CS5238 Combinatorial methods in bioinformatics 2002/2003 Semester 1

Lecture 3: Searching biological database -August 26, 2002

Lecturer: Wing-Kin Sung *Scribe: Chen Jinxiu, LanMan, Ren Li-An, FengYuan, LiQuan*

3.1 INTRODUCTION

3.1.1 Biological Database

Biological database is the database of sequence. As we know,three kinds of biological sequences include protein, DNA and RNA. In recent years biological data is doubled in size every 15 or 16 months. Since there are so many data in biology, biology database has greatly developed and became a part of the biologist's everyday toolbox. 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 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 positive' matchs.The most sensitive search finds all true matches, but might have lots of 'false positives'.Sensitivity can be measured as the probability of finding the match given the query and the database sequence has only 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'.

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 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 filter and refine approaches. Other approaches include CAFE, FD, RAMdb, FLASH, suffix tree, etc.

3.1.4 Smith-Waterman Algorithm

Smith-Waterman algorithm is an exhaustive method. 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

As we learned earlier, the time complexity of Smith-Waterman algorithm is $O(nm)$.It is a brute force algorithm. So, it is the most sensitive algorithm. But it is not practical.

3.2 FASTA

3.2.1 What is FASTA?

Given a database and a query, FastA does local alignment with all sequences in the database and return some good alignments. FastA is a heuristic algorithm which is based on assumption is that good local alignment should have some exact match subsequences.

3.2.2 History of FASTA

In 1983, Wilbur-Lipman algorithm 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 program was described for searching amino acid sequence data bases, 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 was developed, which uses an improved algorithm that increases sensitivity with a small loss of selectivity and a negligible decrease in speed.

3.2.3 FASTP

The search algorithm proceeds through four steps in determining a score for pair-wise 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 sensitivity. Usually, k= 4-6 for DNA sequence and k= 1-2 for protein sequence.

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 the 10 highest scoring diagonal runs for each database sequence.

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

The substitution matrix is a scoring matrix that allows conservative replacements and runs of identities shorter than ktuple to contribute to the similarity score. Using the substitution matrix for amino acid (or nucleotide), the diagonal runs are rescored. For each of these vest diagonal regions, a subregion with maximal score is identified. We will refer to this region as the "initial region"; the best initial regions from the diagonal run. The best score among the 10 sub-regions is called the init1 score.

Step4:Rank the sequence

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

3.2.4 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, discard those with scores smaller than a given threshold. For the remaining sub-regions, attempts to join them by allowing indels. 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 can be computed using dynamic programming and it is called 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 complete the score opt. Finally, rank the sequences based on their opt scores.

3.3 BLAST

3.3.1 What is BLAST?

BLAST stands for Basic Local Alignment Search Tool [Gal00]. The BLAST programs are widely used tools for **searching DNA and protein databases for sequence similarity** - to identify homologies to a query sequence. It compares against all sequences in a database D based on a **heuristic algorithm** and has been designed for speed, with a minimal sacrifice of sensitivity to distant sequence relationships.

3.3.2 History of BLAST

In 1990, BLAST1 was born, which is very fast and dedicated to the search for regions of local similarities without gaps. In 1996-1997, BLAST2 was born, which derives from BLAST1. BLAST2 is different from BLAST1 in that BLAST2 allows for the insertion of gaps. BLAST2 has two versions, developed by two groups of authors independently. The first one is NCBI-BLAST2 and the other is

WU-BLAST2 (NCBI-BLAST2 is developed by National Center for Biotechnology Information in 1997 and WU is developed by Washington University in 1996).

3.3.3 Algorithm of BLAST1

BLAST1 is a heuristic method which searches for local similarity without gap. There are three distinct steps, which are represented as follow:

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

Step1:Preprocessing of the query

In Figure 3.1 we can see for every position p of the query, BLAST1 finds the list of w-tuples (strings of length w) scoring more than a threshold T when compared with the word of the query starting at position p. This list of w-tuples are called **neighbors**.

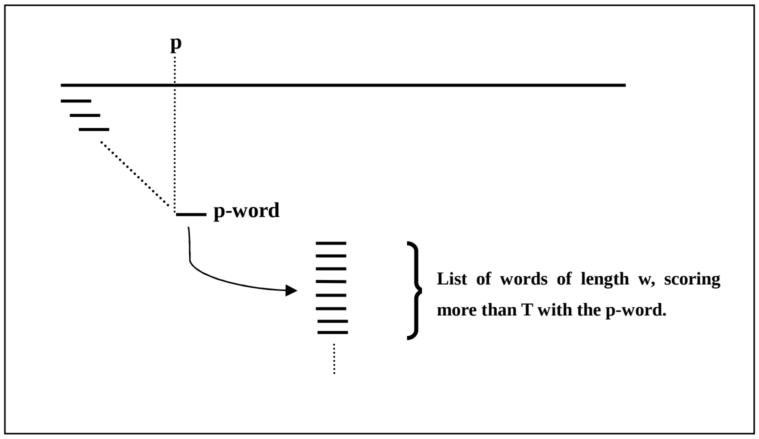

p
p-word
List of words of length w, scoring more than T with the p-word.

Figure 3.1: Preprocessing of the query

Step2:Generation of 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 P and a w-tuple in DB. If it does, **a hit** is made, which is characterized by the positions in both query and DB sequences (see Figure 3.2). All the possible hits between the query sequence and sequences from the database are calculated in that way.

Step 1 and step 2 look similar to the first step of FastA. We know that the first step of FastA is to quickly locate ungapped similarity regions between the query sequence and sequences from the database. Similarly, in BLAST1, all w-tuples (the strings of length w) of the query are compared with those of the database sequences. Unlike FastA, BLAST1 compares each word of the query with each word of the database sequences, and calculates the similarity score for all pairs of words (by summing the scores of the paired amino acids that are part of a paired word). Then, if the score of any pair of words is greater than or equal to the threshold, the pair of words is considered to be pairs of similar words. In this way, we can obtain all the words in the database that are similar to each word of the query.

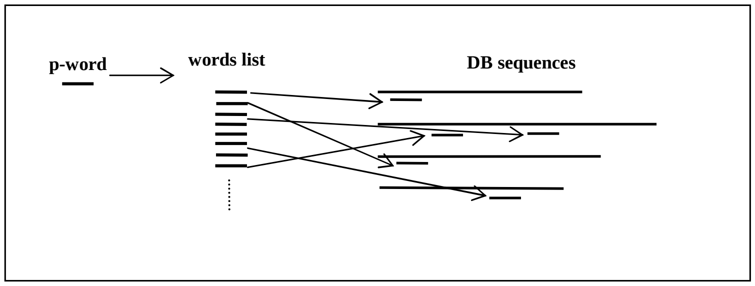


Figure 3.2: Generation of hits

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 3.3). To speed up this extension step, the extension is stopped as soon as the score decreases by more than X (the value chosen for X is a parameter of the program) from the highest value reached so far. Because of the "no more than X" drop off requirement, this manner of searching the best scoring paired segment containing a hit is just an approximate one. In fact, we cannot guarantee the the resulting segment pair has the highest score.

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

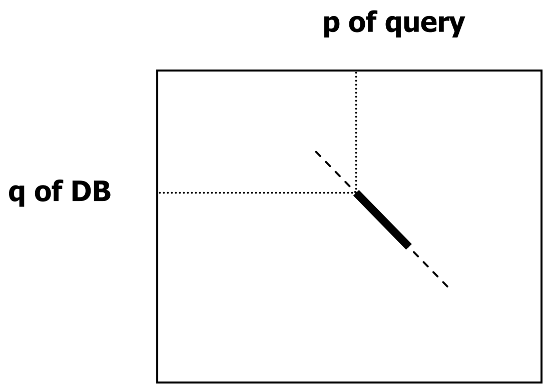


Figure 3.3: Extension of hits

3.3.4 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 that of BLAST1. But there are two major differences between them:

- **Two-hits requirement**

  In this problem, we should take into account the selection of the hits that are going to be extended. A requirement for a hit to be extended is that there is another hit, on the same diagonal, within a distance smaller than A (a parameter of the program, whose value may be changed by the user, by default, A=40). This process is illustrated in Figure 3.4.

  To make the program more sensitive, we can choose a smaller value for the T parameter (threshold for the similarity between words, used at the first step, when generating lists of "neighbors") so that we can generate more hits at the second step. Of course, these two parameters, A and T do not influence the sensitivity at the same level.

- **Gapped extension**

  Similar to the third step of BLAST1, all the hits that satisfying these requirements are selected for an ungapped extension. Among the generated HSP, we perform **gapped extension** for those with score above some

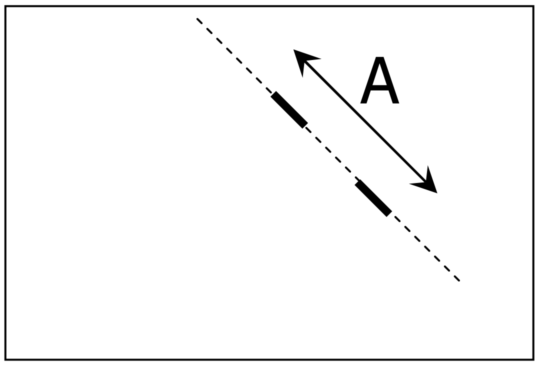


Figure 3.4: "two-hits" requirement

threshold: they are used as start points for performing dynamic programming local 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 3.5(d)) starting from the middle point of the HIT. In addition, when the alignment score drops off by more than Xg, stop.

3.3.5 BLAST1 VS. NCBI-BLAST2

The third step of the BLAST1 algorithm checks whether each hit lies within an alignment with score sufficient to be reported. This is done by extending a hit in both directions, until the running alignment's score has dropped more than X below the maximum score yet attained. This extension step is computationally quite costly; with the T and X parameters necessary to attain reasonable sensitivity to weak alignments, the extension step typically accounts for more than 90 percent of BLAST1's execution time. It is therefore desirable to reduce the number of extensions performed.

For NCBI-BLAST2, due to the two-hit requirement, the number of extensions is reduced. Moreover, NCBI-BLAST2 is about 3 times faster than BLAST1 since it is based upon the observation that an HSP of interest is much longer than a

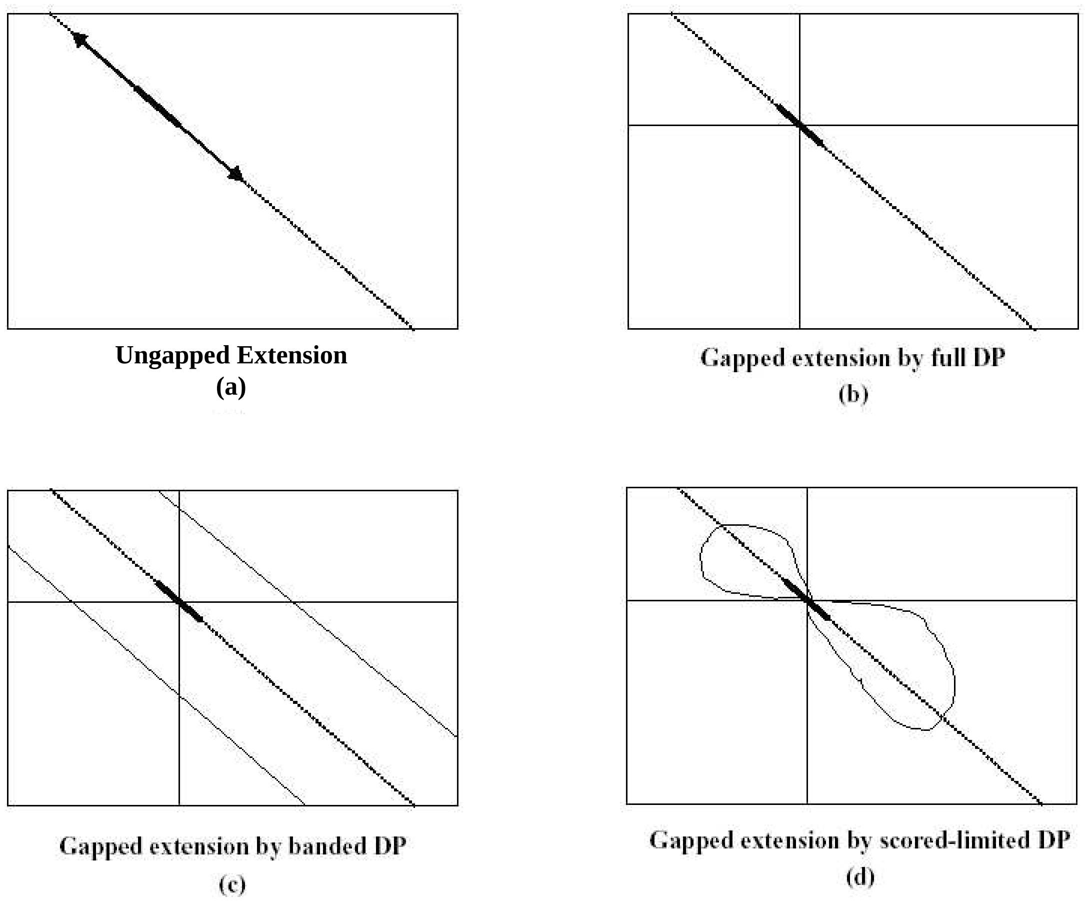

Ungapped Extension
(a)
Gapped extension by full DP
(b)
Gapped extension by banded DP
(c)
Gapped extension by scored-limited DP
(d)

Figure 3.5: Gapped and ungapped extensions

single word pair, and may therefore entail multiple hits on the same diagonal and within a relatively short distance of one anther.

3.3.6 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$^+$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 implements a greedy algorithm for the DNA sequence gapped 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.

3.4 PatternHunter

3.4.1 Introduction

For years, researchers are interested in faster and more sensitive methods for finding all approximate repeats or homologies in one DNA sequence or between two DNA sequences, as performed by the popular BLASTn (Altschul et al, 1990) program.

PatternHunter [MTL01] is a highly sensitive and efficient program for finding homologies within one, or between two DNA sequences. On very long sequences it runs faster than MegaBLAST while being more sensitive than BLASTn (at its default settings). It is the only software in the world that is capable of identifying all approximate repeats in a complete genome in a short time on desktop computer.

3.4.2 Two lines of existing approaches to improve

The first routine approach, such as BLAST, depends on homology searches. It is based on the strategy of finding short exact "seed" matches (hits) which are then extended into longer alignments. However, the exploding genomic data growth presents a dilemma for DNA homology search techniques: increasing seed size decreases sensitivity whereas decreasing seed size slows down computation.

Another line of approach, for example, MUMmer, QUASAR and REPuter, uses suffix trees. They are very awkward in handling mismatches and have in intrinsic large space requirement.

PatternHunter introduces novel seeding schemes and hit-processing methods to improve sensitivity and speed simultaneously.

3.4.3 How to do it?

BLAST searches matches of W (default W =11 in BLASTn and W =28 in MegaBLAST) consecutive letters as seeds. PatternHunter is similar to BLAST. Moreover, it uses non-consecutive W letters as seeds. They call the relative positions of the W letters a model, and W its weight.

They 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, they use 111010010100110111 as the optimal model.
For example ,

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

Figure 3.6: Example of two substrings which are matched according to the model

3.4.4 Advantage 1 — Increase 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 in 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 3.7 and 3.8 show the sensitivity of different models.

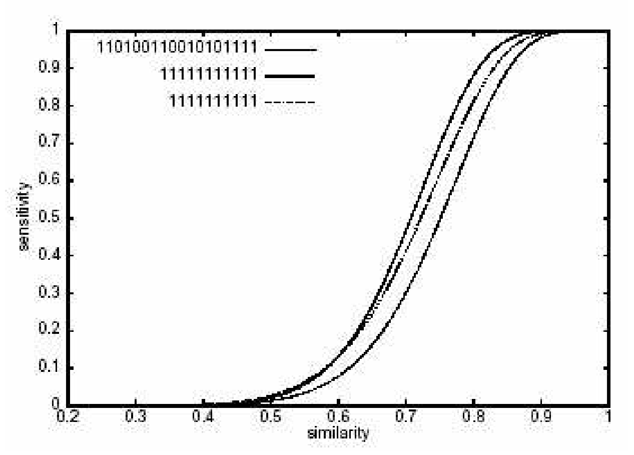


Figure 3.7: 1-hit performance of weight 11 spaced model versus weight 11 and 10 consecutive models, coordinates in logarithmic scale

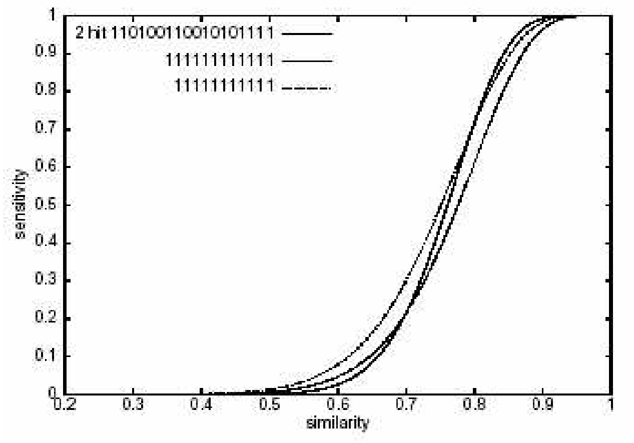


Figure 3.8: 2-hit performance of weight 11 spaced model versus single hit weight 11 and 12 consecutive models

3.4.5 Advantage 2 — Reduce the number of hits

For the same query length of 64, it is covered by 54 ungapped 11-tuples while it is covered by 47 gapped 11-tuples. So, the number of random hits is smaller. Thus, the efficiency is increased!

The expected number of hits in a region can be easily calculated as in the following Lemma.

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 $(L\text{-}M{+}1)$ 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%.

3.4.6 Not just seed!

To improve efficiency, PatternHunter uses a variety of advanced data structures including priority queues, a variation of red-black tree (but many times faster than Java standard), queues, hash tables.

PatternHunter also uses a new method of sequence alignment and has several other algorithmic improvements.

3.4.7 Results

PatternHunter is implemented in Java, hence it is cross-platform compatible. It is able to find homologies between sequences as large as human chromosomes in mere hours on a desktop. The following experimental result is the performance comparison with BLASTn and MegaBLAST, which is done on Personal Computer with pentium III 700MH, 1GB memory.

3.5 QUASAR

Due to the exploding genomic data growth, the expected software must be scalable to large datasets. Surprisingly, even the basic homology search tools (e.g. BLAST) are not scalable. QUASAR (Q-gram Alignment based on Suffix ARrays) is a new database searching algorithm, which was designed to quickly detect sequences with strong similarity to the query in a context where many searches are conducted in one database.

Seq1	Size	Seq2	Size	PH	PH2	MB28	Blastn
M. pneumoniae	828K	*M. genitalium*	589K	10s/65M	4s/48M	1s/88M	47s/45M
E. coli	4.7M	*H. influenza*	1.8M	34s/78M	14s/68M	5s/561M	716s/158M
A.thaliana chr 2	19.6M	*A.thaliana* chr 4	17.5M	5020s/279M	498s/231M	21720s/1087M	∞
H. sapiens chr 22	35M	*H. sapiens* chr 21	26.2M	14512s/419M	5250s/417M	∞	∞

Figure 3.9: Performance comparison with BLASTn, MegaBLAST: assure score for match 1, mismatch -1, gap open -5, gap extension -1. PH denotes PatternHunter with seed weight 11, PH2 denotes same with double hit model (sensitivity similar to BLAST's single hit size 11 seed, see Figure 3.7), MB28 denotes MegaBLAST with default seed size 28, and default affine gap penalties. BLASTn (via BL2SEQ) uses default seed size 11. Table entries under PH, PH2, MB28 and BLASTn indicate time(seconds) and space (megabytes) used; ∞ means out of memory or segmentation fault.

3.5.1 Introduction

The problem
(Approximate matching problem with k differences and window length 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+99].

3.5.2 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 $i = 1, 2, \ldots$

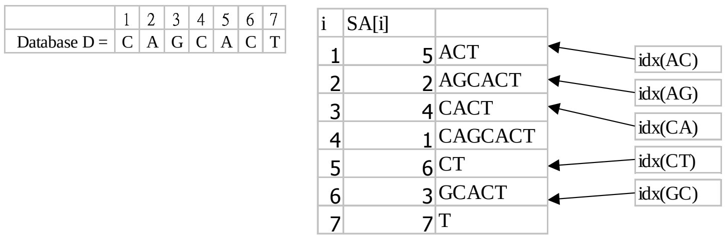
- For every length-w substring Y in D, associate a counter with it and initialize it to zero;
- For each q-gram Q in X
  - Find the *hitlist*, that is, the list of positions in D so that Q occurs
  - Increment the counter for every length-w substring Y in D, which contains Q;

- 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.

3.5.3 How to get the hitlist?

By using an index data structure for all q-grams in D we hope to direct the search for Q towards small portions of D and thus to avoid scanning the whole database. Since q is a parameter in the approach, we may use a full-text indexing data structure so that it is not necessary to rebuild the index if we change q.

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 3.10). The suffix array for D is constructed in a preprocessing step.



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

i	SA[i]	
1	5	ACT
2	2	AGCACT
3	4	CACT
4	1	CAGCACT
5	6	CT
6	3	GCACT
7	7	T

Figure 3.10: 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.

3.5.4 Speedup Feature

3.5.4.1 Window shifting

Previuos algorithm builds the counter list for every $S[i..w+i-1]$, where $i = 1, 2, ..., n-w+1$. It is time consuming. 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 3.11).

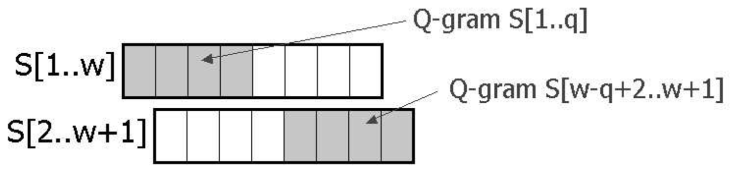


Figure 3.11: Windows Shifting Example.

First we decrement the counter values of all blocks that contain the q-gram $S[1..q]$ and that have not reached the threshold t, i.e., if a counter for a block has already reached t, leave it unmodified. In this way we marks all candidate blocks 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 block counters. Shift the window of length w over the string S until we reach its end.

3.5.4.2 Block Addressing

The other main drawback of above solution is the space required to store the counters. We may adopt *block addressing scheme* to reduce the amount of counters required. The solution works as follows: The database D is conceptually divided into blocks of fixed size b ($b \geq 2w$). Assign a counter to each block. This counter will be incremented whenever a search for a q-gram Q reports an occurrence inside the block. After processing all q-grams in $S[1..w]$, the counter of a certain block indicates how many q-grams from $S[1..w]$ are contained in this segment of the database. These counter values are stored in an array of size $|D|/b$. If a block contains more than t q-grams, this block has to be checked for approximate matches using a sequence alignment algorithm.

On the other hand, 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 3.12). Then, if a situation as described above occurs for blocks B_1 and B_3, block B_2 contains the potential candidate.

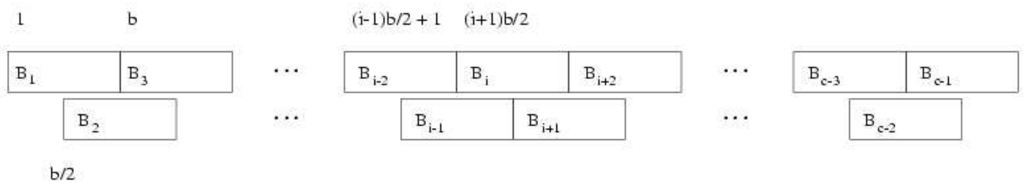


Figure 3.12: Partition of the database D into overlapping blocks of size b.

3.5.5 Complexity

The preprocessing-step (the construction of the suffix array and the precomputation of the search array) can be done in $O(|D|log|D|)$ (where $|D|$ is DB size) time. Searching for a specific q-gram requires constant time but the number of reported occurrences can be linear in $|D|$. There are $O(|S|)$ q-grams, so the 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. The space complexity of the algorithm is dominated by the space used for the suffix array. At construction time this need $9|D|$ space. At query time the algorithm will consume $5|D|$ space. And the algorithm is not suitable for distant homologous sequences.

3.6 Locality-Sensitive Hashing

In comparisons of multimegabase genomic DNA sequences, to process long sequences efficiently, existing algorithms find alignments by expanding around short runs of matching bases with no substitutions or other differences. But, such short exact matches often arise purely by chance in the background sequence. Thus, these algorithms must trade off between efficiency and sensitivity to features without long exact matches.

In this section, we will introduce a new algorithm, *LSH-ALL-PAIRS*, to find pairs of $w - mers$ that differ by at most d substitutions (ungapped local alignment) in a collection of biosequences database D.

3.6.1 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 $w-mers\ s$, we choose k indices $i_1, i_2, ..., i_k$ uniformly at random from the set $\{1, 2, ..., w\}$; assume that the indices are sampled with replacement, so that an index can be chosen multiple times. Define the function $\pi(s) = (s[i_1], s[i_2], ..., s[i_k])$. This function is called the *locality-sensitive hash function.*

Now, consider two $w-mers\ s_1$ and s_2 (see Figure 3.13), 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,

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

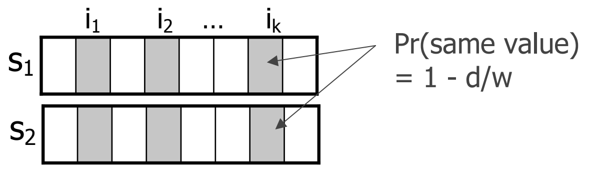


Figure 3.13: LSH Function: 2 $w-mers\ s_1$ and s_2.

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 negative by repeating the test using different $\pi()$ functions.

3.6.2 LSH-ALL-PAIRS Algorithm

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

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

LSH-ALL-PAIRS 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.

3.7 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+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.

[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.

[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.

[MTL01] B. Ma, J. Tromp and M. Li, "PatternHunter: Faster And More Sensitive Homology Search." , *Bioinformatics* , 2001.