Chapter 3

SEQUENCE ALIGNMENTS

Michael S. Waterman

TABLE OF CONTENTS

I. INTRODUCTION

When two or more sequences are displayed with one sequence written over another, the resulting configuration is known as an alignment of the set of sequences. These displays are very common in molecular biology as they communicate information about proposed common evolution or function of the nucleotide positions found in any given column of an alignment. Sequence alignments are frequently obtained in an ad hoc manner, and simply written down in a way that makes the result "look good". Sometimes this means that important but implicit criteria have been satisfied. No mathematical approach devoid of deep biological intuition could hope to compete with this method. What is to be hoped for is that, over a long period of cooperation, mathematical and biological scientists can make explicit some of the biological insights. However, alignments that are pleasing to the eye of the scientist may not have much merit beyond that. The commonly used phrases "aligned so as to maximize homology" and "gaps inserted in order to maximize homology" might conceal a fairly confused attempt to maximize matches and minimize gaps. It is often not clear what has been attempted and "homology" is left undefined. In these cases, an explicit optimization function is an advantage, both for checking whether the alignment is optimal and for discussing the desirability of the optimization function itself.

One of the first mathematical questions concerns how difficult the problem of sequence alignment really is. A naive approach to the problem is to systematically list all alignments, evaluating each one. Alas, even for the two sequence case, when gaps are allowed, there are a huge number of alignments, and this exhaustive approach succeeds on only the smallest of problems. Section II will give an account of what is known about the combinatorics of alignments. The message is that alignment is a difficult problem!

Next, Section III turns to the problem of aligning two sequences by dynamic programming methods. This problem has received more mathematical attention than any other in molecular sequence analysis and the length of Section III reflects this. Computer science has studied the same problem under the description of the string edit problem; the problem statement is to find the minimum number of changes (substitutions, insertions, deletions, inversions) to convert one string (sequence, file, or word) into another. This problem has often been studied in many different forms and a book[1] has appeared on dynamic programming approaches to the problems of sequence comparison. In the present chapter, Section III.D on most similar segments is probably the most useful dynamic programming algorithm for current problems in biology, and some new developments are reported. A program written in C for the similar segments algorithm appears in the Appendix. Both similarity and distance methods are given for aligning two sequences. Methods for producing the optimal and near optimal alignments are described.

Many very interesting problems involve more than two sequences. Section IV gives dynamic programming approaches to the several sequence problem. Line geometries, a recent method employing ideas from the geometry of geodesics, are described and illustrated. While the line geometry approach is practical, it is likely to fail when the sequences are processed using the most unrelated sequences first. Line geometries handle the sequences in a given order, not simultaneously. Another dynamic programming method is presented that does not suffer from this shortcoming although it is prohibitively expensive in terms of time storage.

It will have escaped few readers that alignment is closely related to consensus. While the methods for consensus patterns do not instantly solve the problem of aligning several sequences, Section V describes an important new and practical approach. Based on the consensus word algorithm,[2] these techniques can align many long sequences in reasonable time and storage. Many problems in biology involve more than two sequences.

One of the major tasks facing a sequence analyst is that of comparing new sequences with all, or a major portion of, a large data base. The most successful algorithms for that problem are based on hashing the data base, hashing the new sequence, and then comparing hash tables. Section VI treats these issues.

Finally suppose that these or some other methods have produced an alignment of interest to the scientist. Section VII discusses how the alignment might be examined for statistical significance. Some mathematics has recently been worked out that solves several aspects of this problem. Extreme value theory has been employed to derive the so-called log(n) distribution.[3,4] that gives the statistical distribution of the best matching segments between two or more long sequences.

Recently in Waterman,[5] methods of sequence comparison were reviewed and organized. Many references of that paper do not appear in this chapter. Here, the goal is to more fully describe some major methods of sequence alignment and not to provide a review.

II. THE NUMBER OF ALIGNMENTS

In this section, a combinatorial treatment of sequence alignments is given. Biology provides the motivation for aligning sequences and for considering how difficult alignment is. It is then a mathematical task to estimate the number of sequence alignments. The results are applicable to biology in a negative sense; they assure one that a huge number of possible alignments exist and that direct enumeration is hopeless. The two-sequence case is handled first.

Notation is important here. Let $\mathbf{a} = a_1a_2 \ldots a_n$ and $\mathbf{b} = b_1b_2 \ldots b_m$ be two sequences of length, n and m. One way to think of alignment is that an alignment is produced when null elements, ϕ, are inserted into the sequences; the new sequences must be of the same length, L. Then the two sequences are written, one over the other. $\mathbf{a} = a_1a_2 \ldots a_n$ becomes, with the insertion of ϕ, $\mathbf{a}^* = a_1^*a_2^* \ldots a_L^*$ while $\mathbf{b} = b_1b_2 \ldots b_m$ becomes $\mathbf{b}^* = b_1^*b_2^* \ldots b_L^*$. The subsequence of $\mathbf{a}^*$ or $\mathbf{b}^*$ whose elements are not equal to ϕ is the original sequence. The alignment is

$$a_1^*a_2^* \ldots a_L^*$$
$$b_1^*b_2^* \ldots b_L^*$$

To see this process, let $\mathbf{a}$ = ATAAGC and $\mathbf{b}$ = AAAAACG. To obtain an alignment, one of many possibilities is to set $\mathbf{a}^* = \phi$ATAAGCϕ and $\mathbf{b}^* =$ AAAAAϕCG. For example $b_1^* = A$, $b_6^* = \phi$, and $b_7^* = C$ while $b_1 = A$, $b_6 = C$ and $b_7 = G$. The alignment is written

$$\mathbf{a}^* = \phi\text{ATAAGC}\phi$$
$$\mathbf{b}^* = \text{AAAAA}\phi\text{CG}$$

Here $b_1^* = A$ is said to be inserted into the first sequence or deleted from the second, depending on the point of view. $a_2^* = A$ matches $b_2^* = A$ while $a_3^* = T$ and $b_3^* = A$ constitutes a mismatch.

The problem of this section is to find how many ways can $\begin{smallmatrix}a_i^*\\b_i^*\end{smallmatrix}$ be written. No alignment terms $\begin{smallmatrix}\phi\\\phi\end{smallmatrix}$ are allowed as there is no point in matching two deletions. This makes it clear that $\max\{n,m\} \leq L \leq n + m$. The case $L = n + m$ comes, e.g., by first deleting all a_i and then deleting all b_j:

$$a_1a_2 \ldots a_n\phi\ \phi \ldots \phi$$
$$\phi\ \phi \ldots \phi\ b_1b_2 \ldots b_m$$

Combinatorial insight comes by recognizing that alignments of these two sequences can end in exactly one of three ways

$$\begin{matrix} \cdots a_n & \cdots a_m & \cdots \phi \\ \cdots \phi & \cdots b_m & \cdots b_m \end{matrix}$$

where $\binom{*}{\phi}$ corresponds to an insertion/deletion of a_n, $\binom{*}{*}$ corresponds to a match or mismatch of a and b, and $\binom{\phi}{*}$ corresponds to an insertion/deletion of b_m. Note that the fate of the unseen bases (those not displayed) is not specified. Define

$$f(i,j) = \text{number of all possible alignments of one sequence of } i \text{ letters with another of } j \text{ letters}$$

The above three cases imply

$$f(n,m) = f(n-1,m) + f(n-1,m-1) + f(n,m-1)$$

This follows because, for example, the first term corresponding to deleting a_n can obtain with the number of alignments of $a_1 \ldots a_{n-1}$ and $b_1 \ldots b_m$, which is $f(n-1,m)$.

The author obtained the above recursion equation and with P. R. Stein determined that it specified the Stanton-Cowan numbers.[6] Then Stein communicated the problem of asymptotics to H. T. Laquer who obtained the following theorem in 1981.[7]

Theorem 1 — Let $f(n,m)$ be defined as above. Then

$$f(n,n) \approx (1+\sqrt{2})^{2n+1}\sqrt{n}$$

as $n \to \infty$, where $c(n) \approx d(n)$ means $\lim_{n\to\infty} c(n)/d(n) = 1$.

Two sequences of length 1000, then have

$$f(1000,1000) \approx (1+\sqrt{2})^{2001}\sqrt{1000} = 10^{767.4}$$

alignments! There are approximately 10^{80} elementary particles in the universe; Avogadro's number is on the order of 10^{23}.

If it is agreed not to count

$$\begin{matrix} C\phi & \text{and} & \phi C \\ \phi G & & G\phi \end{matrix}$$

as distinct, the situation improves (slightly). Let $g(n,m)$ denote this smaller number of alignments. $\binom{*}{*}$ has three possibilities

$$\begin{matrix} \cdots a_n & \phi & \cdots \phi & \phi & \cdots a_n\phi \\ \cdots b_m & b_m & \cdots b^{m-1} & b_m & \cdots \phi\, b_m \end{matrix}$$

while $\binom{*}{\phi}$ has

$$\begin{matrix} \cdots a_n & a_n & \cdots a_n & a_n & \cdots \phi\, a_n \\ \cdots b_m & \phi & \cdots \phi & \phi & \cdots b_m\phi \end{matrix}$$

The new version of the recursion equation is

$$g(n,m) = g(n-1,m) + g(n,m-1) + g(n-1,m-1) - g(n-1,m-1)$$

substracting the double count. The result is given in the next theorem, where the asymptotics are derived from Stirling's formula.[8]

Table 1
BEHAVIOR OF $h(b,n) \sim \gamma_b n^{-1/2} D_b^n$, THE NUMBER OF ALIGNMENTS OF TWO SEQUENCES OF LENGTH n WITH MATCHED BLOCKS OF LENGTH AT LEAST b

b	D_b	γ_b
1	5.8284	0.57268
2	4.5189	0.53206
3	4.1489	0.54290
4	4.0400	0.55520
5	4.0103	0.56109
10	4.0001	0.56183
...		
∞	4.0000...	$0.56419... = \pi^{-1/2}$

Theorem 2 — If $g(n,m)$ is defined as above, $g(0,0) = g(0,1) = g(1,0) = 1$, and $g(n,m) = \binom{*}{*}^{m}$. If $n = m$,

$$g(n,n) = \binom{2n}{n} \approx 2^{2n}(4\sqrt{n\pi})^{-1}, \quad \text{as } n \to \infty$$

Two sequences $n = m = 1000$ have $g(1000,1000) \sim 10^{600}$ alignments so that direct search is still impossible.

It is possible to further reduce the number of alignments by requiring matches and mismatches to occur in blocks of length at least b without interruptions by deletions. The motivation for this is that biologists sometimes reject alignments with small groups of matches. The counting scheme of Theorem 1 is readopted with this new requirement. The following theorem appears in Griggs et al.[9] where it is derived via generating functions.

Theorem 3 — Let $h(b,n)$ be the number of alignments of two sequences of length n where matches must occur in blocks of length at least $b \geq 1$. Define $\Phi(x) = (1-x)^2 - 4x(x^b - x + 1)^2$, and let ρ be the smallest real root of $\Phi(0) = 0$. Then

$$h(b,n) \sim (\gamma_b n^{-1/2}) D_b^n, \quad \text{as } n \to \infty$$

where $D_b = \rho^{-1}$ and

$$\gamma_b = (\rho^b - \rho + 1)(-\pi\rho\Phi'(\rho))^{-1/2}$$

Notice that $b = 1$ has $h(1,n) = f(n,n)$. Table 1 shows the behavior with b. When $b = 2$, for example,

$$h(2,n) \sim (0.53206)n^{-1/2}(4.5189)^n$$

More than two sequences is bound to make the problem more complicated, greatly increasing the number of alignments. The question is to determine how many more alignments. Recently, Griggs et al.[10] have provided an answer which is given in the next theorem. The methods of proof are the most difficult of any required so far, using a saddle point techinque.

Theorem 4 — Let $f_k(n)$ be the number of alignments of k sequences of length n. For a fixed $k \geq 2$,

$$\lim_{n\to\infty} \frac{\ell n\, f_k(n)}{n} = \ell n\, c_k$$

where

$$c_k = (2^{1/k} - 1)^{-k} \approx \frac{1}{\sqrt{2}}\left[\frac{k}{\ell n(2)}\right]$$

For our use notice that

$$f_k \cdot c_k^n = (2^{1/k} - 1)^{-kn}$$
$$\cdot 2^{-n/2}\left(\frac{k}{\ell n(2)}\right)^n$$

For $n = 1000$ and $k = 3$, $f_3(1000) \doteq 10^{1751}$.

III. DYNAMIC PROGRAMMING ALIGNMENT OF TWO SEQUENCES

Needleman and Wunsch[11] wrote a paper titled "A general method applicable to the search for similarities in the amino acid sequence of two proteins". It was surely unknown to the authors that their method fit into a broad class of algorithms introduced by Richard Bellman under the name dynamic programming. Their paper has had a great deal of influence in biological sequence alignment. Its great advantage is that an explicit criterion for optimality of alignment is stated, as well as an efficient method of solution given. Insertions, deletions, mismatches (negative similarity), and matches (positive similarity) were allowed in the alignments.

During early 1970s, Stan Ulam and some other mathematicians became interested in defining a distance D(a,b) on sequences. The minimum distance alignment was defined to be an alignment with the smallest weighted sum of mismatches, insertions, and deletions. The advantage of a distance was the construction of a metric space on the space of sequences:

1. $D(a,b) = 0$ if and only if $a = b$.
2. $D(a,b) = D(b,a)$ (symmetry).
3. $D(a,b) \leqq D(a,c) + D(c,b)$ for any c (triangle inequality).

The emphasis on sequence metrics came from the fact that a matrix of sequence distances was often used to construct an evolutionary tree. P.H. Sellers[12] gave a dynamic programming algorithm, very similar to that of Needleman and Wunsch, to calculate the distance.

The historical order is reversed here. Distance methods are described in Section III.A, with similarity methods in Section III.B. As mentioned in the introduction, we find similarity to be the most satisfactory. All problems known to be solvable with distance methods can be solved with similarity methods. However, in Section III.D, a similarity solution is given that has no distance counterpart. Still, the metric space associated with a distance makes it worthwhile to present distance methods. Section III.C shows several simple modifications to solve related problems such as best fit of a short sequence into a long one. Section III.D studies the important problem of locating segments of two sequences which are unexpectedly similar, although the full sequences might not have a good alignment. New results are given here for this problem. Section III.E closes with a recent modification of the dynamic programming algorithms that allows all alignments near the optimal to be produced.

A. Distance Alignment

The sequences $\mathbf{a} = a_1a_2 \ldots a_n$ and $\mathbf{b} = b_1b_2 \ldots b_m$ are written over the alphabet

{A,C,G,T}. Any finite alphabet will of course work here. In particular the 20-letter, amino acid, alphabet of proteins, or the purine/pyrimidine alphabet for DNA can be used. Let d(a,b) be a distance on the alphabet and let g(a) be the positive cost of a gap of the letter "a". The distance d(a,b) represents the cost of a mutation of a into b. If d(a,b) is extended so that $d(a,\phi) = d(\phi,a) = g(a)$, then define

$$D(\mathbf{a},\mathbf{b}) = \min \sum_{i=1}^{k} d(a_i',b_i')$$

where the minimum is extended over all alignments of a with b. Seller's result[12] can be summarized in the next Theorem.

Theorem 1 — If $\mathbf{a} = a_1a_2 \ldots a_n$ and $\mathbf{b} = b_1b_2 \ldots b_m$, define $D_{i,j} = D(a_1a_2 \ldots a_i, b_1b_2 \ldots b_j)$. Also set

$$D_{0,0} = 0, D_{0,j} = \sum_{k=1}^{j} d(\phi,b_k), \quad \text{and} \quad D_{i,0} = \sum_{k=1}^{i} d(a_k,\phi)$$

Then

$$D_{i,j} = \min\{D_{i-1,j-1} + d(a_i,b_j), D_{i-1,j} + d(a_i,\phi), D_{i,j-1} + d(\phi,b_j)\} \qquad (1)$$

If d(.,.)is a metric on the alphabet, then D(.,.) is a metric on the set of finite sequences.

Proof — We verify Equation 1 with reasoning similar to that for verifying the recursion equation for f(n,m) in Section II. The alignment of $a_1 \ldots a_i$ and $b_1 \ldots b_j$ can end in one of three ways

$$\begin{matrix} \ldots a_i & \ldots a_i & \ldots \phi \\ \ldots \phi & \ldots b_j & \ldots b_j \end{matrix}$$

If the optimal alignment ends in $\begin{matrix} a_i \\ b_j \end{matrix}$, the cost must be $D_{i-1,j-1} + d(a_i,b_j)$ since the initial part of the alignment must itself be optimal and align $a_1 \ldots a_{i-1}$ with $b_1 \ldots b_{j-1}$.

If the optimal alignment ends in $\begin{matrix} a_i \\ \phi \end{matrix}$, the cost must be

$D_{i-1,j} + d(a_i,\phi)$ since $a_1 \ldots a_{i-1}$ and $b_1 \ldots b_j$ must be optimally aligned

The case $\begin{matrix} \phi \\ b_j \end{matrix}$ is identical in reasoning with the case $\begin{matrix} a_i \\ \phi \end{matrix}$.

The optimal alignment has least cost of these three possibilities and Equation 1 is proven.

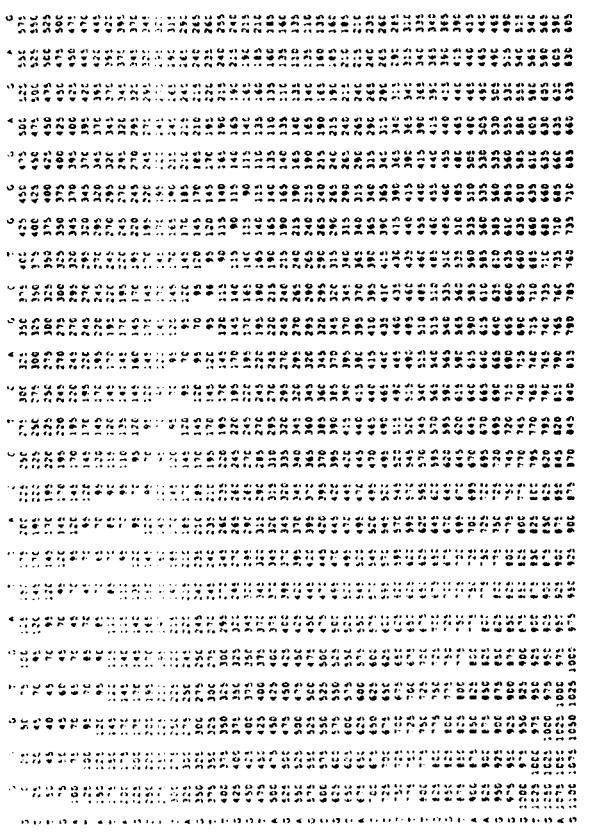
Another statement of Equation 1 is

$$D_{i,j} = \min\{D_{i-1,j-1} + g, D_{i-1,j} + d(a_i,b_j), D_{i,j-1} + g\}$$

when g is a constant gap cost, $g = d(\phi,a) = d(a,\phi)$. These algorithms have computation cost proportional to nm, O(nm).

To illustrate the algorithm, we align two *Escherichia coli* tRNA sequences, threonine tRNA (sequence a; GenBank name ECOTRTACU) and valine tRNA (sequence b; GenBank name ECOTRV1). The algorithm has $d(a,b) = 2$, if $a \neq b$, and $g = 2.5$. Entries in the matrix in Table 2 are multiplied by 10 to allow use of integer arithmetic. Table 2 shows the matrix for the 5' (left) ends of the sequences. There are 72 optimal alignments, one of which is shown next. Portions of the alignment common to all 72 alignments are boxed.

Table 2
DISTANCE MATRIX FOR ALIGNMENT OF THREONINE tRNA (a) WITH VALINE tRNA (b)



ACCA CCC ACTAΦCTGCCCTAGCTTGGCGGCTGGGGGAGGAA CΦA TTCCC TCCA TGAGA
ACCA CΦG ACTATCCGTCTA AGCTTGACGGCTGGAGTGGGAA TGG TCCCC ΦACG TGAGA

GGGTCGACTGCATΦTAGT GGG
TGGTTGACTCGATATAGT CΦG

To obtain all optimal alignments there are two techniques. The first involves saving pointer at each (i,j). The pointers show which of (i − 1,j), (i − 1,j − 1) and (i,j − 1) are involved wit the optimal $D_{i,j}$. Then, when $D_{n,m}$ is found, the pointers are followed to produce alignment The second technique is to ask, at each (i,j) on an optimal path, which of (i − 1,j), (i − 1,j − 1 or (i,j − 1) are optimal by recomputing the three terms. The alignments in these cases a produced by depth first search with stacking, where the stacks are managed by last in-fir out. This is usually referred to as a traceback. Table 2 can be used to illustrate these alignme procedures.

Frequently in sequence evolution, gaps of several adjacent bases are not the sum of sing base gaps but the result of one event. It is thus sometimes required to weight these multip gaps differently from summing single gap weights. Let g_k be the gap weight for a gap of bases. It is reasonable that $g_k \leq kg_1$ holds. In Waterman et al.,[13] an algorithm was giv for this case.

Theorem 2 — Set $D_{i,j} = D(a_1, \ldots, a_i, b_1, \ldots b_j)$, $D_{0,0} = 0$, $D_{0,j} = g_j$, $D_{i,0} = g_i$. Th

$$D_{i,j} = \min\{D_{i-1,j-1} + d(a_i,b_j), \min_{1\leq k\leq i}\{D_{i-k,j} + g_k\}, \min_{1\leq k\leq j}\{D_{i,j-k} + g_k\}\}$$

The computation time of this algorithm is $\Sigma_{i,j}(i+j) = O(n^2m + nm^2)$, or $O(n^3)$ if n m. For sequences of length 1000 or more, it is important to reduce this running time. T was accomplished by Gotoh[14] for this case of g_k linear.

Theorem 3 — Let $g_k = \alpha + \beta(k-1)$ for constants α and β. Set $E_{0,0} = F_{0,0} = D_{0,0} = 0$, $E_{i,0} = D_{i,0} = g_i$, and $F_{0,j} = D_{0,j} = g_j$. If $E_{i,j}$ and $F_{i,j}$ satisfy

$$E_{i,j} = \min\{D_{i,j-1} + \alpha, E_{i,j-1} + \beta\}$$

and

$$F_{i,j} = \min\{D_{i-1,j} + \alpha, F_{i-1,j} + \beta\}$$

then

$$D_{i,j} = \min\{D_{i-1,j-1} + d(a_i,b_j), E_{i,j}, F_{i,j}\}$$

This gives an O(nm) running time for linear gap weights. If g_k is concave, such as $g_k = \alpha + \beta\log(k)$, then Waterman[15] gives an algorithm with a more complex algorithm. Mi and Myers[47] give a $O(n^2\log n)$ algorithm for g_k concave. If g_k is convex, an O(nm) algori is easier to obtain than when g_k is concave. However, convex gap functions seems unlikely situation in biology.

B. Similarity Alignment

Now take s(a,b) to be a similarity measure on the alphabet. That is, we must have s(> 0 for all a. For some (a,b) pairs, it is necessary that s(a,b) < 0. The idea is that simila (e.g., matching a with a) is rewarded by a positive score while aligning dissimilar letter penalized by a negative score. Let $-\hat{g}(a)$ be the gap penalty associated with a. ($\hat{g}$ is w

to distinguish this function from the gap weight of distance alignment.) Set $s(a,\phi) = s(\phi,a) = -\hat{g}(a)$. Then define the similarity of **a** and **b** by

$$S(a,b) = \max \sum_{i=1}^{l} s(a_i', b_i')$$

where the maximum is over all alignments. The theorem of Needleman and Wunsch is

Theorem 1 — If $\mathbf{a} = a_1 a_2 \ldots a_n$ and $\mathbf{b} = b_1 b_2 \ldots b_m$, define

$S_{i,j} = S(a_1 a_2 \ldots a_i, b_1 b_2 \ldots b_j)$. Also set $S_{0,0} = 0$,

$S_{0,j} = \sum_{k=1}^{j} s(\phi, b_k)$, and $S_{i,0} = \sum_{k=1}^{i} s(a_i, \phi)$. Then

$$S_{i,j} = \max\{S_{i-1,j-1} + s(a_i,\phi), S_{i-1,j-1} + s(a_i,b_j), S_{i,j-1} + s(\phi,b_j)\}$$

If $\hat{s}(a,\phi) = \hat{s}(\phi,a) = -\hat{g}$, for all a,

$$S_{i,j} = \max\{S_{i-1,j} - \hat{g}, S_{i-1,j-1} + s(a_i,b_j), S_{i,j-1} - \hat{g}\}$$

This theorem is proved exactly as was Theorem 1, Section III.A.

To illustrate the similarity algorithm, we align the same sequences as above, *E. coli* threonine tRNA and *E. coli* valine tRNA. We use a single gap algorithm and choose the parameters $s(a,a) = +1$, $s(a,b) = -1$ if $a \neq b$, and $\hat{g} = 2$. In Theorem 4 it is shown that for these parameters, the set of optimal similarity alignments is identical to that for the distance optimal alignments with the parameters of Table 2. Table 3 shows the matrix (S_{ij}) for the 5′ (left) ends of the sequences.

The multiple gap case is covered by the next theorem. Gap penalty $\hat{g}_k$ is again assumed to be a function of gap length k.

Theorem 2 — Set $S_{i,j} = S(a_1 a_2 \ldots a_i, b_1 b_2 \ldots b_j)$, $S_{0,0} = 0$, $S_{0,j} = -\hat{g}_j$, $S_{i,0} = -\hat{g}_i$. Then

$$S_{i,j} = \max\{S_{i-1,j-1} + s(a_i,b_j), \max_{1 \le k \le j}\{S_{i,j-k} + \hat{g}_k\}, \max_{1 \le k \le i}\{S_{i-k,j} + \hat{g}_k\}\}$$

The following result obtains the $O(nm)$ running time for the case of linear gap weights.

Theorem 3 — Let $\hat{g}_k = \alpha + \beta(k-1)$ for constants α and β. Set $E_{0,0} = F_{0,0} = S_{0,0} = 0$, $E_{i,0} = S_{i,0} = -\hat{g}_i$ and $F_{0,j} = S_{0,j} = -\hat{g}_j$. If

$$E_{i,j} = \max\{S_{i,j-1} - \alpha, E_{i,j-1} - \beta\}$$

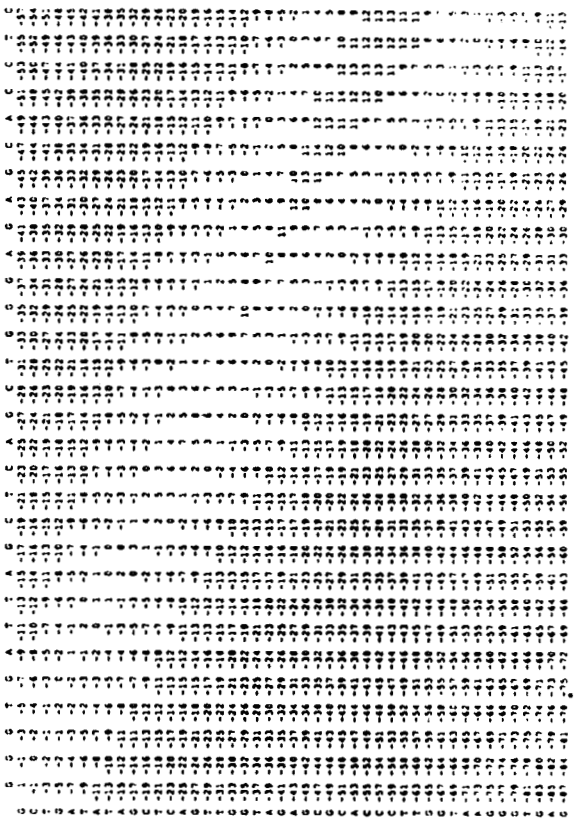
$$F_{i,j} = \max\{S_{i-1,j} - \alpha, F_{i-1,j} - \beta\}$$

then

$$S_{i,j} = \max\{S_{i-1,j-1} + s(a_i,b_j), E_{i,j}, F_{i,j}\}$$

The parallels between the formulas for distance and similarity alignment lead to a natural question. When are similarity and distance algorithms equivalent? This question was an-

Table 3
SIMILARITY MATRIX FOR ALIGNMENT OF THREONINE tRNA (a) WITH VALINE tRNA (b)



swered by Smith and Waterman[16] and Fitch et al.[17] *When full sequences are aligned by* distance (similarity), there is a similarity (distance) algorithm that gives the same set of optimal alignments. That is, finding similarity and distance alignments are dual problems.

Theorem 4 — Let a *similarity measure* be given with s(a,b) and gap penalties $\hat{g}_k$ and a distance measure be given with d(a,b) and gap weight g_k. Assume there is a constant c, $0 \leqq c \leqq \max_{a',b'} d(a',b')$ such that s(a,b) = c − d(a,b) and $\hat{g}_k = g_k - (kc)/2$. Then an alignment is similarity optimal if and only if it is distance optimal.

Proof — For simplicity, the proof uses the single gap case ($g_k = \infty$ for $k \geqq 2$). Now by elementary counting

$$n + m = 2\#\text{matches} + \#\text{gaps}$$

obviously holds. Using this simple equation,

$$\begin{aligned} D(a,b) &= \min\left\{ \sum_{\text{matches}} d(a,b) + g_1\#\text{gaps} \right\} \\ &= \min\left\{ \sum_{\text{matches}} c - \sum_{\text{matches}} s(a,b) + g_1\#\text{gaps} \right\} \\ &= \min\left\{ c(n+m)/2 - \sum_{\text{matches}} s(a,b) + (g_1 - c/2)\#\text{gaps} \right\} \\ &= c(n+m)/2 - \max\left\{ \sum_{\text{matches}} s(a,b) - (g_1 - c/2)\#\text{gaps} \right\} \end{aligned}$$

Notice that

$$D(a,b) + S(a,b) = c(n+m)/2$$

so "large distance" is "small similarity". After seeing this equivalence, it is surprising that there are problems with a simple similarity algorithm for which no equivalent simple distance *algorithm exists. This situation arises in Section* D.

C. Fitting One Sequence into Another

Next the algorithms are modified to solve a new problem: the best fit of a "short" sequence into a "larger" sequence. An example of when this might be of interest is in locating a regulatory pattern in a nucleotide sequence, such as TATAAT in a bacterial promoter. The algorithm finds where the short pattern approximately appears in the longer sequence.

First consider the problem of *fitting* $\mathbf{a} = a_1a_2 \ldots a_n$ *into* $b = b_1b_2 \ldots b_m$. For the purpose of visualizing the problem think of n as much smaller than m. (The relative sizes of n and m are irrelevant to the mathematics.) The problem is to find i and j ($i \leqq j$) such that

$$S(a,b_i b_{i+1} \ldots b_{j-1}b_j) = \max_{s,t} S(a,b_s b_{s+1} \ldots b_{t-1}b_t)$$

This problem is simply solved. We just use the similarity algorithm of Theorem 1, Section III.B, with $S_{0,j} = 0$ for all j. Then the required j is found by

$$S(a,b_1 \ldots b_j) = \max_{1 \leqq j \leqq m} S(n,j)$$

and i is found by tracing back beginning at (n,j) (instead of n,m)). The same procedure works with distance, as first shown by Sellers.[18,19]

To illustrate this algorithm, we take as sequence *b* the *E. coli* promoter sequence of lacI.[20] In *E. coli* promoter sequences, the −10 signal or pattern TATAAT is well known to have functional significance. The designation −10 refers to the distance to the left (5') of the mRNA start. We take **a** = TATAAT. As above s(a,a) = 1, s(a,b) = −1 if $a \neq b$, and $\hat{g} = 2$. The matrix (S(i,j)) is shown in Table 4. Searching the last row of the matrix gives two solutions of $\max_{1 \leqq j \leqq 50} S(6,j) = 2$, at (6,13) and (6,43). The pattern at (6,43) has the alignment

TATAAT
CATGAT

and is CATGAT in the promoter sequence, the canonical −10 pattern. The pattern at (6,13) has the alignment

TATAAT
TCGAAT

with TCGAAT in the promoter sequence, an equally good fit.

This illustrates the utility of the algorithm, in that it locates the putative −10 signal CATGAT in lacI. It also emphasizes the difficulty of promoter signal analysis by finding an equally good pattern TCGAAT 30 bases 5' of the −10 pattern.

D. Identification of Similar Segments

Surprising relationships have been discovered between sequences that overall have little similarity. See Weiss,[21] Doolittle et al.[22] and Naharro[23] for accounts of some unexpected long matching segments between viral and host DNA. The subject of this subsection is a dynamic programming algorithm to find these similar segments. This is probably the most useful dynamic programming algorithm for current problems. For a mathematical statement of the problem, it is necessary to assume a similarity function s(a,b). The object is to find

$$\max_{\substack{1 \leqq s \leqq t \leqq n \\ 1 \leqq u \leqq v \leqq m}} S(a_s a_{s+1} \ldots a_{t-1}a_t, b_u b_{u+1} \ldots b_{v-1}b_v)$$

This amounts to $\binom{n}{2}\binom{m}{2}$ sequence alignment problems, and a new algorithm must be devised.

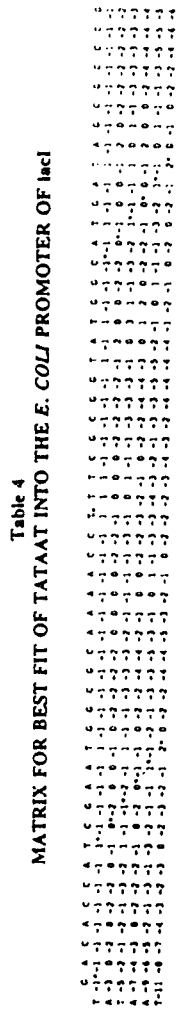
While Sellers[18,19] began the work on problems of this type, his problem formulations were based on distance functions and his algorithms involved forward and backward recursions, each recursion requiring a matrix. Although later[24] similarity functions are recommended by Sellers, those algorithms still involve intersection of path graphs and are quite complex. The similarity formulation given above was presented by Smith and Waterman and is solved in a straightforward way. Define $H_{i,j}$ to be the *maximum* similarity of two segments ending at a_i and b_j:

$$H_{i,j} = \max\{0;\quad S(a_x a_{x+1} \ldots a_i, b_y b_{y+1} \ldots b_j):\quad 1 \leqq x \leqq i,\ 1 \leqq y \leqq j\}$$

A recursion similar to those given for the similarity problems discussed above is obtained for H.[25]

Theorem 1 — Set $H_{i,0} = H_{0,j} = 0$ for $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$. Then

Table 4
MATRIX FOR BEST FIT OF TATAAT INTO THE *E. COLI* PROMOTER OF lacI



$$H_{i,j} = \max\{0, H_{i-1,j-1} + s(a_i,b_j), \max_{1\le k\le i}\{H_{i-k,j} - \hat{g}_k\}$$

$$\max_{1\le \ell\le j}\{H_{i,j-\ell} - \hat{g}_\ell\}\}$$

Of course, single or linear gaps can be treated as discussed above (see Gotoh[14]). A traceback from (i,j) satisfying

$$H_{i,j} = \max_{1\le k\le i,\ 1\le \ell\le j}\{H_{k,\ell}\}$$

will give a maximum similarity segment. What about other highly similar segments?

The following procedure finds one alignment from the highest score and then continues to find the next-best alignment with no matches or mismatches of its alignment in common with those already output. The original algorithm[25] stopped after the maximum segments were output and the algorithm given here is new.[26]

When calculating the matrix H, stack all (i,j,Y) with $Y = H_{i,j}$ and $H_{i,j} \ge C$ = cutoff value. The stack is ordered by $>$ where $(i,j,H_{i,j}) > (k,\ell,H_{k,\ell})$ if

$$\text{i) } H_{i,j} > H_{k,\ell}$$

$$\text{or ii) } H_{i,j} = H_{k,\ell} \text{ and } i + j < k + \ell$$

$$\text{or iii) } H_{i,j} = H_{k,\ell}, i + j = k + \ell, \text{ and } i < k$$

During tracebacks for some stack entry, we only output one alignment. For this one additional concept is needed, that of minimum length alignment. Define the length of an alignment beginning at (p,q) and ending at (i,j) to be $|i + j - (p + q)|$. We only output minimum length alignments, although this is entirely a matter of choice.

The algorithm begins with the top (i,j,Y), i.e., the largest under ">", and that alignment is output. Next we must find the next largest scoring alignment that has no matches or mismatches in common with those already output. The simple concept of recomputing the matrix, not allowing matches or mismatches already used, is employed. This does involve more calculation. As the elements below and to the right of an alignment's end (i.e., (i,j)) must be recomputed, each succeeding alignment takes n*m/4 matrix entry recomputations on the average. While this might be worth the cost because of simplicity, much more efficient algorithms can be given in the cases of single and linear gaps.

Take the single insertion and deletion case, and let (k,ℓ) be the upper leftmost position of a match. (The alignment must end in a match or it is not optimal.) The new matrix N satisfies

$$N_{i,j} = H_{i,j}, \ i < k \text{ and } j < \ell$$

Define $N_{k,\ell}$ by

$$N_{k,\ell} = \max\{0, N_{k-1,\ell} - g_1, N_{k,\ell-1} - g_1\}$$

Note that the match ending the alignment is not allowed. Consider the row (k,j), $\ell < j$. Recompute each entry $j = \ell + 1, \ell + 2, \ldots$ until $H_{k,j} = N_{k,j}$. Then it is clear that $H_{k,\ell} = N_{k,\ell}$ for the rest of the row. Similar considerations hold for the remainder of the alignment Note that on each row and column it is necessary to go at least to the position that was necessary for the preceding row or column. By this device a much more efficient algorithm is obtained. If an alignment has length L the recomputation required is approximately L^2 and if several alignments are output the recomputation is proportional to ΣL_i^2.

		A	G	T	C	G	A	G	G	G	C	T	A	C	T	C	T	A	C	T	G	A	A	C
	r	0	0	0	10	10	0	0	0	0	0	10	0	0	10	0	10	0	0	10	0	0	0	0	10
	C	0	0	0	0	20	1	0	0	0	0	10	1	0	10	1	10	1	0	10	1	0	0	0	10
	A	10	0	0	0	1	11	11	0	0	0	0	1	11	0	1	0	1	11	0	1	0	10	10	0
	A	10	1	0	0	0	0	21	2	0	0	0	0	11	2	0	0	0	11	2	0	0	10	20	1
	T	0	1	11	0	0	0	1	12	0	0	0	10	0	2	12	0	10	0	2	12	0	0	1	11
	C	0	0	0	21	10	0	0	0	3	0	10	0	1	10	0	22	2	1	10	0	3	0	0	11
	T	0	0	10	1	12	1	0	0	0	0	0	20	0	0	20	2	32	12	0	20	0	0	0	0
	A	10	0	0	1	0	3	11	0	0	0	0	0	30	10	0	11	12	42	22	2	11	10	10	0
	C	0	1	0	10	11	0	0	2	0	0	10	0	10	40	20	10	2	22	52	32	12	2	1	20
	T	0	0	11	0	1	2	0	0	0	0	0	20	0	20	50	30	20	2	32	62	42	22	2	0
	A	10	0	0	2	0	0	12	0	0	0	0	0	30	10	30	41	21	30	12	42	53	32	12	0
	C	0	1	0	10	12	0	0	3	0	0	10	0	10	40	20	40	32	12	40	22	33	44	43	42
	T	0	0	11	0	1	3	0	0	0	0	0	20	0	20	50	30	50	30	20	50	30	24	35	34
	G	0	10	0	2	0	11	0	10	10	10	0	0	11	0	30	41	30	41	21	30	60	40	29	26
	C	0	0	1	10	12	0	2	0	1	1	20	0	0	21	10	40	32	21	51	31	40	51	31	30
	T	0	0	10	0	1	3	0	0	0	0	0	30	10	1	31	20	50	30	31	41	41	31	42	22
	T	0	0	10	1	0	0	0	0	0	0	0	10	21	1	11	22	30	41	21	41	52	32	22	33
	G	0	10	0	1	0	10	0	10	10	10	0	0	1	12	0	2	13	21	32	21	51	43	23	13
	C	0	1	0	10	11	0	1	0	1	1	20	0	0	11	1	10	0	4	31	23	31	42	34	33
	A	10	0	0	0	1	2	10	0	0	0	0	0	10	0	2	0	1	10	11	22	14	41	52	32
	G	0	20	0	0	0	11	0	20	10	10	0	0	2	1	0	0	0	0	1	2	32	21	32	43
	T	0	0	30	10	0	0	2	0	11	1	1	10	0	0	11	0	10	0	0	11	12	23	12	23
	A	10	0	10	21	1	0	10	0	0	2	0	0	20	0	0	2	0	20	0	0	2	22	33	13
	C	0	1	0	20	31	11	0	1	0	0	12	0	0	30	10	10	0	0	30	10	0	2	13	43

A

		A	G	T	C	C	G	A	G	G	G	C	T	A	C	T	C	T	A	C	T	G	A	A	C
	C	0	0	0	10	10	0	0	0	0	0	0*	0*	0	10	0	10	0	0	10	0	0	0	0	10
	C	0	0	0	0	20	1	0	0	0	0	10*	0*	0*	10	1	10	1	0	10	1	0	0	0	10
	A	10	0	0	0	1	11	11	0	0	0	0	1*	0*	0*	1	0	1	11	0	1	0	10	10	0
	A	10	1	0	0	0	0	21	2	0	0	0	0	11*	0*	0*	0	0	11	2	0	0	10	20	1
	T	0	1	11	0	0	0	1	12	0	0	0	10	0	2*	0*	0*	10	0	2	12	0	0	1	11
	C	0	0	0	21	10	0	0	0	3	0	10	0	1	10	0*	0*	0*	1*	10	0	3	0	0	11
	T	0	0	10	1	12	1	0	0	0	0	0	20	0	0	20	0*	0*	0*	0*	20	0	0	0	0
	A	10	0	0	1	0	3	11	0	0	0	0	0	30	10	0	11*	0*	0*	0*	0*	11*	10	10	0
	C	0	1	0	10	11	0	0	2	0	0	10	0	10	40	20	10	2*	0*	0*	0*	0*	2*	1	20
	T	0	0	11	0	1	2	0	0	0	0	0	20	0	20	50	30	20	0*	0*	0*	0*	0*	0*	0*
	A	10	0	0	2	0	0	12	0	0	0	0	0	30	10	30	41	21	30*	10*	0*	0*	10*	10*	0*
	C	0	1	0	10	12	0	0	3	0	0	10	0	10	40	20	40	32	12	40*	20*	0*	0*	1*	20*
	T	0	0	11	0	1	3	0	0	0	0	0	20	0	20	50	30	50	30	20	50*	30*	10*	0*	0*
	G	0	10	0	2	0	11	0	10	10	10	0	0	11	0	30	41	30	41	21	30	60	40*	20*	0*
	C	0	0	1	10	12	0	2	0	1	1	20	0	0	21	10	40	32	21	51	31	40	51	31	30*
	T	0	0	10	0	1	3	0	0	0	0	0	30	10	1	31	20	50	30	31	41	41	31	42	22
	T	0	0	10	1	0	0	0	0	0	0	0	10	21	1	11	22	30	41	21	41	52	32	22	33
	G	0	10	0	1	0	10	0	10	10	10	0	0	1	12	0	2	13	21	32	21	51	43	23	13
	C	0	0	1	10	11	0	1	0	1	1	20	0	0	11	1	10	0	4	31	23	31	42	34	33
	A	10	0	0	0	1	2	10	0	0	0	0	0	10	0	2	0	1	10	11	22	14	41	52	32
	G	0	20	0	0	0	11	0	20	10	10	0	0	2	1	0	0	0	0	1	2	32	21	32	43
	T	0	0	30	10	0	0	2	0	11	1	1	10	0	0	11	0	10	0	0	11	12	23	12	23
	A	10	0	10	21	1	0	10	0	0	2	0	0	20	0	0	2	0	20	0	0	2	22	33	13
	C	0	1	0	20	31	11	0	1	0	0	12	0	0	30	10	10	0	0	30	10	0	2	13	43

B

FIGURE 1. Maximum similarity segment analysis of two sequences. Here $s(a,a) = 1$, $s(a,b) = -1$ if $a \neq b$, and $\hat{g} = 2$. In Figure 1A, the initial matrix is shown. Tracing back from the maximum entry 6.2 produces the alignment given in the text. In Figure 1B, the matrix is shown with recomputed entries with * to the right. Now the maximum entry is 6.1.

A computer program written in C for this algorithm is given in the Appendix. The program is also available on tape or disk from the author. To illustrate the algorithm, we set $s(a,a) = 1$, $s(a,b) = -0.9$, if $a \neq b$, and $\hat{g} = 2$. Two sequences are compared and Figure 1A gives the matrix $H(\times 10)$, where the best matching segments are

CCAATCTACT
CTACTCTACT

with score 6.2. The matrix $N(\times 10)$ is shown in Figure 1B where the recomputed entries are shown with * to the right. For this step, the best matching segments are

CTACTACTGCT
CTACTφCTACT

with score 6.1.

E. Near Optimal Alignments

The optimal alignments depend on the input sequences and the algorithm parameters. The weights assigned to mismatches and gaps are determined by experience. An effort is made to use biological data to infer meaningful values. Of course, in addition to assigning weights, there are sometimes unknown constraints on the sequences that cause the correct alignment to differ from the optimal alignment given by an algorithm. Hence, it is of some interest to produce all alignments with score (distance or similarity) within a specified distance of the optimum score. Recently, Waterman[37] and Byers and Waterman,[29] presented a new algorithm which accomplishes this. The algorithm has been previously presented for the distance algorithm. In this chapter, it is given for the similarity algorithm.

To be explicit, let $S = (S_{i,j})$ be the single gap similarity matrix with

$$S_{i,j} = \max\{S_{i-1,j-1} + s(a_i,b_j),\ S_{i-1,j} - \hat{g},\ S_{i,j-1} - \hat{g}\}$$

The task is to find all alignments with score within $e > 0$ of the optimum value $S_{n,m}$. All optimum alignments are included.

At position (i,j) assume a traceback from (n,m) to (0,0) is being performed that can result in an alignment with score greater than or equal to $S_{n,m} - e$. The score from (n,m) to but not including (i,j) is T_{ij}. T_{ij} is the sum of the possibly nonoptimal alignment weights to reach (i,j). From (i,j), as usual, three steps are possible: $(i-1,j)$, $(i-1,j-1)$, and $(i,j-1)$. Each step is in a desired alignment if and only if

$$T_{ij} - \hat{g} + S_{i-1,j} \geq S_{n,m} - e$$

$$T_{ij} + s(a_i,b_j) + S_{i-1,j-1} \geq S_{n,m} - e$$

$$T_{ij} - \hat{g} + S_{i,j-1} \geq S_{n,m} - e$$

respectively. Multiple near-optimal alignments can be produced by stacking unexplored directions. Of course, multiple insertions and deletions can be included.

A study of sequence alignment sensitivity to weights and multiple insertions or deletions has been carried out by Fitch and Smith.[29] The sequences displayed below are chicken hemoglobin mRNA sequences, nucleotides 115-171 from the β chain (upper sequence) and 118-156 from the α chain (lower sequence):

UUUGCGUCCUUUGGGAACCUCUCCAGCCCCACUGCCAUCCUUGUCACACGGCAACCCCAUGGUC
UUUCCCCACUUCG AUCUUUGUCACAC GGCUCCGCUCAAAUC

This alignment is presumed correct from the analysis of the many known amino acid sequences for which such RNA sequences code.

With a distance function, using a mismatch weight of 1 and a multiple insertion or deletion function $x_k = 2.5 + k$, where k is the length of the insertion or deletion, the correct alignment is found among the 14 optimal alignments. (This is region Q of the Fitch and Smith paper.) To indicate the size of neighborhoods in this example, there are 14 alignments within 0% of the optimum, 14 within 1%, 35 within 2%, 157 within 3%, 579 within 4% and 1317 within 5%.

A mismatch weight of 1 and a multiple insertion or deletion function $2.5 + 0.5k$ is region P of Fitch and Smith; accordingly, the correct alignment is not in the list of the two

optimal alignments. This example illustrates the sensitivity of alignment to weighting functions.

IV. DYNAMIC PROGRAMMING ALIGNMENT OF MULTIPLE SEQUENCES

Many sequence alignment problems involve more than two sequences. The straightforward generalization of our two-sequence alignment algorithm to R sequences has complexity to $O(2^R n^R)$ when comparing length n sequences. For R = 3 this is frequently not practical and for R > 3, n must be *very* small. In Section IV.B these algorithms are discussed with some recent improvements. In the next section, IV.A, a geometrical approach, also based on dynamic programming, to these problems is presented.

A. Line Geometries

For this section a new, simple representation of sequences is required. It arises from consideration of a column in an alignment of several sequences. For example, the column

seq1 ... A ...
seq2 ... A ...
seq3 ... T ...
seq4 ... A ...
seq5 ... ϕ ...

is frequently summarized as the "consensus" letter A. Much information is lost in this summary, and our representation of the column is with the "letter" $a = (p_A, p_C, p_G, p_T, p_\phi)$. For example, p_A represents the proportion of A in the given column. In this example p = (3/5,0,0,1/5,1/5) = (0.6,0,0,0.2,0.2). Of course, usual sequences can be represented in this format by using C = (0,1,0,0,0) etc. To compare two "letters" $a = (p_A, p_C, p_G, p_T, p_\phi)$ and $b = (q_A, q_C, q_G, q_T, q_\phi)$, the well-known metric

$$d(a,b) = \left(\sum_i w_i |p_i - q_i|^\alpha\right)^{1/\alpha}$$

is used, where w_i are weighting factors and $\alpha \geq 1$ is a constant. We have found in examples that w_ϕ needs to be larger than the other w_i; certainly ϕ plays a quite different role from A,C,G, or T.

The distance D(a, b) or similarly S(a,b) between two sequences $a = a_1 a_2 \ldots a_n$ and $b = b_1 b_2 \ldots b_n$ is computed by dynamic programming as above even though the sequence elements are themselves vectors. Associated with D(a,b) is an optimal alignment.

$$a_1^* a_2^* \ldots a_l^*$$
$$b_1^* b_2^* \ldots b_l^*$$

Now, for this alignment, define $c(\lambda) = \lambda a \oplus (1-\lambda)b$ by $c_i(\lambda) = \lambda a_i^* + (1-\lambda)b_i^*$. The last "+" is simple vector addition. It is not surprising that c(1/2) is midway between a and b. In fact, much more is true.

Theorem 1 — Let $c(\lambda) = \lambda a \oplus (1-\lambda)b$ where $0 \leq \lambda \leq 1$. Then

$$D(a,b) = D(a,c(\lambda)) + D(c(\lambda),b)$$

$$D(a,c(\lambda)) = (1-\lambda)D(a,b),$$

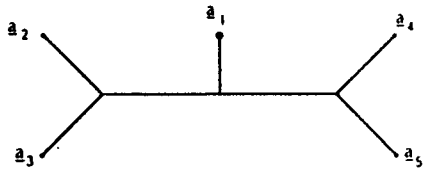


FIGURE 2. Tree relating a_1 (*E. coli*), a_2 (*B. stearo*), a_3 (*H volcanii*), a_4 (*D. discoideum*), and a_5 (*S. cerevisiae*)

and

$$D(c(\lambda_1),c(\lambda_2)) = |\lambda_1 - \lambda_2| D(a,b)$$

This theorem is proved in Waterman and Perlwitz.[30]

This technique was devised to align several sequences, but it does not always behave well on sets of sequences. Methods of finding "center of gravity" sequences were studied and did not always converge rapidly. However, if the sequences are related by a given phylogenetic tree, useful alignments can be produced by use of "$\oplus$".

As an example of this alignment algorithm, consider the tree of Figure 2 with the associated 5 sequences. Alignment is performed by using the tree to suggest the order of sequence alignment:

$$b = \frac{1}{5} a_1 \oplus \frac{4}{5}\left[\frac{1}{2}\left(\frac{1}{2} a_2 \oplus \frac{1}{2} a_3\right) \oplus \frac{1}{2}\left(\frac{1}{2} a_4 \oplus \frac{1}{2} a_5\right)\right]$$

The deletion term needs a heavier penalty than A,C,G,T:

$$d(a,b) = \sum_{i \in \{A,C,G,T\}} |p_i - q_i| + 4|p_\phi - q_\phi|$$

The resulting b is used to obtain an overall alignment by aligning each of $a_1, \ldots, a_5$ with b. The result given below is identical with Woese et al.,[31] who obtain it in Table 31 of their analysis of 16S-like rRNAs.

E. coli (a_1)	CAACCCUUAUCCϕUUUGUAACUCAAAGGAGGAAUGUUGGGA
B. stearo (a_2)	CAACCCUCGCCUϕCUAGUCACUCUAGAGGGGAAGGUGGGGA
H. volcanii (a_3)	AGACCCGCACUUϕCUAAUUACAUUAGAAGGGAAGGAACGGG
D. discoideum (a_4)	AGACCUCGACCUGCUAACCUUCUUAGAGGGGAAGUCCGAGG
S. cerevisiae (a_5)	AGACCUUAACCUACUAAACUUCUUAGAGGGGAAGUUUGAGG

B. Generalization of the Two-Sequence Algorithm

For a straightforward extension of the two-sequence algorithm to more sequences, some notation is first set. There are R sequences, $a = a_1 a_2 \ldots a_m$, $b = b_1 b_2 \ldots b_m$, $\ldots$, $r = r_1 r_2 \ldots r_m$. The algorithm comes from considering an alignment of $a_1 \ldots a_i$; $b_1 \ldots b_j$, $r_1 \ldots r_k$. The last column of the alignment will appear as

$$\epsilon_1 a_i$$
$$\epsilon_2 b_j$$
$$\ldots$$
$$\epsilon_k r_k$$

where $\epsilon_i = 0$ or 1 and $0a = \phi$. If $\epsilon = (\epsilon_1, \ldots, \epsilon_R) \neq 0$, it is required that there are $2^R - 1$ such columns. Recall that for $R = 2$ there are $2^2 - 1 = 3$ terms to optimize. Here, if a "*distance*" function is defined on R variables, the formula extends to

$$D_{i_1 \ldots i_R} = \min_{\epsilon \neq 0} \{D_{i_1 - \epsilon_1, \ldots, i_R - \epsilon_R} + d(\epsilon_1 a_{i_1}, \epsilon_2 b_{i_2}, \ldots, \epsilon_R r_{i_R})\}$$

This formula appeared in Waterman et al.,[13] and has computational complexity proportional to $O(2^R n^R)$ where the sequences are assumed to have length n. Storage takes $O(n^R)$. Carrillo and Lipman[48] have improved the complexity of these algorithms with a branch and bound technique.

A deeper analysis by Sankoff[12] appeared earlier. Sankoff assumes that the sequences are related by a given phylogenetic tree. His algorithm constructs sequences for each interior node of the tree and produces an alignment of the R sequences at the exterior nodes along with the (N) sequences $(X_1, X_2, \ldots X_N)$ constructed for the interior nodes. Sankoff and Cedergren[13] give an excellent account of this procedure. The dynamic programming step is given by

$$D_{i_1 \ldots i_R} = \min_{\epsilon \neq 0} D_{i_1 - \epsilon_1, \ldots, i_R - \epsilon_R} + \min \begin{matrix} \epsilon_1 a_{i_1} \\ \epsilon_2 a_{i_2} \\ \ldots \\ \epsilon_R r_{i_R} \\ X_1 \\ \ldots \\ X_N \end{matrix} \qquad (55)$$

The second minimum is to indicate that $X_1 \ldots X_N$ have been chosen for the interior nodes in such a way as to minimum number of mutations along the tree. A generalization of Fitch's parsimony method[34] is used here and takes N steps. The computation requires $O(2^R n^R N)$ steps. $R = 3$ is nearly the largest practical R and Sankoff has devised an iterative method, which works with groups of three sequences, to build up a solution for larger problems. Altschul and Lipman[49] have extended Carrillo and Lipman[48] to align multiple sequences with a given tree.

V. CONSENSUS ALIGNMENT OF MULTIPLE SEQUENCES

As Section IV illustrates, dynamic programming methods have generally not been found practical for more than two sequences. It is natural to ask whether methods for finding consensus patterns can be applied to find alignments, which are in a real sense consensus patterns themselves. Usually, it will be reasonable to limit the amount of shifting one sequence can have, relative to the others. That is, the alignment

sequence 1 ... a_i

sequence 2 ... b_j

...

sequence r ... r_x

is only allowed when $|i - j|, \ldots |j - x|, |j - x|, \ldots$ are all less than, or equal to, some bound. It is feasible to set the bound equal to sequence length and, therefore, to allow unrestricted shifting. The object of this section is to use the algorithm for consensus words

to make a useful, practical, alignment algorithm (see the chapter on consensus patterns[2]). This algorithm is presented in a recent paper.[15]

To conform with the conventions of the previous chapter, the object is to find consensus words w, k letters in length, where the window width is W. This means the maximum shift between matched words is $W - k$. The bound on $|i - j|, \ldots, |i - x|, |j - x|,$ is thus $W - k$. The window at position i appears as:

	Position	
		$i+1 \ldots i+W$
sequence 1	...	$a_{i-1}a_{i+2} \ldots a_{i+}W$
sequence 2	...	$b_{i-1}b_{i+2} \ldots b_{i+}W$
	...	
sequence R	... ...	$r_{i-1}r_{i+2} \ldots r_{i+}W$

window width = W

A neighborhood of, say, less than, or equal to, 2 mismatches is specified for matching words, and a score $s_w(v)$ is given to a word v in the neighborhood of w, where $0 \leqq s_w(v) \leqq 1$. Define the maximum scoring word in the window of sequence i to be v_i so that v_i best matches w. Then the score of w in the sequence set is

$$S(w) = \sum_{i=1}^{R} \max_{v} \{s_w(v)\} = \sum_{i=1}^{R} s_w(v_i)$$

A consensus word w^* is one satisfying

$$S(w^*) = \max_w S(w)$$

where w ranges over all k-letter words.

An illustration of the location of such a consensus word is given next, where only the window is displayed.

	Position i
sequence 1	.....v_1.....
sequence 2	.....
sequence 3	v_3.....
...	.....
sequence R	.....v_R...
	W

The words v_i are in the neighborhood of w. Here, sequence 2 fails to have a word v_2 in the neighborhood of w.

Define a partial order on consensus words by their location in the sequences as follows: $w^{(1)} < w^{(2)}$ if the occurrence of $w^{(1)}$ in sequence i is to the left of (and not overlapping) the occurrence of $w^{(2)}$ in sequence i, for all $i = 1, 2 \ldots R$. If $v_i^{(1)}$, $v_i^{(2)}$ mark the individual sequence patterns, respectively, in sequence i, then the order appears as

$$\begin{array}{l}
\ldots\ldots v_1^{(1)} \ldots v_1^{(2)} \ldots\ldots \\
\ldots\ldots v_2^{(1)} \ldots\ldots\ldots v_2^{(2)} \ldots\ldots \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots v_3^{(1)} \ldots\ldots \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\ldots\ldots v_M^{(1)} \ldots v_M^{(2)} \ldots\ldots\ldots
\end{array}$$

An optimal alignment A, is one that satisfies

$$S(A) = \max_{A'} S(A') = \max_{A'} \sum_{w_i \mid w_j} S(w_i)$$

In general, it is not known how to find S(A). However, one version of the problem can be solved exactly.

Let $w_1 | w_2$ mean that consensus words w_1 and w_2 can be found in nonoverlapping windows:

$$\begin{array}{l}
\ldots\ldots v_1^{(1)} \ldots\ldots\ldots\ldots\ldots\ldots v_1^{(2)} \ldots\ldots \\
\ldots\ldots v_2^{(1)} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\ldots\ldots v_M^{(1)} \ldots\ldots\ldots\ldots\ldots\ldots v_M^{(2)} \ldots\ldots
\end{array}$$

The modified optimization problem is to find the alignment A, satisfying

$$R(A) = \max_{A'} R(A') = \max_{A'} \sum_{w_i \mid w_j} R(w_i)$$

A simple recursion for R is easily given which is linear in sequence length. Define R_i to be the value of R for the set of sequences ending with column i. Then

$$R_i = \max\{R_{j-1} + S(w_{j,i}) : i - W + 1 \leq j \leq i\}$$

where $w_{j,i}$ is the consensus word with the window from column j to column i.

If much shifting is taking place in the optimal alignment A associated with S(A), then the alignment A* in R(A*) is unsatisfactory and overlapping windows must be considered. To overcome this difficulty in a practical way, find

$$\hat{T}_i = \max\{\hat{T}_{j-1} + \hat{S}(w_{j,i}) : i - W + 1 \leq j \leq i\}$$

where $\hat{S}(w_{j,i})$ is the largest score for a consensus word in the window of width W, such that all occurrences of $w_{j,i}$ are to the right of the matches in $\hat{T}_{j-1}$. While this algorithm is not guaranteed to find S(A), it is much more useful than R(A) for most problems. In addition, it is practical for the alignment of many sequences, while dynamic programming is not.

Another hierarchical algorithm is to find the maximum scoring word w in the entire sequence, $\max_{1 \leq i,j \leq M} S_i(w') = S(w)$, and to realign the sequences on w. This breaks the set into two shorter blocks. The same procedure can be reapplied, recursively, to produce an alignment.

VI. DATA BASE SEARCHES

In summer 1988, there were approximately 20×10^6 bp of data in GenBank, a nucleic acid data base, containing approximately the same data which is in the EMBL data base in Heidelberg. The data represent nearly 5600 nucleotide sequences, at an average of approximately 900 to 1000 bp per sequence (see Chapter 1). When a new DNA sequence is determined, there are usually some sequence comparisions suggested by the nature of the sequence. If the new sequence is a coding region for a rat immunoglobin (Ig), running the

new sequence against all known mammalian Ig sequences is a natural thing to do. However in addition to these sorts of comparisons, there is increasing interest in seeing if any u expected relationships show up when the sequence is run against all mammalian DNA all eukaryotic DNA. The search need not stop even there. Ribosomal RNA structures ha been preserved across all life forms.[31] There seems to be some homology between ribosom proteins from rat and *E. coli*[37]). Therefore, it is not unreasonable to just run the new sequenc against the entire data base and see if any unexpected matches show up.

If the search for a match between a new sequence of 10^3 bp and a data base of 6×1 bp were performed with the dynamic programming algorithm of Section III.D, then t running time is proportional to $6 \times 10^9 = (10^3)(6 \times 10^6)$. On a VAX 11/780, this mig take a day or more of CPU time. With increasing computer capacity, this should not be t disturbing, but it indicates that such a venture is not to be lightly undertaken.

One response to this important problem has been to utilize the computer science techniq of hashing to obtain information about possible regions of high matching. The first use hashing in molecular biology was made by Dumas and Ninio.[38] The method was later utiliz by Wilbur and Lipman[39] and Karlin et al.[40] in developing their own approaches to sequen comparisons.

The basic method begins with a choice of word size, say k letter words. Then a DN sequence $\mathbf{a} = a_1 a_2 \ldots a_n$ is transformed to a sequence of integers by associating each triple, $a_i a_{i+1} \ldots a_{i+k-1}$, with one of the integers $0, 1, \ldots, 4^k - 1$. For example, let

$$\hat{a} = \begin{cases} 0, & \text{if } a = A \\ 1, & \text{if } a = C \\ 2, & \text{if } a = G \\ 3, & \text{if } a = T \end{cases}$$

Then define

$$x_i = \hat{a}_i \cdot 4^{k-1} + \hat{a}_{i+1} \cdot 4^{k-2} + \ldots + \hat{a}_{i+k-2} \cdot 4^1 + \hat{a}_{i+k-1} \cdot 4^0$$

Notice that $x_{i+1} = x_i \cdot 4 + \hat{a}_{i+k} \bmod 4^k$, so that these transformations can be rapidly mad

To continue with a numerical example, let a = TAGAGCA. With $k = 2$, $4^k = 16$ a

$$\begin{aligned}
x_1 &= 3 \cdot 4 + 0 = 12 \\
x_2 &= 12 \cdot 4 + 2 \bmod 16 = 2 \\
x_3 &= 2 \cdot 4 + 0 \bmod 16 = 8 \\
x_4 &= 8 \cdot 4 + 2 \bmod 16 = 2 \\
x_5 &= 2 \cdot 4 + 1 \bmod 16 = 9 \\
x_6 &= 9 \cdot 4 + 0 \bmod 16 = 4
\end{aligned}$$

and $\mathbf{a} = (12)(2)(8)(2)(9)(4)$.

Next, lists are made of locations of all occurrences of the integers $0, 1, \ldots, 4^k - 1$. our numerical example, there are only five nonempty lists

$$\begin{aligned}
0 &: \phi \\
1 &: \phi
\end{aligned}$$

2 : 2,4

3 : ϕ

4 : 6

... : ϕ

8 : 3
9 : 5

... : ϕ

12 : 1

... : ϕ

Another approach is to keep a list of positions of the first occurrences of 0, 1, . . . , $4^k - 1$. In place of the x_i at those locations, are pointers to the next occurrences of the associated integer. In either case, storage is O(n) and the hashing can be performed in time O(n).

Next, two methods, both due to Wilbur and Lipman, which allow use of this information in sequence comparison.

A. A Regions Method

The algorithm described here appears in Wilbur and Lipman. Define a region r by $(\nu;i,j)$ where ν is a word of length k which begins at position i in a and position j in b.

Define $r_1 = (\nu_1;i_1,j_1) < r_2 = (\nu_2;i_2,j_2)$ if $i_1 + k - 1 < i_2$ and $j_1 + k - 1 < j_2$. Set $r_0 = (\phi;0,0)$ as a least element and $r^* = (\phi;n,m)$ as a greatest element. $\Gamma = (r_1 r_2, \ldots r_s)$ is a path if $p < q$ implies $r_p < r_q$. The score of path Γ is given by

$$\text{score}(\Gamma) = \sum_{k=1}^{\ell} s(r_k) - \sum_{k=1}^{\ell-1} g(i_{k+1} - |w_k| - i_k - 1,\ j_{k+1} - |w_k| - j_k - 1)$$

where $s(\cdot)$ is a similarity score for region r_k, like $|w_k|$, and $g(\cdot,\cdot)$ is a gap penalty. Then

$$\text{score}(a,b) = \max\{\text{score}(\Gamma):\ \Gamma \text{ is a path from } r_0 \text{ to } r_*\}.$$

The algorithm makes two lists of regions: L^-, ordered by $<$ and L^+, ordered by the usual order $<<$ of best scores from r_0 to the region listed in L^+.

$$\gamma = \text{score}(r_p) + g(i_q - |w_p| - i_p - 1,\ j_q - |w_p| - j_p - 1) + s(r_q) > \text{score}(r_q)$$

If there is no region r_p in L^+ below r_q under $<$ with a score greater than score $(r_q) - s(r_q)$, or if the inequality cannot be satisfied, go to (C)

(B) Set score $r_q = \gamma$ and go to (A)
(C) Remove r_q from L^- and insert it in L^+ under $<<$.
If $L^- \neq \phi$, go to (A).

Changes can be made to this algorithm to give maximum scoring segments. A more practical algorithm is given next.

B. Rapid Similarity Searches

The method described here is due to Wilbur and Lipman[39] and was extended, especially for protein data base searches, by Lipman and Pearson.[41] Recently, Pearson and Lipman[39] have modified their earlier work to DNA sequences. Each of the 4^k words w, of length k, can be located in each sequence, and the positions of the word w in a can be considered to match the b positions of w. Each such match, say position i in a and position j in b, has an associated offset i−j. The offsets or diagonals with a large number of matches are candidates for good matching regions. This technique is not so rigorous as the dynamic programming methods but it is a great deal faster. In fact, searches of the entire DNA data base can be accomplished on an IBM PC and several groups have programmed this algorithm.

VII. THE STATISTICAL DISTRIBUTION OF ALIGNMENT SCORES

Much has been written about the statistical distribution of distance and similarity scores. In this section, we focus attention on the statistical distribution of the scores of maximum similarity segments (see Section III.D), when the scores are computed for random sequences. The idea is that when the sequences satisfy some model of randomness, such as uniform and independent bases, there is a resulting distribution of maximum similarity segment scores. The scientist can use this distribution to ascertain whether the scores from real, biological sequences are, in the statistical sense, significantly larger than those from random sequences. Of course, there is little agreement about the appropriate model of randomness. Fortunately, as we discuss in Section VII.B, the distribution of maximum similarity segment scores for real, unrelated biological sequences coincides with that of independent, identically distributed sequences of the same composition.

Although simulations are often recommended to determine statistical significance, there are several drawbacks to this approach. First of all, it is expensive in terms of computer time. Also, it often requires more time from the scientist to set up and process the simulations. Frequently, the simulations fail to give the desired results. If statistical significance of a real sequence matching is to be estimated and that significance is α, then $1/\alpha$ simulations must, on the average, be run before seeing a result as extreme as in the real sequences. When $\alpha = 0.0001$, a not unreasonable case, we must do on the order of 10,000 runs. While simulations are often unavoidable, they are not ideal.

Recently, however, some new results in probability theory give very precise answers to certain problems that arise in practice. These are discussed in the subsections on the "log(n) law". What is less well-understood, is that the log(n) law and another equally special result give guidance on what to expect for the distribution of any maximum similarity algorithm. There are only two behaviors of expected similarity score with sequence length: either proportional to sequence length or to logarithm of sequence length. This result, along with a table of means and variances, is given in Section VII.C on expected behavior.

A. The Log(n) Law

Erdos and Renyi[42] proved in 1970 that the length R_n of the longest run of heads in independent coin tosses with P(Heads) = p grows like $\log_{1/p}(n)$:

$$P\left(\lim_{n\to\infty} \frac{R_n}{\log_{1/p}(n)} = 1\right) = 1$$

If our sequences are perfectly aligned

$$a_1\ a_2 \ldots a_n$$
$$b_1\ b_2 \ldots b_n$$

this mathematical result provides the answer to questions about longest match. Simply write H if $a_i = b_i$, T if $a_i \neq b_i$ and calculate

$$p = p_A^2 + p_{Re}^2 + p_O^2 + p_r^2$$

The length of the longest match is the length of the longest head run. Notice that we assume the distributions of the sequences are identical. The results stated here hold if the sequences have base distributions that are "not too different".

When the sequences are allowed to shift relative to one another, the answer changes. Now matches can include offsets such as

$$\begin{array}{l} a_1\ a_2 \ldots a_{n-j+1} \ldots a_n \\ b_1\ b_2 \ldots b_j\ b_{j+1} \ldots b_n \end{array}$$

It has been shown[43] that shifting doubles the growth of the longest match length M_n:

$$P\left(\lim_{n\to\infty} \frac{M_n}{\log_{1/p}(n)} = 2\right) = 1$$

Much more precise results have been obtained. With $a = a_1a_2 \ldots a_n$ and $b = b_1b_2 \ldots b_m$, it is required below that $\log(m)/\log(n) \to 1$. The expectation (mean) of M, the longest match length including k mismatches, is approximately

$$E(M) \cong \log(qmn) + k\log\log(qmn) + k\log(q/p) - \log(k!) + \gamma \log(e) - 1/2$$

where $q = 1-p$, $\log = \log_{1/p}$ and $\gamma = 0.577\ldots$ is the Euler-Mascheroni constant. The variance is approximately

$$\mathrm{Var}[M(n,m)] \cong [\pi \log(e)]^2/6 + 1/12$$

In the case of $k = 0$,

$$E(M) \cong \log(qmn) + \gamma \log(e) - 1/2$$

While it might seem natural to perform the normal approximation at this point, that is not correct! The tail behavior of the distribution is extreme value which has exponential tails. The correct procedure can be found in Theorem 2 of Arratia et al.[4] Results closely related to these were first announced for repeats in a single sequence, with $k = 0$, and minor differences in constants, by Karlin et al.[3] The more general case stated here is from Arratia et al.[4] As discussed in Chapter 6, these formulas can be extended to study the longest match common to R of N sequences. Instead, here we pursue generalizations that allow less exact matching.

B. A Data Base Study

In Smith et al.,[45] the following similarity and gap functions are used:

$$\begin{aligned} s(a,a) &= 1 \\ s(a,b) &= -0.9 \quad \text{if } a \neq b \\ w(1) &= 2 \\ w(k) &= \infty \quad \text{if } k \geq 2 \end{aligned}$$

to study maximum similarity segment scores from real, biological sequences. While the log(n) law appears useful for a fixed number of mismatches, what is the distribution of

$$S(a,b) = \max_{I \subset a,\ J \subset b} S(I,J)$$

for a similarity function such as that specified above? Here, I and J denote contiguo segments of sequence.

In the study cited above, all pairwise comparisons were made with a set of 204 vertebra and eukaryotic DNA sequences and complements for a total of $\binom{2n}{2} = 20{,}706$ comparison The results display remarkable linear behavior with $\log_{1/p}(nm)$. The best fit of the data, af adjusting for outliers by techniques of robust statistics, is

$$\hat{S} = 2.5 \log_{1/p}(nm) - 8.99$$

This fit of real sequence data is identical to the fit of simulated sequences reported belo

Therefore, allowing insertions and deletions does not change the log(nm) growth, althou the slope is certainly changed. Note that $\log(nm) = \log(n^2) = 2\log n$ if $n = m$.

C. Linear and Logarithmic Behavior

Let μ be the non-negative mismatch penalty and δ the non-negative deletion penalty the maximum segment similarity; measure S and write $S = S(\mu,\delta)$ to indicate this dep dence. The goal of this section is to give information about the probability distribution $S(\mu,\delta)$ for all $\mu \geq 0$, $\delta \geq 0$. It was only recently that anything was learned about these questi and much work remains for mathematicians to complete the theory.[46] Still, the broad outli have been established and the results are both theoretically and practically of interest.

Now for two random sequences of length n,

$$S(\infty,\infty) \cong 2\log(n)$$

by the above discussion. In contrast, if no penalty is associated with unmatched bases, known[33] that

$$S(0,0) \cong c \cdot n$$

where c is a constant. While the precise distribution of $S(\infty,\infty)$ is known, even the valu $c = c(0,0)$ has eluded probabilists for more than 10 years. It has recently been shown this contrasting behavior between linear and logarithm growth is general. There is a dimensional curve through $[0,\infty]^2$, such that the behavior is linear on one side of the c and logarithmic on the other.[46] Let R(0) be the region containing (0,0) and $R(\infty)$ the re containing (∞,∞). Then the theorem is that

$$S(\mu,\delta) \approx c(\mu,\delta) \cdot n \quad \text{if} \quad (\mu,\delta) \in R(0)$$

and

$$S(\mu,\delta) \approx d(\mu,\delta) \cdot \log(n) \quad \text{if} \quad (\mu,\delta) \in R(\infty)$$

Table 5 gives means and variances for a large simulation. In this simulation, 28 indepen pairs of sequences were generated of lengths $2^7, 2^8, \ldots, 2^{12}$ where $p_A = p_C = p_G =$ $= 1/4$. At each (μ,δ) of Table 5 and for each of the $28(6) = 168$ pairs of sequences, S was calculated. In the region R(0), (which contains all of column $\mu = 0.50$), a linea was made to the data for each pair (μ,δ). In Table 5, the triple

Table 5
SLOPES, INTERCEPTS, AND STANDARD DEVIATIONS FOR S(μ,δ) IN THE LINEAR, R(0), AND LOGARITHMIC, R(∞), REGIONS

δ=	0.25	0.50	0.75	1.00	1.25	1.50	2.00	2.50	3.00	3.50	4.00	4.50	5.00
0.25	−2.86	−2.82	−2.82	−2.82	−2.82	−2.82	−2.82	−2.82	−2.82	−2.82	−2.82	−2.82	−2.82
	0.51	0.48	0.48	0.48	0.48	0.48	0.48	0.48	0.48	0.48	0.48	0.48	0.48
	5.14	5.73	5.73	5.73	5.73	5.73	5.73	5.73	5.73	5.73	5.73	5.73	5.73
0.50	−2.05	−2.03	−1.73	−1.18	−1.18	−1.18	−1.18	−1.18	−1.18	−1.18	−1.18	−1.18	−1.18
	0.42	0.38	0.34	0.30	0.30	0.30	0.30	0.30	0.30	0.30	0.30	0.30	0.30
	5.64	6.30	6.78	7.25	7.25	7.25	7.25	7.25	7.25	7.25	7.25	7.25	7.25
0.75	−0.93	−0.47	0.03	1.01	2.47	4.53	4.53	4.53	4.53	4.53	4.53	4.53	4.53
	0.35	0.29	0.24	0.20	0.16	0.13	0.13	0.13	0.13	0.13	0.13	0.13	0.13
	5.60	6.24	6.66	7.29	7.82	8.63	8.63	8.63	8.63	8.63	8.63	8.63	8.63
1.00	−0.19	1.54	2.53	4.49	7.79	11.88							
	0.31	0.22	0.16	0.11	0.07	0.03							
	5.63	6.07	6.65	6.99	7.08	7.07							
1.25	0.77	2.92	6.45	9.98			−4.13	−3.08	−3.08	−3.08	−3.08	−3.08	−3.08
	0.28	0.18	0.10	0.04			1.85	1.64	1.64	1.64	1.64	1.64	1.64
	5.79	5.70	6.53	6.63			1.63	1.49	1.49	1.49	1.49	1.49	1.49
1.50	1.60	4.13	8.95		6.48	−3.93	−2.34	−2.01	−2.03	−2.03	−2.03	−2.03	−2.03
	0.25	0.15	0.06		2.21	1.76	1.46	1.38	1.36	1.36	1.36	1.36	1.36
	5.83	5.63	6.13		1.75	1.41	1.33	1.22	1.20	1.20	1.20	1.20	1.20
2.00	2.75	6.70		−5.09	−2.14	−1.40	−1.60	−1.50	−1.42	−1.36	−1.29	−1.29	−1.29
	0.21	0.09		1.95	1.44	1.27	1.23	1.20	1.18	1.17	1.16	1.16	1.16
	5.19	5.99		1.71	1.35	1.27	1.20	1.14	1.12	1.10	1.10	1.10	1.10
2.50	3.98	9.52	−9.10	−2.94	−1.52	−1.19	−1.23	−1.21	−1.22	−1.18	−1.15	−1.14	−1.13
	0.19	0.06	2.69	1.58	1.30	1.20	1.15	1.14	1.13	1.12	1.12	1.11	1.11
	5.25	5.66	2.24	1.44	1.26	1.22	1.14	1.09	1.05	1.04	1.03	1.02	1.01
3.00	5.11	12.02	−6.20	−2.45	−1.46	−1.24	−1.11	−1.08	−1.05	−1.02	−1.00	−0.99	−0.97
	0.17	0.04	2.21	1.47	1.26	1.18	1.12	1.11	1.09	1.09	1.08	1.08	1.08
	5.08	5.32	2.00	1.37	1.23	1.20	1.14	1.08	1.05	1.03	1.02	1.00	1.00
3.50	6.23	14.25	−5.40	−2.00	−1.31	−1.21	−1.06	−1.02	−0.98	−0.94	−0.91	−0.90	−0.89
	0.16	0.02	2.07	1.40	1.23	1.17	1.11	1.10	1.08	1.07	1.07	1.07	1.07
	4.82	5.17	1.93	1.36	1.24	1.20	1.13	1.06	1.03	1.01	1.00	1.00	1.00
4.00	7.25		−5.28	−1.80	−1.24	−1.20	−1.05	−1.01	−0.98	−0.94	−0.91	−0.90	−0.90
	0.15		2.03	1.37	1.22	1.17	1.11	1.09	1.08	1.07	1.07	1.06	1.06
	4.96		1.91	1.37	1.23	1.19	1.12	1.05	1.02	1.01	1.00	1.00	1.00
4.50	8.06		−5.23	−1.74	−1.21	−1.19	−1.04	−0.99	−0.95	−0.92	−0.89	−0.89	−0.89
	0.14		2.01	1.36	1.21	1.16	1.11	1.09	1.07	1.07	1.06	1.06	1.06
	5.20		1.89	1.33	1.22	1.17	1.10	1.04	1.02	1.00	1.00	1.00	1.00
5.00	8.65		−5.26	−1.71	1.19	1.18	−1.02	−0.97	0.92	−0.90	−0.88	−0.88	−0.88
	0.13		2.01	1.35	1.21	1.16	1.10	1.09	1.07	1.07	1.06	1.06	1.06
	5.44		1.87	1.31	1.21	1.16	1.09	1.03	1.02	1.01	1.01	1.00	1.00

a
b
s

in region R(0) means that

$$S(\mu,\delta) = a + b \cdot n$$

with standard deviation s; while for (μ,δ) in region R(∞), the triple means that

$$S(\mu,\delta) = a + b\log_2(n)$$

with standard deviation s.

These means and variances can be used for estimates of statistical significance. Above, it was noted that S(∞,∞) has exponential tails, not the tails of the normal distribution. We conjecture that all of region R(∞) has exponential tails, but that is by no means established mathematically. Moreover, convergence to extreme value distributions is usually very slow, so that except for S(∞,∞), it is unclear what constants to use. To further complicate the situation, in R(0), knowledge beyond linear behavior is completely lacking. Still, it is possible to use the present state of knowledge to obtain useful information.

For illustration, consider *E. coli* threonine tRNA and *E. coli* valine tRNA. Both sequences are 76 in length. To be consistent with the analysis of these sequences in Section III.A, take μ = 1 and δ = 2. A computer sequence comparison yields S(1,2) = 24 − 1(7) − 2(1) = 15. This is the score of the longest boxed portion of the alignment in III.A. The pair (1,2) is in R(∞) and Table 5 gives a = −5.09, b = 1.95, and s = 1.71. Therefore,

$$a + b\log_2(n) = -5.09 + 1.95\log_2(78) = 7.0935$$

and

$$S(1,2) = 15 = 7.1665 + 4.624(1.71)$$

so that S(1,2) is 4.624 standard deviations above the mean.

If additional information is required about the significance, Chebyshev's inequality can be used, as in Smith et al.[45] This conservative result, valid for all probability distributions, is that the probability of $|S(\mu,\delta) - \text{mean}| \geq \lambda$ is less than, or equal to, $(s/\lambda)^2$. In our example, λ = (4.581)s and the statistical significance of our result is therefore no larger than $(1/4.581)^2 = 0.0477$.

APPENDIX

```
/* code for finding segments from two sequences with maximum similarity */
/* Michael S. Waterman and Mark Eggert */
/* see "A NEW ALGORITHM FOR BEST SUBSEQUENCE ALIGNMENTS
        WITH APPLICATION TO TRNA-RRNA COMPARISONS
        Michael S. Waterman and Mark Eggert, Journal of Molecular
        Biology (1987) 197,723-728 */

#include "maxsegs.h"

/* sequence lengths */
int m, n;

/* sequences as arrays of characters */
char *x, /* x[1]...x[m] */
     *y; /* y[1]...y[n] */
```

```
/* score cells as arrays of arrays of short integers */
short
    **F, /* F[0][0]...F[l][n] */
    **S; /* S[0][0]...S[m][n] */

int maxantidiag; /* maximum antidiagonal of traceback end */
int maxS, maxSi, maxSj; /* maximum score and its location */
int maxi; /* maximum i of traceback end */
int force; /* flag to force reporting maximum score */

/* weighting constants for SIMPLE match/mismatch score */
int match, delta; /* s(a,b)=match if a==b
                           delta if a!=b */

/* weighting constants for LINEAR insertion/deletion score */
int alpha, beta; /* w(k)=alpha+beta*(k-1) */

/* cutoff, tracebacks are not done from cells with scores below this value */
int cutoff;

/* structure for cell information */
typedef struct {int i, j} cellinfo;

/* array of cell structures to be ordered */
cellinfo *list;

/* number of cell structures in array */
int listcount;

/*****************************************************************************/
/* maxsegs() is the entry point after all initialization is done             */
/* BEFORE maxsegs() is called:                                               */
/* match, delta, alpha, beta and cutoff must be assigned                     */
/* x and y must be loaded with the sequences to be compared                  */
/* m and n must be assigned the sequence lengths                             */
/* the score arrays must be initialized thus:                                */
/* S[i][0]==0 for all i                                                      */
/* F[0][j]==0 for all j                                                      */
/* S[0][j]==0 for all j                                                      */
/*****************************************************************************/

maxsegs()
{
        make_S();
        trace();
}

#define FAST_REG

/* load score cells with pre-traceback values */

make_S()
{
        register int i, j;
        /* register to hold x[i] */
        register char cx;
        /* registers to make array access easier */
        register short E, *Sp, *Fp=F[0], *bFp=F[1], *sFp;

        maxS=0;
        listcount=0;

        for (j=1; j<=n; ++j)
                Fp[j]=0;
        /* recursive S for LINEAR weighting */
        for (i=1; i<=m; ++i)
        {
                E=0;
                sFp=Fp;
                Fp=bFp;
                bFp=sFp;
                Sp= S[i];
                cx=x[i];
                for (j=1; j<=n; ++j)
                {
                        /* E[i][j] == max { 0, max over k {S[i][j-k]-w(k)} } */
                        E=max(0,
                                Sp[j-1]-alpha,
                                E-beta);

                        /* F[i][j] == max { 0, max over k {S[i-k][j]-w(k)} } */
                        Fp[j]=max(0,
                                S[i-1][j]-alpha,
                                bFp[j]-beta);

                        Sp[j]=max(S[i-1][j-1]+s(cx, y[j]),
                                (int)Fp[j],
                                (int)E);

                        if (Sp[j]>maxS)
                        {
                                maxS=Sp[j];
                                maxSi=i;
                                maxSj=j;
                        }
                        if (Sp[j]>=cutoff)
                        {
                                list[listcount].i=i;
                                list[listcount].j=j;
                                ++listcount;
                        }
                }
        }
}

/* match/mismatch score */

int s(a, b) char a, b;
{
        if (a==b)
                return(match);
        else
                return(delta);
}

/* deletion/insertion score */
int w(k) int k;

{
        return(alpha+beta*(k-1));
}

/* recalculation variables */
int lasti, lastj;
/* If row j is being analysed
   all cells from cell[lasti][j-1] to cell[m][j-1] are unchanged. */
/* If column i is being analysed
   all cells from cell[i-1][lastj] to cell[i-1][n] are unchanged. */

/* Cellorder returns a value of type RELATION. The RELATION type, and its values
   BEFORE, SAME_AS and AFTER are defined in max_segs_defs. The cellorder code
   compares two cell structures, item1 is "cellorder(&item1, &item2)" item2  Our
   program uses this, but its use is hidden in the sorting code */

RELATION cellorder(item1, item2) cellinfo *item1, *item2;
{
        /* order by cell's content */
        if (S[item1->i][item1->j]>S[item2->i][item2->j])
                return(BEFORE);
        if (S[item1->i][item1->j]<S[item2->i][item2->j])
                return(AFTER);

        /* order by distance of cell's antidiagonal from origin */
        if ((item1->i+item1->j)<(item2->i+item2->j))
                return(BEFORE);
        if ((item1->i+item1->j)>(item2->i+item2->j))
                return(AFTER);

        /* order by cell's position on cell's antidiagonal */
        if (item1->i<item2->i)
                return(BEFORE);
        if (item1->i>item2->i)
                return(AFTER);
```

```
        /* same content, same position on same antidiagonal */
        return(SAME_AS);
}

/* make tracebacks */
trace()
{
        int i, j, k;
        int scorei, scorej, scoreS;

        /* enqueue qualifying cells */
        /* if the highest is too low */
        printf("maxS is %d\n", maxS);
        if (maxS<cutoff)
        {
                if (force)
                {
                        list[listcount].i=maxSi;
                        list[listcount].j=maxSj;
                        ++listcount;
                        cutoff=0;
                }
        }
        else
        {
        /* sort array with an algorithm following cellorder's behavior: */
        /* sort(pointer to array, number of objects in array) */
                sort(list, listcount);
        }
        /* start traceback at each queued entry,
           taking entries from "top" of array */
        /* this allows us to easily sort remaining entries
           indexed 0 to listcount-1 */
        while (--listcount>=0)
        {
                scorei=list[listcount].i;
                scorej=list[listcount].j;
                scoreS=S[scorei][scorej];
                /* check if entry is OK */
                if (scoreS>=cutoff)
                {
                        maxantidiag=0;
                        maxi=m;
                        /* finds "shortest" traceback */
                        findnode(scorei, scorej);
                        lasti=lastj=0;
                        /* if this traceback succeeds... */
                        if (plot(scorei, scorej))
                        {
                                /* recalculate scores off end of traceback */
                                redoscores(scorei, scorej);
                                /* print the alignment in */
                                /* a form the user likes  */
                                diagnostic_stuff(scorei, scorej, scoreS);
                                /* resort list */
                                sort(list, listcount);
                        }
                }
                else
                        /* if the top entry is too low, all are too low */
                        break;
        }
}

BOOLEAN plot(i, j) int i, j;
{
        int k, l;

        /* stop at zeros, return TRUE if this is shortest */
        if (i+j<maxantidiag)
                return(FALSE);
        if (S[i][j]==0)
        {
                if (i+j==maxantidiag&&i==maxi)
                        return(TRUE);
                else
                        return(FALSE);
        }

        /* check deletion branch */
        for (k=1; k<i; ++k)
        {
                if (S[i-k][j]==0)
                        break;
                else if (S[i][j]==S[i-k][j]-w(k))
                {
                        if (plot(i-k, j))
                        {
                                for (l=k-1; l>=0; --l)
                                {
                                        /* note that cell i-l,j  */
                                        /* is deleted            */
                                        /* output something like */
                                        /*    x[i-l]             */
                                        /*    -                  */
                                diag(i-l, j, XDEL);
                                redoone(i-l, j);
                                redocolumn(i-l, j);
                        }
                        return(TRUE);
                }
        }
}

/* check match/mismatch branch */
if (S[i][j]==S[i-1][j-1]+s(x[i], y[j]))
{
        if (plot(i-1, j-1))
        {
                /* note that cell i,j has had match or mismatch          */
                /* produce alignment output something like:              */
                /*       x[i]                                            */
                /*       y[j]                                            */

                /* S is built "left to right"                            */
                /* recursion is "right to left"                          */
                /* alignment output occurs on recursion exit             */
                /* so it comes out "left to right"                       */
                diag(i, j, MATCH);
                if (redoone(i, j))
                {
                        redocolumn(i, j);
                        redorow(i, j);
                }
                return(TRUE);
        }
}

/* check other deletion branch */
for (k=1; k<j; ++k)
{
        if (S[i][j-k]==0)
                break;
        else if (S[i][j]==S[i][j-k]-w(k))
        {
                if (plot(i, j-k))
                {
                        for (l=k-1; l>=0; --l)
                        {
                                /* note that cell i,j-l is deleted */
                                /* output something like           */
                                /*    -                            */
                                /*    y[j-l]                       */

                                diag(i, j-l, YDEL);
                                redoone(i, j-l);
                                redorow(i, j-l);
                        }
                        return(TRUE);
                }
        }
}
return(FALSE);
```

```
}

/* find "shortest" traceback */
findplot(i, j) register int i, j;
{
        register int k, l;
        /* stop if already too low */
        if (i+j<maxantidiag)
                return;
        /* stop at zeros */
        if (S[i][j]==0)
        {
                if (i+j>maxantidiag)
                {
                        maxantidiag=i+j;
                        maxi=i;
                }
                else if (i+j==maxantidiag)
                {
                        if (i>maxi)
                                maxi=i;
                }
                return;
        }

        /* check match/mismatch branch */
        if (S[i][j]==S[i-1][j-1]+s(x[i], y[j]))
                findplot(i-1, j-1);

        /* check deletion branch */
        for (k=1; k<i; ++k)
        {
                if (S[i-k][j]==0)
                        break;
                else if (S[i][j]==S[i-k][j]-w(k))
                        findplot(i-k, j);
        }

        /* check other deletion branch */
        for (k=1; k<j; ++k)
        {
                if (S[i][j-k]==0)
                        break;
                else if (S[i][j]==S[i][j-k]-w(k))
                        findplot(i, j-k);
        }
}

/* find cell common to all tracebacks with minimum i+j */
findnode(i, j) int i, j;
{
        int nodei=i, nodej=j, righti=i, rightj=j, lefti=i, leftj=j;
        int k, leftindex=lefti+leftj, rightindex=righti+rightj;

        /* move down left-most and right-most tracebacks
           until they meet for the last time */
        for (;;)
        {
                /* left side */
                do
                {
                        if (S[lefti][leftj]==0)
                        {
                                break;
                        }
                        /* check deletion branch */
                        for (k=1; k<=lefti; ++k)
                        {
                                if (S[lefti-k][leftj]==0)
                                {
                                        if (S[lefti][leftj]==
                                                S[lefti-1][leftj-1]+
                                        s(x[lefti], y[leftj]))
                                        {
                                                --lefti;
                                                --leftj;
                                                leftindex -2;
```



```
                                        }
                                        else
                                        {
                                                for (k=1; k<lefti; ++k)
                                                {
                                                        if (S[lefti][leftj]==
                                                           S[lefti][leftj-k]-w(k))
                                                        {
                                                                leftj-=k;
                                                                leftindex-=k;
                                                                break;
                                                        }
                                                }
                                        }
                                        break;
                                }
                                else if (S[lefti][leftj]==S[lefti-k][leftj]-w(k))
                                {
                                        lefti-=k;
                                        leftindex-=k;
                                        break;
                                }
                        }
                }
                while (leftindex>rightindex);
                if (S[lefti][leftj]==0)
                {
                        if (leftindex>maxantidiag)
                        {
                                maxantidiag=leftindex;
                                maxi=lefti;
                        }
                        else if (leftindex==maxantidiag)
                        {
                                if (lefti>maxi)
                                        maxi=lefti;
                        }
                        break;
                }
                /* right side */
                do
                {
                        if (S[righti][rightj]==0)
                        {
                                break;
                        }
                        for (k=1; k<=rightj; ++k)
                        {
                                if (S[righti][rightj-k]==0)
                                {
                                        if (S[righti][rightj]==
                                                S[righti-1][rightj-1]+
                                                s(x[righti], y[rightj]))
                                        {
                                                --rightj;
                                                --righti;
                                                rightindex-=2;
                                        }
                                        else
                                        {
                                                for (k=1; k<righti; ++k)
                                                {
                                                        if (S[righti][rightj]==
                                                          S[righti-k][rightj]-w(k))
                                                        {
                                                                righti-=k;
                                                                rightindex-=k;
                                                                break;
                                                        }
                                                }
                                        }
                                        break;
                                }
                                else if (S[righti][rightj]==
                                        S[righti][rightj-k]-w(k))
                                {
```

```
                                        rightj--k;
                                        rightindex--k;
                                        break;
                                }
                        }
                }
                while (leftindex<rightindex);
                if (S[righti][rightj]==0)
                {
                        if (rightindex>maxantidiag)
                        {
                                maxantidiag=rightindex;
                                maxi=righti;
                        }
                        else if (rightindex==maxantidiag)
                        {
                                if (righti>maxi)
                                        maxi=righti;
                        }
                        break;
                }
                if (lefti==righti)
                {
                        nodei=lefti;
                        nodej=leftj;
                }
        }
        /* now, find the "shortest" path from this node */
        findplot(nodei, nodej);
}

SE(i, j)
        register int i, j;
{
        register int k, m=0, s0;
        register short *Sp=S[i];

        for (k=1; k<j; ++k)
        {
                s0=Sp[j-k];
                if (s0==0)
                        break;
                s0-=w(k);
                if (s0>m)
                        m=s0;
        }
        return(m);
}

SF(i, j)
        register int i, j;
{
        register int k, m=0, s0;

        for (k=1; k<i; ++k)
        {
                s0=S[i-k][j];
                if (s0==0)
                        break;
                s0-=w(k);
                if (s0>m)
                        m=s0;
        }
        return(m);
}

/* return TRUE if further cells will be affected */
BOOLEAN redoone(i, j) int i, j;
{
        int newS;

        /* recalculate matrix at one place */

        /* omit scores from pathways already taken */
        newS=max(readdiag(i, j, MATCH)?0:(S[i-1][j-1]+s(x[i], y[j])),
                 readdiag(i, j, XDEL)?0:SF(i, j),
                 readdiag(i, j, YDEL)?0:SE(i, j));
```



```
        /* note result */
        if (S[i][j]!=newS)
        {
                S[i][j]=newS;
                return(TRUE);
        }

        /* if further cells will be unaffected, report this */
        if ((i>=lasti)&&(j>=lastj))
        {
                return(FALSE);
        }

        /* report that further cells will be affected */
        return(TRUE);

}

/* recalculate column of matrix */
redocolumn(i, j) int i, j;
{
        int newS;
        int firstj=0; /* j of cell with lowest j of contiguous unchanged cells */

        while (++j<=n)
        {
                /* recalculate S sans previous paths */
                /* readdiag--true if cell[i][j] was MATCHed or XDELed or YDELed */
                newS=max(readdiag(i, j, MATCH)?0:(S[i-1][j-1]+s(x[i], y[j])),
                         readdiag(i, j, XDEL)?0:SF(i, j),
                         readdiag(i, j, YDEL)?0:SE(i, j));

                if (S[i][j]==newS)
                {
                        /* remember lowest cell
                           where all further cells don't change */
                        if (firstj==0)
                                firstj=j;
                        /* when no further cells will change, we're done */
                        if (j>=lastj)
                        {
                                /* remember position for next scan */
                                lastj=firstj;
                                break;
                        }
                }
                else
                {
                        /* note that cell has changed */
                        /* clear j of lowest of contiguous unchanged cells */
                        firstj=0;
                        /* actually change S */
                        S[i][j]=newS;
                }
        }
}

/* recalculate row */
redorow(i, j) int i, j;
{
        int newS;
        int firsti=0;

        while (++i<=m)
        {
                newS=max(readdiag(i, j, MATCH)?0:(S[i-1][j-1]+s(x[i], y[j])),
                         readdiag(i, j, XDEL)?0:SF(i, j),
                         readdiag(i, j, YDEL)?0:SE(i, j));

                if (S[i][j]==newS)
                {
                        if (firsti==0)
                                firsti=i;
                        if (i>=lasti)
                        {
                                /* the first shall be last */
                                lasti=firsti;
                                break;
```

```
                        }
                }
                else
                {
                        first[i]=0;
                        S[i][j]=newS;
                }
        }
}

/* redo cells off end of traceback */
redoscores(i, j) int i, j;
{
        for (; (++i<=m)&&(++j<=n);)
        {
                /* if cells need to be checked, check */
                if (redoone(i, j))
                {
                        redocolumn(i, j);
                        redorow(i, j);
                }
                else
                {
                        return;
                }
        }
}
```

ACKNOWLEDGMENTS

This work was supported by grants from the System Development Foundation and the National Institutes of Health.

REFERENCES

1. **Kruskal, J. B. and Sankoff, D.**, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Sankoff, D. and Kruskal, J. B., Eds., Addison-Wesley, London, 1983.
2. **Waterman, M. S.**, Consensus patterns in sequences, *Mathematical Methods for DNA Sequences*, Waterman, M. S., Ed., CRC Press, Boca Raton, Fla., 1988.
3. **Karlin, S., Ghandour, G., Ost, F., Tavare, S., and Korn, L. J.**, New approaches for computer analysis of nucleic acid sequences, *Proc. Natl. Acad. Sci. U.S.A.*, 80, 5660, 1983.
4. **Arratia, R., Gordon, L., and Waterman, M. S.**, An extreme value theory for sequence matching, *Ann. Stat.*, 14, 971, 1985.
5. **Waterman, M. S.**, General methods of sequence comparison, *Bull. Math. Biol.*, 46, 473, 1984.
6. **Stanton, R. G. and Cowan, D. D.**, Note on a "square functional" equation, *SIAM Rev.*, 12, 277, 1970.
7. **Laquer, H. T.**, Asymptotic limits for a two-dimensional recursion, *Stud. Appl. Math.*, 64, 271, 1981.
8. **Feller, W.**, *An Introduction to Probability Theory and Its Applications*, Vol. 1, 3rd ed., John Wiley & Sons, New York, 1968, 52.
9. **Griggs, J. R., Hanlon, P. J., and Waterman, M. W.**, Sequence alignments with matched sections, *SIAM J. Algorithm Disc. Meth.*, 7, 604, 1986.
10. **Griggs, J. R., Hanlon, P., Odlyzko, A., and Waterman, M. S.**, On the number of alignments of k sequences, *Asian J. Comb.*, in press.
11. **Needleman, S. B. and Wunsch, C. D.**, A general method applicable to the search for similarities in the amino acid sequences of two proteins, *J. Mol. Biol.*, 48, 444, 1970.
12. **Sellers, P.**, Theory and computation of evolutionary distances, *SIAM J. Appl. Math.*, 26, 787, 1974b.
13. **Waterman, M. S., Smith, T. F., and Beyer, W. A.**, Some biological sequence metrics, *Adv. Math.*, 20, 367, 1976.
14. **Gotoh, O.**, An improved algorithm for matching biological sequences, *J. Mol. Biol.*, 162, 705, 1982.
15. **Waterman, M. S.**, Efficient sequence alignment algorithms, *J. Theor. Biol.*, 108, 333, 1984.
16. **Smith, T. F. and Waterman, M. S.**, Comparison of biosequences, *Adv. Appl. Math*, 2, 482, 1981.

17. **Smith, T. F., Waterman, M. S., and Fitch, W. M.**, Comparative biosequence metrics, *J. Mol. Evol.*, 18, 38, 1981.
18. **Sellers, P.**, Pattern recognition in genetic sequences, *Proc. Natl. Acad. Sci. U.S.A.*, 76, 3041, 1979.
19. **Sellers, P.**, The theory and computation of evolutionary distances: pattern recognition, *J. Algorithms*, 1, 359, 1980.
20. **Hawley, D. K. and McClure, W. R.**, Compilation and analysis of *Escherichia coli* promoter DNA sequences, *Nucl. Acids Res.*, 11, 2237, 1983.
21. **Weiss, R.**, Oncogenes and growth factors, *Nature (London)*, 304, 12, 1983.
22. **Doolittle, R. F., Hunkapiller, K., Hood, L. E., Devare, S. G., Robbins, K. C., Aaronson, S. A., and Antoniades, H. M.**, Simian sarcomaviruses onc gene v-sis is derived from the gene (or genes) encoding a platelet-derived growth factor, *Science*, 221, 275, 1982.
23. **Naharro, G. K., Robbins, C., and Reddy, E. P.**, Gene product of v-fgr onc: hybrid protein containing a portion of actin and tyrosin-specific protein kinase, *Science*, 223, 63, 1984
24. **Sellers, P.**, Pattern recognition in genetic sequences by mismatch density, *Bull. Math. Biol.*, 46, 501, 1984.
25. **Smith, T. F. and Waterman, M. S.**, Identification of common molecular subsequences, *J. Mol. Biol.*, 147, 195, 1981a.
26. **Waterman, M. S. and Eggert, M.**, A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons, *J. Mol. Biol.*, 197, 723, 1987.
27. **Waterman, M. S.**, Sequence alignment in the neighborhood of the optimum with general applications to dynamic programming, *Proc. Natl. Acad. Sci. U.S.A.*, 80, 3123, 1983.
28. **Byers, T. H. and Waterman, M. S.**, Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming, *Oper. Res.*, 32, 1381, 1984.
29. **Fitch, W. M. and Smith, T. F.**, Optimal sequence alignments, *Proc. Natl. Acad. Sci. U.S.A.*, 80, 1382, 1983.
30. **Waterman, M. S. and Perlwitz, M.**, Line geometries for sequence comparisions, *Bull. Math. Biol.*, 46, 567, 1984.
31. **Woese, C. R., Gutell, R., Gupta, R., and Noller, H. F.**, Detailed analysis of the higher-order structure of 16S-like ribosomal ribonucleic acids, *Microbiol. Rev.*, 47, 621, 1983.
32. **Sankoff, D.**, Minimal mutation trees of sequences, *SIAM J. Appl. Math.*, 78, 35, 1975.
33. **Sankoff, D. and Cedergreen, R. J.**, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Sankoff, D. and Kruskal, J., Eds., Addison-Wesley, London, 1983, 253.
34. **Fitch, W. M.**, Towards defining the course of evolution: minimum change for a specific tree topology, *Syst. Zool.*, 20, 406, 1971.
35. **Waterman, M. S.**, Multiple sequence alignment by consensus, *Nucl. Acids Res.*, 14, 9095, 1986.
36. **Fickett, W. J. and Burks, C.**, Development of a database for nucleic acid sequences, in *Mathematical Methods for DNA Sequences*, Waterman, M. S., Ed., CRC Press, Boca Raton, Fla., 1988.
37. **Lin, A., McNally, J., and Wool, I. G.**, The primary structure of rat liver ribosomal protein L37, *J. Biol. Chem.*, 258, 10664, 1983.
38. **Dumas, J. P. and Ninio, J.**, Efficient algorithms for folding and comparing nucleic acid sequences, *Nucl. Acids Res.*, 80, 197, 1982.
39. **Wilbur, W. J. and Lipman, D. J.**, Rapid similarity searches of nucleic acid and protein data banks, *Proc. Natl. Acad. Sci. U.S.A.*, 80, 726, 1983.
40. **Karlin, S., Ghandour, G., and Foulser, D. E.**, Comparative analysis of human and bovine papallom viruses, *Mol. Biol. Evol.*, 1, 357, 1984.
41. **Lipman, D. J. and Pearson, W. R.**, Rapid and sensitive protein similarity searches, *Science*, 227, 1435 1985.
42. **Erdos, P. and Renyi, A.**, On a new law of large numbers, *J. Anal. Math.*, 22, 103, 1970; reprinted i *Selected Papers of Alfred Renyi, 1962-1970*, Vol. 3, Akademiai Kiado, Budapest, 1976, 43.
43. **Arratia, R. and Waterman, M. S.**, An Erdos-Renyi law with shifts, in *Advances in Mathematics*, Vo 55, Academic Press, New York, 1985, 13.
44. **Karlin, S., Ost, F., and Blaisdell, B. E.**, Patterns in DNA and amino acid sequences and their statistic significance, in *Mathematical Methods for DNA Sequences*, Waterman, M. S., Ed., CRC Press, Bo Raton, Fla., 1988.
45. **Smith, T. F., Waterman, M. S., and Burks, C.**, The statistical distribution of nucleic acid similaritie *Nucl. Acids Res.*, 13(2), 645, 1985.
46. **Waterman, M. S., Gordon, L., and Arratia, R.**, Phase transitions in sequence matches and nucleic a structure, *Proc. Natl. Acad. Sci. U.S.A.*, 84, 1239, 1987.
47. **Miller, W. and Myers, E. W.**, Sequence comparison with concave weighting functions, *Bull. Math. Bio* in press.