a given state. The book by Durbin et al. [61] is a terrific reference book for this paradigm.

Alignment of two genomic sequences poses problems not well addressed by earlier alignment programs.

PipMaker [178] is a software tool for comparing two long DNA sequences to identify conserved segments and for producing informative, high-resolution displays of the resulting alignments. It displays a percent identity plot (pip), which shows both the position in one sequence and the degree of similarity for each aligning segment between the two sequences in a compact and easily understandable form. The alignment program used by the PipMaker network server is called BLASTZ [177]. It is an independent implementation of the Gapped BLAST algorithm specifically designed for aligning two long genomic sequences. Several modifications have been made to BLASTZ to attain efficiency adequate for aligning entire mammalian genomes and to increase the sensitivity.

MUMmer [119] is a system for aligning entire genomes rapidly. The core of the MUMmer algorithm is a suffix tree data structure, which can be built and searched in linear time and which occupies only linear space. DisplayMUMs 1.0 graphically presents alignments of MUMs from a set of query sequences and a single reference sequence. Users can navigate MUM alignments to visually analyze coverage, tiling patterns, and discontinuities due to misassemblies or SNPs.

The analysis of genome rearrangements is another exciting field for whole genome comparison. It looks for a series of genome rearrangements that would transform one genome into another. It was pioneered by Dobzhansky and Sturtevant [60] in 1938. Recent milestone advances include the works by Bafna and Pevzner [19], Hannenhalli and Pevzner [86], and Pevzner and Tesler [166].

Chapter 4
Homology Search Tools

The alignment methods introduced in Chapter 3 are good for comparing two sequences accurately. However, they are not adequate for homology search against a large biological database such as GenBank. As of February 2008, there are approximately 85,759,586,764 bases in 82,853,685 sequence records in the traditional GenBank divisions. To search such kind of huge databases, faster methods are required for identifying the homology between the query sequence and the database sequence in a timely manner.

One common feature of homology search programs is the filtration idea, which uses exact matches or approximate matches between the query sequence and the database sequence as a basis to judge if the homology between the two sequences passes the desired threshold.

This chapter is divided into six sections. Section 4.1 describes how to implement the filtration idea for finding exact word matches between two sequences by using efficient data structures such as hash tables, suffix trees, and suffix arrays.

FASTA was the first popular homology search tool, and its file format is still widely used. Section 4.2 briefly describes a multi-step approach used by FASTA for finding local alignments.

BLAST is the most popular homology search tool now. Section 4.3 reviews the first version of BLAST, *Ungapped BLAST*, which generates ungapped alignments. It then reviews two major products of BLAST 2.0: Gapped BLAST and Position-Specific Iterated BLAST (PSI-BLAST). Gapped BLAST produces gapped alignments, yet it is able to run faster than the original one. PSI-BLAST can be used to find distant relatives of a protein based on the profiles derived from the multiple alignments of the highest scoring database sequence segments with the query segment in iterative Gapped BLAST searches.

Section 4.4 describes BLAT, short for "BLAST-like alignment tool." It is often used to search for the database sequences that are closely related to the query sequences such as producing mRNA/DNA alignments and comparing vertebrate sequences.

PatternHunter, introduced in Section 4.5, is more sensitive than BLAST when a hit contains the same number of matches. A novel idea in PatternHunter is the use of an optimized spaced seed. Furthermore, it has been demonstrated that using optimized multiple spaced seed will speed up the computation even more.

Finally, we conclude the chapter with the bibliographic notes in Section 4.6.

4.1 Finding Exact Word Matches

An exact word match is a run of identities between two sequences. In the following, we discuss how to find all short exact word matches, sometimes referred to as *hits*, between two sequences using efficient data structures such as hash tables, suffix trees, and suffix arrays.

Given two sequences $A = a_1a_2 \ldots a_m$, and $B = b_1b_2 \ldots b_n$, and a positive integer k, the *exact word match* problem is to find all occurrences of exact word matches of length k, referred to as k-mers between A and B. This is a classic algorithmic problem that has been investigated for decades. Here we describe three approaches for this problem.

4.1.1 Hash Tables

A hash table associates keys with numbers. It uses a hash function to transform a given key into a number, called hash, which is used as an index to look up or store the corresponding data. A method that uses a hash table for finding all exact word matches of length w between two DNA sequences A and B is described as follows.

Since a DNA sequence is a sequence of four letters A, C, G, and T, there are 4^w possible DNA w-mers. The following encoding scheme maps a DNA w-mer to an integer between 0 and $4^w - 1$. Let $C = c_1c_2 \ldots c_w$ be a w-mer. The hash value of C is written $V(C)$ and its value is

$$V(C) = x_1 \times 4^{w-1} + x_2 \times 4^{w-2} + \cdots + x_w \times 4^0,$$

where $x_i = 0, 1, 2, 3$ if $c_i =$ A, C, G, T, respectively. For example, if C=GTCAT, then

$$V(C) = 2 \times 4^4 + 3 \times 4^3 + 1 \times 4^2 + 0 \times 4^1 + 3 \times 4^0 = 732.$$

In fact, we can use two bits to represent each nucleotide: A(00), C(01), G(10), and T(11). In this way, a DNA segment is transformed into a binary string by compressing four nucleotides into one byte. For C=GTCAT given above, we have

$$V(C) = 732 = 1011010011_2.$$

Initially, a hash table H of size 4^w is created. To find all exact word matches of length w between two sequences A and B, the following steps are executed. The first step is to hash sequence A into a table. All possible w-mers in A are calculated by

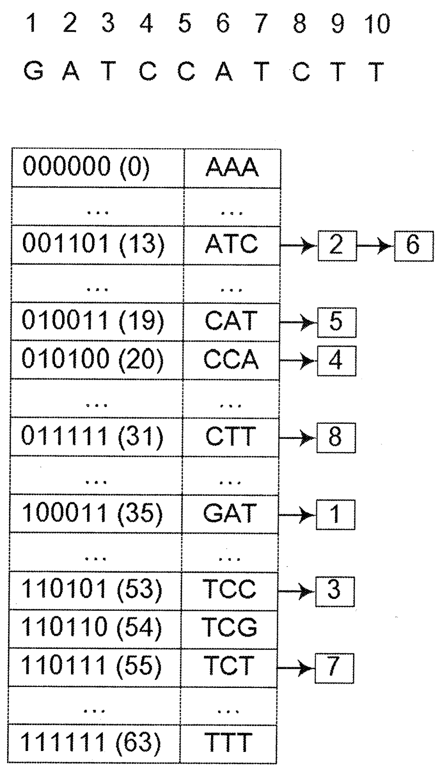

Fig. 4.1 A 3-mer hash table for GATCCATCTT.

sliding a window of size w from position 1 to position $m - w + 1$. For each word C, we compute $V(C)$ and insert C to the entry $H[V(C)]$. If there is more than one window word having the same hash value, a linked list or an array can be used to store them. Figure 4.1 depicts the process of constructing a hash table of word size 3 for GATCCATCTT.

Once a hash table for sequence A has been built, we can now scan sequence B by sliding a window of size w from position 1 to position $n - w + 1$. For each scan S, we compute $V(S)$ and find its corresponding exact word matches, if any, in A by looking up the entry $H[V[S]$ in the hash table H. All the exact word matches can be found in an order of their occurrences.

A hash table works well in practice for a moderate word size, say 12. However, it should be noted that for some larger word sizes, this approach might not be feasible. Suppose an exact word match of interest has 40 nucleotides. There are 4^{40} possible combinations. If we use an array to store all possible keys, then we would need 4^{40} entries to assign a different index to each combination, which would be far beyond the capacity of any modern computers. A more succinct indexing technique, such as suffix trees or suffix arrays, is required for this particular application.

4.1.2 Suffix Trees

A sequence $A = a_1a_2 \ldots a_m$ has m suffixes, namely, $a_1 \ldots a_m$, $a_2 \ldots a_m$, $a_3 \ldots a_m$, $\ldots$, and a_m. A suffix tree for sequence A is a rooted tree such that every suffix of A corresponds uniquely to a path from the root to a tree node. Furthermore, each edge of the suffix tree is labeled with a nonempty substring of A, and all internal nodes except the root must have at lease two children.

Figure 4.2 constructs a suffix tree for GATCCATCTT. The number of a node specifies the starting position of its corresponding suffix. Take the number "5" for example. If we concatenate the labels along the path from the root to the node with the number "5," we get CATCTT, which is a suffix starting at position 5. Notice that some internal node might associate with a number, *e.g.*, the node with number "10" in this figure. The reason is that T is suffix starting at position 10, yet it is also a prefix of another three suffixes TCCATCTT, TCTT, and TT.

For convenience, one may require that all suffixes correspond to paths from the root to the *leaves*. In fact, if sequence A is padded with a terminal symbol, say \$, that does not appear in A, then every suffix would correspond to a path from the root to a leaf node in the suffix tree because a suffix with "\$" will not be a prefix of any other suffix. Figure 4.3 constructs a suffix tree for GATCCATCTT\$. Now every suffix corresponds to a path from the root to a leaf, including the suffix starting at position 10.

For a constant-size alphabet, the construction of a suffix tree for a sequence of length m can be done in $O(m)$ time and space based on a few other crucial observations, including the use of suffix links. Once a suffix tree has been built, we can answer several kinds of pattern matching queries iteratively and efficiently. Take the exact word match problem for example. Given are two sequences $A = a_1a_2 \ldots a_m$, and $B = b_1b_2 \ldots b_n$, and a positive integer w. An exact word match of length w occurs in $a_ia_{i+1} \ldots a_{i+w-1}$ and $b_jb_{j+1} \ldots b_{j+w-1}$ if and only if the suffixes $a_ia_{i+1} \ldots a_m$ and $b_jb_{j+1} \ldots b_n$ share a common prefix of length at least w. With a suffix tree at hand, finding a common prefix becomes an easy job since all suffixes with a common prefix will share the path from the root that labels out that common prefix in the suffix tree. Not only does it work well for finding all exact word matches of a fixed length, it can also be used to detect all maximal word matches between two sequences or among several sequences by employing a so-called generalized suffix tree, which is a suffix tree for a set of sequences. Interested readers are referred to the book by Gusfield [85].

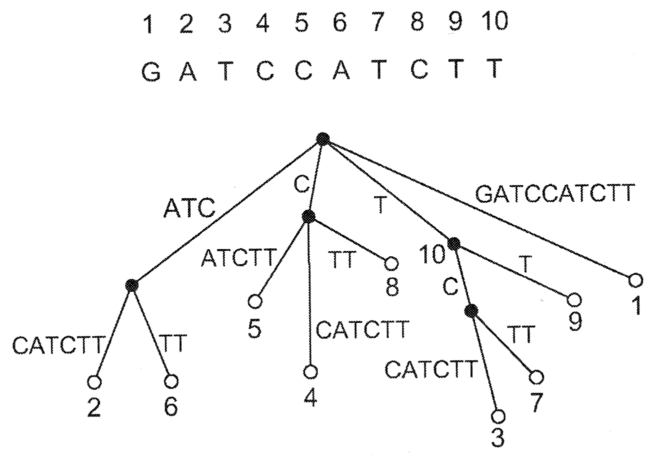

Fig. 4.2 A suffix tree for GATCCATCTT.

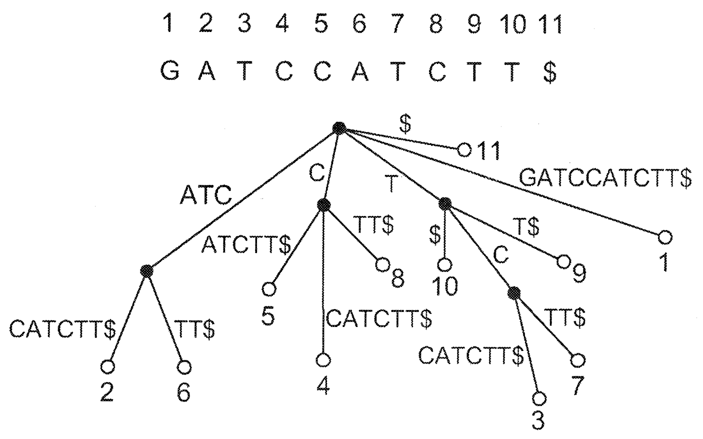

Fig. 4.3 A suffix tree for GATCCATCTT\$.

4.1.3 Suffix Arrays

A suffix array for sequence $A = a_1a_2 \ldots a_m$ is an array of all suffixes of A in lexicographical order. Figure 4.4 constructs a suffix array for GATCCATCTT. At first glance, this conceptual representation seems to require quadratic space, but in fact the suffix array needs only linear space since it suffices to store only the starting positions for all sorted suffixes.

Recall that an exact word match of length w occurs in $a_ia_{i+1} \ldots a_{i+w-1}$ and $b_jb_{j+1} \ldots b_{j+w-1}$ if and only if the suffixes $a_ia_{i+1} \ldots a_m$ and $b_jb_{j+1} \ldots b_n$ share a common prefix of length at least w. Once a suffix array has been built, one can look up the table for any particular prefix by a binary search algorithm. This search can be done even more efficiently if some data structure for querying the longest common prefixes is employed.

1	2	3	4	5	6	7	8	9	10
G	A	T	C	C	A	T	C	T	T

(a) all suffixes

GATCCATCTT	1
ATCCATCTT	2
TCCATCTT	3
CCATCTT	4
CATCTT	5
ATCTT	6
TCTT	7
CTT	8
TT	9
T	10

(b) suffix array

ATCCATCTT	2
ATCTT	6
CATCTT	5
CCATCTT	4
CTT	8
GATCCATCTT	1
T	10
TCCATCTT	3
TCTT	7
TT	9

Fig. 4.4 A suffix array for GATCCATCTT.

4.2 FASTA

FASTA uses a multi-step approach to finding local alignments. First, it finds runs of identities, and identifies regions with the highest density of identities. A parameter *ktup* is used to describe the minimum length of the identity runs. These runs of identities are grouped together according to their diagonals. For each diagonal, it locates the highest-scoring segment by adding up bonuses for matches and subtracting penalties for intervening mismatches. The ten best segments of all diagonals are selected for further consideration.

The next step is to re-score those selected segments using the scoring matrix such as PAM and BLOSUM, and eliminate segments that are unlikely to be part of the alignment. If there exist several segments with scores greater than the cutoff, they will be joined together to form a chain provided that the sum of the scores of the joined regions minus the gap penalties is greater than the threshold.

Finally, it considers the band of a couple of residues, say 32, centered on the chain found in the previous step. A banded Smith-Waterman method is used to deliver an optimal alignment between the query sequence and the database sequence.

Since FASTA was the first popular biological sequence database search program, its sequence format, called FASTA format, has been widely adopted. FASTA format is a text-based format for representing DNA, RNA, and protein sequences, where each sequence is preceded by its name and comments as shown below:

```
>HAHU Hemoglobin alpha chain - Human
VLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTK
TYFPHFDLSHGSAQVKGHGKKVADALTNAVAHVDDMPNAL
SALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPA
VHASLDKFLASVSTVLTSKYR
```

4.3 BLAST

The BLAST program is the most widely used tool for homology search in DNA and protein databases. It finds regions of local similarity between a query sequence and each database sequence. It also calculates the statistical significance of matches. It has been used by numerous biologists to reveal functional and evolutionary relationships between sequences and identify members of gene families.

The first version of BLAST was launched in 1990. It generates ungapped alignments and hence is called *Ungapped BLAST*. Seven years later, BLAST 2.0 came to the world. Two major products of BLAST 2.0 are Gapped BLAST and Position-Specific Iterated BLAST (PSI-BLAST). Gapped BLAST produces gapped alignments, yet it is able to run faster than the original one. PSI-BLAST can be used to find distant relatives of a protein based the profiles derived from the multiple alignments of the highest scoring database sequence segments with the query segment in iterative Gapped BLAST searches.

4.3.1 Ungapped BLAST

As discussed in Chapter 3, all possible pairs of residues are assigned their similarity scores when we compare biological sequences. For protein sequences, PAM or

	C	T	A	T	C	A	T	T	C	T	G
G	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	5
A	-4	-4	5	-4	-4	5	-4	-4	-4	-4	-4
T	-4	5	-4	5	-4	-4	5	5	-4	5	-4
C	5	-4	-4	-4	5	-4	-4	-4	5	-4	-4
C	5	-4	-4	-4	5	-4	-4	-4	5	-4	-4
A	-4	-4	5	-4	-4	5	-4	-4	-4	-4	-4
T	-4	5	-4	5	-4	-4	5	5	-4	5	-4
C	5	-4	-4	-4	5	-4	-4	-4	5	-4	-4
T	-4	5	-4	5	-4	-4	5	5	-4	5	-4
T	-4	5	-4	5	-4	-4	5	5	-4	5	-4

Fig. 4.5 A matrix of similarity scores for the pairs of residues of the two sequences GATCCATCTT and CTATCATTCTG.

BLOSUM substitution matrix is often employed, whereas for DNA sequences, an identity is given a positive score and a mismatch is penalized by a negative score. Figure 4.5 depicts the similarity scores of all the pairs of the residues of the two sequences `GATCCATCTT` and `CTATCATTCTG`, where an identity is given a score +5 and a mismatch is penalized by -4.

Let a sequence segment be a contiguous stretch of residues of a sequence. The score for the aligned segments $a_i a_{i+1} \dots a_{i+\ell-1}$ and $b_j b_{j+1} \dots b_{j+\ell-1}$ of length ℓ is the sum of the similarity scores for each pair of aligned residues (a_{i+k}, b_{j+k}) where $0 \leq k < \ell$. A maximal-scoring segment pair (MSP) is the highest scoring pair of segments of the same length chosen from the two sequences. Since its score is the highest, any stretch of this aligned segment pair will not increase the similarity score. In order to compute the MSP score, a straightforward approach is to compute the maximum-sum segment for each diagonal of the similarity scores matrix of the two sequences. Fix a diagonal, the maximum-sum segment can be found by a linear-time algorithm for the maximum-sum segment problem given in Section 2.4.2. Figure 4.6 locates a maximal-scoring segment pair in Figure 4.5.

However, there are $O(m+n)$ diagonals to be processed. If we apply the linear-time algorithm to all the diagonals, the resulting method takes the time proportional to the product of the lengths of the sequences. To speed up the computation, BLAST computes approximate MSPs, often referred to as high-scoring segment pairs (*HSPs*), in two phases. The first phase is to scan the database for hits, which are word pairs of length w with score at least T. The second phase is to extend each hit to see if it is contained within a segment pair whose score is no less than S.

Let us now explain these two phases in greater detail. In the first phase, BLAST seeks only segment pairs containing a *hit*, which is a word pair of length w with score at least T. For DNA sequences, these word pairs are exact word matches of fixed

		1	2	3	4	5	6	7	8	9	10	11
		C	T	A	T	C	A	T	T	C	T	G
1	G	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	5
2	A	-4	-4	5	-4	-4	5	-4	-4	-4	-4	-4
3	T	-4	5	-4	5	-4	-4	5	5	-4	5	-4
4	C	5	-4	-4	-4	5	-4	-4	-4	5	-4	-4
5	C	5	-4	-4	-4	5	-4	-4	-4	5	-4	-4
6	A	-4	-4	5	-4	-4	5	-4	-4	-4	-4	-4
7	T	-4	5	-4	5	-4	-4	5	5	-4	5	-4
8	C	5	-4	-4	-4	5	-4	-4	-4	5	-4	-4
9	T	-4	5	-4	5	-4	-4	5	5	-4	5	-4
10	T	-4	5	-4	5	-4	-4	5	5	-4	5	-4

Fig. 4.6 A maximum-scoring segment pair of the two sequences `GATCCATCTT` and `CTATCATTCTG`.

length w, whereas for protein sequences, these word pairs are those fixed-length segment pairs who have a score no less than the threshold T.

Section 4.1 gives three methods for finding exact word matches between two sequences. Figure 4.7 depicts all the exact word matches of length three between the two sequences `GATCCATCTT` and `CTATCATTCTG`.

For protein sequences, we are not looking for exact word matches. Instead, a hit is a fixed-length segment pair having a score no less than the threshold T. A query word may be represented by several different words whose similarity scores with the query word are at least T.

The second phase of BLAST is to extend a hit in both directions (*diagonally*) to find a locally maximal-scoring segment pair containing that hit. It continues the extension in one direction until the score has dropped more than X below the maximum score found so far for shorter extensions. If the resulting segment pair has score at least S, then it is reported.

It should be noted that both the Smith-Waterman algorithm and BLAST asymptotically take the time proportional to the product of the lengths of the sequences. The speedup of BLAST comes from the reduced sample space size. For two sequences of lengths m and n, the Smith-Waterman algorithm involves $(n+1) \times (m+1)$ entries in the dynamic-programming matrix, whereas BLAST takes into account only those w-mers, whose number is roughly $mn/4^w$ for DNA sequences or $mn/20^w$ for protein sequences.

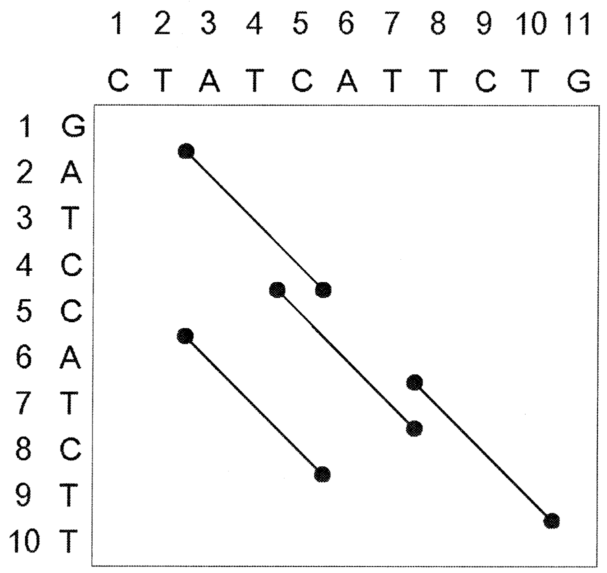

Fig. 4.7 Exact word matches of length three between the two sequences `GATCCATCTT` and `CTATCATTCTG`.

4.3.2 Gapped BLAST

Gapped BLAST uses a new criterion for triggering hit extensions and generates gapped alignment for segment pairs with "high scores."

It was observed that the hit extension step of the original BLAST consumes most of the processing time, say 90%. It was also observed that an HSP of interest is much longer than the word size w, and it is very likely to have multiple hits within a relatively short distance of one another on the same diagonal. Specifically, the two-hit method is to invoke an extension only when two non-overlapping hits occur within distance D of each other on the same diagonal (see Figure 4.8). These adjacent non-overlapping hits can be detected if we maintain, for each diagonal, the coordinate of the most recent hit found.

Another desirable feature of Gapped BLAST is that it generates gapped alignments explicitly for some cases. The original BLAST delivers only ungapped alignments. Gapped alignments are implicitly taken care of by calculating a joint statistical assessment of several distinct HSPs in the same database sequence.

A gapped extension is in general much slower than an ungapped extension. Two ideas are used to handle gapped extensions more efficiently. The first idea is to trigger a gapped extension only for those HSPs with scores exceeding a threshold S_g. The parameter S_g is chosen in a way that no more than one gap extension is invoked per 50 database sequences.

The second idea is to confine the dynamic programming to those cells for which the optimal local alignment score drops no more than X_g below the best alignment

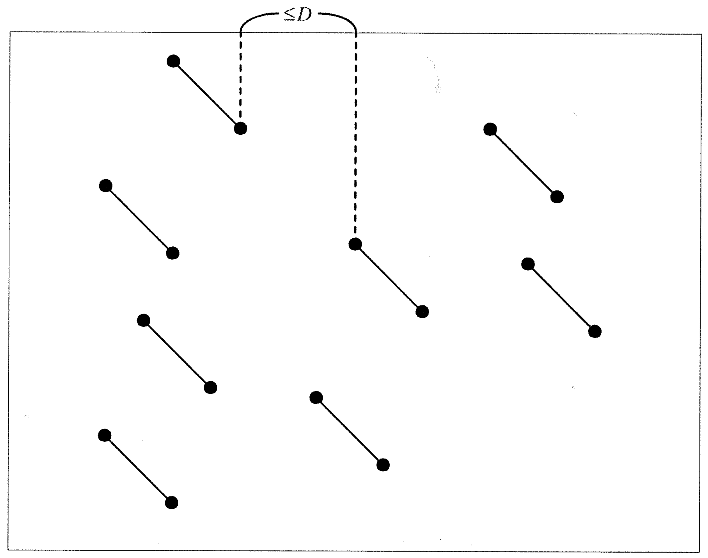

Fig. 4.8 Two non-overlapping hits within distance D of each other on the same diagonal.

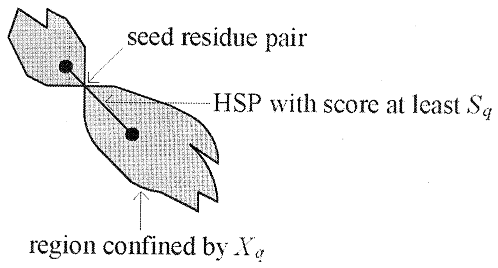

Fig. 4.9 A scenario of the gap extensions in the dynamic-programming matrix confined by the parameter X_g.

score found so far. The gapped extension for a selected HSP starts from a seed residue pair, which is a central residue pair of the highest-scoring length-11 segment pair along the HSP. If the HSP is shorter than 11, its central residue pair is chosen as the seed. Then the gapped extension proceeds both forward and backward through the dynamic-programming matrix confined by the parameter X_g (see Figure 4.9).

4.3.3 PSI-BLAST

PSI-BLAST runs BLAST iteratively with an updated scoring matrix generated automatically. In each iteration, PSI-BLAST constructs a position specific score matrix (PSSM) of dimension $\ell \times 20$ from a multiple alignment of the highest-scoring segments with the query segment of length ℓ. The constructed PSSM is then used to score the segment pairs for the next iteration. It has been shown that this iterative approach is often more sensitive to weak but biologically relevant sequence similarities.

PSI-BLAST collects, from the BLAST output, all HSPs with E-value below a threshold, say 0.01, and uses the query sequence as a template to construct a multiple alignment. For those selected HSPs, all database sequence segments that are identical to the query segment are discarded, and only one copy is kept for those database sequence segments with at least 98% identities. In fact, users can specify the maximum number of database sequence segments to be included in the multiple alignment. In case the number of HSPs with E-value below a threshold exceeds the maximum number, only those top ones are reported. A sample multiple alignment is given below:

```
query:  GVDIIIMGSHGKTNLKEILLGSVTENVIKKSNKPVLVVK
seq1:   GADVVVIGSR-NPSISTHLLGSNASSVIRHANLPVLVVR
seq2:   PAHMIIIASH-RPDITTYLLGSNAAAVVRHAECSVLVVR
seq3:   QAGIVVLGTVGRTGISAAFLGNTAEQVIDHLRCDLLVIK
```

If all segments in the alignment are given the same weight, then a small set of more divergent sequences might be suppressed by a much larger set of closely related sequences. To avoid such a bias, PSI-BLAST assigns various sequence weights by a sequence weighting method. Thus, to calculate the observed residue frequencies of a column of a multiple alignment, PSI-BLAST takes its weighted frequencies into account. In Chapter 8, we shall discuss the theoretical foundation for generating scoring matrices from a given multiple alignment.

4.4 BLAT

BLAT is short for "BLAST-like alignment tool." It is often used to search for database sequences that are closely related to the query sequences. For DNA sequences, it aims to find those sequences of length 25 bp or more and with at least 95% similarity. For protein sequences, it finds those sequences of length 20 residues or more and with at least 80% similarity.

A desirable feature is that BLAT builds an index of the whole database and keeps it in memory. The index consists of non-overlapping K-mers and their positions in the database. It excludes those K-mers that are heavily involved in repeats. DNA BLAT sets K to 11, and protein BLAT sets K to 4. The index requires a few gigabytes of RAM and is affordable for many users. This feature lets BLAT scan linearly through the query sequence, rather than scan linearly through the database.

BLAT builds a list of hits by looking up each overlapping K-mer of the query sequence in the index (see Figure 4.10). The hits can be single perfect word matches or near perfect word matches. The near perfect mismatch option allows one mismatch in a hit. Given a K-mer, there are $K \times (|\Sigma| - 1)$ other possible K-mers that match it in all but one position, where $|\Sigma|$ is the alphabet size. Therefore, the near perfect mismatch option would require $K \times (|\Sigma| - 1) + 1$ lookups per K-mer of the query sequences.

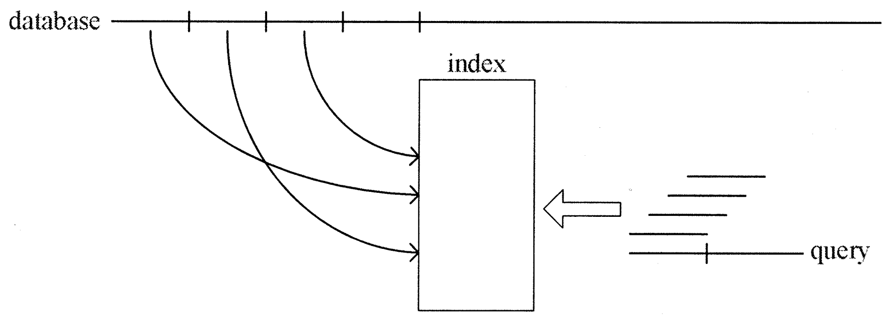

Fig. 4.10 The index consists of non-overlapping K-mers in the database, and each overlapping K-mer of the query sequence is looked up for hits.

BLAT identifies homologous regions of the database by clumping hits as follows. The hits are distributed into buckets of 64k according to their database positions. In each bucket, hits are bundled into a clump if they are within the gap limit on the diagonal and the window limit on the database coordinate. To smooth the boundary cut, the hits and clumps within the window limit of the next bucket are passed on for possible clumping and extension. If a clump contains a certain amount of hits, then it defines a region of the database that are homologous to the query sequence. If two homologous regions of the database are within 300 nucleotides or 100 amino acids, they are merged as one. Finally, each homologous region is flanked with a few hundred additional residues on both sides.

The alignment stage of BLAT delivers alignments in the homologous regions found in the clumping process. The alignment procedures for nucleotide sequences and protein sequences are different. Both of them work well for aligning sequences that are closely related.

For nucleotide alignment, the alignment procedure works as follows. It stretches each K-mer as far as possible allowing no mismatches. An extended K-mer forms a hit if it is unique or exceeds a certain length. Overlapping hits are merged together. To bridge the gaps among hits, a recursive procedure is employed. A recursion continues if it finds no additional hits using a smaller K or the gap is no more than five nucleotides.

For protein alignment, the alignment procedure works as follows. The hits are extended into high-scoring segment pairs (HSPs) by giving a bonus score +2 to a match and penalizing a mismatch by -1. Let H_1 and H_2 be two HSPs that are non-overlapping in both the database and the query sequences, and assume that H_1 precedes H_2. An edge is added to connect from H_1 to H_2, where the edge weight is the score of H_2 minus the gap cost based on the distance between H_1 and H_2. For those overlapping HSPs, a cutting point is chosen to maximize the score of the joint HSP. Then a dynamic-programming method is used to find the maximal-scoring path, *i.e.*, the maximal-scoring alignment, in the graph one by one until all HSPs are reported in some alignment.

4.5 PatternHunter

As discussed in Section 4.3, BLAST computes HSPs by extending so-called "hits" or "seeds" between the query sequence and the database sequence. The seeds used by BLAST are short contiguous word matches. Some homologous regions might be missed if they do not contain any seed.

An advanced homology search program named PatternHunter has been developed to enhance the sensitivity by finding short word matches under a spaced seed model. A spaced seed is represented as a binary string of *length l*, where a "1" bit at a position means that a base match is required at the position, and a "$*$" bit at a position means that either a base match or mismatch is acceptable at the position. The number of 1 bits in a spaced seed is the *weight* of the seed. For example, the words ACGTC and ATGAC form a word match under spaced seed 1*1*1, but not under 11**1. Note that BLAST simply uses a consecutive model that consists of consecutive 1s, such as 11111.

In general, a spaced seed model shares fewer 1s with any of its shifted copies than the contiguous one. Define the number of overlapping 1s between a model and its shifted copy as the number of 1s in the shifted copy that correspond to 1s in the model. The number of non-overlapping 1s between a model and its shifted copy is the weight of the model minus the number of overlapping 1s. If there are more non-overlapping 1s between the model and its shifted copy, then the conditional probability of having another hit given one hit is smaller, resulting in higher sensitivity. For rigorous analysis of the hit probabilities of spaced seeds, the reader is referred to Chapter 6.

A model of length l has $l-1$ shifted copies. For a model π of length l, the sum of overlapping hit probabilities between the model and each of its shifted copies, $\phi(\pi, p)$, is calculated by the equation

$$\phi(\pi,p) = \sum_{i=1}^{l-1} p^{n_i}, \tag{4.1}$$

where p is the similarity level and n_i denotes the number of non-overlapping 1s between the model π and its i^{th} shifted copy. Figure 4.11 computes $\phi(\pi,p)$ for π=1*11*1.

Both empirical and analytical studies suggest that, among all the models of fixed length and weight, a model is more sensitive if it has a smaller sum of overlapping hit probabilities. A model of length l and weight w is an *optimal* model if its ϕ value is minimum among all models of length l and weight w. For example, the spaced seed model 111*1**1*1**11*111 used in PatternHunter is an optimal one for length 18 and weight 11 with similarity level $p = 0.7$.

In order to calculate the value of ϕ for a spaced seed model, we need to count the number of non-overlapping 1s between the model and each of its shifted copies, which can be computed by subtracting the number of overlapping 1s from the weight w. This can be done by a straightforward dynamic-programming method, which takes $O(\ell^2)$ time to compute ϕ for any model of length ℓ. By observing that at most $O(\ell)$ bit overlaps differ for two models with only one pair of * and 1 swapped, one

i	i^{th} shifted copy	non-overlapping 1s	overlapping hit probability
0	1*11*1		
1	1*11*1	3	p^3
2	1*11*1	2	p^2
3	1*11*1	2	p^2
4	1*11*1	4	p^4
5	1*11*1	3	p^3

Fig. 4.11 Calculation of the sum of overlapping hit probabilities between the model π and each of its shifted copies, $\phi(\pi,p) = p^3 + p^2 + p^2 + p^4 + p^3$, for π=1*11*1.

can develop an $O(\ell)$ time algorithm for updating the numbers of non-overlapping 1s for a swapped model. This suggests a practical computation method for evaluating ϕ values of all models by orderly swapping one pair of * and 1 at a time.

Given a spaced seed model, how can one find all the hits? Recall that only those 1s account for a match. One way is to employ a hash table like Figure 4.1 for exact word matches but use a spaced index instead of a contiguous index. Let the spaced seed model be of length ℓ and weight w. For DNA sequences, a hash table of size 4^w is initialized. Then we scan the sequence with a window of size ℓ from left to right. We extract the residues corresponding to those 1s as the index of the window (see Figure 4.12).

Once the hash table is built, the lookup can be done in a similar way. Indeed, one can scan the other sequence with a window of size ℓ from left to right and extract the residues corresponding to those 1s as the index for looking up the hash table for hits.

Hits are extended diagonally in both sides until the score drops by a certain amount. Those extended hits with scores exceeding a threshold are collected as

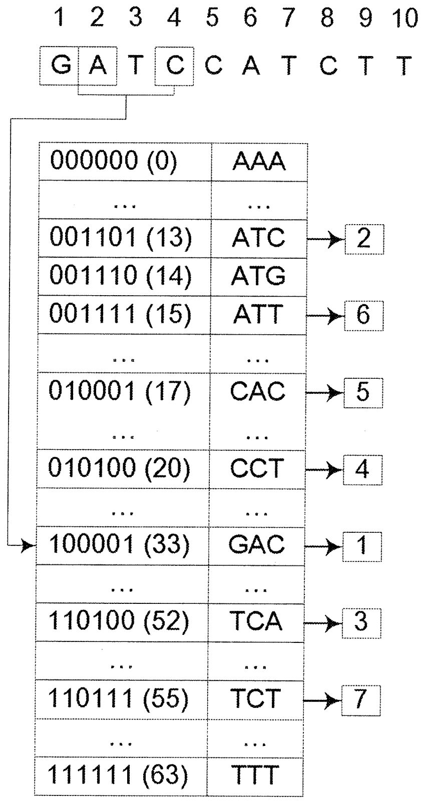

Fig. 4.12 A hash table for GATCCATCTT under a weight three model 11*1.

high-scoring segment pairs (HSPs). As for the gap extension of HSPs, a red-black tree with diagonals as the key is employed to manipulate the extension process efficiently.

4.6 Bibliographic Notes and Further Reading

Twenty years ago, it would have been a legend to find similarities between two sequences. However, nowadays it would be a great surprise if we cannot find similarities between a newly sequenced biomolecular sequence and the GenBank database. FASTA and BLAST were the ones that boosted this historical change. In Table C.2 of Appendix C, we compile a list of homology search tools.

4.1

The first linear-time suffix tree was given by Weiner [202] in 1973. A space-efficient linear-time construction was proposed by McCreight [135] in 1976. An on-line linear-time construction was presented by Ukkonen [191] in 1995. A review of these three linear-time algorithms was given by Giegerich and Kurtz [75]. Gusfield's book [85] gave a very detailed explanation of suffix trees and their applications.

Suffix arrays were proposed by Manber and Myers [134] in 1991.

4.2

The FASTA program was proposed by Pearson and Lipman [161] in 1988. It improved the sensitivity of the FASTP program [128] by joining several initial regions if their scores pass a certain threshold.

4.3

The original BLAST paper by Altschul et al. [7] was the most cited paper published in the 1990s. The first version of BLAST generates ungapped alignments, whereas BLAST 2.0 [8] considers gapped alignments as well as position-specific scoring schemes. The idea of seeking multiple hits on the same diagonal was first proposed by Wilbur and Lipman in 1983 [204]. The book of Korf, Yandell, and Bedell [116] gave a full account of the BLAST tool. It gave a clear understanding of BLAST programs and demonstrated how to use BLAST for taking the full advantage.

4.4

The UCSC Genome Browser contains a large collection of genomes [104]. BLAT [109] is used to map the query sequence to the genome.

4.5

Several variants of PatternHunter [131] are also available to the public. For instance, PatternHunter II [123] improved the sensitivity even further by using multiple spaced seeds, and tPatternHunter [112] was designed for doing protein-protein, translated protein-DNA, and translated DNA-DNA homology searches. A new version of BLASTZ also adapted the idea of spaced models [177] by allowing one transition (`A-G`, `G-A`, `C-T` or `T-C`) in any position of a seed.

Some related but different *spaced* approaches have been considered by others. Califano and Rigoutsos [37] introduced a new index generating mechanism where k-tuples are formed by concatenating non-contiguous subsequences that are spread over a large portion of the sequence of interest. The first stage of the multiple filtration approach proposed by Pevzner and Waterman [167] uses a new technique to preselect similar m-tuples that allow a few number of mismatches. Buhler [33] uses locality-sensitive hashing [96] to find K-mers that differ by no more than a certain number of substitutions.