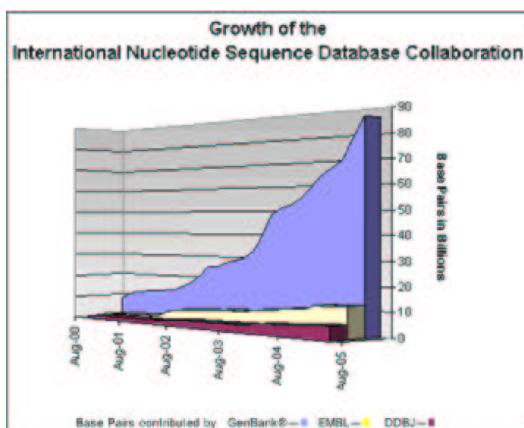


Figure 3.1: Growth in size of genomic databases over time

In molecular biology, searching for common sequences (and subsequences) is an operation which is commonly carried out on these databases, because sequence similarity often leads to functional similarity. For example, in 2002, GenBank alone served 100,000 search queries daily which had risen to over 140,000 queries daily by 2004(See [MM$^+$04]).

Considering the size of the biological databases to be searched and the large number of daily queries made to these databases, the need for time-efficient search algorithms is self-evident.

3.1.2 Database Searching

The Database searching problem can be defined as follows:

Given a database D of genomic (or protein) sequences and a query string Q, the goal is to find the string(s) S in D which is/are the closest match(es) using a given scoring function to the query string Q.

The term **closest match** can have the following meanings:

Semi-Global Alignment The spaces at the end of Q are ignored for the purposes of scoring.

Local Alignment Find an A which is a substring of S and a B which is a substring of Q such that A and B give the best possible score.

The main measures used to evaluate the effectiveness of a searching algorithm are *Sensitivity* and *Specificity* which are described below.

Sensitivity is the ratio of the number of true positives (substrings in the database matching the query sequence) found by the algorithm to the actual number of true mathes. It measures an algorithm's ability to find all true positives. An algorithm with 100% sensitivity finds all true positives.

Sensitivity can be viewed as the probability of finding a particular true positive. The higher/lower the sensitivity of the algorithm, the more/less likely that a given true positive will be found.

Specificity is the ability to reject *'false positives'*. Specificity is strongly related to the efficiency of an algorithm as the more false positives which can be rejected in an earlier stage, the less work which needs to be done to determine if a given match is indeed a true positive.

As such, a good search algorithm should be sensitive and at the same time specific.

3.1.3 Types of Algorithms

This Lecture focuses on search algorithms which solve the problem of finding the best *Local Alignment* using various algorithms.

Exhaustive Search Algorithms Exhaustive methods enumerate and test all possible solutions to find the best solution. Thus, exhaustive search algorithms are the most sensitive class of algorithms. But in practice, the number of possible solutions is very large, and due to time and space constraints , exhaustive algorithm is impractical. However, it is a useful benchmark

and test suite for other faster algorithms.

An example of an exhaustive search algorithm is the Smith-Waterman algorithm described in Section 3.2.

Heuristic Search Algorithms Heuristic Algorithms trade off sensitivity for speed. Instead of performing an exhaustive search, they follow various rules to look for the best possible matches.

A key point about heuristic algorithms is that unlike exhaustive search algorithms, there is no guarantee on the quality of the results returned. A heuristic which works well for a given problem instance could fail horribly in another situation. However, in practice, heuristics give acceptable results quickly and as such, most practical algorithms for database searching are heuristic algorithms.

Examples of such algorithms are: FastP, FastA(Section 3.3), BLAST(Section 3.4), BLAT, and PatternHunter(Section 3.5). A point of note about these heuristic algorithms is that they often need to solve the exact string matching problem to find the position of an exact match for a short string in a much larger string[1].

Filter based and other refined algorithms These algorithms use a filter to select candidate positions in the database where the query sequence possibly occurs with a high level of similarity.

Examples of such algorithms include QUASAR(Section 3.6) and LSH(Section 3.7). Note that unlike heuristic methods, QUASAR does not miss any solution while LSH is within a known margin of error(i.e. the probability that the program returns a false negative is within a known error threshold.)

Note that the above list is non-exhaustive and only describes some of the more commonly used algorithm approaches, There are also many other search methods.

3.2 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 discussed in previous lecture. Hence it is an exhaustive search algorithm,

[1] See `http://www-igm.univ-mlv.fr/~lecroq/string/index.html` for a description of various exact string matching algorithms

and is in fact the most sensitive known algorithm, but at the same time it is also the slowest and most space consuming algorithm presented in this lecture.Its time and space complexity are both O(nm) where n is the sum of the length of all sequences in the database and m is the length of the query string.

3.2.1 Algorithm

This algorithm can 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 score between S and Q for some substrings of S and Q.
- Return all alignments with the best score.

For more details on the Smith-Waterman Algorithm, refer to Lecture 2.

3.3 FastA

FastA [LP+88] is a heuristic search algorithm 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.

3.3.1 History of FastA

In 1983, the 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 one hand and speed on the other. In 1985, Wilbur and Lipman proposed the FastP [LP+85] algorithm for searching amino acid sequence databases. It identifies good alignments with no insert and delete operations (indels). Then in 1988, an improved version of FastP, called FastA [LP+88] was proposed, which allowed the identification of alignments with indels. It also increased the sensitivity with a small loss of specificity over FastP with no loss in performance.

3.3.2 FastP Algorithm

There are 4 steps in the FastP algorithm 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. A matching pair of query and database k-tuples is called a **hot spot**.

In this step, the FastP algorithm looks for all the hot spots as shown in Figure 3.2. The parameter k of the algorithm determines how many consecutive identities are required in a match. A larger k increases searching speed but decreases sensitivity. By Default,

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

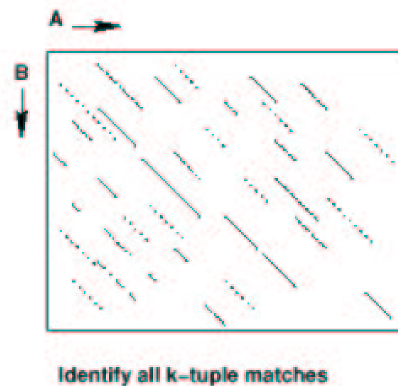


Figure 3.2: Hot spots appear as continuous diagonals on the Dynamic Programming Table

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

A **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 decreases with the length of the space. 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 the appropriate substitution matrix (such as **PAM** 120 matrix described in next section) for amino acid (or nucleotide). This substitution matrix is a scoring matrix that allows conservative replacements (no indels allowed) 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 3.3(b).

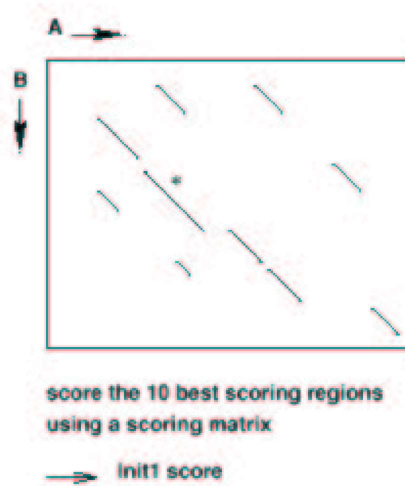


Figure 3.3: This diagram shows the 10 best scoring initial regions. The region marked with '*' has the highest score (initl score)

Step 4: Rank the sequence

The sequences are ranked based on their *init1* scores. At this point the FastP algorithm terminates.

3.3.3 FastA Algorithm

The FastA algorithm attempts to find alignments with gaps (caused by insertions and deletions). It uses the first 3 steps of the FastP algorithm. It then executes the following 2 steps. Figure 3.4 shows the flow of FASTA algorithm.

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

After the 10 best initial regions are found for each sequence in the database, we check whether several initial regions could be joined together.

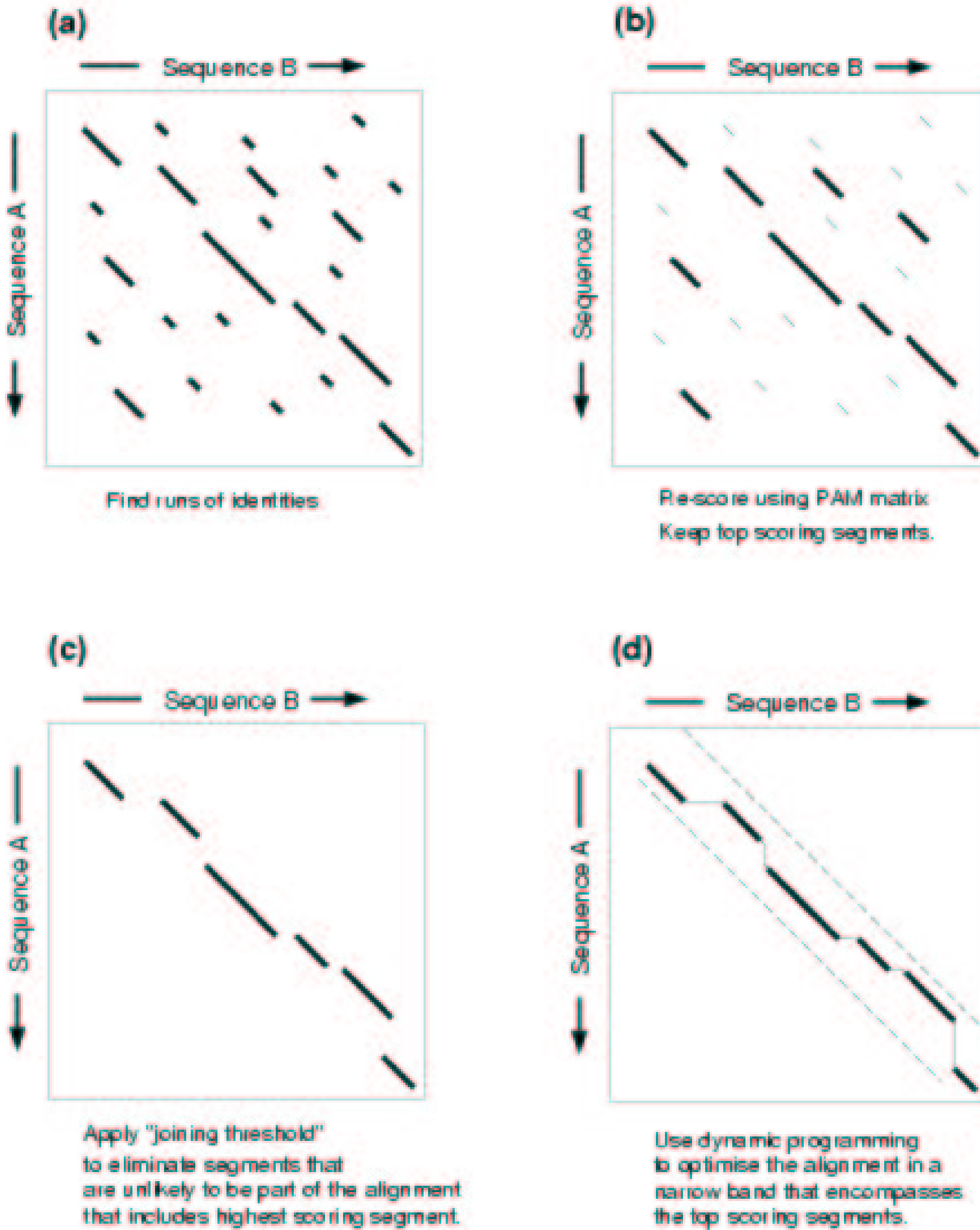


Figure 3.4: FastA Algorithm

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 3.4(c). 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 called the *initn* score for this sequence.

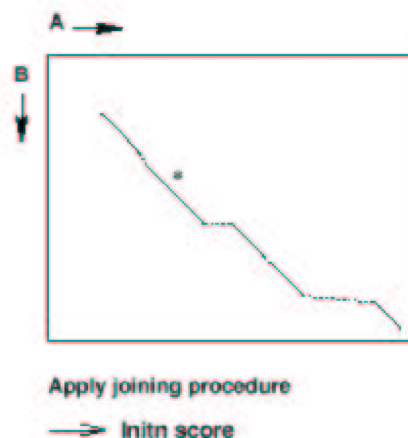


Figure 3.5: Join the gaps to get an *initn* score

Step 5: Banded Smith-Waterman Dynamic Programming

Sequences with *initn* scores smaller than a (user-defined) threshold are discarded. Then, for the remaining sequences, the banded Smith-Waterman dynamic programming algorithm (A more time and space efficient version of the standard algorithm) is applied to get the optimum alignment and score, as shown in Figure 3.4(d). Finally, the sequences are ranked based on their optimum scores.

3.4 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 alignment 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 extending 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.

BLAST is designed for speed and is faster than FastA. However there is a corresponding reduction in the sensitivity of BLAST compared to FastA. However the results returned by it are still good enough and as such, most public repositories like GenBank use BLAST (or one of it's variants, some of which are described in Section 3.4.6)

3.4.1 History of BLAST

BLAST1 [AGM+90] was first 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 (similar to the difference in capability between FastP and FastA). 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. NCBI-BLAST2 is the version which is more commonly used today.

3.4.2 BLAST1 Algorithm

BLAST1 is a heuristic method, which searches for local similarity *without gaps*. There are 3 steps in the BLAST1 algorithm: query preprocessing, database scanning and hit extension. One useful feature of BLAST1 is that it permits a tradeoff between speed and sensitivity, by setting a parameter called similarity threshold in query processing. A higher similarity threshold yields greater speed, but at the cost of increasing the probability of missing weak similarities.

Step 1: Query Preprocessing
In BLAST, the similarity of two length-w strings are computed by performing substitutions first and then adding up the scores of the substitutions based on a table called **PAM** (Point/Percent Accepted Mutation) 120 matrix[PAM120]. If the score is more than a similarity threshold T, then they are called **(w, T)-neighbors**. Note that PAM refers to an evolutionary event, and is called as

substitution too. (It is also used as a unit to measure the amount of evolutionary divergence between two amino acid sequences).

Example *Consider two amino acid sequences ql and qm, by checking PAM 120 matrix, q-q has a score of 6, and l-m has a score of 3, thus the similarity score is 9, and they are* $(2, 8)$*-neighbors. ql is its own* $(2, 8)$*-neighbor too, because ql and ql have a similarity score of 11.*

In query preprocessing, the values of w and T are set first, then for every w-tuple (length-w substring) of the query sequence s, generate all the (w, T)-neighbors for it. These neighbors will be used as queries in later processing.

BLAST uses a word size $w = 3$ for peptide sequences (protein) and $w = 11$ for nucleic acid sequences (DNA and RNA).

Step 2: Database scanning

In step 2, the database D is scanned to find exact matches for the neighbors found in step 1. Such a match is called a **hit**. A hit is characterized by its positions in both query and database sequences. (Note here that BLAST makes an assumption similar to the FASTA and FASTP algorithms that for the best match some substring of s is also a substring of D).

The authors of BLAST investigated 2 approaches for finding exact matches to the w-tuples in the database sequences: The Karp-Rabin string matching algorithm and the use of Deterministic Finite State Automatons (DFA) to look for exact string matches. They eventually settled on the DFA approach due to its better performance.

Step 3: Hit extension

To determine whether each hit may be part of a longer matching pair with higher score, every hit that has been found in Step 2 is extended **in both directions** by checking if the sequence beyond the ends of the hit in the database match the sequence beyond the hit in the query string allowing for mismatches (however indels are not considered in BLAST1) by assigning a penalty to them when they occur as shown in Figure 3.6.

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 score better than or equal to S (the cutoff score for a high scoring segment pair). In other words, BLAST like

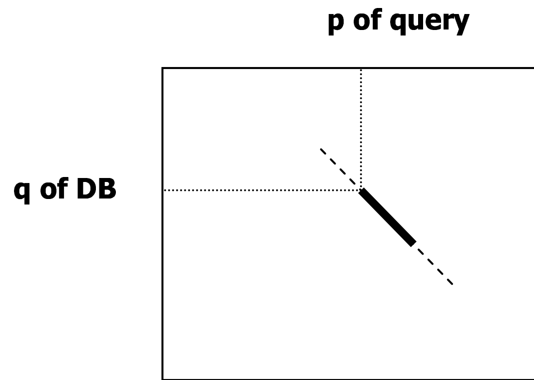


Figure 3.6: Extension of a hit

other heuristic methods may not return all the matching pairs which have a score greater than the threshold S.

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 a **High Scoring Segment Pair**(HSP). These HSPs are the results which will be returned to the user. For every sequence in the database, the best scoring HSPs are called the **Maximal Segment Pair**(MSP).

BLAST1 is summarized in figure 3.7.

3.4.3 NCBI-BLAST2

NCBI-BLAST2 was developed at National Center for Biotechnology Information (NCBI). The most important improvement of NCBI-BLAST2 over BLAST1 is that NCBI-BLAST2 allows the finding of local alignment with gaps (due to mutations which cause insertions or deletions to the sequence) similar to the evolution of FastP to FastA.

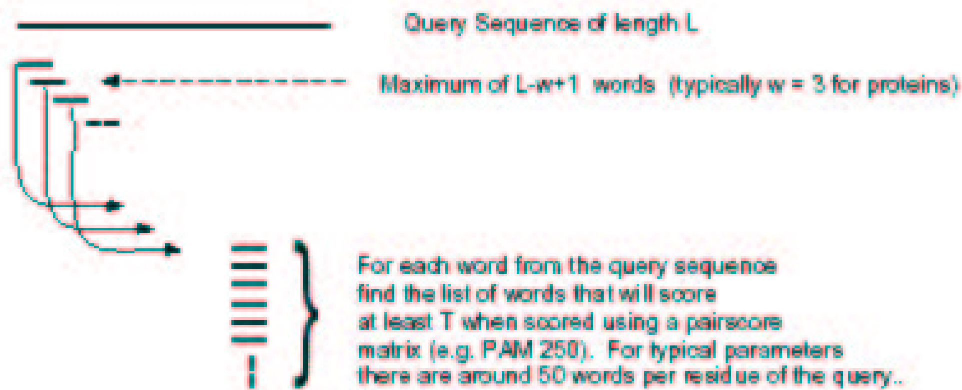
BLAST2 uses the same methods as BLAST1 to generate the initial list of hits to be considered (Step 1 of BLAST1).

The key difference between BLAST2 and BLAST1 is the third step which includes the following two major refinements:

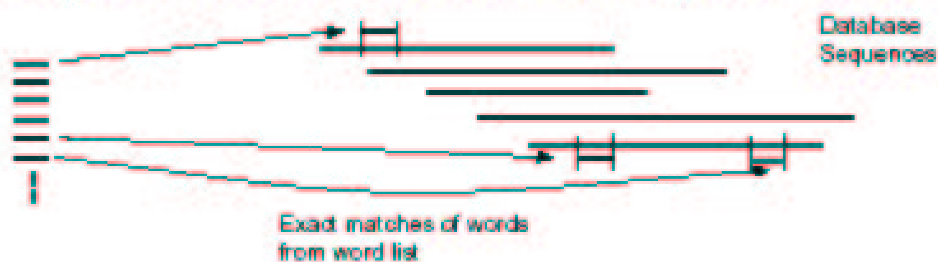
- **Two-hits requirement**

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.

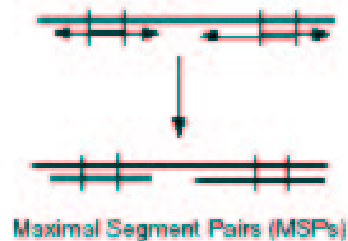


Figure 3.7: BLAST Algorithm

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 3.8. Since only a small fraction of hits can meet this requirement and undergo the gapped extension stage described next, the computation time of BLAST2 is reduced over the computation time of BLAST1.

The intuition behind this requirement is that in gapped extension carried out later, the matching subsequences with the best scores can be found in the areas where hits in the same diagonal are in close proximity with each other, so hits which do not meet the two-hit requirement are not likely to give the best possible match (note that this will cause a loss in sensitivity as there can be some high scoring matching subsequences which do not meet these requirements) and can thus be filtered out at this stage.

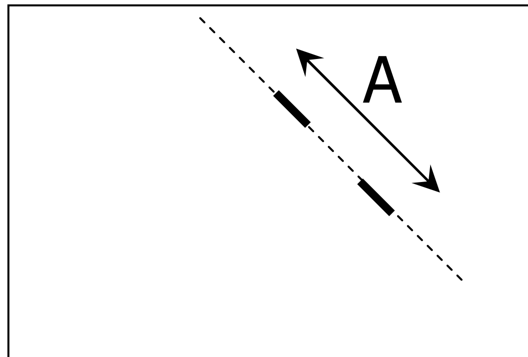


Figure 3.8: "two-hits" requirement

- **Gapped extension**
  NCBI-BLAST2 performs an ungapped extension (as in BLAST1) on all the hits satisfying the previously mentioned "two-hits requirement". Among the generated HSPs, **gapped extension** is performed for those segment pairs with score above some threshold; more precisely, using the subsequences found in ungapped extension as the starting points, the Smith-Waterman algorithm is applied for finding local alignment of 2 sequences.

  The gapped extension algorithm allows gaps (insertions and deletions) to be introduced into the alignments that are returned (from ungapped extension). Allowing gaps means that similar regions are not broken into several

segments. Thus, the scoring of these gapped alignments tends to reflect biological relationships more closely where mutations cause insertions and deletions in sequences and not just substitutions as in the case of BLAST1.

The best gapped local alignments are found via a modified Smith-Waterman algorithm. The Dynamic Programming matrix is explored in both directions (Refer to Figure 3.9(d)) starting from the mid-point of the *combined hit*. To reduce the time and space requirements for the Smith-Waterman algorithm, only the scores in the area denoted by the "bubbles" in Figure 3.9(d) are calculated. Note the contrast in the amount of processing needed as compared to using the banded Smith-Waterman algorithm as used in FastA as seen in Figure 3.9(c)

In addition, when the alignment score drops off by more than X_g(given as a parameter to the program), the extension will be truncated, further improving running time.

3.4.4 BLAST1 vs NCBI-BLAST2

BLAST1 spends 90 percent of its time on extension of the hits (Step 3) which is the most computationally intensive step of BLAST1.

NCBI-BLAST2 is about 3 times faster than BLAST1. This speedup is attributed to the "two-hit requirement", which reduces the number of extensions which must be performed significantly. The "two-hit requirement" is formulated based upon the observation that a HSP of interest is much longer than a single word pair (matching w-tuples), and may therefore entail multiple hits on the same diagonal within relatively short distances of each another.

Generally, with NCBI-BLAST2 the "two-hit requirement" is only used when searching for peptide sequences (protein) and not used with DNA sequences. Hence, for DNA sequence searching, NCBI-BLAST2 is slower than BLAST1. However, the increase in sensitivity of BLAST2 due to it's ability to detect local alignments with gaps makes the speed tradeoff worthwhile.

3.4.5 BLAST vs FastA

BLAST is an order of magnitude faster than FastA, it however achieves this speed at the expense of similarity. BLAST is less effective than FastA in identifying important sequence matches with lower similarities, particularly when the most significant alignments contain spaces. Both BLAST and FastA are significantly faster than Smith-Waterman algorithm. Figures 3.4 and 3.7 summarizes the

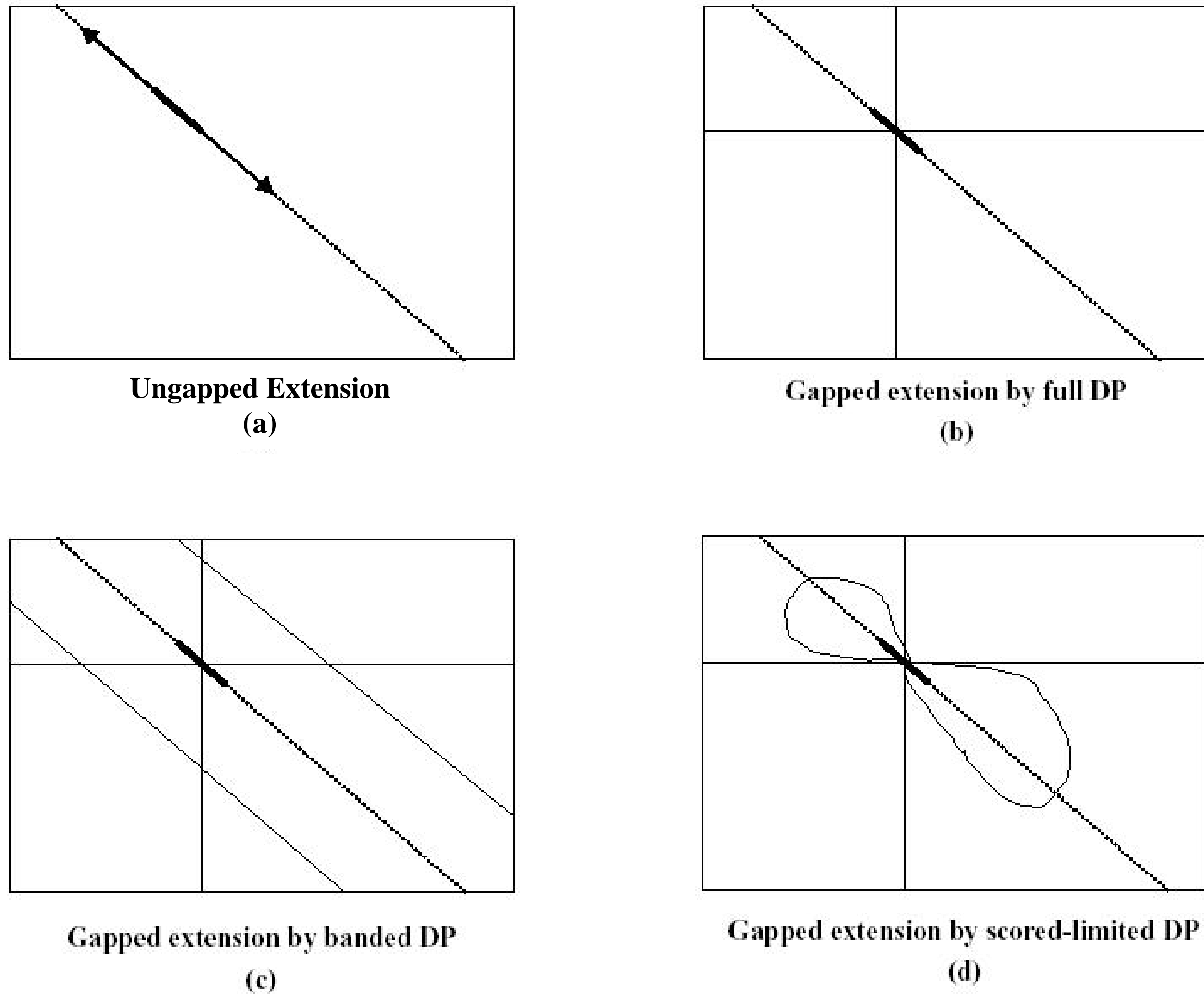


Figure 3.9: Gapped and ungapped extensions

main ideas behind FastA and BLAST algorithms.

In practice, for most applications, the decreased sensitivity of BLAST over FastA is not an issue while the increased speed is (due to the number of queries made daily), so most biological databases use BLAST (and it's variants) for finding local alignment.

3.4.6 Variations of the BLAST Algorithm

As BLAST is still among the best heuristics (in terms of striking an acceptable balance between speed and sensitivity) available for the problem of finding lo-

cal alignments between sequence in large biological sequences, is it natural that researchers have extended the original BLAST algorithm to deal with various specialized situations (In particular making sensitivity vs runtime tradeoffs). Some of the interesting variants are described below.

- **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 (but is slower than standard BLAST).

- **MegaBLAST**

  MegaBLAST [ZSWM00] is a greedy algorithm used for the DNA gapped sequence alignment search. MegaBLAST is designed to work only with DNA sequences. To improve efficiency, MegaBLAST uses longer w-tuples (by default, $w = 28$). But, this reduces the sensitivity of the algorithm. As such MegaBLAST is used to search larger databases where standard BLAST would be too slow.

  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 resorted 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), then 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 (which is stored in main memory) 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. That is, BLAT is able to extend 3 or more hits which are in close proximity to join them together[2].
- 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, unlike BLAST which only delivers a list of exons, 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 much faster, in fact in some cases it's claimed to be over 100 times faster), but it is more sensitive than MegaBLAST.

3.5 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. 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 (See Section 3.5.1). 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, PatternHunter can achieve a speed similar to MegaBLAST with an accuracy similar to standard BLASTn (at the default BLASTn settings). Despite this speed and sensitivity advantage, the adaptation of PatternHunter over BLAST is slow as PatternHunter is a commercial product..

[2]See `http://www.soe.ucsc.edu/~kent/intronerator/algo.html#cDNA%20Alignment` for more details

3.5.1 Concept

BLAST uses w-tuples as seeds(short subsequences of the query which match a subsequence in the database and which are used to build a longer match) and the default value for w is 11. PatternHunter is somewhat similar to BLAST except that it uses gapped w-tuples (see Figure 3.10) as seeds(This approach is called the **Spaced Seed approach**).
Below, we show that using gapped (non-consecutive) w-tuples can significantly increase hits to true homologous regions and reduce false matches as compared to using ungapped w-tuples (See Section 3.5.2). In other words, it can increase both sensitivity and specificity at the same time. For $w = 11$, the pattern 111010010100110111 is considered to be an optimal pattern. For example,

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

Figure 3.10: Example of two substrings which are matched according to the model

3.5.2 Advantage of gapped w-tuple

- Improving the sensitivity

  The reason for the increased sensitivity is that the event of having a match at different positions becomes more independent for spaced models compared to unspaced models. For example, as shown in the Figures 3.11 and 3.12, 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. If the w-tuples are more independent, the probability of having at least one hit in a homologous region of interest is higher. Figures 3.13 and 3.14 depicts the sensitivity of different models.

```
1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 1 1
```

Figure 3.11: 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 3.12: Two adjacent gapped 11-tuples

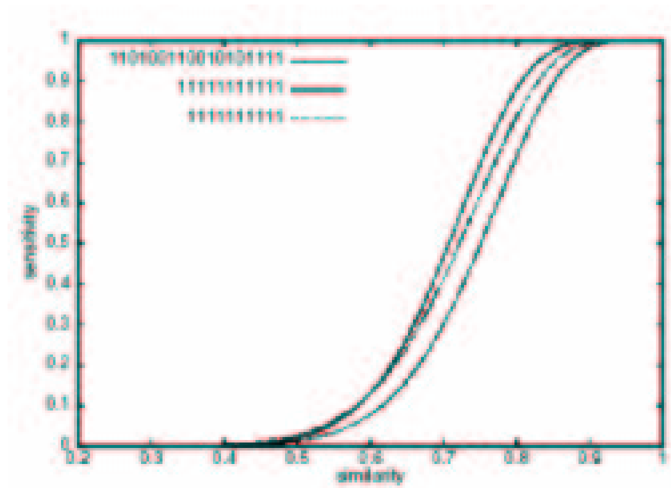


Figure 3.13: 1-hit performance of weight 11 spaced model vs weight 11 and weight 10 consecutive models

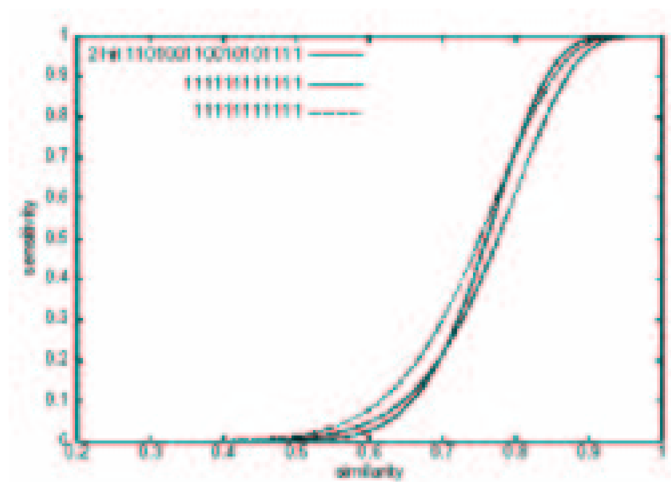


Figure 3.14: 2-hit performance of weight 11 spaced model vs single hit weight 11 and weight 12 consecutive models

- 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 Lemma 3.1.

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 by 14%.

3.5.3 Recent Developments

While the commercial nature of Patternhunter has meant that is has not attracted as much attention from researchers (both end users and algorithm developers) as BLAST, research in the use of gapped w-tuples is still ongoing and a newer version of PatternHunter called PatternHunter II [MTL03] has been developed by the original developers which attempts to reach a sensitivity similar to that of the Smith-Waterman algorithm while matching the speed of BLAST. The main difference between the original PatternHunter and PatternHunter II is the use of several optimal spaced seeds in PatternHunter II which allows PatternHunter II to achieve a higher sensitivity. This approach was not implemented in the original PatternHunter due to memory constraints and the difficulty of finding an optimal gapped seed.

3.6 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]. Note that QUASAR can only detect ungapped local alignments.

3.6.1 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, the difference k, and the window w
- Output: a set of (X, Y) where
  - X and Y are length-w substrings in D and S.
  - $\text{edit_dist}(X, Y) \leq k$

A pair of substrings X and Y 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, all these methods use the same threshold (significant number of q-grams) value which is given by the lemma below:

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: Let (X',Y') be an optimal alignment of X and Y, r be the number of differences between X',Y'(r≤k), L be the length of X' and Y' (L ≥ w). Consider the L+1-q pairs of q-grams of X', Y' starting at the same position. Each difference in X', Y' can make at most q such pairs to be different, thus X', Y' must have at least L+1-q-rq ≥ w+1-(k+1)q common q-grams. Last, note that any common q-gram for X', Y' is also common for X, Y. This completes the proof. ∎

The threshold given by lemma 3.2 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 3.2 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*.

3.6.1.1 Algorithm for finding potential approximate matches

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

- 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.

3.6.2 Suffix Array as Index Data Structure

QUASAR uses an indexed data structure(Suffix Array) which stores the location for all q-grams in D to direct the search for Q towards small portions of D without scanning the whole database. The entire database is indexed. Thus it is unnecessary to rebuild the index for different queries.

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 3.15. The suffix array for D is constructed in the preprocessing step.

	1	2	3	4	5	6	7
Database D =	C	A	G	C	A	C	T

i	SA[i]		
1	5	ACT	idx(AC)
2	2	AGCACT	idx(AG)
3	4	CACT	idx(CA)
4	1	CAGCACT	
5	6	CT	idx(CT)
6	3	GCACT	idx(GC)
7	7	T	

Figure 3.15: 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.

3.6.3 Speeding up and Reducing the Space for QUASAR

3.6.3.1 Window Shifting

The algorithm described in Section 3.6.1.1 builds the counter list for every $S[i..(w+i-1)]$, where $i = 1, 2, ..., (n-w+1)$ which is a very time consuming process. Window shifting reduces this time complexity.

Suppose the counters 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 need to consider the "old" q-gram $S[1..q]$ and the "new" q-gram $S[(w-q+2)..(w+1)]$ as shown in Figure 3.16.

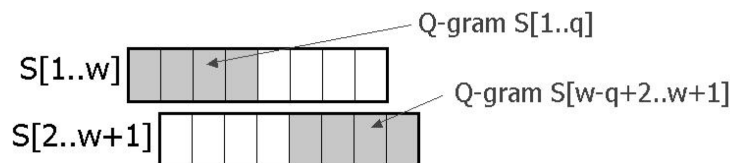


Figure 3.16: Windows Shifting

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.

3.6.3.2 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.

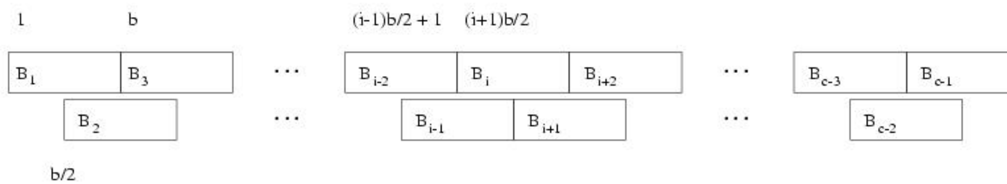


Figure 3.17: Partition of the database D into overlapping blocks of size b

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 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 3.17. Then, if a situation as described above occurs for blocks B_1 and B_3, block B_2 contains the potential candidate solving this issue.

Intuitively, no subblock can have more q-grams then the whole block and thus only the subblocks which can have a counter greater than the threshold t are investigated.

3.6.4 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 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.

3.7 Locality-Sensitive Hashing

LSH-ALL-PAIRS is a randomized search algorithm 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 enough to find local similarities with as little as 63 percent identity in mammalian genomic sequences with tens of megabases [JB01].

3.7.1 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 is constructed. Consider a w-mer (length w string) s, and choose k indices $i_1, i_2, ..., i_k$ uniformly at random from the set $\{1, 2, ..., w\}$. It is assumed that the indices are sampled with replacement, so that an index can be chosen multiple times. A function $\pi(s) = (s[i_1], s[i_2], ..., s[i_k])$ 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 3.18. 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,

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

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 $\pi(s_1) = \pi(s_2)$.

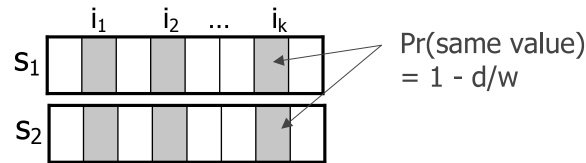


Figure 3.18: LSH Function: 2 w-mers s_1 and s_2.

  - False positive can be distinguished from true positive by computing the actual hamming distance between s_1 and s_2 (rather than the hamming distance between $\pi(s_1)$ and $\pi(s_2)$.).

- False negative: s_1 and s_2 are similar but $\pi(s_1) \neq \pi(s_2)$.
  - False negatives cannot be detected.
  - However, the number of false negatives can be reduced by repeating the test using different $\pi()$ functions.

3.7.2 LSH-ALL-PAIRS Algorithm

There are 3 steps in this algorithm.

1. Generate m random locality-sensitive hash functions $\pi_1(), \pi_2(), ..., \pi_m()$
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 $hammingDist(s,t) < d$, 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.

Thus, it can be seen that LSH-ALL-PAIRS is a Monte Carlo randomized algorithm which may not return correct results (in the sense that there may be false negatives), but the results returned fall within a bounded error probability (which can be derived from the number of hash functions used). This is in contrast to the heuristic algorithms like FastA and BLAST presented earlier where there is no guarantee that the returned answer is the correct answer (also due to the fact that false negatives may not be detected), but also, no guarantee can be made that the returned results fall within a bounded error probability.

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 $O(N^2)$.

3.8 Conclusion

This lecture presents some database searching algorithms. For example: The Smith-Waterman algorithm, FastA, BLAST, PatternHunter, QUASAR and LSH.

However the algorithms covered today only touch the tip of the iceberg. The problem of finding local alignments among sequences in a database is still an active field of research, resulting in various other algorithms being developed. For example: FLASH, RAMdb, FD, suffix tree, suffix array, compressed suffix array etc.

References

[MM+04] Scott McGinnis and Thomas L. Madden, "BLAST: at the core of a powerful and diverse set of sequence analysis tools", *Nucleic Acids Research*, Vol. 32. 2004, pg W20-W25

[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. 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+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.

[MTL03] Ming Li, Bin Ma, Derek Kisman and John Tromp, “PatternHunter II: Highly Sensitive and Fast Homology Search”, *Genome Informatics*, Vol. 14. 2003, pg 164-175

[WUBLAST] `http://blast.wustl.edu/`

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

[PAM120] `http://eta.embl-heidelberg.de:8000/misc/mat/pam120.html`