CHAPTER 10

HOMOLOGY SEARCH METHODS

Daniel G. Brown
University of Waterloo
browndg@monod.uwaterloo.ca

Ming Li
University of Waterloo
mli@pythagoras.math.uwaterloo.ca

Bin Ma
University of Western Ontario
bma@uwo.ca

Homology search methods have advanced substantially in recent years. Beginning with the elegant Needleman-Wunsch and Smith-Waterman dynamic programming techiques of the 1970s, algorithms have been developed that were appropriate for the data sets and computer systems of their times. As data sets grew, faster but less sensitive heuristic algorithms, such as FASTA and BLAST, became a dominant force in the late 1980s and 1990s. As datasets have grown still larger in the post-genome era, new technologies have appeared to address these new problems. For example, the optimal spaced seeds of PatternHunter increase speed and sensitivity. Using these ideas, we can achieve BLAST-level speed and sensitivity approaching that of slow algorithms like the Smith-Waterman, bringing us back to a full circle. We wish to take you with us on this round trip, with some detours along the way so as to study both global and local alignment. We present methods for general purpose homology that are widely adopted, not individual programs.

ORGANIZATION.

Section 1. We begin with a brief history of the development of homology search methods.

Section 2. Then we introduce the notions of edit distance, alignment, score matrix, and gap penalty.

Section 3. Next, the classic dynamic programming approach to sequence alignment is presented. We discuss both global and local alignments, as well as the issues of gaps and memory usage.

Section 4. Then we proceed to the probabilitistic issues underlying sequence alignment. In particular, we discuss in some details the PAM and BLOSUM scoring matrices and their derivations. We also briefly visit the issue of assessing the significance of an alignment.

Section 5. After that, we introduce the second generation of sequence homology search methods. These methods sacrifice sensitivity for speed by using simple seeds to index into possible matching regions before more expensive dynamic programming alignments are performed.

Section 6. Then we come to the third generation of sequence homology search methods. These methods rely on the more advanced idea of spaced seeds to achieve simultaneously the high sensitivity of the first-generation classical methods and the high speed of second-generation methods. We discuss the optimality of spaced seeds with respect to different sequence models. We also discuss the effectiveness of using multiple spaced seeds.

Section 7. Finally, we show some experiments comparing the sensivity and speed of these different generations of sequence homology search methods.

1. Overview

Two sequences are homologous if they share a common evolutionary ancestry. Unfortunately, this is a hypothesis that usually cannot be verified simply from sequence data. Therefore, our title is really a convenient misuse of terminology. Homology search methods provide evidence about homologies, rather than demonstrating their existence.

Homology search is important as its product—high scoring alignments—is used in a range of areas, from estimating evolutionary histories, to predicting functions of genes and proteins, to identifying possible drug targets. All contemporary molecular biologists use it routinely, and it is used in many of the largest supercomputing facilities worldwide. The NCBI BLAST server for homology search is queried over 100,000 times a day and this rate is growing by 10–15% per month.

The basic homology search problem is so easy that it is usually the first topic in a bioinformatics course. However, the problem is also very hard, as queries and databases grow in size, and the emphasis is on very efficient algorithms with high quality. More programs have been developed for homology search than for any other problem in bioinformatics, yet after 30 years of intensive research, key problems in this area are still wide open.

As is true of many topics in early bioinformatics, the first two important sequence alignment algorithms—the Needleman-Wunsch algorithm for global alignment and the Smith-Waterman algorithm for local alignment—were both identified during the 1970s and early 1980s by a variety of different groups of authors working in different disciplines. However, these algorithms ran in time that was too slow as databases of DNA and protein sequences grew during the 1980s. Since the mid 1980s and 1990s, heuristic algorithms—like FASTA[512] and BLAST[23]—that sacrificed sensitivity for speed became popular. These algorithms offer far faster performance, while missing some fraction of good sequence homologies. The development of good homology search software makes another advance recently, as researchers focus on the cores of alignments that are identified by heuristic search programs. This is seen in local alignment programs—such as PatternHunter[533] and BLAT[427]—that allow substantial improvement in sensitivity at minimal cost. The spaced seeds of PatternHunter, in particular, can be optimized to be highly sensitive for alignments matching a particular model of alignments,[102, 103, 118, 425, 500] which allows substantial improvement in sensitivity. In fact, one can use these spaced models to approach Smith-Waterman sensitivity at BLAST speed.[500] This chapter takes you through this round trip, stopping to note key ideas along the way, rather than specific programs.

2. Edit Distance and Alignments

2.1. *Edit Distance*

The central question in this field is whether two given DNA or protein sequences are homologous. This question cannot be answered precisely without intimate knowledge of the origin of the biological sequences, even if they are very similar in their sequence. However, the number of evolutionary mutations required to change one sequence to the other can be used to estimate the probability that the two sequences are homologous. We use a distance metric—"edit distance"—to measure the evolutionary similarity of two sequences.

When a DNA sequence is copied from a parent to a child, three common types of mutations can be introduced: substitution, insertion, and deletion. Substitution is the change of one nucleotide to another. Insertion inserts a new nucleotide to the sequence, and deletion deletes an existing nucleotide. See Figure 1.

These three operations are similar to the operations used when one edits a text file. Any sequence can be converted to any other sequence using these three edit operations, since we can change any sequence to the empty sequence by deletions, and the empty sequence to any other sequence by insertions. However, such a sequence of operations is rarely the shortest. The minimum number of operations that are required to change a sequence s to a sequence t is called the edit distance of s and t, denoted here by $d(s, t)$.

parent:	A$\underline{\texttt{C}}$GTCT	AGTCT	A$\underline{\texttt{C}}$GTCT
	$\downarrow$	$\downarrow$	$\downarrow$
child:	AGTCT	A$\underline{\texttt{C}}$GTCT	A$\underline{\texttt{G}}$GTCT
	deletion	insertion	substitution

Fig. 1. The three types of modifications.

Before we examine the algorithm that computes the actual edit distance of two given sequences, we prove that edit distance is a metric.

Theorem 1: Edit distance is a metric. That is, for any three sequences x, y, and z, $d(\cdot, \cdot)$ satisfies the following three properties:

(1) Isolation: $d(x, y) = 0$ if and only if $x = y$.
(2) Symmetry: $d(x, y) = d(y, x)$.
(3) Triangular inequality: $d(x, y) \leq d(x, z) + d(z, y)$.

Proof: For Property 1, it is trivial.

For Property 2, first note that a deletion in one sequence is equivalent to an insertion in the other sequence. Therefore, if d_1 insertions, d_2 deletions and d_3 substitutions convert x to y, then d_3 substitutions, d_2 insertions and d_1 deletions convert y to x.

For Property 3, we can transform x to z in $d(x, z)$ steps, and z to y in $d(z, y)$ steps, for any z. This gives an upper bound of $d(x, z) + d(z, y)$ steps on the edit distance from x to y. □

Another measure that is often used in the comparison of strings is the "Hamming distance". For two strings $s = s_1 s_2 \ldots s_n$ and $t = t_1 t_2 \ldots t_n$ of the same length, their Hamming distance $d_H(s, t)$ is the number of positions i, $1 \leq i \leq n$, such that $s_i \neq t_i$. Unfortunately, despite its simplicity, Hamming distance is not suitable for measuring the similarity of long genetic sequences, because of the existence of insertions and deletions.

2.2. *Optimal Alignments*

Edit distance is defined in the form of the minimum number of edit operations between two sequences, where insertions, substitutions and deletions all count as one mutation. However, edit distance is usually formulated as sequence alignment. Let $s = s_1 s_2 \ldots s_m$ and $t = t_1 t_2 \ldots t_n$ be two sequences over a finite alphabet Σ. We augment the alphabet Σ with a "space" symbol denoted by "$-$" that is not in

Σ, yielding the alignment alphabet Σ'. Any equal-length sequences S and T over the alphabet $\Sigma \cup \{-\}$ that result from inserting space characters between letters of s and t are called an alignment of the original sequences s and t.

Usually, the objective of the sequence alignment of s and t is to maximize the similarity of S and T, so that characters in the same position in both sequences are closely related or identical. The simplest goal is to minimize the Hamming distance $d_H(S, T)$. Figure 2 shows an optimal alignment of two sequences ACGTCT and AGTACG under this objective.

```
ACGT-CT
A-GTACG
```

Fig. 2. The optimal alignment of two sequences

Computing the optimal alignment to minimize Hamming distance is equivalent to computing edit distance, as shown in the following theorem:

Theorem 2: Let s and t be two sequences, and $d(s, t)$ be their edit distance. Let S and T be the optimal alignment of s and t that minimizes $d_H(S, T)$. Then $d_H(S, T) = d(s, t)$.

Proof: It is easy to see that in the edit operations that convert s to t, each insertion corresponds to a space symbol in S, each deletion corresponds to a space symbol in T, and each substitution corresponds to a mismatch between S and T. $\square$

2.3. *More Complicated Objectives*

It is possible to perform biologically meaningful sequence alignments using only the edit distance. However, typical homology search programs optimize somewhat more complicated functions of the alignments. As we will see in Section 4, there is actually a probabilistic basis to these more complicated functions. In the interim, we provide a descriptive overview here.

2.3.1. *Score Matrices*

The edit distance measure causes the same cost to be incurred for every substitution. Yet—especially for protein sequences—some mutations are far less conservative than others, and are less expected in truly homologous sequences. In these cases, a "score matrix" is used to discriminate different types of matches and mismatches. The score matrix value $M(a, b)$ is a real number that is the contribution to the alignment score of aligning a and b, for any $a, b \in \Sigma'$. Given two sequences s and t, and their alignment (S, T), the score of the alignment is defined by $\sum_{i=1}^{k} M(S[i], T[i])$. When a score matrix is used, we usually let $M(a, b) > 0$ if a and b are closely related. Therefore, the objective of the alignment is to find an alignment that maximizes the alignment score $\sum_{i=1}^{k} M(S[i], T[i])$.

In this framework, minimizing Hamming distance can be regarded as a special case, using a score matrix $M(a, a) = 0$ and $M(a, b) = -1$ for $a \neq b$.

2.3.2. *Gap Penalties*

Another important way of scoring alignments is to look at the lengths of regions consisting entirely of space characters in an alignment. In homologous sequences, such "gaps" correspond to either deletion or insertion mutations. If we use a typical scoring matrix where we score a negative constant $M(a, -)$ for aligning any letter a to a gap, the cost of the gap is proportional to its length. In both theory and practice, this is undesirable. As such, the typical response is to penalize gaps through a length-dependent gap penalty, rather than only through the scoring matrix. The most common type of gap penalty is the affine penalty, where the score of a gap of length i is $o + i \times e$. In this scheme, the gap "opening" penalty o is typically much more negative than the "extension" penalty e paid per gap position. Both costs, however, are typically negative.

3. Sequence Alignment: Dynamic Programming

3.1. *Dynamic Programming Algorithm for Sequence Alignment*

We return to the simplest case of sequence alignment, where we incorporate the costs of insertions and deletions in the score matrix M. Here, the optimal alignment of two sequences is easily computed by dynamic programming.[300,607]

Let $s = s[1]s[2] \ldots s[m]$ and $t = t[1]t[2] \ldots t[n]$. Let $DP[i, j]$ be the optimal alignment score for $s[1..i]$ and $t[1..j]$. The dynamic programming algorithm computes $DP[i, j]$ recursively, and $DP[m, n]$ ends up as the score of the optimal alignment between s and t. The actual alignment of the two sequences can be computed by a standard backtracking procedure after $DP[i, j]$ is computed for all $0 \leq i \leq m$ and $0 \leq j \leq n$.

Suppose the optimal alignment of $s[1..i]$ and $t[1..j]$ has k columns. There are three possibilities for the last column of the alignment:

(1) $s[i]$ is aligned to "$-$". In this case, the first $k - 1$ columns of the alignment should be an optimal alignment of $s[1..i - 1]$ and $t[1..j]$. Otherwise, we could

replace the first $k-1$ columns with the optimal alignment of $s[1..i-1]$ and $t[1..j]$, and get a better alignment for $s[1..i]$ and $t[1..j]$. Therefore, $DP[i,j] = DP[i-1,j] + M(s[i],-)$.

(2) $t[j]$ is aligned to "$-$". For the same reason as in Case 1, $DP[i,j] = DP[i,j-1] + M(-,t[j])$.

(3) $s[i]$ is aligned to $t[j]$. For the same reason as in Cases 1 and 2, $DP[i,j] = DP[i-1,j-1] + M(s[i],t[j])$.

By now we have almost derived a way to compute $DP[i,j]$ recursively, except that for each (i,j), we must choose the correct case, which is the case that yields the highest score. This gives the following recursive algorithm to compute $DP[i,j]$.

$$DP[i,j] = \max \begin{cases} DP[i-1,j] + M(s[i],-) \\ DP[i,j-1] + M(-,t[j]) \\ DP[i-1,j-1] + M(s[i],t[j]) \end{cases} \tag{1}$$

By definition, $DP[0,0]$ is the alignment score of two empty sequences. Therefore, $DP[0,0] = 0$. Also, $DP[i,0]$ is the alignment score between $s[1..i]$ and an empty sequence. Therefore, $DP[i,0] = \sum_{k=1}^{i} M(s[k],-)$. Similarly we can compute $DP[0,j]$.

To fill in the DP matrix, we may calculate Formula 1 for i from 1 to m and for j from 1 to n. Therefore, when we compute $DP[i,j]$, all the needed right-hand side values in Formula 1 are already known, and $DP[i,j]$ can be computed in constant time. Hence, the time complexity of this algorithm is $O(m \times n)$.

When the algorithm is implemented in a programming language, the values of $DP[i,j]$ are usually stored in a two dimensional array, as shown in Figure 3. The first row and column of the array must be initialized. For each of the other entries, its value is determined by the three adjacent entries—*viz.* the upper, left, upper left entries. The arrows in Figure 3 illustrate that the three entries determine the value of the last entry.

After the DP table is filled, the value of $DP[m,n]$ is the score of the optimal alignment of the two sequences. In constant time, we can determine which of the three cases maximizes the value in Formula 1, and thus determine the last column of the alignment. For example, in Figure 3, it is the third case that maximizes the value of the last entry. Therefore, the last column of the alignment is $s[m]$ matching $t[n]$, preceded by the optimal alignment of $s[1..m-1]$ and $t[1..n-1]$. These preceding columns can be identified by examining $DP[m-1,n-1]$, and so on. Eventually, we trace back to $DP[0,0]$ and get the optimal alignment of s and t. This procedure is called backtracking. Figure 4 shows the backtracking

		A	C	G	T	C	T
	0	–1	–2	–3	–4	–5	–6
A	–1	1	0	–1	–2	–3	–4
G	–2	0	0	1	0	–1	–2
T	–3	–1	–1	0	2	1	0
A	–4	–2	–2	–1	1	1	0
C	–5	–3	–1	–2	0	2	1
G	–6	–4	–2	0	–1	1	1

Fig. 3. The DP table for the alignment of two sequences ACGTCT and AGTACG.

through the table in Figure 3, giving the alignment of Figure 2.

During backtracking, it is possible that for an entry, more than one of the three cases give the entry its value. If we want all possible optimal alignments, we must examine all adjacent entries that give this optimal value. Therefore, there are potentially an exponential number of backtracking paths, each corresponding to an optimal alignment with the same value. If we only need one optimal alignment, we can choose an arbitrary one that maximizes the value. Because each step of the backtracking reduces either i or j or both by one, the total number of steps does not exceed $m + n$. Thus, most computation time in computing a single optimal alignment is spent filling the DP table.

3.1.1. *Reducing Memory Needs*

The algorithm we have given requires $O(m \times n)$ space to store the entire DP matrix. However, if all that is desired is the score of the optimal alignment, this can be computed in $O(n)$ space by only keeping track of two columns or rows of the DP matrix—since when computing a row, we need only to know the value of the previous row.

A slightly more complicated trick[354] allows the computation of the optimal alignment in $O(n)$ space as well. Here, we keep track of the most recent two rows of the matrix, and after we have computed row $m/2$ of the matrix, we also remember which cell in the $m/2$ row the alignment path that is optimal in the current cell passed through last. When we compute this for the $[m, n]$ entry of the

		A	C	G	T	C	T
	0	−1	−2	−3	−4	−5	−6
A	−1	1	0	−1	−2	−3	−4
G	−2	0	0	1	0	−1	−2
T	−3	−1	−1	0	2	1	0
A	−4	−2	−2	−1	1	1	0
C	−5	−3	−1	−2	0	2	1
G	−6	−4	−2	0	−1	1	1

Fig. 4. Backtracking.

DP matrix, this allows us to divide the problem in half and recurse on smaller problems, which preserves both the $O(m \times n)$ runtime and the $O(n)$ space.

Since these heuristics work fine for columns as well, they can actually be used to produce optimal alignments in $O(\min(m, n))$ space.

3.2. *Local Alignment*

The dynamic programming algorithm given in Section 3.1 gives optimal alignments of complete sequences. However, we often want to find closely homologous parts of the two sequences. For example, a protein usually consists of a few domains. Two proteins may have one domain in common, but be otherwise unrelated. In this case, we may want to find the similar domains. Identifying these regions of similarity, such that the alignment score of the two local regions are maximized, is called "local alignment". In contrast, the alignment we described in Section 3.1 is sometimes called "global alignment".

One way to consider local alignment is to think that eliminating prefixes and suffixes of the two sequences is free. Therefore, if the global alignment has negative scores at its either end, we can eliminate the end to get a better local alignment. Consequently, if the same scoring matrix is used, the local alignment score of two sequences is always higher than or equal to the global alignment score. However, the optimal local alignment may not be a part of the global alignment. For example, the optimal local alignment of the two sequences in Figure 2 is the substring ACG of the first sequence aligned with the substring ACG of the second sequence, which is not a part of the alignment shown in Figure 2.

To compute optimal local alignments, we must make two modifications to the algorithm for global alignments, to deal with prefixes and suffixes respectively. First, we note that to find the optimal alignments that can eliminate suffixes of the strings, we are actually seeking the highest scoring alignment of two prefixes of s and t. Recall that $DP[i,j]$ is the score of the optimal alignment between $s[1..i]$ and $t[1..j]$. We must thus examine the entire DP table, and find the (i,j) pair that maximizes $DP[i,j]$. We then do the backtracking for the generation of the actual alignments of $s[1..i]$ and $t[1..j]$ from entry $DP[i,j]$.

Next, we consider eliminating prefixes. For the substrings $s[1..i]$ and $t[1..j]$, let

$$DP'[i,j] = \max_{\substack{i' \leq i+1, \\ j' \leq j+1}} \{\text{optimal alignment score for } s[i' \ldots i] \text{ and } t[j' \ldots j]\}$$

That is, $DP'[i,j]$ is the maximum possible alignment score of two suffixes of $s[1..i]$ and $t[1..j]$. It is possible that both suffixes are empty; the value then is just zero.

Similar to the computation of $DP[i,j]$ in a global alignment, we examine the last column of the optimal alignment of the two optimal suffixes of $s[1..i]$ and $t[1..j]$. The three cases in Section 3.1 still exist. However, there is one more case where both suffixes are empty strings, and $DP'[i,j] = 0$. This gives a new recursive formula for the score:

$$DP'[i,j] = \max \begin{cases} DP'[i-1,j] + M(s[i],-) \\ DP'[i,j-1] + M(-,t[j]) \\ DP'[i-1,j-1] + M(s[i],t[j]) \\ 0 \end{cases} \tag{2}$$

Combining the two modifications, we may fill the DP' matrix using Formula 2. We require that $DP'[0,i] = DP'[j,0] = 0$ for all values of i and j. Lastly, we search the matrix to find the highest value, and backtrace from there until we reach a position where $DP[i,j] = 0$, when we stop.

This algorithm is typically called the Smith-Waterman algorithm,[780] in honor of the authors of one of the original papers introducing it. Its runtime and space complexity are both $O(m \times n)$, and the space complexity can be reduced to $O(\min(m,n))$ using methods analogous to those given in Section 3.1.1.

Exercise: Given two sequences s and t, give an algorithm that identifies a substring t' of t that maximizes the optimal global alignment score of s and t'. This type of alignment is called "fit alignment".

3.3. *Sequence Alignment with Gap Open Penalty*

We now move to consider optimal global alignments when there are specific score penalties imposed for opening gaps, as in Section 2.3.2. Let M be a score matrix, and $o \leq 0$ be the gap open penalty. We also assume that the score $M(a, -)$ of aligning any letter $a \in \Sigma$ to a "$-$" symbol is a constant negative value e.

The dynamic programming algorithm in Section 3.1 cannot be straightforwardly used here to compute the optimal alignment with gap open penalty. The problem is in the first two cases in Formula 1. If the "$-$" symbol is the first one in a gap, then we are opening a gap and should add the gap open penalty o to the score. Otherwise, we should not. The DP table does not provide enough information to differentiate these cases.

To solve this problem, we use three different DP tables—$DP_1[i,j]$, $DP_2[i,j]$ and $DP_3[i,j]$—to record the optimal alignment score of the alignments whose last column is respectively

(1) $s[i]$ matches "$-$",
(2) "$-$" matches $t[j]$, and
(3) $s[i]$ matches $t[j]$.

Consider these three DP tables, using the computation of $DP_1[i,j]$ as an example. The last column must be $s[i]$ matching "$-$". The preceding column can be any of the three cases:

(1) $s[i-1]$ matches "$-$", and therefore the gap in the last column is not a gap opening. Thus, $DP_1[i,j] = DP_1[i-1,j] + e$.
(2) "$-$" matches $t[j]$, and therefore the gap in the last column is a gap opening. Thus, $DP_1[i,j] = DP_2[i-1,j] + e + o$.
(3) $s[i-1]$ matches $t[j]$, and therefore the gap in the last column is a gap opening. Thus, $DP_1[i,j] = DP_3[i-1,j] + e + o$.

Because $DP_1[i,j]$ is the best of these three possibilities, we have:

$$DP_1[i,j] = e + \max \begin{cases} DP_1[i-1,j] \\ DP_2[i-1,j] + o \\ DP_3[i-1,j] + o \end{cases}$$

We make a special note that Case 2 indicates that in the alignment, a column of aligning $s[i]$ to "$-$" is followed immediately by a column of aligning "$-$" to $t[j]$. The two columns can obviously be replaced by a column of aligning $s[i]$ to $t[j]$. For many scoring matrices M, the latter alignment is better. Therefore, usually we can discard Case 2 in the computation.

Similarly, we can fill the other two DP tables with the following two formulas:

$$DP_2[i,j] = e + \max \begin{cases} DP_1[i,j-1] + o \\ DP_2[i,j-1] \\ DP_3[i,j-1] + o \end{cases}$$

$$DP_3[i,j] = M(s[i],t[j]) + \max \begin{cases} DP_1[i-1,j-1] \\ DP_2[i-1,j-1] \\ DP_3[i-1,j-1] \end{cases}$$

After appropriate initialization of the three DP tables, we can compute all three tables row-by-row. Then the maximum value of $DP_1[m,n]$, $DP_2[m,n]$, and $DP_3[m,n]$ is the optimal alignment score with gap open penalty. Which of the three is the maximum determines which of the three cases is the last column of optimal alignment. A backtracking from that table entry to $DP_3[0,0]$ gives the actual optimal alignment.

Similarly to Section 3.2, one can define the local alignment with gap open penalty. The extension from global alignment with gap open penalty to local alignment with gap open penalty is similar to what we have done in Section 3.2.

Exercise: Similar to Section 3.2, extend the algorithm for global alignment with gap open penalty to local alignment with gap open penalty.

4. Probabilistic Approaches to Sequence Alignment

Underlying the simple dynamic programming algorithms of the previous sections is a rich mathematical theory.[413, 414, 579] In particular, the score of an alignment can be seen as a measure of the surprisingness of that alignment, given probabilistic models for related and unrelated sequences. The richest part of this theory—which we do not visit in this tour—estimates the probability that an alignment with a particular score or higher would occur in random sequences, to allow researchers to estimate the statistical significance of an alignment.

We begin by considering how to pick the members of a scoring matrix.

4.1. *Scoring Matrices*

What values should be used in a scoring matrix? Intuitively, the value $M[i,j]$ should be more positive for more closely related symbols. But how do we quantify this?

For DNA sequences, the scoring matrix is often very simple. For example, the default score matrix used in the DNA homology search program BLASTN

	A	C	G	T
A	91	–114	–31	–123
C	–114	100	–125	–31
G	–31	–125	100	–114
T	–123	–31	–114	91

Fig. 5. Blastz score matrix.

is simple. Matches between the same base score 1, mismatches between bases score -3, and matches between a "$-$" symbol and a base score -1. BLAST also includes a gap open penalty, which by default is -5, for the alignments it generates. Other programs—such as Blastz—use a more carefully investigated scoring scheme, proposed in Chiaromonte*et al.*[156] and shown in Figure 5 with gap open penalty -400 and gap extension -30. This matrix comes from a similar origin to the probabilistic explanation given below for protein alignments.

For protein sequences, a more complicated approach is used, based on alignments of sequences known to be homologous. Two complementary approaches are found in PAM matrices[192] (Point Accepted Mutation, or Percent Accepted Mutation) and BLOSUM matrices[346] (BLOcks SUbstitution Matrices). But both use fundamentally similar ideas.

The central idea is to have a scoring matrix where higher-scoring entries are more likely to be aligned in homologous sequences than in random sequences. To encapsulate this, entries in both PAM and BLOSUM matrices are related to the logarithm of the odds ratio—also known as "log-odds"—that a given pair of symbols are aligned due to homology versus by chance.

For two amino acids i and j, let p_i and p_j be the probabilities of amino acids i and j occurring at a random position of a random protein sequence, respectively. Then $p_i \times p_j$ is the probability that i is aligned to j in a random position in the alignment of two unrelated sequences. Let q_{ij} be the probability that a column of a true alignment of two related sequences aligns i to j. The odds ratio that an i matched to a j comes from a true alignment versus a random alignment is thus $q_{ij}/(p_i \times p_j)$, and its logarithm is $\log(q_{ij}/(p_i \times p_j))$.

4.1.1. *PAM Matrices*

PAM matrices—also called Dayhoff matrices in honor of their inventor—are based on a very simple model of protein evolution, in which amino acid sequences evolve by a series of independent single substitution events. PAM matrices describe the frequency with which amino acids "safely" mutate to other amino acids. Because different biomolecules may mutate at different rates, the PAM unit—instead of the time—is used to measure the amount of variation.

One PAM of mutation describes an amount of evolution which changes, on the average, 1% of the amino acids. The PAM-1 mutation matrix has been calculated by Dayhoff*et al.*,[192] by examining the mutations that separate closely related protein sequences. Here we denote the PAM-1 mutation matrix by Q, where $Q[i,j]$ is the probability that a given amino acid i is mutated to j after one PAM of mutations have occurred. Given the assumption that evolution acts as independent mutation events, it is easy to prove that if n PAMs of mutations have occured, the mutation matrix PAM-n is equal to Q^n, the n-th power of the PAM-1 matrix.

After computing the PAM-n matrix Q_n, we usually use it in practice as a log-odds scoring matrix. To do this, we need the probability p_i with which amino acid i appears in a random position of random protein sequences. In a random alignment, i and j are aligned with probability $p_i \times p_j$. But in a random position of an alignment of two sequences separated by n PAMs of mutation, the probability that i and j are aligned is $p_i \times Q_n[i,j]$. Given these two possible hypotheses for why i is aligned to j, the odds ratio is just

$$M_n[i,j] = \frac{p_i \times Q_n[i,j]}{p_i \times p_j} = \frac{Q_n[i,j]}{p_j}$$

This is then converted to a score matrix by taking the logarithm in base 10 and multiplying by 10:

$$M'_n[i,j] = 10 \times \log_{10} \frac{Q_n[i,j]}{p_j}$$

If i is aligned to j more frequently at a random column of the alignment of two homologous proteins than that of two random proteins, $q_{ij} > p_i \times p_j$ and $M'_n[i,j] > 0$. Otherwise, $M'_n[i,j] \leq 0$.

4.1.2. *BLOSUM Matrices*

An alternative scoring approach for protein sequences is the BLOSUM matrices. Recall that PAM matrices are derived from alignments of closely related sequences. By contrast, the BLOSUM matrices invented by Henikoff and Henikoff[346, 347] are derived from contiguous segments of multiple alignments of sequences that are not known to be closely related or even homologous in an evolutionary sense.

The database used is the BLOCKS database, which identifies protein motifs from multiple alignments of amino acid sequences. The database includes thousands of blocks, where each block contains many sequence segments that are similar with each other. For the computation of BLOSUM n matrix, the segments that are identical at $n\%$ or more of the positions within a block are clustered and weighted as a single sequence. This reduces the multiple contributions to amino acid pairs from the most closely related members of a protein family. From these blocks, one can then identify how often two amino acids i and j are aligned to each other in these blocks, versus how often they appear unrelatedly in the BLOCKS database.

If i is aligned to j a total of $a(i,j)$ times in the blocks, out of a total of P aligned pairs in blocks, then the probability of i being aligned to j in a random aligned pair is clearly $a(i,j)/P$. By contrast if the probability of that i occurs at a random position in the database is p_i and the probability of that j occurs is p_j, then the probability of aligning i to j in unrelated sequences is just $p_i \times p_j$. The odds ratio is therefore

$$B[i,j] = \frac{a(i,j)}{p_i \times p_j \times P}$$

Typically, BLOSUM matrices result from doubling the logarithm in base 2 of this B matrix. As such, they are again log-odds matrices.

Reducing the value of n allows the BLOSUM n matrix to be biased toward finding homologies between sequences that are not very similar, ideally allowing their use in searching for homologies that are not very strong. In practice, BLOSUM matrices like BLOSUM 62 or BLOSUM 50 have largely supplanted the use of PAM matrices.

4.1.3. *Weaknesses of this Approach*

A serious issue with these scoring methods is that they are based on encapsulating statistics from existing alignments. In particular, they can be inappropriate in estimating the quality of an alignment between membrane proteins, given that the alignments that are used to develop both PAM and BLOSUM matrices are of largely globular proteins.

4.2. *Probabilistic Alignment Significance*

In addition to the log-odds interpretation of protein scoring matrices, a similar approach can be used to interpret alignment scores. We make a completely unjustified assumption: that positions of an alignment are all independent of one another. Now, consider an alignment A between two unrelated sequences s and t, in a scoring system with no added penalties for opening gaps. If the probability of a symbol i occurring is p_i in a random aligned sequence, then the probability of alignment A appearing randomly is just $\prod_i p_{s_i} \times p_{t_i}$. If, by contrast, the probability that i is aligned to j in sequence that matches a model of homologous sequence is q_{ij}, then the probability of that alignment occuring at random is $\prod_i q_{s_i t_i}$. The odds ratio of these probabilities is just

$$\prod_i \frac{q_{s_i t_i}}{p_{s_i} \times p_{t_i}}$$

If the alignment is scored with a log-odds scoring matrix M, where $M[i, j]$ is the logarithm of the ratio between the probability that i is aligned to j in the known homologous sequences versus at random, then the score of the alignment is just

$$\sum_i \log \frac{q_{s_i t_i}}{p_{s_i} \times p_{t_i}}$$

which is exactly the logarithm of the odds ratio for the alignment.

How do we use this observation? There are two common approaches, from Bayesian and frequentist statistics. In the Bayesian approach, we assess the probability that the alignment is of homologous sequences, given a prior estimate of that probability, using the odds ratio. In the more complicated frequentist approach,[413,414] the probability that a local alignment with score greater than x would appear in unrelated sequences of length $|s|$ and $|t|$ is estimated, and this is used as a measure of the statistical significance of the alignment. This can be extended to consider the sequence content of s and t as well in the estimation.

5. Second Generation Homology Search: Heuristics

Filling in an entire dynamic programming matrix when aligning two sequences is quite time consuming. As a result, in the late 1980s and early 1990s, obvious heuristic methods were proposed. These methods share a common theme: sacrifice sensitivity for speed. That is, they run much faster than full dynamic programming, but they may miss some alignments. The two most popular heuristics are found in FASTA[512] and BLAST.[23] To focus on central ideas, we concentrate on DNA sequence homology search. Protein sequence homology search involves similar strategies.

5.1. *FASTA and BLAST*

One of the earliest of these heuristics is FASTA. FASTA uses a hashing approach to find all matching k-tuples (between 4 and 6 for DNA), between the query and database. Then nearby k-tuples, separated by a constant distance in both sequences, are joined into a short local alignment. With these short local alignments as seeds, Smith-Waterman dynamic programming is applied to larger gaps between two high scoring pairs still separated by short distances, with the restriction that only the part of the dynamic programming matrix nearest the diagonal is filled in. FASTA outputs only one alignment per query sequence, after the dynamic programming phase, and estimates the probability of the alignment occurring by chance.

More popular has been the BLAST family of heuristics. BLAST works similarly at its beginning, identifying seed matches of length k (= 9–11 for DNA). Each seed match is extended to both sides until a drop-off score is reached. Along the way, seed matches that are being extended in ways that are not typical of truly homologous sequences are also thrown out. BLAST can be set so that two nonoverlapping seed matches may be required before alignments are extended. Newer versions of BLAST[24] allow gapped alignments to be built. BLAST outputs all alignments found, and estimates for each alignment the expected number of alignments of unrelated sequences whose score would be as large. This quantity is estimated in ways described in Section 4.2.

5.2. *Large-Scale Global Alignment*

Another topic of great interest in the last years has been the identification of global alignments between long stretches of sequenced genomes. Beginning with the work of Delcher*et al.*,[194] a number of other authors have begun to use somewhat different methods than classical dynamic programming ones for this problem.

This problem is of interest partially because portions of these global alignments in some genomes have undergone enough mutations that their homology is no longer significant, yet they may have regulatory or other significant functions that are still conserved. At the same time, researchers identifying homologous blocks that have not been broken by major genome rearrangement events may want to use the non-exonic regions as a tag to identify which of two duplicated blocks is truly homologous to another sequence. As such, the whole-genome alignment problem has been the subject of considerable research in the last few years.

Two methodologies in particular augment and complement the techniques used in heuristic local alignment. The first is the use of seed alignments; but instead of using them to build local alignments, whole genome alignment packages typically use them as anchors that are required to be used in the chosen alignment. The second is the use of variable-length regions as these anchors—most often accomplished by the use of suffix trees or similar data structures, which are used to identify long completely conserved regions as anchors.

6. Next-Generation Homology Search Software

In the post-genome era, supercomputers and specialized hardware implementing sequence alignment methods in digital logic are employed to meet the ever expanding needs of researchers. Pharmaceutical corporations and large scientific funding agencies proudly spend much money to support such supercomputing centers. Unfortunately, the reliability of these solutions must be considered in light of the consistent doubling of sequence databases, as GenBank doubles in size every 18 months.[606]

In the late 1990s, however, several methods have been developed that improve the sensitivity of homology search software to a level comparable to that of full-scale dynamic programming, while avoiding very large runtime complexities. These have largely focused on characterizing the central seeds from which heuristic alignment programs build their local alignments.

6.1. *Improved Alignment Seeds*

BLAST-like heuristics first find short seed matches which are then extended. This technique faces one key problem: As seeds grow longer, we can expect fewer homologies to have the large conserved regions. However, shorter seeds yield many random hits that significantly slow down the computation.

To resolve this problem, a novel seeding scheme has been introduced in PatternHunter.[533] BLAST looks for matches of k—default $k = 11$ in BLASTN and $k = 28$ in MegaBlast—consecutive letters as seeds. PatternHunter instead uses *non*-consecutive k letters as seeds. The relative positions of the k letters is called a "spaced seed model", and k its "weight". For convenience, we denote a model by a 0-1 string, where ones represent required matches and zeros represent "don't care" positions. For example, if we use the weight 6 model 1110111, then the alignment **ACTGCCT** versus **ACTTCCT** matches the seed, as does **ACTGCCT** versus **ACTGCCT**. In this framework, BLAST can be thought of as using models of the form $111\ldots1$.

Let L be the length of a homologous region with no indels, and M be the length of a seed model. Then there are $L - M + 1$ positions that the region may contain a hit; see Figure 6. In a BLAST type of approach, as long as there is one hit in such a region, the region can be detected. Therefore, although the hit probability at a specific position is usually low, the probability that a long region contains a hit can be reasonably high.

```
   TACTGCCTG
   |||| ||||
   TACTACCTG
1: 1110101
2:  1110101
3:   1110101
```

Fig. 6. There are many positions that a homology may contain a hit. In this figure, the seed model 1110101 hits the region at the second position.

Ma, Tromp, and Li[533] notice that different seed models with identical weight can result in very different probabilities to hit a random homology. For a seed with weight W, the fewer zeros it has, the shorter the seed is, and the more positions it can hit the region at. Therefore, intuitively, BLAST's seed model with W consecutive ones seems to have the highest hit probability among all the weight-W seed models. Quite surprisingly, this is not true. The reason is that the hits at different positions of a region are not independent. For example, using BLAST's seed, if a hit at position i is known, the chance to have a second hit at position $i + 1$ is then very high because it requires only one extra base match. The high dependency between the hits at different positions make the detection of homologies "less efficient".

The same authors observe that the dependency can be reduced by adding some zeros into the seed model. For example, if seed model 1110101 is used and there is a hit at position i, then the hit at position $i+1$ requires three extra base matches, compared to one extra base match of the BLAST's seed. Thus, hits at different positions are less dependent when spaced seed models are used. On the other hand, spaced seed models are longer than the consecutive seed model with the same weight, and therefore have fewer positions to hit a region at. As a result, the optimal seed must balance these two factors. In the same paper, the authors have developed a method to find the optimal seed model that maximizes the hit probability in a simple model, and the optimal seeds are then used to develop the PatternHunter program.

Some other seeding or related strategies have also been developed before or after PatternHunter's spaced seed model. In the program WABA,[428] Kent proposes the use of a simple pattern in identifying homologous coding sequences. Since these sequences often vary in the third, "wobble", position of codons, WABA ignores these positions when identifying positions that match a seed. In the framework of PatternHunter, this is equivalent to using spaced seeds of form 110110 . . ..

Kent's approach takes advantage of the special properties of the coding region homologies. Kent has also introduced a different approach for detecting noncoding region homologies in his program BLAT.[427] BLAT uses consecutive seeds, but allows one or two mismatches to occur in any positions of the seed. For example, a BLAT hit might require at least ten matches in twelve consecutive positions. This scheme naturally allows more false negatives, but the resultant collection of hits is more enriched for true positives at a given level of false positives than for consecutive seeds where all positions are required to match.

In a random hashing strategy, Buhler[117] uses his experience with identifying sequence motifs using random projection[119] to speed up detection of homology search. This idea is previously used by Indyk and Motwani.[382] Basically, this approach is to find all hits by random hashing over long sequence intervals. A simple probability calculation allows the computation of how many projections are required to ensure a given probability that a homologous alignment has a hit to at least one of these random projections. For good choices of projection weights, this approaches 100% sensitivity. Other than the fact that high-weight random projections are not suitable and not designed for BLAST-type searches, this approach also ignores the possibility of optimizing the choice of those projections.

Of these first three approaches—specific spaced seeds, consecutive seeds allowing a fixed number of mismatches, and random spaced seeds—the first, typified by PatternHunter, allows for optimization. That is, one can use a seed specifically tuned for the types of alignments one expects to see, with the highest sensitivity at a particular false positive rate.

The optimal seed models in the PatternHunter paper[533] are optimized for noncoding regions. Later, Brejová, Brown, and Vinař[102] develop an algorithm for optimizing the seeds in more complicated models, specifically for coding regions. In a later paper,[103] they also propose a unified framework to represent all of the above mentioned seeding methods. In their framework, the seeds are represented by a pair (v, T): a vector v that represents the seed sequence—*e.g.*, the zeros and ones from PatternHunter seeds—and a threshold T that in its simplest case identifies how many "one" positions from the seed vector v must be matching in the two sequences to yield a hit.

6.2. *Optimized Spaced Seeds and Why They Are Better*

Optimized spaced seeds can have substantially greater sensitivity than the consecutive seed models of BLAST. Here we give one example. The simplest model of an alignment is of a region of a fixed length where each position matches with some probability p, independent of all other positions. Figure 7 compares the optimal spaced seed model of weight 11 and length at most 18—*viz.*

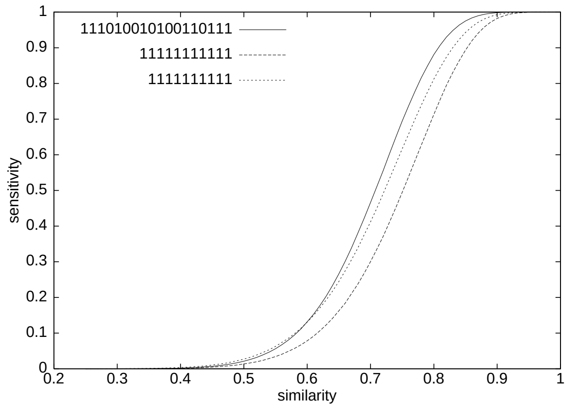

Fig. 7. 1-hit performance of weight 11 spaced model versus weight 11 and 10 consecutive models, coordinates in logarithmic scale.

111010010100110111—with BLAST's consecutive models of weight 11 and 10, for alignments of this type, of fixed length 64. For each similarity rate p shown on the x-axis, the fraction of regions with at least 1 hit is plotted on the y-axis as the sensitivity for that similarity rate. From the figure, one observes that the seemingly trivial change in the seed model significantly increases sensitivity. At 70% homology level, the spaced seed has over 50% higher probability—at 0.47—to have a hit in the region than BLAST weight 11 seed—at probability 0.3.

However, the added sensitivity does not come at the cost of more false positive hits or more hits inside true alignments:

Lemma 3: *The expected number of hits of a weight-W length-M model within a length L region of similarity* $0 \leq p \leq 1$ *is* $(L - M + 1) \times p^W$.

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. □

Lemma 3 reveals that spaced seeds have fewer expected hits, but have higher probability to hit a homologous region, as shown in Figure 7. This is a bit counter intuitive. The reason is that a consecutive seed often generates multiple hits in a region, because a hit at position i increases the hit probability at position $i + 1$ to p as only one extra base match is required. However, optimized spaced seeds are less likely to have multiple hits in a region because the second hit requires more base matches. Therefore, given many homologous regions, although the total number of hits generated by a spaced seed is comparable to the number for a consecutive seed with the same weight, the spaced seed hits can cover more regions.

To quantify this, let $p = 0.7$ and $L = 64$, as for the original PatternHunter model. Given that the BLASTN seed 11111111111 matches a region, the expected number of hits in that region is 3.56, while the expected number of hits to the spaced seed 101101100111001011, given that there is at least one, is just 2.05.

Thus, using an optimized spaced seed, a homology search program increases sensitivity but not running time. Inverting the above reasoning, we can use an optimal weight-12 spaced seed to achieve the BLAST weight 11 seed sensitivity, but generating four times fewer hits. This speeds up the search process by roughly a factor of four.

6.3. *Computing Optimal Spaced Seeds*

The probability of a seed generating a hit in a fixed length region of a given level similarity can be computed by dynamic programming[102, 103, 118, 158, 425, 500, 533] under various assumptions. To choose an optimal seed, we compute the hit probability for all seeds, and pick the one with the highest probability.

Suppose we are given a seed s, of length M and weight W, and a homology region R, of length L and homology level p, with all positions independent of each other. In this model, we can compute the probability of s having a hit in R. We represent R by a random string of zeros and ones, where each position has probability p of being a one.

We say that seed s has a seed match to R at location i if the L-length substring of R starting at position i has a one in each position with a one in the seed s. Let A_i be the event that seed s has a seed match at location i in R, for all $0 \leq i \leq L-M$. Our goal is to find the probability that s hits R : $\Pr[\cup_{i=0}^{L-M} A_i]$.

For any $M \leq i \leq L$ and any binary string b such that $|b| = M$, we use $f(i, b)$ to denote the probability that s hits the length i prefix of R that ends with b:

$$f(i,b) = \Pr[\cup_{j=0}^{i-M} A_j \mid R[i-l, \ldots, i-1] = b].$$

In this framework, if s matches b,

$$f(i,b) = 1.$$

Otherwise, we have the recursive relationship:

$$f(i,b) = (1-p) \times f(i-1, 0b') + p \times f(i-1, 1b'),$$

where b' is b deleting the last bit.

Once we have used this dynamic programming algorithm to compute $f(L - M, b)$ for all strings b, we can compute the probability of s hitting the region. It is:

$$\sum_{|b|=M} \Pr[R[i-1,\ldots,i-1] = b] \times f(L-M, b)$$

This is the simplest algorithm, used in the original paper[533] but not published anywhere, to compute the hit probability of a seed. Other algorithms generalize this algorithm to more sophisticated probabilistic assumptions of the region R and improving time complexity.[102, 118, 158, 425, 500]

6.4. *Computing More Realistic Spaced Seeds*

The simple model of homologous regions described above is not very representative of real alignments. Homologous regions are not all of length 64, and vary internally in how conserved they are. Of course, they also include gaps, but we are not going to consider this in producing seeds.

For example, more than 50% of significant alignments in the human and mouse genomes are between exonic regions, and these regions have much more conservation in the first two positions of a codon than in the third, which has traditionally been called the "wobble" position.[428] A seed that takes advantage of this periodicity by ignoring the third position in codons is much more likely to hit than a seed that does not. There is also substantial dependence within a codon: If the second position is not matched, it is quite likely that neither are the first or third.

Similarly, real alignments vary substantially internally in their alignment as well. This is particularly true for coding alignments. Such alignments tend to have core regions with high fidelity, surrounded by less well-conserved regions.

Models that do not account for these variabilities can substantially underestimate the hit probability of a seed.

6.4.1. *Optimal Seeds for Coding Regions*

Two recent papers try to address this need to optimize better models in different ways. Brejová *et al.*[102] use a Hidden Markov Model (HMM) to represent the conservation pattern in a sequence. Their model accounts for both internal dependencies within codons, and also for multiple levels of conservation throughout a protein. They update the dynamic programming model above to this new framework, and show that one can still relatively efficiently compute the probability that a given seed matches a homologous region. Meanwhile, Buhler *et al.*[118] represent the sequences by Markov chains, and present a different algorithm, based on finite automata to compute the probability of a seed hit in that model.

The difference in using seeds tuned to find hits in homologous coding regions versus seeds for general models is quite large. In particular, the optimal seed—111001001001010111—of weight 10 and length at most 18 for noncoding regions is ranked 10,350 among the 24,310 possible seeds in its theoretical sensitivity in the HMM trained by Brejová*et al.*, and ranked 11,258 among these seeds in actual sensitivity on a test set of coding region alignments, matching just 58.5% of them. By contrast, the three optimal coding region seeds—which are also optimal for the theoretical hidden Markov model—match between 84.3% and 85.5% of alignments. These seeds—11011011000011011, 11011000011011011, and 11000011011011011—all ignore the third positions of codons, and also skip entire codons. As such, they model the conservation pattern of real coding sequences much better than the non-periodic seeds optimized for noncoding regions.

Much of the advantage does of course come from the modeling. A previous program, WABA[428] uses three-periodic seeds of the form 110110110... with no formal justification. In practice, this seed has sensitivity close to the optimum, at 81.4%. Still, the optimal seeds give a good improvement over this seed, and also allow the optimization of multiple seeds for still greater sensitivity.

For the homology search in coding regions, an alternative approach is the use of a "translated homology search" program—*e.g.*, tblastx. Such a program first translates DNA sequences to protein sequences, from which the homologies are then found. The translated homology search is supposed to be more sensitive than a DNA-based program for coding regions, however, is substantially slower.

Kisman, Ma, and Li[434] have recently extended the spaced seed idea to translated homology search and developed tPatternHunter, which is both faster and more sensitive than tblastx.

6.4.2. *Optimal Seeds for Variable-Length Regions*

As to the length of regions, it is quite simple for all of these algorithms to incorporate distributions on the length of homologous regions into the model. For a given generative model, we simply compute the probability $\sigma(\ell)$ of a hit in a region of length ℓ, and the distribution $\pi(\ell)$ of lengths of the region; the probability of a hit in a random region is then just $\sum_\ell \sigma(\ell) \times \pi(\ell)$.

6.5. *Approaching Smith-Waterman Sensitivity Using Multiple Seed Models*

Another idea that comes directly from the idea of optimal spaced seeds is the one of using multiple seed models, which together optimize the sensitivity. In such an approach, a set of several seed models are selected first. Then all the hits generated by all the seed models are examined to produce local alignments. This obviously increases the sensitivity because more hits than using one seed model are examined. But now, the several seed models need to be optimized together to maximize the sensitivity.

This idea appears in the PatternHunter paper [533] and is further explored in several recent papers.[103, 118, 500] Brejová*et al.*[103] use a heuristic method to design a good pair of seeds. Buhler*et al.*[118] use hill-climbing to locally improve good sets of seeds and a pair of seeds have been designed by their method. Li*et al.*[500] extend the dynamic programming algorithm in Section 6.3 to compute a suboptimal set of seeds greedily.

Li*et al.*[500] show that, in practice, doubling the number of seeds can achieve better sensitivity than reducing the weight of the seeds by one. However, for DNA homology search, the former only approximately doubles the number of hits, whereas the latter increases the number of hits by a factor of four (the size of DNA alphabet). Thus, multiple seeds are a better choice.

It is noteworthy that the multiple-seed approach is only possible when spaced seeds are used—there is only one BLAST-type of consecutive seed with a given weight. The newest version of PatternHunter implements the multiple-seed scheme,[500] having greedily chosen a set of sixteen seeds of weight 11 and length at most 21 in 12 CPU days on a Pentium IV 3GHz PC. When the random region has length 64 and similarity 70%, the first four seeds are: 111010010100110111, 111100110010100001011, 110100001100010101111, 1110111010001111. Multiple seeds for coding regions are also computed and implemented. The experimental results shown later in Section 7 demonstrate that using carefully selected multiple seeds can approach Smith-Waterman sensitivity at BLAST's speed.

6.6. *Complexity of Computing Spaced Seeds*

Many authors[102, 103, 118, 158, 425, 533] have proposed heuristic or exponential time algorithms for the general seed selection problem: Find one or several optimal spaced seeds so that a maximum number of target regions are each hit by at least one seed. A seemingly simpler problem is to compute the hit probability of k given seeds. Unfortunately, these are all NP-hard problems.[500] Thus the greedy algorithm and the exponential time dynamic programming are the best we can do. Although the proofs are beyond the scope of this tutorial, we to list some of the recent results for these problems. Let $f(n)$ be the maximum number of 0's in each seed, where n is the seed length, the following are true.[500]

(1) If $f(n) = O(\log n)$, then there is a dynamic programming algorithm that computes the hit probability of k seeds in polynomial time; otherwise the

problem is NP-hard.

(2) If $f(n) = O(1)$, one or a constant number of optimal seeds can be computed in polynomial time by enumerating all seed combinations and computing their probabilities; otherwise, even selecting one optimal seed is NP-hard.

(3) If $f(n) = O(1)$, then the greedy algorithm of picking k seeds by enumeration, then adding the seed that most improves the first seed, and so on, approximates the optimal solution within ratio $1 - \frac{1}{\mathsf{e}}$ in polynomial time, due to the bound for the greedy algorithm for the maximum coverage problem;[359] otherwise the problem cannot be approximated within ratio $1 - \frac{1}{\mathsf{e}} + \epsilon$ for any $\epsilon > 0$, unless $NP = P$.

7. Experiments

Here, we present some experimental comparisons between heuristic search techniques and full Smith-Waterman dynamic programming. As expected, Smith-Waterman dynamic programming is too slow for practical use when the database is large. What is striking is that a good choice of multiple optimal spaced seeds can allow near-total success in detecting alignments, with vastly better runtime.

The results that we show in this section are originally reported by Li*et al.*[500] In the paper, several software packages—SSearch,[654] BLAST, and PatternHunter—are used to find homologies between 29715 mouse EST sequences and 4407 human EST sequences. Those sequences are the new or newly revised mouse and human EST sequences in NCBI's GenBank database within a month before 14 April 2003. After downloading the EST sequences, a simple "repeat masking" is conducted to replace all the sequences of ten or more identical letters to "N"s. This is because they are low complexity regions and their existence generates so many trivial sequence matches that overwhelm the real homologies.

SSearch is a subprogram in the FASTA package and implements the Smith-Waterman algorithm that gurantees to find the optimal alignment of every pair of sequences. Therefore, SSearch's sensitivity is regarded to be 100% in the comparison, and both BLAST and PatternHunter can only find a subset of the homologies found by SSearch. The performance of BLAST version 2.2.6, and PatternHunter version 2.0 are compared against SSearch. Each program uses a score scheme equivalent to:

match	:	1
mismatch	:	–1
gap open penalty	:	–5
gap extension penalty	:	–1

All pairs of ESTs with a local alignment of score at least 16 found by SSearch are

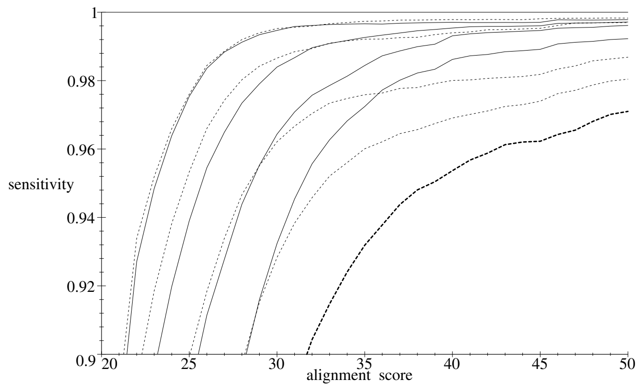

Fig. 8. The thick dashed curve is the sensitivity of Blastn, seed weight 11. From low to high, the solid curves are the sensitivity of PatternHunter using 1, 2, 4, and 8 weight 11 coding region seeds, respectively. From low to high, the dashed curves are the sensitivity of PatternHunter using 1, 2, 4, and 8 weight 11 general purpose seeds, respectively.

recorded. Also, if a pair of ESTs has more than two local alignments, only the one with the highest score is considered. All of these alignments are kept as being the correct set of homologies, noting of course that some of these alignments may be between sequences that are not evolutionally related.

As expected, SSearch takes approximately 20 CPU days, while BLASTN takes 575 CPU seconds, both on a 3GHz Pentium IV. SSearch finds 3,346,700 pairs of EST sequences that have local alignment score at least 16, with maximum local alignment score 694.

It is difficult to compare SSearch's sensitivity with BLASTN and Pattern-Hunter. This is because BLASTN and PatternHunter are heuristic algorithms, and need not compute optimal alignments. Thus Li*et al.* have decided—a bit arbitrarily—that if SSearch finds a local alignment with score x for a pair of ESTs, and BLAST (or PatternHunter) finds an alignment with score $\geq x/2$ for the same pair of ESTs, then BLAST (or PatternHunter) "detects" the homology. The successful detection rate is then regarded the sensitivity of BLAST (or PatternHunter) at score x. PatternHunter is run several times with different number of spaced seeds. Two sets of seeds—coding-region seeds and general-purpose seeds—are used, respectively. The results are in Figure 8.

SSearch	Blastn	PatternHunter				
		seeds	1	2	4	8
20 days	575 s	general	242 s	381 s	647 s	1027 s
		coding	214s	357s	575s	996s

Fig. 9. Running time of different programs.

Figure 9 lists the running time of different programs, with weight 11 seeds for Blastn and PatternHunter, on a Pentium IV 3GHz Linux PC. This benchmark demonstrates that PatternHunter achieves much higher sensitivity than Blastn at faster speed. Furthermore, PatternHunter with 4 coding region seeds runs at the same speed as Blastn and 2880 times faster than the Smith-Waterman SSearch, but with a sensitivity approaching the latter. It also demonstrates that the coding region seeds not only run faster—because there are less irrelevant hits—but are also more sensitive than the general purpose seeds. This is not a surprise because the EST sequences are coding regions.

For now, our brief tour through the world of homology search methods is coming to an end. Early landmarks include the classic global and local alignment algorithms that use full dynamic programming. Later highlights have been heuristic search algorithms, including increasingly sophisticated ones based on optimizing seeds for a particular type of alignment. At the end of it all, we have an algorithm that is almost as sensitive as Smith-Waterman, but requiring 3 orders of magnitude less time. We cannot believe, however, that the tour is entirely over, and encourage our readers to enjoy some time sightseeing on their own.

Acknowledgments

This work was supported in part by the Natural Science and Engineering Research Council of Canada (NSERC), the Ontario PREA award, Canada Council Research Chair program, the Killam Fellowship, and the Human Frontier Science Program. We would like to thank those who provided preprints or unpublished material that was used in this article.